CROWDSTRIKE

FROM JMP ESP TO CET: THE CURRENT STATE OF EXPLOIT DEVELOPMENT

CONNOR MCGARR, ASSOCIATE RED TEAM CONSULTANT, CROWDSTRIKE

- $ whoami
- Why Do We Care About Binary Exploitation?
- Vulnerability Classes
- Exploit Mitigations and Bypassing Them
- The Future Of Exploitation
- Is Vulnerability Research Still Practical For Red Teams/Non-Research Firms?

$ WHOAMI

About me

CROWDSTRIKE

$ WHOAMI

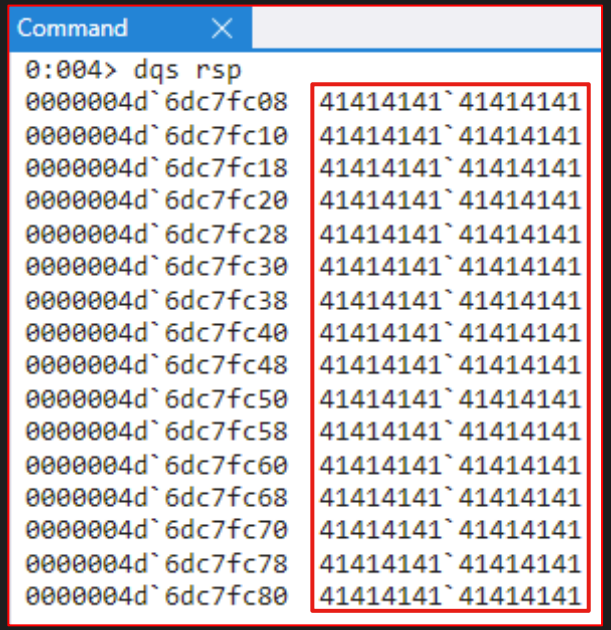
- Connor McGarr
- Associate Red Team Consultant at CrowdStrike
- OSCP | OSCE
- Interested in…
  - Exploit development, vulnerability research, and red teaming
- Twitter:
  - @33y0re
- Personal blog:
  - https://connormcgarr.github.io
- CrowdStrike Red Team Blog:
  - https://www.crowdstrike.com/blog/author/red-team-labs/

WHY DO WE CARE ABOUT BINARY EXPLOITATION?

What does this talk aim to address?

WHY DO WE CARE ABOUT BINARY EXPLOITATION?

- Exploit development
  - Memory corruption vulnerabilities
    - Taking advantage of flaws within compiled code
  - “Point and shoot”
    - No social engineering required
    - Remote exploits often are unauthenticated
  - Cons:
    - Often difficult and tedious to develop
      - Low level skills often needed
      - Mitigations hurt portability
      - Time consuming to develop



WHY DO WE CARE ABOUT BINARY EXPLOITATION? (CONT'D)

- Red team toolkits once contained many binary exploits
  - Software wasn't written as well as today
  - Operating systems didn't natively support exploit mitigations
  - Stack overflows were very common and were easy to exploit "back in the day"
    - Overflow a stack allocation -> overwrite the return address to jump to shellcode, typically with a "JMP ESP" instruction -> profit
- As time has gone on, this is no longer the case
  - Red teams now place more focus on malware development and social engineering to execute code
- The question remains – is the price of admission to develop exploits portably and reliably worth it still for red teams/non-research firms?
  - Let's find out!

VULNERABILITY CLASSES

Exploitation “delivery mechanisms” – user mode & kernel mode

VULNERABILITY CLASSES

- Vulnerability classes refer to the way an exploit is delivered
- Common
  - Stack overflow
    - Overflow an allocation located on the stack and overwrite the return address
  - Use-after-free
    - An object on the heap is prematurely freed, but a reference to the object remains. Using a few primitives, an adversary could place a “fake/malicious” object in the freed object’s place and use the old reference/handle to that object to execute it
  - Type-confusion
    - An object of one data type is referenced as a different data type, typically due to developer error. Due to different data types have different layouts in memory, unintended memory may be referenced
  - Arbitrary writes
    - Very generic
    - Often the result of a vulnerability class, but doesn’t have to be
    - The ability to arbitrarily write data to a location – controlled or uncontrolled

VULNERABILITY CLASSES (CONT'D)

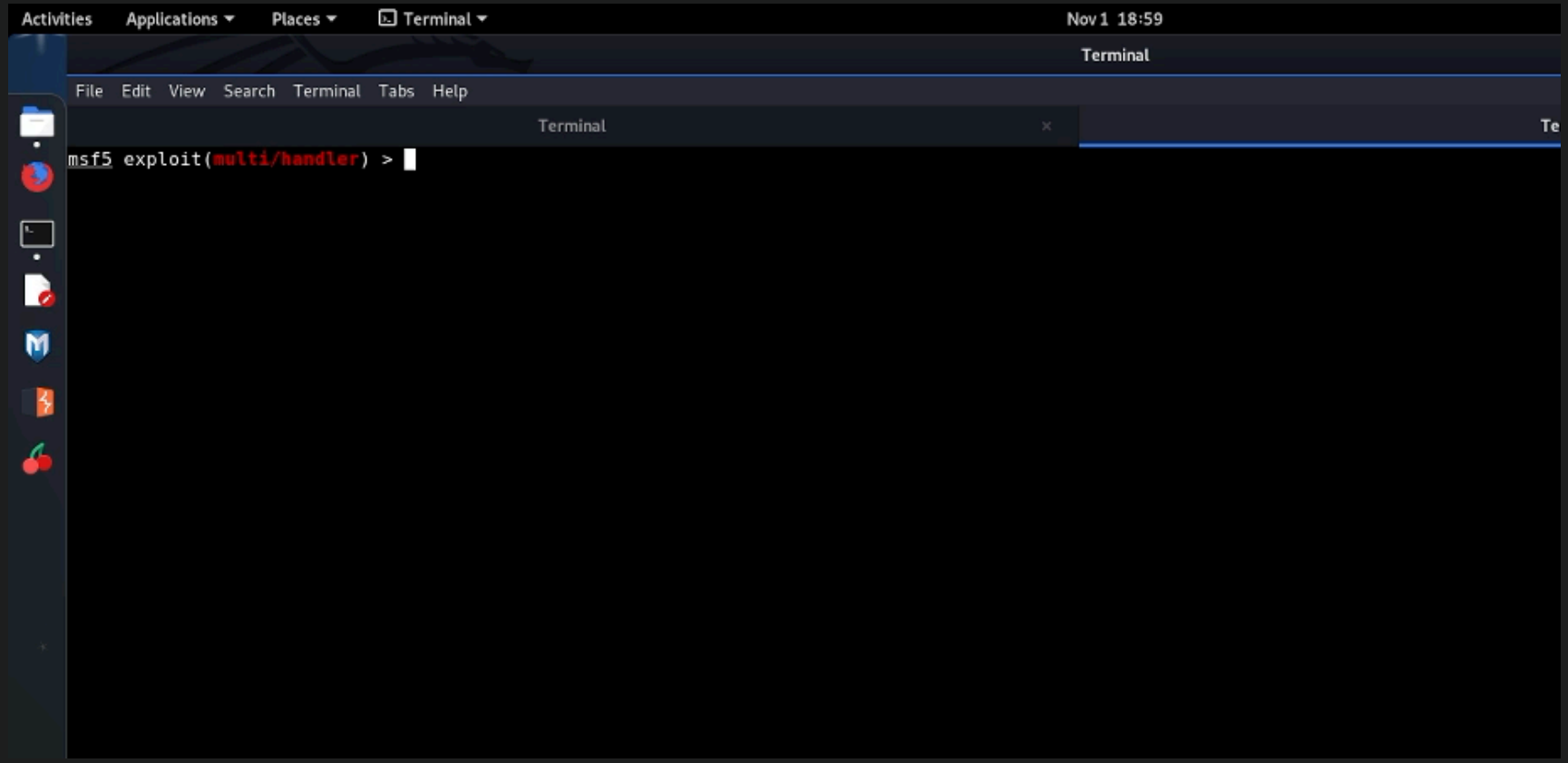
- Vulnerabilities are located in kernel mode and user mode
  - Due to the complexity of writing kernel mode drivers, third party developers often "just get it working" and may miss key security components
    - Results in more "trivial" vulnerability classes (but not limited to)
  - User mode code is often times more secure due to copious amounts of documentation
    - Exploitation often involves more complex exploit chains such as use-after-frees and type-confusions
  - At the end of the day, any of the aforementioned vulnerability classes can be located in kernel mode and/or user mode

VULNERABILITY CLASSES (CONT'D)

- User mode exploits are often combined with kernel mode exploits – known as a "local remote" exploit chain
  - An adversary may have a 0-day for a browser, such as Chrome or Firefox
    - This would result in initial access, but as a "sandboxed" unprivileged user
    - Sandboxes severely limit adversarial capabilities of an attacker. This results in the need for a privilege escalation exploit, usually in the form of a kernel exploit, in order to obtain privileges
- In some cases, kernel exploits can be exploited remotely
  - Commonly done in protocols that have some form of kernel interaction
    - Includes SMB exploits such as EternalBlue and SMBGhost
    - Also includes RDP exploits such as BlueKeep

VULNERABILITY CLASSES (CONT'D)

- Example demo of a "local remote" exploit chain
- Scenario
  - Vulnerable piece of remote software running on a Windows 7 host with all mitigations disabled
  - Vulnerable kernel mode driver running on the same host
- Pros
  - Two exploits away from remote, unauthenticated SYSTEM privileges
  - Doesn't require end user interaction such as social engineering
- Cons
  - Timely to develop two portable exploits, especially when mitigations come into play
  - As we will find out, some mitigations are less documented than others, which may mean additional research to develop the exploit – resulting in a greater overall cost



VULNERABILITY CLASSES (CONT'D)

- The lack of exploit mitigations makes the aforementioned exploitation trivial
  - What happens when more roadblocks are added?

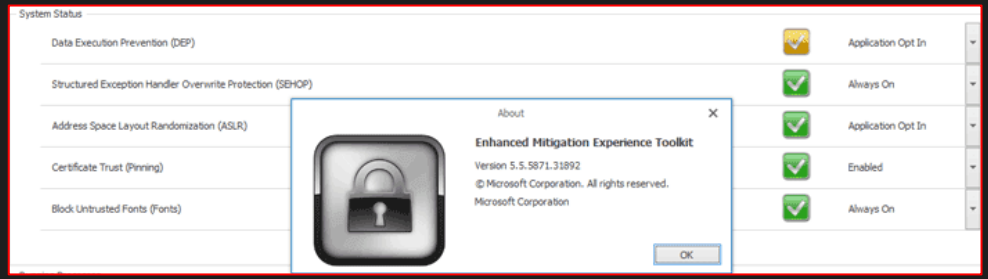
CROWDSTRIKE

EXPLOIT MITIGATIONS AND BYPASSING THEM

Addressing Windows specific exploit mitigations

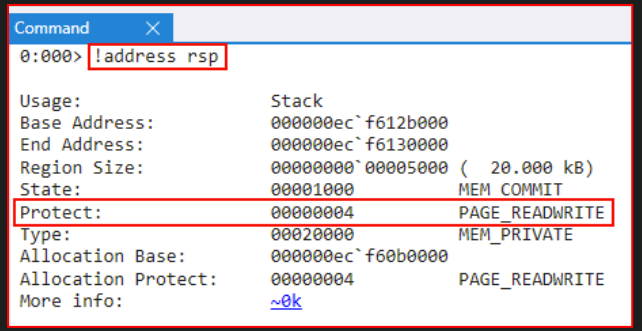
EXPLOIT MITIGATIONS AND BYPASSING THEM

- Exploit mitigations
  - Additional OS features, either hardware enforced or software enforced, that aim to break popular exploit methodologies or completely obliterate vulnerability classes all together
  - Modern systems, such as Windows 10, have many mitigations enabled by default on most Microsoft processes, that end users may not even be aware of
  - Difficulties remain getting third party developers to enable mitigations for their applications



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Data Execution Prevention (DEP)
  - Among the first mitigations put in place – Windows XP SP2
  - Places boundaries between "code" segments and "data" segments of memory
    - Memory is not both writable and executable
  - DEP prevents code execution in non-executable portions of memory
    - `STATUS_ACCESS_VIOLATION` status code
    - The stack is only writable and readable with DEP



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Code segments vs. data segments
  - Code segments refer to memory that contain executable instructions (opcodes)
    - CS segment register is referenced when executing code (CS:RIP)
    - Code segments are generally readable/executable – but NOT writable
  - `WriteProcessMemory` when disassembled, contains executable instructions – not data

```
loc_1400015B4:          ; lpNumberOfBytesWritten
mov     qword ptr [rsp+428h+flProtect], 0
mov     r9d, 362h       ; nSize
lea     r8, [rsp+428h+Buffer] ; lpBuffer
mov     rdx, [rsp+428h+lpBaseAddress] ; lpBaseAddress
mov     rcx, [rsp+428h+hProcess] ; hProcess
call    cs:WriteProcessMemory
mov     [rsp+428h+var_40], eax
cmp     [rsp+428h+var_40], 0
jnz     short loc_140001610
```

Command

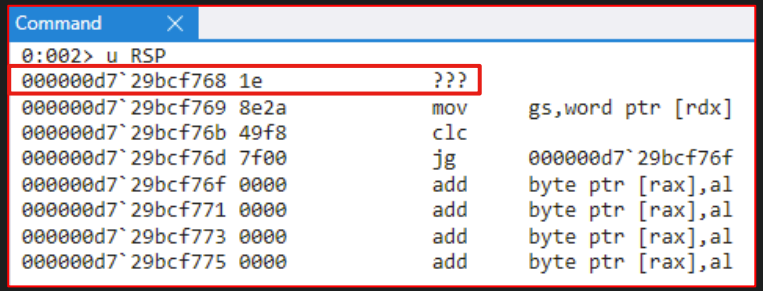
```
0:004> u KERNELBASE!WriteProcessMemory
KERNELBASE!WriteProcessMemory:
00007ff8`46a702b0 488bc4          mov     rax,rsp
00007ff8`46a702b3 48895808        mov     qword ptr [rax+8],rbx
00007ff8`46a702b7 4c894820        mov     qword ptr [rax+20h],r9
00007ff8`46a702bb 4c894018        mov     qword ptr [rax+18h],r8
00007ff8`46a702bf 48895010        mov     qword ptr [rax+10h],rdx
00007ff8`46a702c3 55              push    rbp
00007ff8`46a702c4 56              push    rsi
00007ff8`46a702c5 57              push    rdi
```

Command

```
0:002> !vprot KERNELBASE!WriteProcessMemory
BaseAddress:       00007ff846a70000
AllocationBase:    00007ff846a00000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00000000000f7000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              01000000  MEM_IMAGE
```

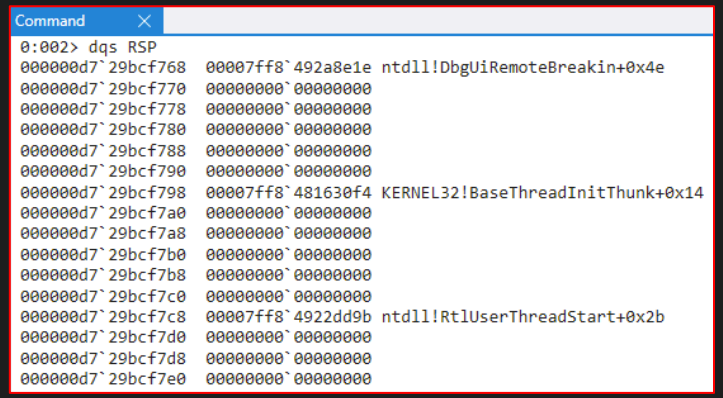

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Code segments vs. data segments
  - Data segments refer to memory used for things like strings, variables, and return addresses
  - Represented by the DS segment register. The SS segment register refers to the stack, and is also a type of data segment.
    - When attempting to view the opcodes that make up the RSP (stack) register, illegal instructions appear
      - This is because we are attempting to interpret data as code



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

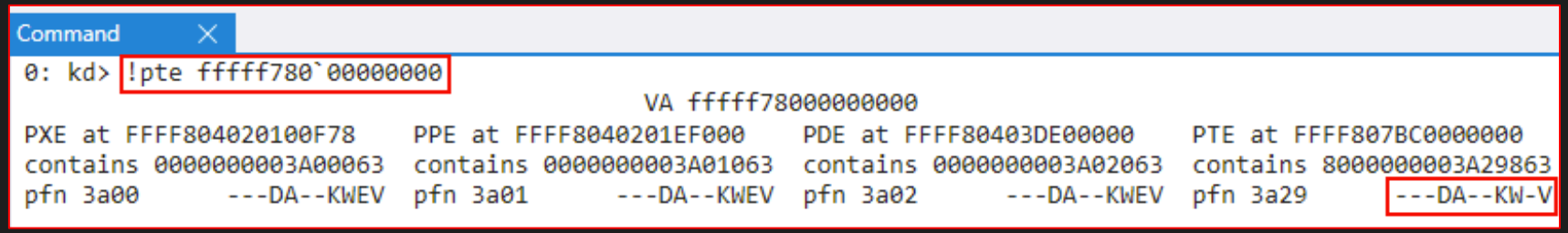
- Interpreting the stack as data, things make much more sense



- Why do we need to know the difference between code and data?
  - DEP leverages these concepts by not allowing code (executable instructions) to run in data portions of memory (most notably the stack and the heap)

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- DEP also has support in the kernel
  - The default allocation on Windows, before DEP, was `NonPagedPool`
    - Non-paged pool memory = executable memory that will always be backed by physical memory
- DEP is enforced in the kernel, through a page-table entry "control bit"
  - Page table entries enforce various properties/permissions of memory



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Scenario:
  - Let's say we have a vulnerability, in which we can write our own data to the stack. Before DEP, returning into the malicious data was enough for successful exploitation.
  - With DEP, this is no longer possible, as the stack isn't a code section of memory. How can we overcome this?
    - Can we take advantage of the fact the stack interprets its contents as data?
  - The most common way to do this is with an exploitation technique known as Return-Oriented Programming (ROP)

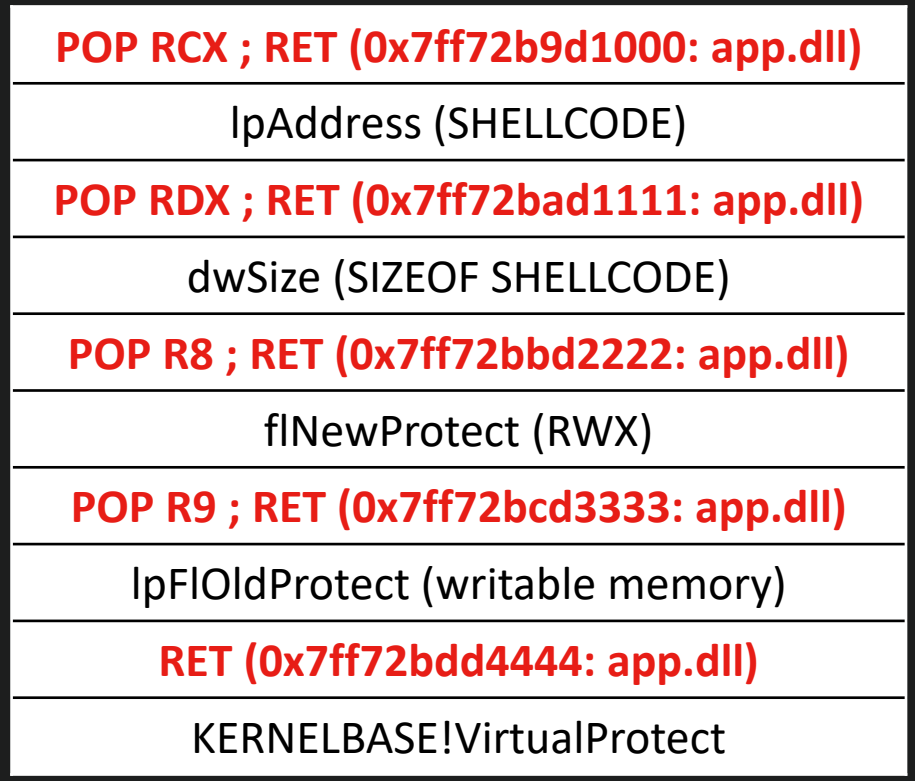
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- What is ROP?
  - "Code-reuse" attack
    - Adversaries can parse the a PE/DLL (application specific or native Windows DLLs) for pointers to existing code (generally from the .text section of an application – which generally contains an PE's executable instructions) instead of supplying their own code
      - Very much "living off the land"
    - Think of ROP like this
      - Making due with what you have
      - Can't supply our own code

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- ROP terminology
  - Gadget
    - 1 ROP instruction – `add rax, 0x100 ; ret`
  - Chain
    - A set of ROP gadgets - `add rax, 0x100 ; ret ; sub rbx, 0x100 ; ret`
- Caveats of ROP
  - Need stack control
    - Each return back to the stack picks up the next gadget
  - Limited to the program's instruction set
    - Try to avoid Windows native DLLs
- Bypassing DEP with ROP?
  - 64-bit + `__fastcall` calling convention

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)



RSP (Stack Pointer)

We can use pointers to gadgets to execute our own code while adhering to DEP!

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Address Space Layout Randomization (ASLR)
  - The address space of the machine is randomized on a per-boot basis

```
Command
0: kd> lm m nt
Browse full module list
start             end                 module name
fffff800`0fe00000 fffff800`108b6000   nt         (pdb symbols)
```

  - Upon reboot, `ntoskrnl.exe` is located at a different address

```
Command
0: kd> lm m nt
Browse full module list
start             end                 module name
fffff800`07000000 fffff800`07ab6000   nt         (pdb symbols)
```


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Addresses randomized in user mode and kernel mode
- Bypassing DEP with ROP requires being able to place pointers in memory on the stack
  - How can we know where these pointers are, if the addresses they are located in are randomized when a machine is rebooted?

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Three main methods
  - 1. Utilize application/application modules not compiled with ASLR
    - Does still exist, although farther and fewer between than historically
  - 2. Brute force the address space
    - With 32-bit applications that spawn a new thread/restart after a crash, it may be possible to brute force the address space
    - This is not feasible on 64-bit systems, due to a larger possible address space
  - 3. Information leaks
    - The "best" method
    - Includes vulnerabilities such as "out-of-bounds reads", generic information leaks, or the result of another vulnerability

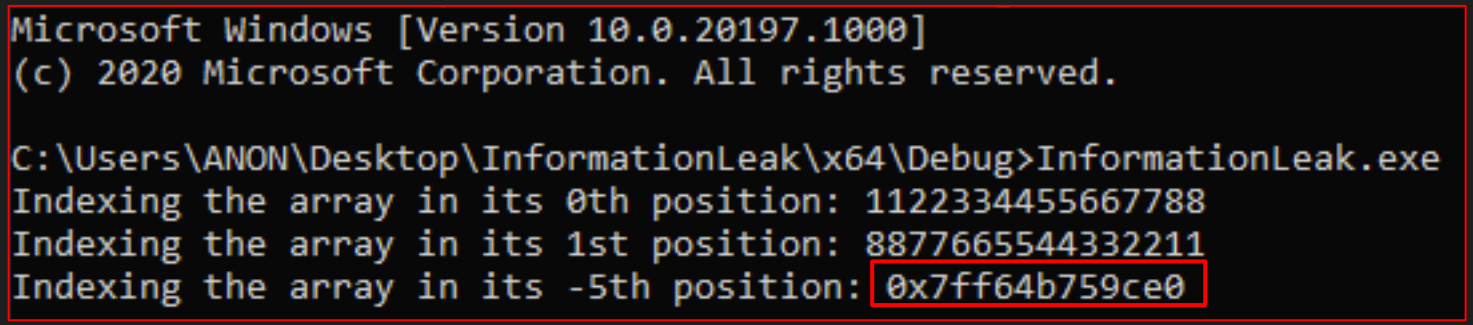
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Information leaks
  - Simple example
    - What if we possess the ability to index an array without any bounds checking?

```c
#include <stdio.h>

int main()
{
    unsigned long long array[5] = { 0x1122334455667788, 0x8877665544332211, 0x8811772266335544, 0x1111111111111111, 0x8888888888888888 };

    printf("Indexing the array in its 0th position: %llx\n", array[0]);
    printf("Indexing the array in its 1st position: %llx\n", array[1]);
    printf("Indexing the array in its -5th position: 0x%llx\n", array[-5]);

    return 0;
}
```


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

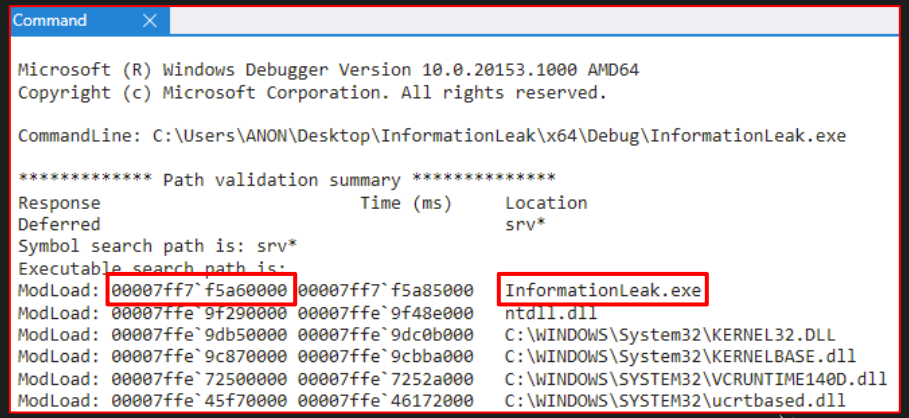
- Indexing the array at an unexpected location results in a 64-bit address being disclosed



- This is great, but this address alone is just one address – how can this be leveraged to completely bypass ASLR?

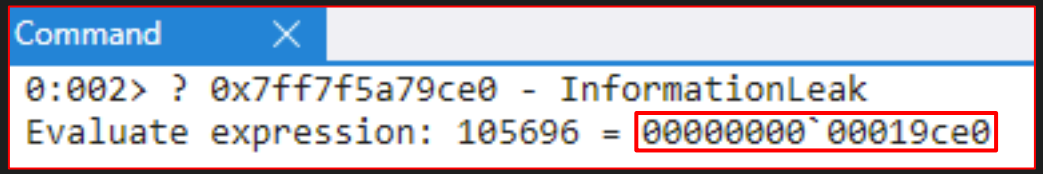
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Process memory is laid out in such a way that although the addresses are randomized, *THE DISTANCE BETWEEN* addresses remain the same
  - Distance between addresses is also known as "the offset"
    - `InformationLeak.exe` loads at **0x7ff7f5a60000**
    - The address leaked was **0x7ff7f5a79ce0**
  - Subtract leaked address from base



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- The distance between the leaked address and the base of the module is 0x19ce0



- Upon reboot, this will still hold true!
  - This means, if it is possible to leak one address – it is possible to find the base address for the DLL/EXE
  - With the base address in hand, all that is needed is to find the ROP gadgets needed to bypass DEP, identify their offset from the base address, and then it is possible to dynamically resolve the needed ROP gadgets to bypass DEP!

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Kernel ASLR (kASLR)
  - Same purpose as user mode ASLR
  - Windows API exposes two commonly abused functions, `EnumDeviceDrivers` and `NtQuerySystemInformation`, that can retrieve the base load address of all drivers on the system from user mode – bypassing kASLR
    - There are some caveats
      - Process executing these functions must be that of medium integrity
      - Must have local access to the system – otherwise need an information leak
  - Only effective against remote exploitation

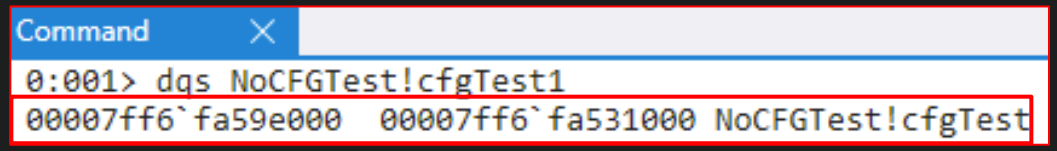
```
BOOL EnumDeviceDrivers(
  LPVOID  *lpImageBase,
  DWORD   cb,
  LPDWORD lpcbNeeded
);
```


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- To bypass DEP/ASLR on modern systems, you need a write vulnerability and a read vulnerability
  - We are up to two vulnerabilities needed for exploitation currently
  - From a "local remote" exploitation perspective, such as a browser exploit, we will need additional vulnerabilities and primitives
    - Can you start to see how exploitation becomes more difficult on modern systems?

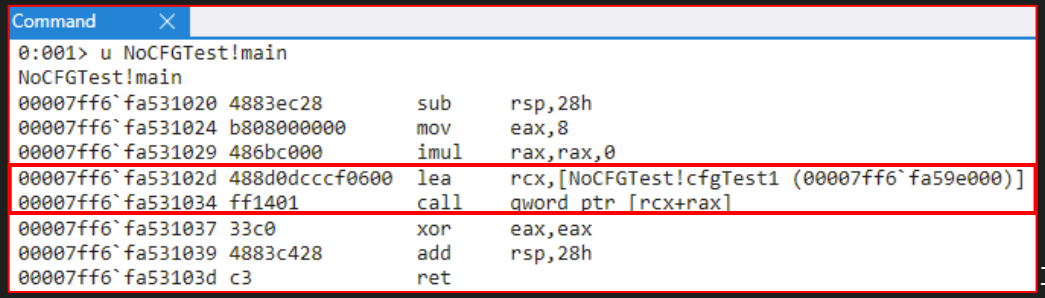
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Control Flow Guard (CFG)
  - Supported in user mode and kernel mode
  - Most binary exploitation requires overwriting a function pointer and then executing that function pointer to obtain code execution
    - The `cfgTest1` function points to the `cfgTest` function



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- The `main` function loads the `cfgTest1` function and executes it (which will actually execute `cfgTest` function, because `cfgTest1` is a function pointer to `cfgTest`



- `fgTest` in this case), with the address of user controlled memory, execution of `cfgTest1` will execute the attacker's code instead

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- With CFG enabled, it is no longer possible to overwrite a function pointer with a "non-valid" target
  - Indirect function call targets are "checked" by CFG
  - Targets are managed by a "bitmap" – to reduce overhead of lookups
  - Valid targets are all of the functions loaded within a process
    - This includes all functions that make up the program and native Windows DLLs loaded within the process space (`NTDLL.dll`, `KERNEL32.dll`, etc.)
  - `dumpbin` can be used to view "valid" targets

```
    Guard CF Function Table

          Address
          --------
           0000000140001000  cfgTest
```


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Function calls without vs. function calls with CFG
  - Indirect function calls are passed to a CFG dispatch function to determine if the targeted address is a "valid function" – an invalid target results in an access violation

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near
sub     rsp, 28h
mov     eax, 8
imul    rax, 0
lea     rcx, cfgTest1
call    qword ptr [rcx+rax]
xor     eax, eax
add     rsp, 28h
retn
main endp
```

```
; Attributes: noreturn

; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_18= qword ptr -18h

sub     rsp, 38h
mov     eax, 8
imul    rax, 0
lea     rcx, cfgTest1
mov     rax, [rcx+rax]
mov     [rsp+38h+var_18], rax
mov     rax, [rsp+38h+var_18]
call    cs:__guard_dispatch_icall_fptr
main endp
```


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

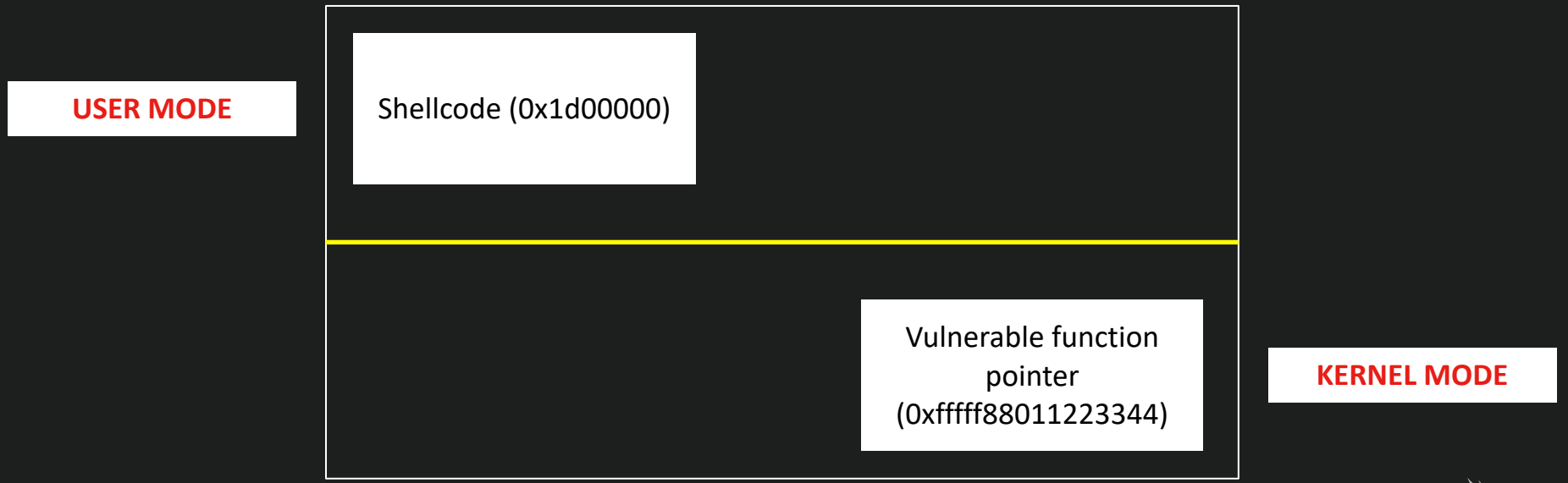
- Bypassing CFG?
  - Overwrite a function pointer with another "valid target"
    - Need to find a way to control function arguments
  - Overwrite a return address on the stack
    - When a `call` instruction is performed, a return address is pushed onto the stack
    - CFG only inspects indirect function calls (forward edge cases) – not returns (backwards edge cases)
      - If an adversary can overwrite a return address for a function call, when the function call returns – it will return to the "fake return address"
      - Need to leak the stack location

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Supervisor Mode Execution Prevention (SMEP)
  - Specifically for local kernel mode exploitation – hardware based

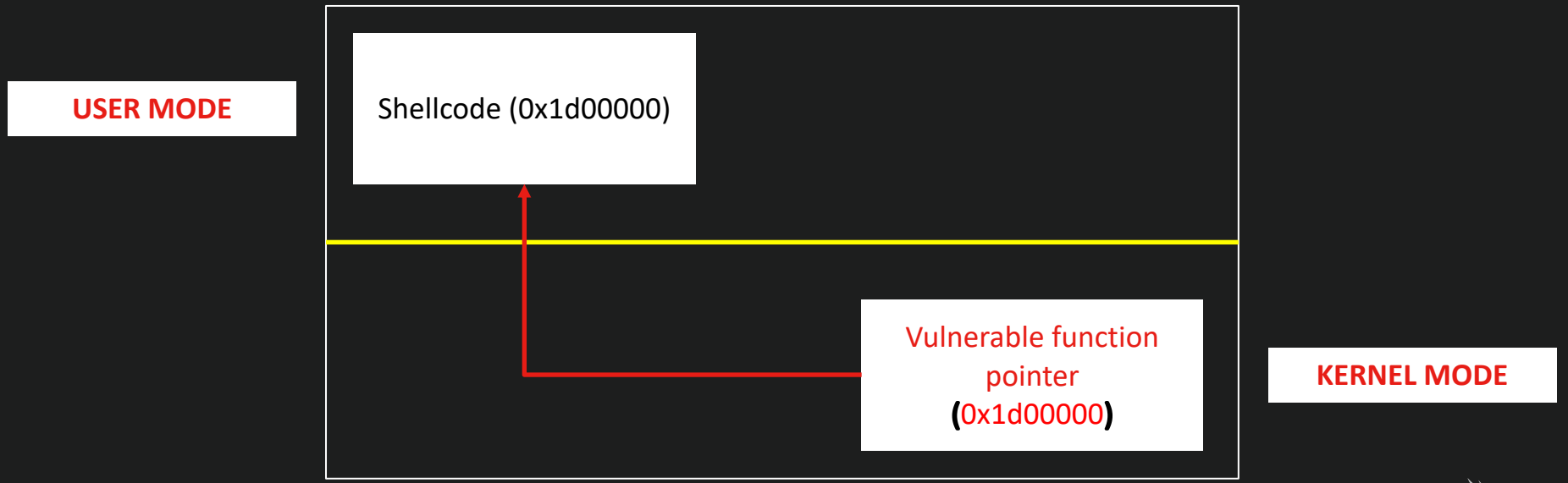
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Typical local kernel mode exploit execution



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Typical local kernel mode exploit execution

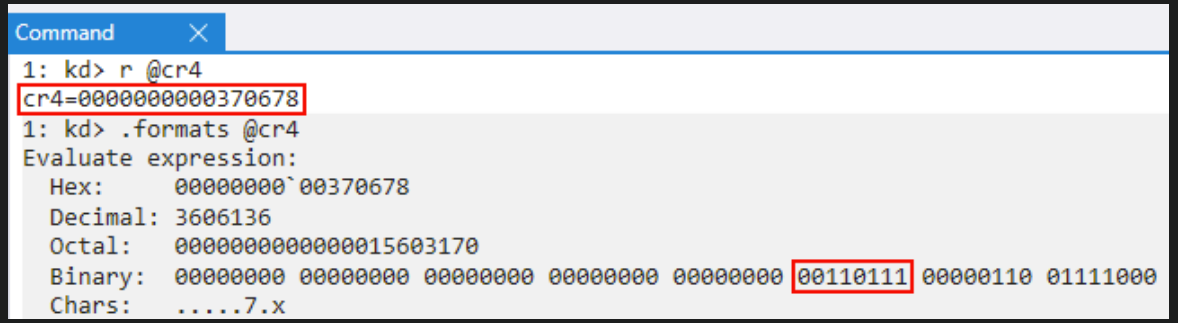


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Why is this method preferred?
  - DEP is enabled in the kernel – meaning DEP would need to be bypassed
  - Since the adversary has local access to the machine, it is more feasible to allocate the shellcode in user mode with an API like `VirtualAlloc` – setting the permissions to `RWX`
    - Calling the shellcode from kernel mode executes the shellcode in context of the kernel

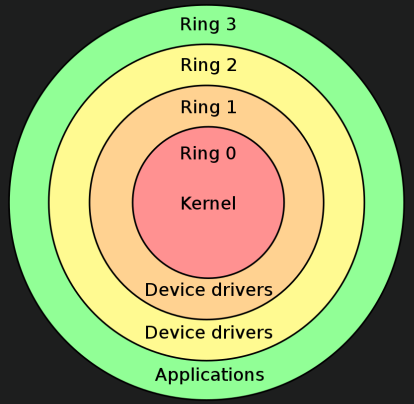
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- SMEP mitigates this by disallowing execution of user mode code from the kernel
  - Implemented by a bit in the CR4 control register
    - 20th bit set = SMEP enabled



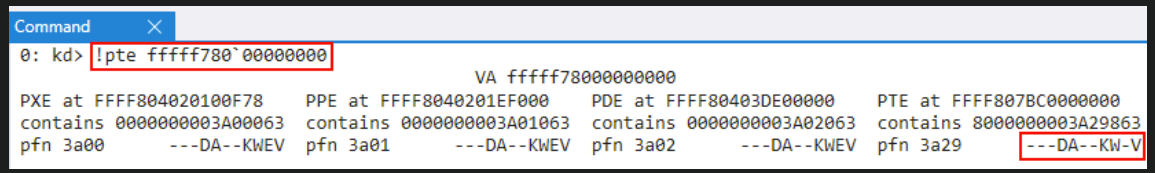
EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- How is this done?
  - x86 based CPUs have an internal state known as the "current privilege level" (CPL)
  - CPL 3 = user mode, CPL 0 = kernel mode – other "rings" aren't used on Windows
    - If CPL 3 code is executed in context of CPL 0 – the system crashes



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

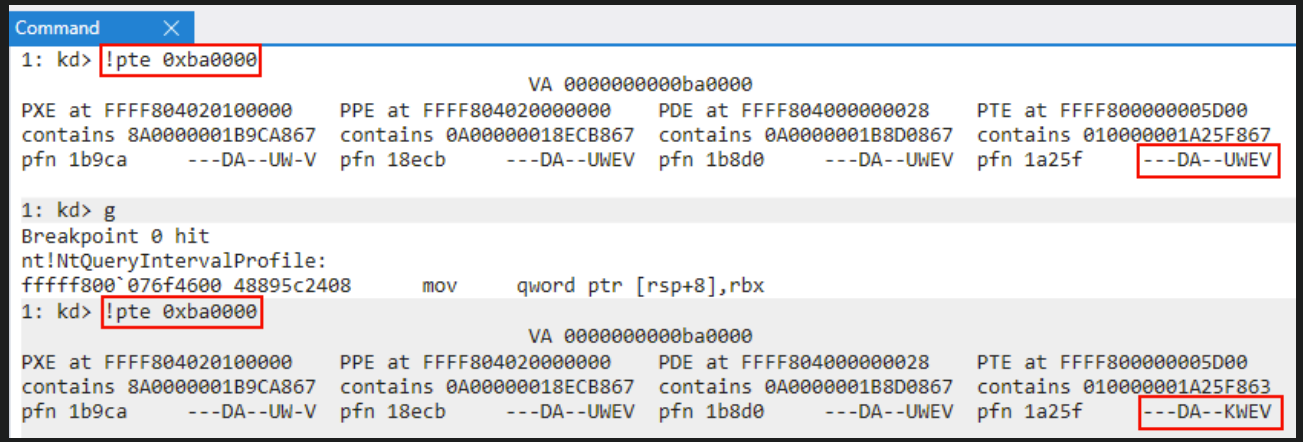
- Bypassing SMEP?
  - SMEP is enabled by the CR4 control register *BUT IT IS ENFORCED* by a page table entry



  - Page table entries contain a bit, known as the `U/S` bit
    - U = User mode page
    - S = Supervisor mode (kernel mode) page
  - It is possible to utilize an information leak vulnerability to locate the page table entry for a memory address and corrupt the `U/S` bit
    - Morten Schenk – DEF CON 25 – Taking Windows 10 Kernel Exploitation To The Next Level

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- To bypass SMEP, locate the page table entry for your user mode shellcode page and corrupt the `U/S` bit, making it a "kernel mode page" – even if the page resides in the user mode address space

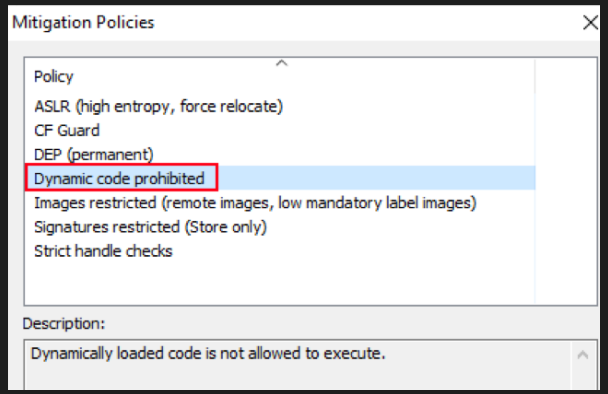


EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Arbitrary Code Guard (ACG)
  - User mode mitigation
    - Hypervisor-Protected Code Integrity (HVCI) is essentially the kernel mode version of ACG
  - "DEP" on steroids essentially
  - With DEP, we could utilize ROP to dynamically change permissions of memory, so that we could execute code inside of a data segment (e.g. the stack)
  - ACG makes this impossible

EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- ACG makes memory "immutable" e.g. it cannot change
  - Code pages are read/execute and cannot be changed
  - Data pages are read/write and cannot be changed
  - On top of that, new code cannot be generated that is read/write/execute
    - From an adversarial perspective, we cannot use a technique like ROP to make existing memory read/write/execute – and we cannot allocate new memory to do the same thing



EXPLOIT MITIGATIONS AND BYPASSING THEM (CONT'D)

- Bypassing ACG?
  - Not a lot in terms of public documentation
  - Sticking to data only attacks
    - We can still use ROP gadgets to manipulate data – we just cannot execute code
    - Could still corrupt data structures, etc.
    - Build the payload entirely in ROP
  - JIT engines are a potential target
    - JIT (just-in-time compilers) are used to convert bytecode into machine code at runtime
      - By default, JIT engines constantly create dynamically executable code

CROWDSTRIKE

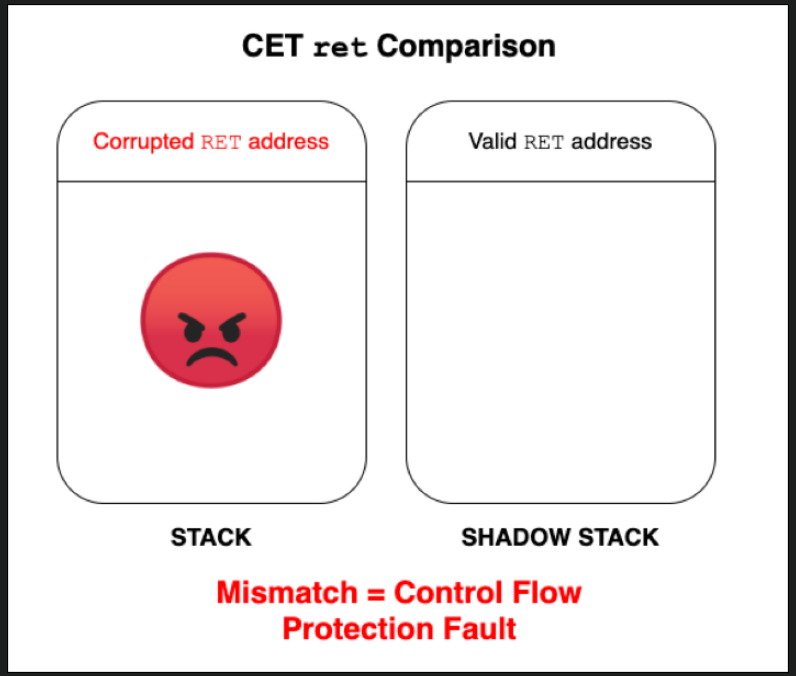
THE FUTURE OF EXPLOITATION

Additional Windows mitigations that are not mainstream (yet)

THE FUTURE OF EXPLOITATION

- Intel Control-Enforcement Technology (CET)
  - Hardware-based control flow integrity
    - CFG only protects forward-edge cases (e.g. calls)
      - CET most notably protects return addresses with a “shadow stack” (backwards-edge case)
  - CET also provides forward-edge protect (Indirect Branch Tracking – IBT)
    - Microsoft doesn’t use IBT due to CFG

THE FUTURE OF EXPLOITATION (CONT'D)



THE FUTURE OF EXPLOITATION

- Bypassing CET?
  - Not much is yet known
  - Need proper hardware to test
  - Only time will tell

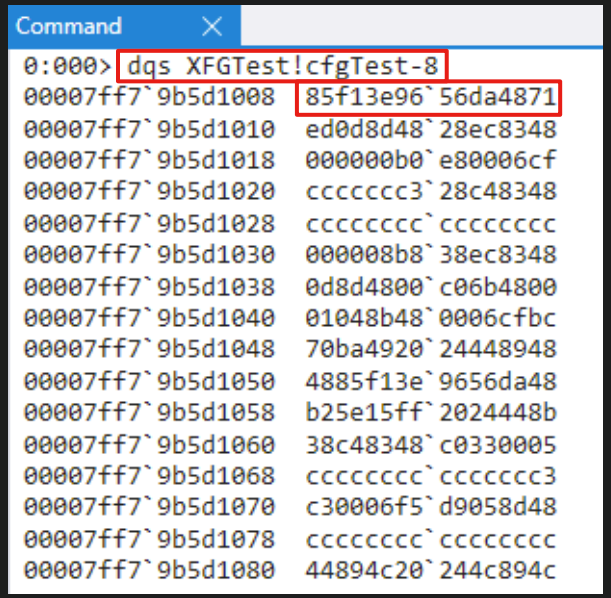
THE FUTURE OF EXPLOITATION (CONT'D)

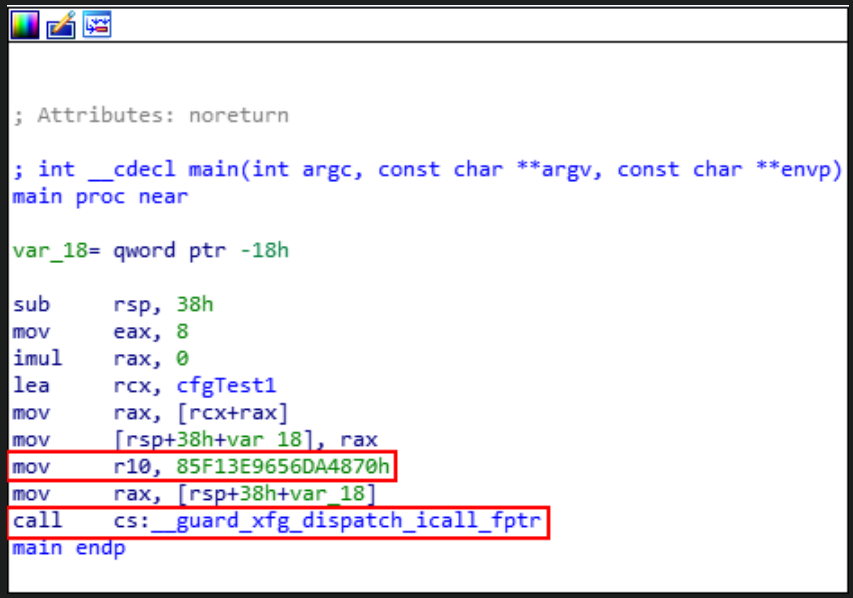
- Xtended Flow Guard (XFG)
  - "Enhanced" version of CFG
  - As we saw with normal CFG, it is still possible to call functions other than the intended protected function

THE FUTURE OF EXPLOITATION (CONT'D)

- XFG Takes into account the data type, parameters, and return values of a protected function
  - This is represented by a ~55-bit hash – which is placed 8 bytes above each protected function
  - When XFG checks a function, it takes the 8 byte hash mentioned above, stores it in the R10 register and uses it as an additional control-flow transfer check
    - The hash 8 bytes above the function is compared against the hash stored in R10

THE FUTURE OF EXPLOITATION (CONT'D)





THE FUTURE OF EXPLOITATION (CONT'D)

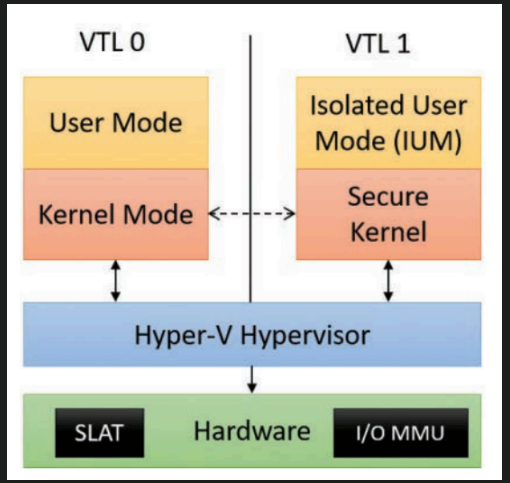
- If these hashes do not match, the process crashes
- Instead of limiting function calls to calls within the CFG bitmap, calls now have to be within the CFG bitmap AND of the same data type, have the same amount of parameters, and return the same type of value
  - Greatly decreases the amount of functions that can be called tremendously

THE FUTURE OF EXPLOITATION (CONT'D)

- Bypassing XFG?
  - Return addresses are still a target
  - Calling an identical function
  - By a stroke of luck, the 8 bytes above the function you are trying to call *MAYBE* will be the same as the XFG hash
    - Very, very, very ,very unlikely – but not impossible
  - Leveraging an arbitrary read
    - Dereference the entire process memory to search for 8 bytes that are *IDENTICAL* to the in scope XFG hash and call whatever is 8 bytes below that
      - Will most likely not be very fruitful, but still is possible

THE FUTURE OF EXPLOITATION (CONT'D)

- Virtualization-Based Security (VBS)/Hypervisor-Protected Code Integrity (HVCI)
  - Very powerful mitigations
  - VBS manages resources on Windows from the hypervisor
    - Done through "trust" levels



THE FUTURE OF EXPLOITATION (CONT'D)

- HVCI
  - Enables immutable bits (in context of VTL 0) that set VTL 1 permissions on VTL 0 pages
    - This mitigates the ability to manipulate page table entries to bypass SMEP or DEP in the kernel
  - Additionally doesn't allow new, `RWX` memory in the kernel
    - Essentially is ACG in kernel mode

THE FUTURE OF EXPLOITATION (CONT'D)

- Bypassing VBS/HVCI?
  - Still more unknowns
  - Data only attacks
  - Hypervisor exploitation

IS VULNERABILITY RESEARCH STILL PRACTICAL FOR RED TEAMS/NON-RESEARCH FIRMS?

As an optimist – I think so :-)

THE FUTURE OF EXPLOITATION (CONT'D)

- Modern exploits, in order to be portable and reliable, will probably check in at about 4 separate vulnerabilities within the exploit chain
  - Arbitrary write in user mode
  - Arbitrary write in kernel mode
  - Arbitrary read in user mode
  - Arbitrary read in kernel mode
- Definitely more difficult than 2000
  - But does anything top a non-authenticated remote kernel exploit that works across all major versions of Windows?
- I think it is always worth staying curious and keeping binary exploitation skill sharp – as there is crossover with malware development and other red team categories

THE FUTURE OF EXPLOITATION (CONT'D)

- Thank you for attending my talk!
- Thank you BSides Dayton!
- I will see you all in the Q&A!