Writing a PE Loader from Scratch in Unicorn

Wai Tuck Wong
@waituck

Today’s Topics

- Introduction to **Unicorn**
- Using Unicorn in CTFs
  - TWCTF 2016 » reverse_box
- Pushing it Further
  - PE Loading Basics
  - Adapting to API Obfuscation Techniques
- Conclusion

Introduction to **Unicorn**

"lightweight, multi-platform, multi-architecture CPU Emulator"

- Supports **a ton** of architectures
  - Arm, Arm64 (Armv8), M68K, Mips, Sparc, & X86 (include X86_64)
- **Raw CPU** emulator!
  - You have to write your own loaders
  - Imports, files, machine devices etc. not supported
- Powered by **QEMU**

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a"                               C edx
mu = Uc(UC_ARCH_X86, UC_

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
      len(X86_CODE32))
```

We need our shellcode to exist in some memory! Set arbitrary address as start of memory.

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
        len(X86_CODE32))
```

Actual shellcode.

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
      len(X86_CODE32))
```

Instantiate Unicorn emulator instance with architecture and mode

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
        len(X86_CODE32))
```

We need to map memory before we can write memory (duh). Any size that is page aligned is fine.

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
      len(X86_CODE32))
```

Write the shellcode to the address.

Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
        len(X86_CODE32))
```

Write registers
(they default to 0)

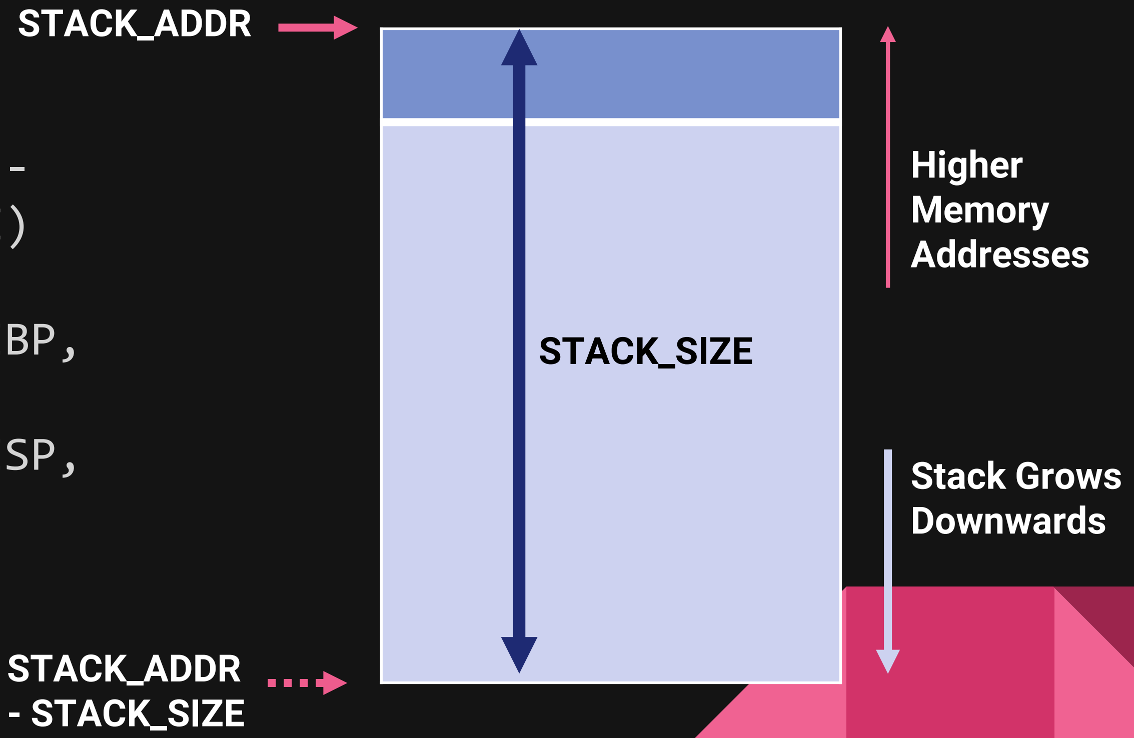
Introduction to Unicorn » Loading Shellcode

```
ADDRESS = 0x1000000
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu = Uc(UC_ARCH_X86, UC_MODE_32)

mu.mem_map(ADDRESS, 2 * 1024 * 1024)
mu.mem_write(ADDRESS, X86_CODE32)

mu.reg_write(UC_X86_REG_ECX, 0x1234)
mu.reg_write(UC_X86_REG_EDX, 0x7890)

mu.emu_start(ADDRESS, ADDRESS + \
       len(X86_CODE32))
```

Run the emulator, until PC reaches end address.

Introduction to Unicorn » Stack

```
mu.mem_map(STACK_ADDR -
STACK_SIZE, STACK_SIZE)

mu.reg_write(X86_REG_EBP,
STACK_ADDR)
mu.reg_write(X86_REG_ESP,
STACK_ADDR)
```

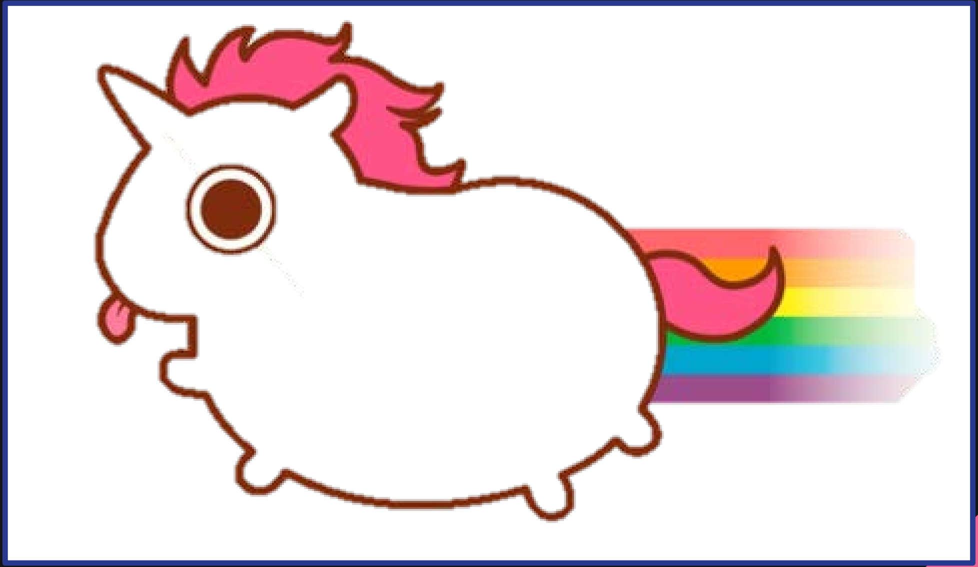


Introduction to Unicorn » Hooks!

```
mu.hook_add(UC_HOOK_MEM_READ_UNMAPPED |
UC_HOOK_MEM_WRITE_UNMAPPED | UC_HOOK_MEM_FETCH_UNMAPPED,
hook_mem_invalid)

mu.hook_add(UC_HOOK_CODE, stop_hook)

mu.hook_add(UC_HOOK_MEM_WRITE, mem_write_hook)

mu.hook_add(UC_HOOK_BLOCK, hook_block)
```

Using Unicorn in CTFs

- Instrument small pieces of code which are **independent**



Using Unicorn in CTFs » TWCTF: reverse_box

```
$ ./reverse_box ${FLAG}
95eeaf95ef94234999582f722f492f72b19a7aaf72e6e776b57aee722fe77ab5ad9aaeb156729676ae7a236d99b1df4a
```

Using Unicorn in CTFs » TWCTF: reverse_box

```
void main(int argc, char **argv)

{
    .........
    if (argc < 2) {
        printf("usage: %s flag\n",*argv);
        exit(1);
    }
    create_enc_buf(enc_buf);
    index = 0;
    while( true ) {
        len_flag = strlen(argv[1]);
        if (len_flag <= index) break;
        printf("%02x",(uint)enc_buf[(int)argv[1][index]]);
        index = index + 1;
    }
}
```

Substitution Cipher!

Using Unicorn in CTFs » TWCTF: reverse_box

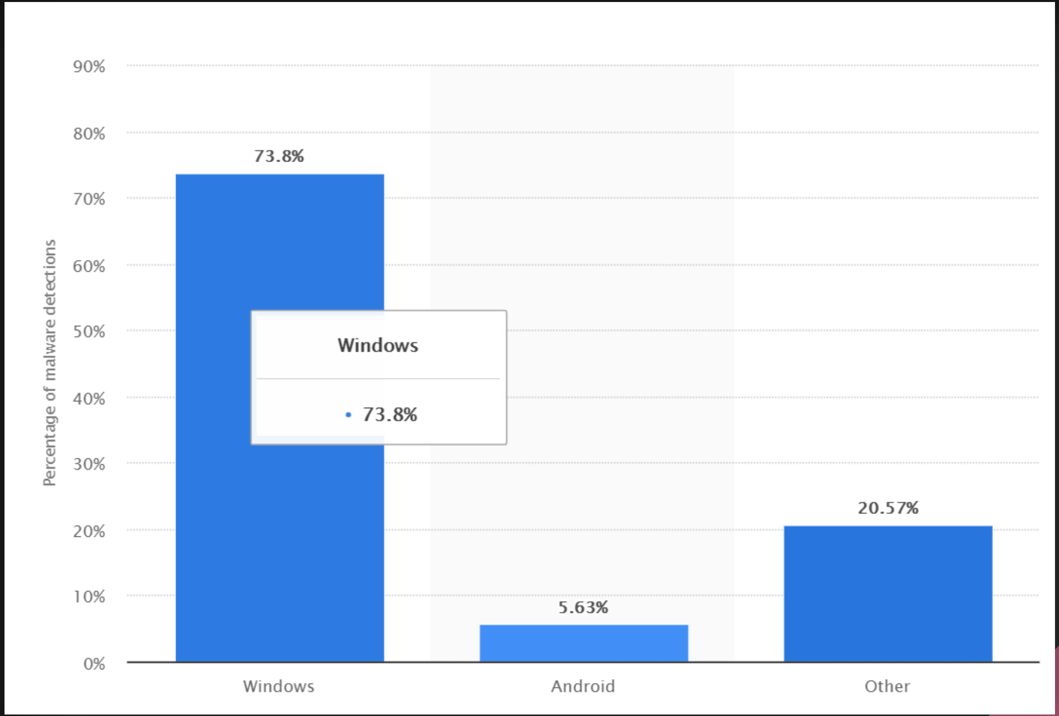
```
void create_enc_buf(byte *enc_buf)

{
    uint __seed;
    uint rand_int;
    byte index;

    __seed = time(NULL);
    srand(__seed);
    do {
        rand_int = rand();
    } while ((rand_int & 0xff) == 0);
   *enc_buf = (byte)(rand_int & 0xff);
    …
}
```

Only 256 combinations!
Modify and bruteforce!

Using Unicorn in CTFs » TWCTF: reverse_box

```
void main(int argc, char **argv)

{
    .........
    if (argc < 2) {
        printf("usage: %s flag\n",*argv);
        exit(1);
    }
    create_enc_buf(enc_buf);
    index = 0; // emulate until here
    while( true ) {
        len_flag = strlen(argv[1]);
        if (len_flag <= index) break;
        printf("%02x",(uint)enc_buf[(int)argv[1][index]]);
        index = index + 1;
    }
}
```

Pushing it Further » Real Life Applications



Statistica.com (2018) *Operating systems most affected by malware as of 1st quarter 2018*

Pushing it Further » Loading PE Files

- Not the first time this has been done!

- Tavis Ormandy’s LoadLibrary
  (*https://github.com/taviso/loadlibrary*)
- Angr
  (https://github.com/angr/angr/blob/master/angr/simos/windows.py)

Pushing it Further » The Challenge

- PE file is well documented

https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format

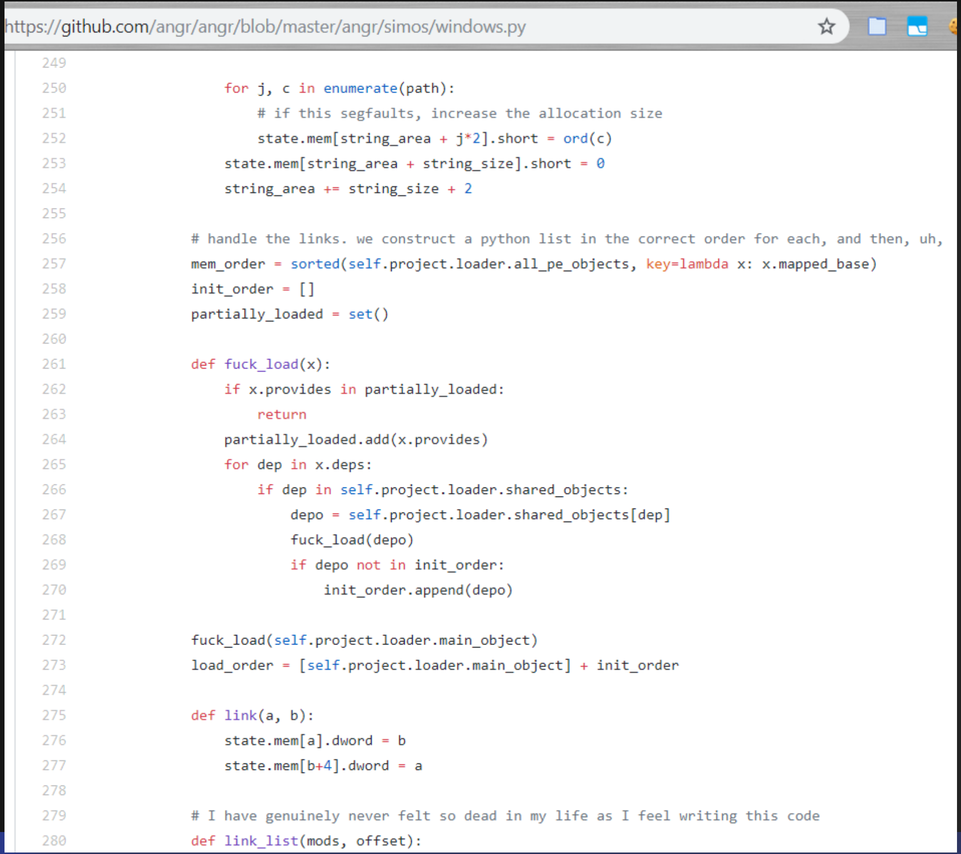
- (Mostly) undocumented loading process!

Exciting!

Pushing it Further » The Challenge

“For the entries *(in the directory table)* that exist for a given executable**, the system must perform operations to ensure that the memory will be in the expected state for the program to be able to function**. Each entry requires different handling, **not all of which I have figured out yet**. ” - http://www.cultdeadcow.com/tools/pewrap.html

Pushing it Further » Made Angr Angry

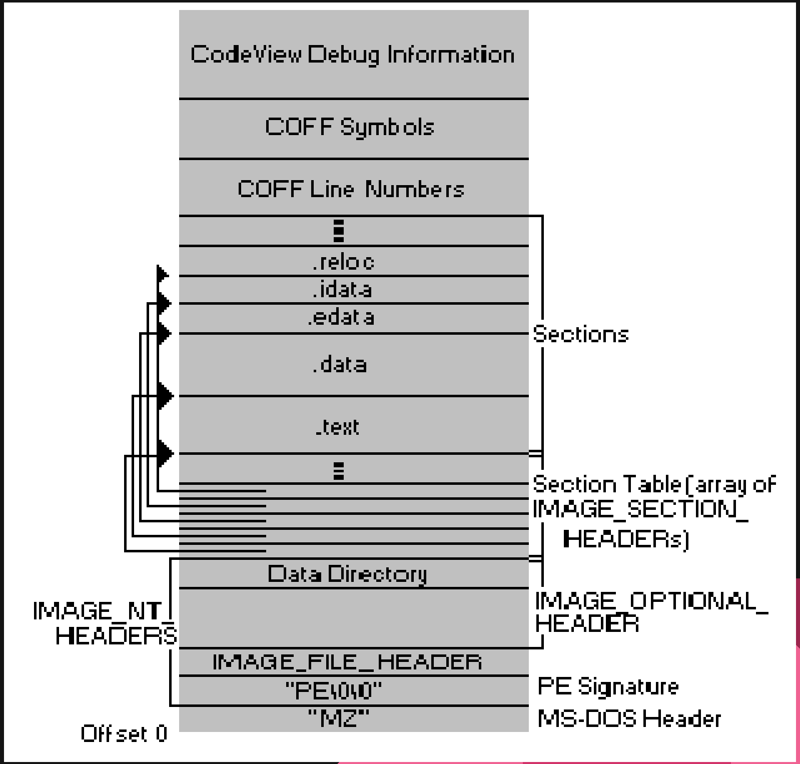

https://github.com/angr/angr/blob/master/angr/simos/windows.py

```
            for j, c in enumerate(path):
                # if this segfaults, increase the allocation size
                state.mem[string_area + j*2].short = ord(c)
            state.mem[string_area + string_size].short = 0
            string_area += string_size + 2

        # handle the links. we construct a python list in the correct order for each, and then, uh,
        mem_order = sorted(self.project.loader.all_pe_objects, key=lambda x: x.mapped_base)
        init_order = []
        partially_loaded = set()

        def fuck_load(x):
            if x.provides in partially_loaded:
                return
            partially_loaded.add(x.provides)
            for dep in x.deps:
                if dep in self.project.loader.shared_objects:
                    depo = self.project.loader.shared_objects[dep]
                    fuck_load(depo)
                    if depo not in init_order:
                        init_order.append(depo)

        fuck_load(self.project.loader.main_object)
        load_order = [self.project.loader.main_object] + init_order

        def link(a, b):
            state.mem[a].dword = b
            state.mem[b+4].dword = a

        # I have genuinely never felt so dead in my life as I feel writing this code
        def link_list(mods, offset):
```

- “OKAY ITS TIME TO SUFFER”
- “HACK HACK HACK HACK”
- “f***_load”
- “I have genuinely never felt so dead in my life as I feel writing this code”

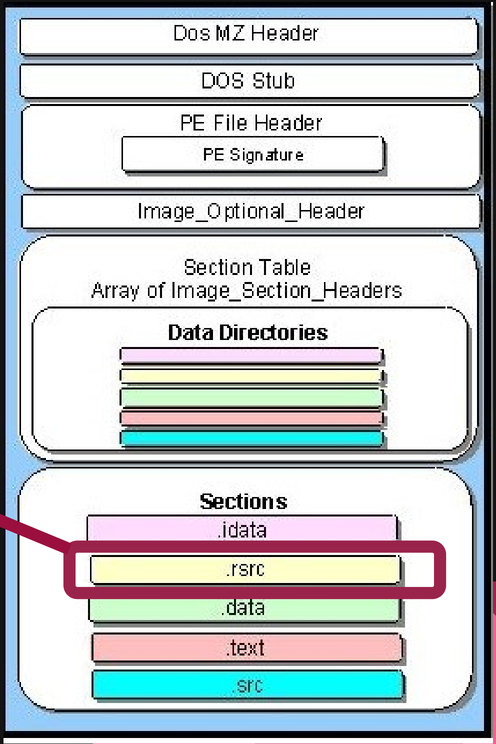
Pushing it Further » Parsing the PE File

pefile.py



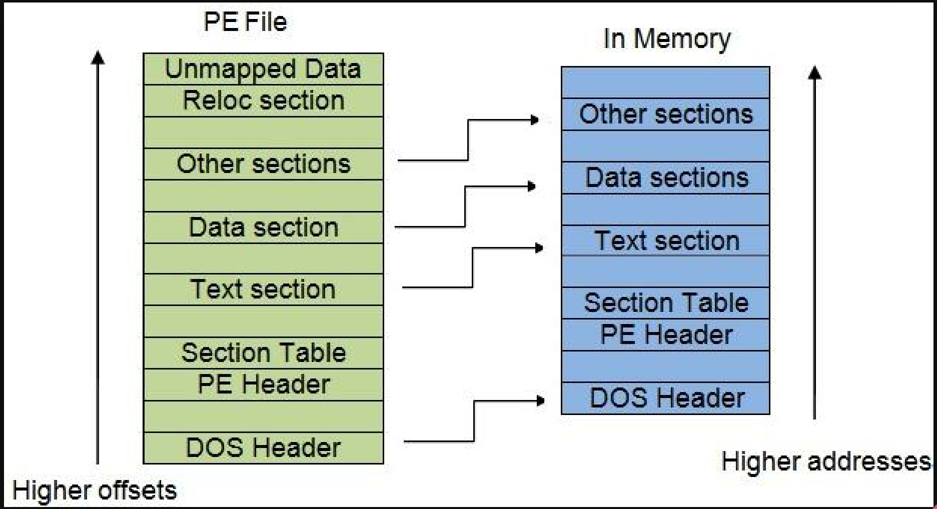
Pushing it Further » Understanding Sections

```
typedef struct _IMAGE_SECTION_HEADER {
  UCHAR Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    ULONG PhysicalAddress;
    ULONG VirtualSize;
  } Misc;
  ULONG VirtualAddress;
  ULONG SizeOfRawData;
  ULONG PointerToRawData;
  ULONG PointerToRelocations;
  ULONG PointerToLinenumbers;
  USHORT NumberOfRelocations;
  USHORT NumberOfLinenumbers;
  ULONG Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```



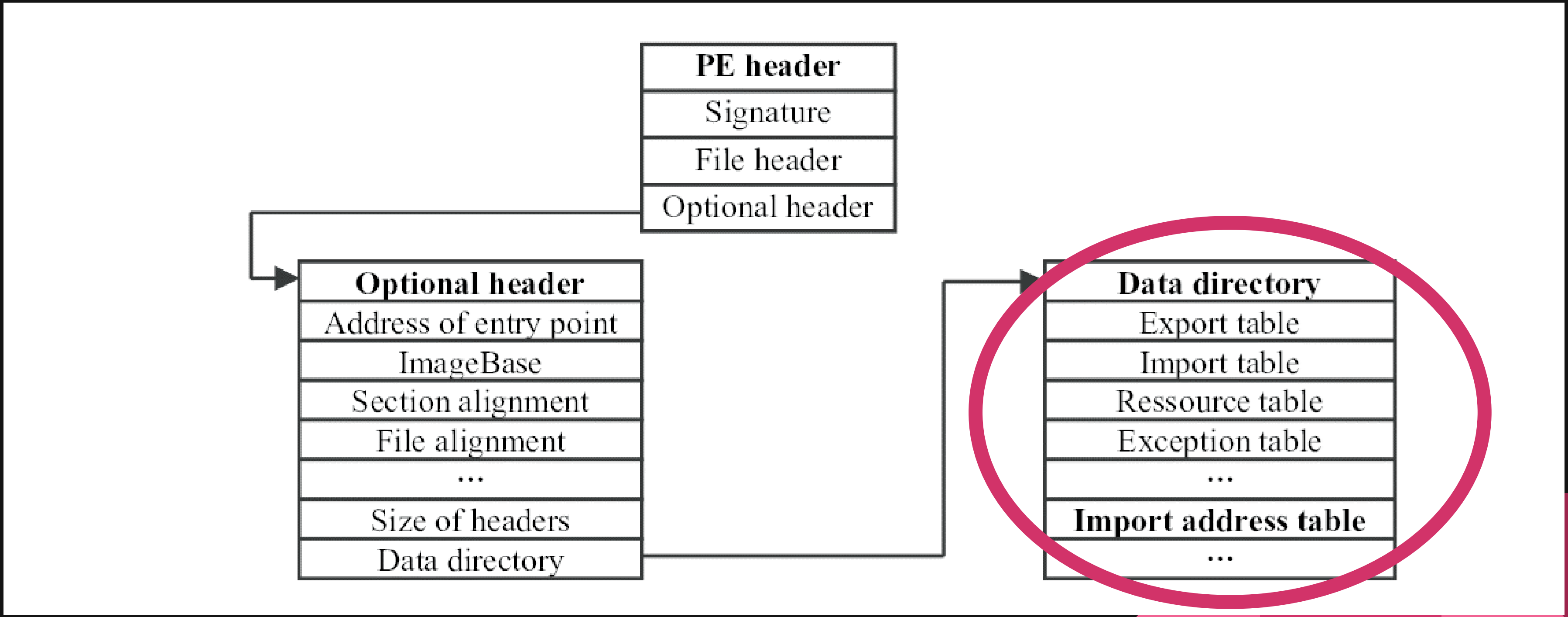
Pushing it Further » Mapping Sections

- **IMAGEBASE + VirtualAddress** of size **VirtualSize**

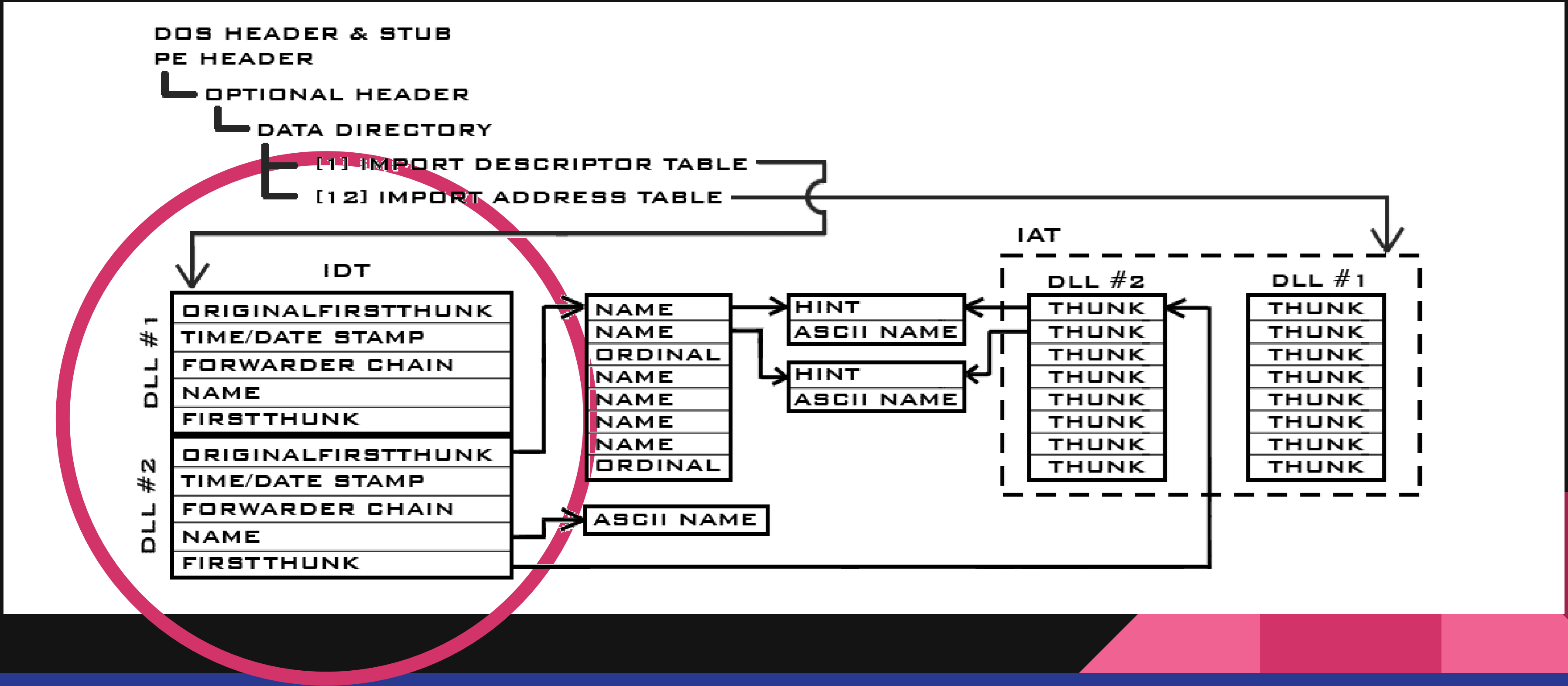


Don’t forget to map the headers too!

Pushing it Further » Importing DLLs

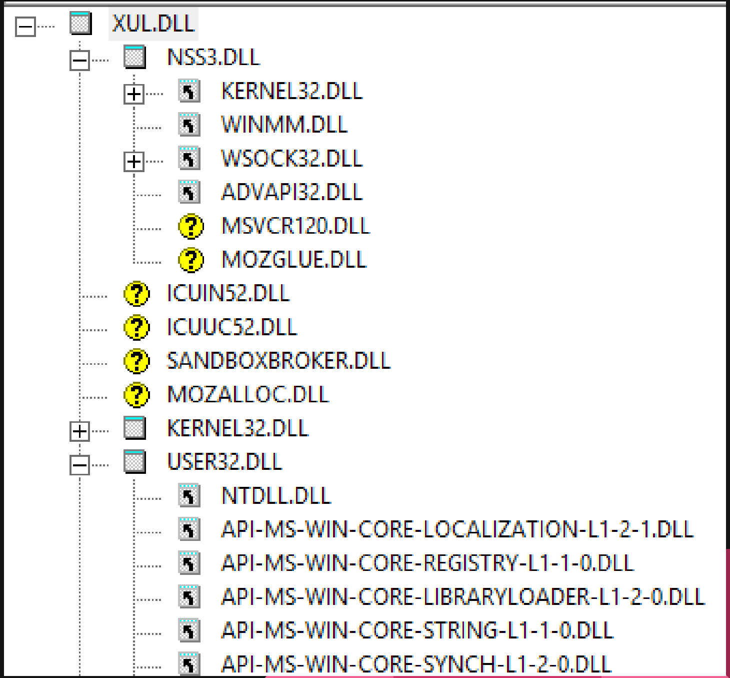


Pushing it Further » Importing DLLs



Pushing it Further » Importing DLLs

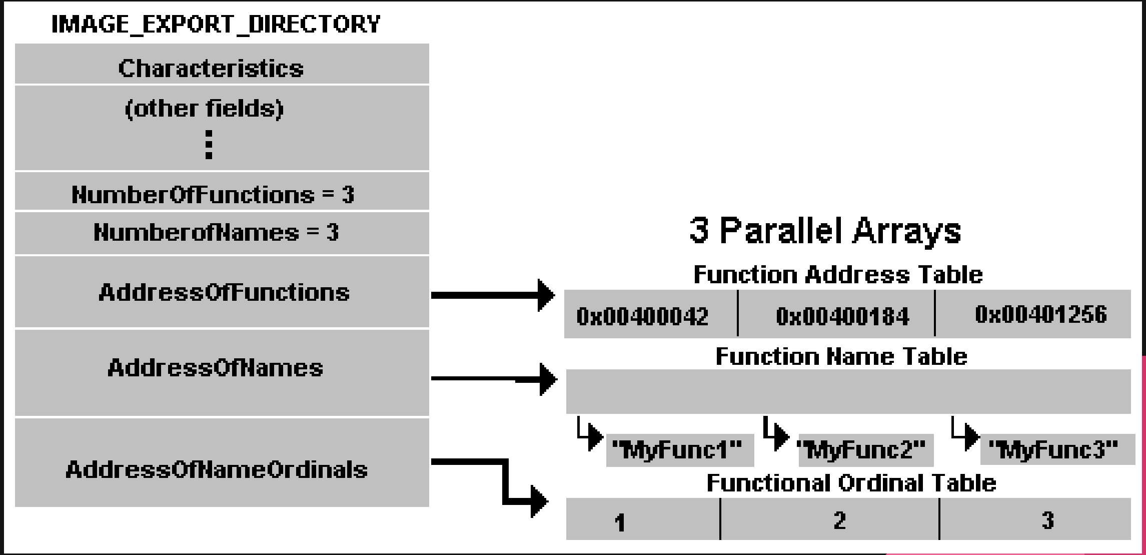
- Recursively **load DLLs**
- **Fun fact:**
  - **System32** contains **64** bit binaries
  - **SysWOW64** contains **32** bit binaries
  - Windows ¯\_(ツ)_/¯



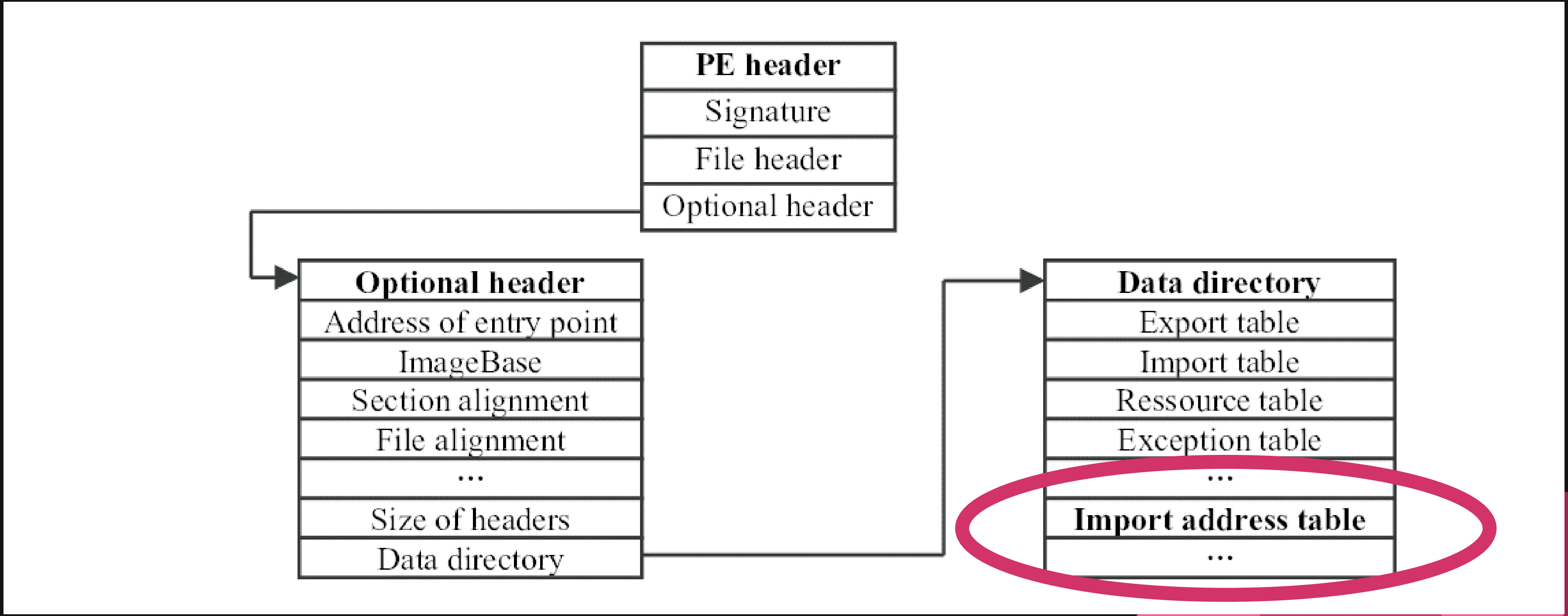
```
XUL.DLL
  NSS3.DLL
    KERNEL32.DLL
    WINMM.DLL
    WSOCK32.DLL
    ADVAPI32.DLL
    MSVCR120.DLL
    MOZGLUE.DLL
  ICUIN52.DLL
  ICUUC52.DLL
  SANDBOXBROKER.DLL
  MOZALLOC.DLL
  KERNEL32.DLL
  USER32.DLL
    NTDLL.DLL
    API-MS-WIN-CORE-LOCALIZATION-L1-2-1.DLL
    API-MS-WIN-CORE-REGISTRY-L1-1-0.DLL
    API-MS-WIN-CORE-LIBRARYLOADER-L1-2-0.DLL
    API-MS-WIN-CORE-STRING-L1-1-0.DLL
    API-MS-WIN-CORE-SYNCH-L1-2-0.DLL
```

Pushing it Further » Loading DLLs

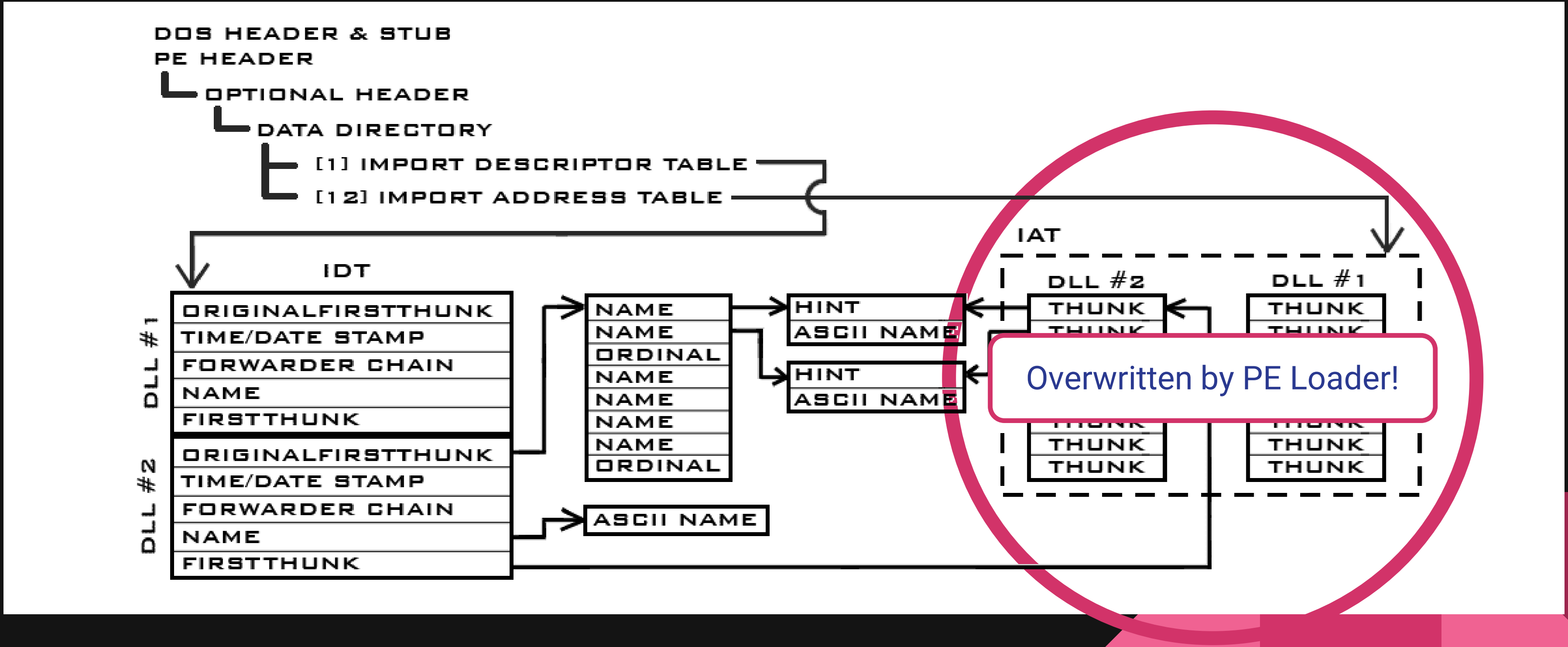
- Map Sections
- Save mapping of **Name of Functions** and **RVA of Exported Functions**



Pushing it Further » Importing DLLs



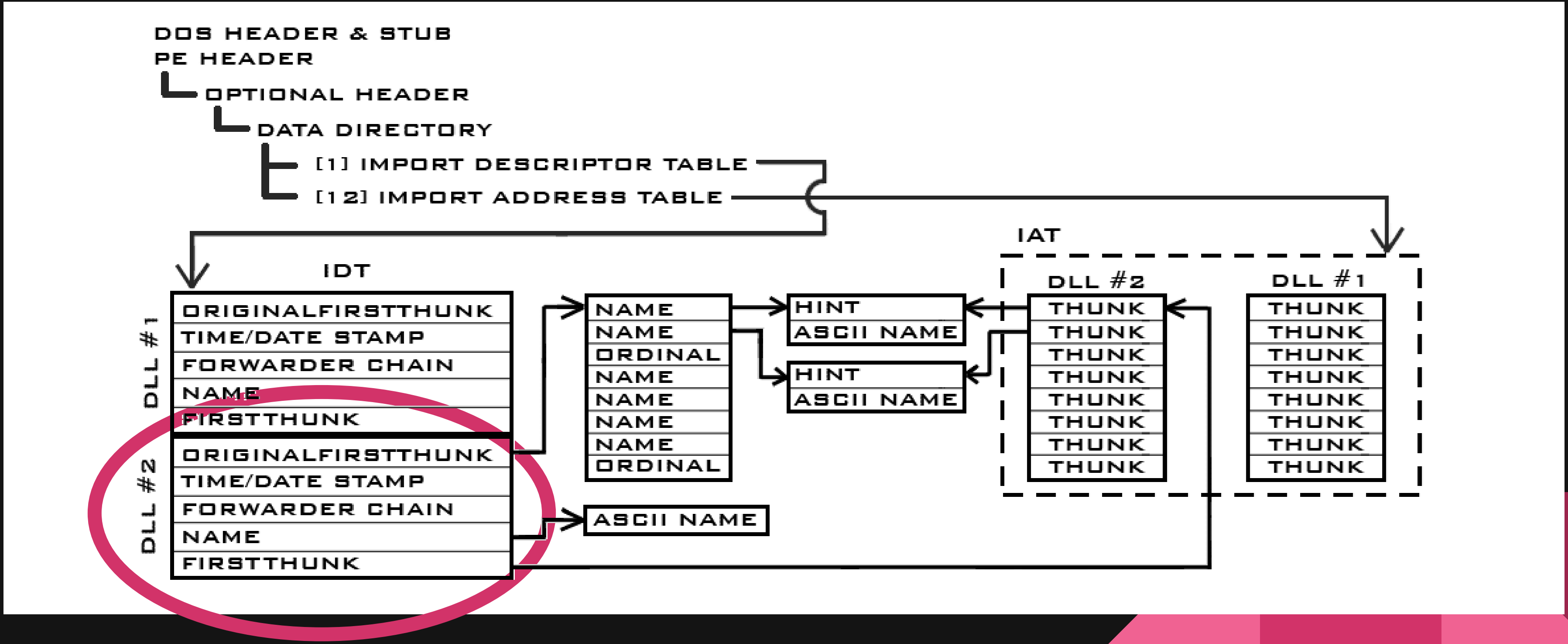
Pushing it Further » Loading the IAT



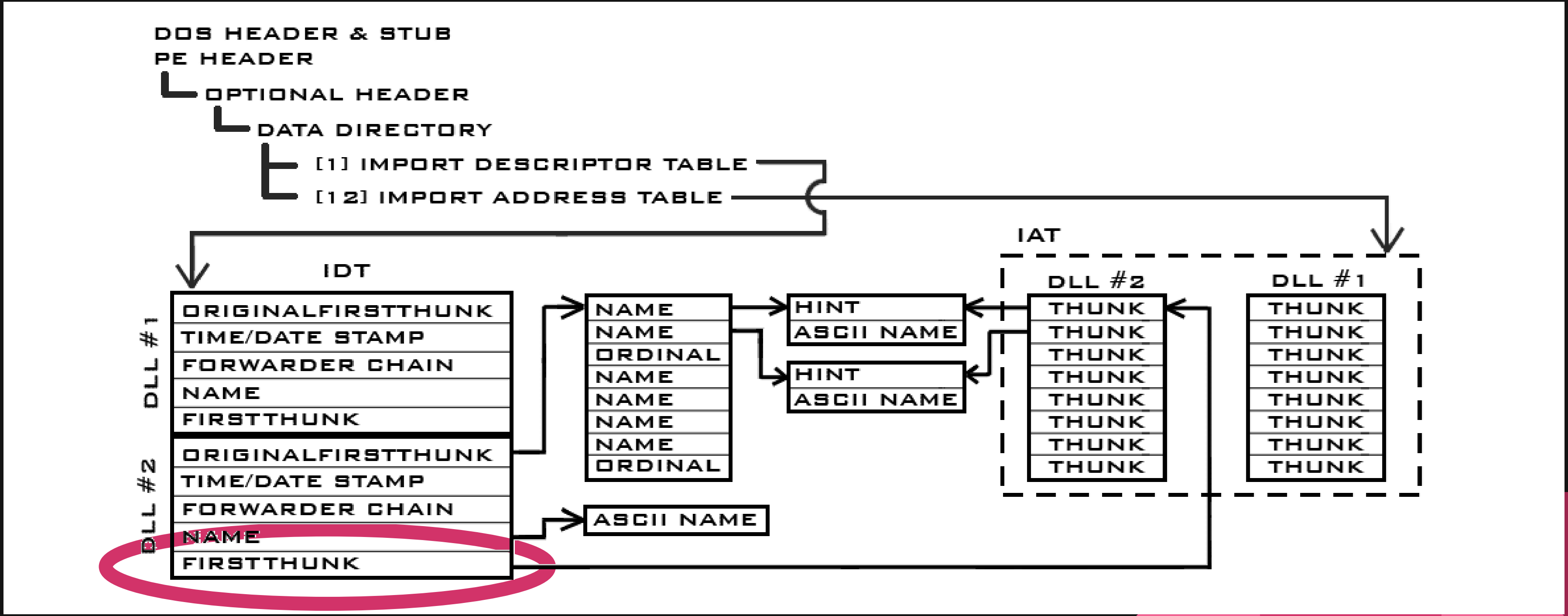
Pushing it Further » Overwriting Thunks

- For each DLL, find their first thunk
- Iterate and find the NULL thunk to find the number of thunks
- For each thunk:
  - Dereference thunk. If they are imported by name, use the internally stored name and RVA to replace thunk
  - If they are imported by ordinal, where the value is > 0x80000000, skip (*handled by .edata section*)

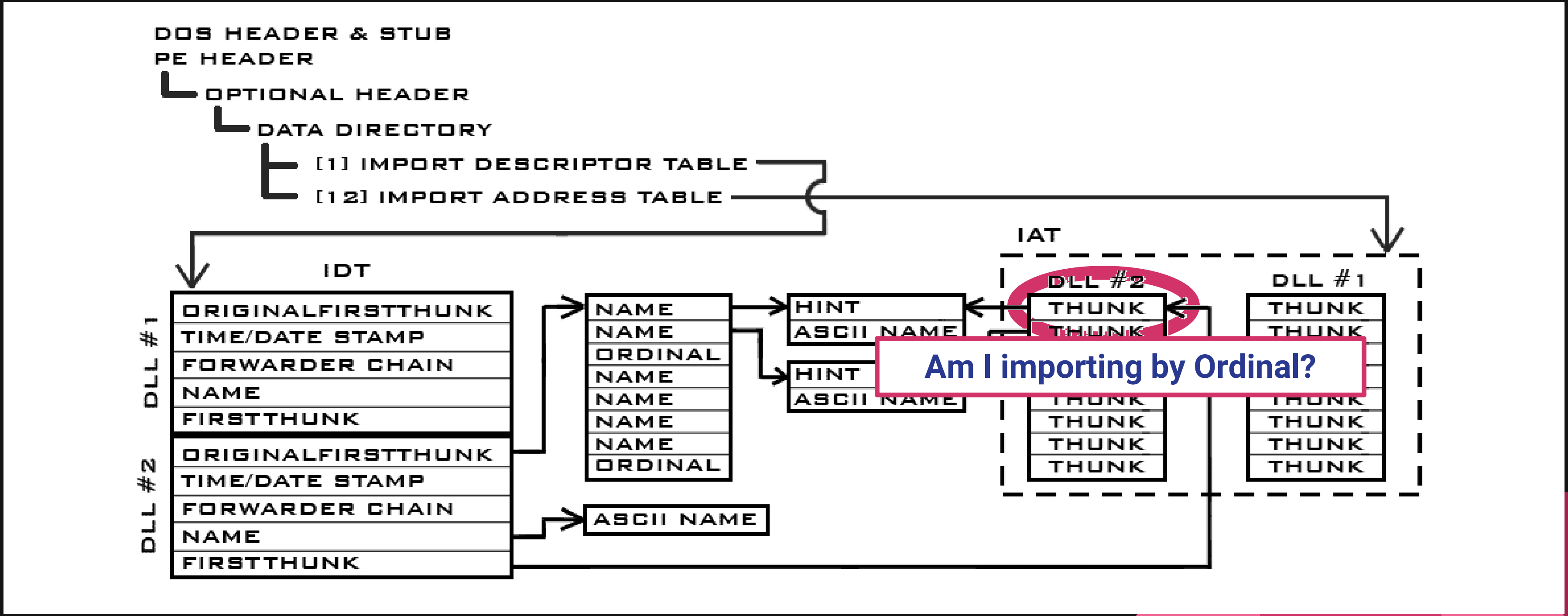
Pushing it Further » Overwriting Thunks



Pushing it Further » Overwriting Thunks

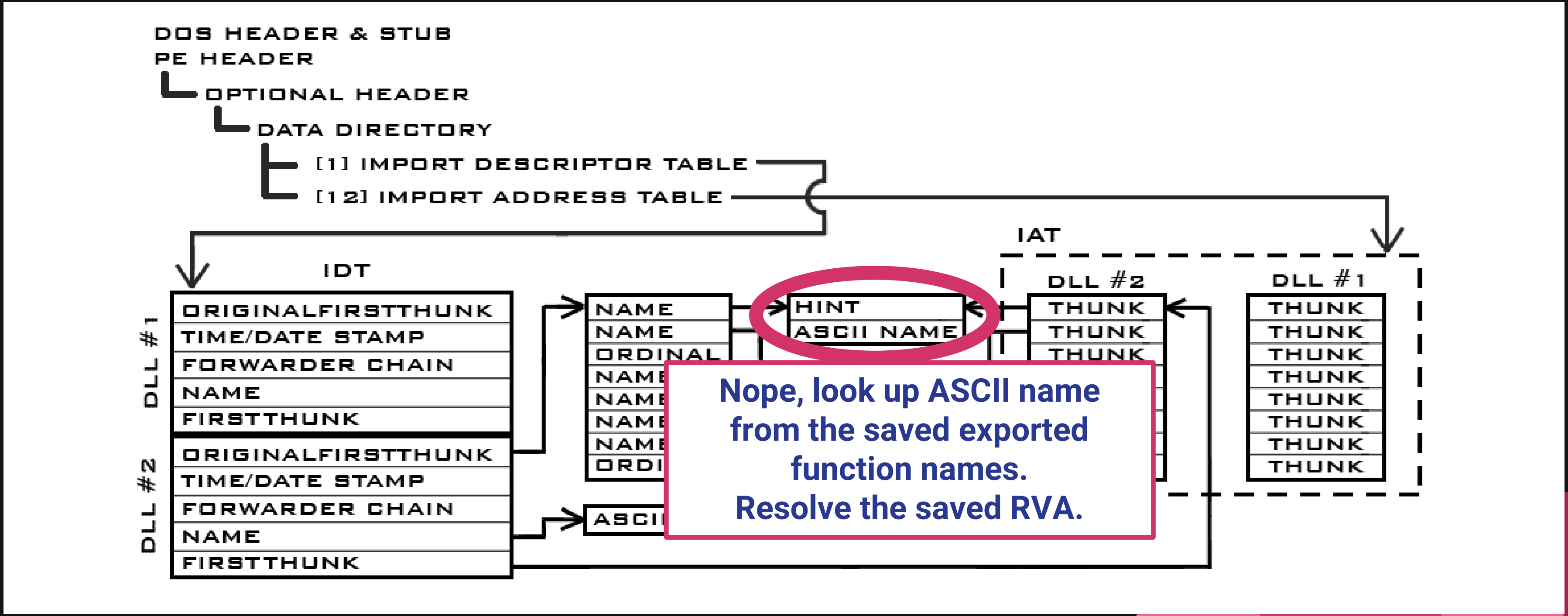


Pushing it Further » Overwriting Thunks



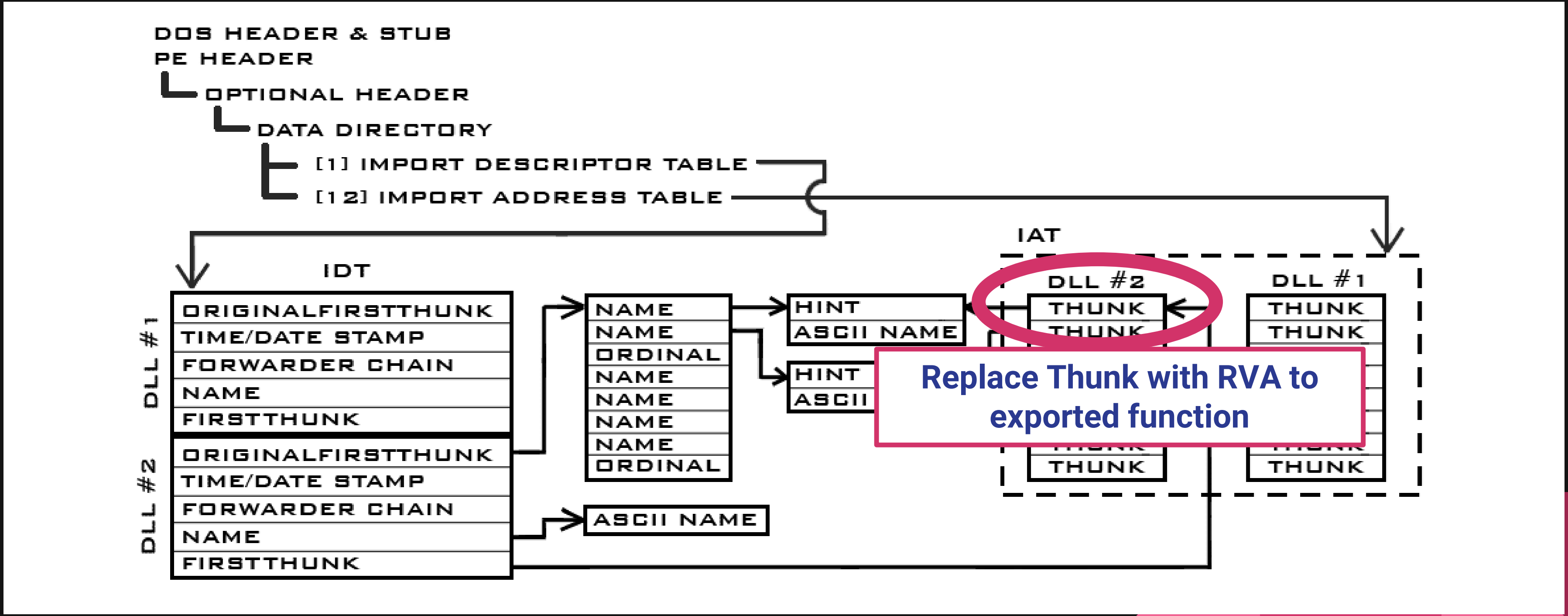
Am I importing by Ordinal?

Pushing it Further » Overwriting Thunks



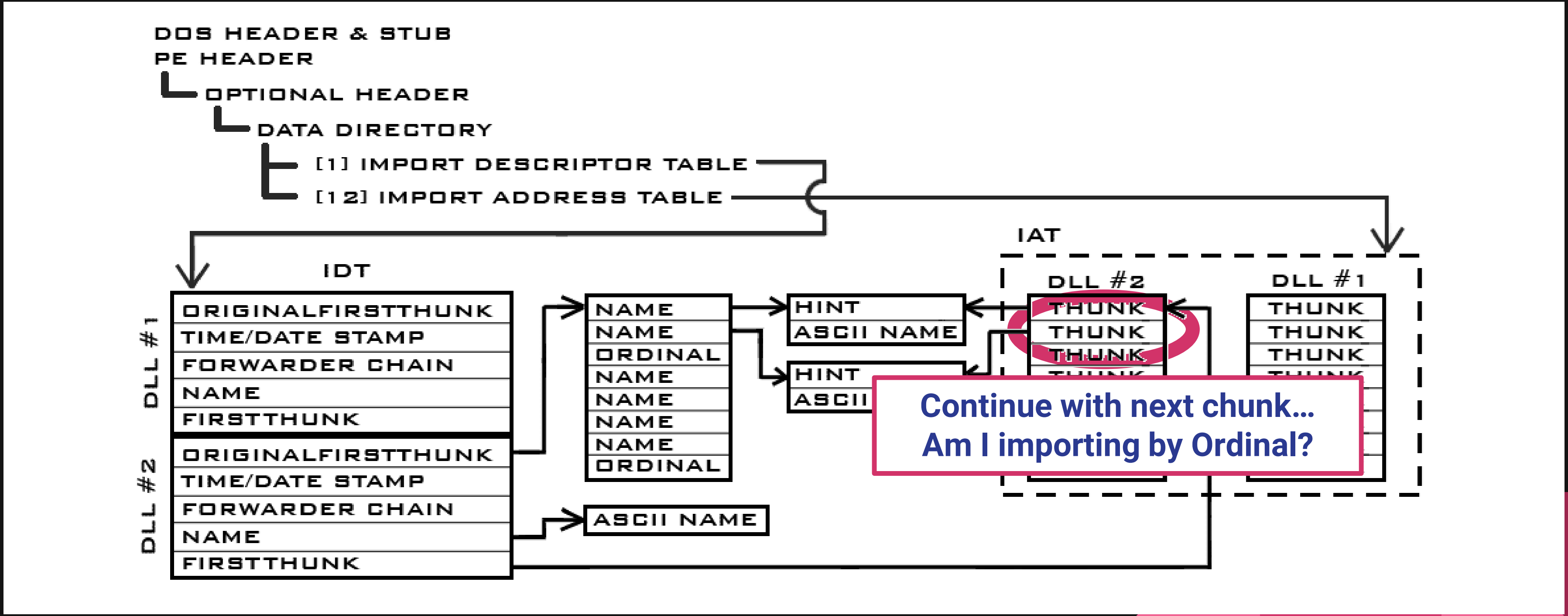
Nope, look up ASCII name from the saved exported function names.
Resolve the saved RVA.

Pushing it Further » Overwriting Thunks



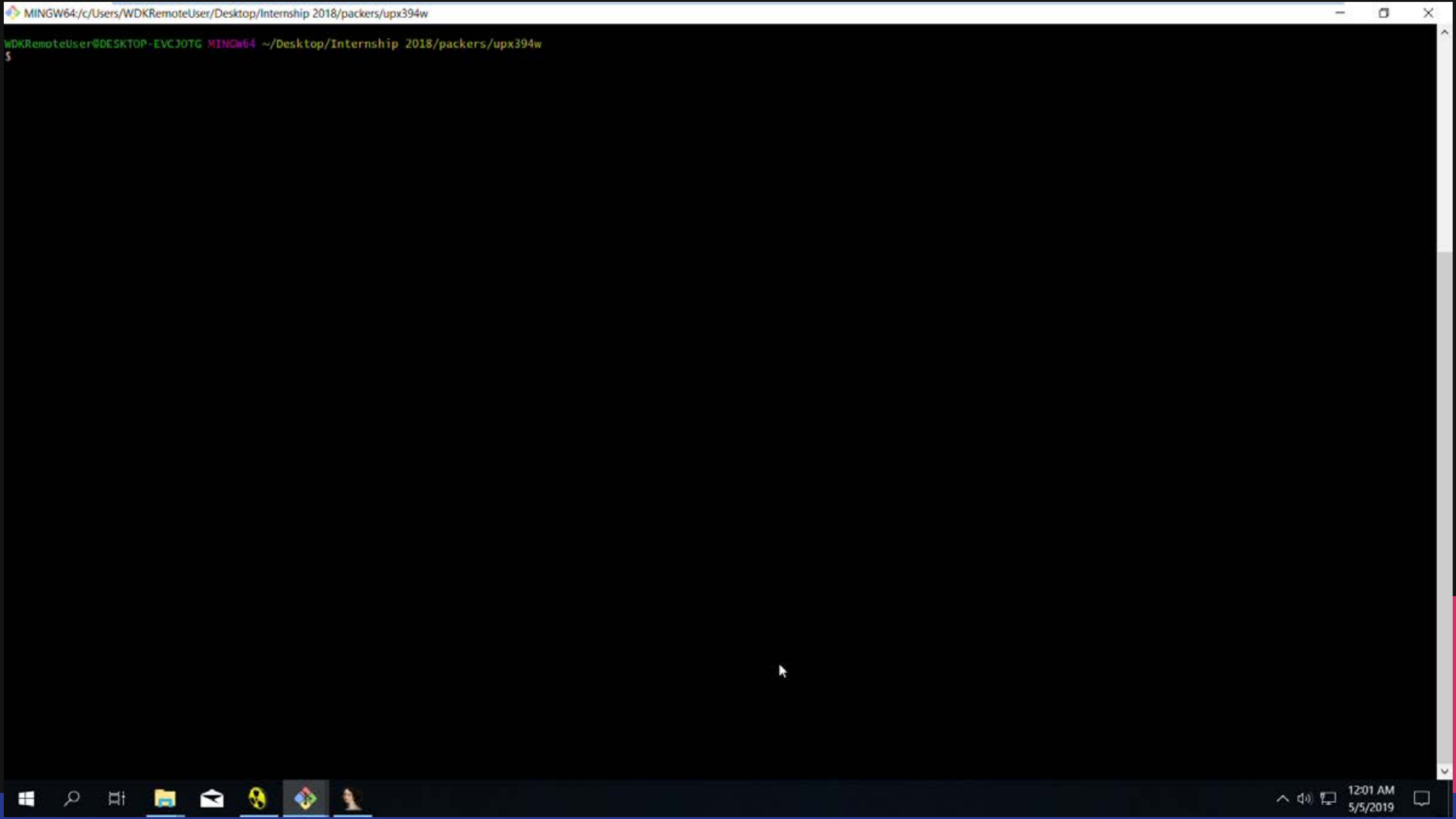
Replace Thunk with RVA to exported function

Pushing it Further » Overwriting Thunks



Continue with next chunk...
Am I importing by Ordinal?

Pushing it Further » Unpacking UPX


MINGW64:/c/Users/WDKRemoteUser/Desktop/Internship 2018/packers/upx394w

```
WDKRemoteUser@DESKTOP-EVCJOTG MINGW64 ~/Desktop/Internship 2018/packers/upx394w
$
```

Pushing it Further » Are We Done?

No.

Pushing it Further » API Call Obfuscation

❖ Meterpreter Packer Shellcode:

```
api_call:
   pushad               ; We preserve all the registers for the caller, bar EAX and ECX.
   mov ebp, esp           ; Create a new stack frame
   xor edx, edx          ; Zero EDX
   mov edx, [fs:edx+48]   ; Get a pointer to the PEB
   mov edx, [edx+12]      ; Get PEB->Ldr
   mov edx, [edx+20]      ; Get the first module from the InMemoryOrder module list
                              (this is a doubly-linked list, and is important!)
 next_mod:               ;
   mov esi, [edx+40]      ; Get pointer to modules name (unicode string) //BaseDllName
   movzx ecx, word [edx+38] ; Set ECX to the length we want to check
   xor edi, edi           ; Clear EDI which will store the hash of the module name
```

Pushing it Further » API Call Obfuscation

❖ Meterpreter Packer Shellcode:

```
api_call:
   pushad               ; We preserve all the registers for the caller, bar EAX and ECX.
   mov ebp, esp           ; Create a new stack frame
   xor edx, edx           ; Zero EDX
   mov edx, [fs:edx+48]   ; Get a pointer to the PEB
   mov edx, [edx+12]      ; Get PEB->Ldr
   mov edx, [edx+20]      ; Get the first module from the InMemoryOrder module list
                              (this is a doubly-linked list, and is important!)
  next_mod:              ;
   mov esi, [edx+40]      ; Get pointer to modules name (unicode string) //BaseDllName
   movzx ecx, word [edx+38] ; Set ECX to the length we want to check
   xor edi, edi           ; Clear EDI which will store the hash of the module name
```

Pushing it Further » API Call Obfuscation

❖ Meterpreter Packer Shellcode:

```
api_call:
    pushad              ; We preserve all the registers for the caller, bar EAX and ECX.
    mov ebp, esp        ; Create a new stack frame
    xor edx, edx        ; Zero EDX
    mov edx, [fs:edx+48]  ; Get a pointer to the PEB
    mov edx, [edx+12]   ; Get PEB->Ldr
    mov edx, [edx+20]   ; Get the first module from the InMemoryOrder module list
                             (this is a doubly-linked list, and is important!)
  next_mod:             ;
    mov esi, [edx+40]   ; Get pointer to modules name (unicode string) //BaseDllName
    movzx ecx, word [edx+38] ; Set ECX to the length we want to check
    xor edi, edi        ; Clear EDI which will store the hash of the module name
```

Pushing it Further » TEB

```
struct _TEB {
0x000 _NT_TIB NtTib;
0x01c void* EnvironmentPointer;
0x020 _CLIENT_ID ClientId;
0x028 void* ActiveRpcHandle;
0x02c void* ThreadLocalStoragePointer;
0x030 _PEB* ProcessEnvironmentBlock;
0x034 DWORD LastErrorValue;
…
};
```

Pushing it Further » PEB

```
struct _PEB {
0x000 BYTE InheritedAddressSpace;
0x001 BYTE ReadImageFileExecOptions;
0x002 BYTE BeingDebugged;
0x003 BYTE SpareBool;
0x004 void* Mutant;
0x008 void* ImageBaseAddress;
0x00c _PEB_LDR_DATA* Ldr;
0x010 _RTL_USER_PROCESS_PARAMETERS*
ProcessParameters;
0x014 void* SubSystemData;
0x018 void* ProcessHeap;
…
}
```

Pushing it Further » API Call Obfuscation

❖ Meterpreter Packer Shellcode:

```
api_call:
   pushad               ; We preserve all the registers for the caller, bar EAX and ECX.
   mov ebp, esp           ; Create a new stack frame
   xor edx, edx          ; Zero EDX
   mov edx, [fs:edx+48]   ; Get a pointer to the PEB
   mov edx, [edx+12]      ; Get PEB->Ldr
   mov edx, [edx+20]      ; Get the first module from the InMemoryOrder module list
                              (this is a doubly-linked list, and is important!)
  next_mod:              ;
   mov esi, [edx+40]      ; Get pointer to modules name (unicode string) //BaseDllName
   movzx ecx, word [edx+38] ; Set ECX to the length we want to check
   xor edi, edi          ; Clear EDI which will store the hash of the module name
```

Pushing it Further » PEB_LDR_DATA

```
typedef struct _PEB_LDR_DATA
{
0x00 ULONG Length;
0x04 BOOLEAN Initialized;
0x08 PVOID SsHandle;
0x0c LIST_ENTRY InLoadOrderModuleList;
0x14 LIST_ENTRY InMemoryOrderModuleList;
0x1c LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA,*PPEB_LDR_DATA;
```

Pushing it Further » PEB_LDR_DATA

```
typedef struct _PEB_LDR_DATA
{
0x00 ULONG Length;
0x04 BOOLEAN Initialized;
0x08 PVOID SsHandle;
0x0c LIST_ENTRY InLoadOrderModuleList;
0x14 LIST_ENTRY InMemoryOrderModuleList;
0x1c LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA,*PPEB_LDR_DATA;
```

These point to imported DLL metadata!

Pushing it Further » LIST_ENTRY

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

Pushing it Further » LDR_DATA_TABLE_ENTRY

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
  LIST_ENTRY InLoadOrderLinks; /* 0x00 */
  LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
  LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
  PVOID DllBase; /* 0x18 */
  PVOID EntryPoint;
  ULONG SizeOfImage;
  UNICODE_STRING FullDllName; /* 0x24 */
  UNICODE_STRING BaseDllName;
  ULONG Flags;
  ...
}
```

Pushing it Further » LDR_DATA_TABLE_ENTRY

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
  LIST_ENTRY InLoadOrderLinks; /* 0x00 */
  LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
  LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
  PVOID DllBase; /* 0x18 */
  PVOID EntryPoint;
  ULONG SizeOfImage;
  UNICODE_STRING FullDllName; /* 0x24 */
  UNICODE_STRING BaseDllName;
  ULONG Flags;
  ...
}
```

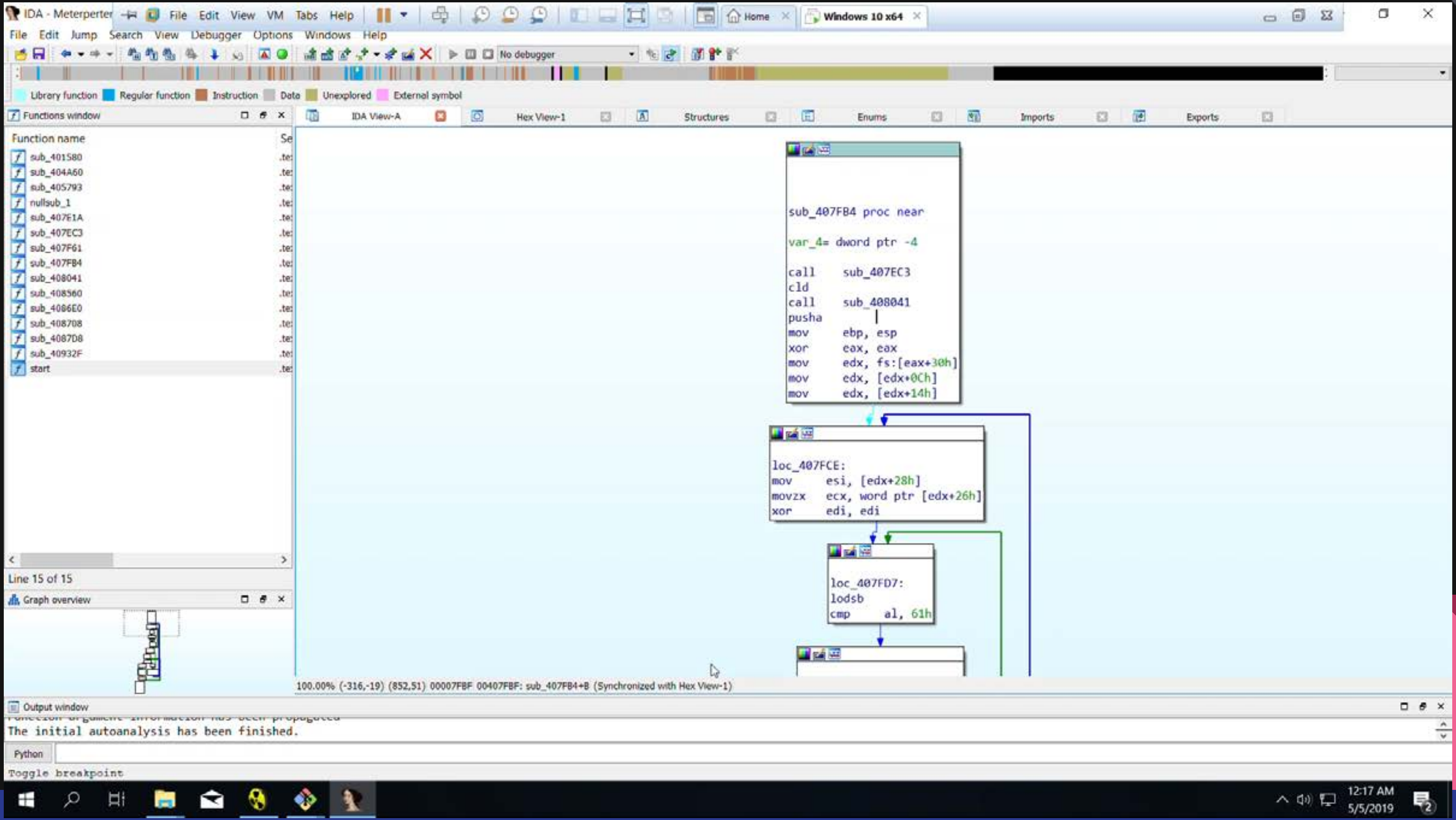
Implement Doubly Linked Lists

Pushing it Further » LDR_DATA_TABLE_ENTRY

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    ...
}
```

Each time a DLL is loaded **create an entry populating this**

Pushing it Further » Unpacking Shikata-ga-nai



```
sub_407FB4 proc near

var_4= dword ptr -4

call    sub_407EC3
cld
call    sub_408041
pusha
mov     ebp, esp
xor     eax, eax
mov     edx, fs:[eax+30h]
mov     edx, [edx+0Ch]
mov     edx, [edx+14h]

loc_407FCE:
mov     esi, [edx+28h]
movzx   ecx, word ptr [edx+26h]
xor     edi, edi

loc_407FD7:
lodsb
cmp     al, 61h
```

```
Function name
sub_401580
sub_404A60
sub_405793
nullsub_1
sub_407E1A
sub_407EC3
sub_407F61
sub_407FB4
sub_408041
sub_408560
sub_4086E0
sub_408708
sub_4087D8
sub_40932F
start
```

```
100.00% (-316,-19) (852,51) 00007FBF 00407FBF: sub_407FB4+B (Synchronized with Hex View-1)
```

```
The initial autoanalysis has been finished.
```

Pushing it Further » How Do I Know?

- Try with different programs
- Write a debugger on top of Unicorn
    - Use **UC_HOOK_CODE** and dump memory at or before crash regions
    - **Run with a debugger on Windows** and dump the same memory; compare and see what could be different
    - Infer potential structs and how they can be loaded

Conclusion

- Windows is hard

Thank you for listening!