Weekend Project: A Custom IDA Loader Module for the Hidden Bee Malware Family

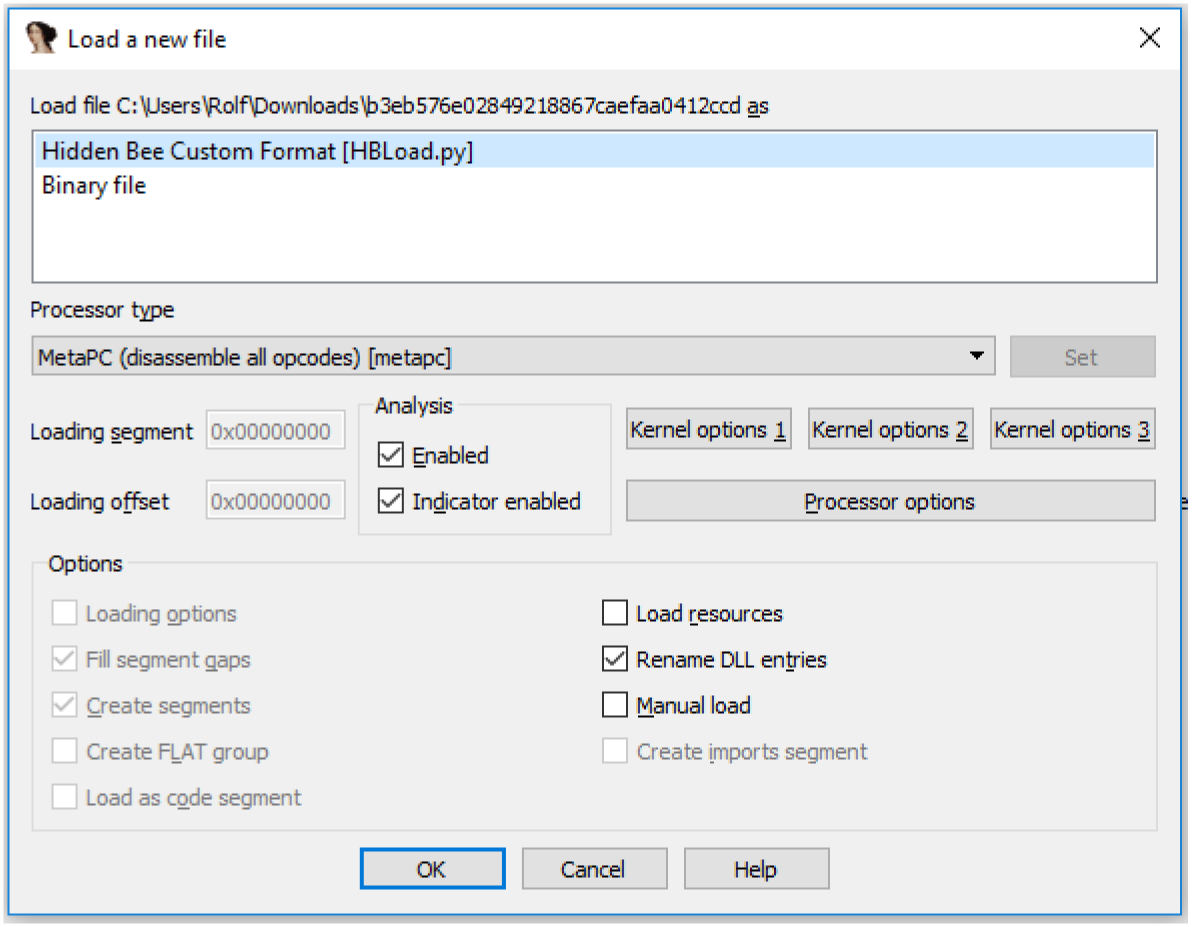
msreverseengineering.com/blog/2018/9/2/weekend-project-a-custom-ida-loader-module-for-the-hidden-bee-malware-family

September 2, 2018

September 2, 2018 Rolf Rolles

Here's a half-day project that I did this weekend for my own edification. Perhaps someone will benefit from the source code in the future.

While reading hasherezade's research on the Hidden Bee malware family's custom file format (samples here), I was struck with the thought that this use-case seemed particularly well-suited for an IDA custom loader module. The IDA loader module approach has a few advantages over the previous approach: it's fully automated, requiring no additional programs, plugins, or scripts; the imports have proper names and type information, allowing IDA's ordinary P.I.T. algorithms to propagate the information; and the user can relocate the database to an arbitrary base address.


Load a new file

Load file C:\Users\Rolf\Downloads\b3eb576e02849218867caefaa0412ccd as

Hidden Bee Custom Format [HBLoad.py]
Binary file

Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0x00000000
Loading offset: 0x00000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [ ] Loading options
- [✓] Fill segment gaps
- [✓] Create segments
- [ ] Create FLAT group
- [ ] Load as code segment
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [ ] Create imports segment

Given that custom loaders are the only variety of IDA plugin that I haven't yet written, this seemed like a nice small-scope project for the weekend to round out my knowledge. My very minor contribution with this entry is the IDA custom loader for the Hidden Bee format, which can be found on my GitHub. The IDAPython code requires that Ero Carrera's pefile module be installed, say via pip.

Hidden Bee

In brief, the Hidden Bee malware family distributes payloads in a customized file format, which is a majorly stripped-down version of the PE file format. You can see all of the details in hasherezade's write-up. I did no original malware analysis for this project; I merely read her blog entry, figured out how to convert the details into a loader plugin, and then debugged it against the sample links she gave. As usual, Chris Eagle's The IDA Pro Book, 2nd Edition was useful. Some details about the loader API have changed with the IDA 7.x API port, but Hex-Rays' porting guide was informative, and the loader examples in the IDA 7.1 SDK have also been ported to the newest API.

IDA Loader Modules in Brief

An IDA loader module is simply an IDA plugin with a well-defined interface. IDA loader modules will be called when loading any file into IDA. They have two primary responsibilities:

1. Given access to the bytes of a file, determine whether the file is of a format that the loader module can handle. Every IDA loader module must export a function named accept_file for this purpose. This function returns 0 if it can't recognize the file format, or a non-zero value if it can.
2. If the file type can be loaded by the module, and the user chooses to use this module to load the file, perform the actual loading process e.g. creating segments within the IDB, copying bytes out of the file into the segments, processing relocations, parsing imports, adding entrypoints, and so on. Every IDA loader module must export a function named load_file for this purpose.

Both of these functions take as input an "linput_t *" object that behaves like a C FILE * object, which supports seeking to specified positions, reading byte arrays out of the file, and so on. Since Hidden Bee's format includes relocations, I chose to implement a third, optional IDA loader module function: move_segm. This function will be called by the IDA kernel when the user requests that the database be relocated to another address.

Writing a Loader Module for Hidden Bee

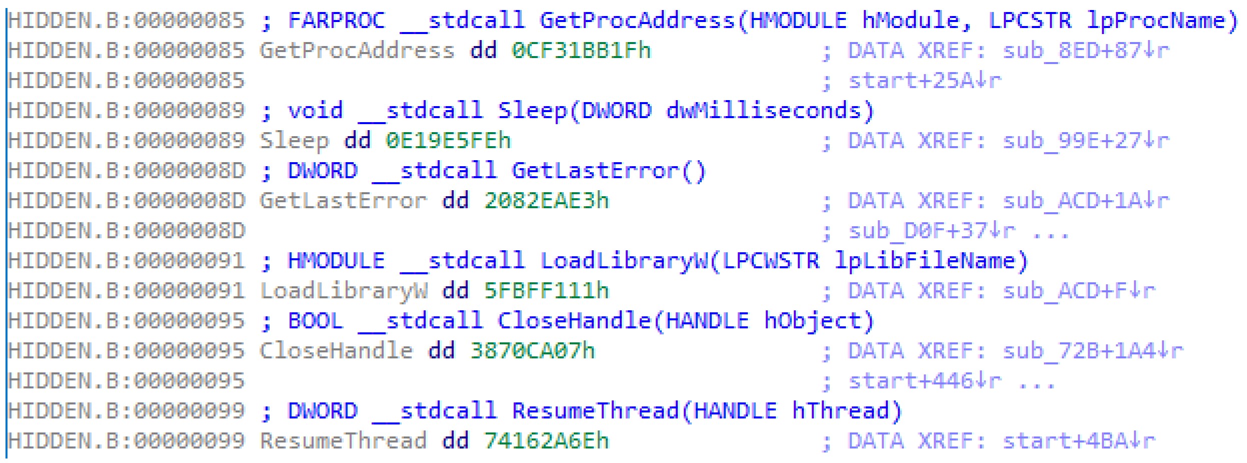
After reading the aforementioned write-up, I figured that the only difficulties in loading Hidden Bee images in IDA would be A) that the Hidden Bee customized header specifies API imports via hash rather than by name, and B) that it includes relocation information.

Relocations and import lookup via hash are simple enough conceptually, but the precise details about how best to integrate them with IDA are not obvious. Sadly, I did not feel confident in these tasks even after reading the loader module examples in the SDK. Four out of the five hours I spent on this project were reverse engineering %IDADIR%\loaders\pe.dll -- the loader module for the PE file format -- focusing in particular on its handling of relocations and imports. As expected, the results are idiosyncratic and I don't expect them to generalize well.

Imports

For dealing with the imports by hash, hasherezade's toolchain ultimately generates a textual file with the addresses of the import hash names and their corresponding plaintext API string. Then, she uses one of her other plugins to create repeating comments at the addresses of the import hash DWORDs. Instead, I wanted IDA to show me the import information the same way it would in a normal binary -- i.e., I wanted IDA to set the proper type signature on each import. I figured this might be difficult, but after a few hours reverse engineering the virtual functions for the pe_import_visitor_t class (partially documented in %IDASDK%\ldr\pe\common.hpp), it turns out that all you have to do to trigger this functionality is simply to set the name of the DWORD to something from a loaded type library.

Here's a screenshot showing IDA successfully applying the type information to the APIs:

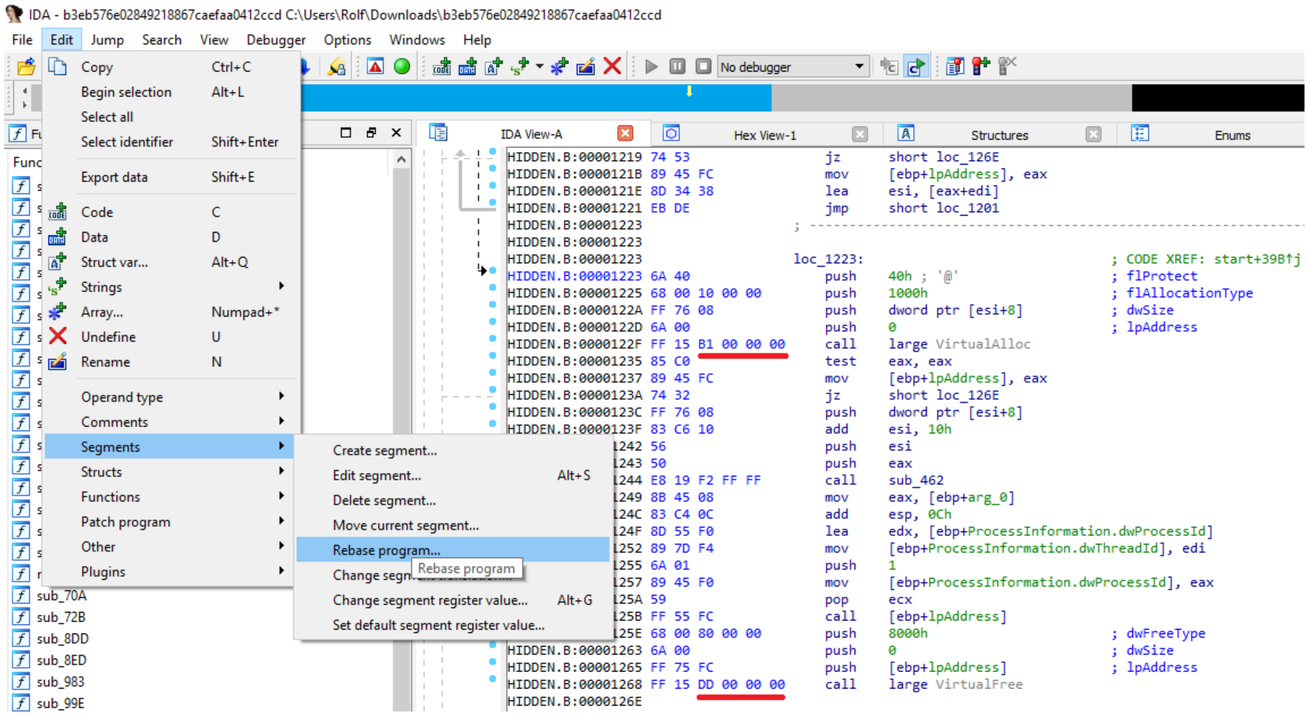


```
HIDDEN.B:00000085 ; FARPROC __stdcall GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
HIDDEN.B:00000085 GetProcAddress dd 0CF31BB1Fh          ; DATA XREF: sub_8ED+87↓r
HIDDEN.B:00000085                                       ; start+25A↓r
HIDDEN.B:00000089 ; void __stdcall Sleep(DWORD dwMilliseconds)
HIDDEN.B:00000089 Sleep dd 0E19E5FEh                    ; DATA XREF: sub_99E+27↓r
HIDDEN.B:0000008D ; DWORD __stdcall GetLastError()
HIDDEN.B:0000008D GetLastError dd 2082EAE3h             ; DATA XREF: sub_ACD+1A↓r
HIDDEN.B:0000008D                                       ; sub_D0F+37↓r ...
HIDDEN.B:00000091 ; HMODULE __stdcall LoadLibraryW(LPCWSTR lpLibFileName)
HIDDEN.B:00000091 LoadLibraryW dd 5FBFF111h             ; DATA XREF: sub_ACD+F↓r
HIDDEN.B:00000095 ; BOOL __stdcall CloseHandle(HANDLE hObject)
HIDDEN.B:00000095 CloseHandle dd 3870CA07h              ; DATA XREF: sub_72B+1A4↓r
HIDDEN.B:00000095                                       ; start+446↓r ...
HIDDEN.B:00000099 ; DWORD __stdcall ResumeThread(HANDLE hThread)
HIDDEN.B:00000099 ResumeThread dd 74162A6Eh             ; DATA XREF: start+4BA↓r
```

Relocations

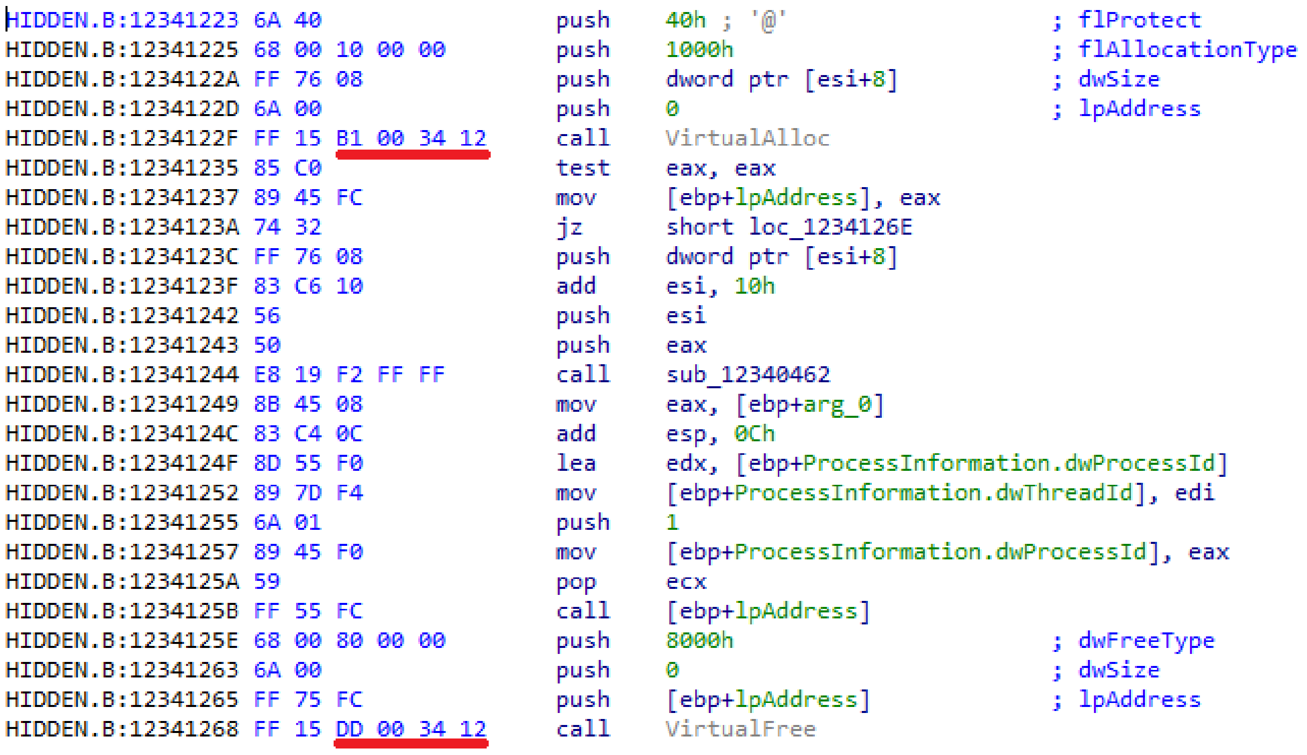
For the IMAGE_REL_BASED_HIGHLOW relocations common in PE files, each can ultimately be processed via straightforward translation of the relocation information into IDA's fixup_data_t data structures, and then passing them to the set_fixup API. The SDK examples did not give a straightforward idea of what I needed to do to handle PE IMAGE_REL_BASED_HIGHLOW relocations properly, so I reverse engineered pe.dll to figure out exactly what needed to happen with the relocations. (Fortunately, reverse engineering IDA is trivial due to the availability of its SDK.) If you wish, you can see the results in the do_reloc function. Don't ask me to explain why it works; however, it does work.

Here's a before and after comparison of rebasing the database from base address 0x0 to base address 0x12340000. Note particularly that the red underlined bytes change. Before:



```
HIDDEN.B:00001219 74 53                jz      short loc_126E
HIDDEN.B:0000121B 89 45 FC             mov     [ebp+lpAddress], eax
HIDDEN.B:0000121E 8D 34 38             lea     esi, [eax+edi]
HIDDEN.B:00001221 EB DE                jmp     short loc_1201
HIDDEN.B:00001223                  ; ---------------------------------------------------------------
HIDDEN.B:00001223
HIDDEN.B:00001223                  loc_1223:                               ; CODE XREF: start+39B↑j
HIDDEN.B:00001223 6A 40                push    40h ; '@'                   ; flProtect
HIDDEN.B:00001225 68 00 10 00 00       push    1000h                       ; flAllocationType
HIDDEN.B:0000122A FF 76 08             push    dword ptr [esi+8]           ; dwSize
HIDDEN.B:0000122D 6A 00                push    0                           ; lpAddress
HIDDEN.B:0000122F FF 15 B1 00 00 00    call    large VirtualAlloc
HIDDEN.B:00001235 85 C0                test    eax, eax
HIDDEN.B:00001237 89 45 FC             mov     [ebp+lpAddress], eax
HIDDEN.B:0000123A 74 32                jz      short loc_126E
HIDDEN.B:0000123C FF 76 08             push    dword ptr [esi+8]
HIDDEN.B:0000123F 83 C6 10             add     esi, 10h
         1242 56                       push    esi
         1243 50                       push    eax
         1244 E8 19 F2 FF FF           call    sub_462
         1249 8B 45 08                 mov     eax, [ebp+arg_0]
         124C 83 C4 0C                 add     esp, 0Ch
         124F 8D 55 F0                 lea     edx, [ebp+ProcessInformation.dwProcessId]
         1252 89 7D F4                 mov     [ebp+ProcessInformation.dwThreadId], edi
         1255 6A 01                    push    1
         1257 89 45 F0                 mov     [ebp+ProcessInformation.dwProcessId], eax
         125A 59                       pop     ecx
         125B FF 55 FC                 call    [ebp+lpAddress]
         125E 68 00 80 00 00           push    8000h                       ; dwFreeType
HIDDEN.B:00001263 6A 00                push    0                           ; dwSize
HIDDEN.B:00001265 FF 75 FC             push    [ebp+lpAddress]             ; lpAddress
HIDDEN.B:00001268 FF 15 DD 00 00 00    call    large VirtualFree
HIDDEN.B:0000126E
```

After:



```
HIDDEN.B:12341223 6A 40                push    40h ; '@'                   ; flProtect
HIDDEN.B:12341225 68 00 10 00 00       push    1000h                       ; flAllocationType
HIDDEN.B:1234122A FF 76 08             push    dword ptr [esi+8]           ; dwSize
HIDDEN.B:1234122D 6A 00                push    0                           ; lpAddress
HIDDEN.B:1234122F FF 15 B1 00 34 12    call    VirtualAlloc
HIDDEN.B:12341235 85 C0                test    eax, eax
HIDDEN.B:12341237 89 45 FC             mov     [ebp+lpAddress], eax
HIDDEN.B:1234123A 74 32                jz      short loc_1234126E
HIDDEN.B:1234123C FF 76 08             push    dword ptr [esi+8]
HIDDEN.B:1234123F 83 C6 10             add     esi, 10h
HIDDEN.B:12341242 56                   push    esi
HIDDEN.B:12341243 50                   push    eax
HIDDEN.B:12341244 E8 19 F2 FF FF       call    sub_12340462
HIDDEN.B:12341249 8B 45 08             mov     eax, [ebp+arg_0]
HIDDEN.B:1234124C 83 C4 0C             add     esp, 0Ch
HIDDEN.B:1234124F 8D 55 F0             lea     edx, [ebp+ProcessInformation.dwProcessId]
HIDDEN.B:12341252 89 7D F4             mov     [ebp+ProcessInformation.dwThreadId], edi
HIDDEN.B:12341255 6A 01                push    1
HIDDEN.B:12341257 89 45 F0             mov     [ebp+ProcessInformation.dwProcessId], eax
HIDDEN.B:1234125A 59                   pop     ecx
HIDDEN.B:1234125B FF 55 FC             call    [ebp+lpAddress]
HIDDEN.B:1234125E 68 00 80 00 00       push    8000h                       ; dwFreeType
HIDDEN.B:12341263 6A 00                push    0                           ; dwSize
HIDDEN.B:12341265 FF 75 FC             push    [ebp+lpAddress]             ; lpAddress
HIDDEN.B:12341268 FF 15 DD 00 34 12    call    VirtualFree
```