Floki-Bot

A new Zeus-malware variant known as Floki-Bot is for sale on various darknet markets. The codebase for this virus is same as that of the infamous Zeus trojan. The source code for Zeus was leaked back in 2011. Floki-Bot not only copies the features of Zeus, it claims to have several new characteristics that make it attractive for criminal activities. Let us take a deep dive inside the Floki-Bot malware and analyze its characteristics and reach.

As we analyze the malware, we identify modification made to the dropper mechanism present in the source code of the Zeus malware so that Floki-Bot becomes difficult to detect. The Floki-Bot also makes use of the Tor network.

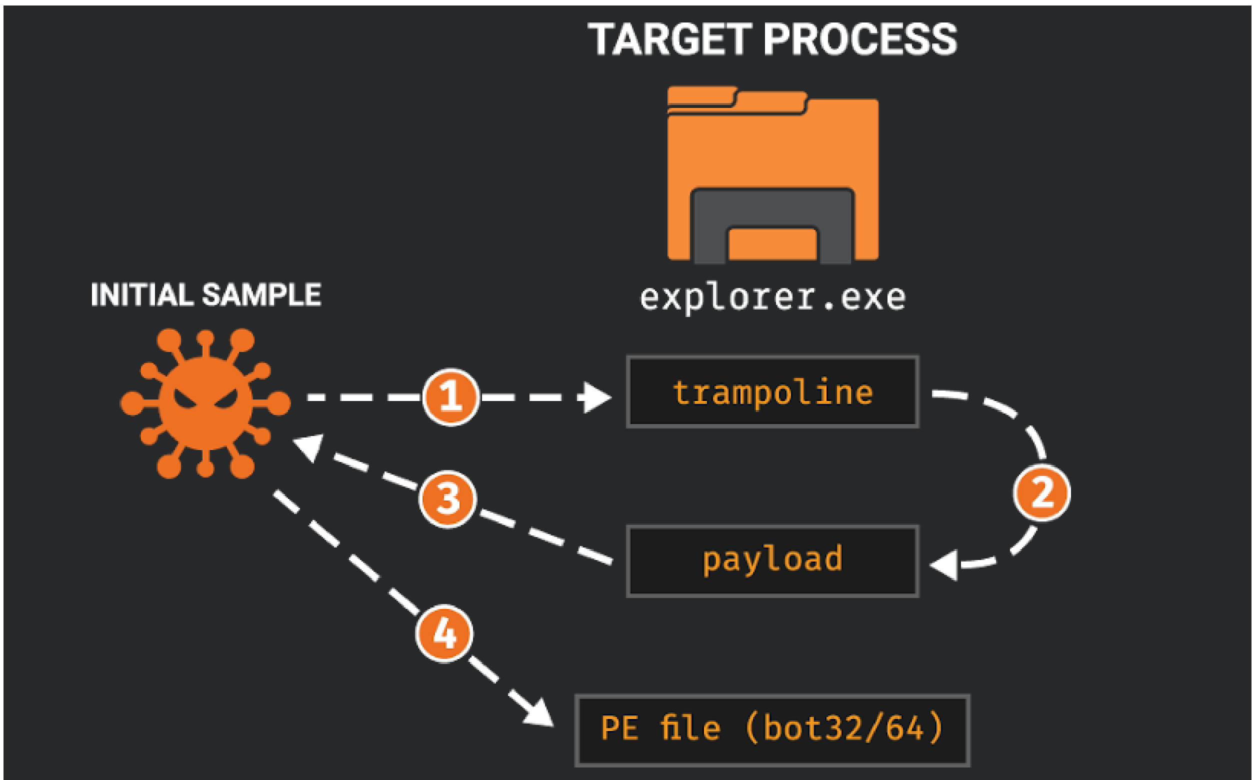


Figure 1: Inject Sequence of Malware Code

Upon execution, the trojan attempts and injects malicious code into ‘explorer.exe’ which is the Microsoft Windows file manager. If it cannot open the ‘explore.exe’ executable, the trojan injects into ‘svchost.exe’. The first injection is a trampoline and it performs two main function calls, first is sleep() for 100 ms , second call passes the control to the payload. The argument to that payload call is a structure with the initial sample’s PID, decryption key for remaining payloads and pointer and size of payload resource in the first samples address space. Even though the initial sample has resources like ‘bot32’ and ‘bot64’ , the address of the ‘bot32’ resource is the only one passed to the injected payload.

Following is the reversed code.

```
BOOL __userpurge sub_4025C7@<eax>(_DWORD *a1@<edi>, HMODULE hModule)
{
  HRSRC v2; // eax@1
  int v3; // eax@2
  HRSRC v4; // eax@7
  int v5; // eax@8
  HRSRC v6; // eax@10
  unsigned int v7; // eax@11
  int v9; // [esp+4h] [ebp-4h]@1

  v9 = 0;
  v2 = FindResourceW(hModule, L"key", (LPCWSTR)0xA);
  if ( v2 )
    v3 = sub_40209B(hModule, (int)&v9, v2);
  else
    v3 = 0;
  if ( v3 && v9 )
  {
    sub_401831(a1 + 21, v9, 16);
    sub_401811();
  }
  v4 = FindResourceW(hModule, L"bot32", (LPCWSTR)0xA);
  if ( v4 )
    v5 = sub_40209B(hModule, (int)(a1 + 1), v4);
  else
    v5 = 0;
  a1[3] = v5;
  v6 = FindResourceW(hModule, L"bot64", (LPCWSTR)0xA);
  if ( v6 )
    v7 = sub_40209B(hModule, (int)(a1 + 2), v6);
  else
    v7 = 0;
  a1[4] = v7;
  return a1[1] && a1[3] > 0u && a1[2] && v7 > 0;
}
```

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

The figure below shows the assembly instructions responsible for the shellcode preparation for the injection. The figure after that shows the resultant due to injection of the 'explorer.exe' process. We clearly observe that disassembly is based on previous shellcode and contains the above two calls. The call at 0xA001F is responsible for invoking the payload.

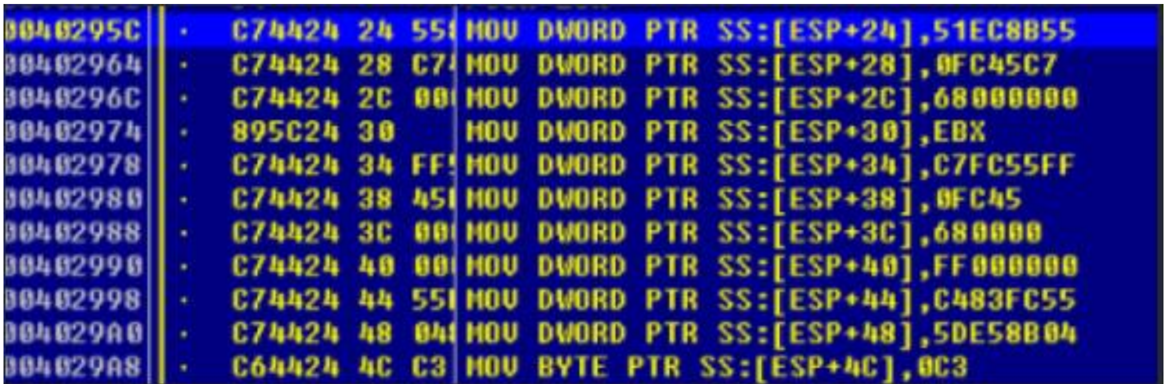


Figure 3: Shellcode Preparation

Figure 4: Disassembly of the Injected Shellcode

After this, another injection takes place within the 'explorer.exe' and this time the payload resolves the API's required via CRC lookup which then maps the bot32 resource section from the initial binary. The resource is encrypted using the RC4 algorithm which can be decrypted from the 'key' resource using the 16-byte key data and passed to the injected code as an argument. The resource which is compressed using the LZNT1 algorithm, invokes RtlDecompressBuffer and is extracted. A script called 'Payload Dump' is available which extracts these bot payloads. The bot is final component and has banking trojan functionality. The bot loads and is injected into 'explorer.exe'.

The trojan uses hashing to obfuscate module names and function names used in the resolution of dynamic libraries. The bot and the sample use the same CRC32 implementation and then XOR the result with a static key, in this case, 0x5E58 while the payload also uses the same CRC32 implementation but the XOR key is different, 0x3086 in this case. Module names are converted to lowercase before computation as Windows file names are traditionally case insensitive.

The sample was extracted from both a physical memory dump of the explorer.exe process as well as from the resource section of the binary. This sample differs from the Zeus malware in that it uses the Tor network which is activated when the C2 domain specified in the malware ends with a .onion. A standard proxy server in Tor is configured to listen on localhost:9050, as we can see in the screenshots below.

```
if ( q_StrStr(v6, ".onion") )
  v3 = sub_40B4CB((int)".onion", v6);
else
```

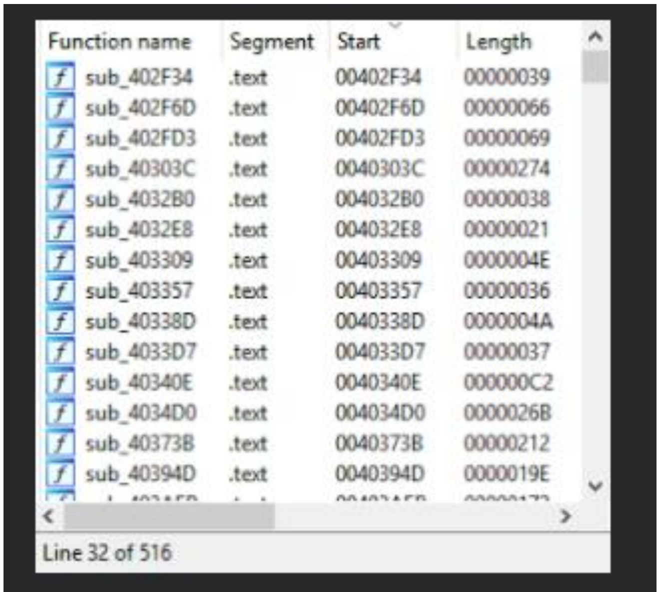
```
v16 = 0;
v2 = sub_40B32D(9050, "127.0.0.1");
if ( v2 != -1 )
{
```

Figure 5: Floki Bot Tor Functionality

Floki-Bot does not use an encrypted loader. The trojan also does not implement any anti-debugging techniques. It does hide the system calls that inject the malicious payload into other running processes. The bot uses an HTTPS connection to communicate with the C2 server. The malware author introduces an anti-DPI feature. The bytes in the network communication are packaged using BinStorage structures sent over HTTPS. Each byte in the structure is XORed by the previous byte and then additionally encrypted with RC4. So, on breaking the HTTPS connection and decrypting the payloads, we find that the trojan sends back information about the attacked machine such as computer name and screen resolution. The malware is not known to use certificate pinning for its communications.

After performing a memory-based analysis in a sandboxed VM, we listed all the processes using pslist which works by walking the double linked list connecting all the _EPROCESS objects. Then netscan was used it showed network traffic from 'explorer.exe'. Doing PEHeader analysis led us to 'malfind' on 'explorer.exe'

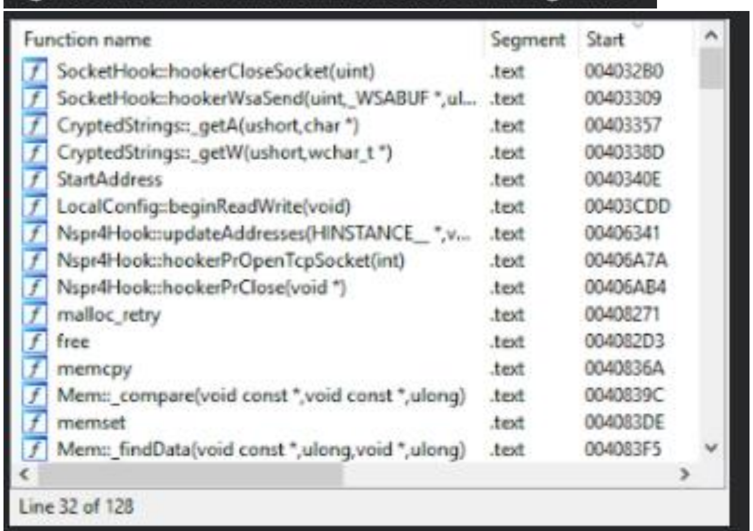
Snapshots:



Function name	Segment	Start	Length
sub_402F34	.text	00402F34	00000039
sub_402F6D	.text	00402F6D	00000066
sub_402FD3	.text	00402FD3	00000069
sub_40303C	.text	0040303C	00000274
sub_4032B0	.text	004032B0	00000038
sub_4032E8	.text	004032E8	00000021
sub_403309	.text	00403309	0000004E
sub_403357	.text	00403357	00000036
sub_40338D	.text	0040338D	0000004A
sub_4033D7	.text	004033D7	00000037
sub_40340E	.text	0040340E	000000C2
sub_4034D0	.text	004034D0	0000026B
sub_40373B	.text	0040373B	00000212
sub_40394D	.text	0040394D	0000019E

Line 32 of 516

Figure 6: IDA Pro Function List before running FIRST



Function name	Segment	Start
SocketHook::hookerCloseSocket(uint)	.text	004032B0
SocketHook::hookerWsaSend(uint,_WSABUF *,ul...	.text	00403309
CryptedStrings::_getA(ushort,char *)	.text	00403357
CryptedStrings::_getW(ushort,wchar_t *)	.text	0040338D
StartAddress	.text	0040340E
LocalConfig::beginReadWrite(void)	.text	00403CDD
Nspr4Hook::updateAddresses(HINSTANCE__ *,v...	.text	00406341
Nspr4Hook::hookerPrOpenTcpSocket(int)	.text	00406A7A
Nspr4Hook::hookerPrClose(void *)	.text	00406AB4
malloc_retry	.text	00408271
free	.text	004082D3
memcpy	.text	0040836A
Mem::_compare(void const *,void const *,ulong)	.text	0040839C
memset	.text	004083DE
Mem::_findData(void const *,ulong,void *,ulong)	.text	004083F5

Line 32 of 128

Figure 7: IDA Pro Function List after running FIRST

```
00413249 sub_413249 proc near
00413249
00413249 var_538= byte ptr -538h
00413249 var_4E1= byte ptr -4E1h
00413249 var_104= 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    sub_41321C
0041325D push    102h
00413262 lea     eax, [ebp+var_4E1]
00413268 push    eax
00413269 lea     eax, [ebp+var_104]
0041326F push    eax
00413270 call    sub_40836A
00413275 mov     eax, 1E6h
0041327A push    eax
0041327B push    offset unk_41E2C4
00413280 push    [ebp+arg_0]
00413283 call    sub_40836A
00413288 push    eax
00413289 push    [ebp+arg_0]
0041328C lea     eax, [ebp+var_104]
00413292 call    sub_409899
00413297 leave
00413298 retn    4
00413298 sub_413249 endp
```

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

```
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

```
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

SOURCE : CISCO TALOS INTELLIGENCE