Floki Bot and the stealthy dropper

November 10, 2016 by Malwarebytes Labs

Last updated: November 14, 2016

Floki Bot, described recently by Dr. Peter Stephenson from SC Magazine, is yet another bot based on the leaked Zeus code. However, the author came up with various custom modifications that makes it more interesting.

According to the advertisements announced on the black market, this bot is capable of making very stealthy injections, evading many mechanisms of detection. We decided to take a look at what are the tricks behind it. It turned out, that although the injection method that the dropper uses is not novel by itself, but it comes with few interesting twists, that are not so commonly used in malware.

Analyzed sample

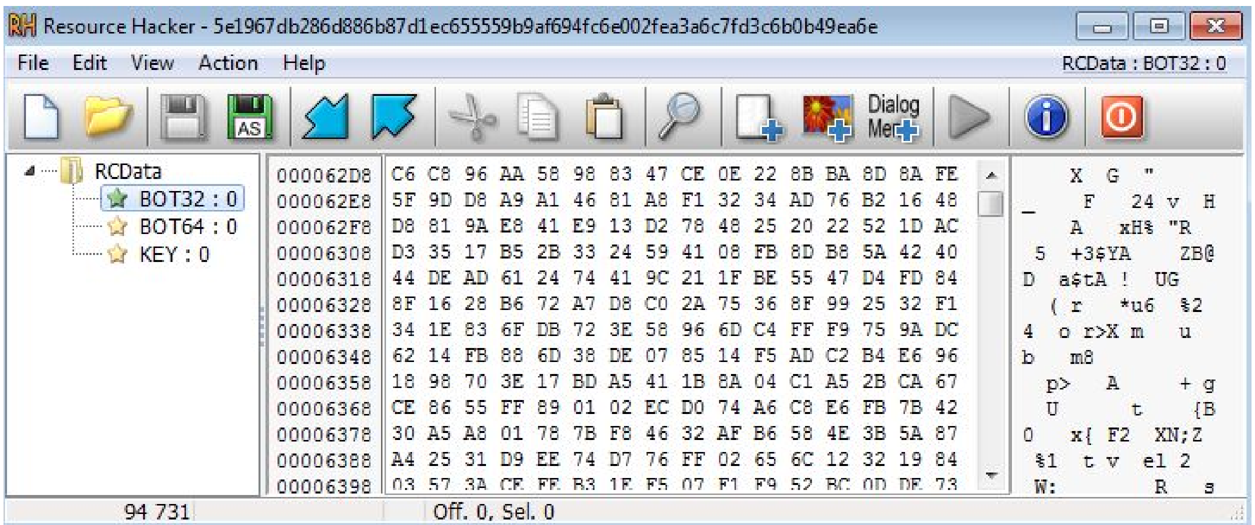
- **5649e7a200df2fb85ad1fb5a723bef22 – dropper <- main focus of this analysis**
    - e54d28a24c976348c438f45281d68c54 – core module – bot 32bit

○ [d4c5384da41fd391d16eff60abc21405](#) – core module – bot 64bit

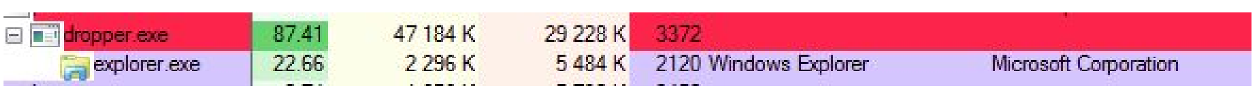
NOTE: The core modules depend on a data prepared by the dropper and they crash while run independently.

The Floki Dropper

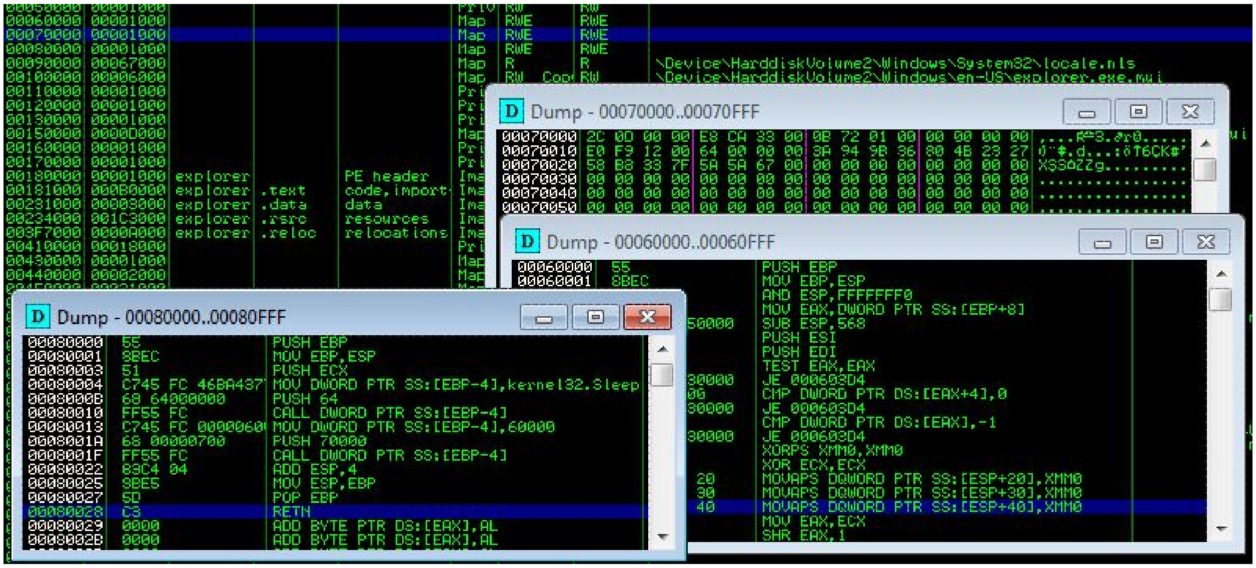
The Floki dropper looks simple and it has been found in wild without any outer protection layer. It has 3 resources with descriptive names – ***bot32***, ***bot64***, and ***key***:



When we try to observe its activity, we can see it making an injection into explorer.



Indeed, when we attach the debugger to the newly created explorer process, we can see some alien code implanted – it is written on three additional memory areas with full permissions (RWE):



However, when we trace the API calls, we cannot find any reference to a function that will write the code into the explorer process. Fragment of the trace:

[...]

```
28a8;called module: C:\Windows\system32\kernel32.dll:CreateProcessW
210f;called module: C:\Windows\system32\kernel32.dll:IsWow64Process
1d94;called module: C:\Windows\SYSTEM32\ntdll.dll:ZwClose
210f;called module: C:\Windows\system32\kernel32.dll:IsWow64Process
1d94;called module: C:\Windows\SYSTEM32\ntdll.dll:ZwClose
292c;called module: C:\Windows\system32\kernel32.dll:DuplicateHandle
210f;called module: C:\Windows\system32\kernel32.dll:IsWow64Process
1d94;called module: C:\Windows\SYSTEM32\ntdll.dll:ZwClose
2a1e;called module: C:\Windows\system32\kernel32.dll:GetThreadContext
2a37;called module: C:\Windows\system32\kernel32.dll:SetThreadContext
210f;called module: C:\Windows\system32\kernel32.dll:IsWow64Process
2aa1;called module: C:\Windows\system32\kernel32.dll:WaitForSingleObject
1818;called module: C:\Windows\system32\kernel32.dll:IsBadReadPtr
182a;called module: C:\Windows\SYSTEM32\ntdll.dll:RtlFreeHeap
2aad;called module: C:\Windows\system32\kernel32.dll:ExitProcess
```

We can see that a new process is created, and it’s context is being changed – that suggests manipulation – but where is the write? In order to find an answer to this question, we will take a deep dive inside the code.

Inside

At the beginning, the dropper dynamically loads some of the required imports:

```
00402679 push    ebp
0040267A mov     ebp, esp
0040267C and     esp, 0FFFFFFF8h
0040267F sub     esp, 634h
00402685 push    ebx
00402686 push    esi
00402687 push    edi
00402688 call    load_imports_by_hashes
0040268D xor     ebx, ebx
0040268F push    84C006A5h       ; CRC("ndll.dll") ^ 0x58E5
00402694 mov     syscalls_array, ebx
0040269A mov     syscalls_num, ebx
004026A0 call    search_and_open_ntdll
004026A5 mov     [esp+640h+var_62C], eax
004026A9 cmp     eax, 0FFFFFFFFh
```

The used approach depicts, that the author was trying not to leave any artifacts that could allow for easy detection of what modules and functions are going to be used. Instead of loading DLLs by their names, it picks them enumerating all the DLLs in the system32 directory:

```
004013CB  .   CMP EAX,ESI
004013CD  .v  JE SHORT dropper.004013DC
004013CF  .   PUSH dropper.00401068                 UNICODE "\\*.dll"
004013D4  .   LEA EAX,[LOCAL.2]
004013D7  .   CALL dropper.00401998
004013DC  >   LEA EAX,[LOCAL.151]
004013E2  .   PUSH EAX                              wininet.760D0000
004013E3  .   PUSH [LOCAL.2]
004013E6  .   CALL DWORD PTR DS:[0x407FB0]          kernel32.FindFirstFileW
004013EC  .   MOV EDI,EAX                           wininet.760D0000
004013EE  .   CMP EDI,-0x1
004013F1  .v  JE SHORT dropper.00401445
004013F3  >   LEA EBX,[LOCAL.140]
004013F9  .   CALL dropper.004018BE
004013FE  .   MOV ESI,EAX                           wininet.760D0000
00401400  .   TEST ESI,ESI
00401402  .v  JE SHORT dropper.00401421
00401404  .   MOV ECX,ESI
00401406  .   CALL dropper.004019E4
0040140B  .   PUSH EAX                              wininet.760D0000
0040140C  .   PUSH ESI
0040140D  .   CALL dropper.00401C9C                 crc32
00401412  .   XOR EAX,0x58E5
00401417  .   CMP EAX,[ARG.1]
0040141A  .v  JE SHORT dropper.00401435
0040141C  .   CALL dropper.00401811
00401421  >   LEA EAX,[LOCAL.151]
00401427  .   PUSH EAX                              wininet.760D0000
00401428  .   PUSH EDI
00401429  .   CALL DWORD PTR DS:[0x407F10]          kernel32.FindNextFileW
0040142F  .   TEST EAX,EAX                          wininet.760D0000
00401431  .^  JNZ SHORT dropper.004013F3
00401433  .v  JMP SHORT dropper.00401445
00401435  >   LEA EAX,[LOCAL.140]
0040143B  .   PUSH EAX                              wininet.760D0000
0040143C  .   CALL DWORD PTR DS:[0x408030]          kernel32.LoadLibraryW
00401442  .   MOV [LOCAL.3],EAX                     wininet.760D0000
00401445  >   PUSH EDI
00401446  .   CALL DWORD PTR DS:[0x407F40]          kernel32.FindClose
0040144C  .   MOV ESI,[LOCAL.2]
```

For the sake of obfuscation, it doesn't use string comparison. Instead, it calculates a checksum of each found name. The checksum is created by CRC32 from the name XORed with some hardcoded value, that is constant for a particular sample (in the described sample it is 0x58E5):

```
00401404 mov     ecx, esi
00401406 call    str_len
0040140B push    eax
0040140C push    esi
0040140D call    crc32
00401412 xor     eax, 58E5h
```

The resulting checksums are compared with the expected value, till the appropriate module is found and loaded. In similar way the export table of a particular module is enumerated and the required functions are being resolved.

After the initial imports load, exactly the same method is used to search NTDLL.DLL.

As we know, NTDLL.DLL provides an interface to execute native system calls. Every version of Windows may use a different number of a syscall in order to do the same thing. That's why it is recommended to use them via wrappers, that we can find among functions exported by NTDLL. For example, this is how the implementation of the *NtAllocateVirtualMemory* may look on Windows 7:

	Hex	Disasm
452D8	B813000000	MOV EAX, 0X13
452DD	BA0003FE7F	MOV EDX, 0X7FFE0300
452E2	FF12	CALL DWORD NEAR [EDX]
452E4	C21800	RET 0X18
452E7	90	NOP

Another variant, from Windows 8 looks a bit different:

	Hex	Disasm
6C1D0	B89B010000	MOV EAX, 0X19B
6C1D5	E803000000	CALL 0X6A26C1DD
6C1DA	C21800	RET 0X18
6C1DD	8BD4	MOV EDX, ESP
6C1DF	0F34	SYSENTER
6C1E1	C3	RET

The common part is, that the number of the syscall to be executed is moved into the EAX register.

The dropper loads NTDLL into the memory and extracts syscalls from selected functions:

0 : NtCreateSection
1 : NtMapViewOfSection
2 : ZwAllocateVirtualMemory
3 : ZwWriteVirtualMemory

4 : NtProtectVirtualMemory
5 : NtResumeThread
6 : ZwOpenProcess
7 : NtDuplicateObject
8 : NtUnmapViewOfSection

It checks a beginning of each function's code by comparing it with *0xB8*, that is a bytecode for moving a value into EAX:

```
00402003 movzx   edx, word ptr [ebx]
00402006 mov     esi, [edi+1Ch]
00402009 lea     edx, [esi+edx*4]
0040200C mov     esi, [edx+eax]
0040200F add     esi, eax
00402011 cmp     byte ptr [esi], 0B8h ; MOV EAX,imm32
00402014 jnz     short loc_40206E
```

If the check passed, the syscall value, that was moved into EAX, is extracted and stored in a buffer:

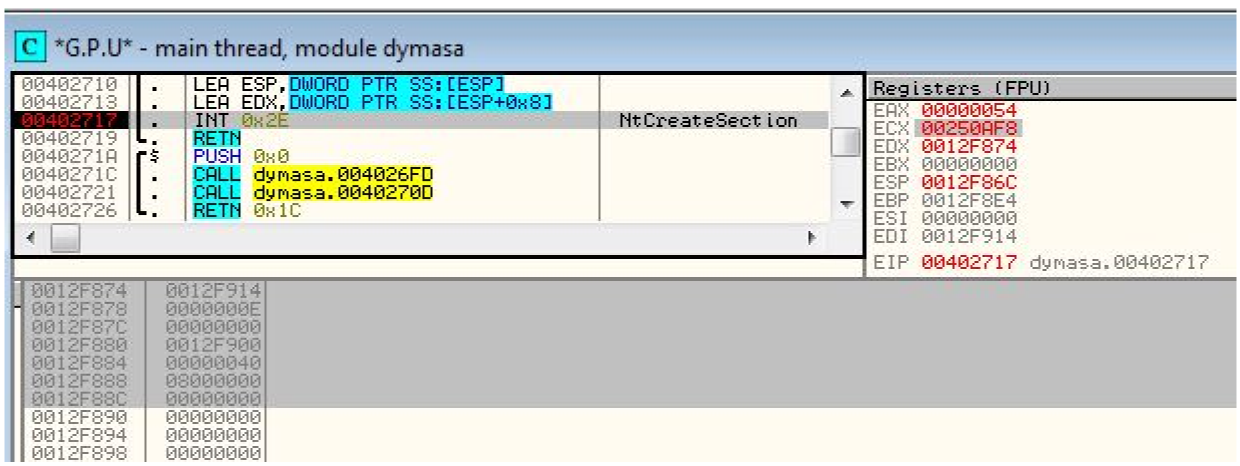
```
00402045 and     [ebp+syscall_buf], 0
00402049 push    4               ; 4 bytes - syscall value length
0040204B lea     ecx, [esi+1]    ; move pointer by 1 byte
0040204E push    ecx
0040204F lea     ecx, [ebp+syscall_buf]
00402052 push    ecx
00402053 call    copy_bytes
00402058 mov     ecx, [ebp+syscall_buf]
0040205B inc     [ebp+counter]
```

Then, when the dropper wants to call some of the functions, it uses those extracted values. The number of the syscall is fetched from the array where it was saved, and copied to EAX. Parameters of the function are pushed on the stack. The pointer to the parameters is loaded into EDX – and the syscall is triggered by with the help of an interrupt – ***INT 0x2E:***

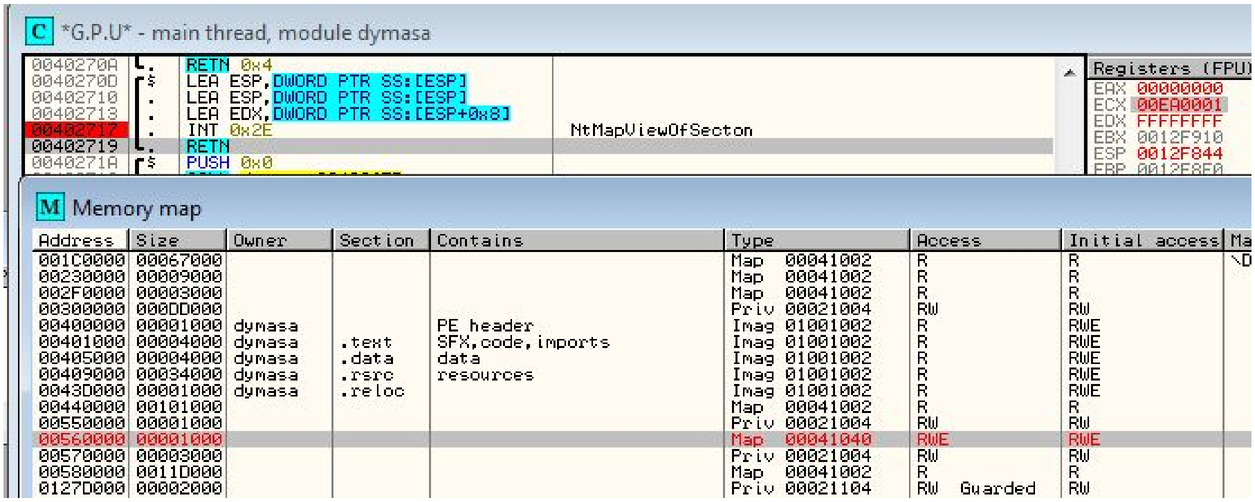
```
0040212E
0040212E make_syscall proc near
0040212E
0040212E arg_4= byte ptr  8
0040212E
0040212E lea     esp, [esp]
00402131 lea     esp, [esp]
00402134 lea     edx, [esp+arg_4]
00402138 int     2Eh                ; DOS 2+ internal - EXECUTE COMMAND
00402138                            ; DS:SI -> counted CR-terminated command string
0040213A retn
0040213A make_syscall endp
```

That's how the functions *NtCreateSection*, *NtMapViewOfSection* and *NtResumeThread* are being called. Those were the missing elements of the API calls' trace, so it explains a lot!

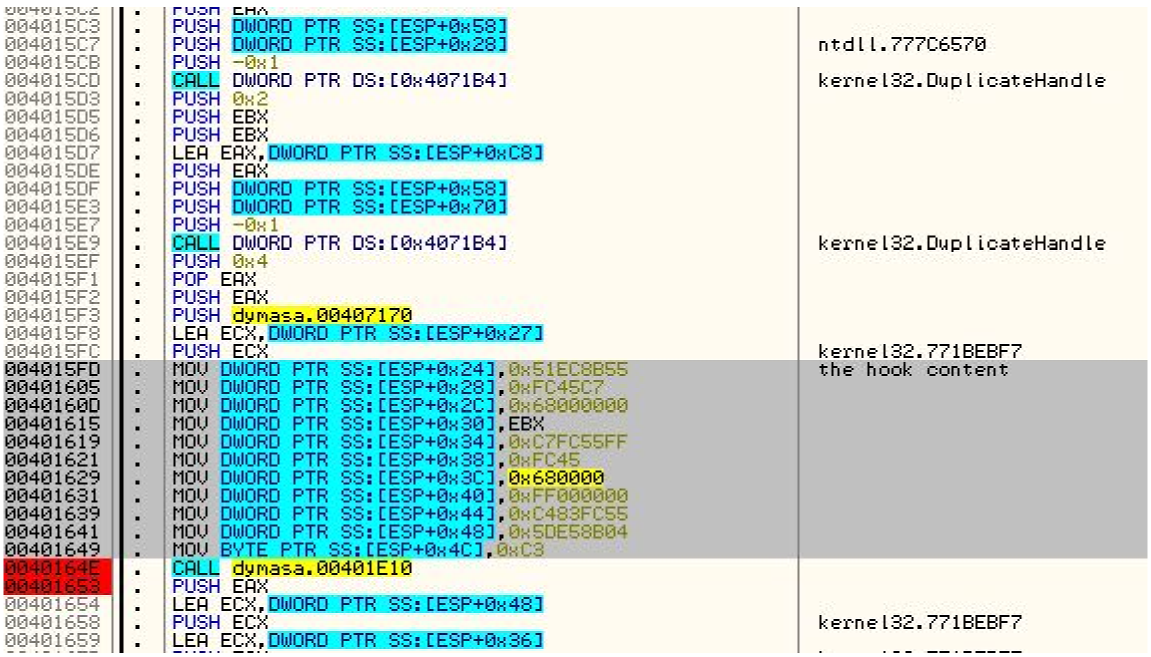
Example 1 – dropper makes a call that is the equivalent of calling the function *NtCreateSection*:



Example 2 – the dropper mapped a section by using a syscall – it is an equivalent of calling the function *NtMapViewOfSection*:



Once the memory is prepared, the shellcode is copied there:



After the preparations, those sections are mapped into the context of the explorer process, that has been created as suspended. Using *SetThreadContext*, it's Entry Point is being redirected to the injected memory page. When the explorer process is being resumed, the new code executes and proceeds with unpacking the malicious core.

At this point of the injection, it's malicious core is not yet revealed – it's decryption process takes place inside the shellcode implanted in the *explorer*. This is also additional countermeasure that this dropper takes against detection tools.

Another trick that this bot uses, is a defense against inline hooking – a method utilized by various monitoring tools. All the mapped DLLs are compared with their raw versions, read from the disk by the dropper. If any anomaly is detected, the dropper overwrites the mapped DLL by the code copied from it's raw version. As a results, the functions are getting "unhooked" and the monitoring programs are loosing the trace on the executed calls. Example from Cuckoo – the unhooking procedure was executed after calling *NtGetThreadContext* – as a result the sandbox lost control over executed calls:

2016-11-07 04:39:06,453	**CreateProcessInternalW**	ApplicationName: `C:\WINDOWS \explorer.exe` ProcessId: `1924` CommandLine: ThreadHandle: `0x000000c4` ProcessHandle: `0x000000c0` ThreadId: `580` CreationFlags: `0x08000004`	success
2016-11-07 04:39:06,453	**NtGetContextThread**	ThreadHandle: `0x000000c4`	success
2016-11-07 04:39:06,674	**__anomaly__**	ThreadIdentifier: `584` Subcategory: `unhook` Message: `Function was unhooked/restored!` FunctionName: `LdrLoadDll`	success
2016-11-07 04:39:06,674	**__anomaly__**	ThreadIdentifier: `584`	success

Conclusion

The illustrated concept is not novel, however it was utilized in an interesting way. Many programs detect malicious activity by monitoring API calls, that are most often misused by malware. Also, applications used for automated analysis hooks API functions, in order to monitor where and how they are being used. The presented method allows to bypass them – at the same time being relatively easy to implement.

In this case, the author didn't use the full potential of the technique, because he could have implement all the injection-related functions via direct syscalls – instead, he chose to use only some subset, related to writing into remote memory area. Some other syscalls has been loaded but not used – it may suggest that the product is still under development. Creation of the new process and changing it’s context still could be detected via API monitoring – and it was enough to rise alerts and make the dropper less stealthy than it was intended.

Appendix

https://www.evilsocket.net/2014/02/11/on-windows-syscall-mechanism-and-syscall-numbers-extraction-methods/ – On Windows Syscall Mechanism and Syscall Numbers Extraction Methods

---

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.