Win32/Upatre.BI - Part One - Unpacking

Archived: 2026-04-06 15:27:24 UTC

This is the first part of the four-part series on "Win32/Upatre.BI". Check out the other parts here:

- Part 2: *Config* (June 14, 2015)
- Part 3: *Main Loop* (June 16, 2015)
- Part 4: *Payload Format* (June 20, 2015)

Win32/Upatre.BI is a recent member of the Upatre downloader family. The malware is usually spread by email attachments. It can steal user information and download a variety of other malicious software such as Zeus, Rovnix, Dyzap or Cutwail. In this multi-part blog post I analyze the inner workings of Upatre.BI by reversing the following sample:

Filename
: OGdbzU.exe

Filesize
: 48128 bytes

MD5
: 99807df2c2b2cdbff9e373611b07cf48

SHA-256
: 08acd0c50dbb6d712c54d25a5ccd99c1d6faf9ad2031e56884a0af991e3fda78

Malwr
: https://malwr.com/analysis/YThlYTU5MmE4OTAyNDEzNTllYjdkZThmODQ5ZTRkYjA/

The sample is well detected by all popular antivirus scanners:

Kaspersky
: Trojan-Downloader.Win32.Upatre.aacs

McAfee
: RDN/Upatre-FABV!a

Microsoft
: Win32/Upatre.BI

Sophos
: Troj/Upatre-LD

Symantec
: Downloader.Upatre!gen9

TrendMicro
: TROJ_UPATRE.SMC1

This first part of the series focuses on the initial step in reversing the malware: **unpacking**. The Upatre sample is not technically packed — in fact the unpacked payload is about a quarter the size — but it is protected by two

decryption stages. The first stage uses a simple XOR and ROL encryption, while the second stage uses a variant of RC4. The following illustration gives an overview of the two unpacking stages.

1. The first unpacking stub modifies part of the memory with ROL and XOR.
2. At the end of the first stage, the tail jump enters the unpacked memory.
3. The second stage copies the unpacking stub code region to a newly allocated page.
4. A call is made to the copied region.
5. Most of the loaded executable, including the PE header, is zeroed out.
6. The encrypted payload is loaded from the executable file and decrypted.
7. The decrypted plain text contains the new executable. It is copied to the image base.
8. The unpacking stub handles relocation and resolving the imports.
9. The tail jump into the decrypted executable concludes the second unpacking stage.

The remainder of the blog posts provides details about the individual unpacking steps.

Stage 1

The first unpacking stub is very simple. It first checks the size of a Windows DLL before decrypting the second stage and jumping to it.

CatSrv.dll

Upatre.BI checks the `catsrv.dll` in the Windows system directory. The system directory is determined with the following call:

```
 0040103F call    ds:GetSystemDirectoryA
00401045 call    append_catsrv_dll
```

The routine `append_catsrv_dll` then appends the name `\catsrv.dll`. The DLL name is hard-coded with some slight obfuscation:

```
 004011E1 mov     edx, 'lld'
004011E6 push    edx
004011E7 mov     edx, '.vru'
004011EC sub     edx, 2      ; -> .vrs
004011EF push    edx
004011F0 mov     edx, 'tac[' ; -> tac\
004011F5 inc     edx
004011F6 push    edx
```

The same obfuscation method — shifting letters — is also used in the unpacked payload. This might indicate that the paylaod and the protector were written by the same authors.

Image Size of CatSrv.dll

The sample reads the contents of `c:\windows\system32\catsrv.dll` with `fopen`. The address of the API function is manually determined by reading the import address table, whose location is hard-coded:

```
 00401006 mov     ebx, 65758h
...
00401021 add     ebx, 39E8A8h    ; -> 0x404000 (.idata)
```

The offsets 0x404000 points to the import directory if the .idata section is placed there:

If the import directory is loaded somewhere else, then a memory reference error will occur and the process crashes. The sample retrieves the import at offset 0x7C:

```
 0040105E                sub     eax, eax
00401060                add     eax, 7Ch
00401065                add     eax, ebx ; -> eax = fopen
...
0040107C                call    dword ptr [eax]
```

which is `fopen`:

```
.idata:0040407C                extrn fopen:dword
```

It then uses this MS VC++ routine to read the `SizeOfCode` value of the *catsrv.dll* by reading field 0x1C into the PE header:

```
 0040109E                 call    ds:fread
004010A4                mov     eax, esi
004010A6                add     eax, 3Dh
004010AB                dec     eax ; -> offset 3C = real PE header
004010AC                mov     eax, [eax]
004010AE                add     eax, esi
004010B0                push    1Eh
004010B2                pop     edi
004010B3                dec     edi
004010B4                dec     edi ; -> edi = 1C
004010B5                mov     eax, [eax+edi] ; -> size of code
```

The SizeOfCode needs to be 768 or larger than 131581, otherwise the unpacking step is skipped and the malware will crash.

Unpacking

Upatre arrives at this unpacking loop if the DLL size is in order:

The two XOR instructions at the initialization calculates 0x3E00, which is the size of the packed region. The start of the packed region is pointed to by esi, which was set to 0x405200 at the entry point:

```
 00401000 push    83h
...
0040101B add     esi, 40517Dh
```

The unpacking routine decompiles to the following simple routine:

```
 for i from 0x405200 to 0x408FFF
    Byte(i) = ROL(Byte(i), 1) ^ 0x12
endfor
```

Where ROL denotes the rotate on left instruction.

Tail Jump

The first unpacking stage concludes by jumping to 0x405604 inside the decrypted area:

```
 0040119C                mov     edi, 40520Ah
004011A1                xor     eax, eax
004011A3                add     esp, 3Ch
004011A6                jmp     return
...
00401375 return:
00401375
00401375                add     edi, 3FAh
0040137B                push    edi
0040137C                retn            -> 0x405604
```

Stage 2

The second unpacking stage is slightly more complicated than the first. Instead of using simple XOR/ROL obfuscation, it uses an RC4 variant as protection. It also features manual resolving of API addresses.

Find Kernel32

The first step of the second stage is to laod the base address of the kernel32 library:

```
 0040560C                call    get_kernel32
```

The routine performs the following steps:

```
 mov     ebx, large fs:30h     -> TIB -> PEB
mov     ebx, [ebx+0Ch]        -> PEB -> PEB_LDR_DATA
mov     ebx, [ebx+0Ch]        -> PEB_LDR_DATA -> InLoadOrder
mov     ebx, [ebx]            -> second element (ntdll)
```

```
mov     ebx, [ebx]            -> third element (kernel32)
mov     eax, [ebx+18h]        -> LDR_MODULE -> DllBase
```

The first three lines get the *InLoadOrder ListEntry*, located at offset 12 into the *PEB_LDR_DATA*, which is located at offset 12 into the PEB at offset 0x30 into the TIB. The code then follows the forward link of the doubly linked list *InLoadOrder* twice to get to the third element. The third module that is loaded after the process and ntdll is kernel32. The code finally retrieves the DllBase at offset 0x18 into the LDR_MODULE for kernel32.

Modify the Entry Point

While the malware relies on the kernel32 being the third loaded module, it uses a more robust method to locate the current process. The malware first determines it's image base with the following instructions:

```
 00405614                call    $+5
00405619                pop     eax
0040561A                xor     ax, ax
```

These lines round the current EIP down to the image base. As in `get_kernel32`, the code looks up the *InLoadOrder* list. It traverses this list until it finds the module that matches the image base, i.e., the current process:

```
 0040562C                push    large dword ptr fs:30h
00405633                pop     eax
00405634 loc_405634:
00405634                mov     eax, [eax+0Ch]
00405637                mov     eax, [eax+0Ch]
0040563A loc_40563A:
0040563A                cmp     [eax+18h], ebx
0040563D                jz      short loc_405643
0040563F                mov     eax, [eax]
00405641                jmp     short loc_40563A
```

The malware then sets the image base of the current process to zero (will be fixed later), and the entry point at offset 0x1C to the original entrypoint 0x127A. Both values will later be overwritten.

```
 00405627                mov     esi, 127Ah
00405643                mov     [eax+18h], edi
00405646                mov     [eax+1Ch], esi
```

Resolve API addresses

Upatre.BI imports all system APIs manually by calling a routine `getProcByHash`. It has the following prototype:

```
getProcByHash(dll, hash)
```

The first parameter `dll` is the address of the kernel32 DLL resolved earlier. The second argument `hash` is 32 bit value tied to a specific procedure in kernel32. The disassembly of the routine looks as follows.

This constitutes a very common way of finding imports by malware, as illustrated by the following image:

1. The offset to the PE header is read from the DOS header.
2. The offset to the export directory is read from the PE header.
3. The offset to the export names table is read from the export table.
4. Every name in the names table is hashed and the result compared to the desired hash.
5. When a name produces the correct hash (in the example the routine *GetProcAddress*, then the corresponding row is looked up in the export ordinals table, e.g., 0x244.
6. In the export address table the row that corresponds to the ordinal is read. This is the RVA to the desired procedure.
7. To get the absolute address, the base address of the DLL is added (in our example 0x75200000).

The hashing routine is simple, yet not trivial to reverse:

```
 for letter in string
    hash = 0
    hash = ROL(hash, 7)
    hash = hash ^ letter
endfor
```

It is used to conceal the names of the following kernel32 functions:

- GetProcAddress

- VirtualProtect
- LoadLibraryExA
- GetModuleHandleA
- VirtualAlloc
- GetModuleFileNameA
- CreateFileA
- SetFilePointer
- ReadFile
- VirtualFree
- CloseHandle
- GetModuleHandleA

Copy Unpacking Stub

The second unpacking stage overwrites large parts of the loaded code. It therefore first saves the remaining code by copying it to newly allocated pages. The size is set to the image size of the loaded exe (which is more than enough):

```
 00405709                mov     [ebp+image_base], eax
0040570C                mov     eax, [eax+pe.e_lfanew]
0040570F                add     eax, [ebp+image_base]
00405712                mov     [ebp+pe_header_], eax
00405715                mov     eax, [eax+pe.SizeOfImage]
00405718                mov     [ebp+size_of_image], eax
0040571B                push    40h
0040571D                push    1000h
00405722                push    [ebp+size_of_image]
00405725                push    0
00405727                call    [ebp+kernel32_VirtualAlloc]
0040572A                mov     [ebp+unpacking_stub], eax
```

Then 0x694 bytes starting from 0x4057F (0x40573A + 0x15) are copied to the newly allocated space:

```
 00405732                mov     [ebp+hThisProcess], eax
00405735                call    $+5
0040573A                pop     eax
0040573B                add     eax, 15h
0040573E                push    694h
00405743                push    [ebp+unpacking_stub]
00405746                push    eax ; 0040574f
00405747                call    copy_
```

Finally, the copied code is run:

```
0040574C                call    [ebp+unpacking_stub]
```

The copied code follows right after the previous call. In the following I report the offsets at their original location.

Unpacking

Before the payload is unpacked, Upatre cleans any trace of the current image by zeroing out the memory from the image base.

```
00405828 mov     edi, [ebp+image_base]
0040582B mov     ecx, [ebp+SizeOfImage]
0040582E shr     ecx, 2
00405831 xor     eax, eax
00405833 rep stosd
```

The original payload lies encrypted at a fixed location into the executable file. The encryption method is VMPC RC4 with a 64 bit IV and 64 bit key, generated by a modified LCG. Starting at 0x5576 bytes into the exe, Upatre first reads 64 bytes as the initialization vector. Following that, it reads the 0x1A44 bytes long ciphertext:

```
GetModuleFileNameA(-1, &path_to_exe, ...)
h = CreateFileA(&path_to_exe, ...)
SetFilePointer(h, 0x5576, 0, 0)
ReadFile(h, &iv, 0x40, ...)
ReadFile(h, &ciphertext, 0x1A44, ...)
```

Upatre then creates the decryption key:

```
00405838                push    [ebp+key]
0040583B                call    generate_key
```

The key generation routine decompiles to the following pseudo-code:

```
procedure generate_key(key[64], seed)

for i from 0 to 63
    ix = 16807 * ((seed ^ 123456789) % 12773) - 2836 * ((seed ^ 123456789) / 12773)
    if ix < 0 then
        ix += 0x7FFFFFFF
    end if
    key[i] = ix
    seed = ix ^ 123456789
end for
```

This resembles the LCG-rand implementation in C, except:

1. the additional encryption of the seed with XOR 123456789
2. 12773 is used instead of 127773.

The second modification makes no sense, because the number can't be freely chosen but rather depends on the modulus and factor of the LCG. Chosing a different values breaks Schrage's trick.

The next step of RC4 is the initialization of the permutation S:

```
 procecedure vmpc_initialization(S[256])

for i from 0 to 255
    S[i] = i
end for
```

The permutation is then scrambled with the following algorithm:

```
 procedure vmpc_scrambling(S[256], vec, veclen, j)

for m from 0 to 3*256 - 1
    i = m % 256
    j = S[j + vec[m % veclen] + S[i]]
    swap( S[i], S[j] )
end for
return j
```

The scrambling routine is first called with the key as the second parameter, then again with the IV. After these KSA steps, the decryption of the ciphertext is carried out using the PRGA variant of VMPC:

```
 procedure vmpc_prga(S[256], ciphertext, ciphertext_len, j)

for a from 0 to ciphertext_len -1
    i = m % 256
    j = S[(S[i] + j) % 256]
    ciphertext[m] = c[m] ^ S[(S[S[j]] + 1) % 256]
    swap( S[i], S[j] )
return j
```

As seen earlier, the key for the decryption is calculated by a modified LCG based on a seed. This seed stored nowhere in the binary, but rather brute-forced by the unpacking stub: at first, the decryption is carried out for the first four bytes of the ciphertext for every seed starting with 0 until the plaintext reads "AMPC" (for my sample with the seed 0x45). The key is then used to decrypt the remainder of the ciphertext. The following pseudo-code summarizes the decryption of the payload.

```
 seed = 0
IV = exe_file[0x5576 : 0x557A]
c = exe_file[0x557A : 0x6FBE]
while True
    generate_key(key, seed)
    vmpc_initialization(S)
    j = vmpc_scrambling(S, key, 64, 0)
    j = vmpc_scrambling(S, iv, 64, j)
    magic = c[0:4]
    j = vmpc_prga(S, magic, 4, j)
    if magic = "AMPC"
        break
    else
        seed = seed + 1
    end if
end while

vmpc_prga(S, c[4 : 0x6FBE], 0x1A44, j)
```

Copy

The decrypted payload is written to the current image base. First the PE headers protection is changed to be writable:

```
 004058CD                mov     eax, [ebp+plaintext]
004058D0                mov     eax, [eax+pe.e_lfanew]
004058D3                add     eax, [ebp+plaintext]
004058D6                mov     [ebp+pe_header], eax
004058D9                mov     eax, [eax+pe.nr_of_sections]
004058DC                mov     word ptr [ebp+nr_of_sections], ax
004058E0                mov     ebx, [ebp+pe_header]
004058E3                mov     ebx, [ebx+pe.SizeOfHeaders]
004058E6                mov     eax, ebp
004058E8                sub     eax, 48h
004058EB                push    eax ; prev header protectiong
004058EC                push    PAGE_EXECUTE_READWRITE
004058EE                push    ebx
004058EF                push    [ebp+image_base]
004058F2                call    [ebp+kernel32_VirtualProtect]
```

then the header is overwritten by the header of the payload:

```
 004058F5                push    ebx ; size of headers
004058F6                push    [ebp+image_base] ; dst
```

```
004058F9                push    [ebp+plaintext] ; src
004058FC                call    copy
```

Upatre then iterates over all sections of the payload — .text and .idata — and also copies them to the current image.

Relocation and Imports

After the header and the sections are copied, the unpacker performs relocation if necessary:

```
 00405959                mov     ebx, [ebp+image_base]
0040595C                call    relocate
```

The payload is position independent and therefore has no relocation information, so the relocation call does nothing even if the image base should not be 0x400'000. In this case, however, the first stage of unpacking would have crashed already.

Finally, the imports are manually resolved the same way the loader would resolve imports:

1. The offset to the PE header is read from the DOS header.
2. The offset to the import directory is read from the PE header.
3. For each element in the import directory, the name at offset 0x0C is retrieved and the corresponding DLL is loaded with `LoadLibraryExA`.
4. The array of hint names is visited by following `OriginalFirstThunk`.
5. For each thunk in the HintNames array, the pointer is followed to the `IMAGE_IMPORT_BY_NAME` structure.
6. At offset 2, the name is read and the corresponding procedure is loaded by calling `GetProcAddress()` (using the DLL address from step 3).
7. The resolved address is stored in the IAT, which is pointed to by the `FirstThunk` value at offset 0x10 into the import directory.

The second stage finally jumps to the unpacked payload. The unpacking stubs tries to veil the jump by setting the original entry point as the return address for an API call *VirtualFree*:

```
 00405A57                mov     eax, [esp+14h+image_base_]
00405A5E                add     eax, 127Ah
00405A63                mov     [esp], eax
00405A67                pop     eax
00405A68                jmp     [esp+10h+VirtualFree]
```

The unpacked exe can then be dumped. No import reconstruction or other fixes are necessary. Interestingly, the unpacked executable has a lower detection rate than the packed version. Sophos, Symantec and TrendMicro for example classify the payload as clean (as of June 9th):

Kaspersky
: Trojan-Downloader.Win32.Upatre.aasc

McAfee
: Upatre-FABV!7347D2130AB5

Microsoft
: Win32/Upatre.BI

Sophos
: clean

Symantec
: clean

TrendMicro
: clean

Part 2 of this blog post series will document the data structures used by Upatre and their content.

Source: https://johannesbader.ch/2015/06/Win32-Upatre-BI-Part-1-Unpacking/