# Too Log; Didn't Read — Unknown Actor Using CLFS Log Files for Stealth | Mandiant

By Mandiant

Published: 2021-09-01 · Archived: 2026-04-05 23:38:11 UTC

Written by: Adrien Bataille, Blaine Stancill

---

The Mandiant Advanced Practices team recently discovered a new malware family we have named PRIVATELOG and its installer, STASHLOG. In this post, we will share a novel and especially interesting technique the samples use to hide data, along with detailed analysis of both files that was performed with the support of FLARE analysts. We will also share sample detection rules, and hunting recommendations to find similar activity in your environment.

Mandiant has yet to observe PRIVATELOG or STASHLOG in any customer environments or to recover any second-stage payloads launched by PRIVATELOG. This may indicate malware that is still in development, the work of a researcher, or targeted activity.

### CLFS and Transaction Files

PRIVATELOG and STASHLOG rely on the Common Log File System (CLFS) to hide a second stage payload in registry transaction files.

CLFS is a log framework that was introduced by Microsoft in Windows Vista and Windows Server 2003 R2 for high performance. It provides applications with API functions—available in clfsw32.dll—to create, store and read log data.

Because the file format is not widely used or documented, there are no available tools that can parse CLFS log files. This provides attackers with an opportunity to hide their data as log records in a convenient way, because these are accessible through API functions. This is similar in nature to malware which may rely, for example, on the Windows Registry or NTFS Extended Attributes to hide their data, which also provide locations to store and retrieve binary data with the Windows API.

In Microsoft Windows, CLFS is notably used by the Kernel Transaction Manager (KTM) for both Transactional NTFS (TxF) and Transactional Registry (TxR) operations. These allow applications to perform a number of changes on the filesystem or registry, all grouped in a single transaction that can be committed or rolled back. For example, to open a registry key in a transaction, the functions *RegCreateKeyTransacted()*, *RegOpenKeyTransacted()*, and *RegDeleteKeyTransacted()* are available.

Registry transactions are stored in dedicated files with the following naming scheme: .TMContainer.regtrans-ms or .TxR..regtrans-ms. These are CLFS containers that are referenced in a master .blf file that only contains metadata and can be found in various locations including user profile directories.

Registry transaction forensics were briefly explored in a previous blog post. The CLFS master and container file formats are mostly undocumented; however, previous research is available on GitHub.

### Malware Obfuscation

As with many malware families, most of the strings used by PRIVATELOG and STASHLOG are obfuscated. Yet the technique observed here is uncommon and relies on XOR’ing each byte with a hard-coded byte inline, with no loops. Effectively, each string is therefore encrypted with a unique byte stream.

```
34 28                  xor     al, 28h
88 05 32 CA 01 00      mov     cs:g_obfuscated_PrintNotify, al
8A 05 21 CA 01 00      mov     al, cs:byte_18002255D
34 29                  xor     al, 29h
88 05 25 CA 01 00      mov     cs:byte_180022569, al
8A 05 14 CA 01 00      mov     al, cs:byte_18002255E
34 C1                  xor     al, 0C1h
88 05 18 CA 01 00      mov     cs:byte_18002256A, al
8A 05 07 CA 01 00      mov     al, cs:byte_18002255F
34 78                  xor     al, 78h
88 05 0B CA 01 00      mov     cs:byte_18002256B, al
8A 05 FA C9 01 00      mov     al, cs:byte_180022560
34 9B                  xor     al, 9Bh
88 05 FE C9 01 00      mov     cs:byte_18002256C, al
8A 05 ED C9 01 00      mov     al, cs:byte_180022561
34 49                  xor     al, 49h
88 05 F1 C9 01 00      mov     cs:byte_18002256D, al
8A 05 E0 C9 01 00      mov     al, cs:byte_180022562
34 2E                  xor     al, 2Eh
88 05 E4 C9 01 00      mov     cs:byte_18002256E, al
8A 05 D3 C9 01 00      mov     al, cs:byte_180022563
34 05                  xor     al, 5
88 05 D7 C9 01 00      mov     cs:byte_18002256F, al
8A 05 C6 C9 01 00      mov     al, cs:byte_180022564
34 57                  xor     al, 57h
```

Figure 1: Sample string deobfuscation for "PrintNotify"

Interestingly, some of the deobfuscated strings from the installer are used for logging error messages and have spelling errors or typos such as:

- Log index=%d, data border exceed bounday.\n
- Interal data hash mismatch.\n
- Log buffer size=%u too small, expect aleast %u bytes.\n

### Introducing STASHLOG

In addition to containing obfuscated strings, the installer’s code is protected using various control flow obfuscation techniques that make static analysis cumbersome. Figure 2 is a graph overview of the installer’s *main()* function demonstrating the effects of the control flow obfuscation.

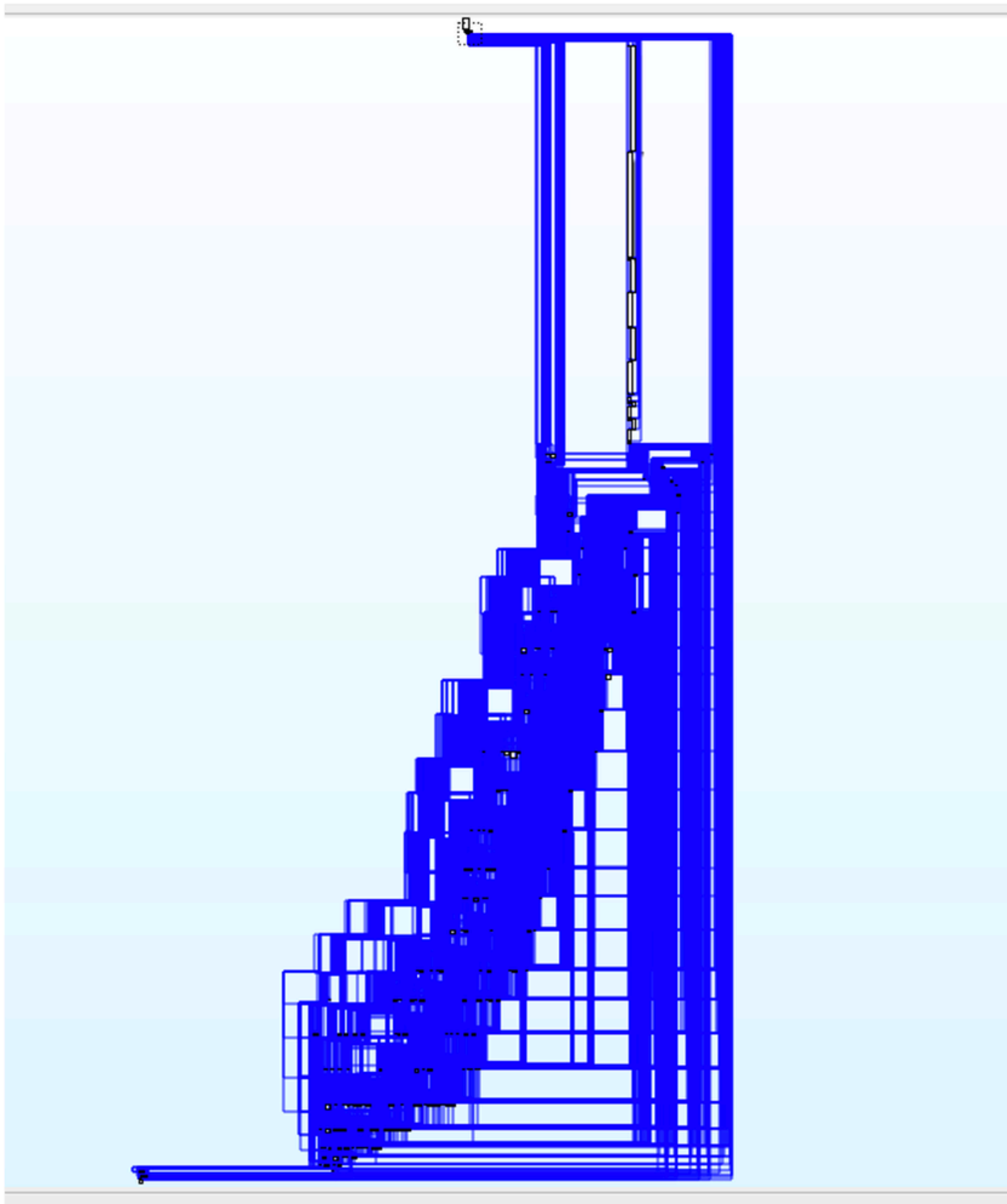

Figure 2: Graph view of main()

STASHLOG has two different modes of operation:

- Without any arguments, during which it will prepare the environment
- With a single argument, which is a file that should be hidden in a CLFS file

**Preparing the Environment**

Executed without arguments, the installer prints two values to the console:

- The GUID returned from the registry value of HKLM\SOFTWARE\Microsoft\Cryptography\MachineGUID

- A 56-byte value derived from a randomly generated GUID with *CoCreateGUID()*

```
C:\Users\Mandiant\Desktop>Shiver.exe
{DA8F616A-7404-44F6-B089-A2CBCADC7C44}
9282EFB70FA2204994D4ADD909CF87AE
71E25DD7F72B5FD894F6EEE6EAD15FED
832B55465DFD9097AECD4C8CD673E843
CE5E44FE30EF84B9
```

Figure 3: Sample console output

The 56-byte value is a concatenation of the random GUID, its SHA1 hash, and the SHA1 hash of the previous values. So: GUID+sha1(GUID)+sha1(GUID+sha1(GUID)).

The randomly generated GUID is stored as a string in the GlobalAtom table, prefixed with win::. This table resides in memory and contains strings with their identifiers available to all applications.

If a string prefixed with win:: already exists when the installer is executed, then the pre-existing GUID in the GlobalAtom table is reused.

Effectively, when executed with no arguments, the installer generates and prints out encryption keys that the actor uses to pre-encrypt the payload before it is written to disk.

#### Stashing the Payload

When launched with an argument, the installer opens and decrypts the contents of the file passed as an argument. It verifies that the file is suffixed by its SHA1 hash, and then generates the same 56-byte value using the stored GlobalAtom GUID string in memory.

The 56-byte value is SHA1 hashed again and the first 16-bytes form the initialization vector (IV), while the key is the 16-byte MachineGUID value from the host's registry. The encryption algorithm is HC-128, which is rarely seen used in malware.

The expected decrypted file contents have a 40-byte header:

```
struct payloadHeader {
  DWORD magic;
  DWORD minWinVer;
  DWORD maxWinVer;
  DWORD totalSize;
  WORD numBlocks;
  WORD unknown;
```

```
   BYTE sha1sum[20];
}
```

In the analyzed installer, the “magic” value is referred to as a *checksum*; however, STASHLOG verifies this value matches the hard-coded value 0x00686365. The number of blocks, specified at offset 16, must be between 2 and 5. The malware also checks that the operating system version is within a lower and upper boundary and that the SHA1 hash of the decrypted data matches the payload header value at offset 20.

Following the payload header, the malware expects blocks of encrypted data with 8-byte headers. Each block header has the following structure:

```
 struct blockHeader {
   DWORD magic;
   DWORD blockSize;
}
```

Once the malware has checked and validated the structure of the payload, it searches for .blf files in the default user’s profile directory and uses the .blf file with the oldest creation date timestamp.

In practice, the malware should typically find the file used for registry transaction logs: C:\Users\Default\NTUSER.DAT.TM.blf

If a matching .blf file is indeed found, it is opened with the *CreateLogFile()* API from clfsw32.dll. This function opens CLFS logs and expects a file name in the following format, without the .blf extension: log:<*LogName*>[::<*LogStreamName*>]

The log file is reset using the *CloseAndResetLogFile()* function and will be opened again to insert the data.

Before inserting data into the CLFS log file, the malware decrypts each block using HC-128. The key is the 16-byte atom GUID and the IV is the first 16-bytes of the atom GUID SHA1 hash. Each block is then re-encrypted with the new key material as follows:

- The encryption key is the 16-byte GUID from *GetVolumeNameForVolumeMountPointW()*.
- The IV is the first 16-bytes of the SHA1 hash of the concatenated GUIDs from:
    - *GetVolumeNameForVolumeMountPointW()*
    - The registry value HKLM\SOFTWARE\Microsoft\Cryptography\MachineGUID

The contents are written to the CLFS log file using the clfsw32.dll API function *ReserveAndAppendLog()*. The payload header is written to the log file as the first entry, followed by separate entries for each block.

The data is effectively stored in the first container file for the registry transaction log: C:\Users\Default\NTUSER.DAT<GUID>.TMContainer00000000000000000001.regtrans-ms.

**Onto PRIVATELOG**

The PRIVATELOG sample recovered by Mandiant is an un-obfuscated 64-bit DLL named prntvpt.dll. It contains exports, which mimic those of legitimate prntvpt.dll files, although the exports have no functionality. PRIVATELOG expects to be loaded from PrintConfig.dll, which is the main DLL of a service named PrintNotify, via DLL search order hijacking.

The malicious code is executed at the DLL's entry point. It starts by verifying the command-line arguments of the process it is running in and expects to be running under svchost.exe -k print. If this matches, the malware resolves the function address for the ServiceMain export function of PrintConfig.dll, which is the service entry point of the service using this command line. This function is patched using Microsoft Detours—a publicly available library used for [instrumenting Win32 functions](#)—so that the execution flow appears to happen in the legitimate service DLL.

The patched ServiceMain function is where PRIVATELOG executes most of its functionality.

Similarly to STASHLOG, PRIVATELOG starts by enumerating *.blf files in the default user's profile directory and uses the .blf file with the oldest creation date timestamp.

If a matching .blf file is found, PRIVATELOG opens it with the clfsw32.dll function *CreateLogFile()*. The log file is then marshalled and parsed using other functions specific to CLFS, such as *CreateLogMarshallingArea()*, *ReadLogFile()* and *ReadNextLogFile()*. The malware expects to find specific entries which match our analysis of the installer.

PRIVATELOG expects the first log entry to have the following format:

- A size greater than 40 (payload header size)
- A WORD value of 2, 3, 4, or 5 at offset 16 (number of blocks)

If the first entry matches the aforementioned criteria, then subsequent records are read until one has an 8-byte header with the following:

- Its first DWORD must equal 2 (assumed magic value)
- Its second DWORD must be less than the entry's size minus the header. This value equals the size of the payload which will be decrypted.

Once the expected log entry is found, its contents are decrypted using the HC-128 encryption algorithm. The decryption key and IV are generated using the same unique host properties that were used by STASHLOG.

It is worth noting that PRIVATELOG only decrypts the first matching block and that at least 2 to 5 blocks are expected to be inserted by STASHLOG.

PRIVATELOG finally uses a rarely seen technique to execute the DLL payload, which this time relies on NTFS transactions. The injection process is similar to [Phantom DLL hollowing](#) and is described as follows:

- Open a transacted handle to a copied file via the API *CreateFileTransactedA()*
  - In the sample analyzed by Mandiant, the file used for the transaction is a copy of the legitimate binary C:\Windows\System32\dbghelp.dll, which is copied to C:\Windows\system32\WindowsPowerShell\v1.0\dbghelp.dll.

- Overwrite the transacted file with the decrypted payload contents
- Create a section backed by the transacted file with SEC_IMAGE attributes via the API *NtCreateSection()*
- Map a view of the newly created section
    - This implicitly loads the transacted file data to some degree. The PE header is validated, and sections mapped into memory; however, it does not fix section permissions or resolve imports.
- Fix the section permissions
- Resolve the imports in the Import Table
- Execute the payload's entry point
- Find and execute the export function named SvcMain

## Hunting for PRIVATELOG

### Container Sample

Figure 4 shows a fabricated container file representing a sample expected log file created by STASHLOG and loaded by PRIVATELOG.

```
00000000  15 00 05 00 01 00 01 00 00 00 00 00 00 00 00 00
00000010  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020  00 00 00 00 FF FF FF FF 70 00 00 00 00 00 00 00
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060  00 00 00 00 00 00 00 00 F8 01 00 00 00 00 00 00
00000070  00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF
00000080  00 00 00 00 FF FF FF FF 50 00 00 00 00 00 00 00
00000090  00 00 28 00 25 00 00 00 65 63 68 00 B1 1D 00 00
000000A0  B1 1D 00 00 D0 0B 01 00 02 00 58 4A 45 8E 30 81
000000B0  17 98 C7 1A E7 26 E7 7C 4E 08 10 2B 31 9C 6C D2
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 64 05
00000200  15 00 05 00 87 00 87 00 00 00 00 00 00 00 00 00
00000210  01 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00
00000220  00 00 00 00 FF FF FF FF 70 00 00 00 00 00 00 00
00000230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000250  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000260  00 00 00 00 00 00 00 00 F0 0C 01 00 00 00 00 00
00000270  00 02 00 00 00 00 00 00 00 00 00 00 FF FF FF FF
00000280  00 00 00 00 FF FF FF FF F8 0B 01 00 00 00 00 00
00000290  00 00 28 00 25 00 00 00 02 00 00 00 C8 0B 01 00
000002A0  65 E7 8A CF ED BF 29 F0 7B 8D AA 23 1F 59 21 5D
000002B0  22 6B 1C 3D 72 42 22 A5 20 6B F3 5A 01 6D 7A ED
000002C0  D9 FD CE C3 BB E7 11 0F 92 42 57 CF 4C 20 46 55
000002D0  F1 85 72 86 4A CA 50 0A 8E 24 2D 83 C4 46 F0 00
000002E0  DE A0 E7 93 B1 98 9C 20 C6 27 BE 22 49 83 4A 5D
000002F0  5B 39 FE BD 7D 31 C0 23 CC 14 5E 9F 46 B9 44 86
```

Payload header

Block header

Encrypted data

Figure 4: Example log file created by STASHLOG

**YARA Rules**

Mandiant created YARA rules to hunt for PRIVATELOG and STASHLOG as well as possible variants based on various methodologies and unique strings that they use. Rules to detect CLFS containers matching PRIVATELOG structures or containing encrypted data are also provided. These rules should be tested thoroughly before they are run in a production environment.

```
 import "math"
import "pe"
rule HUNTING_Win_PRIVATELOG_CLFS {
    meta:
        author = "adrien.bataille@mandiant.com"
        description = "This rule looks for CLFS containers containing possible data used by PRIVATELOG. As this
    condition:
        filesize < 100MB and filesize >= 512KB
        and uint16(0) == 0x0015 // signature
        and uint8(2) != 0 // fixup value upper byte
        and uint8(3) == 0 // always 0
        and uint16(4) == uint16(6) and uint16(4) != 0 // num sectors
        and uint32(8) == 0 // always 0
        and uint32(16) == 1 // always 1
        and uint32(20) == 0 // always 0
        and uint32(40) == 0x70 // size of record header
        // size of data at least 0x28 for first record
        and uint32(0x70+0x18) - 0x28 >= 0x28
        // payloadHeader.numblocks (payloadHeader at 0x70+uint16(0x70+0x22))
        and (uint16(0x70+uint16(0x70+0x22)+0x10) == 0x2 or uint16(0x70+uint16(0x70+0x22)+0x10) == 0x3 or uint16(
        // this is a size, assume it is less than our filesize
        and uint32(0x70+uint16(0x70+0x22)+0xC) < filesize
        // confirm malware using 2 different methods
        and (
            // look for hardcoded magic in first log record
            uint32(0x70+uint16(0x70+0x22)) == 0x00686365 or
            // loop through each possible sector to look for a blockheader struct
            for any i in (0 .. (filesize \ 512) - 1):
            (
            // look for record header, num sectors and size of record
            uint16(i*512)==0x0015 and uint16(i*512+4) == uint16(i*512+6) and uint16(i*512+4) != 0 and uint32(i*5
            // look for magic and blockheader.blocksize in payload
            uint32(i*512+0x70+uint16(i*512+0x70+0x22)) == 2 and uint32(i*512+0x70+uint16(i*512+0x70+0x22)+4) ==
            )
        )
}
```

```
rule HUNTING_Win_CLFS_Entropy {
    meta:
        author = "adrien.bataille@mandiant.com"
        description = "This rule looks for CLFS containers with records containing high entropy. As this rule ma
    condition:
        filesize < 100MB and filesize >= 512KB
        and uint16(0) == 0x0015 // signature
        and uint8(2) != 0 // fixup value upper byte
        and uint8(3) == 0 // always 0
        and uint16(4) == uint16(6) and uint16(4) != 0 // num sectors
        and uint32(8) == 0 // always 0
        and uint32(16) == 1 // always 1
        and uint32(20) == 0 // always 0
        and uint32(40) == 0x70 // size of record header
        and for any i in (0 .. (filesize \ 512) - 1) :
        (
        // look for record header, num sectors and size of record
        uint16(i*512)==0x0015 and uint16(i*512+4) == uint16(i*512+6) and uint16(i*512+4) != 0 and uint32(i*512+4
        // look for high entropy in the record[8:] to account for possible block header
        and math.entropy(i*512+0x70+uint16(i*512+0x70+0x22)+8, i*512+0x70+uint16(i*512+0x70+0x22)+uint32(i*512+(
        )
}
```

```
rule HUNTING_Win_PRIVATELOG_1_strict {

    meta:
        author = "adrien.bataille@mandiant.com"
        description = "Detects PRIVATELOG and STASHLOG variants based on strings and imports"
        md5 = "91b08896fbda9edb8b6f93a6bc811ec6"
    strings:
        $hvid = "Global\\HVID_" ascii
        $apci = "Global\\APCI#" wide
    condition:
        uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and
        (
            all of them and
            (
                pe.imports("clfsw32.dll","CreateLogMarshallingArea") and
                pe.imports("kernel32.dll", "VirtualProtect") and
                pe.imports("ktmw32.dll", "CreateTransaction") and
                pe.imports("kernel32.dll", "CreateFileTransactedA")
            )
        )
}
```

```
rule HUNTING_Win_PRIVATELOG_2_notstrict {
    meta:
        author = "adrien.bataille@mandiant.com"
        description = "Detects possible PRIVATELOG and STASHLOG variants based on strings or imports. This rule
        md5 = "91b08896fbda9edb8b6f93a6bc811ec6"
    strings:
        $hvid = "Global\\HVID_" ascii
        $apci = "Global\\APCI#" wide
    condition:
        uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and
        (
            any of them or
            (
                pe.imports("clfsw32.dll","CreateLogMarshallingArea") and
                pe.imports("kernel32.dll", "VirtualProtect") and
                pe.imports("ktmw32.dll", "CreateTransaction") and
                pe.imports("kernel32.dll", "CreateFileTransactedA")
            )
        )
}
```

```
rule HUNTING_Win_hijack_prntvpt {
    meta:
        author = "adrien.bataille@mandiant.com"
        description = "Detects possible hijack of legitimate prntvpt.dll based on missing export"
        md5 = "91b08896fbda9edb8b6f93a6bc811ec6"
    condition:
        uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and
        pe.exports("PTOpenProviderEx")
        and not pe.exports("MergeAndValidatePrintTicketThunk")
}
```

**EDR / SIEM**

To complement static hunting with Yara, Mandiant also recommends hunting for similar indicators of compromise in “process”, “imageload” or “filewrite” events of typical EDR logs. These would cover cases where PRIVATELOG may resolve imports dynamically with *LoadLibrary()* and *GetProcAddress()*, versus static imports in currently known samples.

Figure 5 identifies key modules loaded by PRIVATELOG that may be used to create hunting queries: ktmw32.dll, dbghelp.dll and clfsw32.dll.

| | | | | | |
|---|---|---|---|---|---|
| 0x7ff6e20b0000 | Image | 64 kB | WCX | C:\Windows\System32\svchost.exe | 20 kB |
| 0x7fff5b710000 | Private | 64 kB | RWX | | 4 kB |
| 0x7fff9b700000 | Image | 3,496 kB | WCX | C:\Windows\System32\spool\drivers\x64\3\PrintConfig.dll | 44 kB |
| 0x7fffa8d80000 | Image | 548 kB | WCX | C:\Windows\System32\winspool.drv | 32 kB |
| 0x7fffae8f0000 | Image | 44 kB | WCX | C:\Windows\System32\ktmw32.dll | 16 kB |
| 0x7fffb0080000 | Image | 104 kB | WCX | C:\Windows\System32\WindowsPowerShell\v1.0\dbghelp.dll | 16 kB |
| 0x7fffb00a0000 | Image | 180 kB | WCX | C:\Windows\System32\prntvpt.dll | 76 kB |
| 0x7fffb89f0000 | Image | 100 kB | WCX | C:\Windows\System32\clfsw32.dll | 16 kB |
| 0x7fffbe970000 | Image | 40 kB | WCX | C:\Windows\System32\version.dll | 16 kB |
| 0x7fffc2460000 | Image | 956 kB | WCX | C:\Windows\System32\propsys.dll | 24 kB |
| 0x7fffc5020000 | Image | 232 kB | WCX | C:\Windows\System32\IPHLPAPI.DLL | 16 kB |

Figure 5: Memory view of a running PRIVATELOG process

Example hunting queries include:

- Any process writing or loading C:\Windows\System32\WindowsPowerShell\v1.0\dbghelp.dll
- Any process loading both clfsw32.dll and ktmw32.dll
- svchost.exe -k print loading clfsw32.dll or ktmw32.dll
- Any svchost.exe process loading clfsw32.dll

Concerning svchost.exe, although we have observed many cases of other svchost.exe processes loading ktmw32.dll, we have only rarely observed svchost.exe processes loading clfsw32.dll.

File writes to .regtrans-ms or .blf files are fairly common, however stacking the process name and file paths may also provide good results. For example, file writes to the registry transaction file for the default user are likely to be uncommon.

## Hashes

### PRIVATELOG

Prntvpt.dll:

1e53559e6be1f941df1a1508bba5bb9763aedba23f946294ce5d92646877b40c

### STASHLOG

Shiver.exe:

720610b9067c8afe857819a098a44cab24e9da5cf6a086351d01b73714afd397

## MITRE ATT&ACK Techniques

| ID | Technique |
|---|---|
| T1012 | Query Registry |
| T1564 | Hide Artifacts |
| T1574 | Hijack Execution Flow |

| T1574.002 | DLL Side-Loading |
| --- | --- |
| T1055.013 | Process Injection: Process Doppelgänging |

**FireEye Product Detections**

| **Platform(s)** | **Detection Name** |
| --- | --- |
| Network Security<br>Email Security<br>Detection On Demand<br>Malware Analysis<br>File Protect | FE_APT_Loader_Win_PRIVATELOG<br>FE_APT_Installer_Win_STASHLOG |
| HX Security | Generic.mg.0c605276ff21b515 |

Posted in

- Threat Intelligence
- Security & Identity

---

Source: https://www.mandiant.com/resources/unknown-actor-using-clfs-log-files-for-stealth