Deep Analysis: FormBook New Variant Delivered in Phishing Campaign – Part III | FortiGuard Labs

By Xiaopeng Zhang

Published: 2021-04-27 · Archived: 2026-04-05 22:49:00 UTC

FortiGuard Labs Threat Research Report

Affected platforms: Microsoft Windows
Impacted parties: Windows Users
Impact: Collect Sensitive Information from Victim's Devices
Severity level: Critical

"FormBook" is a malware designed to steal sensitive information from a victim's device, as well as to receive control commands to perform additional malicious tasks. I have researched the entire campaign launched by a new variant of FormBook and this is the final part of my analysis.

In the two previous posts (Part I, Part II), I explained how the phishing campaign starts from an attached PowerPoint file. Its VBA code, embedded in a Macro, then downloads a PowerShell file and extracts a .Net framework executable file. A FormBook payload file is eventually executed across three .Net modules. I also showed what anti-analysis techniques this FormBook variant uses to confound security researchers. And I revealed the Windows and target processes that are injected with FormBook code to cause them to perform malicious acts on a victim's device, as well the relationship among between these processes.

In this final part, I explain what the tasks are performed once FormBook has injected malicious code into a Windows process (like ipconfig.exe), and the processes of various targets. I explain how inline hooks are set to the target processes from which FormBook steals its victim's sensitive information. I will also demonstrate how that stolen data is sent to C2 server. At finally, I provide the control commands that are used in this variant of FormBook.

FormBook Loaded in Target Processes Sets Inline Hooks to Steal Victim's Input and Data on Clipboard

When FormBook starts in a target process, it loads an ntdll.dll module and then overrides its data with the deployed FormBook malware. This disguises FormBook as an ntdll.dll module when it runs.

Figure 1.1 is a screenshot of the module view in a debugger, with FormBook being loaded in a second ntdll.dll within a target process —in this case, iexplorer.exe.

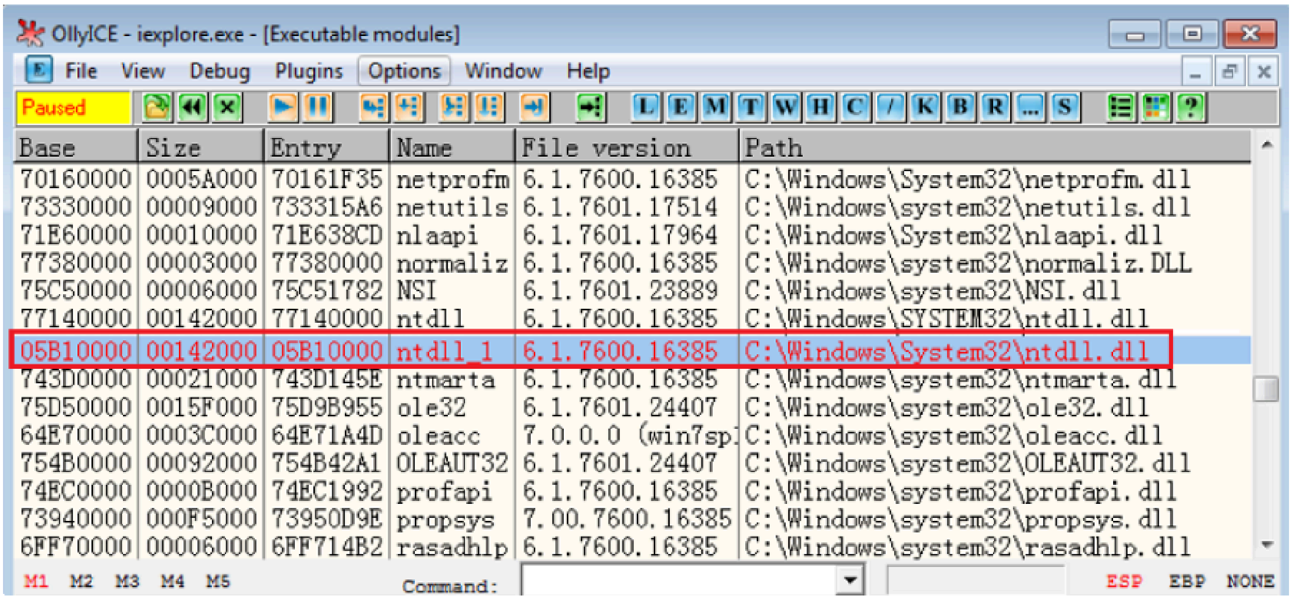

Figure 1.1 – FormBook being disguised as an ntdll module

FormBook then sets inline hooks on a number of APIs to connect them to its local hook functions, such as “GetMessageA()”, “GetMessageW()”, “PeekMessageA()”, “PeekMessageW()”, “SendMessageA()”, and “SendMessageW()”. When one API is set with an inline hook, it calls a function of the ntdll module, which is actually the disguised FormBook malware. This obfuscation is designed to confuse analysts.

FormBook modifies the code of the hooked APIs so it first calls the local hook function within FormBook, and then, after the message has been handled, it returns to its original API routine.

For example, when a victim types inputs to a target process (such as their IE browser), a message (WM_KEYDOWN) is generated. The Windows system then transfers it to the target process, which then calls its “GetMessageA()” or “GetMessageW()” APIs to handle the message. However, because of the inline hook, the API first calls the corresponding local hook function within FormBook, where the message is processed. The message is then sent back from FormBook to the original routine, where it is handled by the target process. As you can see, an inline hook works much like a MITM [(Man in the Middle)]() attack. In Figure 1.2, below, you can see a comparison between the original GetMessaageA() API and the modified one, where “call ntdll_1.05B3FE38” has been inserted to call FormBook’s local hook function.

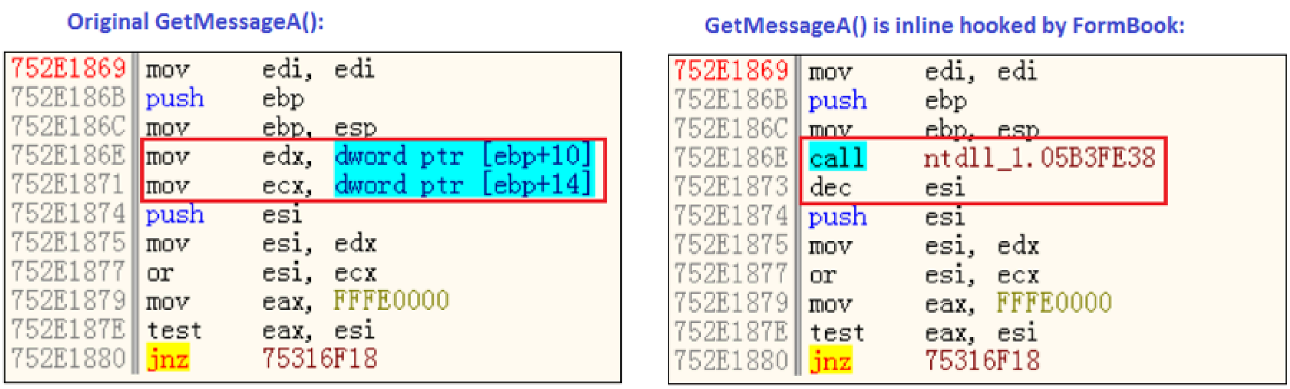

Figure 1.2 – Comparison between the original and inline hooked API GetMessageA()

Local hook functions only focus on WM_KEYDOWN (100H) or WM_SYSKEYDOWN (104H) messages (the key events) for recording victim's inputs, and use WM_LBUTTONDOWN (201H) and WM_RBUTTONDOWN (204H) messages (mouse-generated events) for recording data from Clipboard.

When any above message is matched, the malware calls two APIs, GetForegroundWindow() and GetWindowTextW(), to obtain the title of the target process the victim is actively using. It then records the title and inputs from key press events, as well as records clipboard data for mouse click events, and places them into a shared memory section. Figure 1.3 shows a view of stolen inputs from an IE browser, where I input "www.fortinet.com" and then hit the enter key in its address bar. In addition, Figure 1.4 shows FormBook stealing current data on the system Clipboard when I clicked the left button of my mouse on the IE browser.

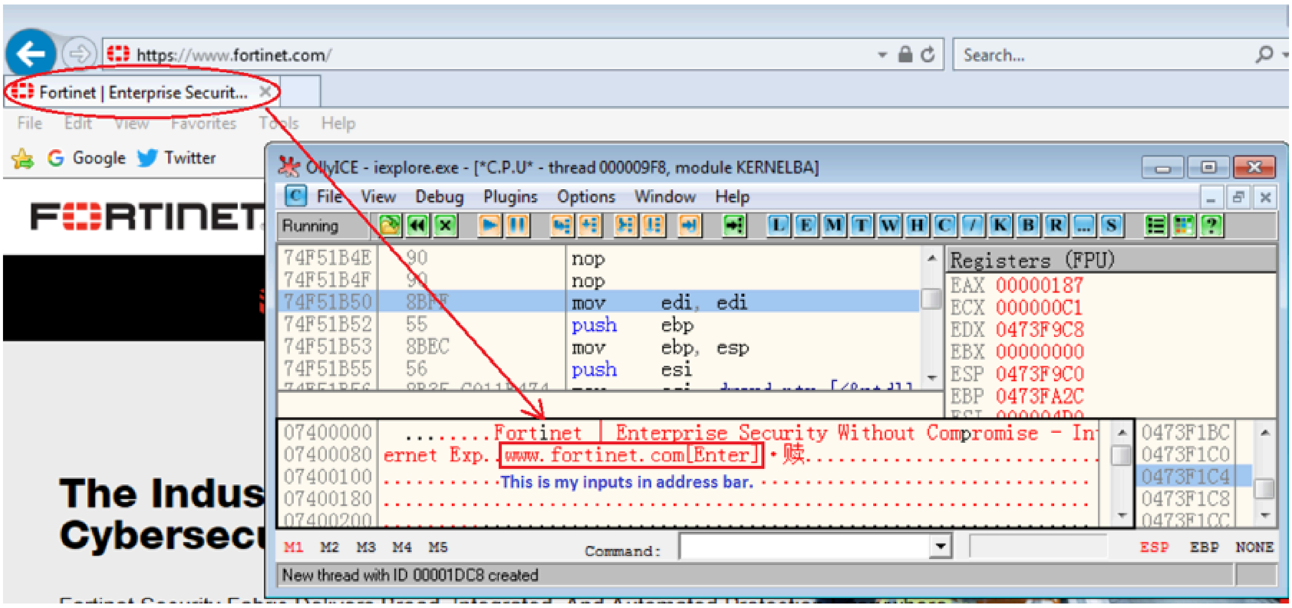

Figure 1.3 – View of the victim's stolen inputs

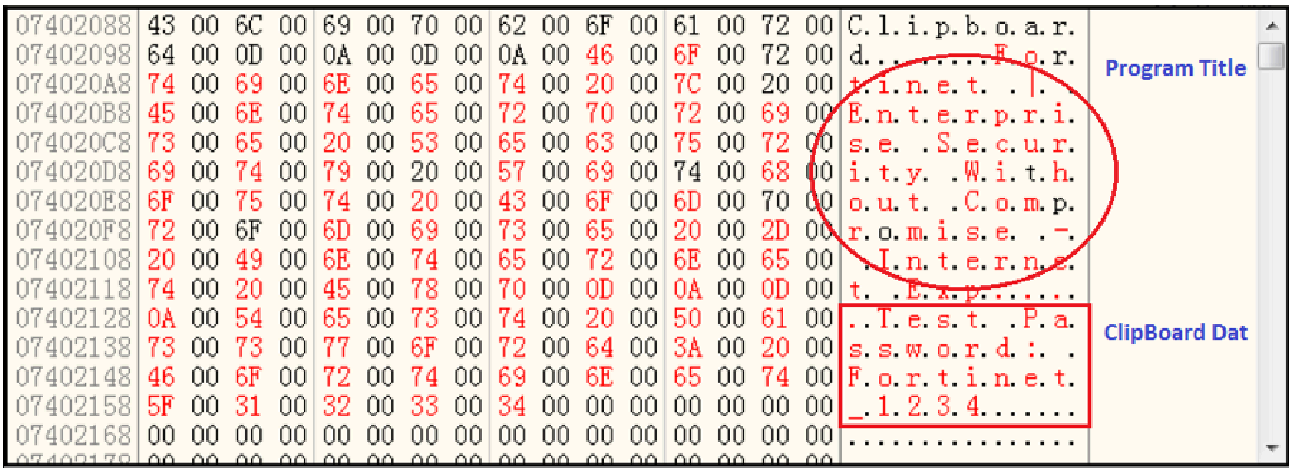

Figure 1.4 – View of stolen Clipboard data on the victim's device

Other than APIs above, the malware also places inline hooks on some socket relevant APIs, such as InternetQueryOptionW(), HttpSendRequestA(), HttpSendRequestW(), SealMessage(), send(), WSASend(), PR_Write()—only for FireFox, InternetConnectA(), InternetConnectW(), and so on. Through these APIs, FormBook is able to collect the network activities of the target processes, such as login credentials.

As you may have noticed, Explore.exe once again appears in the target process list. This second time, the FormBook instance installed within Explorer.exe has a special task, which is to collect stolen data produced by other target processes and then communicate with the C2 server.

The Strategy of FormBook Injected Into Windows Process, Explorer.exe, and Target Processes

Before I explain how FormBook communicates with its C2 server, I need to explain how the three FormBook instances installed in the victim's system work together.

First of all, a large shared memory section (size 9C4000h) is created in a Windows process (such as ipconfig.exe) by calling the API ZwCreateSection(), which is commonly used throughout all FormBook instances, as well as in Windows process, Explorer.exe, and target processes for sharing various data.

1. The FormBook instance inside the Window process (like ipconfig.exe) not only injects FormBook into target processes (including Explorer.exe) but also makes FormBook persistent on the victim's device. It installs its own family folder under "%AppData%" in which it stores stolen recorded data within a number of record files (.ini).

It continuously saves Clipboard and Keylogger data stolen from other target processes into the record files. It also collects browser credential data and Outlook profile data and saves them into other record files. And it has a function for parsing response packets with control commands from the C2 server, which I will discuss it in detail. FormBook performs the above tasks every five seconds.

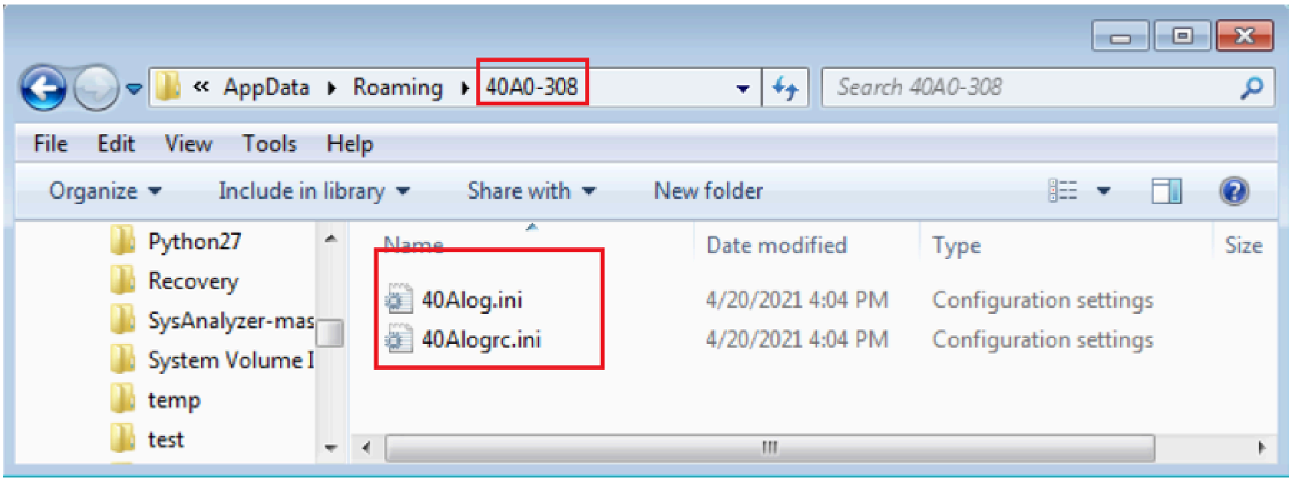

Figure 2.1 – The Family folder of FormBook

The names of the family folder and record files are random. They are generated using the victim's UserName and a string "/dyt/" (a partial of the C2 server's URL). In this case, it created "40A0-308" for the folder name and "40Alog{*}.ini" for the record file name in my test machine. Figure 2.1 shows the family folder in my testing machine.

In Figure 2.2 you see a screenshot of the system registry with the added FormBook instance loaded into the Auto-Run group. It does this after it sends data to C2 server. FormBook is copied into a random string folder under %ProgramFiles% and in a random file name. As a result, FormBook will launch at system startup.

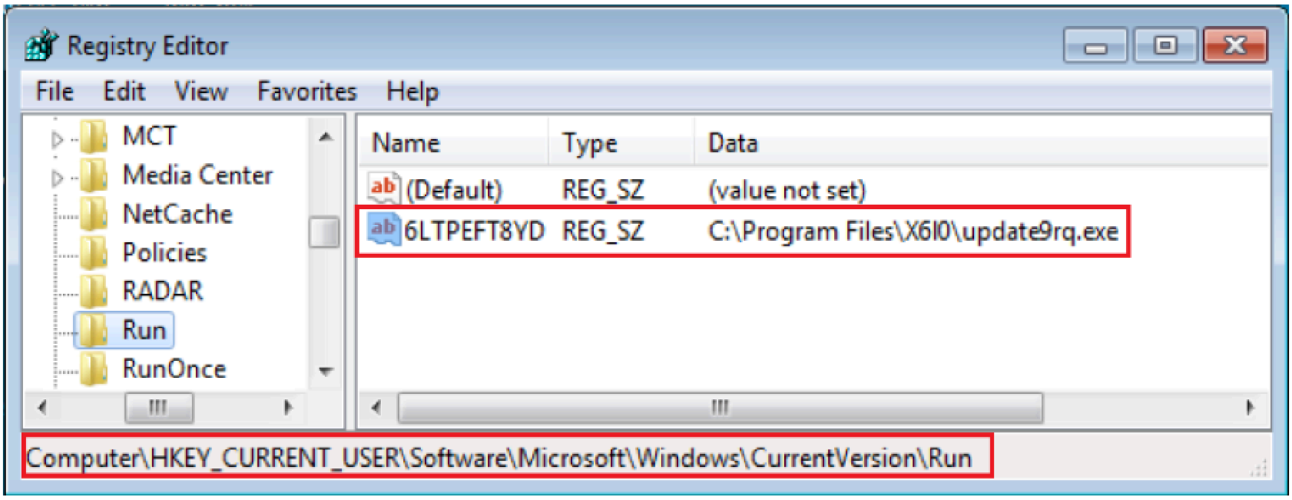

Figure 2.2 – Added Auto-Run item for creating persistence in a victim's device

2. The main responsibility of the FormBook instance injected into the special target process "Explorer.exe" is to communicate with the C2 server using one of its five decrypted functions, with function magic code 4A909090h, specifically designed for that task. It reads the record files within the family folder and sends them to the C2 server, receives responses, and puts them into the large shared memory section. This information is then checked and parsed by the FormBook instance inside the Windows process.

3. The FormBook instance running inside various target processes keeps stealing victim's data (inputs and data of Clipboard) using the inline hooked APIs. It directly copies the data into the large shared memory section, which is then saved into the record files ("*.ini") in the family folder, and is found by the FormBook instance injected into the Windows process (like ipconfig.exe).

For the other inline hooks on network relevant APIs, which I discussed in the previous section, the local hook function directly processes the stolen data and sends the matched data (containing key words "login", "pass", "user", and "auth") to the C2 server. Figure 2.3 shows the ASM code defining the key words.

```
00417D98        push    eax
00417D99        push    edi
00417D9A        mov     [ebp+var_8], 72657375h ; ;; "user "
00417DA1        mov     [ebp+var_4], 20h
00417DA7        mov     [ebp+var_10], 73736170h ; "pass "
00417DAE        mov     [ebp+var_C], 20h
00417DB4        mov     [ebp+var_18], 68747561h ; "auth "
00417DBB        mov     [ebp+var_14], 20h
00417DC1        mov     [ebp+var_20], 69676F6Ch ; "login "
00417DC8        mov     [ebp+var_1C], 206Eh
00417DCE        call    sub_41C030       ; it contains stirng comparison
00417DD3        add     esp, 0Ch
00417DD6        test    eax, eax
00417DD8        jz      short loc_417DE5
00417DDA
```

Figure 2.3 – Key words defined for filtering stolen data from network APIs

Communicating with the C2 Server

There are, in total, sixty-four C2 server host strings in this variant of FormBook. They are decrypted in another decrypted function (magic code 49909090h). The full C2 server host strings are attached in the IOCs section below.

Each FormBook instance injected into a target process randomly picks sixteen hosts as its own C2 server to which it sends stolen data.

Inside Explorer.exe, the injected FormBook instance has a special task running in a thread to monitor the family folder every two seconds. Once a record file is saved by a Windows process (like ipconfig.exe), it reads them one by one and then sends them to the C2 server.

I will use two examples to explain how plaintext data is sent to the C2 server.

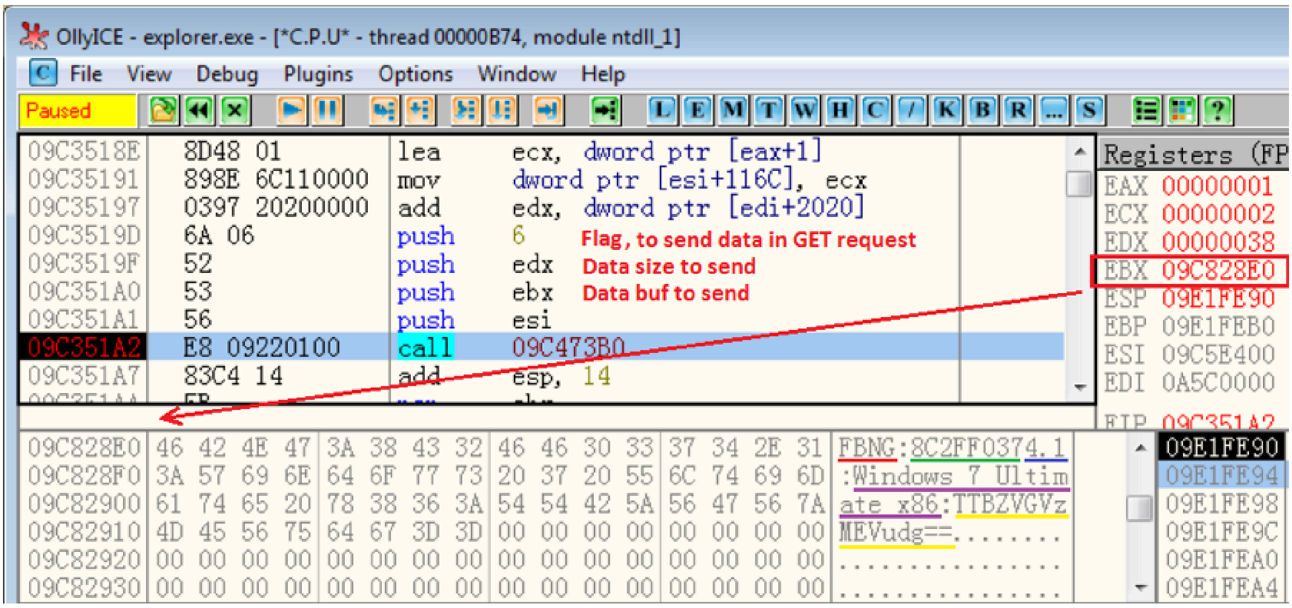

Figure 3.1 – Calling a decrypted function to send basic information to the C2 server

As seen in Figure 3.1, the first packet in the memory section contains the basic information. It calls the decrypted function 09C473B0 (magic code 4A909090h) to encrypt, encode, and send the data to a C2 server.

The basic information is divided by “:”, where “FBNG” is the packet magic code; “8C2FF037” can be thought as a victim ID that is generated from the User’s access token; “4.1” is the hard coded FormBook version; ”Windows 7 Ultimate x86” is the Windows version (and platform on my test machine); the last data is the base64 encoded User name.

Figure 3.2 shows a snippet of ASM code with the hard coded FormBook version “4.1” and the packet magic code “FBNG”.

```
00403C60        mov     [ebp+var_5C], 3Ah
00403C66        mov     [ebp+var_5A], eax
00403C69        mov     [ebp+var_56], eax
00403C6C        mov     [ebp+var_52], eax
00403C6F        mov     [ebp+var_4E], ax
00403C73        mov     [ebp+var_88], 31h ; 1
00403C7C        mov     [ebp+var_86], eax
00403C82        mov     [ebp+var_82], eax
00403C88        mov     [ebp+var_7E], eax
00403C8B        mov     [ebp+var_7A], ax
00403C8F        mov     [ebp+var_78], 312E34h ; "4.1"   FormBook version.
00403C96        mov     [ebp+var_74], eax
00403C99        mov     [ebp+var_70], eax
00403C9C        mov     [ebp+var_6C], eax
00403C9F        mov     [ebp+var_24], 474E4246h ; "FBNG"
00403CA6        mov     [ebp+var_20], 3Ah ; ":"
00403CAC        mov     [ebp+var_1E], eax
00403CAF        mov     [ebp+var_1A], eax
00403CB2        mov     [ebp+var_16], ax
00403CB6        add     edi, 2000h
00403CBC        call    sub_40CEF0
00403CC1        add     esp, 38h
```

Figure 3.2 – Code view of hard coded FormBook version

This decrypted function performs the following actions to send data to a C2 server:

- Randomly selects a C2 host string from the 16 loaded C2 hosts.
- Determines whether to use the POST or GET method. Because the flag parameter is set to 6 (refer to Figure 3.1), it will send this data using the “GET” method.
- Encrypts the data with an RC4 algorithm, whose encryption key (saved in ConfigObj) was generated from a combination of the selected C2 sever host string and the constant string “/dyt/”.
- Base64 encodes the RC4 encrypted data, and if the flag is not 6 or 10, it replaces the symbols within the base64 encoded data.
- Combines the above data into a GET or POST packet to send to a C2 server.

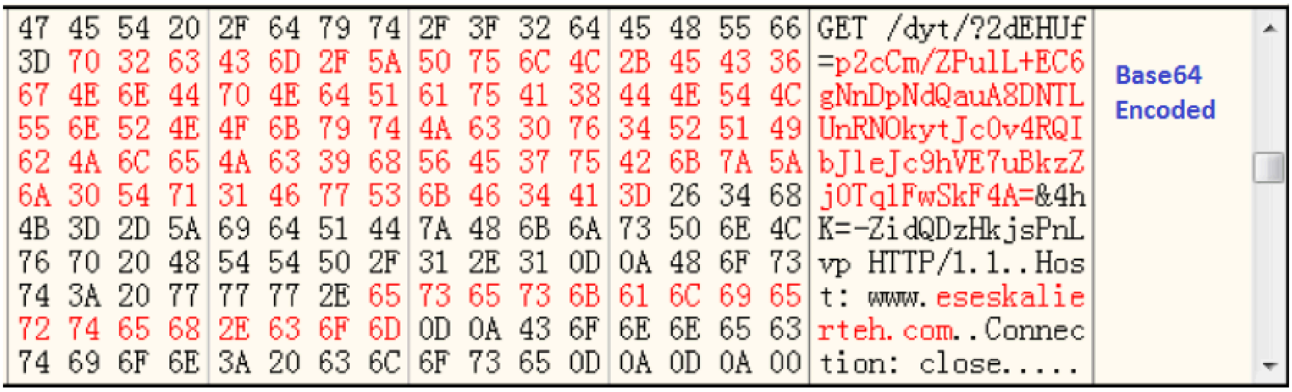

Figure 3.3 – A complete GET packet about to be sent

Figure 3.3 is a GET packet. It simulates reporting data with a URL, as it starts with "/dyt/?". The following "dEHUf=" uses a random string as the parameter name. The value for this parameter is the RC4 encrypted basic information after being base64 encoded. It also has another parameter, "4hK=" and value, which are a randomly generated faked pair to make the URL look normal.

Another example is for reporting data in a POST packet. The steps are same as for the GET packet, but the final data (RC4 encrypted and base64 encoded) is sent in the body. In addition, the base64 encoded data is additionally transformed by replacing the symbols (symbols like "+" , "/", and "=" are transformed to others in a special way) to make base64 decoding performed by other people (such as analysts) fail.

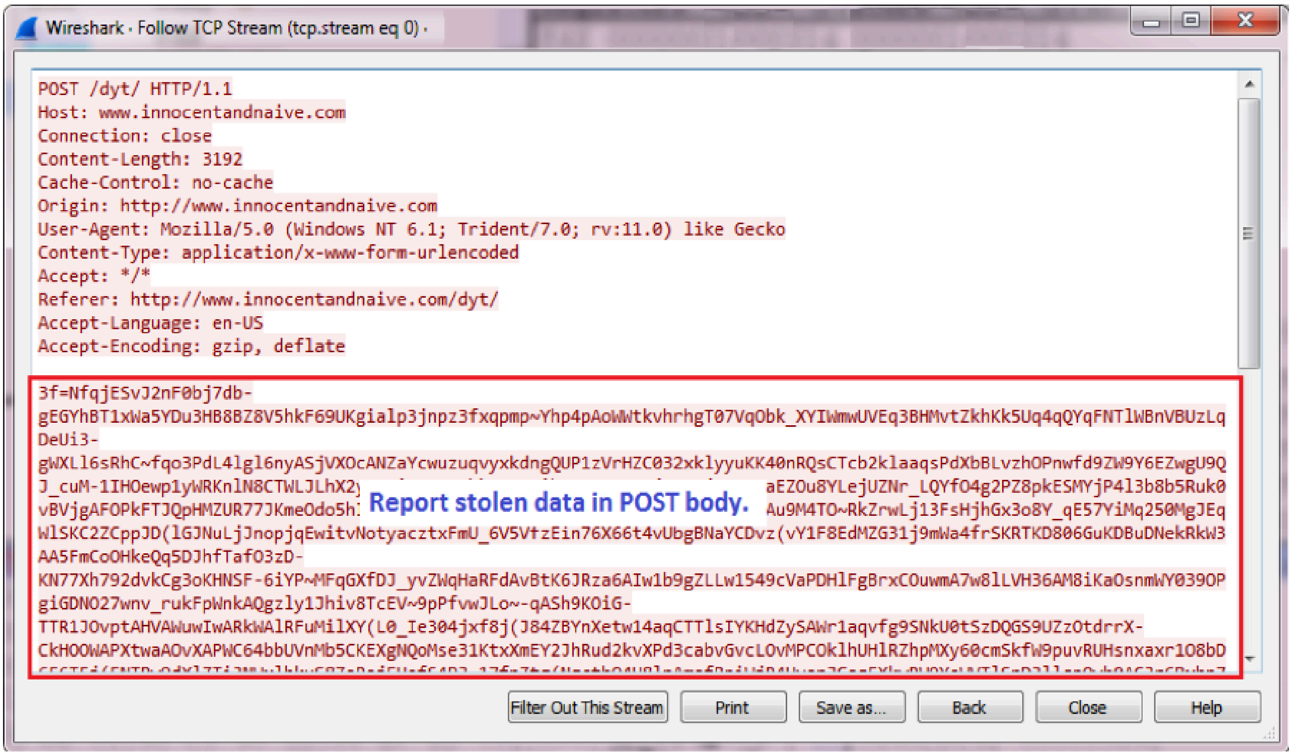

Figure 3.4 – Report of stolen Outlook profile data with saved credentials

The Wireshark screenshot, shown in Figure 3.4, is of FormBook reporting MS Outlook profile data and credentials stolen from my testing machine within the body component of a POST request (The flag parameter was set to 3 for the decrypted function 09C473B0 in Figure 3.1). The URL element is only "/dyt/" and the data is enclosed in a key/value pair, where the key “3f” is a random string; the value is RC4-encrypted and base64-encoded stolen data and is finally transformed.

Control Commands

Although the C2 servers did not work well during my analysis, I was able to perform a static code analysis as well as simulate the C2 packet to analyze its features. According to my analysis, and comparison with previous analysis (you can see it in References section) for FormBook, the control commands have not changed a lot in this new variant. Please follow below to observe what features the control commands offer and how.

There is a special function—that I call handle_control_command()—in the FormBook instance that is injected into a running Windows process (like ipconfig.exe). It is used to handle a C2’s response packet copied into the large shared memory session by the FormBook instance running in Explorer.exe. Figure 4.1 shows the pseudocode of this function. The code flow is very clear and easy to read and understand.

```
signed int __usercall handle_control_command@<eax>(size_t a1@<edi>, int ConfigObj, _BYTE *a3)
{ ●●●
  Shared_memory_session = *(_DWORD *)(ConfigObj + 0x7D8);
  v19 = 0x20303032;
  v20 = 0xD4B4F;
  memset_(&v21, 0, 56);
  if ( !unicode_string_compare((char *)&v19, Shared_memory_session + 0x743D, 6u)// Check if received packet status is "200 OK".
    && *( DWORD *)(Shared_memory_session + 0x7434) != 0x474E4246 )
  { return 0; }
  ●●●           // Rc4 decrypts data followed the magic "FBNG" and command number  in response packet.
  RC4_Decrypt(*(_DWORD *)(ConfigObj + 0xB38), v25, Shared_memory_session + 0x6C14);
  Control_Command = *(_BYTE *)v8;
  if ( *(_BYTE *)v8 != '3' )
  {
    if ( Control_Command == '5' )
    {
      if ( **(_DWORD **)(ConfigObj + 0xB38) == 0x474E4246 )// "FBNG"
      { cmd_5(ConfigObj); }
      return 0;
    }
    if ( Control_Command == '6' )
    {
      if ( **( DWORD **)(ConfigObj + 0xB38) == 0x474E4246 )// "FBNG"
      { cmd_6_7(ConfigObj, 0x12); }
      return 0;
    }
    if ( Control_Command == '7' )
    {
      if ( **(_DWORD **)(ConfigObj + 0xB38) == 0x474E4246 )// "FBNG"
        cmd_6_7(ConfigObj, 0x18);
      return 0;
    }
    if ( Control_Command == '8' )
    {
      if ( **(_DWORD **)(ConfigObj + 0xB38) == 0x474E4246 )// "FBNG"
        cmd_8(ConfigObj);
      return 0;
    }
    ●●●
    if ( Control_Command == '4' )
      return cmd_4(ConfigObj, *(_DWORD *)(ConfigObj + 0xB38), v23);
    v18 = v23;
    ●●●
    if ( Control_Command == '1' )
      return cmd_1(ConfigObj, *(_DWORD *)(ConfigObj + 0xB38), v23);
    if ( Control_Command == '2' && cmd_2(&v22, *(_DWORD *)(ConfigObj + 0xB38)) )
      return sub_404B20(ConfigObj, *(_DWORD *)(ConfigObj + 0xB38), v18);
    if ( *(_BYTE *)v8 == '9'  )
      return cmd_9(ConfigObj, *(_DWORD *)(ConfigObj + 0xB38), v18);
    ●●●
  }                                    // "FBNG"
  if ( **(_DWORD **)(ConfigObj + 0xB38) == 0x474E4246 )// Handle command '3' here.
  {
    *(_DWORD *)(Shared_memory_session + 0x6890) = 1;
    cmd_3(ConfigObj);
    sub_40D120(ConfigObj, &a3, *(_DWORD *)(ConfigObj + 0xB54), 1);
    sub_41A0A0(ConfigObj, 0);
  }
  return 0;
}
```

Figure 4.1 – Structure of the function handle_control_command()

FormBook offers nine commands—named “1”, ”2”, ”3”, ”4”, ”5”, ”6”, ”7”, ”8”, and ”9”—to control a victim’s device. Each command starts with the magic code “FBNB” with a one-byte command number followed by the encrypted command content.

Command “1” (0x31):

FormBook extracts a PE file from decrypted command data with a random name under "%Temp%" and the runs it by calling the API ShellExecuteA().

Command “2” (0x32):

This command is used to upgrade FormBook on the victim’s device. It calls function for Command “3” to uninstall FormBook. An executable file is then extracted from a decrypted packet into the “%Temp% folder and executed by calling CreateProcessInternalW(). It finally exits the current FormBook instance running in the Windows process (like ipconfig.exe).

Command “3” (0x33):

This command uninstalls the current version of FormBook from the victim’s device. To do this, it kills the Explorer.exe process running the FormBook instance ,removes the item from Auto-run in the system registry by calling the API NtDeleteValueKey(), deletes the FormBook executable files ("C:\Program Files\X6l0\update9rq.exe" in this example), and eventually exits the Windows process (like ipconfig.exe) where the current FormBook instance is injected.

Command “4” (0x34):

This command calls the API ShellExecuteA() to execute a given command from the decrypted packet.

Command “5” (0x35):

Command 5 deletes the files ".sqltie" and "Cookies" and searches for three main user data paths.

Command “6” (0x36) and 7 (0x37):

This command calls the API ExitWindowsEx() with the parameter **EWX_POWEROFF** (command 7) or **EWX_REBOOT** (command 6) to power off or reboot the victim’s device.

Command “8” (0x38):

This one asks FormBook to add itself to the Auto-Run group, sends all stolen data (including Keylogger, data from Clipboard, and credentials from browsers, email clients, etc.) to the C2 server.

Command “9” (0x39):

This command extracts a ZIP file from the beginning of the decrypted packet until reaching the magic code "FBNG" into a {random name}.zip file under the %Temp% folder. It then unzips it to the same folder. It might work together with the command 4, where it executes a command line to execute the unzipped file.

Conclusion on FormBook Phishing Campaign

In this final part of my analysis, I have focused on how FormBook performs its malicious actions on a victim's device, including what data it is able to steal, such as victim inputs, data on the system Clipboard, saved credentials in browsers and other client software. I also explained how stolen data is encrypted, encoded, and sent to the C2 servers. along with examples to demonstrate how it is done. And finally, I have defined the control commands that FormBook uses, as well as the extended set of functions FormBook can perform with these commands.

In part I, I demonstrated this FormBook variant's infection process. It started with a phishing email with a PowerPoint file attached to it. The VBA code inside the file downloads a PowerShell file and executed. It then extracts a .Net program (item3.jpg) into %temp% folder and executes. After that, FormBook is injected into a newly-created process, "AddInProcess32.exe", through three .Net modules, as displayed in Figure 5.1.

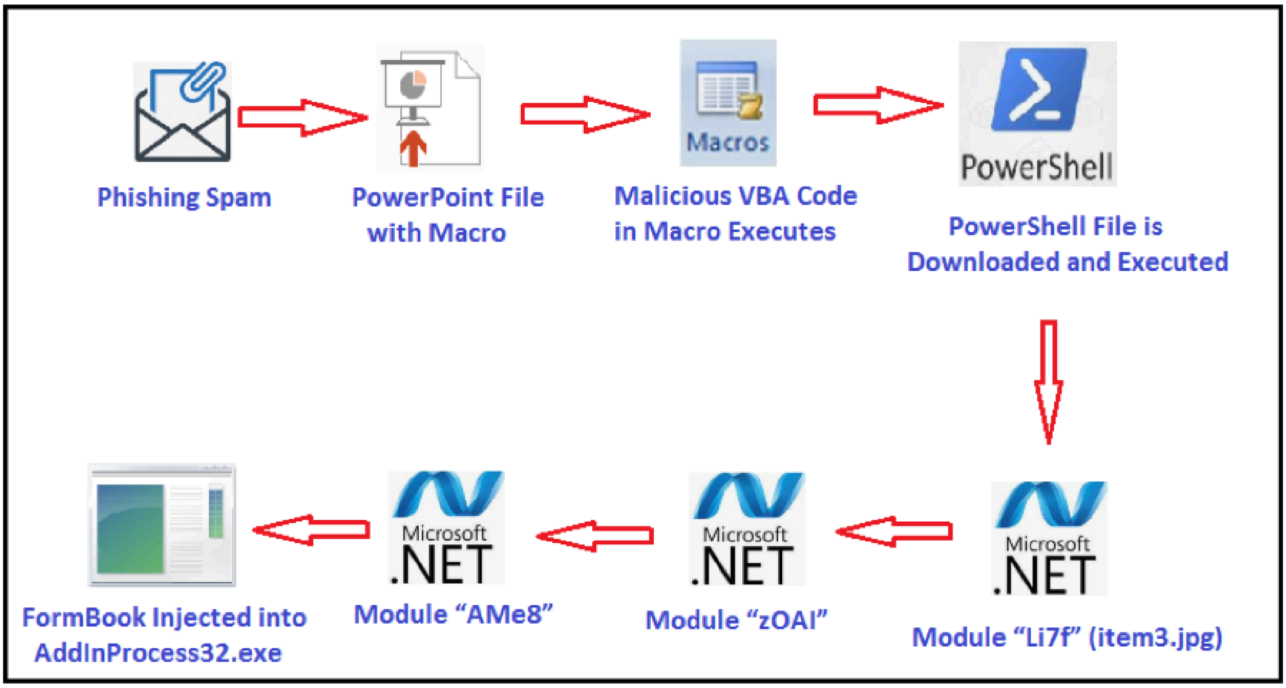

Figure 5.1 – FormBook infection process.

In part II, I elaborated on how FormBook started its work inside "AddInProcess32.exe", what anti-analysis techniques it used, and what Windows processes and target processes it deployed to perform malicious tasks on a victim's device. Figure 5.2 shows how a FormBook instance is deployed into other processes and the relationship between them.

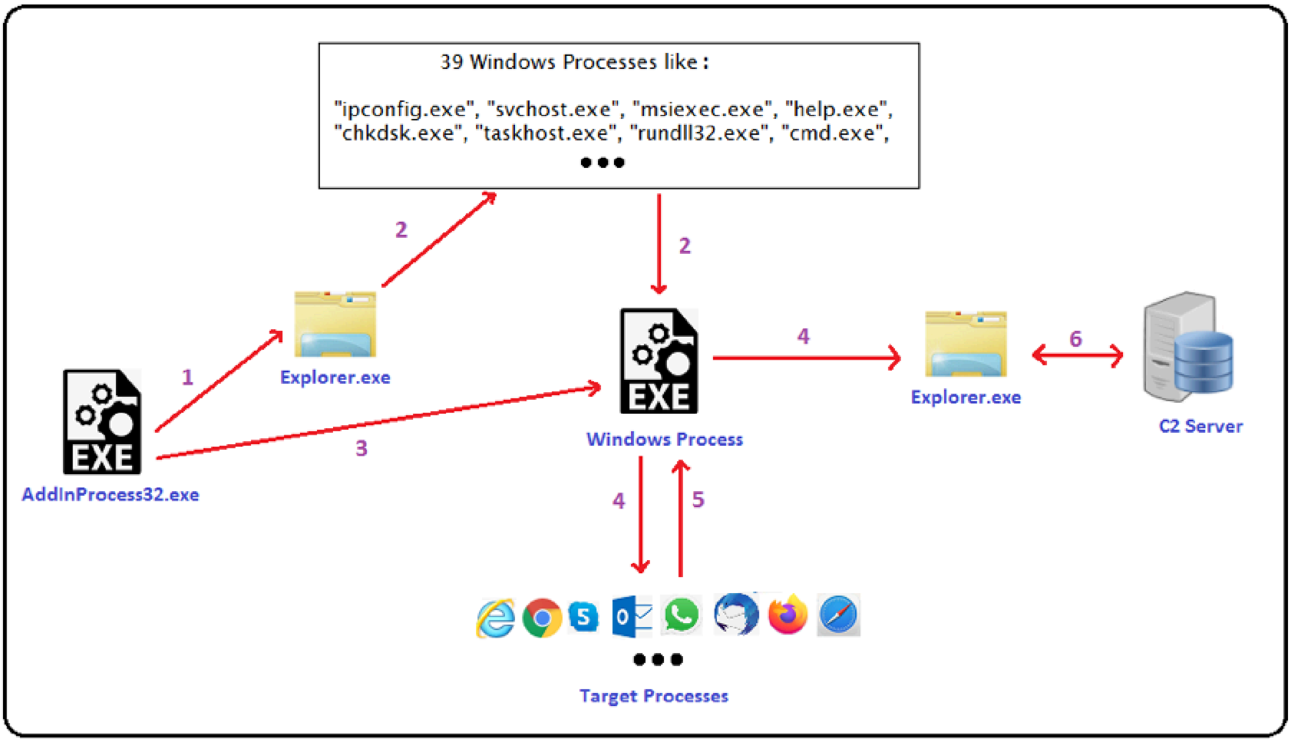

Figure 5.2 – How FormBook deploys itself into other processes.

Fortinet Protections

Fortinet customers are already protected from this FormBook variant with FortiGuard’s Web Filtering and AntiVirus services, as follow:

The attached PowerPoint file is detected as “**VBA/FormBook.C393!tr**” and the “item3.jpg” file is detected as "**MSIL/FormBook.ZXL!tr**" and blocked by the FortiGuard AntiVirus service.

The download URL launched from the PowerPoint sample is rated as "**Malicious Websites**" by the FortiGuard Web Filtering service.

FortiSandbox is able to detect the PowerPoint sample as malicious.

Besides these, Fortinet customers are also protected with FortiGuard’s Web Filtering. The C2 server hosts within this variant of FormBook are already rated as "**Malicious Websites**" by the FortiGuard Web Filtering service.

We also suggest that readers go through the free NSE training — NSE 1 – Information Security Awareness, which has a module on Internet threats designed to help end users learn how to identify and protect themselves from phishing attacks.

Fortinet’s Phishing Simulation Service, FortiPhish, can also be used to proactively test the susceptibility of your organization to these kinds of phishing attacks.

IOCs:

C2 Server URLs:

Numerous websites have been hacked and are being used as C2 servers. Because the list of impacted websites is constantly being updated, we will not attempt to list them all here. Organizations should search for "/dyt/" appended after domain names and remove individual web site entries. For example: "www[.]domainname[.]com/dyt/"

References:

https://www.virusbulletin.com/virusbulletin/2019/01/vb2018-paper-inside-formbook-infostealer/

Learn more about FortiGuard Labs threat research and the FortiGuard Security Subscriptions and Services portfolio.

Learn more about Fortinet's free cybersecurity training initiative or about the Fortinet NSE Training program, Security Academy program, and Veterans program.
