Robbinhood Malware Analysis with Radare2

goggleheadedhacker.com/blog/post/12

Jacob Pimental July 1, 2019

01 July 2019

This article will provide an overview of how we can extract function names from Windows GoLang binaries to make reversing easier and to give a brief analysis on the Robbinhood Ransomware that attacked Baltimore recently. GoLang is a programming language designed around multi-threaded applications. The difficulty in reversing GoLang binaries is that all libraries are statically linked which means there will be a large number of functions in the application, most of which are not even used during execution. For example, in a normal Hello World compiled GoLang binary, radare2 detects 1800 functions.

The gopclntab section in a GoLang program contains a table of function locations along with their names. Radare2 is able to parse out this table and label every function accordingly, however this only works on binaries compiled for Linux. When it comes to Windows, Radare2 is not able to find the gopclntab and we are left with thousands of unlabeled functions with no clue as to what they do. I decided to learn how the gopclntab works and create a parser using r2pipe to label all functions.

Gopclntab

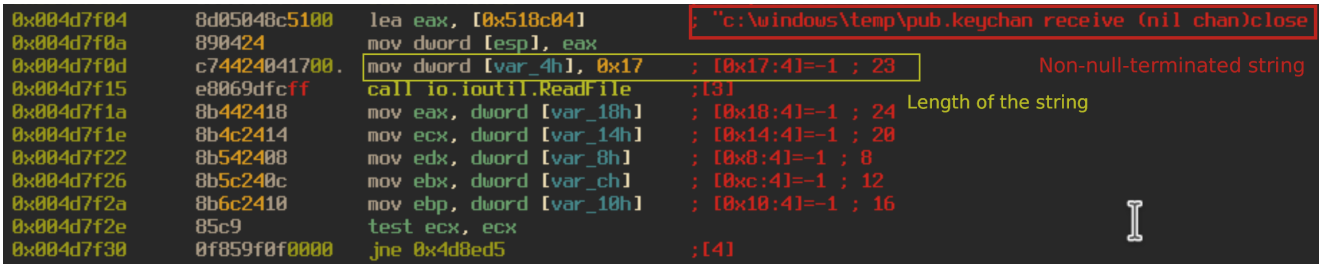
The gopclntab section always seems to start with the bytes 0xfbffffff, and after that contains the size of the table. The next 8 bytes contains the location of the first function, then the offset of that function from the start of the table. If you go to that offset from the start of the table you'll get the offset for the name of the function.

Using this table we can get all of the function information for the binary. I created a small python script that uses r2pipe to parse this table and rename/create the necessary functions. You can get the script on my gitHub. There is also a really in-depth article you can read about the gopclntab here.

Robbinhood

In May, Baltimore was hit by a ransomware known as Robbinhood, that took out multiple services. The ransomware itself was written in GoLang, so I thought this would be a good opportunity to become familiar with GoLang reversing and analyze the sample.

When reversing a GoLang binary, the main function will be located at *main.main*. Seeking to that function in radare2, we can see the binary trying to open the file "C:\Windows\Temp\pub.key". Strings in a GoLang binary are not null-terminated like most C-type programming languages. Instead, they have a variable that contains the length of the string.



If the file "pub.key" does not exist, the program throws an error and exits. If it does exist, the program will get a list of drives that the computer has access to and run the function *main.CoolMaker*. The CoolMaker function spawns multiple processes of "cmd.exe" to run the service control manager in order to to shut down any endpoint agents or antivirus on the infected host.

```
lea eax, [0x51fa39]          ; "cmd.exe /c sc.exe stop \"Symantec System Recovery\" /ycrypto/rsa
mov dword [esp], eax
mov dword [var_4h], 0x34     ; '4' ; [0x34:4]=-1 ; 52
call main.runme              ;[1]
lea eax, [0x5204f0]          ; "cmd.exe /c sc.exe stop \"Veeam Backup Catalog Data Service\" /y0
mov dword [esp], eax
mov dword [var_4h], 0x3d     ; '=' ; [0x3d:4]=-1 ; 61
call main.runme              ;[1]
lea eax, [0x51ce7e]          ; "cmd.exe /c sc.exe stop AcronisAgent /ycmd.exe /c sc.exe stop FA_
mov dword [esp], eax
mov dword [var_4h], 0x26     ; '&' ; [0x26:4]=-1 ; 38
call main.runme              ;[1]
lea eax, [0x51ca49]          ; "cmd.exe /c sc.exe stop AcrSch2Svc /y cmd.exe /c sc.exe stop EsgS
mov dword [esp], eax
mov dword [var_4h], 0x25     ; '%' ; [0x25:4]=-1 ; 37
call main.runme              ;[1]
```

After the CoolMaker function has been called, Robbinhood will spawn 4 processes of the function main.main.func1, which appears to be where the actual encryption occurs. The function calls another function, *main.doit*, which creates an encryption key based on the "pub.key" file found at the start of execution. These processes will then recursively walk through the file system and encrypt any files it finds.

The program also logs the encrypted files names to four different log files, *rf_l, rf_s, ro_l, ro_s*. Both of the rf_* files log the names of the files the program deems "interesting". The ro_* logs contain all other filenames. Both sets of logs are separated into large file sizes (rf_l, ro_l), and small file sizes (rf_s, ro_s). These logs are deleted as soon as execution is completed.

```
lea eax, [0x51816f]          ; "[INF] Others(LARGE): [INF] Others(SMALL):
mov dword [var_4h], eax
mov dword [var_8h], 0x15     ; [0x15:4]=-1 ; 21
mov eax, dword [arg_1f0h]    ; [0x1f0:4]=-1 ; 496
mov dword [var_ch], eax
mov ecx, dword [arg_1f4h]    ; [0x1f4:4]=-1 ; 500
mov dword [var_10h], ecx
call runtime.concatstring2   ;[2]
mov eax, dword [var_14h]     ; [0x14:4]=-1 ; 20
mov ecx, dword [var_18h]     ; [0x18:4]=-1 ; 24
mov dword [esp], eax
mov dword [var_4h], ecx
call main.WriteLog           ;[3]
```

```
lea eax, [0x519a64]          ; "[INF] Interesting(LARGE): [INF] Interesting(SMALL):
mov dword [var_4h], eax
mov dword [var_8h], 0x1a     ; [0x1a:4]=-1 ; 26
mov eax, dword [arg_1f0h]    ; [0x1f0:4]=-1 ; 496
mov dword [var_ch], eax
mov ecx, dword [arg_1f4h]    ; [0x1f4:4]=-1 ; 500
mov dword [var_10h], ecx
call runtime.concatstring2   ;[4]
mov eax, dword [var_14h]     ; [0x14:4]=-1 ; 20
mov ecx, dword [var_18h]     ; [0x18:4]=-1 ; 24
mov dword [esp], eax
mov dword [var_4h], ecx
call main.WriteLog           ;[1]
```

The dropped ransom note is contained in the binary as a base64 encoded string. The different aspects of the note, such as payment amount and user id are defined by different variables in the note to make them easily interchangeable depending on who the victim is. For example, the wallet id is defined in the ransom note as “#WALLAD#” and is replaced by the string “14yos7dpe4bx3imnoGVUcMsNBwU1hLutfj”. I did check to see if any payments had been made to the bitcoin wallet address and it looks like Jack Young was true to his word and the city didn’t pay the ransom at all.

```
|          0x004df4f8      8d05384c5100    lea eax, [0x514c38]          ; "#WALLAD#%!Month(, idle: 2.5.4.102.
|          0x004df4fe      89442408        mov dword [var_8h], eax
|          0x004df502      c744240c0800.   mov dword [var_ch], 8
|          0x004df50a      8d0595bc5100    lea eax, [0x51bc95]          ; "14yos7dpe4bx3imnoGVUcMsNBwU1hLutfj
|          0x004df510      89442410        mov dword [var_10h], eax
Press <enter> to return to Visual mode.0.  mov dword [var_14h], 0x22    ; "" ; [0x22:4]=-1 ; 34
:> pr 34 @ 0x51bc95
14yos7dpe4bx3imnoGVUcMsNBwU1hLutfj
```

Summary	
Address	14yos7dpe4bx3imnoGVUcMsNBwU1hLutfj
Hash 160	2ba6ca95765559c929c17a3bc2b5d5f414efff5e
Transactions	
No. Transactions	0
Total Received	0 BTC
Final Balance	0 BTC

Overall, this was not a very advanced Ransomware. It is very loud due to the fact that it is shutting down multiple Endpoint agents and AV. It also did not have the ability to spread at all. Every infected computer had to be targeted individually for it to cause real damage.

However, this was a fun sample to analyze and taught me a lot about GoLang reversing. As always, if you have any questions or comments on this, or any of my other articles, feel free to reach out to me on my Twitter and Linkedin.

Thanks for reading and happy reversing!

Tutorial, Radare2, Malware Analysis, Malware, Linux, Windows, Scripting, Automation, r2pipe, GoLang

More Content Like This: