 © Invoke RE 2024

Automating Malware Deobfuscation with Binary Ninja

Recon 2024

© Invoke RE 2024

Introduction 3

Setup 3

Plugin Install 4

HLIL and Scripting with Binary Ninja 5

Qakbot Unpacking Stub 11

Import Hash Resolution 14

Hash Resolution Exercise 17

XOR Key Identification and Extraction 18

Resource Extraction and Decryption 18

Resource Identifier Extraction Exercise 19

Resource Extraction using PEFile 19

Resource Decryption 19

Carving Portable Executables 20

Testing Against Another Sample 20

Generically Identifying XOR Key and Resource ID 21

Conclusion 21

 © Invoke RE 2024

Introduction

Binary Ninja is a powerhouse reverse engineering suite that provides a plethora of functionality that is useful when reverse engineering malware. It has a robust Python API for interacting with abstractions (semantic representations) generated by their multiple levels of Binary Ninja Intermediate Languages (BNILs). These abstractions result in large simplifications of disassembled instructions into intrinsic functions and high level languages that can be accessed directly and easily, which we will be leveraging throughout this workshop.

This workshop will use Binary Ninja to acquire information needed to deobfuscate and extract a Qakbot sample from its packed form.

Setup

For the Binary Ninja components of this workshop, you will need a personal, commercial or enterprise version of Binary Ninja. This will give you access to the Python API that we will be using to extract information from the Binary Ninja database.

In addition to Binary Ninja, we will be using two Python modules to extract a resource from the packed binary (https://github.com/erocarrera/pefile) and carve embedded Portable Executables (https://github.com/binref/refinery).

To add these modules in Binary Ninja, perform the following steps:

- Press `CMD/CTRL+P` to open the command palette and type in "`Install python3 module`", which will highlight this command within the command palette window, as shown in Figure 1.

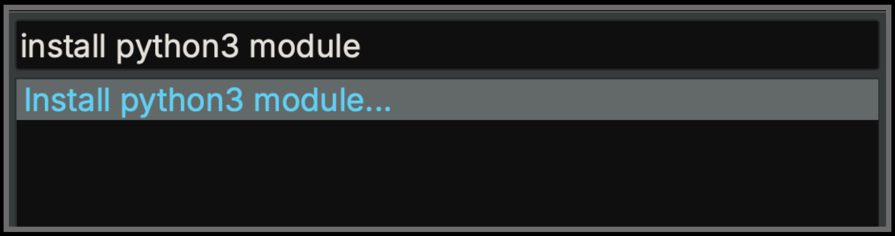

Figure 1. Install Python 3 Module Palette Option

- Press `Enter` to bring up the `Install python3 modules` window, as shown in Figure 2.

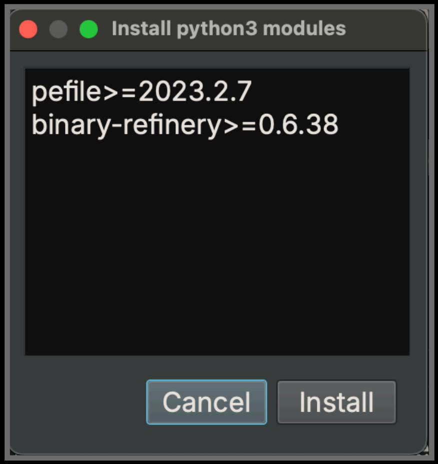

Figure 2. Modules Window

 © Invoke RE 2024

- Enter the following modules and press `install`:

```
pefile>=2023.2.7
binary-refinery>=0.6.38
```

This will install these dependencies in your Binary Ninja Python directory.

Plugin Install

We will be using a plugin called `Snippets` to visualize and execute the automation scripts that we will be writing. To install this plugin, navigate to the plugin manager by clicking on the `Plugins->Manage Plugins` menu item. This will open a new `Manage Plugins` tab, as shown in Figure 3.

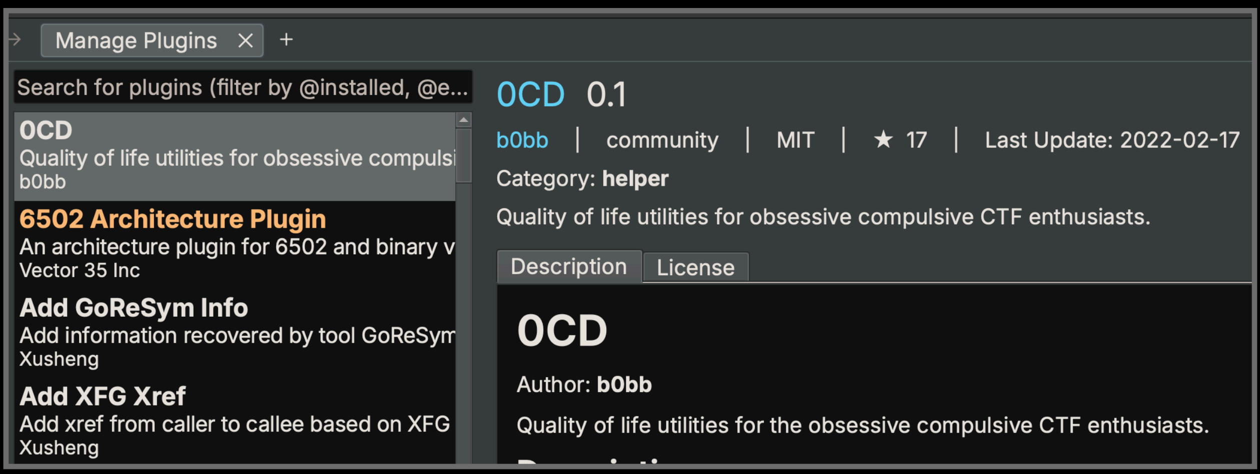

Figure 3. Plugin Manager

In the search box type in `Snippet UI Plugin`. Once displayed, right click the `Snippet UI Entry` and click on `Install Plugin`, as shown in Figure 4.

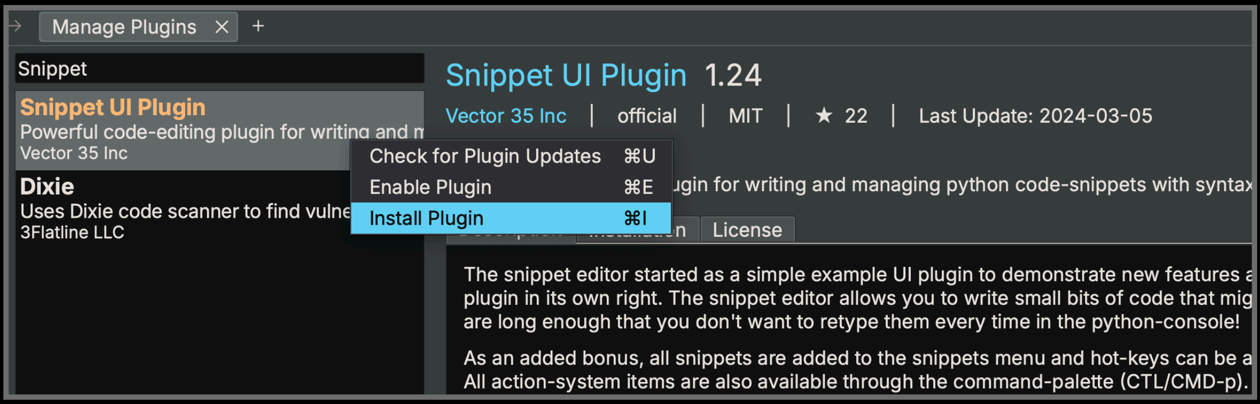

Figure 4. Snippet UI Plugin Installation

© Invoke RE 2024

In addition to the Snippet plugin, install the HashDB plugin written by Cindy Xiao.

We will be dealing with real malware samples. We will not be executing these, but if your system has antivirus it may pick up the packed or unpacked samples in these exercises. Please proceed with caution if you are using your host system.

Now that all required dependencies and plugins are installed, clone the workshop repository from GitHub using `git clone https://github.com/Invoke-RE/workshops`. This repository contains the baseline automation scripts and samples that we will be using throughout this workshop under `recon2024`. Unzip the `samples.zip` with the password "infected" and open `780be7a70ce3567ef268f6c768fc5a3d2510310c603bf481ebffd65e4fe95ff3` in Binary Ninja using `File->Open...` and selecting it from the file explorer dialogue.

HLIL and Scripting with Binary Ninja

Once the sample has been loaded and processed by Binary Ninja, the user interface will navigate to the `_start` function (`AddressOfEntryPoint` from the PE header) and display the High Level Intermediate Language (HLIL) representation of this function (Figure 5).

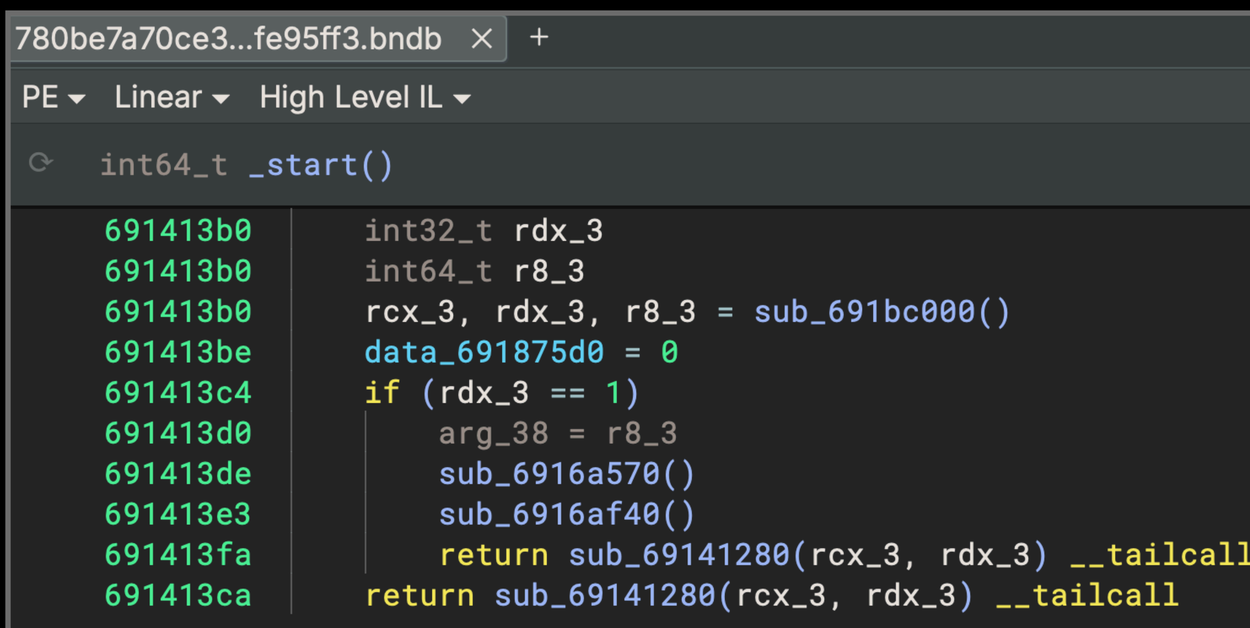

Figure 5. _start Function in HLIL Representation

We will be leveraging HLIL throughout this workshop to acquire a decompiled representation of instructions at a level similar to IDA Pro's Hex Rays and Ghidra.

To view the disassembled equivalent of this HLIL code, select the Split View icon in the top right-hand corner (Figure 6).

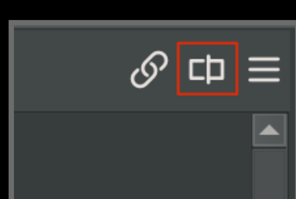

Figure 6. Split View Icon

© Invoke RE 2024

This will split the view into two vertical panes, as shown in Figure 7.

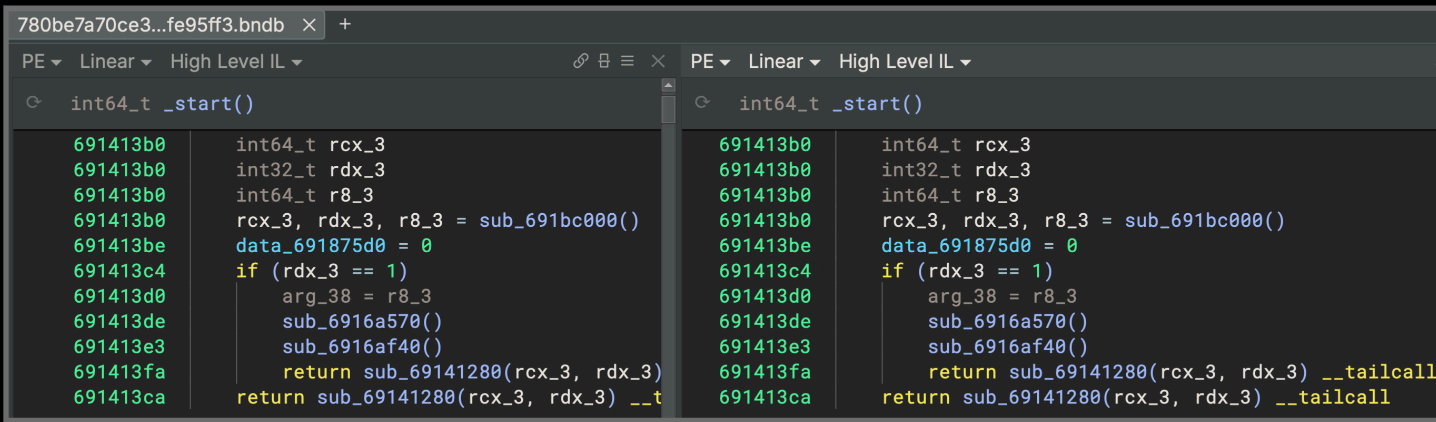

Figure 7. HLIL Split View

Change the instruction representation using the dropdown above the left pane (currently set to `High Level IL`) to `Disassembly`, as shown in Figure 8.

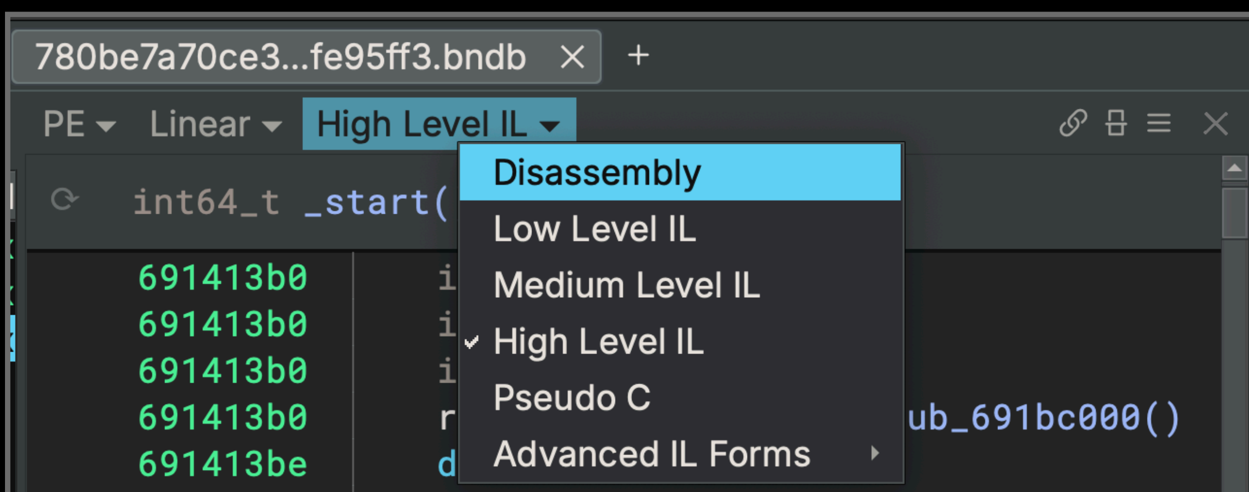

Figure 8. Select Disassembly View Dropdown

This displays the instructions in their disassembled form within this pane, as shown in Figure 9.

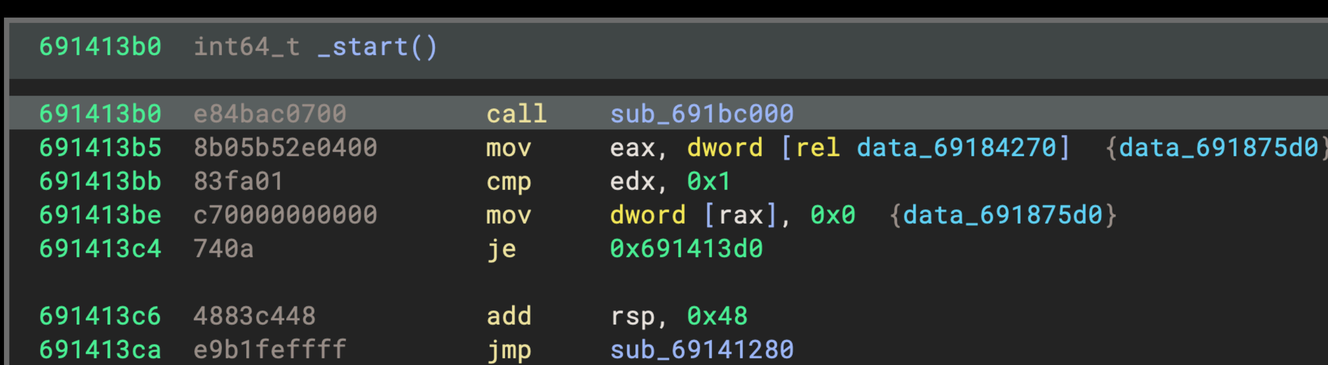

Figure 9. Disassembled _start Function

© Invoke RE 2024

The HLIL view attempts to provide a representation that can be semantically understood in the same way as a programming language, rather than solely relying on disassembled instructions. Having each representation side-by-side has multiple benefits. The first is that the disassembled instructions can be referenced inline with the HLIL in order to understand the recovered instructions more thoroughly. The second is that there are instances where the HLIL representation is inaccurate, and therefore the only guaranteed method of understanding the functionality correctly is to read the disassembled instructions.

Now that we have the HLIL and disassembly for the sample, let's take a look at interacting with the database using the Binary Ninja Python API. Binary Ninja provides a Read-Eval-Print Loop (REPL) Python console that provides code completion and a number of other useful functionality. Display this console using the Console button in the bottom left of the screen (Figure 10).

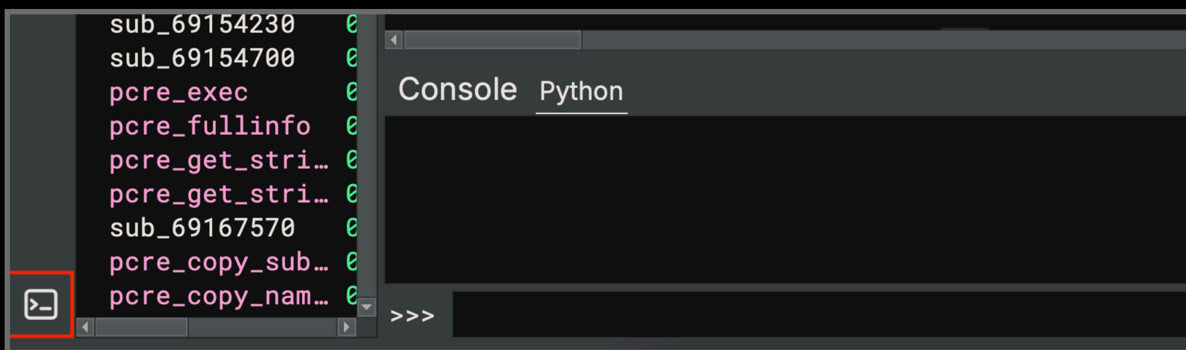

Figure 10. Python Console Button

The `STDOUT` and `STDERR` outputs from the Snippet plugin will be written to the Log view. Open the Log view by clicking on the Log button in the bottom right of the screen (Figure 11).

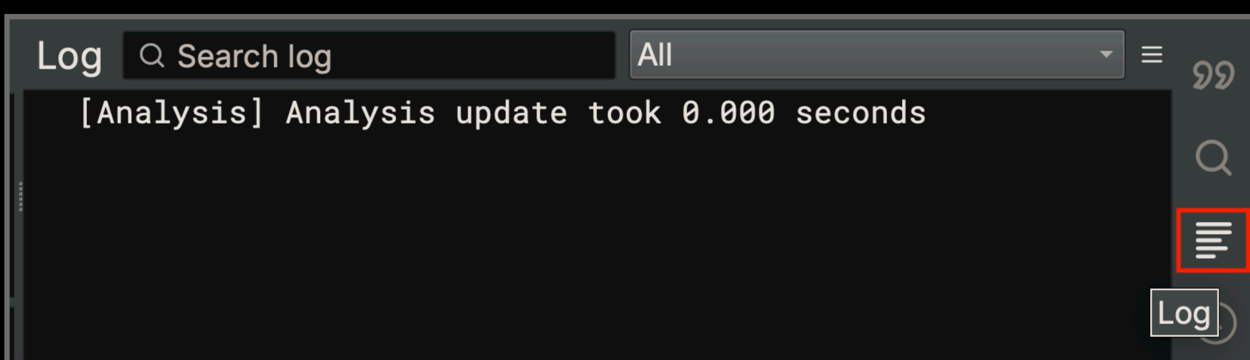

Figure 11. Log View Button

We can interact directly with the database using the BinaryView (or `bv`) through the Python console. For example, we can acquire the list of functions that have been discovered by Binary Ninja using `bv.functions`. This returns a generator, so we can acquire a list of these functions using `list(bv.functions)`, as shown in Figure 12.

© Invoke RE 2024

Console Python

```
>>> list(bv.functions)
[<func: x86_64@0x69141000>, <func: x86_64@0x69141050>,
<func: x86_64@0x69141280>, <func: x86_64@0x691413b0>,
<func: x86_64@0x69141400>, <func: x86_64@0x69141420>,
<func: x86_64@0x69141430>, <func: x86_64@0x69141510>,
<func: x86_64@0x691418a0>, <func: x86_64@0x69141910>,
<func: x86_64@0x69141ad0>, <func: x86_64@0x69141ca0>,
<func: x86_64@0x69141f10>, <func: x86_64@0x69141fb0>,
```

Figure 12. Function List from BinaryView

We can acquire all HLIL instructions from the database using `bv.hlil_instructions`. This also returns a generator, so we can acquire the first HLIL instruction, for example, using `list(bv.hlil_instructions)[0]`, as shown in Figure 13.

```
>>> list(bv.hlil_instructions)[0]
<HighLevelILVarInit: int64_t* rax = malloc(0x100)>
```

Figure 13. Get First HLIL Instruction in Database

In this example, the HLIL instruction is of type `HighLevelILVarInit`, because it is a variable that's being initialized by the memory allocation being performed. We can traverse these instructions by accessing its operands, as shown in Figure 14.

```
>>> a = list(bv.hlil_instructions)[0]
>>> a.operands
[<var int64_t* rax>, <HighLevelILCall: malloc(0x100)>]
```

Figure 14. Get HLIL Instruction Operands

Here we can see this instruction is made up of a variable and a `HighLevelILCall` to `malloc`. If we wanted to access the size of the memory allocation, for example, we could access the malloc instruction at its index and access this instruction's operands, as shown in Figure 15.

 © Invoke RE 2024

```
>>> a.operands[1].operands[1]
[<HighLevelILConst: 0x100>]
>>> a.operands[1].operands[1][0]
<HighLevelILConst: 0x100>
>>> a.operands[1].operands[1][0].value
<const 0x100>
>>> a.operands[1].operands[1][0].value.value
256
```

Figure 15. Get Malloc Allocation Size from HLIL Representation

Traversing the HLIL in this manner can be cumbersome. We can use a built-in helper function called traverse to recursively walk the abstract syntax tree (AST) of this instruction to look for a constant and return its value once found. This is done by providing a callback function that's called on each sub-instruction within the HLIL instruction. The return value of each callback is returned within a generator from the `traverse` call. An example of this is shown in Figure 16.

```
>>> def is_const(inst: HighLevelILInstruction):
...     if isinstance(inst, HighLevelILConst):
...         return inst.value.value
...
>>> list(a.traverse(is_call))
[256]
```

Figure 16. Traverse HLIL Recursively using Helper Function

The above example has the prerequisite of knowing that the type of value we're seeking from the instruction is a `HighLevelILConstant`. Since typing function definitions and other complex code into a REPL isn't ideal, this is where the Snippet editor plugin comes in. Open the Snippet editor by going to `Plugins->Snippets->Snippet Editor…` and click on the `New Snippet` button, as shown in Figure 17.

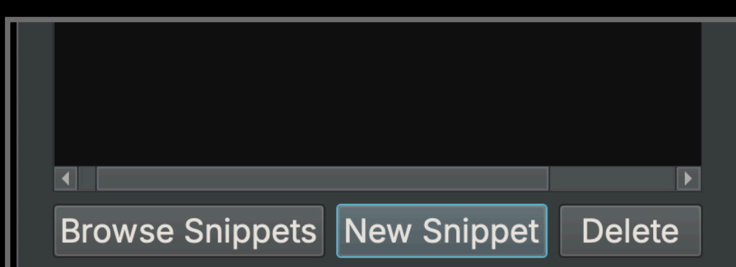

Figure 17. New Snippet Button

© Invoke RE 2024

This will open the Snippet `Name` window, as shown in Figure 18.

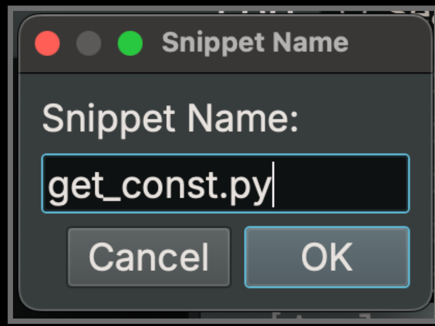

Figure 18. Snippet Name

Enter the name `get_const.py` and click `OK`. A new Snippet will be opened and we can now write our script here. Copy the code from `get_const.py` into this snippet, as shown in Figure 19.

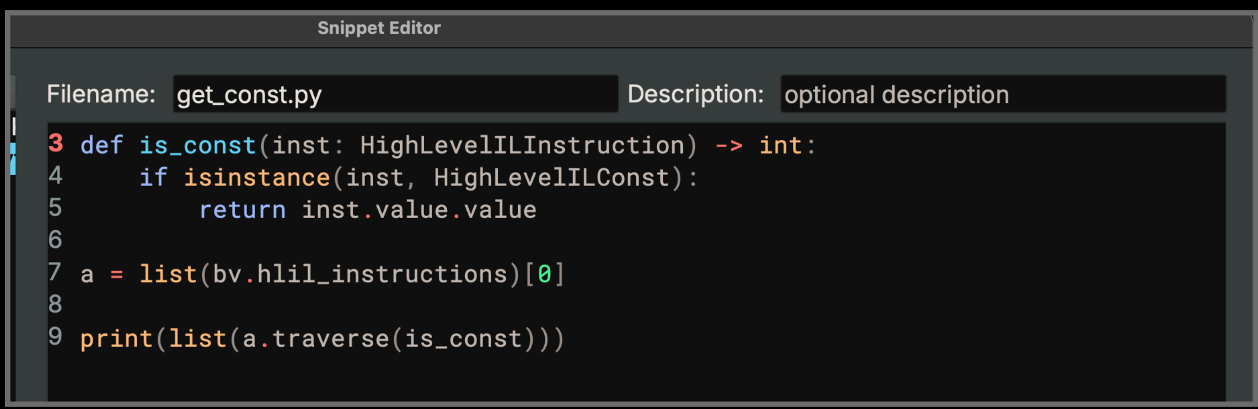

Figure 19. get_const Snippet Example

Ensure your High Level IL pane is selected and click on the `Run` button in the Snippet Editor, and you will see the same result is displayed in the Log view, as shown in Figure 20.

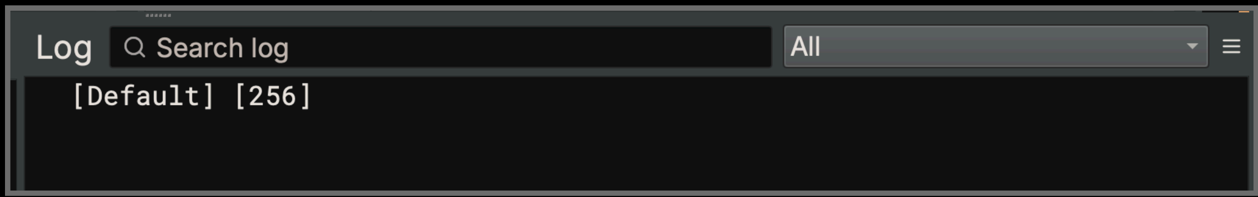

Figure 20. Result from Running Snippet

 © Invoke RE 2024

Qakbot Unpacking Stub

Now let's look at the unpacking code in this Qakbot sample and extract information from it that we can use to automatically unpack it. Packers typically have an outer program (commonly referred to as a stub) that deobfuscates a final malware payload that is executed in memory.

The sample we are analyzing in Binary Ninja (`780be7a70ce3567ef268f6c768fc5a3d2510310c603bf481ebffd65e4fe95ff3`) is the stub that decrypts position independent shellcode that decrypts, loads and executes a final Qakbot payload in memory. The stub extracts the encrypted shellcode ciphertext from an embedded resource and decrypts it using a basic XOR cipher. In this workshop we will be writing code to automatically extract the decryption key and the resource from the stub, decrypt the shellcode and carve the final Qakbot payload from it.

From the `_start` function you will see a call to `sub_691bc000` (Figure 21).

```
691413b0  int64_t _start()

691413b0      int64_t rcx_3
691413b0      int32_t rdx_3
691413b0      int64_t r8_3
691413b0      rcx_3, rdx_3, r8_3 = sub_691bc000()
691413be      data_691875d0 = 0
```

Figure 21. Call to sub_691bc000

Navigate to this function by double-clicking on `sub_691bc000` for further analysis.

Within the disassembly pane, you can see a large amount of `mov` operations, as shown in Figure 22.

```
int32_t sub_691bc000()

691bc02b  c644247040    mov     byte [rsp+0x70 {var_148}], 0x40
691bc030  c644247141    mov     byte [rsp+0x71 {var_147}], 0x41
691bc035  c64424726c    mov     byte [rsp+0x72 {var_146}], 0x6c
691bc03a  c64424737a    mov     byte [rsp+0x73 {var_145}], 0x7a
691bc03f  c644247473    mov     byte [rsp+0x74 {var_144}], 0x73
691bc044  c644247551    mov     byte [rsp+0x75 {var_143}], 0x51
691bc049  c644247631    mov     byte [rsp+0x76 {var_142}], 0x31
691bc04e  c644247744    mov     byte [rsp+0x77 {var_141}], 0x44
691bc053  c644247853    mov     byte [rsp+0x78 {var_140}], 0x53
691bc058  c644247953    mov     byte [rsp+0x79 {var_13f}], 0x53
691bc05d  c644247a3e    mov     byte [rsp+0x7a {var_13e}], 0x3e
691bc062  c644247b49    mov     byte [rsp+0x7b {var_13d}], 0x49
691bc067  c644247c39    mov     byte [rsp+0x7c {var_13c}], 0x39
```

Figure 22. Large Amount of Mov Operations

 © Invoke RE 2024

The bytes are moved into local variables that are relative to the stack pointer (RSP) and are within the ASCII range. Select any of these hex bytes and press the `r` key to turn them back into their ASCII representation. An example is shown in Figure 23.

```
int32_t sub_691bc000()

691bc000  4881ecb8010000     sub    rsp, 0x1b8
691bc007  48c784245801000…   mov    qword [rsp+0x158 {var_60}], 0x0
691bc013  48c784249001000…   mov    qword [rsp+0x190 {var_28}], 0x0
691bc01f  48c784245001000…   mov    qword [rsp+0x150 {var_68}], 0x0
691bc02b  c644247040         mov    byte [rsp+0x70 {var_148}], '@'
691bc030  c644247141         mov    byte [rsp+0x71 {var_147}], 0x41
```

Figure 23. Character Change to ASCII Representation Example

This is a technique known as “stack strings” where a string is constructed on the stack dynamically at runtime. This is an obfuscation technique that prevents recovery of strings using simple techniques, such as running the `strings` utility.

Even though this technique is used, the Binary Ninja HLIL simplifies this into a single `__builtin_strncpy` operation (Figure 24).

```
int32_t sub_691bc000()

    int64_t s_1
    __builtin_memset(s: &s_1, c: 0, n: 0x14)
    char var_148
    __builtin_strncpy(dest: &var_148, src: "@AlzsQ1DSS>I9XX7kB7M1MT3?CH8B1ggtV_!RTX0zJSbzmUYpW5H2n@o$"
```

Figure 24. __builtin_strncpy Operation

The `__builtin_` prefix specifies that this is an “intrinsic” which means the instruction is inferred from the `mov` operations being performed. Select the destination variable (`var_148`) in order to display the cross-references in the `Cross References` pane (Figure 25).

```
Cross References
▸ Filter (6)
▾ Data References                                              {1
  |→ 691be1d8 char* __builtin_strncpy(char* dest, char const* src
▾ Variable References                                          {5
 ▾char var_148                                                 {5
   |← 691bc02b char var_148
   |← 691bc02b __builtin_strncpy(&var_148, "@AlzsQ1DSS>I9XX7kB7M1M
   |← 691bc72b rax_25[sx.q(i_1 - var_78_1 * var_2c_1 + var_104_1 +
   |→ 691bc02b char var_148
   |→ 691bc02b __builtin_strncpy(&var_148, "@AlzsQ1DSS>I9XX7kB7M1M
```

Figure 25. var_148 Cross References

The third entry (highlighted in red) shows a number of operations being performed with this string. Double-click on this entry to navigate to this location (`0x691bc72b`). A number of operations are performed within a for loop with the result being written to `rax_25` (Figure 26).

```
for (int32_t i_1 = 0; i_1 u< rax_23; i_1 += 1)
    rax_25[sx.q(i_1 - i * var_2c_1 + var_104_1 + i - i
result = rax_25()
```

Figure 26. rax_25 Operations within For Loop and Shellcode Execution

This includes an XOR operation with a byte from the var_148 string (Figure 27).

```
rax_24[sx.q(i_1 + i * 2)] ^ (&var_148)[modu.dp.q(0:(sx.q(i_1)), 0x3a)]
```

Figure 27. var_148 XOR Operation

A modulus operator is being applied to the current index, which causes the index to not exceed the length of `var_148`. Based on these operations, we can infer that the `var_148` string is an XOR key being used to decrypt a buffer within `rax_24` and the result is written to `rax_25` . We then see `rax_25` being executed as a function pointer (Figure 26), which results in the decrypted bytes being executed.

We can rename these variables by highlighting the variable name and pressing the `n` key. This will display the `Define name` dialogue where a variable name can be entered (Figure 27).

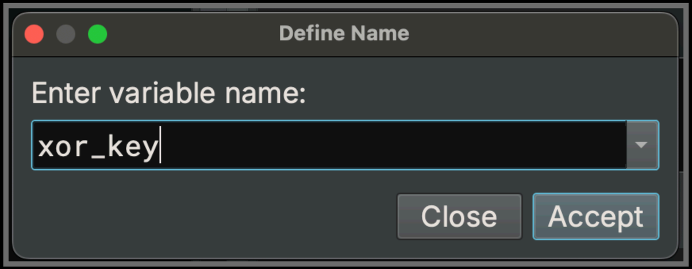

Figure 27. Define Variable Name

Rename `var_148` to `xor_key`, `rax_24` to `input_buffer` and `rax_25` to `shellcode`. By looking at the operations above the for loop, we can see the `input_buffer` and `shellcode` pointers are the result of two function calls that use unknown function pointers (Figure 28).

```
691bc5b1        int64_t rax_22 = rax_12(var_68, zx.q(var_100), 0xa)
691bc5ca        int32_t rax_23 = rax_15(var_68, rax_22)
691bc5e2        char* input_buffer = rax_18(var_68, rax_22)
691bc629        char* shellcode = rax_9(0, zx.q(rax_23), 0x3000, zx.q
```

Figure 28. Input Buffer and Shellcode Initialization

 © Invoke RE 2024

Import Hash Resolution

These function pointers are assigned the return values of calls to `sub_691bcd90` (Figure 29).

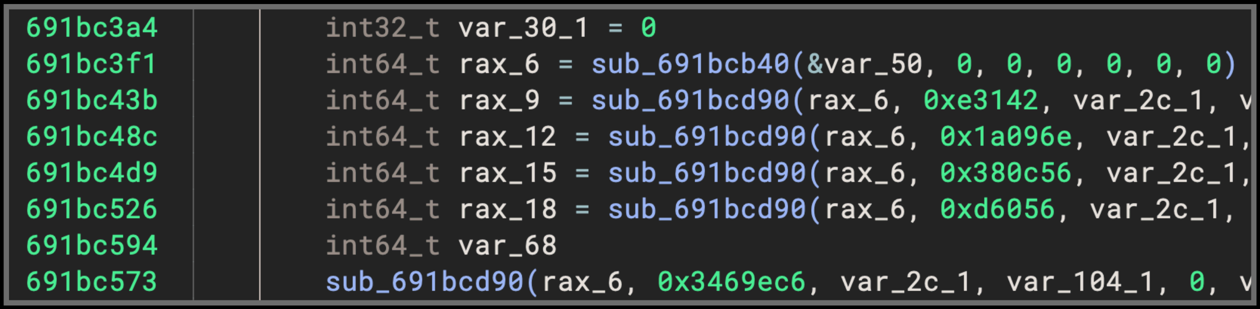

Figure 29. Function Pointer Initialization

The `sub_691bcd90` function performs dynamic import resolution, which is a technique often used by malware and shellcode to resolve Windows API functions at runtime. This involves parsing in-memory structures in order to resolve the addresses of particular functions that are required for the malware to execute.

This process requires resolution of the module (in this case kernel32) containing the function, and then the export table of the resolved DLL is walked in order to identify the address of the target function. Malware authors will often use hashing algorithms to iterate over each function that is parsed in memory, hash each function name and compare it against the hash of a desired function to resolve.

In this instance, the second parameter being passed to `sub_691bcd90` is the hash being resolved, whose resolved function pointer is then assigned to a local variable that is later called where needed. We can automatically identify the hashing algorithm using a plugin called hashdb, which queries an online database that contains precomputed hashes that map to common strings, such as Windows API function names.

The second parameter passed to `sub_691bcd90` is the hash that is being resolved by this function. We can confirm this by right-clicking on the first hash (`0xe3142`) and going to `HashDB->Hunt`, as shown in Figure 30.

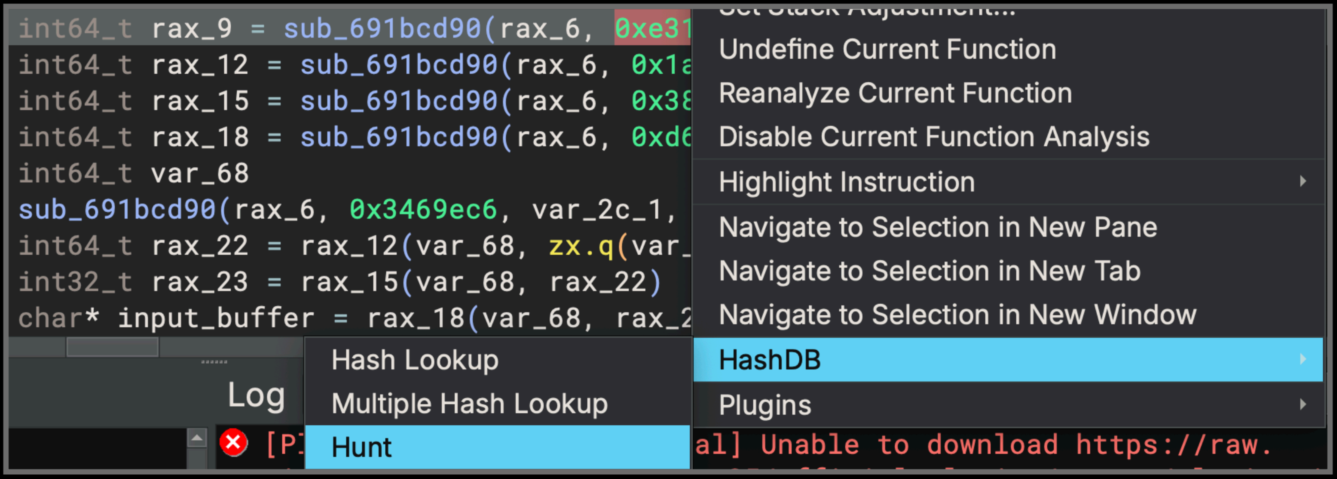

Figure 30. HashDB Hunt for Hash

 © Invoke RE 2024

This will query HashDB for this hash, and “hunt” for any hashing algorithms that have produced this hash. In this instance, a number of hashing algorithms are found, as shown in Figure 31.

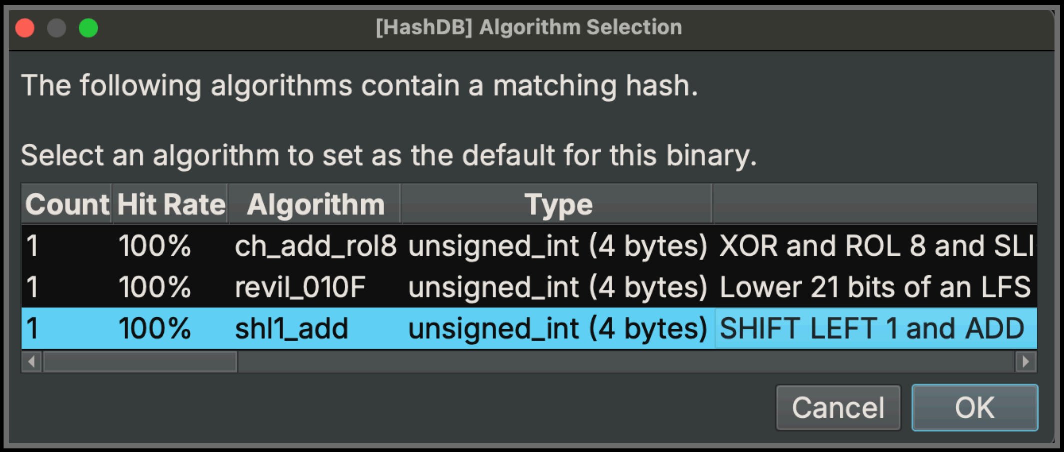

Figure 31. HashDB Hunting Results

This particular hash was produced by a number of different hashing algorithms that are stored in HashDB. Select the `shl1_add` algorithm, as shown in Figure 31, and click `OK`. This will set the hashing algorithm to `shl1_add` as shown in the Log window (Figure 32).

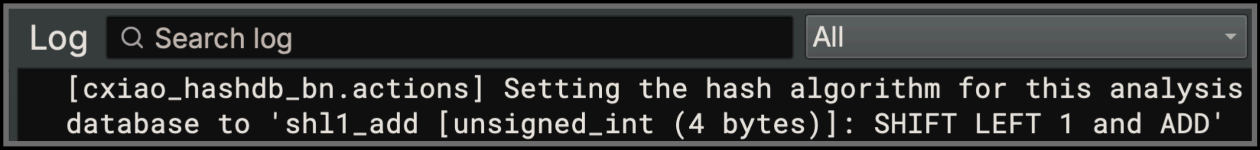

Figure 32. Hashing Algorithm Set

Now that the hashing algorithm has been identified and set, we can query HashDB for each hash by right-clicking on `0xe3142` and selecting `HashDB->Hash Lookup`. This will display the `String Selection` window, as shown in Figure 33.

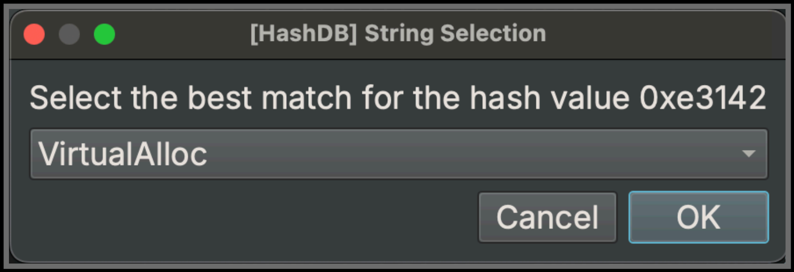

Figure 33. String Selection from Hash Value

This will provide a dropdown of strings that correspond to the provided hash. It is common to have hashing collisions (the same hash for two different strings) from these simple algorithms, so this may involve some trial and error. However, in this instance, `VirtualAlloc` is correct. Click `OK` to add the selected string to the Binary Ninja database as an enum.

© Invoke RE 2024

In addition to the `String Selection` window, you will be presented with the `Bulk Import` window that will allow you to add all export function string hashes related to a particular module (DLL) as an enumeration to the database. This will allow related function names to be applied to other hashes that make use of the same hashing algorithm within the database. The `VirtualAlloc` export is contained within a number of different modules. Select `kernel32` from the dropdown, as shown in Figure 34, and click `OK`.

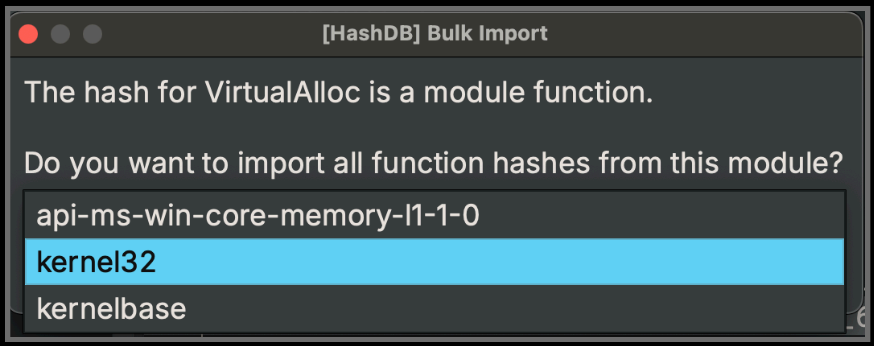

Figure 34. Bulk Import Kernel32 Exports

Now that all hashes for `kernel32` have been imported, we can apply them to each hash by pressing the `m` key. For example, select `0xe3142` in the HLIL view and press the `m` key to display the `Select Enum` window for the `VirtualAlloc` enum member (Figure 35).

Figure 35. Select VirutalAlloc Enum Member

Press the `Select Enum` button to apply this enum member to the selected hash. This will then be displayed in the HLIL view (Figure 36).

```
int64_t rax_9 = sub_691bcd90(rax_6, VirtualAlloc, var_2c_1, var_104_1,
int64_t rax_12 = sub_691bcd90(rax_6, 0x1a096e, var_2c_1, var_104_1, 0,
int64_t rax_15 = sub_691bcd90(rax_6, 0x380c56, var_2c_1, var_104_1, 0,
int64_t rax_18 = sub_691bcd90(rax_6, 0xd6056, var_2c_1, var_104_1, 0,
```

Figure 36. Applied VirtualAlloc Enum Member

 © Invoke RE 2024

Hash Resolution Exercise

Rename `sub_691bcd90` to `mw_resolve_hash` using the `n` key and apply enums to the remaining hashes using the `m` key. The result should look something like Figure 37.

```
int64_t rax_9 = mw_resolve_hash(rax_6, VirtualAlloc, var_2c_1, var_104_1, 0, v
int64_t rax_12 = mw_resolve_hash(rax_6, FindResourceA, var_2c_1, var_104_1, 0,
int64_t rax_15 = mw_resolve_hash(rax_6, SizeofResource, var_2c_1, var_104_1, 0
int64_t rax_18 = mw_resolve_hash(rax_6, LoadResource, var_2c_1, var_104_1, 0,
int64_t var_68
mw_resolve_hash(rax_6, GetModuleHandleExW, var_2c_1, var_104_1, 0, var_30_1, i
```

Figure 37. Resolved Enumerations Applied and Function Renamed

Given that `mw_resolve_hash` returns function pointers, rename each variable using the `n` key to each respective function being resolved. The result should look something like Figure 38.

```
int64_t VirtualAlloc = mw_resolve_hash(rax_7, VirtualAlloc, var
int64_t FindResourceA = mw_resolve_hash(rax_7, FindResourceA, v
int64_t SizeofResource = mw_resolve_hash(rax_7, SizeofResource,
int64_t LoadResource = mw_resolve_hash(rax_7, LoadResource, var
int64_t var_68
mw_resolve_hash(rax_7, GetModuleHandleExW, var_2c_1, var_104_1,
int64_t rax_19 = FindResourceA(var_68, zx.q(var_100), 0xa)
int32_t rax_20 = SizeofResource(var_68, rax_19)
char* input_buffer = LoadResource(var_68, rax_19)
char* shellcode = VirtualAlloc(0, zx.q(rax_20), 0x3000, zx.q(0x
```

Figure 38. Assigned Variable Names to Pointers

As you can see, things are beginning to take shape now that we can see which Windows API functions are being called. A handle to a resource embedded within the binary is acquired using `FindResourceA`, the size of the resource is acquired using `SizeofResource` and the resource is loaded using `LoadResource` with the result being assigned to `input_buffer`. The second parameter of `FindResourceA` is `lpName` which is used to identify the embedded resource. This value is stored within `var_100`. Selecting `var_100` shows a cross-reference for this variable at the beginning of this function where it is initialized (Figure 39).

```
Variable References                    {6}
int16_t var_100                        {3}
  |←  691bc1cb int16_t var_100 = 0x3b4
```

Figure 39. var_100 Initialization Cross-Reference

 © Invoke RE 2024

Rename `var_100` to `rsrc_id` for readability. In order for us to extract the final Qakbot payload, we need to extract the `rsrc_id` to identify which resource we need to extract from the PE and extract the `xor_key` to decrypt the embedded resource. We will do this programatically using the Binary Ninja API.

XOR Key Identification and Extraction

We can acquire all HLIL instructions for our target function (`sub_691bc000`) using:

```
list(bv.get_function_at(0x691bc000).hlil.instructions)
```

We can then enumerate these instructions and look at the first call to `__builtin_strncpy` using the following (see `extract_qakbot.py`):

```
finstr = None
addr = 0x691bc000
func = bv.get_function_at(addr)

for instr in func.hlil.instructions:
    for token in instr.tokens:
        if '__builtin_strncpy' == token.text:
            finstr = instr
    if finstr:
        break

if finstr:
    # Access second operand of __builtin_strncpy
    key_param = finstr.params[1]
    #Access constant data from parameter
    key = bytes(key_param.constant_data.data)
```

Here we enumerate all text tokens that make up each HLIL instruction and look for the `__builtin_strncpy` operation. We can then access the second parameter of the `__builtin_strncpy` instruction and acquire the data using the following code:

```
if finstr:
    # Access second operand of __builtin_strncpy
    key_param = finstr.params[1]
    #Access constant data from parameter
    key = bytes(key_param.constant_data.data)
```

This will acquire the key data in `byte` format that we can use to decrypt the embedded resource.

Resource Extraction and Decryption

The resource identifier assignment comes after the call to `__builtin_strncpy`, as shown in Figure 40.

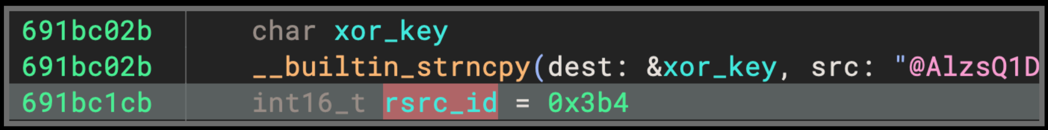

Figure 40. Resource Identifier Assignment

 © Invoke RE 2024

We can access this instruction by adding to the current HLIL instruction's index using the following code:

```
list(func.hlil.instructions)[finstr.instr_index+1]
```

Here we are accessing the `__builtin_strncpy`'s index and using that to acquire the next instruction in the list of all HLIL instructions within the function.

Resource Identifier Extraction Exercise

Now that we have the resource identifier instruction, use the techniques described in the *HLIL and Scripting with Binary Ninja* section to acquire the resource identifier constant value and add it to `extract_qakbot.py`.

Resource Extraction using PEFile

Once the resource identifier has been acquired, we can use it to extract the resource using a module called pefile. The pefile module allows parsing and accessing PE structures and attributes, including resources. We can access the resource using the following:

```
def extract_resource(fpath, rsrcid):
    rsrc_data = None
    pe = pefile.PE(fpath)
    pe_mapped = pe.get_memory_mapped_image()
    for rsrc in pe.DIRECTORY_ENTRY_RESOURCE.entries:
        for entry in rsrc.directory.entries:
            if entry.struct.Name == rsrcid:
                rsrc_offset = 
entry.directory.entries[0].data.struct.OffsetToData
                rsrc_size = 
entry.directory.entries[0].data.struct.Size
                rsrc_data = pe_mapped[rsrc_offset:rsrc_offset + 
rsrc_size]
    return rsrc_data
```

This function accesses `IMAGE_DIRECTORY_ENTRY_RESOURCE` data directory entries of the provided PE and compares each entry's name to the acquired resource identifier. The `pefile` module allows accessing PE data in its mapped format using the `get_memory_mapped_image` function. This is used to acquire the resource data with the identified resource offset and resource size.

Resource Decryption

We can now decrypt the extracted resource data by performing a rotating XOR decryption with the extracted key from the *XOR Key Identification and Extraction* section and the following code:

```
def xor(key: bytes, ct: bytes) -> bytes:
    r = bytes()
    for i, b in enumerate(ct):
        r += (b ^ key[i % len(key)]).to_bytes(1, 'little')
    return r
```

 © Invoke RE 2024

This results in plaintext shellcode that maps an embedded PE into memory and executes it. This first stage PE contains another PE that is our final Qakbot payload.

Carving Portable Executables

In order to carve all embedded PE files, we can use Binary Refinery, which is "a collection of Python scripts that implement transformations of binary data such as compression and encryption". Binary Refinery is often thought as a command-line version of CyberChef, which provides data transformations used by malware analysts to recover and deobfuscate information. We will be using Binary Refinery to carve embedded PE files from a given data blob using the carve_pe module using the following code:

```
def carve_pe(data: bytes) -> list:
    from refinery import carve_pe
    # This syntax is specific to Binary Refinery's
    # operator overloading and is valid Python.
    carved = data | carve_pe | []
    return carved
```

Here this function makes use of Binary Refinery's pipe format to pipe our resource data into the `cave_pe` module, which will look for, acquire the size of and extract a portable executable from the provided blob and return this within a list. As mentioned, since this blob contains two embedded PE files, we need to call this function twice using the following code:

```
first_pe = carve_pe(pt)[0]
second_pe = carve_pe(first_pe)[0]
```

The second PE is then written to disk using the following code:

```
fw = open(F"{path}_qakbot.bin", "wb")
fw.write(second_pe)
fw.close()
```

Save the opened Binary Ninja database by pressing `CMD/CTRL+s` (in order for our script to find the binary file on disk) and run `extract_qakbot.py` in the Snippets editor.

Testing Against Another Sample

Open the `12094a47a9659b1c2f7c5b36e21d2b0145c9e7b2e79845a437508efa96e5f305` sample in Binary Ninja and navigate to the `sub_180005556` function. This is the same type of unpacking function identified in the first sample that we analyzed (Figure 41).

```
int32_t sub_180005556()

    int64_t s_1
    __builtin_memset(s: &s_1, c: 0, n: 0x14)
    char var_128
    __builtin_strncpy(dest: &var_128, src: "F?fzfMN(JNfU3)sNT0TY61J!4Qo"
    int32_t var_100 = 0x2f2
```

Figure 41. Unpacking Stub Key Copy and Resource Identifier

 © Invoke RE 2024

Replace the `addr` variable value with this address (`0x180005556`) and run the script. Since the function contains a `__builtin_strncpy` call followed by the resource identifier variable assignment, the script automatically identifies these values, extracts them, decrypts the embedded resource and writes the final payload to disk.

Generically Identifying XOR Key and Resource ID

A generic version of the script is provided `extract_heuristic_generic.py`. Since we do not want to manually identify the function containing these values for every packed sample, we want to establish a heuristic for identifying these functions.

Since the `__builtin_strncpy` intrinsic is not called frequently, we can look for this within the database as a starting point. We can acquire the address of the this function using `bv.get_symbol_by_raw_name('__builtin_strncpy').address` which resolves this symbol to the virtual address within the database. We can then get the cross-references to this address using `bv.get_callers(addr)`.

We can then verify that we have the correct call location by acquiring all HLIL instructions for the function and ensuring the instructions following this call are an integer assignment and variable declaration:

```
rsrc_instr = list(c.function.hlil.instructions)[c.hlil.instr_index+1]
var_init_instr = list(c.function.hlil.instructions)
[c.hlil.instr_index+2]
if isinstance(rsrc_instr, HighLevelILVarInit) \
    and isinstance(rsrc_instr.operands[1], HighLevelILConst) \
    and isinstance(var_init_instr, HighLevelILVarDeclare):
    return c.function
```

Developing these heuristics typically requires identifying unique attributes that can be used to enumerate locations needed to extract information. Here we've simply used the two types of HLIL instructions that follow our target location, which is unique to this function.

Conclusion

This workshop has demonstrated common automation techniques that can be used by analysts to access information from binaries using Binary Ninja and deobfuscate them using plugins and Python modules. Although these types of tasks are common, malware samples often require custom automation to be written for them, but the general concepts can be applied across many malware families. Additionally, the automation can be applied to multiple malware samples that have been obfuscated using the same techniques in order to extract information from them automatically, which drastically saves time and manual analysis efforts.