Thinking Outside the Bochs: Code Grafting to Unpack Malware in Emulation

fireeye.com/blog/threat-research/2020/04/code-grafting-to-unpack-malware-in-emulation.html

Threat Research Blog

April 07, 2020 | by Michael Bailey

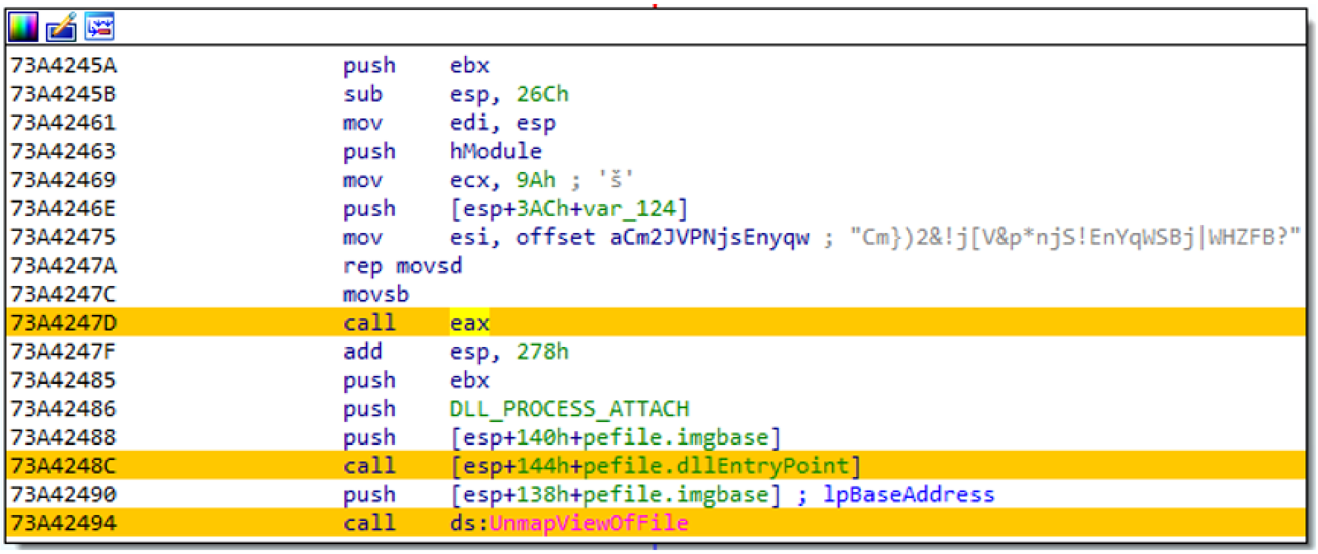
This blog post continues the FLARE script series with a discussion of patching IDA Pro database files (IDBs) to interactively emulate code. While the fastest way to analyze or unpack malware is often to run it, malware won't always successfully execute in a VM. I use IDA Pro's Bochs integration in IDB mode to sidestep tedious debugging scenarios and get quick results. Bochs emulates the opcodes directly from your IDB in a Bochs VM with no OS.

Bochs IDB mode eliminates distractions like switching VMs, debugger setup, neutralizing anti-analysis measures, and navigating the program counter to the logic of interest. Alas, where there is no OS, there can be no loader or dynamic imports. Execution is constrained to opcodes found in the IDB. This precludes emulating routines that call imported string functions or memory allocators. Tom Bennett's flare-emu ships with emulated versions of these, but for off-the-cuff analysis (especially when I don't know if there will be a payoff), I prefer interactively examining registers and memory to adjust my tactics ad hoc.

What if I could bring my own imported functions to Bochs like flare-emu does? I've devised such a technique, and I call it code grafting. In this post I'll discuss the particulars of statically linking stand-ins for common functions into an IDB to get more mileage out of Bochs. I'll demonstrate using this on an EVILNEST sample to unpack and dump next-stage payloads from emulated memory. I'll also show how I copied a tricky call sequence from one IDB to another IDB so I could keep the unpacking process all in a single Bochs debug session.

EVILNEST Scenario

My sample (MD5 hash 37F7F1F691D42DCAD6AE740E6D9CAB63 which is available on VirusTotal) was an EVILNEST variant that populates the stack with configuration data before calling an intermediate payload. Figure 1 shows this unusual call site.



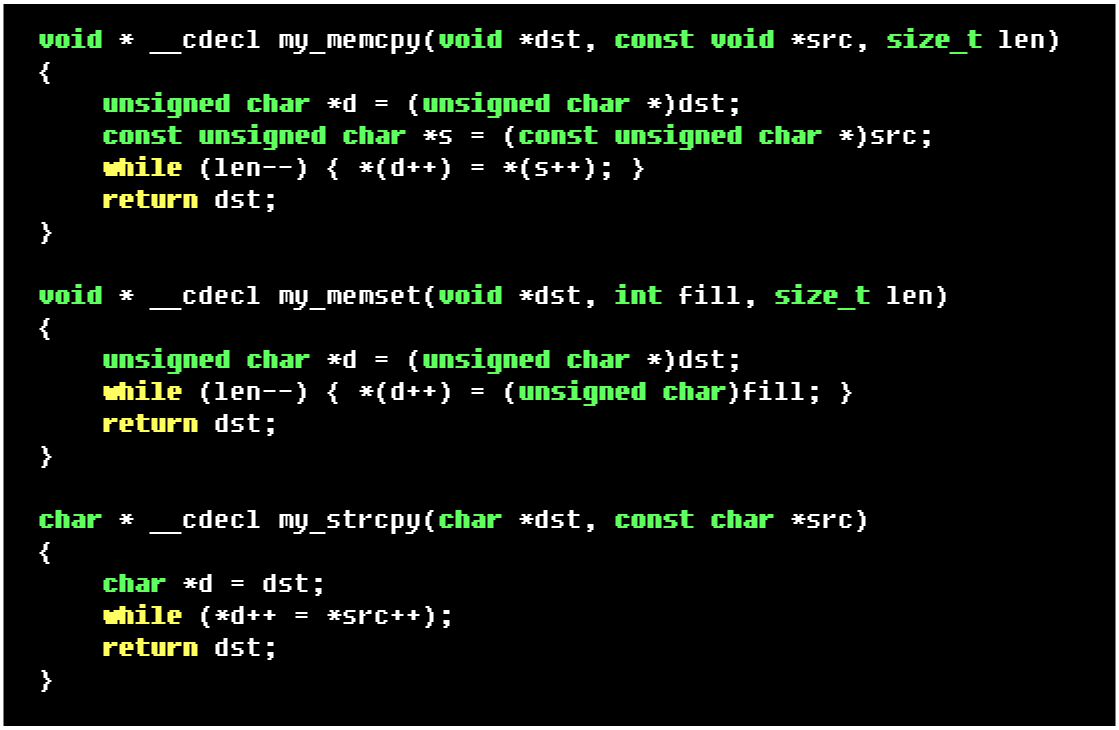
```
73A4245A                push    ebx
73A4245B                sub     esp, 26Ch
73A42461                mov     edi, esp
73A42463                push    hModule
73A42469                mov     ecx, 9Ah ; 'š'
73A4246E                push    [esp+3ACh+var_124]
73A42475                mov     esi, offset aCm2JVPNjsEnyqw ; "Cm})2&!j[V&p*njS!EnYqWSBj|WHZFB?"
73A4247A                rep movsd
73A4247C                movsb
73A4247D                call    eax
73A4247F                add     esp, 278h
73A42485                push    ebx
73A42486                push    DLL_PROCESS_ATTACH
73A42488                push    [esp+140h+pefile.imgbase]
73A4248C                call    [esp+144h+pefile.dllEntryPoint]
73A42490                push    [esp+138h+pefile.imgbase] ; lpBaseAddress
73A42494                call    ds:UnmapViewOfFile
```

Figure 1: Call site for intermediate payload

The code in Figure 1 executes in a remote thread within a hollowed-out iexplore.exe process; the malware uses anti-analysis tactics as well. I had the intermediate payload stage and wanted to unpack next-stage payloads without managing a multi-process debugging scenario with anti-analysis. I knew I could stub out a few function calls in the malware to run all of the relevant logic in Bochs. Here's how I did it.

Code Carving

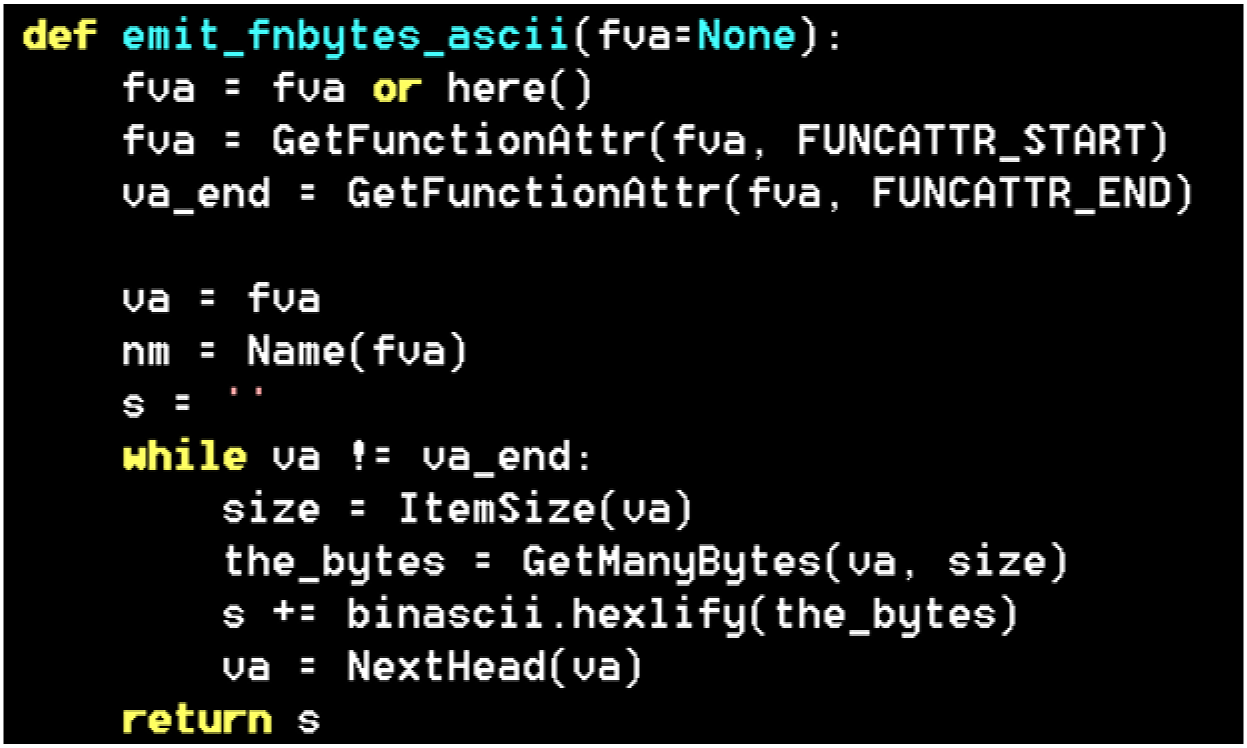
I needed opcodes for a few common functions to inject into my IDBs and emulate in Bochs. I built simple C implementations of selected functions and compiled them into one binary. Figure 2 shows some of these stand-ins.



```
void * __cdecl my_memcpy(void *dst, const void *src, size_t len)
{
    unsigned char *d = (unsigned char *)dst;
    const unsigned char *s = (const unsigned char *)src;
    while (len--) { *(d++) = *(s++); }
    return dst;
}

void * __cdecl my_memset(void *dst, int fill, size_t len)
{
    unsigned char *d = (unsigned char *)dst;
    while (len--) { *(d++) = (unsigned char)fill; }
    return dst;
}

char * __cdecl my_strcpy(char *dst, const char *src)
{
    char *d = dst;
    while (*d++ = *src++);
    return dst;
}
```

Figure 2: Simple implementations of common functions

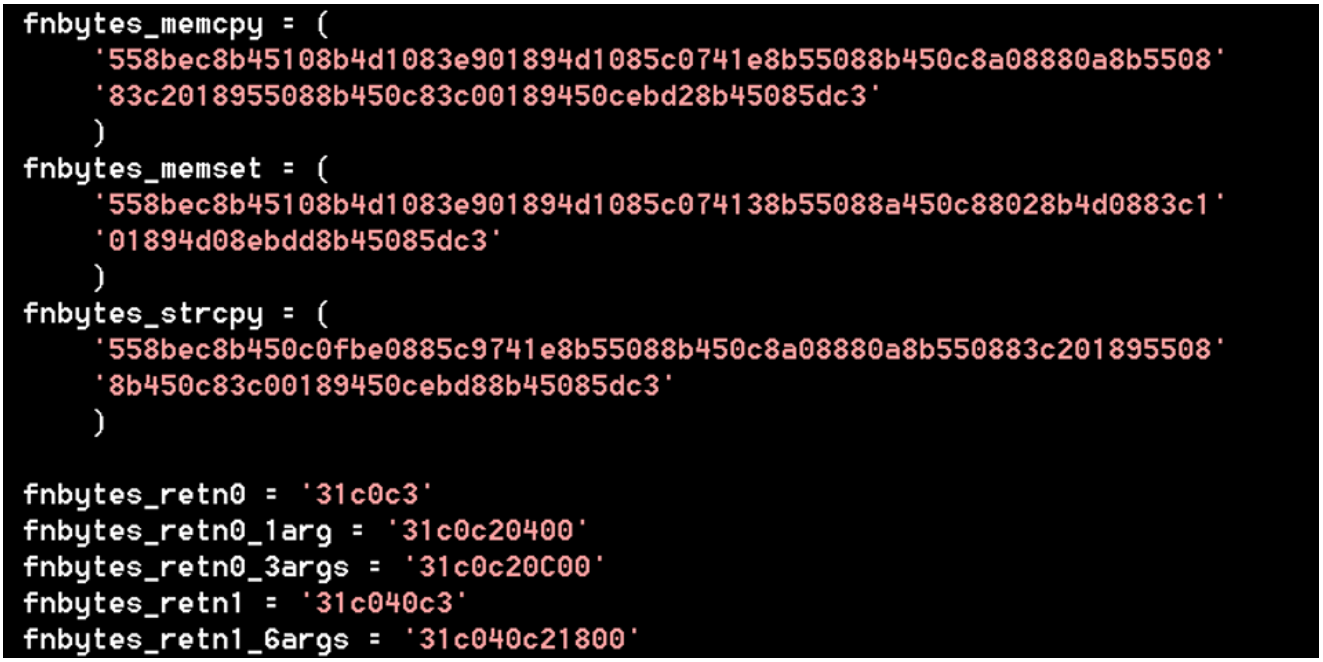
I compiled this and then used IDAPython code similar to Figure 3 to extract the function opcode bytes.



```
def emit_fnbytes_ascii(fva=None):
    fva = fva or here()
    fva = GetFunctionAttr(fva, FUNCATTR_START)
    va_end = GetFunctionAttr(fva, FUNCATTR_END)

    va = fva
    nm = Name(fva)
    s = ''
    while va != va_end:
        size = ItemSize(va)
        the_bytes = GetManyBytes(va, size)
        s += binascii.hexlify(the_bytes)
        va = NextHead(va)
    return s
```

Figure 3: Function extraction

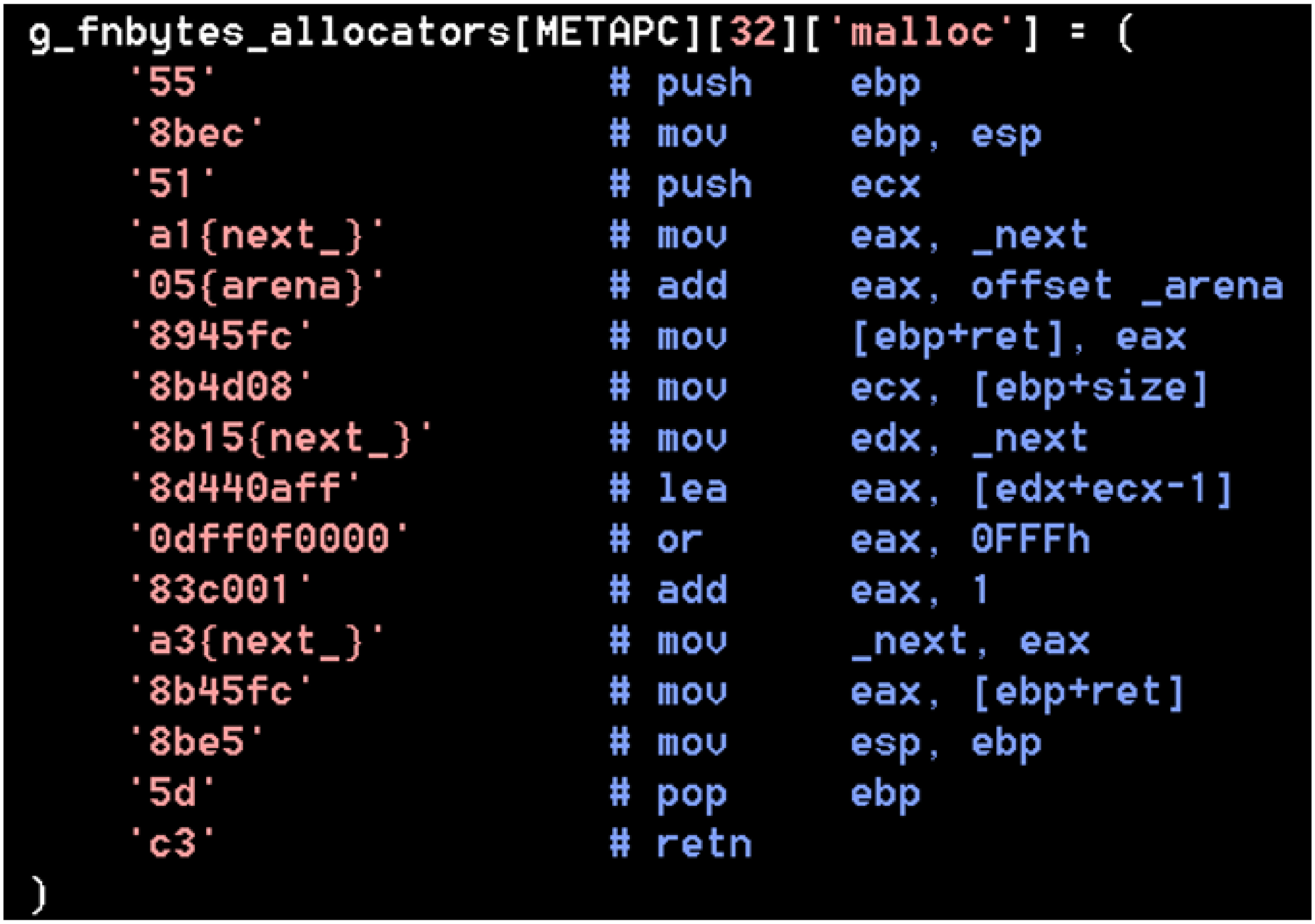
I curated a library of function opcodes in an IDAPython script as shown in Figure 4. The nonstandard function opcodes at the bottom of the figure were hand-assembled as tersely as possible to generically return specific values and manipulate the stack (or not) in conformance with calling conventions.



```
fnbytes_memcpy = (
    '558bec8b45108b4d1083e901894d1085c0741e8b55088b450c8a08880a8b5508'
    '83c2018955088b450c83c00189450cebd28b45085dc3'
    )
fnbytes_memset = (
    '558bec8b45108b4d1083e901894d1085c074138b55088a450c88028b4d0883c1'
    '01894d08ebdd8b45085dc3'
    )
fnbytes_strcpy = (
    '558bec8b450c0fbe0885c9741e8b55088b450c8a08880a8b550883c201895508'
    '8b450c83c00189450cebd88b45085dc3'
    )

fnbytes_retn0 = '31c0c3'
fnbytes_retn0_1arg = '31c0c20400'
fnbytes_retn0_3args = '31c0c20C00'
fnbytes_retn1 = '31c040c3'
fnbytes_retn1_6args = '31c040c21800'
```

Figure 4: Extracted function opcodes

On top of simple functions like memcpy, I implemented a memory allocator. The allocator referenced global state data, meaning I couldn’t just inject it into an IDB and expect it to work. I read the disassembly to find references to global operands and templatize them for use with Python’s format method. Figure 5 shows an example for malloc.



```
g_fnbytes_allocators[METAPC][32]['malloc'] = (
    '55'                # push    ebp
    '8bec'              # mov     ebp, esp
    '51'                # push    ecx
    'a1{next_}'         # mov     eax, _next
    '05{arena}'         # add     eax, offset _arena
    '8945fc'            # mov     [ebp+ret], eax
    '8b4d08'            # mov     ecx, [ebp+size]
    '8b15{next_}'       # mov     edx, _next
    '8d440aff'          # lea     eax, [edx+ecx-1]
    '0dff0f0000'        # or      eax, 0FFFh
    '83c001'            # add     eax, 1
    'a3{next_}'         # mov     _next, eax
    '8b45fc'            # mov     eax, [ebp+ret]
    '8be5'              # mov     esp, ebp
    '5d'                # pop     ebp
    'c3'                # retn
)
```

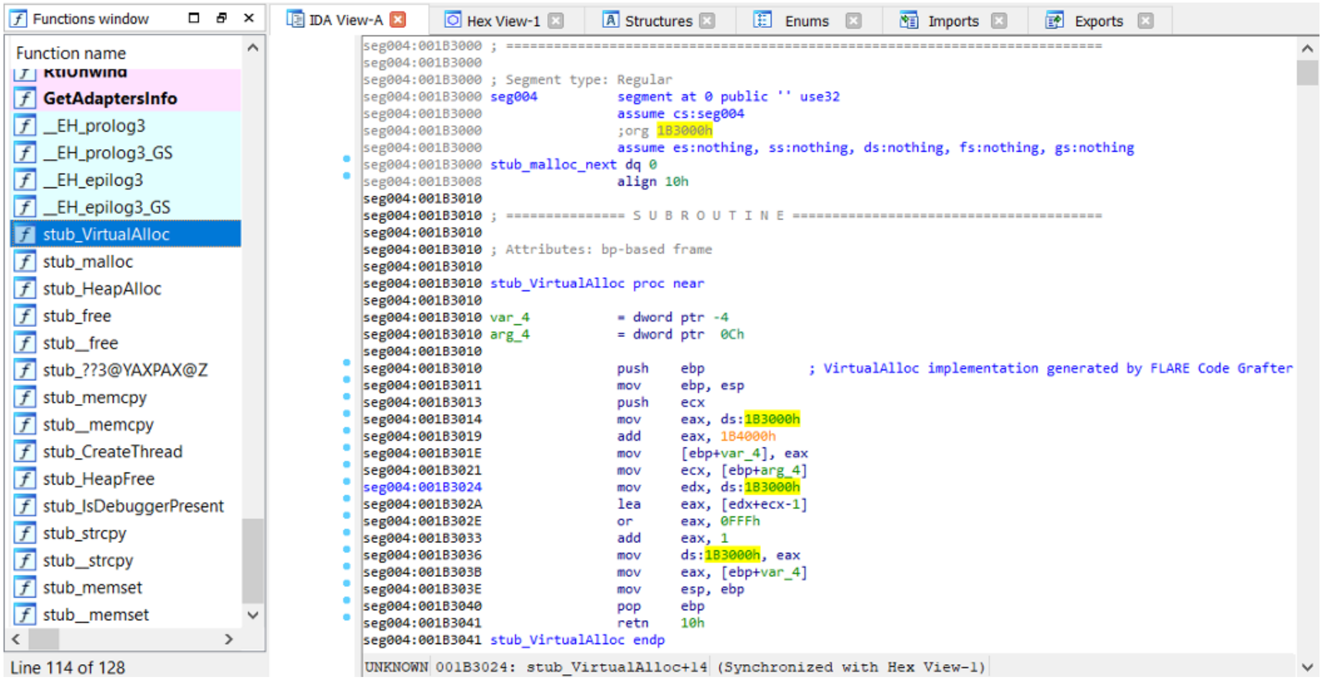
Figure 5: HeapAlloc template code

I organized the stubs by name as shown in Figure 6 both to call out functions I would need to patch, and to conveniently add more function stubs as I encounter use cases for them. The mangled name I specified as an alias for free is operator delete.

```
stubs = {
    ('IsDebuggerPresent',): fnbytes_retn0,
    ('CreateThread',): fnbytes_retn1_6args,
    ('free', '_free', '??3@YAXPAX@Z'): fnbytes_retn0,
    ('HeapFree',): fnbytes_retn0_3args,
    ('strcpy', '_strcpy'): fnbytes_strcpy,
    ('memcpy', '_memcpy'): fnbytes_memcpy,
    ('memset', '_memset'): fnbytes_memset,
}
```

Figure 6: Function stubs and associated names

To inject these functions into the binary, I wrote code to find the next available segment of a given size. I avoided occupying low memory because Bochs places its loader segment below 0x10000. Adjacent to the code in my code segment, I included space for the data used by my memory allocator. Figure 7 shows the result of patching these functions and data into the IDB and naming each location (stub functions are prefixed with stub_).

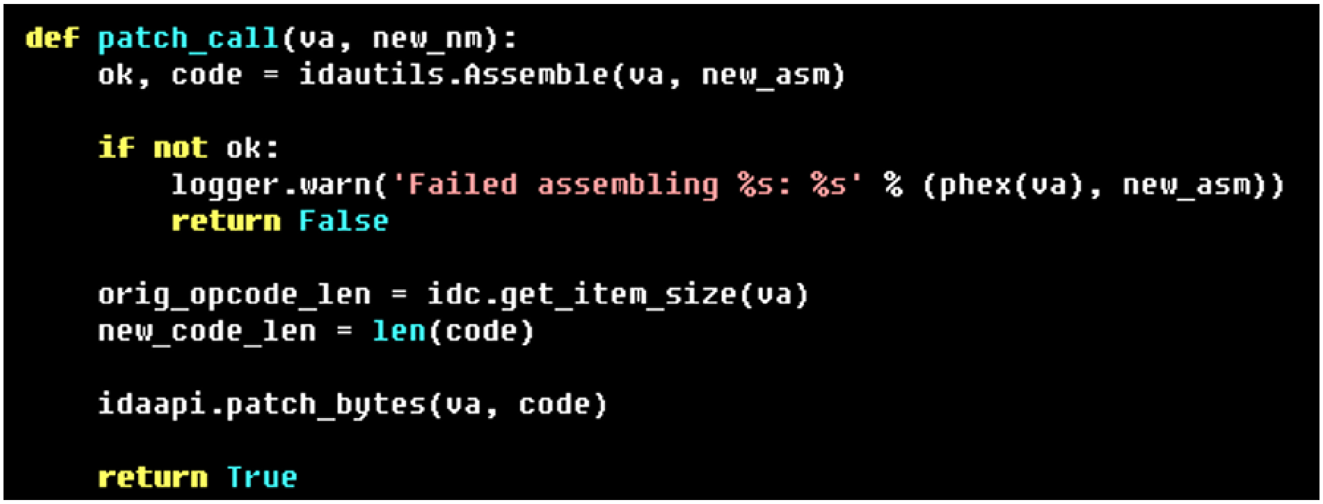


RtlUnwind
GetAdaptersInfo
_EH_prolog3
_EH_prolog3_GS
_EH_epilog3
_EH_epilog3_GS
stub_VirtualAlloc
stub_malloc
stub_HeapAlloc
stub_free
stub__free
stub_??3@YAXPAX@Z
stub_memcpy
stub__memcpy
stub_CreateThread
stub_HeapFree
stub_IsDebuggerPresent
stub_strcpy
stub__strcpy
stub_memset
stub__memset

```
seg004:001B3000 ; ===========================================================================
seg004:001B3000
seg004:001B3000 ; Segment type: Regular
seg004:001B3000 seg004          segment at 0 public '' use32
seg004:001B3000                 assume cs:seg004
seg004:001B3000                 ;org 1B3000h
seg004:001B3000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg004:001B3000 stub_malloc_next dq 0
seg004:001B3008                 align 10h
seg004:001B3010
seg004:001B3010 ; =============== S U B R O U T I N E =======================================
seg004:001B3010
seg004:001B3010 ; Attributes: bp-based frame
seg004:001B3010
seg004:001B3010 stub_VirtualAlloc proc near
seg004:001B3010
seg004:001B3010 var_4           = dword ptr -4
seg004:001B3010 arg_4           = dword ptr  0Ch
seg004:001B3010
seg004:001B3010                 push    ebp             ; VirtualAlloc implementation generated by FLARE Code Grafter
seg004:001B3011                 mov     ebp, esp
seg004:001B3013                 push    ecx
seg004:001B3014                 mov     eax, ds:1B3000h
seg004:001B3019                 add     eax, 1B4000h
seg004:001B301E                 mov     [ebp+var_4], eax
seg004:001B3021                 mov     ecx, [ebp+arg_4]
seg004:001B3024                 mov     edx, ds:1B3000h
seg004:001B302A                 lea     eax, [edx+ecx-1]
seg004:001B302E                 or      eax, 0FFFh
seg004:001B3033                 add     eax, 1
seg004:001B3036                 mov     ds:1B3000h, eax
seg004:001B303B                 mov     eax, [ebp+var_4]
seg004:001B303E                 mov     esp, ebp
seg004:001B3040                 pop     ebp
seg004:001B3041                 retn    10h
seg004:001B3041 stub_VirtualAlloc endp
UNKNOWN 001B3024: stub_VirtualAlloc+14 (Synchronized with Hex View-1)
```

Figure 7: Data and code injected into IDB

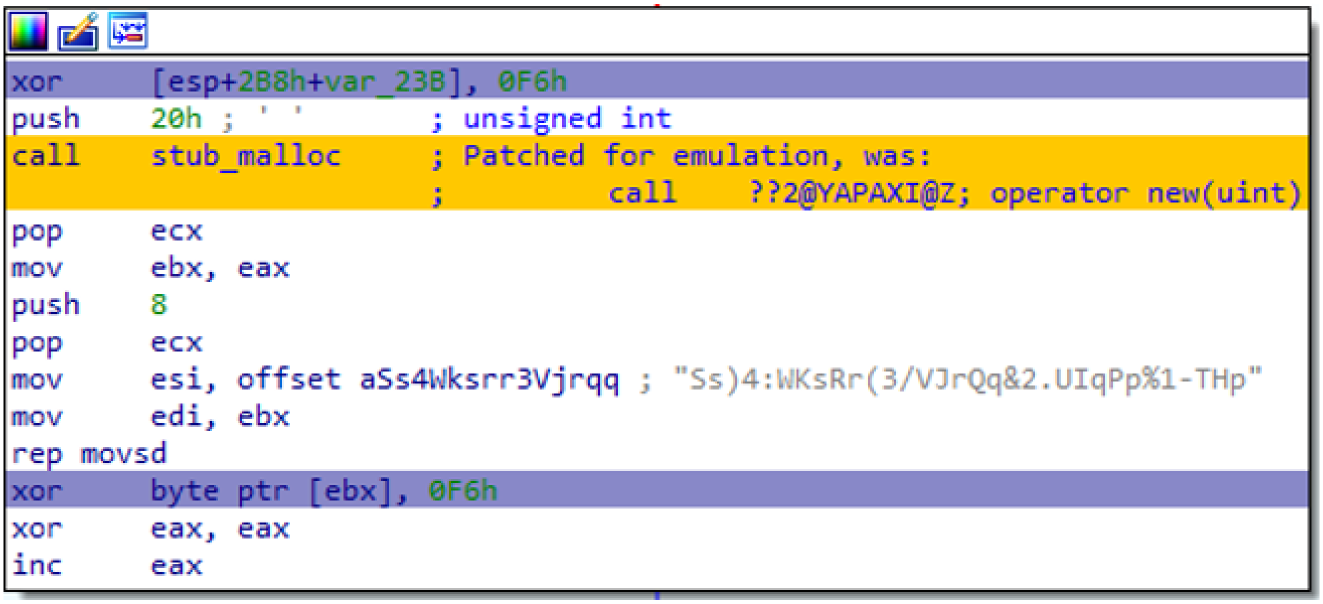
The script then iterates all the relevant calls in the binary and patches them with calls to their stub implementations in the newly added segment. As shown in Figure 8, IDAPython's Assemble function saved the effort of calculating the offset for the call operand manually. Note that the Assemble function worked well here, but for bigger tasks, Hex-Rays recommends a dedicated assembler such as Keystone Engine and its Keypatch plugin for IDA Pro.



```
def patch_call(va, new_nm):
    ok, code = idautils.Assemble(va, new_asm)

    if not ok:
        logger.warn('Failed assembling %s: %s' % (phex(va), new_asm))
        return False

    orig_opcode_len = idc.get_item_size(va)
    new_code_len = len(code)

    idaapi.patch_bytes(va, code)

    return True
```

Figure 8: Abbreviated routine for assembling a call instruction and patching a call site to an import

The Code Grafting script updated all the relevant call sites to resemble Figure 9, with the target functions being replaced by calls to the stub_ implementations injected earlier. This prevented Bochs in IDB mode from getting derailed when hitting these call sites, because the call operands now pointed to valid code inside the IDB.



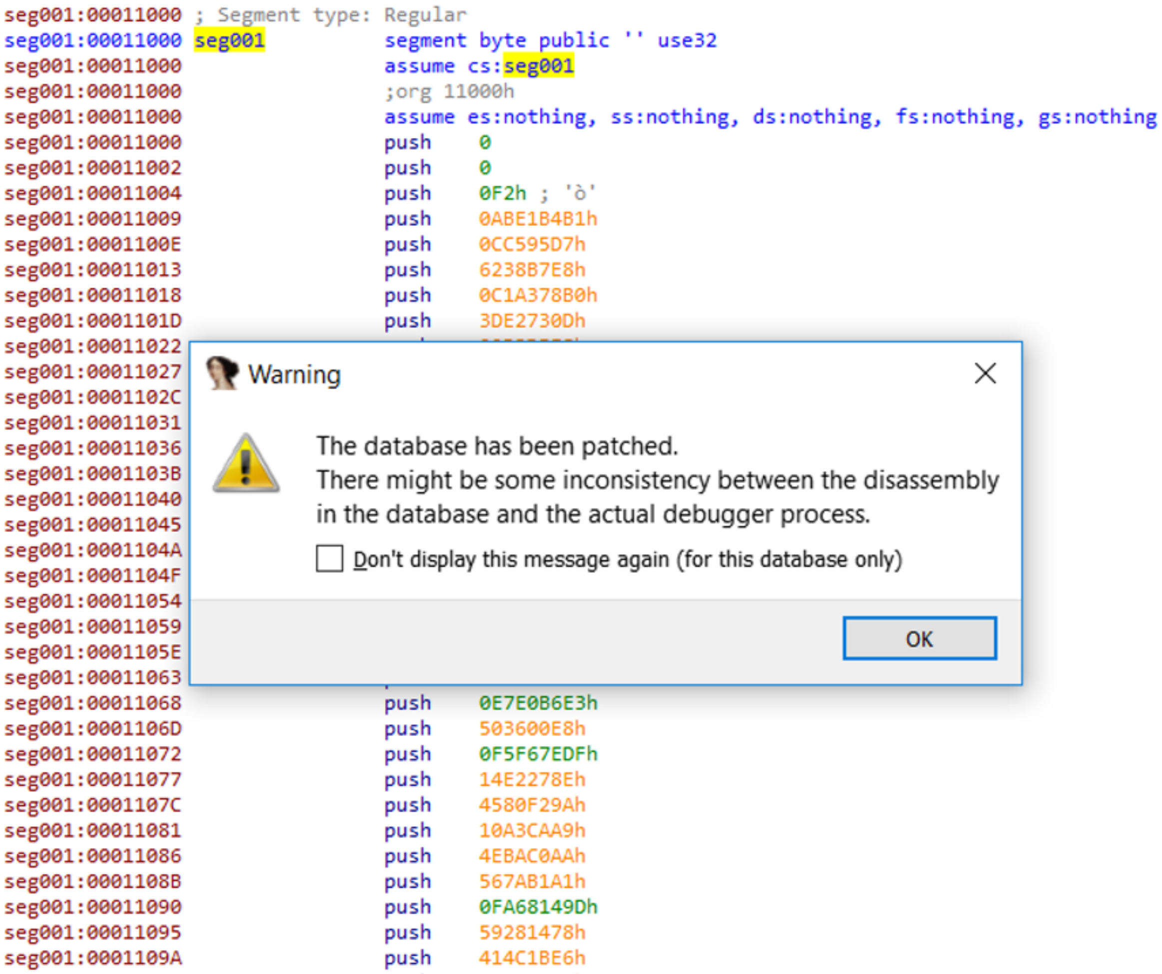
```
xor     [esp+2B8h+var_23B], 0F6h
push    20h ; ' '           ; unsigned int
call    stub_malloc         ; Patched for emulation, was:
                            ;           call    ??2@YAPAXI@Z; operator new(uint)
pop     ecx
mov     ebx, eax
push    8
pop     ecx
mov     esi, offset aSs4Wksrr3Vjrqq ; "Ss)4:WKsRr(3/VJrQq&2.UIqPp%1-THp"
mov     edi, ebx
rep movsd
xor     byte ptr [ebx], 0F6h
xor     eax, eax
inc     eax
```

Figure 9: Patched operator new() call site

Dealing with EVILNEST

The debug scenario for the dropper was slightly inconvenient, and simultaneously, it was setting up a very unusual call site for the payload entry point. I used Bochs to execute the dropper until it placed the configuration data on the stack, and then I used IDAPython's idc.get_bytes function to extract the resulting stack data. I wrote IDAPython script code to iterate the stack data and assemble push instructions into the payload IDB leading up to a call instruction pointing to the DLL's export. This allowed me to debug the unpacking process from Bochs within a single session.

I clicked on the beginning of my synthesized call site and hit F4 to run it in Bochs. I was greeted with the warning in Figure 10 indicating that the patched IDB would not match the depictions made by the debugger (which is untrue in the case of Bochs IDB mode). Bochs faithfully executed my injected opcodes producing exactly the desired result.



```
seg001:00011000 ; Segment type: Regular
seg001:00011000 seg001          segment byte public '' use32
seg001:00011000                 assume cs:seg001
seg001:00011000                 ;org 11000h
seg001:00011000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg001:00011000                 push    0
seg001:00011002                 push    0
seg001:00011004                 push    0F2h ; 'ò'
seg001:00011009                 push    0ABE1B4B1h
seg001:0001100E                 push    0CC595D7h
seg001:00011013                 push    6238B7E8h
seg001:00011018                 push    0C1A378B0h
seg001:0001101D                 push    3DE2730Dh
seg001:00011022
seg001:00011027
seg001:0001102C
seg001:00011031
seg001:00011036
seg001:0001103B
seg001:00011040
seg001:00011045
seg001:0001104A
seg001:0001104F
seg001:00011054
seg001:00011059
seg001:0001105E
seg001:00011063
seg001:00011068                 push    0E7E0B6E3h
seg001:0001106D                 push    503600E8h
seg001:00011072                 push    0F5F67EDFh
seg001:00011077                 push    14E2278Eh
seg001:0001107C                 push    4580F29Ah
seg001:00011081                 push    10A3CAA9h
seg001:00011086                 push    4EBAC0AAh
seg001:0001108B                 push    567AB1A1h
seg001:00011090                 push    0FA68149Dh
seg001:00011095                 push    59281478h
seg001:0001109A                 push    414C1BE6h
```

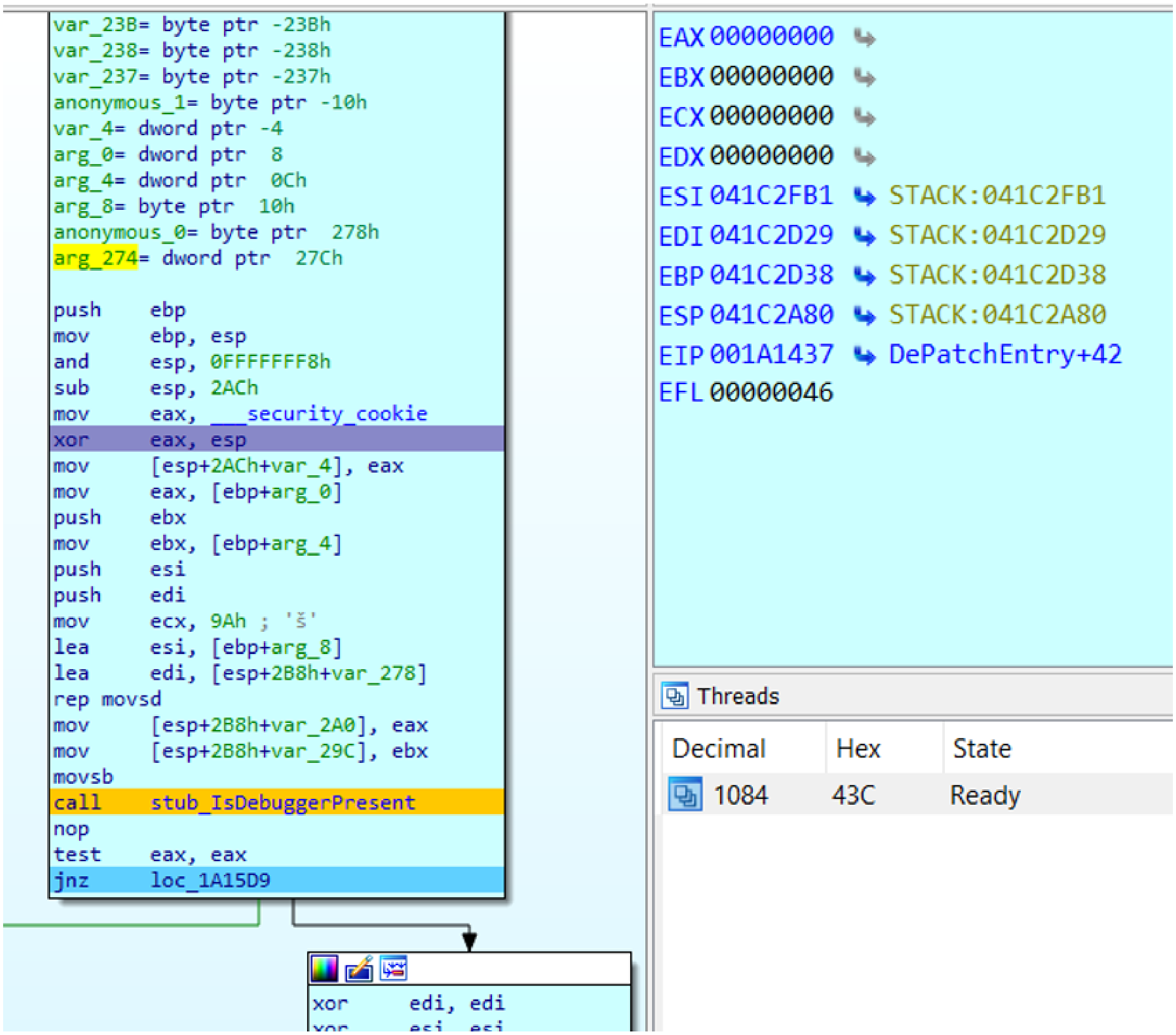
Warning

The database has been patched.
There might be some inconsistency between the disassembly in the database and the actual debugger process.

☐ Don't display this message again (for this database only)

Figure 10: Patch warning

I watched carefully as the instruction pointer approached and passed the IsDebuggerPresent check. Because of the stub I injected (stub_IsDebuggerPresent), it passed the check returning zero as shown in Figure 11.



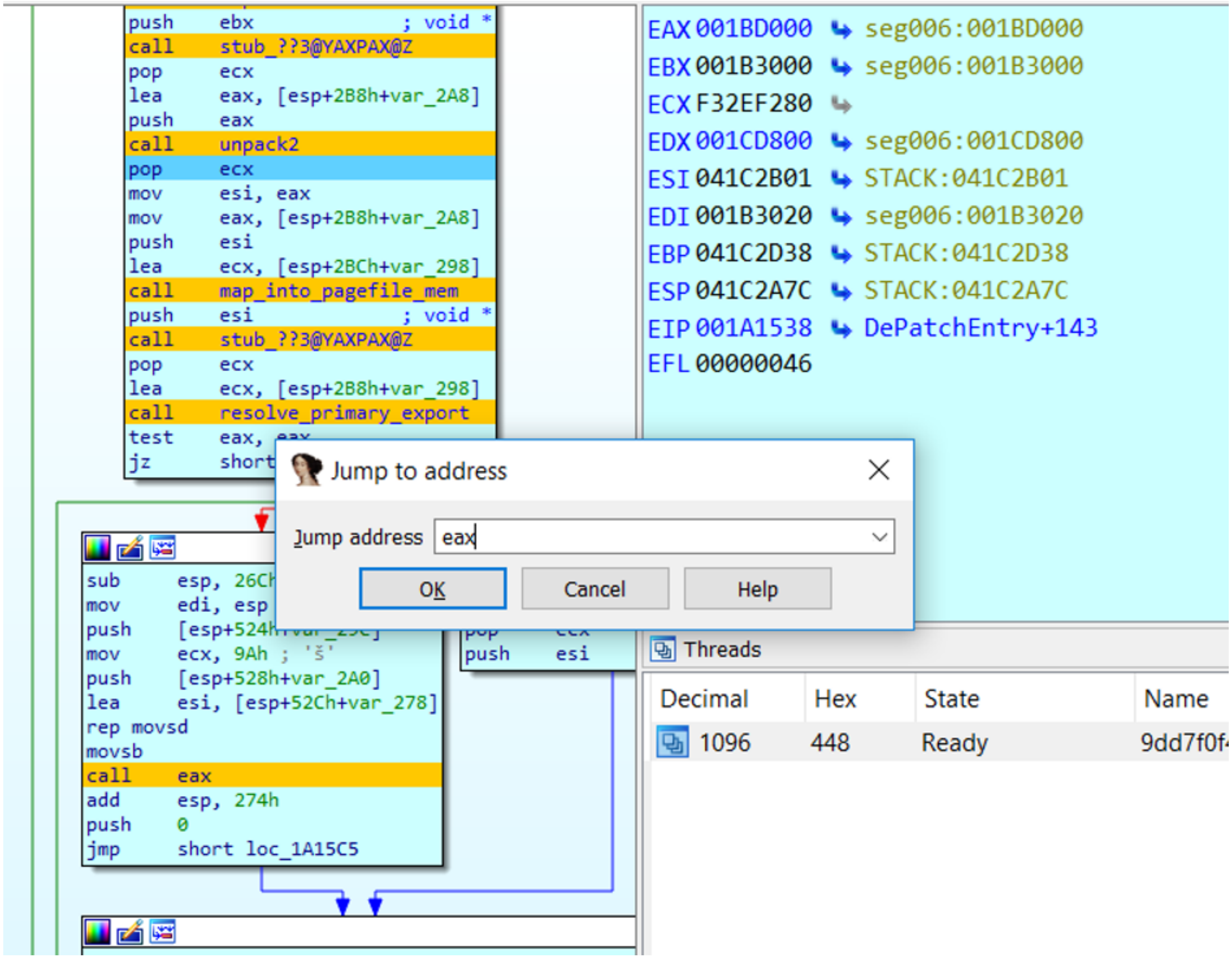
```
var_23B= byte ptr -23Bh
var_238= byte ptr -238h
var_237= byte ptr -237h
anonymous_1= byte ptr -10h
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= byte ptr  10h
anonymous_0= byte ptr  278h
arg_274= dword ptr  27Ch

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
sub     esp, 2ACh
mov     eax, ___security_cookie
xor     eax, esp
mov     [esp+2ACh+var_4], eax
mov     eax, [ebp+arg_0]
push    ebx
mov     ebx, [ebp+arg_4]
push    esi
push    edi
mov     ecx, 9Ah ; 'š'
lea     esi, [ebp+arg_8]
lea     edi, [esp+2B8h+var_278]
rep movsd
mov     [esp+2B8h+var_2A0], eax
mov     [esp+2B8h+var_29C], ebx
movsb
call    stub_IsDebuggerPresent
nop
test    eax, eax
jnz     loc_1A15D9

xor     edi, edi
```

```
EAX 00000000
EBX 00000000
ECX 00000000
EDX 00000000
ESI 041C2FB1  STACK:041C2FB1
EDI 041C2D29  STACK:041C2D29
EBP 041C2D38  STACK:041C2D38
ESP 041C2A80  STACK:041C2A80
EIP 001A1437  DePatchEntry+42
EFL 00000046
```

Decimal	Hex	State
1084	43C	Ready

Figure 11: Passing up IsDebuggerPresent

I allowed the program counter to advance to address 0x1A1538, just beyond the unpacking routine. Figure 12 shows the register state at this point which reflects a value in EAX that was handed out by my fake heap allocator and which I was about to visit.



```
push    ebx                     ; void *
call    stub_??3@YAXPAX@Z
pop     ecx
lea     eax, [esp+2B8h+var_2A8]
push    eax
call    unpack2
pop     ecx
mov     esi, eax
mov     eax, [esp+2B8h+var_2A8]
push    esi
lea     ecx, [esp+2BCh+var_298]
call    map_into_pagefile_mem
push    esi                     ; void *
call    stub_??3@YAXPAX@Z
pop     ecx
lea     ecx, [esp+2B8h+var_298]
call    resolve_primary_export
test    eax,
jz      short

sub     esp, 26C
mov     edi, esp
push    [esp+524h
mov     ecx, 9Ah ; 'š'
push    [esp+528h+var_2A0]
lea     esi, [esp+52Ch+var_278]
rep movsd
movsb
call    eax
add     esp, 274h
push    0
jmp     short loc_1A15C5

push    esi
```

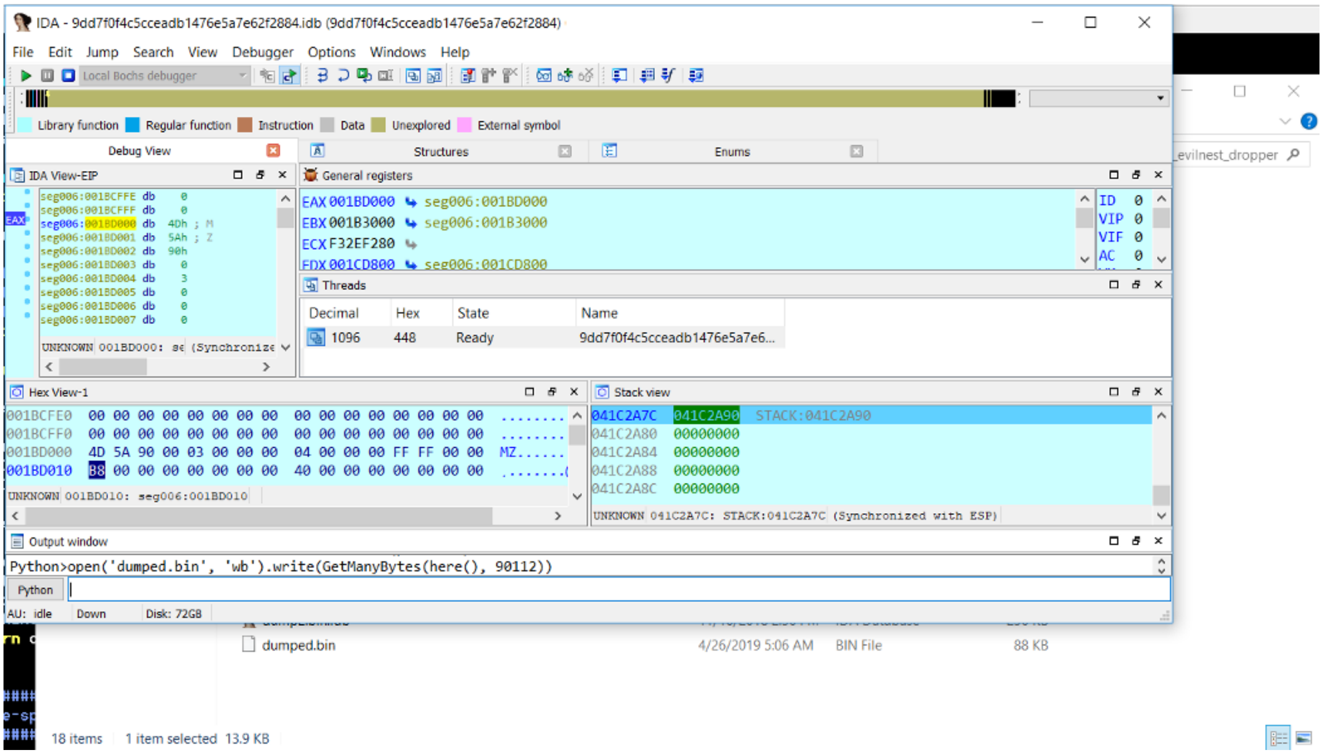
```
EAX 001BD000 ↳ seg006:001BD000
EBX 001B3000 ↳ seg006:001B3000
ECX F32EF280 ↳
EDX 001CD800 ↳ seg006:001CD800
ESI 041C2B01 ↳ STACK:041C2B01
EDI 001B3020 ↳ seg006:001B3020
EBP 041C2D38 ↳ STACK:041C2D38
ESP 041C2A7C ↳ STACK:041C2A7C
EIP 001A1538 ↳ DePatchEntry+143
EFL 00000046
```

Jump address: eax

Decimal	Hex	State	Name
1096	448	Ready	9dd7f0f

Figure 12: Running to the end of the unpacker and preparing to view the result

Figure 13 shows that there was indeed an IMAGE_DOS_SIGNATURE (“MZ”) at this location. I used idc.get_bytes() to dump the unpacked binary from the fake heap location and saved it for analysis.



```
seg006:001BCFFE db    0
seg006:001BCFFF db    0
seg006:001BD000 db  4Dh ; M
seg006:001BD001 db  5Ah ; Z
seg006:001BD002 db  90h
seg006:001BD003 db    0
seg006:001BD004 db    3
seg006:001BD005 db    0
seg006:001BD006 db    0
seg006:001BD007 db    0
UNKNOWN 001BD000: se (Synchronize
```

```
EAX 001BD000 seg006:001BD000
EBX 001B3000 seg006:001B3000
ECX F32EF280
EDX 001CD800 seg006:001CD800
ID  0
VIP 0
VIF 0
AC  0
```

Decimal	Hex	State	Name
1096	448	Ready	9dd7f0f4c5cceadb1476e5a7e6...

```
001BCFE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........
001BCFF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........
001BD000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ......
001BD010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  .......(
UNKNOWN 001BD010: seg006:001BD010
```

```
041C2A7C  041C2A90  STACK:041C2A90
041C2A80  00000000
041C2A84  00000000
041C2A88  00000000
041C2A8C  00000000
UNKNOWN 041C2A7C: STACK:041C2A7C (Synchronized with ESP)
```

```
Python>open('dumped.bin', 'wb').write(GetManyBytes(here(), 90112))
```

dumped.bin 4/26/2019 5:06 AM BIN File 88 KB

Figure 13: Dumping the unpacked binary

Through Bochs IDB mode, I was also able to use the interactive debugger interface of IDA Pro to experiment with manipulating execution and traversing a different branch to unpack another payload for this malware as well.

Conclusion

Although dynamic analysis is sometimes the fastest road, setting it up and navigating minutia detract from my focus, so I've developed an eye for routines that I can likely emulate in Bochs to dodge those distractions while still getting answers. Injecting code into an IDB broadens the set of functions that I can do this with, letting me get more out of Bochs. This in turn lets me do more on-the-fly experimentation, one-off string decodes, or validation of hypotheses before attacking something at scale. It also allows me to experiment dynamically with samples that won't load correctly anyway, such as unpacked code with damaged or incorrect PE headers.

I've shared the Code Grafting tools as part of the flare-ida GitHub repository. To use this for your own analyses:

1. In IDA Pro's IDAPython prompt, run code_grafter.py or import it as a module.
2. Instantiate a CodeGrafter object and invoke its graftCodeToIdb() method:

   CodeGrafter().graftCodeToIdb()
3. Use Bochs in IDB mode to conveniently execute your modified sample and experiment away!

This post makes it clear just how far I’ll go to avoid breaking eye contact with IDA. If you’re a fan of using Bochs with IDA too, then this is my gift to you. Enjoy!

Previous Post
Next Post