Operation CuckooBees: A Winnti Malware Arsenal Deep-Dive

Written By
Cybereason Nocturnus

May 4, 2022 | 19 minute read

In part one of this research, the Cybereason Nocturnus Incident Response Team provided a unique glimpse into the Winnti intrusion playbook, covering the techniques that were used by the group from initial compromise to stealing the data, as observed and analyzed by the Cybereason Incident Response team.

This part of the research zeroes in on the Winnti malware arsenal that was discovered during the investigation conducted by the Cybereason IR and Nocturnus teams. In addition, our analysis of the observed malware provides a deeper understanding of the elaborate and multi-layered Winnti infection chain, including evasive maneuvers and stealth techniques that are baked-in to the malware code, as well as the functionality of the various malware.

Perhaps one of the most interesting and striking aspects of this report is the level of sophistication introduced by the malware authors. The infection and deployment chain is long, complicated and interdependent–should one step go wrong, the entire chain collapses - making it somewhat vulnerable, yet at the same time provides an extra level of security and stealth for the operation.

These steps have proven themselves effective time and time again, as the operation remained under-the-radar for years. While there have been past reports describing some aspects of these intrusions, at the time of writing this report there was no publicly available research that discussed all of the tools and techniques and the manner in which they all fit together, as mentioned in this report.

Key Findings

Attribution to the Winnti APT Group: based on the analysis of the forensic artifacts, Cybereason estimates with medium-high confidence that the perpetrators of the attack are linked to the notorious Winnti APT group, a group that has existed since at least 2010 and is believed to be operating on behalf of Chinese state interests and specializes in cyberespionage and intellectual property theft.

- **Discovery of New Malware in the Winnti Arsenal:** the report exposes previously undocumented malware strain called DEPLOYLOG used by the Winnti APT group and highlights new versions of known Winnti malware, including Spyder Loader, PRIVATELOG, and WINNKIT.
- **Rarely Seen Abuse of the Windows CLFS Feature:** the attackers leveraged the Windows CLFS mechanism and NTFS transaction manipulations which provided them with the ability to conceal their payloads and evade detection by traditional security products.

Intricate and Interdependent Payload Delivery: the report includes an analysis of the complex infection chain that led to the deployment of the WINNKIT rootkit composed of multiple interdependent components. The attackers implemented a delicate "house of cards" approach, meaning that each component depends on the others to function properly, making it very difficult to analyze each component separately. The malware from the Winnti arsenal that are analyzed in this report include:

Spyder: A sophisticated modular backdoor

**STASHLOG:** The initial deployment tool "stashing" payloads in Windows CLFS

SPARKLOG: Extracts and deploys PRIVATELOG to gain privilege escalation and achieve persistence

PRIVATELOG: Extracts and deploys DEPLOYLOG

DEPLOYLOG: Deploys the WINNKIT Rootkit and serves as a userland agent

WINNKIT: The Winnti Kernel-level Rootkit

The following graph describes the infection chain presented in this attack:
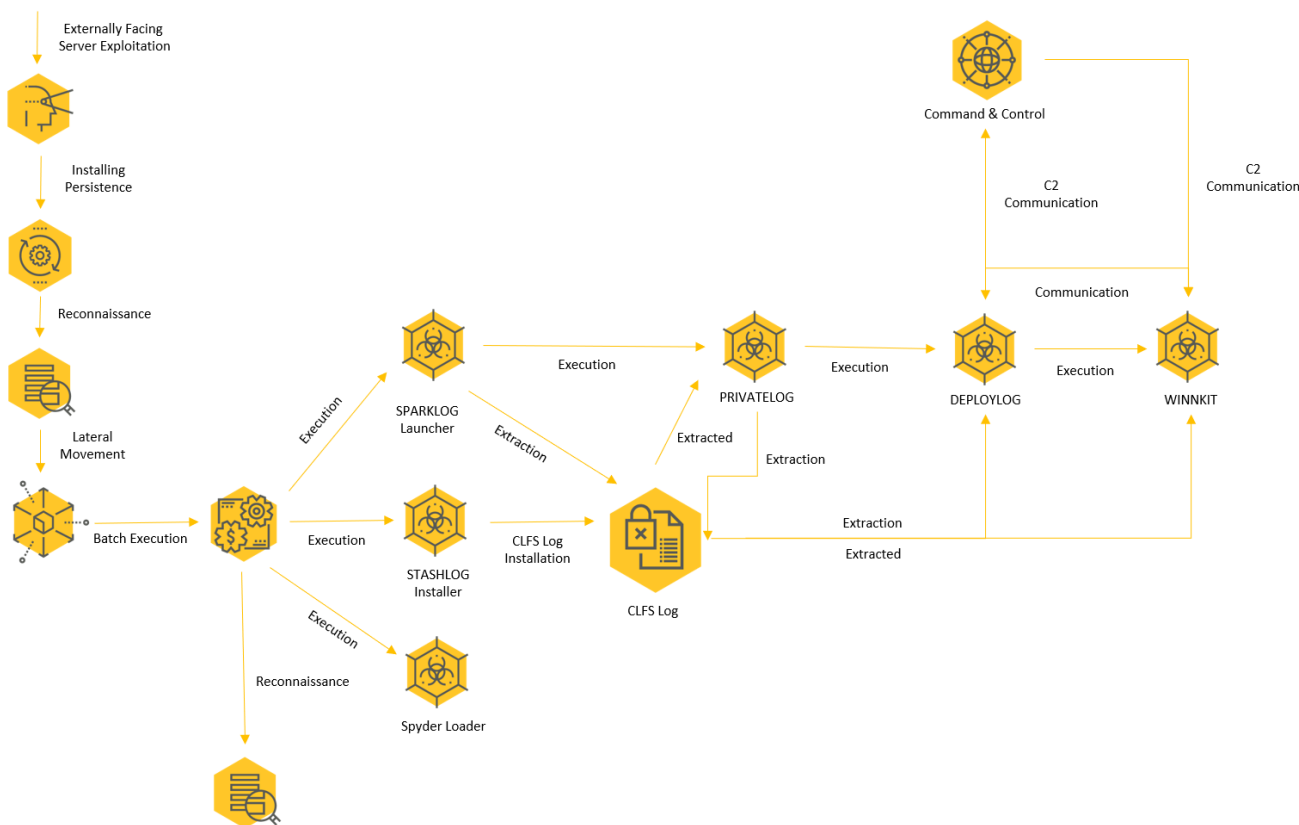
Winnti infection chain as observed in Operation CuckooBees

Initial Payload: Weaving in the Spyder Loader

Batch Execution → Spyder Loader

The Spyder loader is the first malicious binary the attackers execute on a targeted machine. This malware is executed from the batch files we discussed in our blog's part 1 - *cc.bat* or *bc.bat*:

Batch file execution command

```
cc.bat - Notepad
File  Edit  Format  View  Help
rundll32.exe C:\Windows\System32\iumatl.dll,#138 C:\Windows\System32\x64.tlb
```

The loader's purpose is to decrypt and load additional payloads and is being delivered in 2 variations. The first variation, is a modified *SQLite3* DLL, that uses the export's ordinal number 138 to serve malicious code, that loads and executes a file argument provided at runtime, in our case *C:\Windows\System32\x64.tlb*:

| f | sqlite3_profile | 00000001800A5C00 | 137 |
| f | sqlite3_profile_v2 | 00000001800020B0 | 138 |

Malicious export ordinal number 138 "sqlite3_profile_v2"

As seen above, the loader is executed via the famous *LOLBIN rundll32.exe*, in the following manner:

rundll32.exe <modified sqlite3.dll file>,#138 C:\Windows\System32\x64.tlb

Interestingly, Cybereason found this loader in different names and in different locations across infected machines:

- C:\Windows\System32\iumatl.dll
- C:\Windows\System32\msdupld.dll
- C:\Windows\System32\mscuplt.dll
- C:\Windows\System32\msdupld.dll
- C:\Windows\System32\netapi.dll
- C:\Windows\System32\rpcutl.dll
- C:\Windows\System32\dot3utl.dll
- C:\Windows\System32\nlsutl.dll

- C:\Windows\Branding\Basebrd\language.dll
- C:\Program Files\Internet Explorer\SIGNUP\install.dll

The attackers utilized the System32 directory, which holds a multitude of TLB and DLL files, to hide their external "TLB" payload and DLL loader to make it harder to detect.

This DLL wasn't the only Spyder Loader we found, as Cybereason discovered a second variation of this malware in the form of a standalone executable called *sqlite3.exe,* masquerading as a *SQLite3*-related executable as well.

This version featured some improvements, such as logging messages, which shed some light on some of its functionality and capabilities:

```
wmain proc near

nSize= dword ptr -158h
Block= qword ptr -150h
var_140= qword ptr -140h
var_138= qword ptr -138h
Buffer= byte ptr -128h
var_127= byte ptr -127h
var_18= qword ptr -18h


sub     rsp, 178h
mov     rax, cs:qword_7FF7594261A0
xor     rax, rsp
mov     [rsp+178h+var_18], rax
lea     rcx, [rsp+178h+var_127] ; void *
xor     edx, edx        ; Val
mov     r8d, 103h        ; Size
mov     [rsp+178h+nSize], 0
mov     [rsp+178h+Buffer], 0
call    memset
lea     rcx, aProfileParsing ; "Profile parsing, part %u.\n"
mov     edx, 1
call    printf
lea     rdx, [rsp+178h+nSize] ; nSize
lea     rcx, [rsp+178h+Buffer] ; lpBuffer
mov     [rsp+178h+nSize], 104h
call    cs:GetComputerNameA
cmp     [rsp+178h+Buffer], 0
jz      short loc_7FF759332100
```

```
lea     rcx, aProfileParsing ; "Profile parsing, part %u.\n"
mov     edx, 2
call    printf
lea     rdx, [rsp+178h+Buffer]
lea     rcx, [rsp+178h+Block]
call    sub_7FF759331DB0
cmp     [rsp+178h+var_140], 0DBBA0h
jnz     short loc_7FF7593320EE
```

Malicious export ordinal number 138 "sqlite3_profile_v2"

The Spyder Loader Bag of Tricks

Throughout its execution, the Spider Loader implements a few interesting methods to evade detection and to maintain stealth:

Anti Analysis/Sandboxing

At the beginning of execution, the loader checks if the file argument exists:

```
                    loc_7FF94C6B2115:
                    ; __unwind { // __GSHandlerCheck
48 89 9C 24 C0 01+mov      [rsp+1B0h+arg_0], rbx
00 00
33 DB              xor     ebx, ebx
45 33 C9           xor     r9d, r9d          ; lpSecurityAttributes
48 89 5C 24 30     mov     [rsp+1B0h+hTemplateFile], rbx ; hTemplateFile
44 8D 43 01        lea     r8d, [rbx+1]      ; dwShareMode
BA 00 00 00 80     mov     edx, GENERIC_READ ; dwDesiredAccess
48 8B CF           mov     rcx, rdi          ; lpFileName - C:\Windows\System32\x64.tlb
C7 44 24 28 80 00+mov      [rsp+1B0h+dwFlagsAndAttributes], 80h ; '€' ; dwFlagsAndAttributes
00 00                     ; } // starts at 7FF94C6B2115
```

```
                    loc_7FF94C6B213B:
                    ; __unwind { // __GSHandlerCheck
48 89 B4 24 C8 01+mov      [rsp+1B0h+arg_8], rsi
00 00
89 5C 24 40        mov     [rsp+1B0h+nSize], ebx
48 89 5C 24 48     mov     qword ptr [rsp+1B0h+FileSize], rbx
C7 44 24 20 03 00+mov      [rsp+1B0h+dwCreationDisposition], 3 ; dwCreationDisposition
00 00
FF 15 36 E0 0C 00 call     cs:__imp_CreateFileW
```

File argument validation

If it does, the loader checks for its size: if it is larger than 1.04 MB, it deletes it; if it is smaller than 1.04 MB or equal to it, it decrypts it in memory using the open-source CryptoPP C++ library and then deletes it from disk.

Cybereason assesses this condition is intended to validate that the loader won't try to decrypt the wrong file, and as a precaution against analysis environments or Sandboxes.

EDR Bypass Tricks

After decrypting the payload, the attackers copy the system ntdll.dll file to *C:\Windows\System32\TN{random_characters}.dll*, and load it to memory:

```
cmovnb  rcx, [rsp+1B0h+to_be_system32_path] ; lpExistingFileName - C:\Windows\System32\ntdll.dll
mov     r8d, 1             ; bFailIfExists
lea     rdx, [rbp+0B0h+system_dir_path] ; lpNewFileName - C:\Windows\System32\TN{random_chars}.dll
call    cs:CopyFileA
```

Copying ntdll.dll to C:\Windows\System32\TN{randoms_chars}.dll

Then, they acquire the *NtProtectVirtualMemory* address from the copied and loaded file and call a specific routine (which we named as "BypassEdrHook") multiple times using 2 arguments:

- The acquired address of *NtProtectVirtualMemory* in the loaded, copied DLL.

- A string representing a native api function:

```
if ( TN_NtProtectVirtualMemory )
{
  BypassEdrHook(
    (__int64 (__fastcall *)(__int64, FARPROC *, __int64 *, __int64, unsigned int *))TN_NtProtectVirtualMemory,
    "LdrLoadDll");
  BypassEdrHook(
    (__int64 (__fastcall *)(__int64, FARPROC *, __int64 *, __int64, unsigned int *))::TN_NtProtectVirtualMemory,
    "KiUserApcDispatcher");
  BypassEdrHook(
    (__int64 (__fastcall *)(__int64, FARPROC *, __int64 *, __int64, unsigned int *))::TN_NtProtectVirtualMemory,
    "NtAlpcConnectPort");
```

Calling for the BypassEdrHook routine

The *BypassEdrHook* function will compare the first bytes in memory of the native API functions in the loaded *ntdll.dll* image to the first bytes in memory of the same function in the loaded/copied DLL memory image.

If the *ntdll* function's first bytes are different from the first bytes of the copied DLL in memory, the attackers will conclude that this native function is hooked by an EDR tool.

To override it, the attackers count the number of different bytes at the beginning of these two functions, then they change the permissions of the relevant patched bytes in the original *ntdll.dll* image to *READWRITE_EXECUTE*, copy the original bytes from the loaded/copied DLL memory, and restore the previous page protection settings:

```
ntdll_func_addr = GetProcAddress(ntdll_dll, called_function);
if ( !ntdll_func_addr )
  return GetLastError();
TN_dlll_func_addr = GetProcAddress(TN_dll, called_function);
TN_dlll_func_addr_copy = TN_dlll_func_addr;
if ( !TN_dlll_func_addr )
  return GetLastError();
counter = 0;
ntdll_func_addr_copy = ntdll_func_addr;
do
{
  if ( *ntdll_func_addr_copy == ntdll_func_addr_copy[(char *)TN_dlll_func_addr - (char *)ntdll_func_addr] )
    break;
  ++counter;
  ++ntdll_func_addr_copy;
}
while ( counter < 10 );
if ( !counter )
  return 0;
size = counter;
oldProtection = 0;
ntdll_func_addr_copy_2 = ntdll_func_addr;
num_bytes = counter;
result = TN_NtProtectVirtualMemory(-1i64, &ntdll_func_addr_copy_2, &num_bytes, PAGE_EXECUTE_READWRITE, &oldProtection);
if ( result >= 0 )
{
  memmove(ntdll_func_addr, TN_dlll_func_addr_copy, (unsigned int)size);
  num_bytes = size;
  ntdll_func_addr_copy_2 = ntdll_func_addr;
  TN_NtProtectVirtualMemory(-1i64, &ntdll_func_addr_copy_2, &num_bytes, oldProtection, &oldProtection);
  return 0;
}
```

Check for EDR hook and bypass if true

This procedure will occur for the following native API functions:

- LdrLoadDll

- KiUserApcDispatcher
- NtAlpcConnectPort
- NtAllocateVirtualMemory
- NtFreeVirtualMemory
- NtMapViewOfSection
- NtQueueApcThread
- NtReadVirtualMemory
- NtSetContextThread
- NtUnmapViewOfSection
- NtWriteVirtualMemory
- RtlInstallFunctionTableCallback

Right afterward, the Spyder Loader will execute the payload reflectively, and lastly, will delete *TN{random_characters}.dll* to leave no traces.

Attribution of the Spyder Payloads

The above-mentioned PE files share similar code with other known Spyder loaders, such as the *oci.dll* payload mentioned in a SonicWall blog from March 2021:

Allocate memory and save WINAPI functions in array

lumatl.dll   oci.dll


```
mov    rcx, rbx
call   sub_7FF942874550
```

```
loc_7FF942874979:
mov    rcx, rbx
call   sub_7FF942874600
test   eax, eax
jz     short loc_7FF9428749BA
```

```
mov    rcx, rbx
call   sub_7FF9428743B0
test   eax, eax
jz     short loc_7FF9428749BA
```

```
mov    rax, [rbx]
mov    ecx, [rax+28h]
test   ecx, ecx
jz     short loc_7FF9428749EA
```

```
xor    r8d, r8d
lea    rax, [rdi+rcx]
mov    rcx, rdi
lea    edx, [r8+1]
call   rax
test   eax, eax
jnz    short loc_7FF9428749E3
```

```
mov    rcx, rdi
call   sub_180001200
```

```
loc_1800016A4:
mov    rcx, rdi
call   sub_1800012B0
test   eax, eax
jz     short loc_1800016E1
```

```
mov    rcx, rdi
call   sub_1800010F0
mov    r11, [rdi]
mov    eax, [r11+28h]
test   eax, eax
jz     short loc_180001711
```

```
xor    r8d, r8d
add    rax, rbp
mov    rcx, rbp
lea    edx, [r8+1]
call   rax
test   eax, eax
jnz    short loc_18000170A
```

Jump to the payload

Moreover, our PE files also share a similar evasion technique in masquerading as a legitimate executable. In the aforementioned blog post, it disguised as *D3D DLL - a Direct3D 11* runtime DLL, now it disguise as *SQLite3:*

Name	Address	Ordinal		Name	Address	Ordinal
sqlite3_get_autocommit	00000001800A7A60	97		D3D11On12CreateDevice	000000018000FC68	10
sqlite3_get_auxdata	000000018003A70	98		D3DKMTCloseAdapter	000000018000FCAB	11
sqlite3_get_table	00000001800822C0	99		D3DKMTCreateAllocation	000000018000FCEF	12
sqlite3_global_recover	000000018007E6A0	100		D3DKMTCreateContext	000000018000FD34	13
sqlite3_initialize *iumatl.dll*	00000001800A3520	101		D3DKMTCreateDevice *oci.dll*	000000018000FD75	14
sqlite3_interrupt	00000001800A51B0	102		D3DKMTCreateSynchronizationObject	000000018000FDC4	15
sqlite3_keyword_check	00000001800A1F90	103		D3DKMTDestroyAllocation	000000018000FE18	16
sqlite3_keyword_count	00000001800A1F80	104		D3DKMTDestroyContext	000000018000FE5F	17
sqlite3_keyword_name	00000001800A1F40	105		D3DKMTDestroyDevice	000000018000FEA2	18
sqlite3_last_insert_rowid	00000001800A41D0	106		D3DKMTDestroySynchronizationObject	000000018000FEF3	19
sqlite3_libversion	00000001800A3500	107		D3DKMTEscape	000000018000FF3D	20
sqlite3_libversion_number	00000001800A3510	108		D3DKMTGetContextSchedulingPriority	000000018000FF87	21
sqlite3_limit	00000001800A6880	109		D3DKMTGetDeviceState	000000018000FFD9	22
sqlite3_load_extension	000000018006D7D0	110				

Same method with different targeted DLLs

These similarities, in addition to others, have led us to conclude that this file is an evolution of the Winnti Spyder Loader.

A Long and Winnti(ng) Road: The Winnti Multi-Phased Arsenal Deployment

After deploying the initial payload, Winnti employs a sophisticated and unique multi-staged infection chain with numerous payloads. Each payload fulfills a unique role in the infection chain, which is successful only upon the complete deployment of all of the payloads.

In the upcoming sections, we will discuss the following payloads:

- **STASHLOG**: Stashes encrypted data in a CLFS log
- **SPARKLOG:** Extracts data from the CLFS log and deploys PRIVATELOG while escalating privileges
- **PRIVATELOG:** Extracts data from the CLFS log and deploys DEPLOYLOG. This payload also enables persistence in some cases
- **DEPLOYLOG:** Extracts data from the CLFS log, deploys the WINNKIT Rootkit driver, and acts as the user-mode agent
- **WINNKIT**: The Winnti Kernel-level Rootkit

Several unique techniques are used by the Winnti group to store data, evade detection, and thwart investigations during this infection flow. One of those techniques, which Winnti heavily uses, is the CLFS mechanism.

Abusing the Rarely Used CLFS Mechanism for Evasion

So what is CLFS?

CLFS (Common Log File System) is a logging framework that was first introduced by Microsoft in Windows Server 2003 R2, and is included in later Windows operating systems.

This mechanism provides a high-performance logging system for a variety of purposes ranging from simple error logs to transactional systems and data stream collection.

One of the main uses of CLFS in the Windows operating system is in the Windows Kernel Transaction Manager (KTM) for both Transactional NTFS (TxF) and Transactional Registry (TxR) operations. Transactional operations bring the concept of atomic transactions to Windows, allowing Windows to log different operations on those components and support the ability to roll back if needed.

The high-performance aspect of this framework is based on the concept of storing the log data in memory buffers for fast writing, reading and flushing them to disk–but not continuously, according to a stated policy.

CLFS employs a proprietary file format that isn't documented, and can only be accessed through the CLFS API functions. As of writing this report, there is no tool which can parse the flushed logs. This is a huge benefit for attackers, as it makes it more difficult to examine and detect them while using the CLFS mechanism.

On disk, the CLFS log store consists of:

- Base Log File (BLF file): Contains the log metadata.
- One or more Container files: Contains the log data, where the container file sizes are registered in the BLF file.

As will be discussed, Winnti group used this mechanism to store and hide the payload that will be extracted from the CLFS file and used by other PEs in the execution chain to build the attacker's next steps.

STASHLOG: Stashing the Winnti Arsenal via CLFS
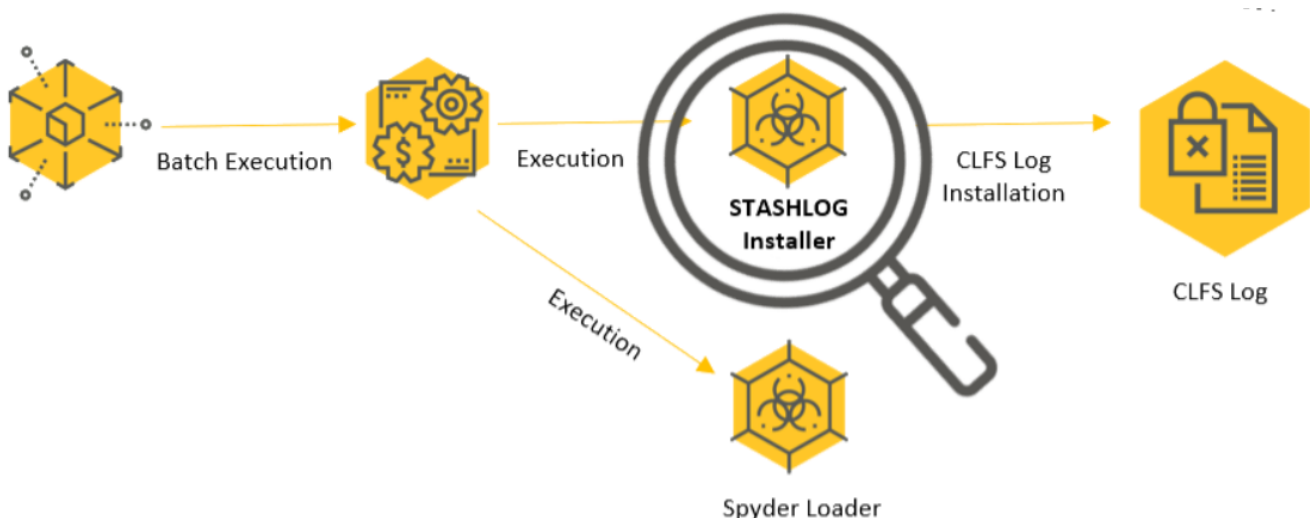
STASHLOG (shiver.exe / forsrv.exe) is a 32 bit executable that is being used to prepare the victim machine for further compromise, and to "stash" a malicious, encrypted payload to a CLFS log file. This payload will be decrypted at each phase to deliver the next phase in the infection.

Both STASHLOG and SPARKLOG, which will be described further in the next section, are executed using a second *cc.bat* file in the following form:

cc.bat - Notepad

File Edit Format View Help

```
C:\Windows\AppPatch\Custom\Custom64\Shiver.exe C:\Windows\AppPatch\Custom\Custom64\log.dat >>
C:\Windows\AppPatch\Custom\Custom64\1.log
C:\Windows\AppPatch\Custom\Custom64\spark.exe
```
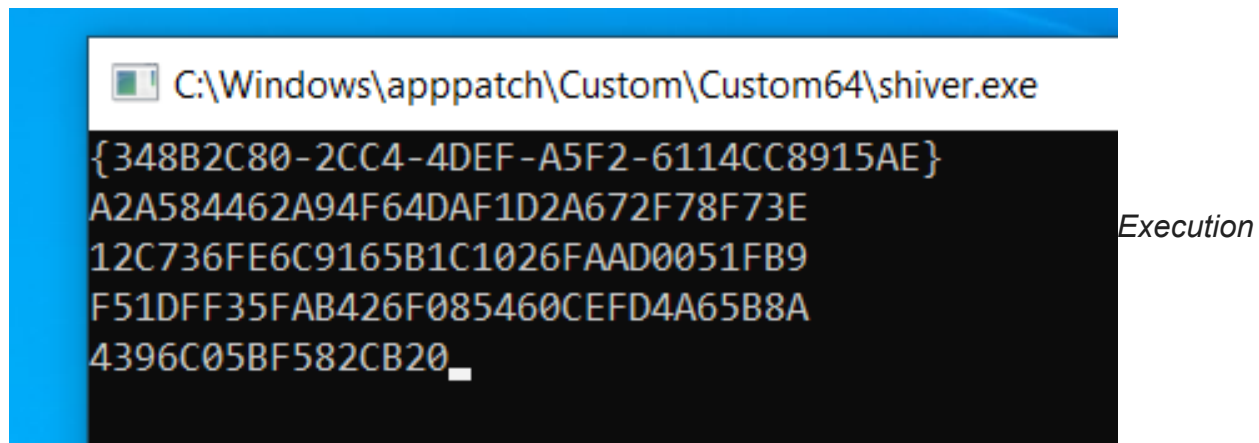
STASHLOG and SPARKLOG execution

Different Modes of Execution

STASHLOG has 2 modes of execution:

- Without arguments to initialize the environment towards infection.
- With one argument to store the CLFS file for further use.

No Arguments Mode

When no arguments are passed, STASHLOG prints the following output:

C:\Windows\apppatch\Custom\Custom64\shiver.exe

```
{348B2C80-2CC4-4DEF-A5F2-6114CC8915AE}
A2A584462A94F64DAF1D2A672F78F73E
12C736FE6C9165B1C1026FAAD0051FB9
F51DFF35FAB426F085460CEFD4A65B8A
4396C05BF582CB20
```

Execution

example of STASHLOG without an argument

This output consists of the machine Globally Unique Identifier (GUID) along with a 56 byte string:

- The machine's GUID - derived from the *HKLM\SOFTWARE\Microsoft\Cryptography* registry key.
- The 56 byte string is generated from a random GUID created by the *CoCreateGUID* function. The string consists of the hex representation of the GUID, SHA1 hash of the hex GUID, and a SHA1 hash of the GUID + SHA1 of the GUID.

This example demonstrates the process of building the 56 byte string:

- Let's take a random GUID: {4684A5A2-942A-4DF6-AF1D-2A672F78F73E}
- The random GUID's Hex representation will be:
  A2A584462A94F64DAF1D2A672F78F73E
- Its SHA1 will be: 12C736FE6C9165B1C1026FAAD0051FB9F51DFF35

- And finally the SHA1 of (GUID + SHA1(GUID)):
  FAB426F085460CEFD4A65B8A4396C05BF582CB20

The final string will be:

    A2A584462A94F64DAF1D2A672F78F73E
    12C736FE6C9165B1C1026FAAD0051FB9F51DFF35
    FAB426F085460CEFD4A65B8A4396C05BF582CB20

The random GUID is then registered as a global atom entry in the form of *win::{GUID}*:

C09E = shell32.dll,-160 --GlobalAtom
C09F = win::{4684A5A2-942A-4DF6-AF1D-2A672F78F73E} --GlobalAtom *An*
C0A1 = shell:::{2559a1f3-21d7-11d4-bdaf-00c04f60b9f0} --GlobalAtom

example of added Global Atom by STASHLOG

If an atom prefixed with *win::* already exists during execution, then the existing atom will be used instead of the newly generated one.

One Argument Mode

Upon execution with command line arguments, STASHLOG checks the existence of the global atom table entry *win::{GUID}*, and the process will quit immediately unless the value exists.

Then, STASHLOG loads the file found in the given argument into memory, checks its content, and stores it in a CLFS log file. STASHLOG prints log information about this process to the terminal:

- Preparing for log format transformation
- Log transform step 1 completed
- Log transform step 2 completed
- Log transform step 3 completed
- Log data transform completed
- Successful STASHLOG execution log

This output in this operation was redirected to a log file, as can be seen on *cc.bat*. Breaking down the transformation steps from the log:

- **Step 1:** Decrypt the given encrypted file and check the file validity by looking for the destination BLF file location: *C:\Users\Default\NTUSER.DAT{<GUID>}.TM.BLF*. If the search yields no results, STASHLOG will create this file.
- **Step 2:** Clear the targeted BLF file.
- **Step 3:** Encrypt the malicious data and write it to a CLFS file.

Anti-Analysis Techniques
String Obfuscation

Throughout the execution, STASHLOG uses the same string decryption technique used in other samples by XORing strings with pre-defined bytes, words or qwords, using the XMM registers. This decryption sequence is present at the beginning of every function that uses strings, where those strings are not saved globally, likely in an attempt to protect them.

As can be seen in the example below from STASHLOG, it uses it to decrypt several debug strings:

```
00FC191E 058 mov      al, byte_FF2835
00FC1923 058 mov      cl, byte_FF2839
00FC1929 058 movups   xmm0, xmmword_FF2930
00FC1930 058 mov      ah, byte_FF2945
00FC1936 058 mov      dl, 25h
00FC1938 058 mov      ch, byte_FF2A49
00FC193E 058 xor      al, 6Eh
00FC1940 058 xorps    xmm0, ds:xmmword_FD9720
00FC1947 058 xor      ch, 0B8h
00FC194A 058 mov      byte_FF2855, al
00FC194F 058 mov      al, byte_FF2836
00FC1954 058 movups   STR_log_prep_error, xmm0
00FC195B 058 movups   xmm0, xmmword_FF29D0
00FC1962 058 xor      al, 9Ch
00FC1964 058 mov      byte_FF2856, al
00FC1969 058 mov      al, byte_FF2837
00FC196E 058 xorps    xmm0, ds:xmmword_FD9730
00FC1975 058 xor      al, 8Bh
00FC1977 058 mov      byte_FF2857, al
00FC197C 058 mov      al, byte_FF2838
00FC1981 058 movups   STR_log_trans_step_1, xmm0
00FC1988 058 movups   xmm0, xmmword_FF29E0
00FC198F 058 xor      al, 0D7h
00FC1991 058 mov      byte_FF2858, al
00FC1996 058 mov      al, 0FDh
00FC1998 058 xorps    xmm0, ds:xmmword_FD9740
00FC199F 058 xor      cl, al
00FC19A1 058 xor      al, byte_FF283D
00FC19A7 058 mov      byte_FF2859, cl
00FC19AD 058 mov      cl, byte_FF283A
00FC19B3 058 movups   xmmword_FF2A10, xmm0
00FC19BA 058 movups   xmm0, xmmword_FF2A30
00FC19C1 058 xor      cl, 3Bh
00FC19C4 058 mov      byte_FF285A, cl
00FC19CA 058 mov      cl, byte_FF283B
00FC19D0 058 xorps    xmm0, ds:xmmword_FD9750
00FC19D7 058 xor      cl, 68h
00FC19DA 058 mov      byte_FF285B, cl
00FC19E0 058 mov      cl, byte_FF283C
00FC19E6 058 movups   STR_get_dst_file_path_error, xmm0
```

String

decryption algorithm from shiver.exe

This technique is also being used in other samples in the chain, including SPARKLOG, PRIVATELOG and DEPLOYLOG.

Anti-Disassembly

This sample uses an interesting Anti-Disassembly technique which thwarts the disassembly process and makes the investigation job harder. Each function inside STASHLOG contains a jump list of every node in the function:

```
data:00FF52E8 50 24 FC 00 84 25 FC 00    off_FF52E8      dd offset loc_FC2450    ; DATA XREF: Huge_switch+E62↑r
data:00FF52E8                                             dd offset loc_FC2584    ; jump table for switch statement
data:00FF52F0 84 25 FC 00                                dd offset loc_FC2584    ; jumptable 00FC2449 case 1
data:00FF52F4 84 25 FC 00                                dd offset loc_FC2584    ; jumptable 00FC2449 case 1
data:00FF52F8 84 25 FC 00                                dd offset loc_FC2584    ; jumptable 00FC2449 case 1
data:00FF52FC                            ; int (*off_FF52FC)(void)
data:00FF52FC 8A 25 FC 00                off_FF52FC      dd offset loc_FC258A    ; DATA XREF: Huge_switch:loc_FC2584↑r
data:00FF5300 90 25 FC 00                                dd offset loc_FC2590
data:00FF5304                            ; int (*off_FF5304)(void)
data:00FF5304 9D 25 FC 00                off_FF5304      dd offset loc_FC259D    ; DATA XREF: Huge_switch:loc_FC258A↑r
data:00FF5308 9D 25 FC 00                                dd offset loc_FC259D
data:00FF530C                            ; int (*off_FF530C)(void)
data:00FF530C EA 28 FC 00                off_FF530C      dd offset loc_FC28EA    ; DATA XREF: Huge_switch+FC5↑r
data:00FF5310 50 24 FC 00                                dd offset loc_FC2450    ; jumptable 00FC2449 case 0
data:00FF5314 50 24 FC 00                                dd offset loc_FC2450    ; jumptable 00FC2449 case 0
data:00FF5318 50 24 FC 00                                dd offset loc_FC2450    ; jumptable 00FC2449 case 0
data:00FF531C                            ; int (*off_FF531C)(void)
data:00FF531C 56 24 FC 00                off_FF531C      dd offset loc_FC2456    ; DATA XREF: Huge_switch:loc_FC2450↑r
data:00FF5320 70 24 FC 00                                dd offset loc_FC2470    ; loc_FC2472
data:00FF5324 70 24 FC 00                                dd offset loc_FC2470    ; loc_FC2472
data:00FF5328 70 24 FC 00                                dd offset loc_FC2470    ; loc_FC2472
data:00FF532C 72 24 FC 00                off_FF532C      dd offset loc_FC2472    ; DATA XREF: Huge_switch+E84↑r
data:00FF5330 7A 24 FC 00                                dd offset loc_FC247A    ; loc_FC247C
data:00FF5334 7A 24 FC 00                                dd offset loc_FC247A    ; loc_FC247C
data:00FF5338 7A 24 FC 00                                dd offset loc_FC247A    ; loc_FC247C
data:00FF533C 7C 24 FC 00                off_FF533C      dd offset loc_FC247C    ; DATA XREF: Huge_switch:loc_FC2472↑r
data:00FF533C                                            ; iid retrieve check
data:00FF533C                                            ; True is err
data:00FF5340 8A 24 FC 00 B2 25 FC 00    off_FF5340      dd offset loc_FC248A    ; DATA XREF: Huge_switch+E9C↑r
data:00FF5340                                            dd offset loc_FC25B2    ; jump table for switch statement
```

Part of a Jump List from a function in STASHLOG

This jump list is then used as a flow control during the function execution. In some cases the address of the next node will be pushed to a register and then there will be a *JMP* to this register:

```
00FC2654 058 mov      eax, off_FF54BC
00FC2659 058 mov      ecx, [esi+14h]
00FC265C 058 mov      edx, [esi+0Ch]
```
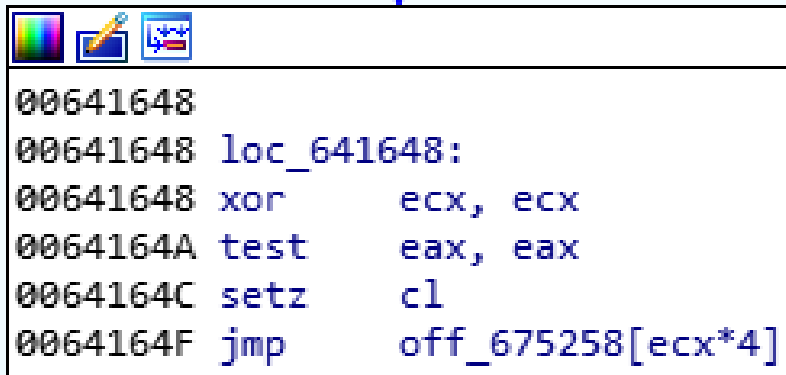
```
00FC265F
00FC265F           loc_FC265F:
00FC265F 058 jmp         eax
```

Jump

to register

The more interesting use of the jump list is replacing the conditional jump commands like *JZ* and *JNZ* with a SET command that changes the register given as an argument to the value of the corresponding checked flag. For example, when using *SETZ eax,* the EAX register will be changed to "1" if the "0" flag is checked.

Using the set register, there will be a JMP to an address on the function's jump list. This JMP is usually resolved for a switch-case mechanism, and IDA Pro disassembles it as one:

```
00641648
00641648 loc_641648:
00641648 xor         ecx, ecx
0064164A test        eax, eax
0064164C setz        cl
0064164F jmp         off_675258[ecx*4]
```

Example of jmp usage in IDA Pro

Using this method, the disassembler displays all the available *JMP* options based on the function's jump list even though there are only two jump options - when the assigned register is either 0 or 1 (in the example above it's the *ECX* register). These methods make the disassembled function very hard to read and investigate:
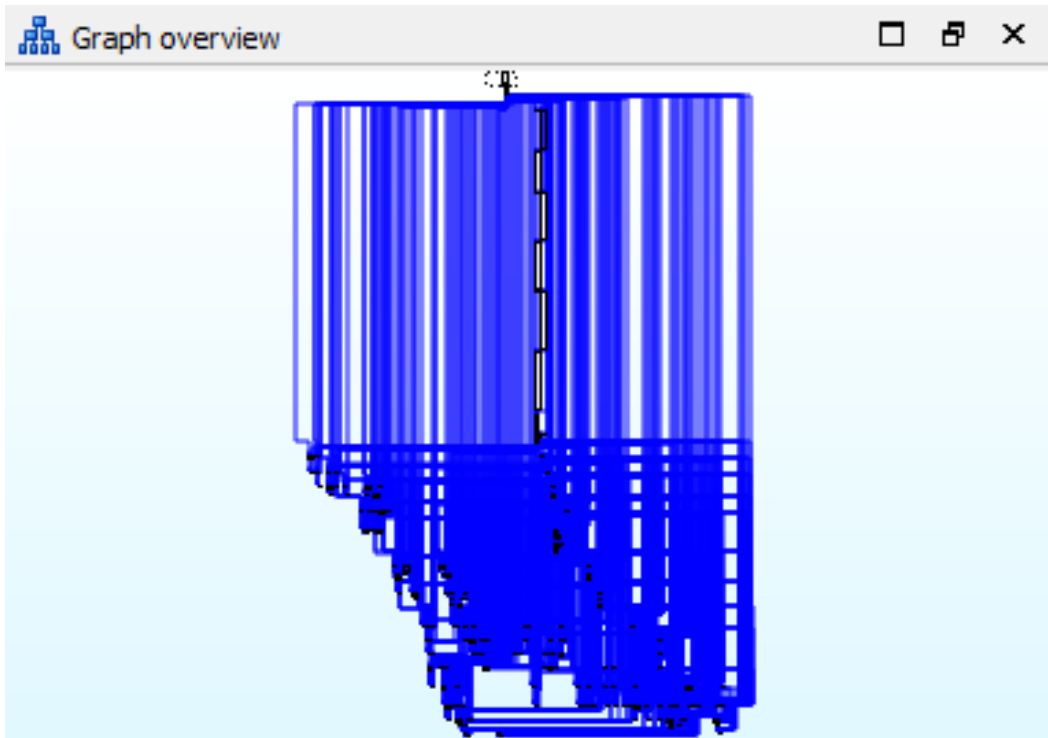
Single function

tree as could be seen in IDA Pro

SPARKLOG: Deploying PRIVATELOG, the Next Link in the Infection Chain
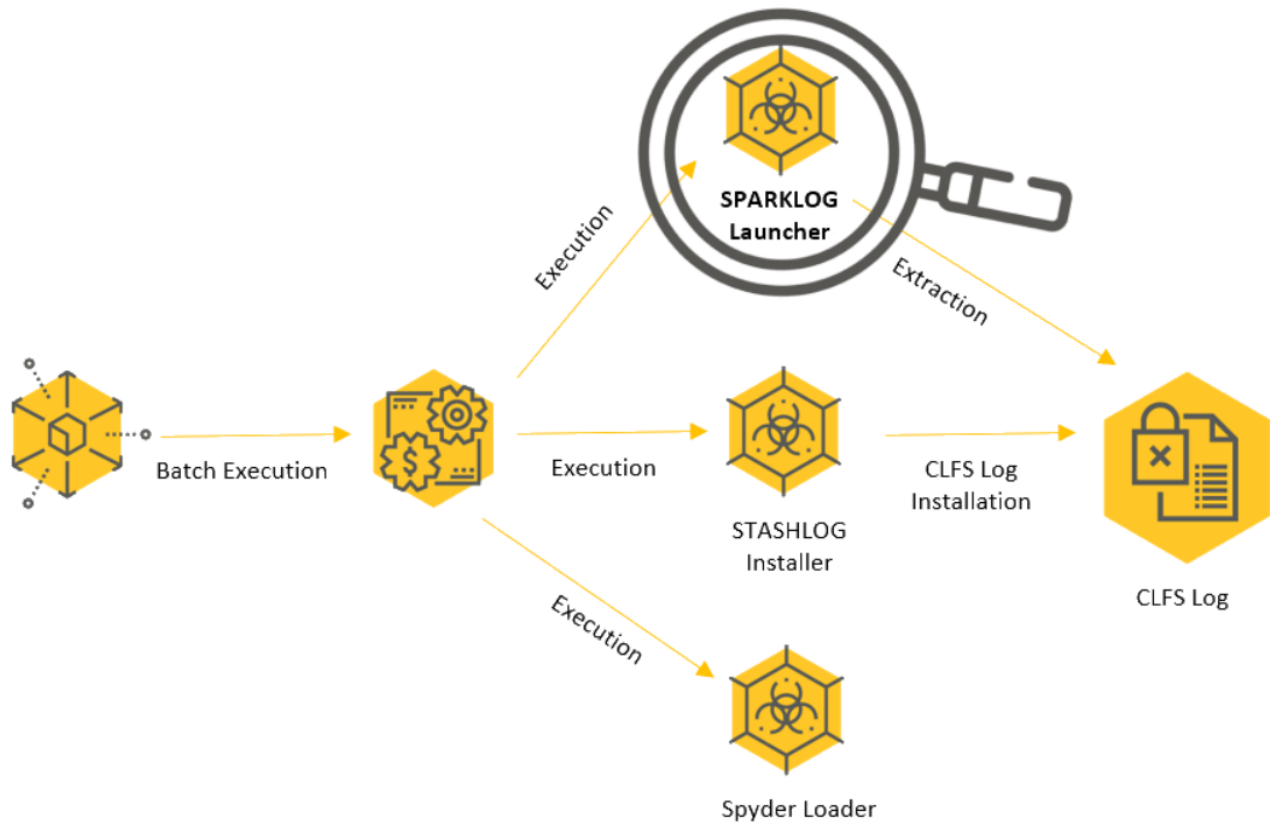
SPARKLOG (spark.exe) is a 32 bit executable written in C++, employed in this attack to extract a DLL from the CLFS file, decrypt it and then launch it for side-loading by Windows services running as SYSTEM. Executing this phase of the attack successfully enables the attackers to

gain Privilege Escalation and also Persistence in a specific case.

SPARKLOG Execution Flow

The execution of SPARKLOG starts by creating a non-visible window followed by a message posted to trigger the execution of the main thread:

```
push    eax ; hMem
call    ds:LocalFree
call    ds:GetCurrentThreadId
mov     idThread, eax
mov     hInstance, esi
push    0 ; lpParam
push    esi ; hInstance
push    0 ; hMenu
push    0 ; hWndParent
push    0 ; nHeight
push    80000000h ; nWidth
push    0 ; Y
push    80000000h ; X
push    0CF0000h ; dwStyle
push    offset WindowName ; lpWindowName
push    offset ClassName ; lpClassName
push    0 ; dwExStyle
call    ds:CreateWindowExA
test    eax, eax
jz      loc_55F82
```

```
push    eax ; hWnd
call    ds:UpdateWindow
push    6Dh ; lpTableName
push    esi ; hInstance
call    ds:LoadAcceleratorsA
mov     esi, eax
push    0 ; int
push    0 ; dwStackSize
push    offset sub_55FA0 ; lpModuleName
call    sub_5A891
add     esp, 0Ch
push    0 ; wMsgFilterMax
push    0 ; wMsgFilterMin
push    0 ; hWnd
push    edi ; lpMsg
call    ds:GetMessageA
test    eax, eax
jz      short loc_55F76
```

Non-visible window creation

The PE then retrieves an encrypted DLL content from the CLFS log file, decrypts it and gets the OS version in use. This OS version will be required later to understand how to deploy the DLL in the compromised machine. Then, it decrypts the strings *Global\HVID_* and *Global\APCI#*. First, it uses the *GetVolumeNameForVolumeMountPointA* API call to get the GUID of the operating system volume and acquires a handle to a *HVID_<OS Volume GUID>* event.

Then, it queries the *MachineGUID* value from the registry
HKLM\SOFTWARE\Microsoft\Cryptography key and creates an event by the name of
Global\APCI#<Machine GUID>. Using these events is a means of communication between the
modules in the attack, and it will be used in further modules as well:

```
C7 85 E0 FC FF FF+mov       [ebp+CDData], 0
00 00 00 00
68 04 01 00 00     push      104h                    ; Size
68 F0 32 30 00     push      offset aGlobalHvid ; "Global\\HVID_"
56                 push      esi                     ; void *
E8 7A 57 00 00     call      _memmove_0
83 C4 0C           add       esp, 0Ch
8D 9D E4 FC FF FF  lea       ebx, [ebp+APCIEvent]
68 08 02 00 00     push      208h                    ; Size
68 F4 33 30 00     push      offset aGlobalApci ; "Global\\APCI#"
53                 push      ebx                     ; void *
E8 61 57 00 00     call      _memmove_0
83 C4 0C           add       esp, 0Ch
56                 push      esi                     ; Str - HVID
E8 98 87 00 00     call      _strlen
83 C4 04           add       esp, 4
B9 04 01 00 00     mov       ecx, 104h
29 C1              sub       ecx, eax
8D 84 05 EC FE FF+lea       eax, [ebp+eax+HVIDEvent]
FF
51                 push      ecx                     ; SizeInBytes
50                 push      eax                     ; Destination
E8 40 FD FF FF     call      GetVolGuid
```
Building the HVID_ and APCI# events

Next, it starts to deploy PRIVATELOG based on the OS version. From Windows Vista to
Windows 7, SPARKLOG uses a popular DLL side loading technique that involves dropping the
DLL with the name *wlbsctrl.dll* to the *%SYSTEM32%\WindowsPowershell\v1.0* directory. It then
stops IKEEXT, a service that was compromised by WINNTI in the past, changes the
configuration using *ChangeServiceConfigA* based on the argument count, then starts it again:

| Name	PID	Description	Status	Group
icssvc		Windows Mobile Hotspot Service	Stopped	LocalServiceN...
IKEEXT		IKE and AuthIP IPsec Keying Modules	Stopped	netsvcs

Stopped IKEEXT service
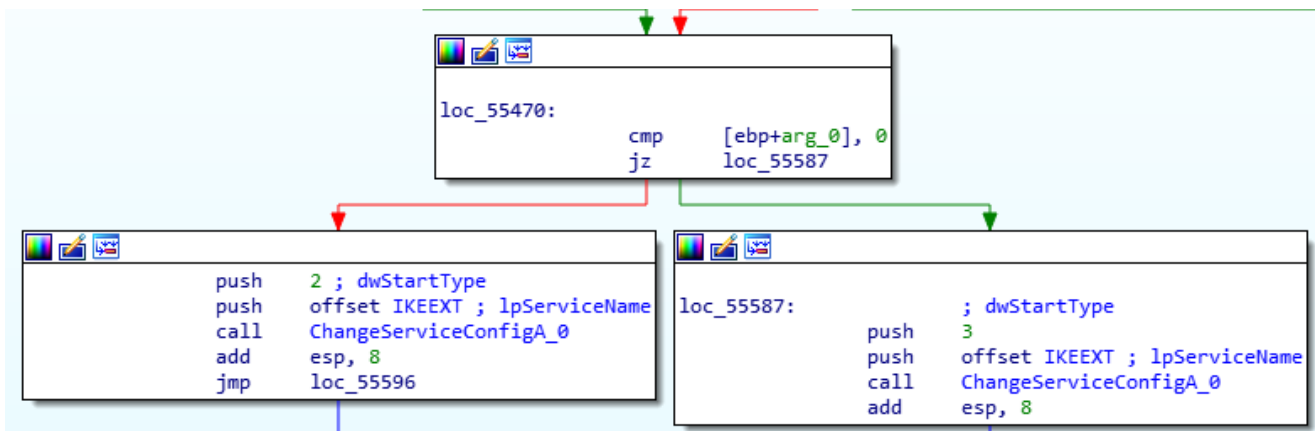
Change IKEEXT service configuration based on argument count

After a successful service start, the service executes and triggers the DLL side-loading vulnerability using SYSTEM privileges. It is interesting to note that in this case, if a command line argument is provided, the DLL will be deleted after the execution starts. This might be due to the fact that abusing *wlbsctrl.dll* is pretty common, and might trigger EDR vendors later on.

From Windows Server 2012 to Windows 10, SPARKLOG acts in a similar fashion, but with a different name and location:

- It drops PRIVATELOG with the name *prntvpt.dll* to the *%SYSTEM32%\spool\drivers\x64\3* directory
- It then stops, changes configuration and starts the PrintNotify service to side-load the DLL (PrintNotify is a legitimate Windows service like IKEEXT, also running as SYSTEM):
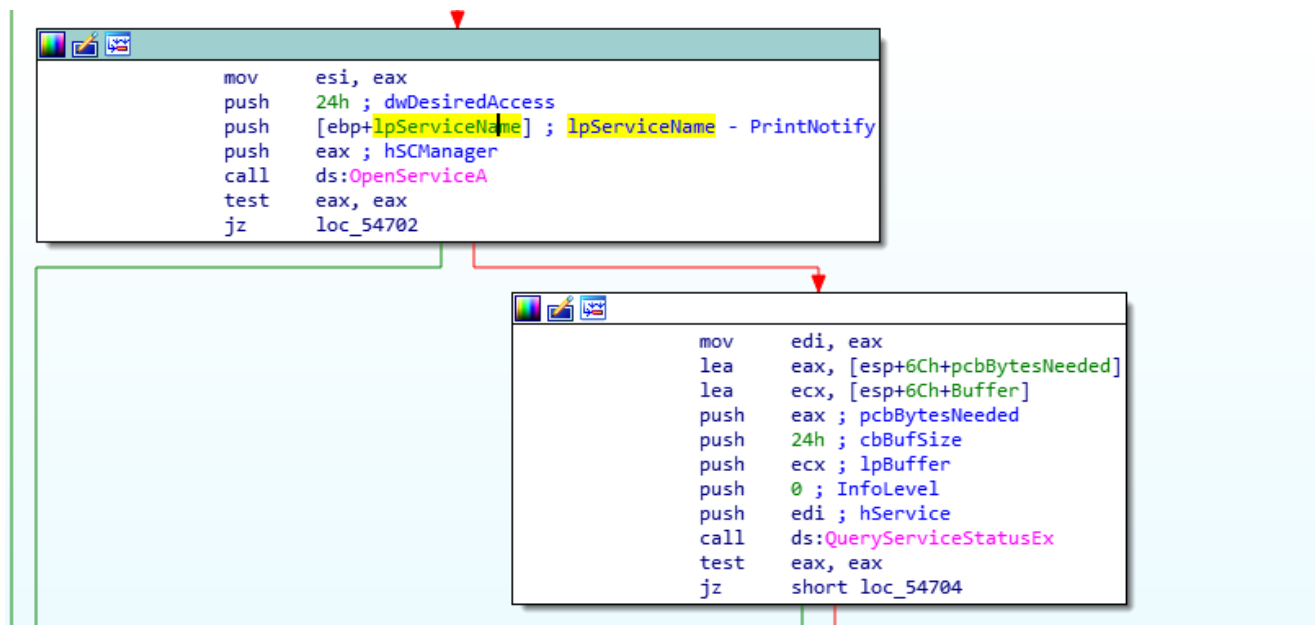
Opening and querying the PrintNotify service

In both cases, the attackers gain stealth by deploying PRIVATELOG while masquerading legitimate file names in privileged locations, as well as gaining persistence and privilege escalation to execute their next step as SYSTEM, the most privileged user in a local machine:
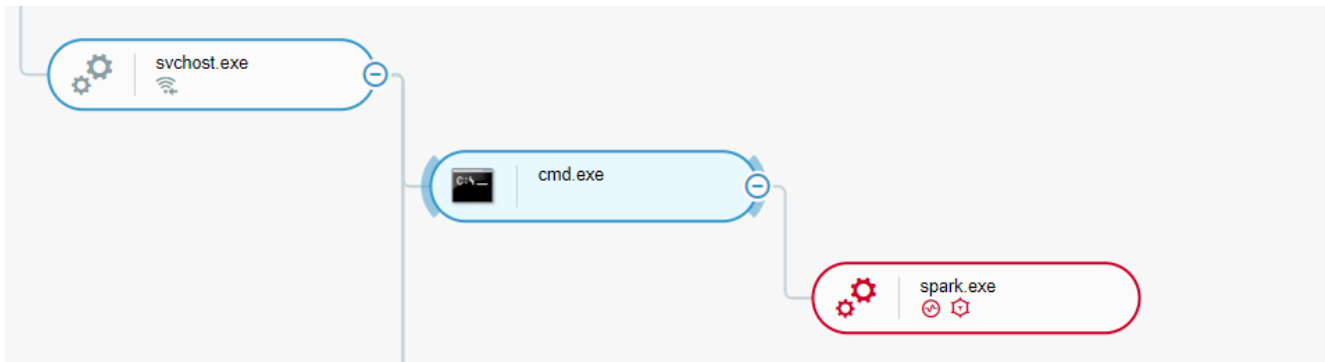
Spark.exe execution as seen in Cybereason's XDR Platform

PRIVATELOG: Extracting and Deploying DEPLOYLOG

PRIVATELOG is a module that exists in 2 forms:

- *Wlbsctrl.dll:* A DLL to be side-loaded by the IKEEXT service, aiming to execute on Windows Vista to Windows 7 operating systems.
- *Prntvpt.dll*: A DLL to be side loaded by the PrintNotify service, aiming to execute on Windows Server 2012 to Windows 10 operating systems.

As both of the DLLs are being loaded by native Windows services, they both live in the context of the svchost process, but differ in their execution flow.

The IKEEXT Hijacker

At the beginning of its execution, *wlbsctrl.dll* is loaded by the IKEEXT service and verifies similarly to *prntvpt.dll* that it's being executed from *svchost* using the right command line (*netsvcs*). After this check, *wlbsctrl.dll* goes straight to dropping DEPLOYLOG.

The PrintNotify Hijacker

At the beginning of its execution, *prntvpt.dll* verifies it is being loaded by the correct process with the right command line (*svchost -k print*), similar to *wlbsctrl.dll*. This command line is the one being executed upon starting the PrintNotify service.

When the PrintNotify service starts, it also loads *PrintConfig.dll*, which is being executed from its *ServiceMain* function. To hijack the execution flow, *prntvpt.dll* loads *PrintConfig.dll* and acquires the address of its *ServiceMain* function. Then, it patches this function, and adds a jump instruction to itself, to continue its execution.

The *prntvpt.dll* component is different, as it is also the persistence tool for the infection, as opposed to the previous samples we discussed which execute only once to infect the machine, this tool runs every time the PrintNotify service is executed. From this point on, the different DLL files act almost the same.

Dropping DEPLOYLOG

PRIVATELOG decrypts DEPLOYLOG in memory from the CLFS log file, then it copies *dbghelp.dll* from its original place in *C:\Windows\System32\dbghelp.dll* to *C:\Windows\System32\WindowsPowerShell\v1.0\dbghelp.dll*. Next, the attackers use a rather unique technique to overwrite the copied *dbghelp.dll* with the aforementioned decrypted buffer using Windows Transactional NTFS (TxF).

Transactional NTFS is a component introduced in Windows Vista that allows developers to create, edit and delete files and directories while giving them the option to roll back in case of errors. This mechanism is used in major operating system components like Windows Update, Task Scheduler and System Restore.

Using Transactional NTFS, the attackers can perform file operations using unconventional methods that can be hard to detect for some security products. They leverage it to create a new malicious *dbghelp.dll* using the following steps:

1.
1. A transaction handle is created for dbghelp.dll:
   - *CreateTransaction*: Creates a new transaction object
   - *CreateFileTransactedA*: with GENERIC_READ_WRITE access.
2. The file is overwritten with the decrypted payload:
   WriteFile: On the transacted file handle.
3. Load the file to a memory section (more about this method is covered <u>here</u>):
   - *NtCreateSection*: with the SEC_IMAGE section attribute.
   - *NtMapViewOfSection*: Mapping the file view in the created section which validates the PE header and splits the section, but doesn't build the import address table and set section permissions.
4. Set the right section permissions and resolve the imports of the DLL:
   - Fixing section permissions using *VirtualProtect* calls.
   - Building the DLL's import address table with *LoadLibrary* and *GetProcAddress* calls.
5. Execute the DEPLOYLOG payload entry point followed by the SvcMain.

DEPLOYLOG: The Winnti Rootkit Deployment and A Usermode Agent
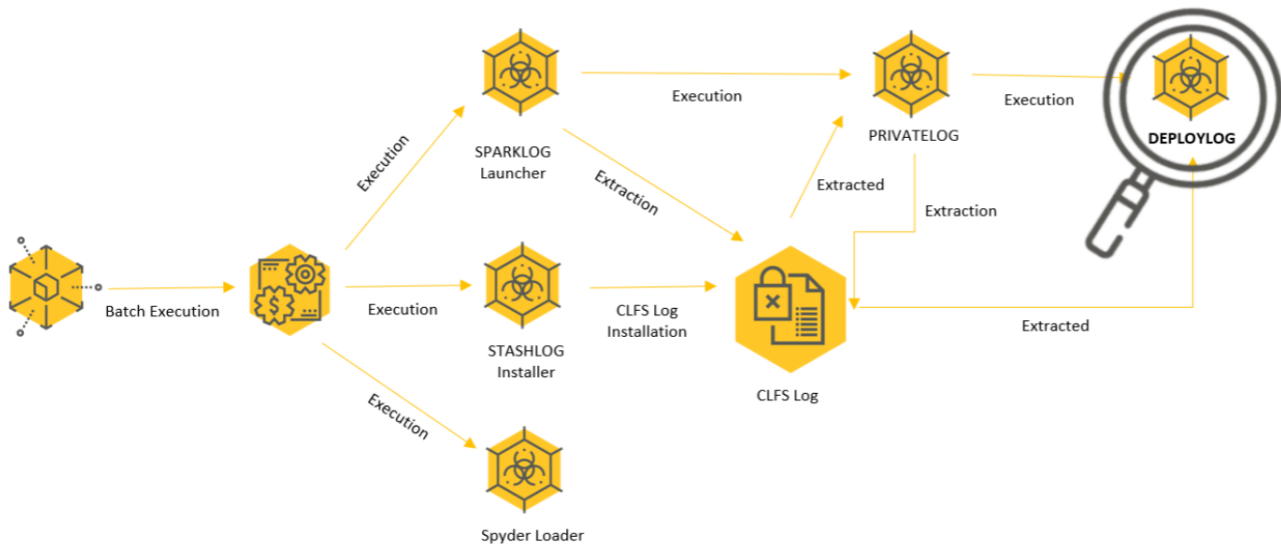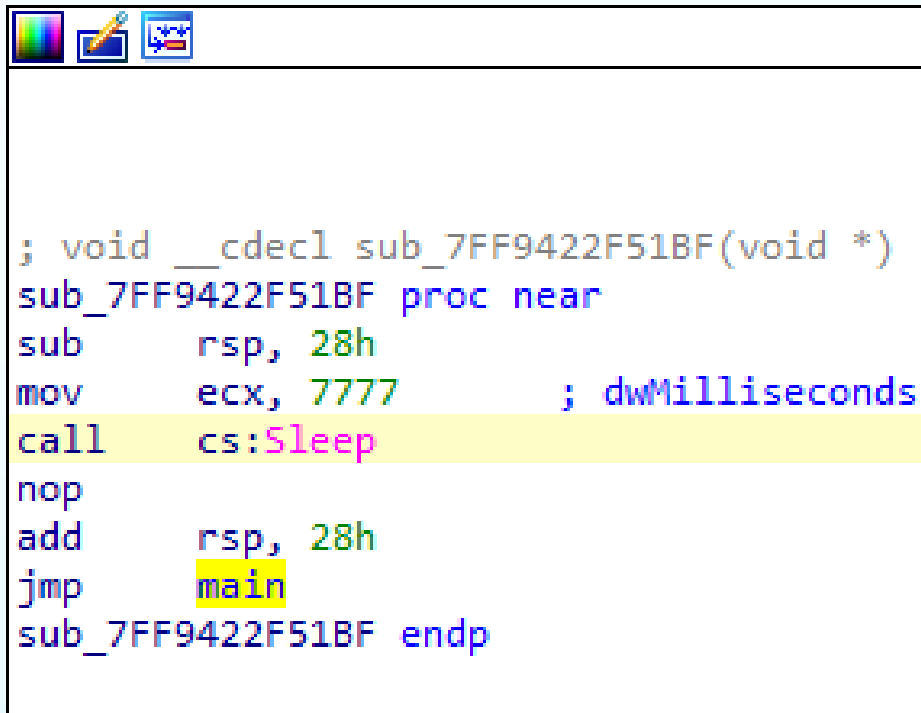
DEPLOYLOG (dbghelp.dll) is a 64 bit DLL, with two purposes:

- The first one is responsible for extracting and executing the attackers' rootkit, dubbed WINNKIT, from the CLFS log file.
- After a successful deployment of the WINNKIT rootkit, DEPLOYLOG switches to its second task, which is communicating both with the remote C2 and the kernel-level rootkit.

It's noteworthy to mention that to evade detection, the attackers deployed DEPLOYLOG as *dbghelp.dll*, a generic, widely used name leveraged to masquerade as a legitimate file at the same location as *PRIVATELOG (C:\Windows\System32\WindowsPowerShell\v1.0)*.

DEPLOYLOG Initialization

Once *DEPLOYLOG* is executed, it starts with a sleep of 7777 milliseconds:

Figure #: Sleep

before the main method

Afterward, it tries to acquire a handle to the earlier *HVID_<OS Volume GUID>* event, and If it doesn't exist it creates it. Then, it initializes the communication channel with the future deployed rootkit:

- First, it tries to acquire a handle to the Beep device object: \\? \GLOBALROOT\Device\Beep.
- If it fails, it tries to do the same for \\?\GLOBALROOT\Device\Null. To test if the rootkit was deployed in the past and is running, DEPLOYLOG tries to send the IOCTL *15E030* to the acquired device handle:

```
   memset(v8, 0, 0x104ui64);
   memset(v7, 0, 0x104ui64);
   device_handle = (__int64)CreateFileA(          // \\?\GLOBALROOT\Device\Beep
                              beep_str,
                              GENERIC_WRITE|GENERIC_READ,
                              FILESHARE_CHANGE_MODIFY,
                              0i64,
                              OPEN_EXISTING,
                              FILE_ATTRIBUTE_DEVICE,
                              0i64);
   if ( device_handle == -1 )
   {
     device_handle = (__int64)CreateFileA(        // \\?\GLOBALROOT\Device\Null
                              null_str,
                              GENERIC_WRITE|GENERIC_READ,
                              3u,
                              0i64,
                              OPEN_EXISTING,
                              FILE_ATTRIBUTE_DEVICE,
                              0i64);
     if ( device_handle == -1 )
     {
       device_handle = -1i64;
LABEL_8:
       v3 = GetLastError();
       goto Error;
     }
   }
   v7[0] = 16;
   v6[0] = 0;
   if ( !DeviceIoControl((HANDLE)device_handle, 0x15E030u, v7, 8u, v8, 0x104u, v6, 0i64) )
     goto LABEL_8;
Error:
   CloseHandle((HANDLE)device_handle);
   return v3;
```
DEPLOYLOG gets handles to device objects and sends an IOCTL

WINNKIT Deployment

Upon succeeding the earlier steps, DEPLOYLOG extracts the rootkit from the CLFS log file, decrypts its content, then stops the *amdk8* service. This service is the AMD K8 processor kernel driver service. Aiming for this specific service can tell something about the Winnti modus operandi, indicating they aim only for AMD-related machines to be infected, which could also indicate having a prior knowledge of the victim's infrastructure.

Then, DEPLOYLOG decrypts the string *SystemRoot\System32\drivers\bqDsp.sys* and changes the *amdk8* service configuration to point to this path:

```
49 8B 4C 24 10      mov    rcx, [r12+10h]  ; String1 - C:\System32\drivers\amdk8.sys
48 89 FA            mov    rdx, rdi        ; String2 - C:\System32\drivers\bqDsp.sys
E8 A5 C8 04 00      call   _stricmp
31 DB               xor    ebx, ebx
85 C0               test   eax, eax
74 58               jz     short loc_7FF94C704D91
```

```
49 8B 74 24 10         mov    rsi, [r12+10h]
48 89 F1               mov    rcx, rsi        ; Str
E8 1A 8D 04 00         call   strlen
4C 89 F9               mov    rcx, r15        ; void *
48 89 F2               mov    rdx, rsi        ; Src
49 89 C0               mov    r8, rax         ; Size
E8 40 D6 FF FF         call   _memove
0F 57 C0               xorps  xmm0, xmm0
0F 11 44 24 48         movups xmmword ptr [rsp+0A8h+lpPassword], xmm0 ; lpPassword
0F 11 44 24 38         movups xmmword ptr [rsp+0A8h+lpDependencies], xmm0 ; lpDependencies
0F 11 44 24 28         movups xmmword ptr [rsp+0A8h+lpLoadOrderGroup], xmm0 ; lpLoadOrderGroup
48 89 7C 24 20         mov    [rsp+0A8h+lpBinaryPathName], rdi ; lpBinaryPathName - C:\System32\drivers\bqDsp.sys
4C 89 F1               mov    rcx, r14        ; hService
BA FF FF FF FF         mov    edx, 0FFFFFFFFh ; dwServiceType
41 B8 FF FF FF FF mov    r8d, 0FFFFFFFFh ; dwStartType
41 B9 FF FF FF FF mov    r9d, 0FFFFFFFFh ; dwErrorControl
FF 15 7B A2 05 00 call   cs:ChangeServiceConfigA
85 C0                  test   eax, eax
75 08                  jnz    short loc_7FF94C704D91
```

Changing amdk8 service configuration to execute WINNKIT

Next, DEPLOYLOG writes the *bqDsp.sys* rootkit driver from the CLFS log file to the
C:\WINDOWS\system32\drivers directory and starts the service again, this time to execute its
malicious payload. By doing so, DEPLOYLOG finishes deploying the rootkit. To cover its tracks,
DEPLOYLOG will then stop the service, restore its previous configuration to point to the
amdk8.sys driver, and finally delete WINNKIT:

*Sending WINNKIT the IOCTL 0x8000C004 using Pipe2PortCtrl*

DEPLOYLOG as a User-Mode Agent

Once deployed, DEPLOYLOG turns to its second mission, effectively acting as a user-mode agent whose purpose is to bridge the newly deployed rootkit and the remote C2. The DLL will start communicating with the C2 servers which will send data that will be intercepted by the driver, as will be explained in the next section discussing WINNKIT. Using this agent, the attackers will load new modules, pop a CMD shell, drop MFSDLL.exe for credential grabbing and more:

DEPLOYLOG loaded by svchost.exe as seen in the Cybereason XDR Platform

Behold WINNKIT, The Evasive Winnti Rootkit
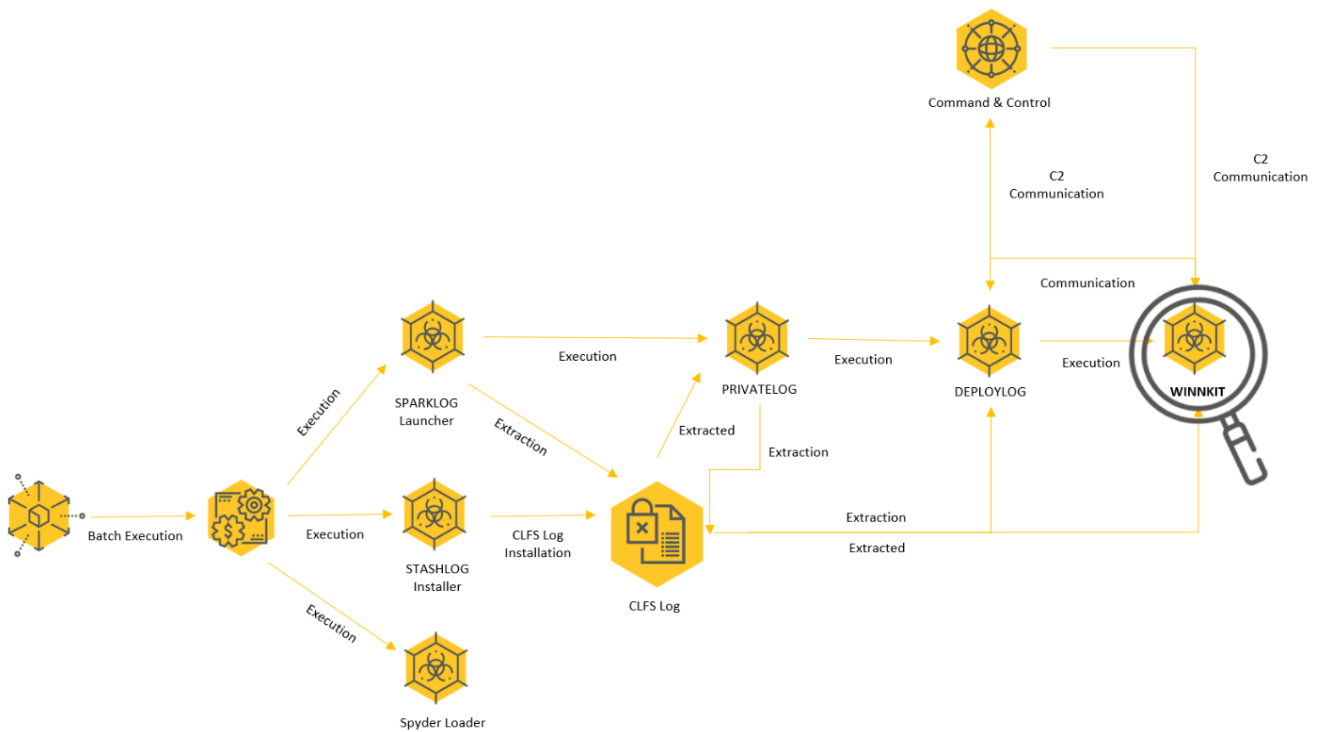
The final payload deployed by Winnti is also the most evasive and sophisticated: a driver acting as a rootkit, dubbed WINNKIT. WINNKIT's previous version was underlined researched in the past, and its purpose is to act as a kernel-mode agent, interacting with the user-mode agent and intercepting TCP/IP requests, by talking directly to the network card. The almost zero detection rate in VirusTotal, together with the compilation timestamp from 2019, illustrates just how evasive this rootkit really is, staying in the shadows for 3 years:
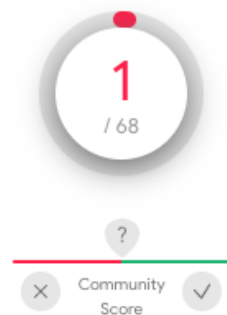
Low detection rate in VirusTotal

compiler-stamp	0x5CD0FF47 (Tue May 07 06:45:11 2019)
debugger-stamp	n/a
resources-stamp	
exports-stamp	n/a
version-stamp	empty
certificate-stamp	0xB7EBC000 (Wed Mar 28 02:00:00 2012)

The rootkit's compilation timestamp

WINNKIT contains an expired BenQ digital signature, which is leveraged to bypass the Driver Signature Enforcement (DSE) mechanism that requires drivers to be properly signed with digital signatures in order to be loaded successfully. This mechanism was first introduced in Windows Vista 64-bit, and is affected for all versions of Windows since then:

Signature Info ⓘ

Signature Verification

⚠ File signature could not be verified

File Version Information

Revoked rootkit certificate and file version

Copyright	Copyright (C) 2019
Product	Monitor Display Driver
Description	Monitor Display Driver
Original Name	BqDisplay.sys
Internal Name	BqDisplay.sys
File Version	2.1.75.491

information as seen in VirusTotal

After successfully loading, WINNKIT hooks network communication, and operates based on custom commands that are being sent from the aforementioned user-mode agent, DEPLOYLOG.

At the beginning of its execution, the driver validates the NDIS version, making sure the system is Windows Vista or above. By using the NDIS API, it communicates directly with the network card, skipping higher level communication protocols:

```
*(_QWORD *)&Parameters.PoolTag = L"IPSecMiniPort";
*(_DWORD *)&String2.Length = 0xC000A;
String2.Buffer = L"TCPIP";
ProtocolCharacteristics.Header.Type = 0;
memset(&ProtocolCharacteristics.Header.Revision, 0, 0x77ui64);
if ( !dword_14000A52C )
{
  v3 = NdisGetVersion();
  dword_14000A520 = v3;
  if ( v3 != NDIS_RUNTIME_VERSION_60 )
  {
    if ( v3 == NDIS_RUNTIME_VERSION_61 )
    {
      dword_14000A524 = 0x3C0;
      dword_14000A52C = 0x180;
      dword_14000A528 = 0x398;
      goto LABEL_12;
```

WINNKIT's communication with the network card

After establishing a connection with the network card, the rootkit tries to open the event
BaseNamedObjects\\{*75F09225-CD50-460B-BF90-5743B8404D73*}. In case it fails, it creates
this event, and then hooks the *Device**Null* device. Hooking this device is somehow risky, as
this device is often being targeted by modern rootkit, thus making it relatively exposed to
detection. Nevertheless, it enabled the authors to stay undetected for years.

Using the above mentioned mechanisms enables WINNKIT a mean of communication with the
user mode agent:

```
ObjectName.Length = 0;
*(_QWORD *)&ObjectName.MaximumLength = 0i64;
*(_DWORD *)((char *)&ObjectName.Buffer + 2) = 0;
HIWORD(ObjectName.Buffer) = 0;
if ( L"\\BaseNamedObjects\\{75F09225-CD50-460B-BF90-5743B8404D73}" )
{
  Handle = 0i64;
  flag = 0;
  v6 = 0;
  memset(Dst, 0, sizeof(Dst));
  event_guid.Length = 0;
  *(_QWORD *)&event_guid.MaximumLength = 0i64;
  *(_DWORD *)((char *)&event_guid.Buffer + 2) = 0;
  HIWORD(event_guid.Buffer) = 0;
  RtlInitUnicodeString(&event_guid, L"\\BaseNamedObjects\\{75F09225-CD50-460B-BF90-5743B8404D73}");
  v6 = 48;
  Dst[0] = 0i64;
  Dst[1] = (__int64)&event_guid;
  LODWORD(Dst[2]) = 704;
  Dst[3] = 0i64;
  Dst[4] = 0i64;
  named_event = ZwOpenEvent(&Handle, 0x80000000i64, &v6);
  if ( named_event < 0 )
  {
    if ( IoCreateSynchronizationEvent(&event_guid, &Handle) )
    {
      named_event = 0;
    }
    else
    {
      Handle = 0i64;
      named_event = STATUS_UNSUCCESSFUL;
    }
  }
  else
  {
    flag = 1;
    ZwClose(Handle);
    Handle = 0i64;
  }
```

WINNKIT event creation

```
RtlInitUnicodeString(&ObjectName, L"\\Device\\Null");
v0 = IoGetDeviceObjectPointer(&ObjectName, 1u, &Object, &DeviceObject);
if ( v0 >= 0 )
{
  v2 = DeviceObject->DriverObject;
  if ( v2 )
  {
    qword_14000A590 = (__int64 (*)(void))v2->MajorFunction[0xE];
    v2->MajorFunction[14] = (PDRIVER_DISPATCH)sub_1400027A4;
  }
  else
  {
    ZwClose(Handle);
    Handle = 0i64;
    ObfDereferenceObject(Object);
    DeviceObject = 0i64;
    Object = 0i64;
    v0 = -1073741772;
  }
}
```

WINNKIT hooks the \\Device\\Null device object

```
if ( v9 + 8 <= v7 )
{
  switch ( *(_DWORD *)Src )
  {
    case 0x100:
      v14 = (unsigned int *)ExAllocatePool(NonPagedPool, v7);
      v15 = v14;
      if ( v14 )
      {
        memmove(v14, Src, v10);
        v4 = sub_1400011CC(v15 + 2, v15[1], a3, a4);
        ExFreePoolWithTag(v15, 0);
      }
      break;
    case 0x200:
      if ( a3 && *a4 >= 4u )
      {
        v4 = 1;
        *a3 = 419504919;
        *a4 = 4;
      }
      break;
    case 0x300:
      if ( a3 && *a4 >= 0x5EAu )
```

The switch case that handling different commands

A summary of the communication flow of DEPLOYLOG and WINNKIT, can be seen in the following diagram:

Rootkit high level operation diagram

Below are the functionality we believe that each code represents, according to our findings and previous conducted research:

Command	Operation
0x100	Hide driver
0x200	Determine version
0x300	Access IRQL shared data
0x400	Map and allocate buffer

0x500	Map a buffer
0x800	Clean up

Winnti Auxiliary Plugins

Winnti used reflective loading injection in order to evade detection. The malicious modules are reflectively injected into the legitimate svchost processes. The following modules were detected by Cybereason and seem consistent with previously reported Winnti plugins:

- **Cmp2.o:** The plugin's purpose is to provide access to the system command line and appears to be a variant of the Winnti "CmdPlus" plugin.
- **Fmg2.o:** This plugin is responsible for listing and modifying files on the targeted machine and appears to be the Winnti "ListFileManager" module.
- **Srv2.o:** The purpose of the plugin is to display information about system services and is assessed to be the Winnti "ListService" plugin.
- **Sck2.o:** The purpose of the plugin is to transfer data over the network using a SOCKS5 proxy server and is assessed to be the "Socks5Client" plugin.
- **Prc2.o:** This plugin can list or kill running processes on the targeted machine.
- **Trs2.o:** This plugin was also used for data transfer via Socks5 proxy.
- **Cme2.o:** The purpose of this plugin is to enable Remote Desktop access to Winnti:

Example: svchost process that loaded sck2.o and trs2.o modules reflectively, as seen in the Cybereason XDR Platform

Conclusions

In part two of the research, we provided a deep dive into the Winnti malware arsenal that was observed by the Cybereason IR and Nocturnus teams. Our analysis provides a unique and holistic view of Winnti operational aspects, capabilities and modus operandi. While some of the tools mentioned in the research were previously reported on, some tools such as DEPLOYLOG were previously undocumented and first analyzed in this report. In addition, our analysis provides further insights regarding some of the known Winnti tools.

Perhaps one of the most interesting things to notice is the elaborate and multi-phased infection chain Winnti employed. The malware authors chose to break the infection chain into multiple interdependent phases, where each phase relies on the previous one in order to execute correctly. This demonstrates the thought and effort that was put into both the malware and operational security considerations, making it almost impossible to analyze unless all pieces of the puzzle are assembled in the correct order.

Furthermore, the rare abuse of the Windows' own CLFS logging system and NTFS manipulations provided the attackers with extra stealth and the ability to remain undetected for years.

We hope that this report helps to shed light on Winnti operations, tools and techniques, and that it will assist to expose further intrusions.

Acknowledgments

Indicators of Compromise

LOOKING FOR THE IOCs? CLICK ON THE CHATBOT DISPLAYED IN LOWER-RIGHT OF YOUR SCREEN FOR ACCESS. Due to the sensitive nature of the attack, not all IOCs observed by Cybereason can be shared in our public report. Please contact us for more information.

MITRE ATT&CK BREAKDOWN

Reconnaissance	Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion

Gather Victim Identity Information: Credentials	Exploit Public-Facing Application	Scheduled Task/Job	Server Software Component: Web Shell	Create or Modify System Process: Windows Service	Hijack Execution Flow: DLL Side-Loading
Gather Victim Network Information	Supply Chain Compromise	Inter-process communication		Hijack Execution Flow: DLL Side-Loading	Rootkit
		Exploitation for Client Execution		Process Injection: Dynamic-link Library Injection	Masquerading: Match Legitimate Name or Location
		Command and Scripting Interpreter: Windows Command Shell	Scheduled Task/Job: Scheduled Task	Scheduled Task/Job: Scheduled Task	Process Injection: Dynamic-link Library Injection
		Command and Scripting Interpreter: Visual Basic	Valid Accounts: Domain Accounts	Valid Accounts: Domain Accounts	Reflective Code Loading
		Native API	Valid Accounts: Local Accounts	Valid Accounts: Local Accounts	Signed Binary Proxy Execution: Rundll32
					Valid Accounts: Domain Accounts
					Valid Accounts: Local Accounts

Credential Access	Discovery	Lateral movement	Collection	Exfiltration	Command and Control
OS Credential Dumping	System Network Configuration Discovery	Exploitation of Remote Services	Archive Collected Data: Archive via Utility	Automated Exfiltration	Application Layer Protocol: Web Protocols
	Remote System Discovery	Remote Services: Remote Desktop Protocol	Automated Collection		Proxy
	Password Policy Discovery				
	Permission Groups Discovery				
	Network Share Discovery				
	System Service Discovery				
	System Time Discovery				
	System Network Connections Discovery				
	Account Discovery				

[System
Owner/User
Discovery](#)

[System
Information
Discovery](#)

[Process
Discovery](#)

About the Researchers:

Chen Erlich

Chen has almost a decade of experience in Threat Intelligence & Research, Incident Response and Threat Hunting. Before joining Cybereason, Chen spent three years dissecting APTs, investigating underground cybercriminal groups and discovering security vulnerabilities in known vendors. Previously, he served as a Security Researcher in the IDF.

Fusao Tanida

Fusao spent over 10 years in the security industry. Before joining, he worked as a mobile malware researcher and a developer at the security vendor and then worked at the global mobile phone manufacturer for the development of AntiVirus, VPN client on their Android mobile phone.

Fusao joined Cybereason in 2019 and was previously the Senior Security Analyst at the Advanced Services Team in Cybereason Japan where delivered various security professional services, Incident Response, consultation and triage malware activity alerts in SOC.

Ofir Ozer

Ofir is a Incident Response Engineer at Cybereason who has a keen interest in Windows Internals, reverse engineering, memory analysis and network anomalies. He has years of experience in Cyber Security, focusing on Malware Research, Incident Response and Threat Hunting. Ofir started his career as a Security Researcher in the IDF and then became a malware researcher focusing on Banking Trojans.

Akihiro Tomita

Akihiro is the Senior Manager of Global Security Practice, leading Incident Response team in the APAC region and Japan. Akihiro has led a substantial number of large-scale Incident Response, Digital Forensics and Compromise Assessment engagements during recent years. Akihiro was also a former Team lead of Advanced Security Services team responsible for managing, developing, delivering a variety of professional services including Proactive threat hunting, Security Posture Assessment, Advanced security training and consulting services at Cybereason.

Niv Yona

Niv, IR Practice Director, leads Cybereason's incident response practice in the EMEA region. Niv began his career a decade ago in the Israeli Air Force as a team leader in the security operations center, where he specialized in incident response, forensics, and malware analysis. In former roles at Cybereason, he focused on threat research that directly enhances product detections and the Cybereason threat hunting playbook, as well as the development of new strategic services and offerings.

Daniel Frank

With a decade in malware research, Daniel uses his expertise with malware analysis and reverse engineering to understand APT activity and commodity cybercrime attackers. Daniel has previously shared research at RSA Conference, the Microsoft Digital Crimes Consortium, and Rootcon.

ASSAF DAHAN, HEAD OF THREAT RESEARCH

Assaf has over 15 years in the InfoSec industry. He started his career in the Israeli Military 8200 Cybersecurity unit where he developed extensive experience in offensive security. Later in his career he led Red Teams, developed penetration testing methodologies, and specialized in malware analysis and reverse engineering.

Operation CuckooBees Indicators of Compromise (IOCs)

Hashes

- BB93AE0FEE817FE56C31BDC997F3F7D57A48C187 - STASHLOG
- 4D1B8791D0715FE316B43FC95BDC335CB31A82CA - STASHLOG
- 2D336978AF261E07B1ECFAF65DC903B239E287A4 - STASHLOG
- F2D04FE529E2D8DAB96242305255CFB84CE81E9C - STASHLOG
- F8D46895E738254238473D650D99BDC92C34EE44 - SPARKLOG
- 9267FE0BB6D367FC9186E89EA65B13BAA7418D87 - PRIVATELOG
- A009A0F5A385683AEA74299CBE6D5429C609F2D2 - PRIVATELOG
- 1316F715D228AE6CC1FBA913C6CC309861F82E14 - PRIVATELOG
- 1275894D8231FE25DB56598DDCF869F88DF5AD8D - WINNKIT
- 9139C89B2B625E2CEEE2CBF72AEF6C5104707A26 - WINNKIT
- 082DBCA2C3CA5C5410DE9951A5C681F0C42235C8 - WINNKIT

File Names & Paths

- C:\Windows\temp\bc.bat
- C:\Windows\AppPatch\Custom\Custom64\cc.bat
- C:\Windows\temp\cc.log
- C:\Windows\AppPatch\Custom\Custom64\log.dat
- C:\Windows\Branding\Basebrd\x64.tlb
- C:\Windows\Branding\Basebrd\language.dll
- C:\Windows\System32\mscuplt.dll
- C:\Windows\System32\rpcutl.dll
- C:\Windows\System32\dot3utl.dll
- C:\Windows\System32\iumatl.dll
- C:\Windows\System32\Nlsutl.dll
- C:\Windows\System32\WindowsPowerShell\v1.0\dbghelp.dll
- C:\Windows\System32\drivers\bqDsp.sys
- C:\Windows\apppatch\en-us\MFSDLL.exe
- C:\Windows\System32\spool\drivers\x64\3\prntvpt.dll
- C:\Windows\System32\WindowsPowerShell\v1.0\wlbsctrl.dll
- C:\Windows\assembly\gac_msil\dfsvc\foserv.exe
- C:\Windows\assembly\temp\foserv.exe
- C:\Windows\apppatch\custom\custom64\shiver.exe
- C:\Windows\apppatch\custom\custom64\spark.exe
- mktzx64.dll

Winnti Malware Modules Names

- Cmp2.o
- Fmg2.o
- Srv2.o

- Sck2.o
- Prc2.o
- Trs2.o
- Cme2.o

Events

- \BaseNamedObjects\{75F09225-CD50-460B-BF90-5743B8404D73}
- \BaseNamedObjects\{7D0DF5FC-3991-4047-921F-32308B1A0459}
- \BaseNamedObjects\{B73AB0F4-A1D0-4406-9066-41E00BA78E9F}
- Global\APCI#<GUID>
- Global\HVID_<GUID>

Named Pipes

Pipe2PortCtrl

Scheduled Task Name

test

About the Author

Cybereason Nocturnus

The Cybereason Nocturnus Team has brought the world's brightest minds from the military, government intelligence, and enterprise security to uncover emerging threats across the globe. They specialize in analyzing new attack methodologies, reverse-engineering malware, and exposing unknown system vulnerabilities. The Cybereason Nocturnus Team was the first to release a vaccination for the 2017 NotPetya and Bad Rabbit cyberattacks.

All Posts by Cybereason Nocturnus