Formbook Research Hints Large Data Theft Attack Brewing

By sharon

Published: 2019-06-12 · Archived: 2026-04-05 17:28:16 UTC

In this blog post we will present the latest droppers of Formbook data stealing malware – an advanced malware that uses diverse and innovative techniques to evade security products. We will reverse engineer all the different droppers and suggest ways to detect them. We also show how Cyberbit EDR detects the latest Formbook dropper.

Formbook is a data stealing malware which is capable of stealing data from web browsers and many other applications. Formbook has been for sale on underground hacking forums since early 2016.

Formbook data stealing malware capabilities include:

• Keystroke logging
• Clipboard monitoring
• HTTP/HTTPS/SPDY/HTTP2 form and network request grabbing
• Browser and email client password grabbing
• Capturing screenshots
• Bot updating
• Downloading and executing files
• Bot removing
• Launching commands via ShellExecute
• Clear browser cookies
• Reboot the system
• Shutdown the system
• Download and unpack ZIP archive

Recently, we came across a new sample of Formbook and dissected it. We noticed that its dropper was different from other Formbook samples we encountered.

We dived deeper into the Formbook samples and in particularly – **their droppers**.

A dropper is a malware file whose purpose is to install malware on the system. It may also achieve persistence for malware or perform other malicious activities.

The dropper is used in the initial infection stage and is not usually involved in the final damage inflicting stage of the malware.

To hide the malware within them, the droppers usually use packers, encryptions and obfuscations on the malware’s final payload. The final payload might come as PE or shellcode.

We discovered substantial variation among the new Formbook droppers. They are written in several programming languages and use numerous packers and anti-analysis techniques. This is done to avoid signature based anti-malware products from detecting it. The final, non-encrypted and non-obfuscated payload of Formbook data stealing malware never resides on the disk, only in the memory and therefore makes detection much more difficult.

Let's reverse engineer the new Formbook droppers and share meaningful insights about them.

At the end of this post, you will find a summary and suggestions for detecting the droppers. You will also see how Cyberbit EDR detects the latest dropper.

We analyzed the samples uploaded to Malpedia, along with a new sample we found in the Cyberbit Malware Lab.

Formbook Data Stealing Malware Research Summary

Sample #	Language	Packed	Anti-Analysis
1	Visual Basic	No	Encryptions, Obfuscations
2	Visual Basic	No	Encryptions
3	Visual Basic	No	Anti-Debug, Encryptions, Anti-Sandbox, Anti-Memory Analysis
4	Visual Basic	No	Anti-Debug, Encryptions, Anti-Sandbox, Anti-Memory Analysis
5	c++	NSIS installer	Encryptions, Anti-Sandbox
6	C#	Yes	Encryptions, Obfuscations, Anti-Sandbox, Anti-Debug, Running processes/Loaded modules checks
7	C#	yes,with Confuser	Encryptions, Obfuscations
8	AutoIt	Heavily obfuscated AutoIt Script	Encryptions, Obfuscations, Running processes checks
9	Delphi	No	Encryptions, Filename checks, Running processes checks, Anti-Debug, Anti-Sandbox
10	Delphi	Yes, with ASProtect	Encryptions, Anti-Sandbox

Formbook samples analysis – droppers

- Most Formbook droppers operate in a typical way: when executed, they perform process hollowing. The file chosen for the hollowed process is usually the sample itself. The code unpacked in the hollowed process is the final Formbook payload.
- When stating that a sample is packed, we mean the first PE file is packed with some packer (e.g: ASProtect, Confuser, etc…). Of course, during execution more code may be unpacked in memory.
- We analyze each sample until the point that the final Formbook malware payload is unpacked.
-

Learn more about Cyberbit EDR Kernel-Based Endpoint Detection vs. Whitelisting

Formbook Sample 1

SHA256: 2b78b42a1f3c58cabd984935a76b0e9c57934adf80b719eb499f4dd328c09f69

Anti-Analysis: Encryptions, Obfuscations

Packed: No

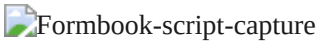
Process Hollowing: No

Programming language: Visual Basic

Highlight: Use of mshta.exe to execute vbscript for persistence

This is the newest Formbook sample we found.

In contrary to many other samples of Formbook, the main sample doesn't unpack itself to another process. Instead, it drops a file called "Rhododendrons8.exe" and creates two processes: one is mshta.exe running this visual basic script:



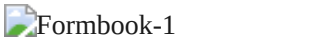


Figure 1 – ShellExecuteA with mshta.exe and the vbscript (truncated)

mshta.exe (Microsoft HTML Application Host) is a Windows utility for executing HTML application files. It can also run Visual Basic scripts.

The script is simple and adds the dropped copy of the malware to registry autorun key, so it will execute as Windows starts.

Notice the simple obfuscation: Instead of writing "CreateObject", "CrXXteObject" is written and "XX" is later replaced with "ea". This is done to prevent signature-based tools from detecting this method being in this script.

It is the first and currently only sample of Formbook data stealing malware we observed that achieves persistency via this method.

The second process created is the dropped “Rhododendrons8.exe.” This file unpacks the Formbook payload in its memory.

Rhododendron is a genus of 1,024 species of woody plants in the heath family (Ericaceae). Quite an unusual name for a malware!

Figure 2 – A Rhododendron

Formbook’s payload lies encrypted at address 0x00402c34 within the code section of Rhododendrons8.exe and is decrypted using two algorithms. The first algorithm is proprietary, the second is RC4 with a 256-bytes key.

RC4 is a symmetric stream cipher that is a favorite among malware authors. It is simple to implement, doesn’t require any external libraries, and can be written easily even in x86 assembly.

There are 3 stages for generating the stream cipher that is used encrypt or decrypt the data:

1. Initialize the S-box (256 bytes array with values 0x00-0xFF)

2. Scramble the S-box (byte-swapping)

3. Generate the key stream using the S-box and decrypt/encrypt the data

The three stages can be seen in Figure 5. You can read here in detail about this algorithm and its implementation in malware.

Below are screenshots of the decryption process:

Figure 3 – Encrypted Formbook payload before first round decryption

Figure 4 – First decryption algorithm using XOR operations with other data

Figure 5 – Payload after 1st round of decryption

- 

Figure 6 – RC4 algorithm is used to decrypt the code after the 1st algorithm was executed

Figure 7 – Things looks better after RC4. The MZ header will be copied later.

The final Formbook payload performs injection to explorer.exe. Formbook's injection is rather sophisticated and uses direct system calls to patch Explorer's memory in an elegant way. It was researched in depth by Remi Jullian.

The payload unpacked by the dropper has only a .text section (code) and looks like this in an initial analysis with PE-Bear:

Figure 8 – Formbook's unpacked payload

This is the actual Formbook payload, and we will see this payload in all the following samples as well.

Formbook Sample 2

SHA256: 5c0b2944e674ce0e31c7d119e922fdc3dfb1b5c98ec660234369469bb3bbaa19

Anti-Analysis: Encryptions

Packed: No

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: Visual Basic

Highlight: Unlike sample 1, it doesn't drop another executable

This sample performs process hollowing with the chosen file as the sample itself and unpacks the final Formbook payload in the hollowed process. The decryption is similar to sample 1 (same algorithm + RC4). Unlike sample 1, the main sample doesn't drop another executable.

As usual, the final payload contains a .text (code) section only. Compare figures 9 and 8 to see how similar they look.

Figure 9 – Formbook's unpacked payload – similar to figure 8

Formbook Sample 3

SHA256: 11ded4e6d3d40463109a935d00da53627b47b01ff323c206e7f715ce97509ccc

(Similar to: 578e7858d8587fe251790e2386d334862ed8b5b5aaa6520a22b5da9bf7f3d3fb)

Anti-Analysis: Encryptions, Anti-Debug (PEB checks) Anti-Sandbox (User Interaction, Time acceleration, API checks), Anti-Memory analysis, Anti-VM (MMX check)

Packed: No

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: Visual Basic

Highlight: A technique for evading detection of reflectively-loaded modules

This sample performs process hollowing with the chosen file as the sample itself and unpacks the final Formbook payload in the hollowed process.

Anti-Debug techniques

This sample checks for the presence of debugger using two different checks. If one of the checks returns true – it jumps to address 0x1342EB5 (see explanation later)

- Check the second byte of the PEB structure (BeingDebugged)

  Figure 10 – Checking the 2nd byte of the PEB structure (eax = 0)

- Check the NtGlobalFlag of the PEB structure. If the process is being debugged, this byte is set to 0x70

  Figure 11 – Check the NtGlobalFlag in the PEB structure

Anti-Sandbox & Anti-VM Techniques

This sample has a User-Interaction check that checks the position of the mouse cursor using the GetCursorPos method. If the mouse cursor doesn't change its position during an interval of 1 millisecond, the malware will keep running in a loop.

Figure 12 – Call GetCursorPo to get the mouse cursor position, sleep for 1 millisecond and check it again. If eax = 0, the positions are equal and the malware will jump back to the sleep instruction. It will repeat this process until the mouse position changes.

This sample uses an Anti-VM trick – it checks if the CPU of the machine is able to run MMX instructions – that are usually *not* supported in virtual machines.

To retrieve the information about the CPU features, such as MMX instructions capabilities. The malware executes the following code:

Figure 13 – Check if the CPU has MMX instructions set

1. It calls the cpuid instruction with the value “1” in the EAX register. Executing cpuid this way will fill the EDX register with features information about the processor.
2. It moves the data in the EDX register to the EAX register
3. It shifts the EAX register to the right 0x17 (23 in decimal) times, so the 23 bit (counting from the right) will be the LSB (least significant bit) in EAX
4. It performs “AND” operation on the EAX register with the value “1”
5. If the MMX bit is not set, EAX will be 0 and the program will jump address 0x1342EB5 – the same as in the Anti-Debug checks.

Another nasty trick in this Formbook dropper can actually defeat plugins such as ScyllaHide. Let’s have a look at it:


14

Figure 14 – Check for time acceleration

It calls GetTickCount – which returns the milliseconds elapsed since the system was started. let’s call this result X. It sleeps for 2 seconds and then calls it again – let’s call this result Y. Then, it calculates Y-X and check if it’s < 1.5 seconds. We would expect Y – X to be around 2 seconds. This way, the malware “kills two birds with one stone.”

Here is why: Sandboxes usually hook the Sleep function for time acceleration. For example: if a malware tries to call sleep with 30 seconds, it can be reduced to 1 second by the hooked function.

GetTickCount may also be hooked by sandboxes. Virtual machines have naturally lower GetTickCount result, because they were started a few minutes earlier. The sandbox may hook GetTickCount and return a result reasonable for a normal PC: a few hours or few days.

The problem however, is that if the sandbox hooks GetTickCount or Sleep, you can still compare the time differences and know if the time was accelerated or slowed down.

Not only sandboxes hook these functions – also plugins such as ScyllaHide.

Naturally, we apply Anti-Anti-Debug plugins when debugging malware, but in this case, it will only ruin our debugging process. Let’s see how:

When applying ScylleHide plugin with “GetTickCount” hook, we got X = 0x010A0B3D = 17435453 milliseconds which is about 4.84 hours. A reasonable time for a machine to be up.

On the second call to GetTickCount, we got Y = 0x010A0B3E = 17435454 – Just 1 millisecond longer, and that’s after 2 seconds of sleep! Of course, this doesn’t make sense and the malware is “smart” enough to detect it.

Now, look what happens if this test fails, assuming ScyllaHide’s GetTickCount hook is on: it jumps to address 0x134022E (see figure 14). In the earlier Anti-Analysis checks (MMX, Anti-Debug) we saw it jumps to 0x1342EB5. So, we a have different address this time. First, let’s look at 0x1342EB5:

Figure 15 – This is where the dropper jumps if it detected a debugger was being used or the CPU has no MMX instruction set

It looks like junk code. The first instruction, stosb, already causes an exception, because it tries to copy the value at EAX to the address at the EDI register, but the address at the EDI register is located within a range of pages that doesn't have write permissions. So, in this case we were directly stopped from being able to debug the program.

Let's look at 0x134022E, specifically at its first instruction. We know EAX = 1 because the difference between the two GetTickCounts calls is 1 millisecond.

Figure 16 – ss:[ebp+C] = EAX = 1. This will revenge us later

The execution continues as usual, but we will be punished later!

Figure 17 – if ss:[ebp+C] = 1, the rep movsb instruction will fail

Later in the execution, the value at ss:[ebp+C] is moved to the EDI register. Then, 0x400 is added and "rep movsb" (copy ECX number of bytes from the address at ESI to the address at EDI) instruction is executed, but since 0x401 is not a valid address, it will fail.

Lastly, as an Anti-Sandbox technique, it checks if the SetErrorMode and SetLastError APIs work as expected:

- It calls SetErrorMode(0x800) and then SetErrorMode(0x0), then it checks the return value of the latter call. It should be 0x800 because the return value of SetErrorMode is the value of the older error mode. If it doesn't, it jumps to address 0x134022E (see figure 16).

Figure 18 – Check if SetErrorMode API works as expected

- It calls SetLastError(5) and checks in the TEB that the Last error number (located at (fs:[34]) is indeed 5

Figure 19 – Check if SetLastError API works as expected

We didn't witness the RC4 algorithm like in the previous VB samples, only proprietary decryption algorithms.

This sample has another nice trick to avoid memory analysis – it first writes the Formbook payload to the suspended process without the "M" of its "MZ" header. It writes the "M" in a separate call.

This trick is employed to avoid detection of reflective loading – since detection of newly loaded executables modules can be done by their magic number – “MZ” at their beginning offset in memory.

When scanning this memory region for the first time, the scanner will not identify this is a reflective loaded module – because it misses the “M”.

Figure 20 – Where is ‘M’?

The unpacked payload looks similar to the one in samples 1 & 2 – and will be similar in all the following samples as well.

Formbook Sample 4

SHA256:

ca4b45af726967f5199f60c15cf518eeba09991164b95cc171ecfea3cb0d6f5b

Anti-Analysis: Same as in sample 3

Packed: No

Process Hollowing: Injection – with the chosen file as the sample itself, but without hollowing.

Programming language: Visual Basic

Highlight: Code injection without hollowing

This sample is similar to sample 3 in its Anti-Analysis techniques.

It performs injection to a remote process with the chosen file as the sample itself. In this case, the dropper does not hollow the original content of the file from the remote process (which means it doesn’t call NtUnmapViewOfSection on the main executable module) – but overwrites its code section.

The fewer API calls used for injection – the better – as anti-malware products rely heavily on hooks to detect suspicious activities.

Since the original dropper module is not unmapped from the targeted process and Formbook’s final payload is smaller than the dropper’s code – We can see part of the code of the original dropper still present in the unpacked payload:

Figure 21 – On the left side – the dropper’s code section. On the right – the unpacked code replaced it on the injected process

Figure 22 – The smaller unpacked code didn't overwrite the rest of the dropper, hence the rest of the code remains the same (left file – original sample, the dropper. right file – dump of the injected process)

Formbook Sample 5

SHA256:

2ad57fe2d71e246028891018476dd3d0ddbbd7d39db8056b17d2b2aaf69bfd45

Anti-Analysis: Encryptions, Anti-Sandbox (User-Interaction)

Packed: NSIS installer

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: C++

Highlight: Leveraging NSIS installer to execute the dropper

This sample comes packed in an NSIS (Nullsoft Scriptable Install System) installer and is written in C++.

Unzipping the .exe with 7-zip, reveals 2 folders: $PLUGINSDIR and $TEMP. $PLUGINSDIR contains a file called System.dll which is a plugin for NSIS used to call functions inside third-party DLLs. $TEMP contains some images, probably to disguise the real intent of the executable, a .css file and two other files – Pym.dll (written in C++) and Marestail.bin. Marestail.bin contains Formbook'sfinal payload – and it is encrypted.

Figure 23 – Pym.dll and Marestail.bin in $TEMP

When the sample is executed, System.dll is loaded and its "Call" function is used to load and execute the DLLMain of Pym.dll

Figure 24 – The call to System.dll "Call". The value 2ad5.40B858 contains a pointer to an address that has the path to Pym.dll

Once Pym.dll is executed – it reads and decrypts Marestail.bin to unveil Formbook's payload. Then, the decrypted payload is injected to a hollowed process. Again, the file chosen for hollowing is the sample itself.

Figure 25 – Part of the decryption code from Pym.dll that decrypts Marestail.bin, using XOR operation.

Figure 26 – After the code above is executed, the MZ header of Formbook's payload can be easily spotted

Pym.dll uses a simple Anti-Sandbox technique (User Interaction). It checks if the mouse cursor had moved during a short time interval (50 milliseconds) or if 6 minutes and 30 seconds had passed without any mouse movement; a time interval long enough to evade most sandbox timeouts. Only if at least one of these conditions is met – it continues its execution.

Figure 27 – The Anti-Sandbox technique used in Pym.dll

Formbook Sample 6

SHA256:

8c6f2ebfffd5afdc10e18e13b579ed5dd3ffb8bfc3362062465fa36f60954dd8

Anti-Analysis: Encryptions, Obfuscations, Anti-Debug, Running processes checks, Loaded modules check, Anti-Emulation

Packed: Yes

Process Hollowing: Yes – with chosen file as Microsoft's svchost.exe

Programming language: C#

Highlight: Anti-Emulation (check for time acceleration), file chosen for process hollowing is Microsoft's svchost.exe

This sample is written in C#. It is obfuscated and contains a lot of functions with random looking names. It has an encrypted resource that is decrypted and then executed.

Figure 28 – A resource is decrypted using the "fukam" function

The decrypted resource is another C# program. It has some Anti-Analysis checks. It checks the following conditions and if at least one of them is true, it terminates the process.

- **Wireshark is running** – by checking all Windows titles for a window with the title "The Wireshark Network Analyze"
- **The process is being debugged** – by calling IsDebuggerPresent
- **Check for time acceleration** – Call GetTickCount, sleep for 500 milliseconds, call GetTickCount again and check if the difference between the GetTickCount calls is smaller than 500 milliseconds. This can indicate hooking of Sleep/GetTickCount APIs (See sample 3)

- **Sandboxie (sandbox product) module is loaded** – By checking if GetModuleHandle("SbieDll.dll") returns a non-null value

If none of the checks above return true, it decrypts a DLL from its resource that achieves persistence for the sample:

Its drops the original sample to:

%USERPROFILE%\AppData\Roaming\SYRPSVT\SYRPSVT.exe

And an .xml file to:

%USERPROFILE%\AppData\Roaming\SYRPSVT\$X.xml

And creates a scheduled task using this command:

"C:\Windows\System32\schtasks.exe" /Create /TN "SYRPSVT\SYRPSVT" /XML "%USERPROFILE%\AppData\Roaming\SYRPSVT\$X.xml"

Where $X = 'a' + five times the same random_char, where random_char in A-Z,a-z,_,0-9 (e.g: abbbbb.xml)

This DLL also injects the final Formbook payload (also lies encrypted in the resource section) into a hollowed Microsoft svchost.exe process.


29

Figure 29- the ecrypted resource. The array "bytes" contain both a DLL (named RunLib) and Formbook payload. "okapise" is the function responsible for maintaining persistence. "Inj" is the function responsible for injecting into the hollowed svchost.exe

Formbook Sample 7

SHA256:

148b7cae0915f50537cf6752b45ac117a92a9d59aaff46518965e56e4f4de542

Anti-Analysis: Encryptions, Obfuscations

Packed: With "Confuser" packer

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: C#

Highlight: Usage of a less-common API in process hollowing (NtAlertResumeThread)

Another sample written in C#. This one is packed with Confuser 1.9.0.0.

Unpacking and looking at the unpacked sample – reveals the injection method, which is slightly different than in other samples.

One of the stages in process hollowing technique is resuming the main thread of the suspended process, after it has been filled with a new code. The usual API call which is used for this operation is ResumeThread.

In this case, the malware chooses a less common API –NtAlertResumeThread. This API performs same operation as ResumeThread but also alerts the thread so that is returns immediately from the wait with status STATUS_ALERTED


30-Formbook

Figure 29 – A snippet from the unpacked C# code shows which APIs are used for the injection

This was probably done to avoid being detected by a hook on the more common ResumeThread. The file chosen for hollowing is again the sample itself.

Formbook Sample 8

SHA256: ce87c1d3d6c13090f1ff62725338786d6b7df1423f01e35f78ccb69ed2abd965

Anti-Analysis: Encryptions, Obfuscations

Packed: Compiled obfuscated AutoIt script

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: AutoIt scripting language

Highlight: Scanning for presence of specific anti-malware products

This is a compiled AutoIt script which is heavily obfuscated that contains Formbook's encrypted payload. It checks for the existence of processes Anti-Virus vendor processes (figure 32) – avastui.exe (Avast), ekrn.exe (ESET NOD32) and seccenter.exe (BitDefender). However – it doesn't try to terminate them. It nevertheless performs process hollowing with the upper level APIs (VirtualAllocEx, WriteProcessMemory…) and again the chosen file for the hollowing is the sample itself.


31-Formbook

Figure 30 – Snippet of the AutoIt script


32-Formbook

Figure 31 – Checking if avastui.exe is running

Formbook Sample 9

SHA256:

e2385a399f96a24f98bdeaee0c4a9f6b3e11a61c64d52f55c80ae37a171eac3b

Anti-Analysis: Encryptions, Anti-Sandbox (User interaction), Check for suspicious file names and running processes, Anti-Debug

Packed: No

Process Hollowing: Yes – with chosen file as the sample itself

Programming language: Delphi

Highlight: Check the file path for specific names as part of anti-analysis

This sample copies itself to a file named “essentialize.exe” and executes it.

The newly executed process “essentialize.exe” performs process hollowing with the chosen file as itself to unpack Formbook’s payload. Its injection technique to the hollowed process is slightly different. Instead of using VirtualAllocEx + WriteProcessMemory, it uses NtCreateSection + NtMapViewOfSection to the remote process.

This sample also has a lot Anti-Analysis techniques – it won’t execute essentialize.exe if one of them returns true:

- Check if the cursor was moved during a time interval using GetCursorPos (appeared in previous samples as well)
- Check if the file path contains one of these strings: self. , virus, sample, malware, sandbox

  
  33-Formbook

  Figure 33 – Anti-Analysis checks by file name

- Check if one of the following processes is running: windbg.exe, procexp.exe, ollydbg.exe, procmon.exe, procmon64.exe, procexp64.exe (favorite programs among malware researchers)

  
  34-Formbook

  Figure 34 – Anti-Analysis check of selected processes

- Three Anti-Debug checks:
- Checking for the BeingDebugged flag in the PEB (assembly equivalent of IsDebuggerPresent)

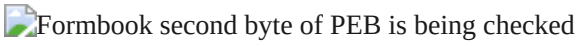
  
  Formbook second byte of PEB is being checked

  Figure 35 – ecx = 0x30, the second byte of the PEB is being checked (BeingDebugged)

- Calling NtQueryInformationProcess with ProcessInformationClass = 0x1F (ProcessDebugFlags)
- Calling NtQueryInformationProcess with ProcessInformationClass = 0x1E (ProcessDebugObjectHandle)

Have a look here If you are interested to learn more about the last two Anti-Debug techniques, which are a bit less common than the first one.

The Anti-Analysis checks are performed both in the original sample and the executed “essentialize.exe” process

Formbook Sample 10

SHA256:

ec74927626eb9a06871380633fbcd296994c99758478262111652253760005e41

Anti-Analysis: Encryptions, Sleep

Packed: Yes – with ASProtect

Process Hollowing: No

Programming language: Delphi

Highlight: No use of process hollowing

This sample is packed with ASProtect. Unpacking reveals that like Formbook sample 9, it was also written in Delphi.

Contrary to the other droppers, it doesn't perform process hollowing or drop another executable. It simply decrypts Formbook's payload from its resource section and executes it. Removing the ASProtect protection, the resource section is revealed to us, it is encoded in base64:

Figure 36 – A base64-encoded resource "DVCLAL" is extracted and contains Formbook's final payload

It also delays execution by calling Sleep, but only for 25 seconds, which is not enough to evade sandboxes.

Summary of Formbook Analysis

We analyzed 10 different samples and discovered lots of tricks used by the latest Formbook variants droppers to evade analysis:

- Anti-Sandbox (User interaction, Delayed execution, Check for loaded modules, Time acceleration checks)
- Search for research tools/anti-malware products/suspicious sample names
- Change of the API calls (e.g: use of NtMapViewOfSection instead of WriteProcessMemory, use of NtAlertResumeThread instead of ResumeThread), Not using NtUnmapViewOfSection
- Anti-Debug
- Use of several programming languages (Visual Basic, Delphi, C#, C++, AutoIt)
- Wide usage of different encryption algorithms, both proprietary and known ones (RC4)
- Obfuscations
- Anti-Memory analysis to avoid detection of reflective loading
- Various packers

The use of so many techniques suggests detecting the Formbook's payload on the disk by signatures is impossible. Behavioral analysis is the only way to fully detect Formbook data-stealing malware and its droppers' malicious activities.

It is unknown to us who is writing and selling Formbook malware combined with these droppers. It might be sold separately or combined. What is certain, a lot of work has been invested in developing these droppers, as they differ so much and employ a lot of different techniques.

Formbook Detection Suggestions

Following the techniques/IOCs we found, here are our suggested way to detect them. It should be taken into consideration that false-positives are also possible.

- Use the hashes (SHA256) we mentioned and blacklist them
- Suspicious file names: essentialize.exe, Rhododendrons8.exe
- Search for execution of mshta.exe – look which script was used to execute it.
- Looks for suspicious command line that has string manipulation (replace method)
- Monitor usage of AutoIt compiled scripts and Delphi compiled files
- Search for processes that create themselves as suspended, then inject PE/shellcode into them and resume – this is very common among malware in general and especially among Formbook's droppers.
- All of the droppers were unsigned – always check if the executable is unsigned
- Keep an eye on the autorun key – especially if it contains executables to run from a temporary folder

Example of formbook detection by Cyberbit's EDR of sample 1

After execution, the following graph will be generated:

Figure 37 – Graph of the first sample on Cyberbit's EDR

Formbook.exe is the dropper as mentioned in the first sample analysis. It drops and executes rhododendrons8.exe and also executes mshta.exe to gain persistency (registry – executable added).

If we look at the event details of mshta.exe, we can see the full script:

Figure 38 – the VBScript executed can be clearly seen

Hod Gavriel is a malware analyst at Cyberbit.

Learn more about Cyberbit EDR Kernel-Based Endpoint Detection vs. Whitelisting

---

Source: https://www.cyberbit.com/blog/endpoint-security/formbook-research-hints-large-data-theft-attack-brewing/