Vidar - payload inspection with static analysis

By map[name:Alessandro Strino]

Published: 2023-10-25 · Archived: 2026-04-06 00:21:52 UTC

Behind this post

Through this blogpost I'm going to talk about one of the latest Vidar samples that I had a chance to analyze. The payload is actually part of a campaign delivered in July 2023 using PEC mails and this analysis comes from a post related to Cert-Agid in the same period. Even if the payload seems to be out of time, it's still a valid example for further analysis of more recents ones.

The purpose of this article is to give an overview of Vidar, helping people that are tracking this threat to properly deal with it. Moreover, it is also an excuse **to tweak a little bit with IDA to show a possible solution related to common issues when we are dealing with highly obfuscated malware**.

Static Analysis

Opening up the Vidar sample with IDA, it's immediately clear that it contains few obfuscated strings and garbage code that prevents analysts from directly examining the sample. More precisely, it has been possible to discover three functions, analyzed in this blogpost, that are in charge of:

- **Detecting VMs execution**;
- **Detecting "default settings"**;
- **Decrypting Strings**.

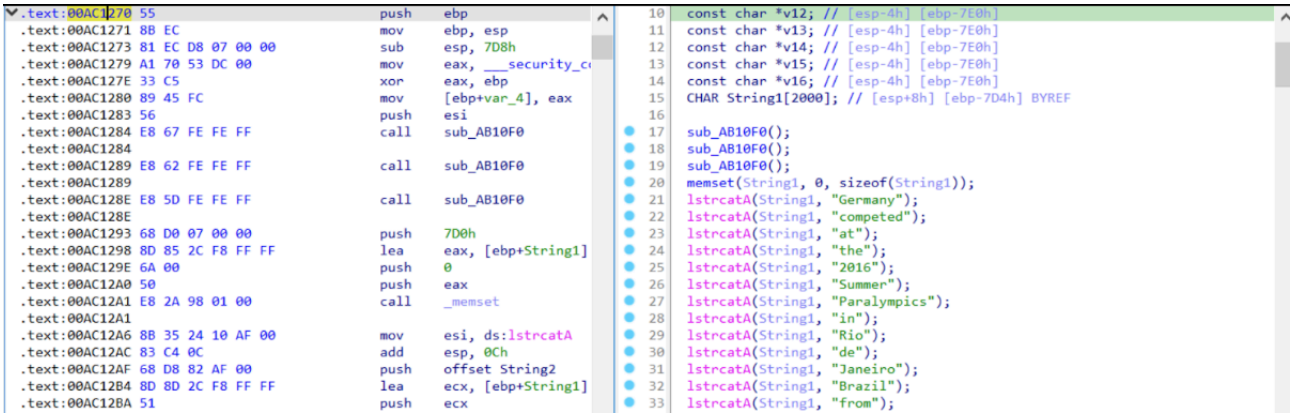

Figure 1 - Vidar main function with garbage code

Anti-Analysis implementation

In this sample, there are three main functions that are in charge of performing anti-analysis checks.

The first one is implemented through the function **VirtualAllocExNuma** that checks if the sample is running on a system with one or more physical CPU:

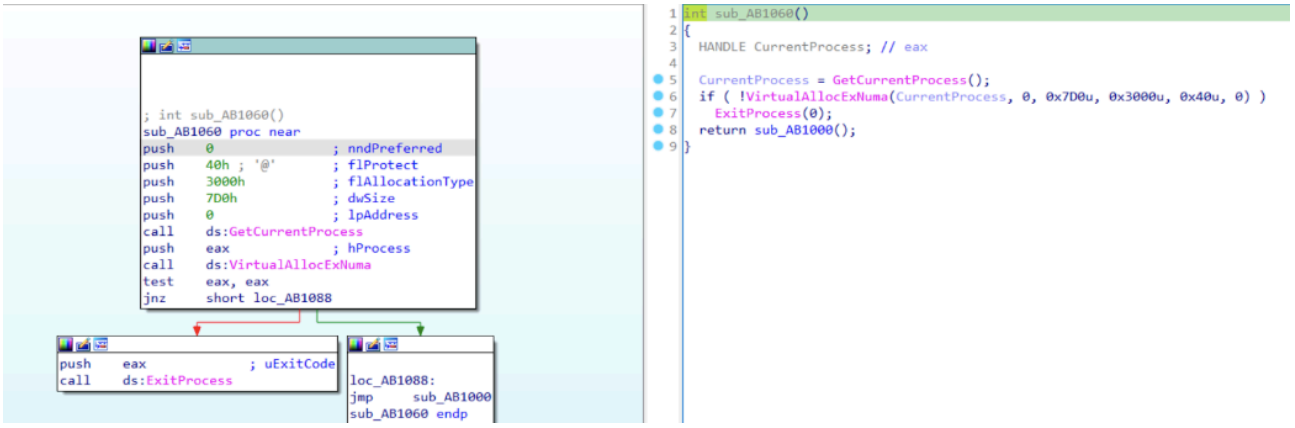

Figure 2 - Call to VirtualAllocExNuma for physical CPU controls.

Another techniques that prevent payload execution is related to **the number of processors available on the machine** that are required to be at least 2:

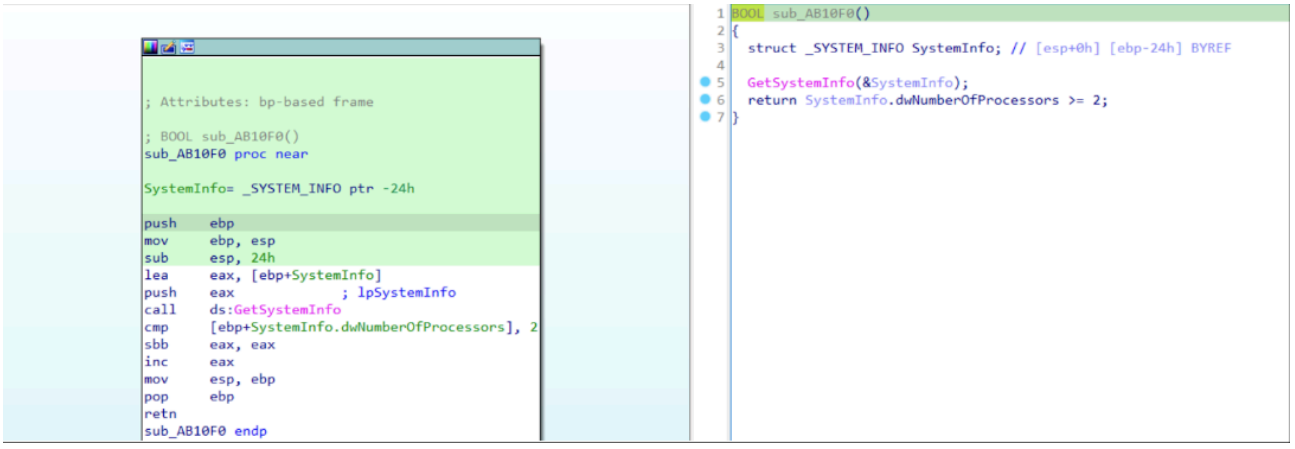

Figure 3 - Call to GetSystemInfo for Processors's checks

The last checks that have been identified are related to the Username and Computer Name that is currently used. In particular there are two matches that verify if the username corresponds to **John Doe** and then the **ComputerName** is equal to **HAL9TH**.

> It turns out that Microsoft Defender's Sandbox computername is HAL9TH, so, you can check for the computer name in your malware before detonation, if the name matches HAL9TH, it means you're inside defender's sandbox, so you can make your program exit.

```
125  default_ComputerName = STR_HAL9TH_0;
126  ComputerName = mw_wrapperGetComputerName();
127  if ( !strcmp(ComputerName, default_ComputerName) )
128  {
129    default_UserName = STR_JohnDoe_0;
130    UserName = mw_wrpperGetUserName();
131    if ( !strcmp(UserName, default_UserName) )
132      PTR_ExitProcess(0);
133  }
```

Figure 4 - Checking for “specific settings”

If one of those checks fails, the payload will call the function **ExitProcess(0)** terminating its execution.

Decryption routine

As already mentioned, Vidar payload contains few encrypted strings to slow down the analysis and probably to evade few monitoring solutions. Because of that, there is a function that is in charge to retrieve the plaintext associated with each encrypted string.

```
5  dword_DC6E74 = sub_AB1110((int)&unk_AF1590, "INBILE", 6u);
6  dword_DC6FDC = sub_AB1110((int)"y9Z9|\"#", "3V2W8MF", 7u);
7  lpProcName = sub_AB1110((int)&unk_AF1560, "TIZVRCE65EQC", 0xCu);
8  dword_DC6F58 = (int)sub_AB1110((int)&unk_AF1548, "Q75PYJUY", 8u);
9  dword_DC6C48 = sub_AB1110((int)&unk_AF1528, "WH0QG0WMJG6X0X", 0xEu);
10 dword_DC6ADC = (int)sub_AB1110((int)&unk_AF1518, "EAF61", 5u);
11 dword_DC681C = (int)sub_AB1110((int)&unk_AF14F8, "WKNMIVF5G7GDN", 0xDu);
12 dword_DC6FFC = (int)sub_AB1110((int)&unk_AF14E0, "OYSMSIP1Q2X", 0xBu);
13 dword_DC69BC = (int)sub_AB1110((int)&unk_AF14B8, "ZI2L225J2S8OI4R5A", 0x11u);
14 dword_DC6B1C = (int)sub_AB1110((int)&unk_AF1490, "5POCCSJCE2R7RAFR0X", 0x12u);
15 dword_DC6BF8 = (int)sub_AB1110((int)"oZA7;\"[\rX=%(", "933CNC7L4QJK", 0xCu);
16 dword_DC6F14 = (int)sub_AB1110((int)&unk_AF1458, "RBMP5AOUWZS", 0xBu);
17 dword_DC6D9C = (int)sub_AB1110((int)"&*5</&#[a", "JYANLKS26", 9u);
18 dword_DC6890 = (int)sub_AB1110((int)&unk_AF1428, "SHRCG3BMF0", 0xAu);
19 dword_DC6FD0 = (int)sub_AB1110((int)&unk_AF1400, "Y324LQUCEECFAKKU", 0x10u);
20 dword_DC6E80 = (int)sub_AB1110((int)&unk_AF13E0, "ONYVBMXN1KTK", 0xCu);
21 dword_DC6DA0 = (int)sub_AB1110((int)&unk_AF13C0, "IB8CO5BZQ3Z7", 0xCu);
22 result = sub_AB1110((int)&unk_AF13A0, "1INX4NGH2WPY", 0xCu);
23 lpLibFileName = result;
24 return result;
25}
```

Figure 5 - Encrypted Strings

The function it’s fairly easy to spot especially observing the number of times it will be called and its signature (that recall a quite simple decryption procedure):

- **decryption_routine**(encrypted_string , key , length)

As expected the decryption routine it’s not so hard to understand, in fact it iterates over the key length and performs an **XOR** operation between the **encrypted_string** and **key** parameters.

```
iterator = 0;
if ( encrypted_string_len )
{
  v6 = a1 - (_DWORD)decrypted_string;
  do
  {
    sub_AD0CA0(String);
    sub_AD0CA0(String);
    lstrlenA(String);
    strlen(String);
    decrypted_string[iterator] = decrypted_string[iterator + v6] ^ key[iterator % strlen(key)];
    sub_AD0CA0(String);
    sub_AD0CA0(String);
    lstrlenA(String);
    strlen(String);
    decrypted_string = v7;
    ++iterator;
  }
  while ( iterator < encrypted_string_len );
}
```

Figure 6 - Decryption routine

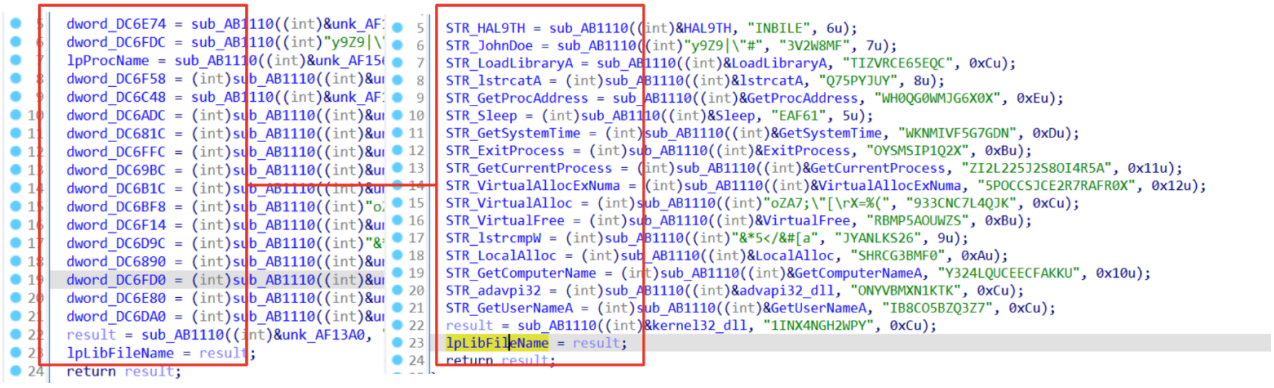

Figure 7 - String decrypted

It’s worth noting that IDA has few limitations, in fact sometimes it does not perform the proper variable renaming and due to the obfuscation implemented few instructions could be misinterpreted. Because of that an effective method to **keep track of decrypted variables is to locate their offset and append a comment**.

In that case, we should have a nice reference that could be used later on, to rename the variable accordingly. Keeping that in mind, it’s possible to speed up our analysis by writing an IDA-python script that takes care of those strings.

Fixing Functions

As mentioned above, IDA sometimes could be confused by obfuscation that could lead to mis-interpret instructions or inhibit its ability to recognize a function. In fact, **at the end of main** there is a jump to a location that is not currently interpreted as a function. However, looking at strings and references to that text section there is clearly an error from the IDA interpreter.

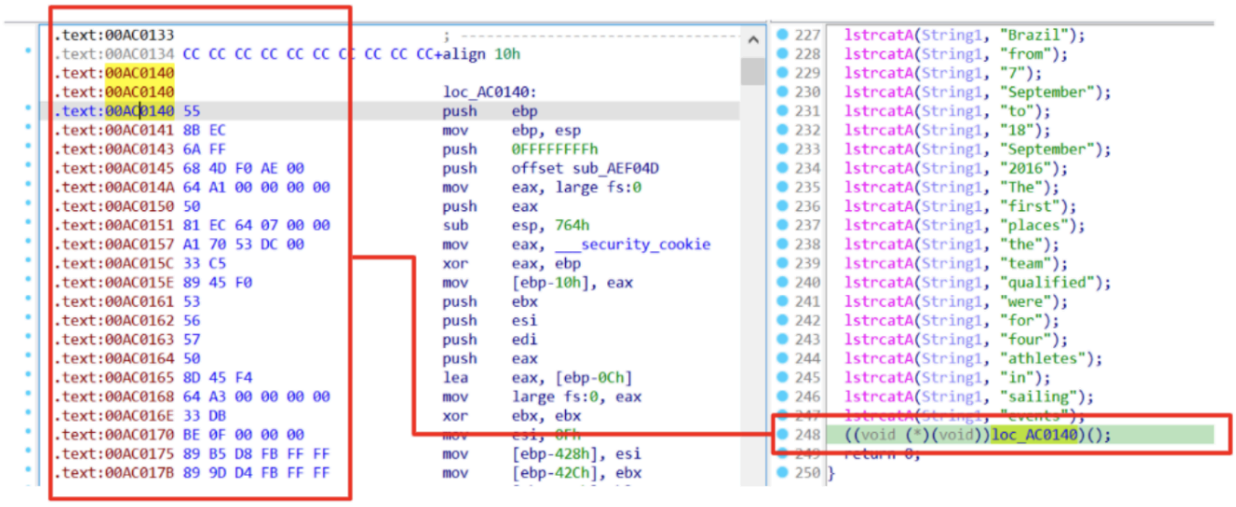

Figure 8 - Mis-intepreted function

To fix that, it's possible to select the block of code and force IDA to treat that as a function. However, this practice it's not always painless. In fact, it's still possible that we could get some issues from IDA that are not capable of interpreting all code correctly. An example is given from the figure below, where we have strings related to **JUMPOUT** and **MEMORY**.

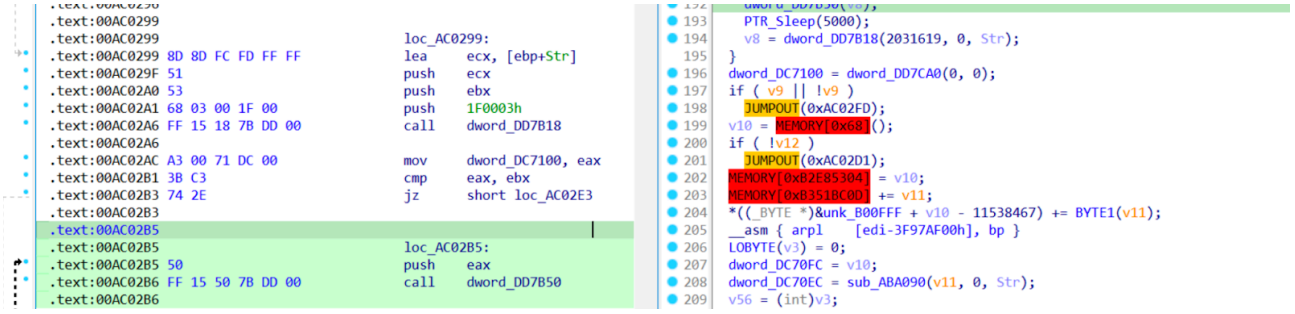

Figure 9 - Function interpreted as data

This issue could be solved easily by fixing the byte related to the **JUMPOUT** instruction, however, in order to avoid losing focus on our main tasks, this issue will probably be discussed in a dedicated thread.

Nevertheless, we have now all pieces to complete our static analysis and go deep in all malicious activities related to this malware.

Additional Analysis

String decryption was an effective method to extract IOCs from this Vidar sample. Examinig those strings we could see that, as expected, it works as an InfoStealer querying browser information (credentials on local storage) and multiple installed programs. At the time of writing, it supports most of the main used browsers, such as: Chrome, Firefox, Opera, Tor, etc.

Another interesting feature is related to the **chrome extension** checks feature, that aims to verify if specific extensions are actually installed. Mainly monitored extensions are related to **crypto wallets** and **password**

managers.

```
51  sub_AC7090("pnndplcbkakcplkjnolgbkdgjikjednm", "Tronium", folder_path, a2, 1, a3, 1, 0, 0);
52  sub_AC7090("egjidjbpglichdcondbcbdnbeeppgdph", "Trust Wallet", folder_path, a2, 1, a3, 1, 0, 0);
53  sub_AC7090("aholpfdialjgjfhomihkjbmgjidlcdno", "Exodus Web3 Wallet", folder_path, a2, 1, a3, 1, 0, 0);
54  sub_AC7090("jnlgamecbpmbajjfhmmmlhejkemejdma", "Braavos", folder_path, a2, 1, a3, 1, 0, 0);
55  sub_AC7090("kkpllkodjeloidieedojogacfhpaihoh", "Enkrypt", folder_path, a2, 1, a3, 1, 1, 1);
56  sub_AC7090("mcohilncbfahbmgdjkbpemcciiolgcge", "OKX Web3 Wallet", folder_path, a2, 1, a3, 1, 0, 0);
57  sub_AC7090("epapihdplajcdnnkdeiahlgigofloibg", "Sender", folder_path, a2, 1, a3, 1, 0, 0);
58  sub_AC7090("gjagmgiddbbciopjhllkdnddhcglnemk", "Hashpack", folder_path, a2, 1, a3, 1, 0, 0);
59  sub_AC7090("kmhcihpebfmpgmihbkipmjlmmioameka", "Eternl", folder_path, a2, 1, a3, 1, 0, 0);
60  sub_AC7090("bgpipimickeadkjlklgciifhnalhdjhe", "GeroWallet", folder_path, a2, 1, a3, 1, 0, 0);
61  sub_AC7090("phkbamefinggmakgklpkljjmgibohnba", "Pontem Wallet", folder_path, a2, 1, a3, 1, 0, 0);
62  sub_AC7090("ejjladinnckdgjemekebdpeokbikhfci", "Petra Wallet", folder_path, a2, 1, a3, 1, 0, 0);
63  sub_AC7090("efbglgofoippbgcjepnhiblaibcnclgk", "Martian Wallet", folder_path, a2, 1, a3, 1, 0, 0);
64  sub_AC7090("cjmkndjhnagcfbpiemnkdpomccnjblmj", "Finnie", folder_path, a2, 1, a3, 1, 0, 0);
65  sub_AC7090("aijcbedoijmgnlmjeegjaglmepbmpkpi", "Leap Terra", folder_path, a2, 1, a3, 1, 0, 0);
66  sub_AC7090("fiedbfgcleddlbcmgdigjgdfcggjcion", "Microsoft AutoFill", folder_path, a2, 0, a3, 1, 0, 0);
67  sub_AC7090("nngceckbapebfimnlniiiahkandclblb", "Bitwarden", folder_path, a2, 0, a3, 1, 0, 0);
68  sub_AC7090("fmhmiaejopepamlcjkncpgpdjichnecm", "KeePass Tusk", folder_path, a2, 0, a3, 1, 0, 0);
69  sub_AC7090("oboonakemofpalcgghocfoadofidjkkk", "KeePassXC-Browser", folder_path, a2, 0, a3, 1, 0, 0);
70  return sub_AC7090("nngceckbapebfimnlniiiahkandclblb", "Bitwarden", folder_path, a2, 0, a3, 1, 0, 0);
```

Figure 10 - Monitored chrome extensions

Network Communication

According to the examined functions related to the network communication, it is possible to recreate the POST request structure that could be monitored and used as an indicator of compromise of this actor:

- **Content Disposition: form-data; name=<Vidar_parameter>**

It’s worth mentioning that parameters observed are:

- **ID** for BOT identification;
- **HWID** that uniquely identifies a machine (used for monitoring multiple infection from the same machine, indicating an analyzing attempts from researcher);
- **Token:** Exfiltrated token available on the victim’ machine;
- **File:** An archive of all information gathered from the victim’s machine.

```
149   PTR_lstrcatA(Src, "\r\n");
150   PTR_lstrcatA(Src, "------");
151   PTR_lstrcatA(Src, ptr_random_delimiter);
152   PTR_lstrcatA(Src, "\r\n");
153   PTR_lstrcatA(Src, "Content-Disposition: form-data; name=\"");
154   PTR_lstrcatA(Src, "token");
155   PTR_lstrcatA(Src, "\"\r\n\r\n");
156   PTR_lstrcatA(Src, 0);
157   PTR_lstrcatA(Src, "\r\n");
158   PTR_lstrcatA(Src, "------");
159   PTR_lstrcatA(Src, ptr_random_delimiter);
160   PTR_lstrcatA(Src, "\r\n");
161   PTR_lstrcatA(Src, "Content-Disposition: form-data; name=\"");
162   PTR_lstrcatA(Src, "hwid");
163   PTR_lstrcatA(Src, "\"\r\n\r\n");
164   PTR_lstrcatA(Src, v41);
165   PTR_lstrcatA(Src, "\r\n");
166   PTR_lstrcatA(Src, "------");
167   PTR_lstrcatA(Src, ptr_random_delimiter);
168   PTR_lstrcatA(Src, "\r\n");
169   PTR_lstrcatA(Src, "Content-Disposition: form-data; name=\"");
170   PTR_lstrcatA(Src, "file");
171   PTR_lstrcatA(Src, "\"\r\n\r\n");
```

Figure 11 - POST request structure

References

Sample:

- 556f8b06b92ddbc4008dea5298eab3934c61647a1cd7333a9087c37cc5a75456 (SHA256)MalwareBazaar

Ida-python scrypt:

- ida_vidar_string_decrypt.py Microsoft Defender's Sandbox:
- BlackHat 2018 detailed analysis

IOCs

- Network indicators
    - https://t.]me/game4serv
    - https://steamcommunity.]com/profiles/76561199523054520
    - http://bigsnowstone.]com/
- Targets

Browsers	**Browser Extensions - Wallets**	**Authenticator/Password Manager**	**Desktop Programs**
Mozilla Firefox	TronLink	Authenticator	LevelDB
Pale Moon	Meta	Authy	Thunderbird
Google Chrome	BinanceChainWallet	EOS Authenticator	Telegram
Chromium	Yoroi	GAuth Authenticator	WinSCP
Amigo	NiftyWallet		IndexedDB
Torch	MathWallet		Steam
Comodo Dragon	Coinbase		Jaxx_Desktop
Epic Privacy Browser	Guarda		Binance Desktop
Vivaldi	EQUALWallet		Bitcoin Core
CocCoc	JaxxLiberty		Bitcoin Core Old
Cent Browser	BitAppWallet		Raven Core
TorBro Browser	iWallet		Ledger Live
Chedot Browser	Wombat		Blockstream
Brave_Old	MewCx		
7Star	GuildWallet		

Browsers	**Browser Extensions - Wallets**	**Authenticator/Password Manager**	**Desktop Programs**
Microsoft Edge	RoninWallet		
360 Browser	NeoLine		
QQBrowser	CloverWallet		
Opera	LiqualityWallet		
OperaGX	Terra_Station		
CryptoTab Browser	Keplr		
Brave	Sollet		
	AuroWallet		
	PolymeshWallet		
	ICONex		
	Harmony		
	EVER Wallet		
	KardiaChain		
	Trezor Password Manager		
	Rabby		
	Phantom		
	BraveWallet		
	PaliWallet		
	BoltX		
	Xdefi		
	Nami		
	MaiarDeFiWallet		
	WavesKeeper		
	Solflare		

Browsers	Browser Extensions - Wallets	Authenticator/Password Manager	Desktop Programs
	CyanoWallet		
	KHC		
	TezBox		
	Temple		
	Goby		
	RoninWalletEdge		
	Wasabi Wallet		
	Daedalus Mainnet		
