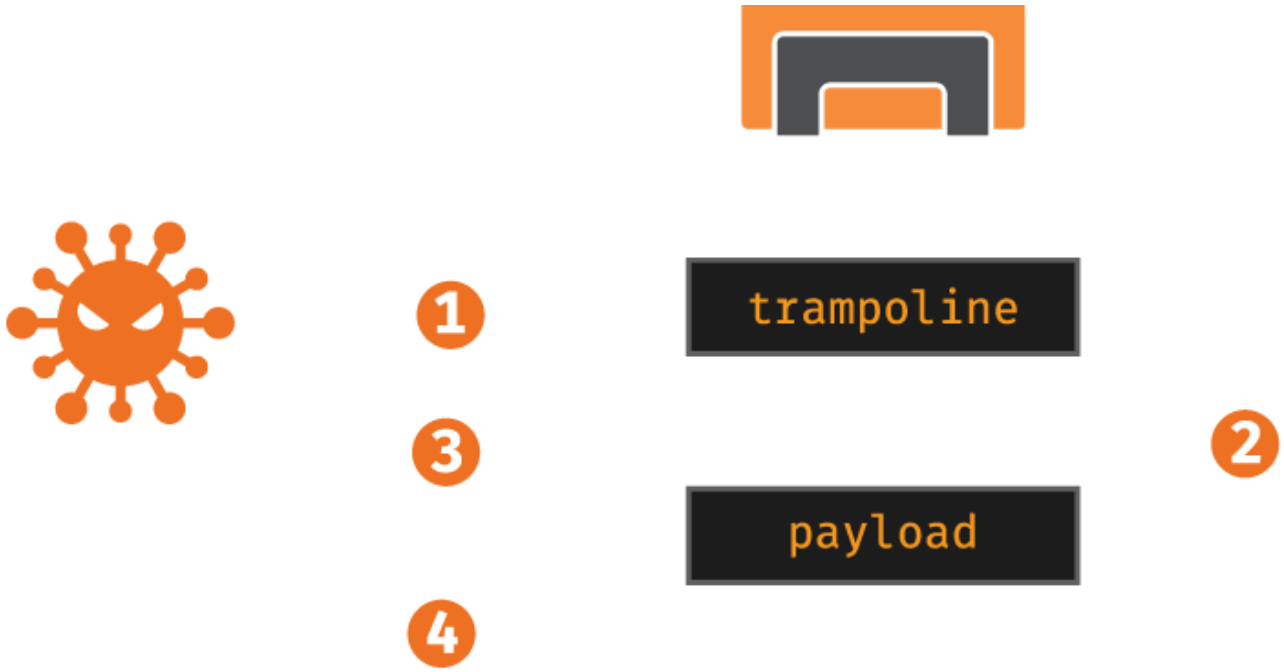
Floki Bot Strikes, Talos and Flashpoint Respond

blog.talosintel.com/2016/12/flokibot-collab.html



Executive Summary

Floki Bot is a new malware variant that has recently been offered for sale on various darknet markets. It is based on the same codebase that was used by the infamous Zeus trojan, the source code of which was leaked in 2011. Rather than simply copying the features that were present within the Zeus trojan "as-is", Floki Bot claims to feature several new capabilities making it an attractive tool for criminals. As Talos is constantly monitoring changes across the threat landscape to ensure that our customers remain protected as threats continue to evolve, we took a deep dive into this malware variant to determine the technical capabilities and characteristics of Floki Bot.

During our analysis of Floki Bot, Talos identified modifications that had been made to the dropper mechanism present in the leaked Zeus source code in an attempt to make Floki Bot more difficult to detect. Talos also observed the introduction of new code that allows Floki Bot to make use of the Tor network. However, this functionality does not appear to be active for the time being. Finally, through the use of the FIRST framework during the analysis process, Talos was able to quickly identify code/function reuse between Zeus and Floki Bot. This made sample analysis more efficient and decreased the amount of time spent documenting various functions present within the Floki Bot samples we analyzed.

Talos worked in collaboration with Flashpoint during the analysis of Floki Bot. This collaborative effort allowed Talos and Flashpoint to quickly communicate intelligence data related to active campaigns distributing Floki Bot as well as data regarding the technical functionality present within the malware. Additionally, Talos is making scripts available to the open source community that will help malware analysts automate portions of the Floki Bot analysis process and make the process of analyzing Floki Bot easier to perform.

Floki Bot Details

The infection process used by Floki Bot is comprised of several steps. At a high level, this process is illustrated in the following diagram:

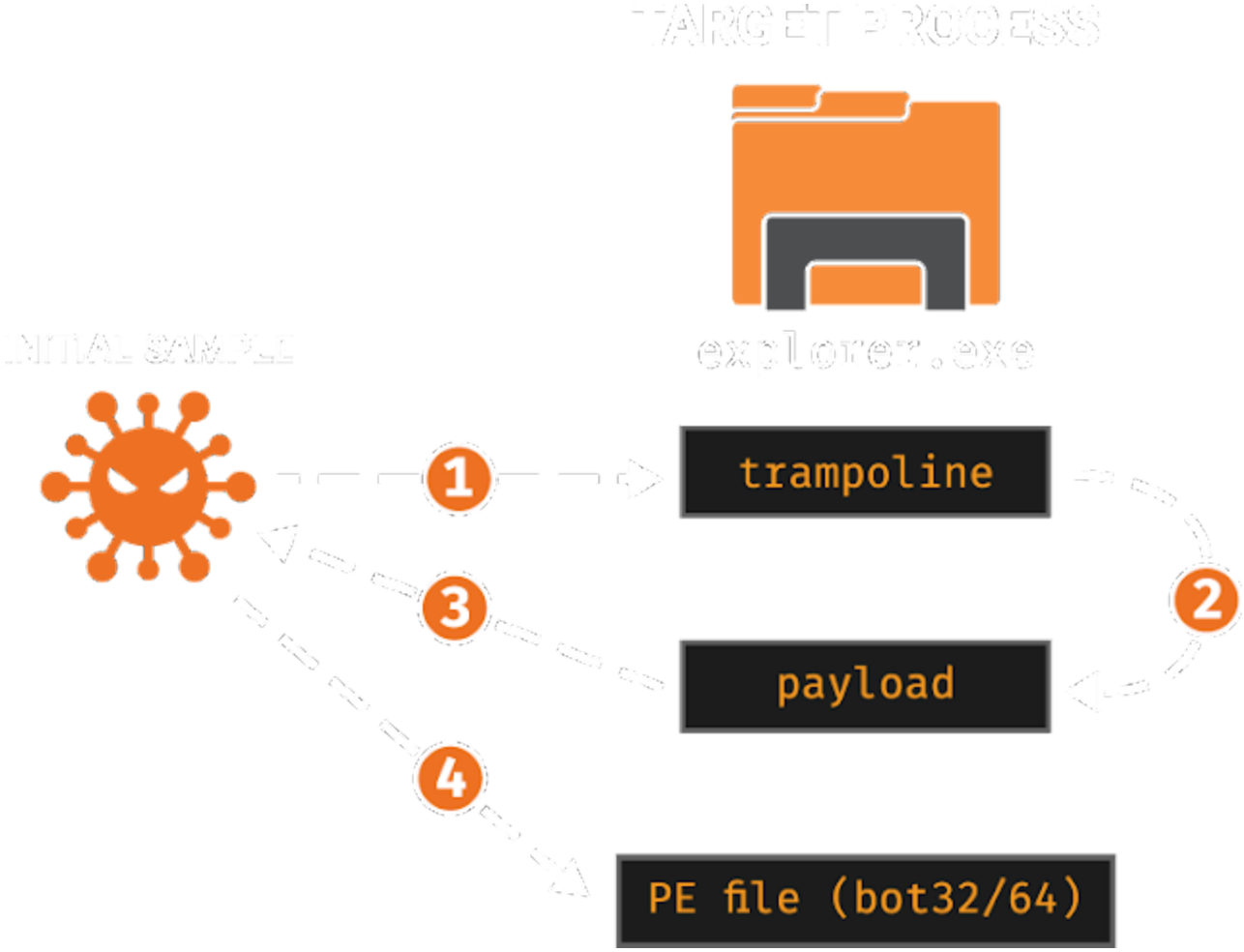


Figure 1: Inject Sequence of Malware Code

We started our analysis of Floki Bot using the following binary:

5e1967db286d886b87d1ec655559b9af694fc6e002fea3a6c7fd3c6b0b49ea6e (SHA256)

Once the malware is executed, it attempts to inject malicious code into 'explorer.exe' - the Microsoft Windows file manager. If it is unable to open 'explorer.exe', it will then inject into 'svchost.exe'. The first injection is simply a trampoline (step one in Figure 1). This trampoline

performs two different calls. The first call is a 'Sleep()' for 100 milliseconds. The second call passes control to another payload function. The argument to that function is a structure with the initial sample's process ID, the decryption key for further binary payloads, and the pointer and size of the payload resource in the initial sample's address space. Curiously enough, though the initial sample has resources labeled 'bot32' and 'bot64', the sample we analyzed is hardcoded to only pass the address of the 'bot32' resource to the injected payload. The reversed code responsible for mapping the 'bot32', 'bot64' and 'key' resources is shown in Figure 2.

```
1 BOOL __userpurge sub_4025C7@<eax>(_DWORD *a1@<edi>, HMODULE hModule)
2 {
3   HRSRC v2; // eax@1
4   int v3; // eax@2
5   HRSRC v4; // eax@7
6   int v5; // eax@8
7   HRSRC v6; // eax@10
8   unsigned int v7; // eax@11
9   int v9; // [esp+4h] [ebp-4h]@1
10
11  v9 = 0;
12  v2 = FindResourceW(hModule, L"key", (LPCWSTR)0xA);
13  if ( v2 )
14    v3 = sub_40209B(hModule, (int)&v9, v2);
15  else
16    v3 = 0;
17  if ( v3 && v9 )
18  {
19    sub_401831(a1 + 21, v9, 16);
20    sub_401811();
21  }
22  v4 = FindResourceW(hModule, L"bot32", (LPCWSTR)0xA);
23  if ( v4 )
24    v5 = sub_40209B(hModule, (int)(a1 + 1), v4);
25  else
26    v5 = 0;
27  a1[3] = v5;
28  v6 = FindResourceW(hModule, L"bot64", (LPCWSTR)0xA);
29  if ( v6 )
30    v7 = sub_40209B(hModule, (int)(a1 + 2), v6);
31  else
32    v7 = 0;
33  a1[4] = v7;
34  return a1[1] && a1[3] > 0u && a1[2] && v7 > 0;
35 }
```

Figure 2: Mapping of 'bot32', 'bot64' and 'key' Resources

As you can see from the following screenshots, Figure 3 shows the code responsible for preparing the shellcode for the injection. This operation is performed in the initial binary. Figure 4 shows the result of the injection into the 'explorer.exe' process. We can clearly observe that the disassembly is based on the previous shellcode and contains the two calls

described above. Specifically, the call at 0xA001F invokes the payload, which is the step two in Figure 1.

```
0040295C  .  C74424 24 55  MOV DWORD PTR SS:[ESP+24],51EC8B55
00402964  .  C74424 28 C7  MOV DWORD PTR SS:[ESP+28],0FC45C7
0040296C  .  C74424 2C 00  MOV DWORD PTR SS:[ESP+2C],68000000
00402974  .  895C24 30     MOV DWORD PTR SS:[ESP+30],EBX
00402978  .  C74424 34 FF  MOV DWORD PTR SS:[ESP+34],C7FC55FF
00402980  .  C74424 38 45  MOV DWORD PTR SS:[ESP+38],0FC45
00402988  .  C74424 3C 00  MOV DWORD PTR SS:[ESP+3C],680000
00402990  .  C74424 40 00  MOV DWORD PTR SS:[ESP+40],FF000000
00402998  .  C74424 44 55  MOV DWORD PTR SS:[ESP+44],C483FC55
004029A0  .  C74424 48 04  MOV DWORD PTR SS:[ESP+48],5DE58B04
004029A8  .  C64424 4C C3  MOV BYTE PTR SS:[ESP+4C],0C3
```

Figure 3: Shellcode Preparation

```
000A0000   55              PUSH EBP
000A0001   8BEC            MOV EBP,ESP
000A0003   51              PUSH ECX
000A0004   C745 FC FF106B  MOV DWORD PTR SS:[EBP-4],756B10FF
000A000B   68 64000000     PUSH 64
000A0010   FF55 FC         CALL DWORD PTR SS:[EBP-4]
000A0013   C745 FC 000008  MOV DWORD PTR SS:[EBP-4],80000
000A001A   68 00000900     PUSH 90000
000A001F   FF55 FC         CALL DWORD PTR SS:[EBP-4]
000A0022   83C4 04         ADD ESP,4
000A0025   8BE5            MOV ESP,EBP
000A0027   5D              POP EBP
000A0028   C3              RETN
```

Figure 4: Disassembly of the Injected Shellcode

The next logical step is another injection which also happens within the 'explorer.exe' address space. This time the payload - the one executed after the trampoline - resolves the required APIs via the use of a CRC lookup and then maps the 'bot32' resource section from the initial binary.

The resource is encrypted with RC4, and can be decrypted with the 16 byte key data from the 'key' resource, which is passed as an argument to the injected code. Moreover, the resource is compressed with the LZNT1 algorithm, and is extracted by invoking RtlDecompressBuffer. Talos has created and is releasing a script called 'PayloadDump' which will extract these bot payloads. This bot is the final component and is the one containing the banking trojan functionality. It is flagged by many AV engines as a classic Zeus bot. The bot is loaded and injected into 'explorer.exe'. These steps are the labeled 3 and 4 in Figure 1.

At every stage, the malware uses hashing to obfuscate module and function names used in dynamic library resolution. Interestingly, the initial sample and the bot (bot32) executable use the same CRC32 implementation and XOR the result with a static key, in our case this was

0x5E58, while the payload uses the same CRC32 implementation but a different XOR key, in our case 0x3086. The names of the modules are converted to lowercase before the computation (Windows file names are traditionally case insensitive).

Currently, the 'bot32' resource is immediately recognized by more than 30 AV engines on Virustotal, with most of the detections identifying it as Zbot, while the 'bot64' resource is detected as malicious by only 10 AV engines. During our analysis, we extracted the sample from both a physical memory dump of the explorer.exe process (See the Memory Analysis section), as well as from the resource section of the initial binary. At first glance, this sample looks like a normal Zeus bot. The main difference is support for the Tor network that should be activated when the C2 domain specified in the malware configuration ends with '.onion' which is the pseudo TLD for Tor related domains.. When this is the case, a standard Tor proxy server is configured to listen on localhost:9050, as you can see in the screenshot below:

```
if ( q_StrStr(v6, ".onion") )
  v3 = sub_40B4CB((int)".onion", v6);
else
 v16 = 0;
 v2 = sub_40B32D(9050, "127.0.0.1");
 if ( v2 != -1 )
 {
```

Figure 5: Floki Bot Tor Functionality

This feature appears to be under development and could not be activated in the samples Talos analyzed.

Floki Bot's Dropper/Loader

The loader used by Floki Bot is not encrypted. It also does not utilize any anti-debugging techniques. The loader does hide the system calls used to inject the malicious payload into other processes. The injection technique used by the Floki Bot loader has already been thoroughly documented here so we will not go into significant detail on how that process works.

Network Analysis

Floki Bot communicates with C2 over an HTTPS connection. Interestingly, the malware author advertises an anti deep packet inspection feature. To achieve this, the bytes in network packets are packaged in BinStorage structures that are sent over HTTPS. Each byte in the BinStorage structure is XOR'd by the previous byte and then additionally encrypted with RC4. This functionality was also present in the leaked Zeus source code and is not new to Floki Bot. By breaking the HTTPS connection and decrypting the packet payloads, we

noticed that the malware sends back information about the infected machine such as the computer name and the screen resolution. Floki Bot claimed it "cannot be detected by Deep-Packet-Inspection unlike Zeus", but the only major change to the leaked source code is Tor support, which was not found to be used by any samples found in the wild. Talos was able to decrypt Floki Bot network packets after intercepting them using mitmproxy as the malware does not use certificate pinning for its communications.

Memory Analysis

During our analysis we also performed a memory-based forensic analysis after infecting a VM with Floki Bot. In this way we used an opposite approach, starting our analysis from the end and then trying to rebuild the different steps of the infection process. First, we took a physical memory dump with win32dd and analyzed it with Volatility - a famous open source memory forensics framework. First we used the 'pslist' Volatility plugin. This plugin lists all processes by walking the double linked list connecting all of the _EPROCESS objects. Nothing suspicious was found from its output. We then used the 'netscan' plugin and it showed network activity from the 'explorer.exe' process, which is something that needed more investigation as it is definitely not normal to have network traffic from the file manager. Based on this finding, we ran 'malfind' on the 'explorer.exe' process and identified interesting traces and the PE file injected into the process. We dumped these artifacts and they matched the partial results of the reversing process. We could observe the trampoline, the payload and the PE file. In relation to the persistence mechanisms employed by Floki Bot, we identified some artifacts using 'filescan' and observed that the binary (with a random name) was also copied into the Startup folder.

Collaboration with Flashpoint

During our investigation into the Floki Bot malware, we leveraged a collaborative relationship with Flashpoint, who we worked with to gather intelligence information and share technical details regarding the malware samples, the campaigns that are currently using Floki Bot, and the darknet markets on which Floki Bot is being bought and sold. Flashpoint has been tracking several Floki Bot actors and campaigns. Flashpoint has also released a blog post that contains relevant intelligence information related to currently active Floki Bot campaigns operating globally. The Flashpoint post can be found here.

Using FIRST to Analyze Floki Bot

During the analysis process, Talos leveraged the Function Identification and Recovery Signature Tool (FIRST) and associated IDA Python plugin to collect and document functionality present within the Floki Bot samples that were analyzed. FIRST is an open

source framework recently released by Talos that allows malware analysts and researchers to collaborate and share analysis data related to the malicious functions present in malware samples.

Using FIRST enables quick and efficient analysis of malware as it minimizes the amount of time spent analyzing malicious code that has already been analyzed and documented. FIRST currently contains information for over 170,000 functions including: common libraries like Zlib and OpenSSL, leaked malware source code, and malicious Windows and Linux files analyzed by the community. It is particularly helpful when analyzing statically linked executables with thousands of library functions. Tools like Bindiff can be very useful, but they only let you compare a handful of files, and you have to find and obtain these files before you can do these comparisons. The FIRST plugin automatically looks for function similarities in each of the thousands of files submitted by the community.

IDA Pro uses FLIRT signatures to attempt to identify commonly used library functions, but it was unable to identify any functions in this sample of Floki Bot. IDA Pro's default response to unidentified functions is to name the functions according to the starting address. In this case we get 516 functions with generic names like "sub_402F34".

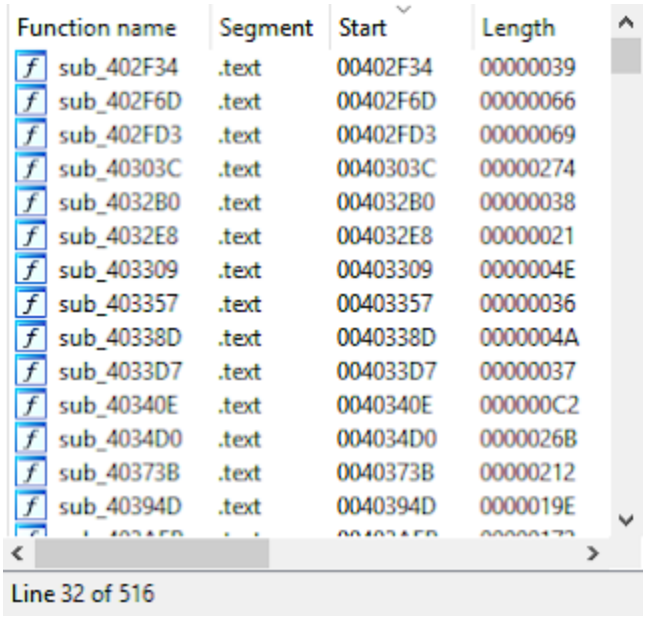


Figure 6: IDA Pro Function List before running FIRST

We queried FIRST, and within seconds had 128 functions labelled with meaningful names, prototypes, and comments. We can now instantly see what these functions do, and what arguments they take, even when those arguments are custom structures.

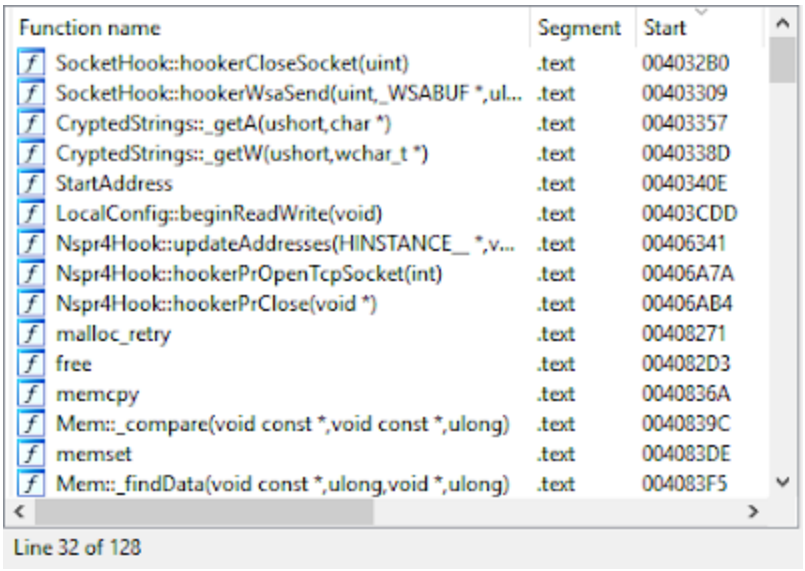


Figure 7: IDA Pro Function List after running FIRST

Many functions are difficult to classify without first analyzing their child-functions. Analysts often use a depth-first approach to label functions with obvious behaviors, then backtrack to the parent once they have a better understanding of the nested functions.

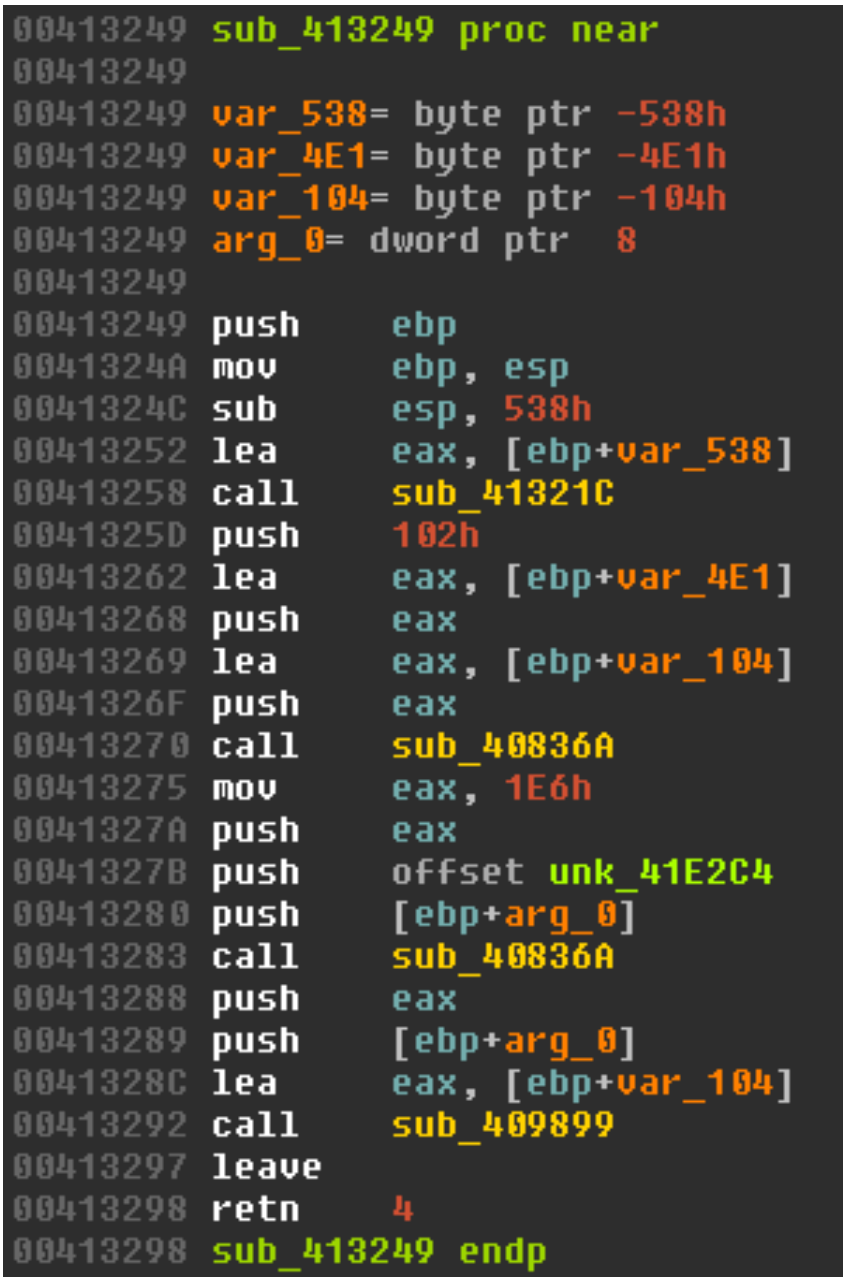


Figure 8: IDA Pro Showing Calls to Unknown Functions without FIRST

FIRST identified all of the functions in this example, and labeled them with their argument names and types. Functions now have comments showing these functions were from leaked Zeus source code, which gives us a substantial lead on where to find more info about the unidentified functions. Some functions not identified by FIRST are similar to functions in the Zeus source, but have been changed by modifications in the source code or compiler options.

```
00413249 void __fastcall Core::getPeSettings(struct PESETTINGS *) proc near
00413249
00413249 var_538= byte ptr -538h
00413249 from_buffer= byte ptr -4E1h
00413249 to_buffer= byte ptr -104h
00413249 arg_0= dword ptr  8
00413249
00413249 push    ebp
0041324A mov     ebp, esp
0041324C sub     esp, 538h
00413252 lea     eax, [ebp+var_538]
00413258 call    Core::getBaseConfig(BASECONFIG *) ; Leaked Source - Zeus Client
0041325D push    102h            ; size
00413262 lea     eax, [ebp+from_buffer]
00413268 push    eax             ; from_buffer
00413269 lea     eax, [ebp+to_buffer]
0041326F push    eax             ; to_buffer
00413270 call    Mem::_copy(void *,void const *,ulong)
00413275 mov     eax, 1E6h
0041327A push    eax             ; size
0041327B push    offset unk_41E2C4 ; from_buffer
00413280 push    [ebp+arg_0]     ; to_buffer
00413283 call    Mem::_copy(void *,void const *,ulong)
00413288 push    eax
00413289 push    [ebp+arg_0]
0041328C lea     eax, [ebp+to_buffer]
00413292 call    Crypt::_rc4(void *,ulong,Crypt::RC4KEY *) ; Leaked Source - Zeus Client
00413297 leave
00413298 retn    4
00413298 void __fastcall Core::getPeSettings(struct PESETTINGS *) endp
```

Figure 9: The Same Function Labeled by FIRST

If you compared this Floki Bot executable with Zeus, you would see the sizes of the BASECONFIG structures are different, and the offsets for global variables have changed as well. One of FIRST's engines identified these functions despite modifications to these parameters. Thanks to FIRST, we are able to quickly find the chunk of leaked source code responsible for this function.

```
void Core::getPeSettings(PESETTINGS *ps)
{
  BASECONFIG baseConfig;
  getBaseConfig(&baseConfig);

  Crypt::RC4KEY rc4k;
  Mem::_copy(&rc4k, &baseConfig.baseKey, sizeof(Crypt::RC4KEY));
  Mem::_copy(ps, &coreData.peSettings, sizeof(PESETTINGS));
  Crypt::_rc4(ps, sizeof(PESETTINGS), &rc4k);
}
```

Figure 10: Leaked Function Source Code

All of the analysis data and function documentation that was created by Talos while analyzing Floki Bot samples have been made available via the public Talos FIRST server

(beta). More information about the FIRST framework and how it can be used can be found here.

Tool Release

During the analysis process, Talos also created scripts to help automate portions of the analysis of Floki Bot, which are now being released to the open source community. These scripts enable analysts to dump the configuration parameters used by Floki Bot samples, as well as the Floki Bot payload itself.

PayloadDump - Extracts the final payload in PE32 format from the initial Floki Bot sample.

ConfigDump - Enables the extraction of the Floki Bot configuration parameters used by the sample.

These scripts can be downloaded from Github here.

Conclusion

Floki Bot is another example of what happens when the source code of successful malware kits gets leaked online. As we have seen several times since the Zeus source code became available, new malware variants based on this codebase continue to emerge. Floki Bot is unique in that the authors of this malware have put effort into expanding upon the functionality that was present in Zeus and have implemented new functionality making Floki Bot very attractive to criminals.

As Floki Bot is currently being actively bought and sold on several darknet markets it will likely continue to be seen in the wild as cybercriminals continue to attempt to leverage it to attack systems in an aim to monetize their efforts. As the leak of the Zeus source code continues to have ripple effects across the threat landscape, Talos will continue to monitor this and other threats that are actively being used in the wild to ensure that customers remain protected as new threats emerge or as existing threats change over time.

Coverage

Additional ways our customers can detect and block this threat are listed below.

PRODUCT	PROTECTION
AMP	✔
CWS	✔
Email Security	N/A
Network Security	✔
WSA	✔

Advanced Malware Protection (AMP) is ideally suited to prevent the execution of the malware used by these threat actors.

CWS or WSA web scanning prevents access to malicious websites and detects malware used in these attacks.

The Network Security protection of IPS and NGFW have up-to-date signatures to detect malicious network activity by threat actors.

Indicators of Compromise (IOCs)

Malware Binaries:

08e132f3889ee73357b6bb38e752a749f40dd7e9fb168c6f66be3575dbbbc63d (SHA256)
5028124ce748b23e709f1540a7c58310f8481e179aff7986d5cfd693c9af94da (SHA256)
0aa1f07a2ebcdd42896d3d8fdb5e9a9fef0f4f894d2501b9cbbe4cbad673ec03 (SHA256)
5e1967db286d886b87d1ec655559b9af694fc6e002fea3a6c7fd3c6b0b49ea6e (SHA256)
d1d851326a00c1c14fc8ae77480a2150c398e4ef058c316ea32b191fd0e603c0 (SHA256)
e0b599f73d0c46a5130396f81daf5ba9f31639589035b49686bf3ef5f164f009 (SHA256)
e43ee2ab62f9dbeb6c3c43c91778308b450f5192c0abb0242bfddb8a65ab883a (SHA256)
2b832ef36978f7852be42e6585e761c3e288cfbb53aef595c7289a3aef0d3c95 (SHA256)
4bdd8bbdab3021d1d8cc23c388db83f1673bdab44288fccae932660eb11aec2a (SHA256)
3c2c753dbb62920cc00e37a7cab64fe0e16952ff731d39db26573819eb715b67 (SHA256)
7bd22e3147122eb4438f02356e8927f36866efa0cc07cc604f1bff03d76222a6 (SHA256)
9d9c0ada6891309c2e43f6bad7ffe55c724bb79a0983ea6a51bc1d5dc7dccf83 (SHA256)
e205a0f5688810599b1af8f65e8fd111e0e8fa2dc61fe979df76a0e4401c2784 (SHA256)
ac5ae89af8d2ffdda465a4038f0f24fcbcb650140741c2b48adadc252a140e54 (SHA256)

Command and Control URLs:

https[:]//193.201.225[.]30/sweetdream/gxve8xj4a7t8t8sug8s57.php
https[:]//shhtunnel[.]at/class/gate.php

https[:]//extensivee[.]bid/000L7bo11Nq36ou9cfjfb0rDZ17E7ULo_4agents/gate.php
https[:]//5.154.190[.]248/gate.php
https[:]//vtraffic[.]su/gate.php
https[:]//springlovee[.]at/adm/config.bin
https[:]//feed.networksupdates[.]com/feed/webfeed.xml
https[:]//wowsupplier[.]ga/cpflkabwbebeu/gtlejbsbu.php
https[:]//adultgirlmail[.]com/mail/gate.php
https[:]//uspal[.]cf/3faf5c96-9c2b-11e6-95d4-00163c75bf83/gate.php