Analyzing Ursnif's Behavior Using a Malware Sandbox

Jun 25th 2019

Share blog:

back to blog overview

Ursnif is a group of malware families based on the same leaked source code. When fully executed Urnsif has the capability to steal banking and online account credentials. In this blog post, we will analyze the payload of a Ursnif sample and demonstrate how a malware sandbox can expedite the investigation process.

Access the VMRay Analyzer Report for Ursnif

This blog post will cover a behavioral analysis of a single Ursnif variant. It does not provide comprehensive insights into web injects, infrastructure or attribution. For additional Ursnif analysis see Appendix D.

Contents

Ursnif Sample Overview

When Ursnif is downloaded and run (say via a malicious attachment in an email), it first spawns its own explorer.exe process and injects itself into the rogue process. As few legitimate applications ever start their own explorer.exe processes, and we cannot think of a valid reason that the application should inject itself into the process, this technique surfaces as a good indicator for detection. The newly created explorer.exe then injects into the legitimate `explorer.exe` process, as shown below:

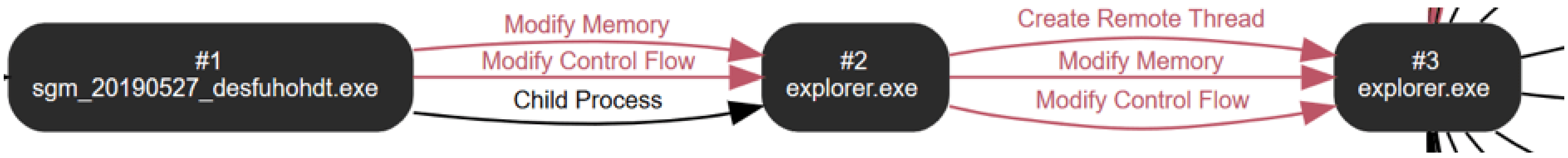

Figure 1: VMRay process graph showing injections

The initial `explorer.exe` process installs its configuration under the registry key `HKCU\Software\AppDataLow\Software\Microsoft\{Machine-specific ID}`, as shown below:

HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580	Access
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Client	Read, Write
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Config	Access
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Exec	Read
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\LastTask	Read
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Run	Access
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Scr	Read
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Sfi	Access
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Sfi\764028EAB3C0274F06	Write
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\Sfi\AECAA210A7EA5C4A0F	Write
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\{46DA6D74-EDEC-6869-A7DA-711CCBAE3510}	Read, Write
HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\3632F5D8-1D04-D8B6-57CA-A18C7B9E6580\{E12FFA4A-CC07-BBA0-DEA5-C01FF2A9F4C3}	Read, Write

Figure 2 – Configuration being written into registry

The malicious behavior happens inside the injected, and legitimate, `explorer.exe` process based on what was written to the config. Examples include various types of credential stealing, browser injection, and system information collection functions.

C2 Check-Ins

Ursnif communicates over standard HTTP using encrypted HTTP request parameters. During the execution it performs a number of C2 check-ins, as shown below:

2 Hosts

Requests	Severity
pilodirsob.com 443	UN
192.168.0.1 53	NA

HTTP Requests (8)

Method	URL	Response	Dest. IP	Dest. Port	Reputation
GET	pilodirsob.com/images/5qbVQlb0ymuWmr_2FkDD/...		5.188.60.53	443	UNKNOWN
GET	pilodirsob.com/images/ALm9doLlVIZDvXXaVPSD5...		5.188.60.53	443	UNKNOWN
POST	pilodirsob.com/images/t_2Bbwrq/4hGdgyKXBVaYl8...		5.188.60.53	443	UNKNOWN
POST	pilodirsob.com/images/VpSwnjfgapjrZm6glom/TOD...		5.188.60.53	443	UNKNOWN
GET	pilodirsob.com/images/jZqrkd6qeE46/7g6Fv_2FdEu...		5.188.60.53	443	UNKNOWN
POST	pilodirsob.com/images/FDqtJFjzhKMo0poCP1/rWlB...		5.188.60.53	443	UNKNOWN
GET	pilodirsob.com/images/08fOuvOECMJ8jg/Vv5aYZds...		5.188.60.53	443	UNKNOWN
POST	pilodirsob.com/images/micpvTcOEehAuAs7/_2F_2F...		5.188.60.53	443	UNKNOWN

Request | Function Logs (1)

Timestamp	96.231000
URL	pilodirsob.com/images/5qbVQlb0ymuWmr_2FkDD/NVO_2FaAbeais0tlU4Y/q6BT_2B9eGflol43LtlhuV/QtnQchMUX6n9F/B3asYZXw/_2FhYDUJMTYaB3PKILEcVcg/WMgDIGrshB/e0T_2F3OwLtl327Jy/bBo858JdBzTI/m9AayoD6ps_/2Box0bRB6Ldta7/Ec_2F84BmjL_2BnKYZQkp/kBMno7exP3mbnkFE/DFUo4OOfG5hYXwg/QTjEpneV/Z.jpeg

Figure 3: Urnsif’s Network Activity

In the function log we can see the request as cleartext before it is encrypted:

```
[0096.123] wsprintfA (in: param_1=0x9a90820, param_2="soft=1&version=%u&user=%08x%08x%08x%08x&server=%u&id=%u&crc=%x&guid=%08x%08x
    %08x%08x" | out: param_1="
    soft=1&version=300054&user=c1bfad56d6eccbb57e3f355780659e32&server=12&id=1000&crc=114f9e9&guid=ea61ad9b12ba194535a941e1b6b4241f")
     returned 127
```

Figure 4: Function log showing the unencrypted HTTP request

Ursnif then prepends a runtime-generated junk parameter to this request, also visible in the function log:

```
[0096.123] sprintf (in: _Dest=0x9770f50, _Format="%s=%s&" | out: _Dest="uhrg=gbeicj&") returned 12
```

Figure 5: Ursnif adding a junk parameter to the HTTP request

After this function completes, the parameters are encrypted and sent over HTTP to one of the C2 servers.

The request contains various identifiers:

- Soft
- Version
- User
- Server
- ID
- CRC
- GUID

During our sandbox execution, the parameters, with the exception of the CRC, remained constant. The CRC parameters for this sample include:

- 114f9e9
- 114f95d
- 1198d90
- 11e2176

This sample leveraged POST requests to upload files for data exfiltration, and added an additional `name=X` parameter used to indicate the filename.

Stealing Functionality

Besides the Man-in-the-Browser (MitB) attack, various stealer modules were also found to be active.

Cryptocurrency + Disk Encryption

In each injected process, the stealer checks if the process name belongs to a supported cryptocurrency wallet, VeraCrypt, or TrueCrypt. The stealer also looks for a process containing the string “*JEdudus.*”, which we couldn’t match to a real application but it is among the cryptocurrency wallet names.

```
[0043.034] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="electrum-") returned 0x0
[0043.034] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="bitcoin") returned 0x0
[0043.034] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="multibit-hd") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="bither") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="msigna.") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="Jaxx.") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="JEdudus.") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="armory-") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="veracrypt") returned 0x0
[0043.035] StrStrIW (lpFirst="C:\\Windows\\Explorer.EXE", lpSrch="truecrypt") returned 0x0
```
Figure 6: Checking if the current process name contains one of the strings known by the module

The stealer module looks for cryptocurrency wallets that contain the following strings:

- electrum-
- bitcoin
- multibit-hd
- bither
- msigna
- Jaxx
- armory-

OLSTEALER & IESTEALER

Besides cryptocurrency stealing, the modules named `OLSTEALER` and `IESTEALER` were also visible.

4/5 Information Stealing Exhibits Spyware behavior

- Tries to read sensitive data of: Microsoft Outlook, Internet Explorer / Edge, Mozilla Firefox.

Figure 7: VMRay Threat Identifier (VTI) match for Ursnif's data stealing

OLSTEALER steals data from Outlook, including login information, and stores it in a local file. The internal name of the module is visible in Function log:

```
[0096.008] lstrlenA (lpString="#OLSTEALER#\n") returned 12
```

Figure 8: OLSTEALER module name visible in the function log

The contents of the created file appear as follows:

```
1  #OLSTEALER#
2  SMTP Server: [redacted]
3  POP3 Server: [redacted]
4  Email: [redacted]
5  POP3 User: [redacted]
```

Figure 9: File created by the OLSTEALER module to store data

The IESTEALER module reads Internet Explorer history and passwords.

```
[0096.536] lstrlenA (lpString="#IESTEALER#\n") returned 12
```

Figure 10 – IESTEALER module name visible in the function log

2/5 Information Stealing Reads sensitive browser data

- Trying to read sensitive data of web browser "Internet Explorer / Edge" by file. •••
- Trying to read sensitive data of web browser "Internet Explorer / Edge" by registry. •••
- Trying to read credentials of web browser "Internet Explorer" by reading from the system's credential vault. •••

Figure 11 – Detection and details for the different password stealing attempts

After stealing from Internet Explorer, the malware also looks for Thunderbird, though the name of the Thunderbird stealer module (TBSTEALER) did not explicitly appear.

```
[0099.209] StrStrIW (lpFirst="Software\\Mozilla", lpSrch="Thunderbird") returned 0x0
[0099.209] LocalAlloc (uFlags=0x40, uBytes=0x1080) returned 0x72f7c00
[0099.209] RegOpenKeyExW (in: hKey=0xffffffff80000002, lpSubKey="Software\\Mozilla", ulOptions=0x0, samDesired=0x20219,
    phkResult=0x54dfd78 | out: phkResult=0x54dfd78*=0xb5c) returned 0x0
[0099.210] GetProcAddress (hModule=0x7fefd710000, lpProcName="RegEnumKeyExW") returned 0x7fefd72c310
[0099.210] RegEnumKeyExW (in: hKey=0xb5c, dwIndex=0x0, lpName=0x72f7c00, lpcchName=0x54dfd68, lpReserved=0x0, lpClass=0x0,
    lpcchClass=0x0, lpftLastWriteTime=0x0 | out: lpName="Firefox", lpcchName=0x54dfd68, lpClass=0x0, lpcchClass=0x0,
    lpftLastWriteTime=0x0) returned 0x0
```

Figure 12: Ursnif looking for Thunderbird data

System Info Gathering

Using built-in Windows system tools Ursnif gathers information about the system. The tools used are:

- `systeminfo.exe` – various info about the system including OS version, installed patches, domain, and basic hardware information
- `net view` – show network shares
- `nslookup 127.0.0.1` – local IP
- `tasklist.exe /SVC` – Services
- `driverquery.exe` – Installed drivers
- (Installed software)`reg.exe query "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`
  `reg.exe query "HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall`

Data Exfiltration

Ursnif caches stolen data to the hard drive into temp files, compresses them into CAB files, and uploads them.

Steps followed to create the CAB:

1. The various stealer modules create files on the hard drive. Some use the `%TEMP%` directory, others use the random directory created earlier.

Create Temp File	C:\Users\aETAdzjz\AppData\Local\Temp\1D0E.tmp	path = C:\Users\aETAdzjz\AppData\Local\Temp\
Create Temp File	C:\Users\aETAdzjz\AppData\Local\Temp\2855.tmp	path = C:\Users\aETAdzjz\AppData\Local\Temp\
Create Temp File	C:\Users\aETAdzjz\AppData\Local\Temp\1FB1.tmp	path = C:\Users\aETAdzjz\AppData\Local\Temp\
Create Temp File	C:\Users\aETAdzjz\AppData\Local\Temp\E3D6.tmp	path = C:\Users\aETAdzjz\AppData\Local\Temp\
Create Temp File	C:\Users\aETAdzjz\AppData\Local\Temp\DB32.tmp	path = C:\Users\aETAdzjz\AppData\Local\Temp\

Figure 13: VMRay's Behavior Tab showing temporary file creation

Create	C:\Users\aETAdzjz\AppData\Roaming\Microsoft\{F5FB2C3C-D05C-EF89-82F9-0493D63D7877}\01D51ED4E3ECF92009	desired_access = GENERIC_WRITE, GENERIC_READ, file_attributes = FILE_ATTRIBUTE_NORMAL	✓	1	FN

Figure 14: Ursnif module creating file in it own randomly named folder

2. Before sending home the data, Ursnif uses the `makecab` tool to compress it. Makecab is able to accept a directive file when being called with the `/F` parameter, which defines the source and target. For each CAB file it needs to create, Ursnif drops a directive file.

- `01D51ED4E3ECF92009` is the output of the OLSTEALER module.

```
#OLSTEALER#
SMTP Server: [redacted]
POP3 Server: [redacted]
Email: [redacted]
POP3 User: [redacted]
```

Figure 15: File containing output of the OLSTEALER module

- `1FB1.bin` contains the directives to compress `01D51ED4E3ECF92009` to `2855.bin`

```
.set MaxDiskSize=0
.set DiskDirectory1="C:\Users\aETAdzjz\AppData\Local\Temp"
.set CabinetName1="2855.bin"
.set DestinationDir=""
"01D51ED4E3ECF92009"
```

Figure 16: File containing directives for makecab

- 2855.bin is the CAB file created by makecab by calling it with the directive file

Command Line	makecab.exe /F "C:\Users\aETAdzjz\AppData\Local\Temp\1FB1.bin"

Figure 17: VMRay's Behavior tab showing the command line for calling makecab

3. To send the data home, Ursnif uses the same C2 channel with the same encryption, but our analyzed variant also adds an additional "`name`" parameter to indicate the filename.

```
[0111.775] wsprintfA (in: param_1=0x278f690, param_2="version=%u&soft=1&user=%08x%08x%08x%08x&server=%u&id=%u&type=%u&name=%s
    &guid=%08x%08x%08x%08x"
    | out: param_1="version=300054&soft=1&user=c1bfad56d6eccbb57e3f355780659e32&server=12&id=1000&type=15&name=2855.bin&guid=ea
    61ad9b12ba194535a941e1b6b4241f") returned 137
```

Figure 18: Function log showing Ursnif adding an additional "name" parameter

4. Just like with normal check-ins, Ursnif adds a junk parameter (e.g. `uhrg=gbeicj&`), then encrypts the parameters.

5. Then it includes the uploaded file in the POST contents to send the file.

Man-in-the-Browser

Overview of Man-in-the-Browser Attacks

- A lot of valuable user data (banking, shopping, etc.) is not stored on the hard drive or registry, but accessed through web applications.
- HTTPS is now the norm. User data is transmitted over a secure connection so Man-in-the-Middle (MitM) from outside the browser is more difficult.
- To get access to this valuable data, malware goes *inside* the browser.
- On Windows, browsers are not well-protected from applications that are already running on the same host – for a website it is very challenging to exploit a modern web browser and escape the browser sandbox. That being said, it's not difficult to get in when the attacker already has code execution on the system.
- The usual goal is a web inject:
  1. A trojan is running on the same host as the browser,
  2. Inject into the browser,
  3. Install API hooks, and then
  4. Modify API calls to include the attacker's JavaScript, which is specific per website.

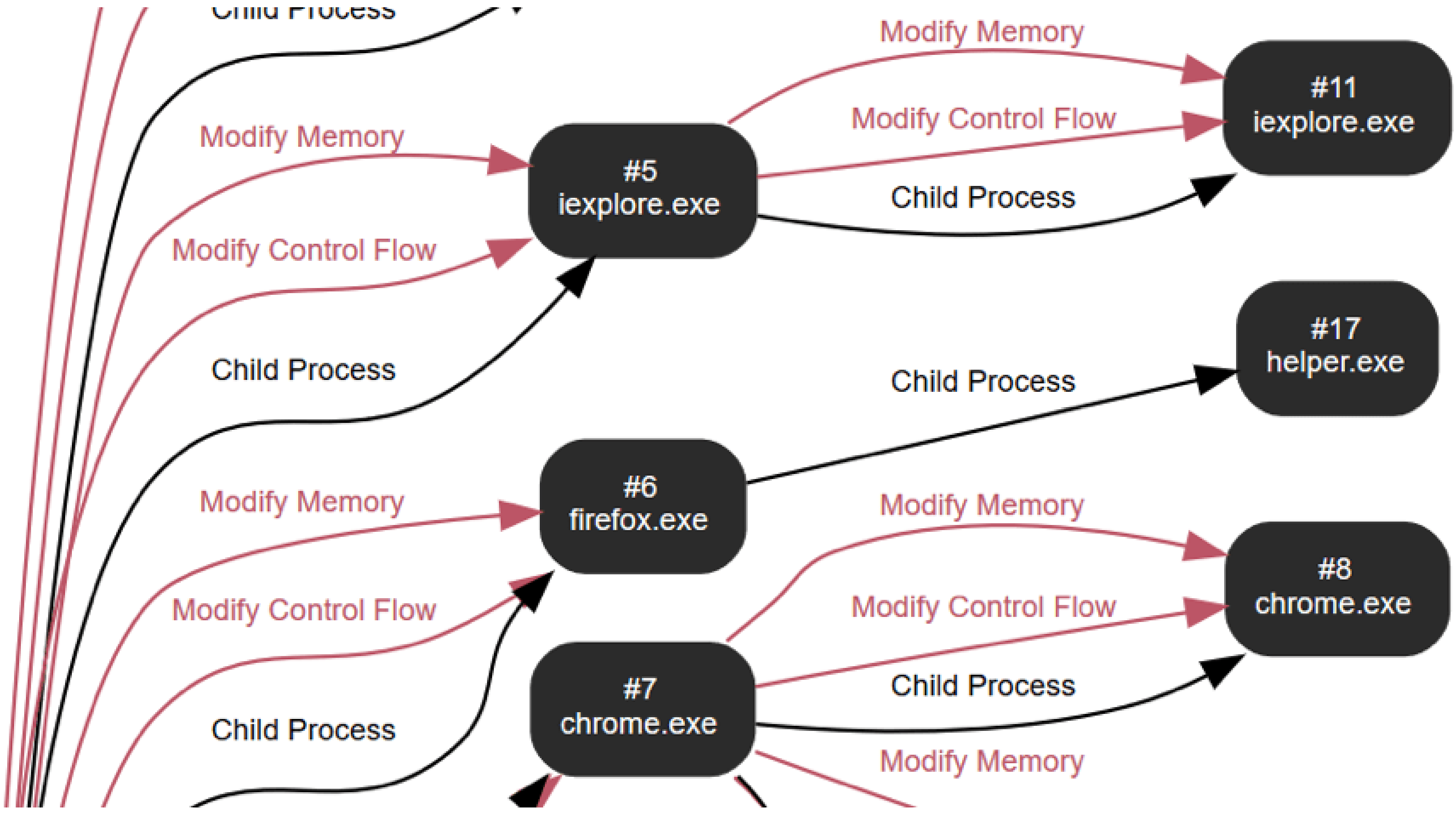

Figure 19: VMRay process graph showing Ursnif injecting code into Internet Explorer, Firefox and Chrome

SPDY & HTTP/2

SPDY (pronounced “speedy”) is a network protocol that enables compression of HTTP-transmitted data, and HTTP/2 is a version of HTTP derived from SPDY with the same goal. Though neither are security features, they just implement compression, they still cause a bit of extra work for attackers looking to implement MitB attacks. Attackers must decompress the HTTP traffic, or they can just turn off SPDY and HTTP/2 altogether – most attackers prefer to turn them off instead of implementing an extra feature. The fact that a process turns this feature off is a good indicator for a defender – legitimate processes have no reason to turn off compression but browser injectors often do it.

Ursnif:

- For Internet Explorer: Sets the ***EnableSPDY3_0*** value in the *`HKCU\\SOFTWARE\Microsoft\Windows\CurrentVersion\Internet Settings`* to **0**.
- For Chrome: Starts a chrome process with the `--use-spdy=off`command line argument.
- For Firefox: this sample didn’t turn off SPDY for Firefox, though we have observed other variants edit the `prefs.js` file in the Firefox Profile folder, adding the following line:
  `user_pref("network.http.spdy.enabled", false);`

A similar approach is possible for HTTP/2. The attacker can edit the registry to disable in Internet Explorer, use a command line parameter for Chrome, and edit the `prefs.js` file for Firefox.

Different Browsers, Different Hooks

The sandbox can report on installed hooks in the “Hook Information” section of each process. For each installed hook it shows the original API, and the (already overwritten) address it points to now. The overwritten address is shown as [closest symbol + offset] to make matching to the overwritten code easier.

Internet Explorer

The observed hooks added to some exported functions of `wininet.dll` were:

- `InternetReadFile`
- `InternetWriteFile`
- `InternetReadFileExW`
- `HttpSendRequestW`
- `InternetQueryDataAvailable`
- `HttpOpenRequestW`
- `InternetCloseHandle`

85. entry of urlmon.dll	4 bytes	wininet.dll:InternetReadFile+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d6a
96. entry of urlmon.dll	4 bytes	wininet.dll:InternetWriteFile+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d6f
89. entry of urlmon.dll	4 bytes	wininet.dll:InternetReadFileExW+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d79
97. entry of urlmon.dll	4 bytes	wininet.dll:HttpSendRequestW+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d83
86. entry of urlmon.dll	4 bytes	wininet.dll:InternetQueryDataAvailable+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d88
92. entry of urlmon.dll	4 bytes	wininet.dll:HttpOpenRequestW+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d8d
116. entry of urlmon.dll	4 bytes	wininet.dll:InternetCloseHandle+0x0 now points to wininet.dll:InternetConfirmZoneCrossing+0x14d97

Figure 20: VMRay Analyzer showing information about the hooks added to Internet Explorer

Firefox

This specific sample was not interested in attacking Firefox with web injects. Some other Ursnif variants add hooks to the nss3.dll loaded by Firefox:

- `PR_Read`
- `PR_Write`
- `PR_Close`

Hook Information

Type	Installer	Target	Size	Information
IAT	pagefile_0x0000000000800000:+0x18bc6	2357. entry of xul.dll	4 bytes	nss3.dll:PR_Read+0x0 now points to pagefile_0x0000000000800000:+0xf353
IAT	pagefile_0x0000000000800000:+0x18bc6	2345. entry of xul.dll	4 bytes	nss3.dll:PR_Write+0x0 now points to pagefile_0x0000000000800000:+0x8168
IAT	pagefile_0x0000000000800000:+0x18bc6	2350. entry of xul.dll	4 bytes	nss3.dll:PR_Close+0x0 now points to pagefile_0x0000000000800000:+0x1c9b0

Figure 21: VMRay Analyzer showing the hooks added to Firefox

The hooked functions are exports of `nss3.dll`

Chrome

Adding hooks is easy with Internet Explorer and Firefox. Since Chrome's DLL doesn't export the necessary functions, however, the attacker needs to manually find them in the binary and add the changes.

```
lstrcmpiA (lpString1="LoadLibraryExW", lpString2="LoadLibraryExW") returned 0
VirtualProtect (in: lpAddress=0x7fefd605248, dwSize=0x8, flNewProtect=0x40, lpflO
VirtualProtect (in: lpAddress=0x7fefd605248, dwSize=0x8, flNewProtect=0x2, lpflOl
VirtualQueryEx (in: hProcess=0xffffffffffffffff, lpAddress=0x7fefee30000, lpBuffe
NtQueryInformationProcess (in: ProcessHandle=0xffffffffffffffff, ProcessInformati
RtlAllocateHeap (HeapHandle=0x1ee0000, Flags=0x0, Size=0x60) returned 0x1ef0dd0
```

237. entry of shell32.dll	4 bytes	kernel32.dll:CreateProcessAsUserW+0x0 now points to pagefile_0x000000001da0000:+0x329f0	•••
252. entry of user32.dll	4 bytes	kernel32.dll:CreateProcessW+0x0 now points to pagefile_0x000000001da0000:+0x326b4	•••
272. entry of user32.dll	4 bytes	kernel32.dll:LoadLibraryExW+0x0 now points to kernel32.dll:RegDeleteTreeA+0x23a	•••
88. entry of msctf.dll	4 bytes	kernel32.dll:CreateProcessW+0x0 now points to pagefile_0x000000001da0000:+0x326b4	•••
89. entry of msctf.dll	4 bytes	kernel32.dll:LoadLibraryExW+0x0 now points to kernel32.dll:RegDeleteTreeA+0x23a	•••
298. entry of ole32.dll	4 bytes	kernel32.dll:CreateProcessW+0x0 now points to pagefile_0x000000001da0000:+0x326b4	•••
28. entry of version.dll	4 bytes	kernel32.dll:LoadLibraryExW+0x0 now points to kernel32.dll:RegDeleteTreeA+0x23a	•••

Figure 22: Ursnif's hooking of Chrome as visible on the sandbox level

Extracting the Modules for Static Analysis

It is common for Ursnif samples to implement the malware functionality in a DLL, compress the DLL, attach it to the loader, and pack the whole binary together (loader+DLL). To extract the uncompressed binary, we need to:

1. Unpack the sample: To achieve this, we execute the packed sample in VMRay Analyzer. The sandbox dumps the memory of the unpacked sample, which contains the compressed module.

Memory Dumps

Name	Start VA	End VA	Dump Reason	PE Rebuilds	Bitness	Entry Points	AV	YARA	Actions
sgm_20190527_desfuhohdt.exe	0x00400000	0x0051FFFF	Relevant Image	-	32-bit	-	✕	✕	•••
sgm_20190527_desfuhohdt.exe	0x00400000	0x0051FFFF	Process Termination	-	32-bit	-	✕	✓	•••

Figure 23: Memory dump of the original executable with a tick in the YARA column indicating a match

2. Extract the compressed module: We need 2 elements: the memory dump itself which contains the compressed module and the offset where the module begins. We get this info with the help of a built-in YARA rule that matches on the memory dump which contains the `apLib`-compressed PE header. The analysis archive contains the memory dump and the offset of the match, we do the extraction based on this.

2/5	YARA	YARA match	1	-

- Rule "APLib_Compressed_PE" from ruleset "Generic" has matched on a memory dump for process "sgm_20190527_desfuhohdt.exe". •••

Figure 24: Detection showing that VMRay's built-in YARA for APLib-compressed PE files has matched

3. Decompress the module: For this we use an open-source `apLib` decompressor by @sandornemes.

The result is the decompressed DLL. According to the headers, the module was compiled on May 26, 2019, and as usual for Ursnif, it is referred to as `client.dll`.

Offset	Name	Value	Meaning
621E0	Characteristics	0	
621E4	TimeDateStamp	5CEB19DF	Sunday, 26.05.2019 22:57:35 UTC
621E8	MajorVersion	0	
621EA	MinorVersion	0	
621EC	Name	63A08	client.dll
621F0	Base	1	

Figure 25: PEBear showing information about the decompressed DLL

The DLL doesn't have any exported functions, everything is only reachable from `DLLMain`, the flow of execution depends on the installed registry entry (described in the very beginning of the post).

Conclusion

We hope you find value in our analysis of Ursnif. The analysis and understanding of the various facets of the malware could have been conducted and collected manually but our investigation was greatly accelerated by using VMRay Analyzer. The publicly-available VMRay Analyzer report for the Ursnif variant discussed throughout this post can be found here: https://www.vmray.com/analyses/53f7d917ad9e/report/overview.html

We look forward to bringing you future detailed reports to help expedite your analysis, understanding, and defensive capabilities. Please view the appendices for the associated IOCs, MITRE ATT&CK mappings, and related work.

We encourage you to sign up for a trial of VMRay Analyzer, upload your own Ursnif samples, and contact us if you notice evolutions or changes in our findings. Until next time!

Appendix A: Indicators for Host-Based Detection & Identification

- User process spawning its own explorer.exe process
- Injection into `explorer.exe`
- The configuration is written under registry key `HKEY_CURRENT_USER\Software\AppDataLow\Software\Microsoft\`
- Usage of `makecab` for compression of staged data
- Injection into browser processes
- Turning off SPDY or HTTP/2
- Usage of the following tools for data collection: `systeminfo`, `net`, `nslookup`, `tasklist`, `driverquery`, and `reg`.
- Additional VMRay Sandbox IOCs can be found here: https://www.vmray.com/analyses/53f7d917ad9e/report/ioc.html

Appendix B: Observed MITRE ATT&CK Techniques in this Analysis

The following does not cover all Ursnif techniques, just the ones that came up in the analysis of this sample.

- T1045 – Software Packing

- T1059 – Command-Line Interface
- T1106 – Execution through API
- T1179 – Hooking
- T1055 – Process Injection
- T1140 – Deobfuscate/Decode Files or Information
- T1112 – Modify Registry
- T1003 – Credential Dumping
- T1081 – Credentials in Files
- T1214 – Credentials in Registry
- T1082 – System Information Discovery
- T1016 – System Network Configuration Discovery
- T1135 – Network Share Discovery
- T1007 – System Service Discovery
- T1119 – Automated Collection
- T1074 – Data Staged
- T1185 – Man in the Browser
- T1043 – Commonly Used Port
- T1071 – Standard Application Layer Protocol
- T1132 – Data Encoding
- T1002 – Data Compressed
- T1041 – Exfiltration Over Command and Control Channel

Appendix C: Hashes

SHA256: 53f7d917ad9ebf5b7d2ccc1a835083bc0c0b92cc69ee584703ea6e4345f5c457

Extracted client.dll:
[illegible]6010c5563634bfcad6b9e3f9855e5fcd48d96c1872510ecd6dadf3a7

Appendix D: Related Work

- Peter Kálnai's and Michal Poslušný's VirusBulletin 2017 paper on browser attack points:
- James Wyke's Botconf 2018 talk about web inject tracking
- Maciej Kotowicz's 2016 paper about ISFB
- 0verfl0w's blog posts about reversing ISFB loaders (parts 1 and 2)