Analysis Report

Trojan-Downloader.Win32.Upatre

Marion Marschalek | @pinkflawd
February 2014

Sample Hard Facts

MD5	6a9d66df6ae25a86fcf1bbfb36002d44
SHA1	1454ce7857f57b38f807c0840b872f3973abd5cb
SSDEEP	192:OUf4wiaTmBzAO+e+fvlRcJ+9kBao5U4cIPe5E9cks5IisvK3UboQTMtfDpY+M/6G:Z4 N2mBb+fvlDoqXsqUzx+M/6HPWuw
Size	20.00 KB (20480 bytes)
Type	PE32
Detections	TrojanDownloader.Win32.Upatre Win32.TrojanDownloader.Waski Trojan.Win32.Agent

Internet Communication

Domain	**IP-Address**
mentoringgroup.com	216.171.192.113
davistructures.com	173.255.128.30

Dropped Files

Filename	**MD5**	**Size**
budha.exe	2F1919F65BD6CDC36949114F9ED1A7A1	20.03 KB (20510 bytes)
html[1].exe	46DF9C332238A88800D0868FA4FCF415	376.00 KB (385024 bytes)
kilf.exe	085AB42FA4A9B41705B4FB5B554A2478	378.00 KB (387072 bytes)
vogiap.exe	C15280E326B8C9835EBE90907D8603CA	378.00 KB (387072 bytes)
KUQ9491.bat	FD85A16C326FF57CCDE9A2025009B931	174 bytes
ehri.ofu	D3690D9FD6962876C4C606631A2A37B2	482 bytes

1. Functionality

The malware at hand is a malicious downloader with the sole purpose to connect to a remote command and control server (C&C) when invoked and to download and execute additional malware. It communicates via HTTPS to one of two hardcoded domains, which are believed to be legitimate websites on compromised web servers.

Malware execution can be parted in a protection layer, an unpacking layer with different stages and the final payload. For an initial infection the malware just copies its own image to the systems %TEMP% directory and executes that copy.

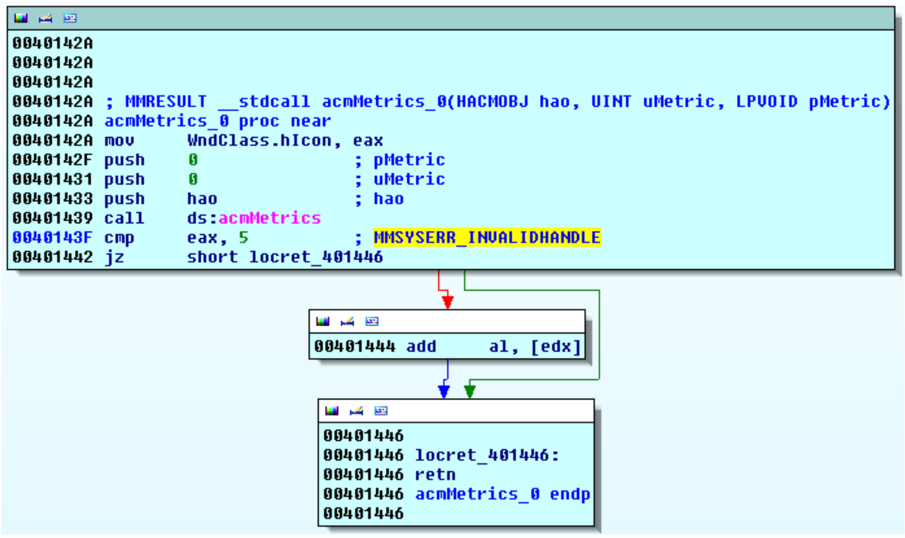
For infection the sample iterates its two hardcoded domains and checks via HTTPS if there is a specific file to download, namely under /images/html.exe. If so, this file will be downloaded and saved in the current directory under the hardcoded filename ./kilf.exe. Sure enough this binary is executed via ShellExecuteW and the downloader terminates.

2. Anti-Analysis and Packer Stages

2.1 Protection Layer

2.1.1 Anti-Simulation: acmMetrics

acmMetrics is an API call present in the msacm32.dll library. Usually it is used for retrieving metrics for ACM objects (Audio Compression Manager). During the startup procedure of a malware sample it is highly likely that this was not the initial intention when placing that call. acmMetrics is part of the multimedia library since at least Microsoft Windows 2000 (according to Microsoft documentation) and in this special case called to trick AV simulation engines.



Picture #1 – Anti-Simulation with ACM call

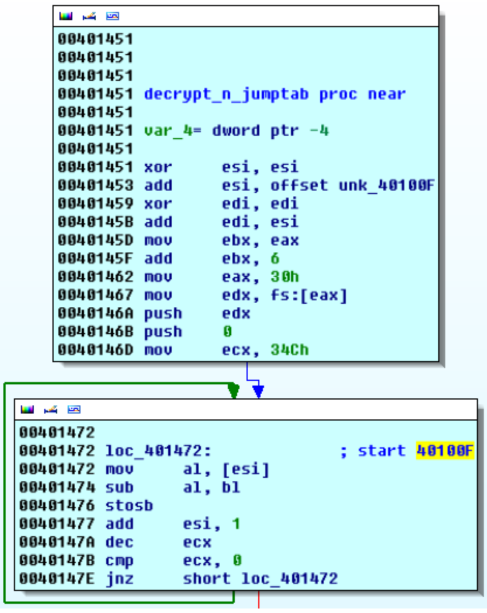
As seen in picture #1 acmMetrics is expected to deliver an error message for an invalid handle, which is not surprising given that the handle parameter is not initialized beforehand. In case the return value is not MMSYSERR_INVALIDHANDLE, code 5, execution continues to access the memory referenced by edx, which at this point always results in a memory access violation. Edx is not initialized thus set to zero.

The point of this check is, on a normal operating system like Windows 2000 or newer this function returns 5 in any case. Simulator engines usually don't support media APIs due to overhead, therefor either crash on the call or later on the access violation.

2.1.2 Decryption & Breakpoint Detection

The protection layer performs minor decryption of a part of its own code, which results in implicit breakpoint detection. The decryption consists in subtracting a key from every opcode of a given section. The key results from the return code of the prior acmMetrics check plus 6, which produces a constant key value of B.

The simple decryption routine iterates code on the position 40100Fh, where execution continues later on. If a software breakpoint is placed in the section to be decrypted the routine produces invalid opcodes and the malware crashes later on.



Picture #2 – Implicit Breakpoint Detection

It is also worth mentioning, that despite an extensive import table numerous relevant API calls are loaded dynamically at runtime and stored as jump table at offset 4064F4h. These include the CreateWindowExA API, which is later on used to direct execution to a window handler function. This aids in obscuring the execution path.

2.1.3 Windowed Confusion

At the end of what could be classified as protection layer stage one the malware invokes CreateWindowExA with a provided WndClass Structure. This structure defines the handler function of the dummy window, which will execute the second part of the protection layer. The created window has no graphical representation, thus can't be seen so just only serves for executing said handler function. If the analyst does not recognize the switch of execution to the handler function and places according breakpoints control of the debugger will be lost.

```
.data:00403059 WndClass WNDCLASSA <0, offset sub_4011A4, 0, 0, offset unk_400000, 0, 0, 10h, 0,\
.data:00403059                                       ; DATA XREF: sub_402000+36↑o
```

Picture #3 – Window Class Structure

It should be noticed, that the early placement of software breakpoints could result in a failure of the initial decryption loop mentioned before. The use of hardware breakpoints or selective enabling and disabling of software breakpoints can prevent this failure.

2.1.4 Second Decryption & Timing Defence

In a second loop content from the .data-section at 40400h is decrypted to allocated memory on the heap; size of that memory is hardcoded in the binary. Data from 40400h is copied to the heap byte wise and decremented by a constant value of 62h.

Interesting in that part of the protection layer is a rdtsc-triggered timing defense. Malware can utilize the system time to verify if a debugger, including a human analyst, is attached to the running process. Windows offers various mechanisms to request the system time, most commonly used are rdtsc or the GetTickCount system call. For detecting an attached debugger/human malware wants to know the time difference between two time stamps, namely if the delta is too big as if the CPU would execute without interruption.

```
.text:0040111A mov     bl, ds:byte_40118F
.text:00401120 rdtsc
.text:00401122 push    eax
.text:00401123 mov     bh, [esi]
.text:00401125 mov     [edi], bh
.text:00401127 inc     edi
.text:00401128
.text:00401128 loc_401128:
.text:00401128 inc     esi
.text:00401129 inc     esi
.text:0040112A inc     esi
.text:0040112B push    eax
.text:0040112C mov     al, [esi]
.text:0040112E stosb
.text:0040112F
.text:0040112F loc_40112F:
.text:0040112F add     [edi-1], bl
.text:00401132 pop     eax
.text:00401133 loop    loc_401128
.text:00401133 decryption_rdtsc endp
.text:00401133
.text:00401135 rdtsc
.text:00401137 pop     edx
.text:00401138 sub     eax, edx
.text:0040113A sub     esp, 18h
.text:0040113D push    0
.text:0040113F push    heap_40304D
.text:00401145 call    dword ptr ds:acmStreamOpen
```

Picture #4 –Timing Detection

The malware at hand issues two rdtsc instructions, wrapped around the decryption loop. The delta is calculated immediately afterwards, but never checked against any threshold. Instead it is kept in eax until the next system call overwrites it with its return value. No other verification could be found, this anti-debugging trick is either broken or the first timestamp servers a different purpose that could not be identified.

2.1.5 Multimedia Threads for Confusion

The Windows media library is used a second time as a means of protection from analysis. The malware issues a call to mciSendStringA with the command “set waveaudio door open”. mciSendStringA sends a command to the media control interface, which is basically used as an interface to multimedia devices on a Windows system.

It is not perfectly clear what purpose the command “set waveaudio door open” usually fulfills, but without doubt the aim of the malware at hand is not to interfere with multimedia devices. An effect of mciSendStringA is that it starts up two additional threads for interaction with devices – the analyst could lose control of the debugger when inappropriately configured. A solution is to configure the debugger to stop on the start-up of a new thread, step back to the original code and continue execution until it returns to the malware code.

2.2 Unpacking Layer

2.2.1 Stage 1

Initially the unpacking routine allocates memory on the heap (subsequently called B1), namely 2432 bytes, and fills it in 4 steps. Effectively, data from four different offsets of the resource section is very simply compressed and written to B1. Data is copied byte-wise, compression is achieved by ignoring a WORD every time a DWORD is finished copying.

```
00970A2A push    dword ptr [ebp-24h]
00970A2D mov     eax, [ebp-20h]
00970A30 mov     bl, 4
00970A32 div     ebx
00970A34 push    eax
00970A35 pop     dword ptr [ebp-18h]    ; mem size div by 4
00970A38 lea     edx, large ds:722Fh
00970A3E add     edx, [ebp-28h]
00970A41 push    dword ptr [ebp-18h]    ; memsize
00970A44 push    dword ptr [ebp-24h]    ; heap mem handle
00970A47 push    edx                    ; addr + offset in .rsrc
00970A48 call    _1_partial_compress
00970A4D mov     eax, [ebp-24h]
00970A50 add     eax, [ebp-18h]
00970A53 mov     [ebp-24h], eax
00970A56 lea     edx, large ds:0DE2h
00970A5C add     edx, [ebp-28h]
00970A5F push    dword ptr [ebp-18h]    ; memsize 260h
00970A62 push    dword ptr [ebp-24h]    ; handle + quarter memsize
00970A65 push    edx
00970A66 call    _1_partial_compress
00970A6B mov     eax, [ebp-24h]
00970A6E add     eax, [ebp-18h]
00970A71 mov     [ebp-24h], eax
00970A74 lea     edx, large ds:75BFh
00970A7A add     edx, [ebp-28h]
00970A7D push    dword ptr [ebp-18h]
00970A80 push    dword ptr [ebp-24h]
00970A83 push    edx
00970A84 call    _1_partial_compress
00970A89 mov     eax, [ebp-24h]
00970A8C add     eax, [ebp-18h]
00970A8F mov     [ebp-24h], eax
00970A92 lea     edx, large ds:26Fh
00970A98 add     edx, [ebp-28h]
00970A9B push    dword ptr [ebp-18h]
00970A9E push    dword ptr [ebp-24h]
00970AA1 push    edx
00970AA2 call    _1_partial_compress    ; decompression in 4 acts
00970AA7 pop     dword ptr [ebp-24h]
```

Picture #5 – Composition of the 1st Stage Malware

That same block of data in B1 subsequently gets decrypted. The decryption routine accepts six parameters, including a handle to the allocated heap memory and its size, a XOR-key, a subtraction-key, and a key-modifier value. The algorithm for decryption is simple and straight forward, explanatory pseudo code is provided in the following diagram.

In a first 10-fold loop the initial XOR-key value is rotated by subtracting a static key-modifier value. In a second loop data from B1 is loaded DWORD-wise, XORed with the XOR-key and subtracted by the subtraction-key. For each iteration, the subtraction-key is decremented by the XOR-key, the latter one is afterwards bitwise rotated to the right (ror).

```
WORD key_modifier = AC601330h
WORD sub_key = E25ED5B0h
WORD xor_key = 86E53278h
(WORD*) code = [offset_buffer]
int memsize = [size_buffer]

for (i=0; i<10; i++)
{
    xor_key -= key_modifier          // underflow
}

for (i=0; i<memsize; i++)
{
    code[i] = code[i] XOR xor_key    // modify data
    code[i] -= sub_key

    sub_key -= xor_key               // modify keys
    ROR(xor_key)
}
```

Picture #6 – Decryption

2.2.2 Stage 2

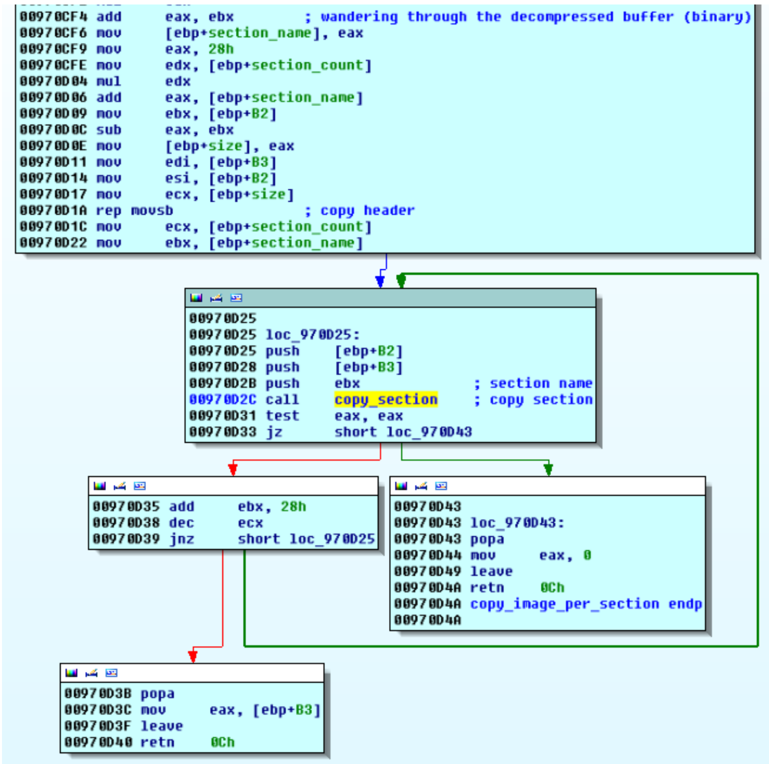
In the second stage another buffer (B2) is allocated and the API function RTLDecompressBuffer is used to automatically decompress the decrypted buffer B1 into the newly allocated memory. The compression format used is identified by 102h, which indicates that LZNT1 (Lempel-Ziv Algorithm) – 2h – with a maximum compression level – 100h – was used for compression.

2.2.3 Stage 3

Stage three starts with a simple check on the header of the now complete executable image in memory B2. It is verified, if the MZ and the PE signature of a PE32-header are in place.

For finalization of the unpacked image a third buffer B3 is allocated and, relying on the section information in the header, the image in B2 is copied section per section into the new buffer, including the header.

After the copy process, the MZ/PE check is repeated on the image in B3.



Picture #7 – Final Image Creation

Finally the unpacking routine enters a function which patches function call offsets in the copied binary image in buffer B3 to suit the memory region the newly created executable is located at. Therefor the difference between the offsets is calculated by subtraction of the imagebase from B3's offset, namely B10000h – 400000h = 710000h. This difference is then added to function call offsets in the unpacked code, which initially accord to the standard image base of the packed executable. So, for example the operation 'call off_402020' will be patched to 'call off_B12020'.

The location of the function call offsets is determined by a jump table, located at offset 5000h of the unpacked binary.

In a nutshell, the statically coded 40xxxxh offsets in the unpacked code are patched to match the new absolute function addresses on the heap, B1xxxxh in the given case.

Picture #8 – Function Address Fixing

2.2.4 IAT Reconstruction

Finally the last step before executing the actual downloader is to reconstruct the import address table (IAT). This means the unpacking routine iterates through a list of library names and accordingly through lists of API names per library to identify the absolute function address and to store it in the import segment.

More specifically, the unpacking routine walks through a list of IMAGE_IMPORT_DIRECTORY structures which stores the library name, a pointer to the import lookup table and a pointer to the import thunk table for each library. Mentioned import lookup table consists of 4-byte records, which can either specify an ordinal value or a pointer to the API function name; for the given binary the latter one was the case.

Following these pointers, the routine can grab the API names and uses GetProcAddress to resolve the API addresses, which get stored in the import thunk table

A comprehensive write up of this IAT reconstruction mechanism can be found at http://sandsprite.com/CodeStuff/Understanding_imports.html.

ATTACHMENT A: List of imported APIs

WININET.dll

InternetOpenW
InternetConnectW
HttpOpenRequestW
InternetQueryOptionW
InternetSetOptionW
HttpSendRequestW
HttpQueryInfoW
InternetReadFile
GetModuleHandleW

KERNEL32.dll

ExitProcess
HeapCreate
HeapAlloc
GetModuleFileNameW
GetTempPathW
CreateFileW
GetFileSize
lstrlenW
ReadFile
lstrcmpW
WriteFile
CloseHandle
DeleteFileW
GetCurrentDirectoryW

USER32.dll

wsprintfW

SHELL32.dll

ShellExecuteW

ATTACHMENT B: List of remarkable strings in final binary

/images/html.exe

davistructures.com

mentoringgroup.com

budha.exe

open

Updates downloader

text/*

application/*

kilf.exe