What has my compiler done for me lately?

Segmentation fault (core dumped)

Agenda

1. Actual shellcode attacks.
2. Stack/Heap based exploits
   a. W^X memory.
3. Return Oriented Programming (ROP)
   a. Stack Cookies
   b. Shadow Call Stack
4. Indirect control flow & c++ vtable attacks.
   a. CFI

Shellcode Attacks

- Shellcode is native (byte) code.
- The encoded instructions that are interpreted by the CPU.

Shellcode Attacks

- Shellcode is native (byte) code.
- The encoded instructions that are interpreted by the CPU.

```
48 89 ec        ; movq  %rsp, %rbb
c3              ; ret
90              ; nop
```

Shellcode Attacks

- Shellcode is native (byte) code.
- The encoded instructions that are interpreted by the CPU.

```
48 89 ec        ; movq  %rsp, %rbb
c3              ; ret
90              ; nop
```

- Managed code (Javascript):
  - Sandboxed. Can only access very specific things.
  - All interaction managed by interpreter.
- Shellcode:
  - Full access to the system. Run directly on hardware.
- Two different attack types:

Shellcode Attack Types

- Remote attacks?

Shellcode Attack Types

- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?

Shellcode Attack Types

- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?
        - Get shellcode running on the machine.
    - Targets?

Shellcode Attack Types

- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?
        - Get native code (shellcode/bytecode) running on the machine.
    - Targets?
        - Browsers, network drivers, video games.
- Local attacks?

Shellcode Attack Types

- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?
        - Get native code (shellcode/bytecode) running on the machine.
    - Targets?
        - Browsers, network drivers, video games.
- Local attacks.
    - We have shellcode access on the machine.
    - Goal?

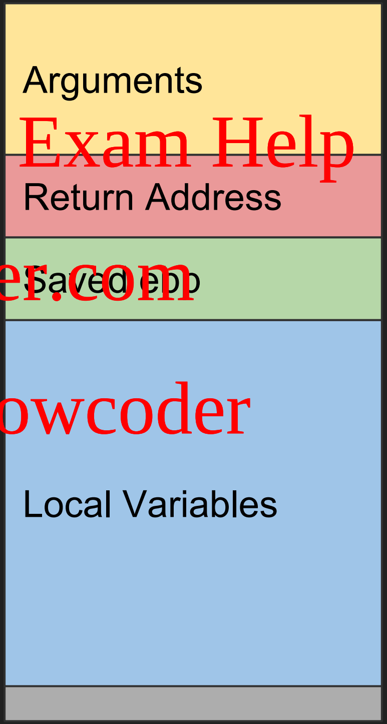
Shellcode Attack Types

- Remote attacks.
  - We have no ability to run unmanaged code.
  - Goal?
    - Get native code (shellcode/bytecode) running on the machine.
  - Targets?
    - Browsers, network drivers, video games.
- Local attacks.
  - We have shellcode access on the machine.
  - Goal?
    - Privilege escalation.
  - Targets?

Shellcode Attack Types

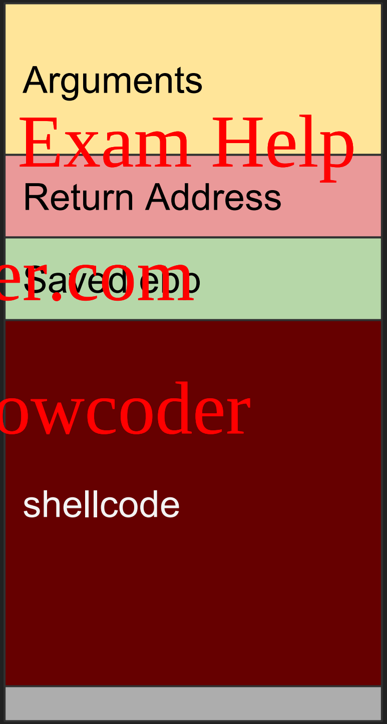
- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?
        - Get native code (shellcode/bytecode) running on the machine.
    - Targets?
        - Browsers, network drivers, video games.
- Local attacks.
    - We have shellcode access on the machine.
    - Goal?
        - Privilege escalation.
    - Targets?
        - Kernel, hypervisor, any process running as a different user/group.

Stack based exploits



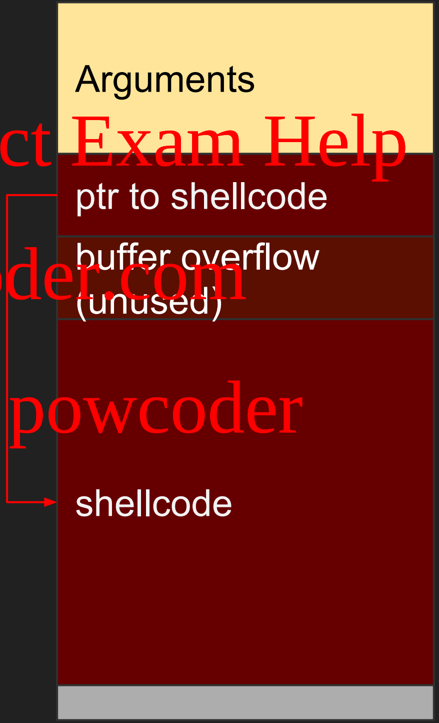
Stack based exploits

1. Overwrite the local variables
   a. Buffer overflow
   b. Use after return
   c. Use after scope



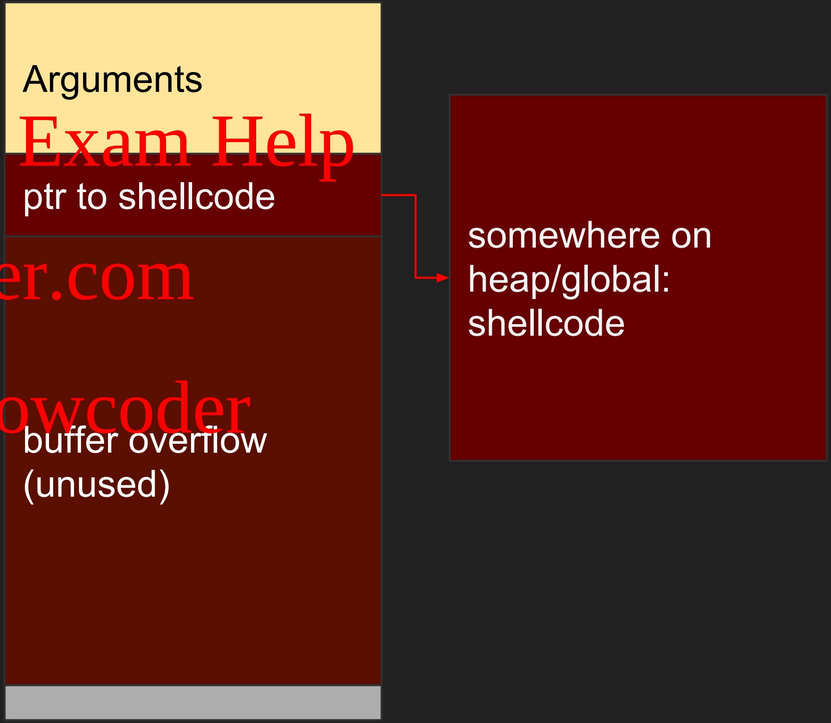
Stack based exploits

1. Overwrite the local variables
   a. Buffer overflow
   b. Use after return
   c. Use after scope
2. Overwrite the return address



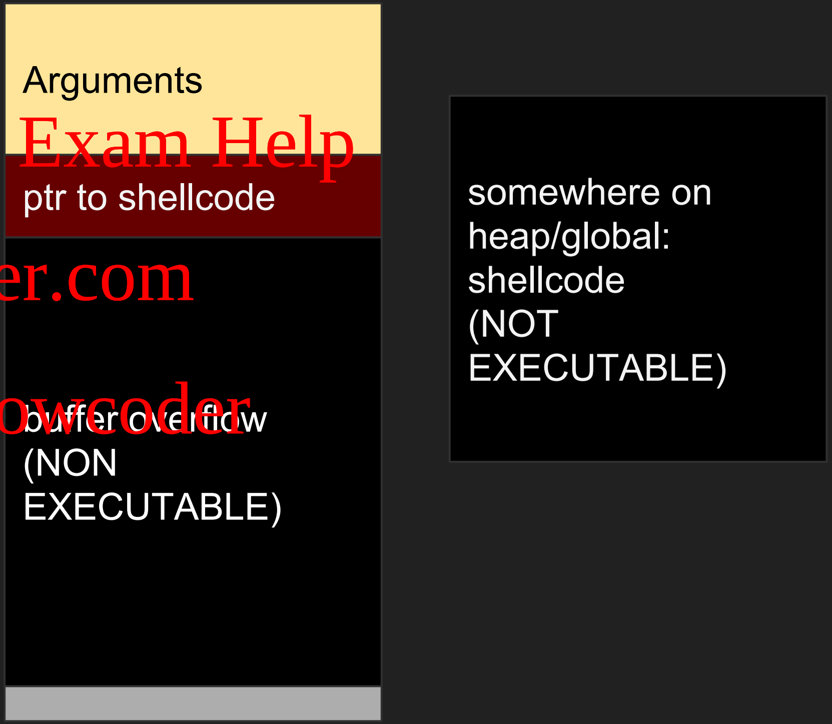
Heap based exploits

1. Add shellcode to heap:
   a. Heap buffer overflow
   b. Use after free
   c. Global buffer overflow
   d. Initialization order bugs
2. Overwrite the return address



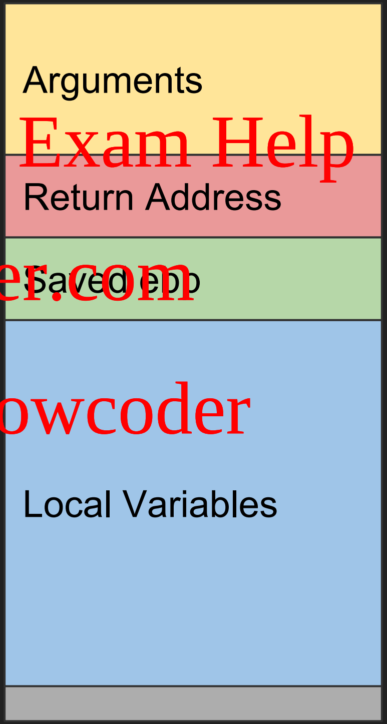
W^X Memory

- Write XOR Execute
- Can still write shellcode.
- Can't execute shellcode.
- Compiler sets sections (.data, .bss, stack, heap, etc) metadata to be no-execute.
- Done by default on all compilers.
- Problem solved, right…?
  - Chrome still RWX v8 pages.


Arguments
ptr to shellcode
buffer overflow (NON EXECUTABLE)
somewhere on heap/global: shellcode (NOT EXECUTABLE)

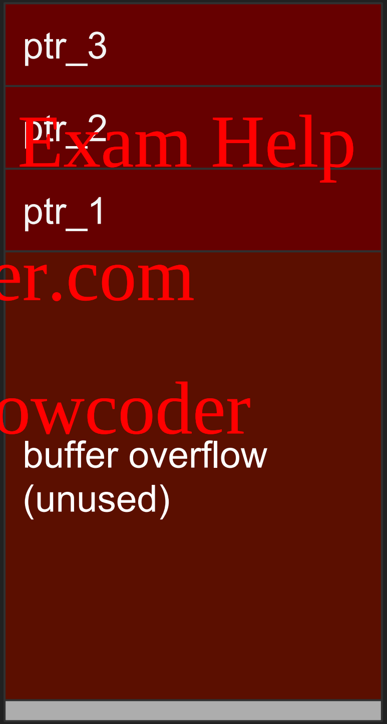
Return Oriented Programming

- Chains together "gadgets", which is a sequence of a few instructions followed by 'ret'.
- Smash stack with lots of addresses to gadgets.



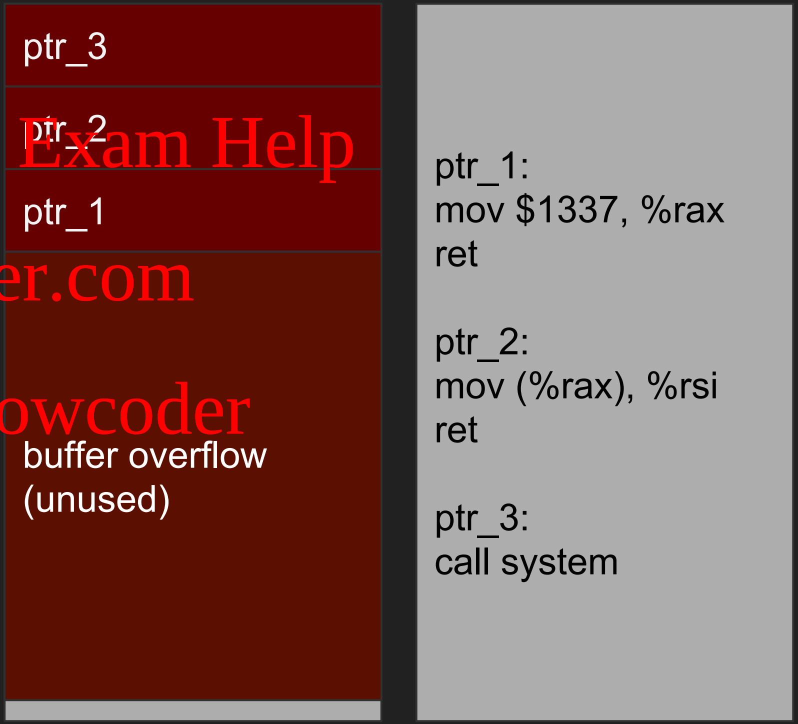
Return Oriented Programming

- Chains together “gadgets”, which is a sequence of a few instructions followed by ‘ret’.
- Smash stack with lots of addresses to gadgets.
- Choose gadgets to execute shellcode we want.



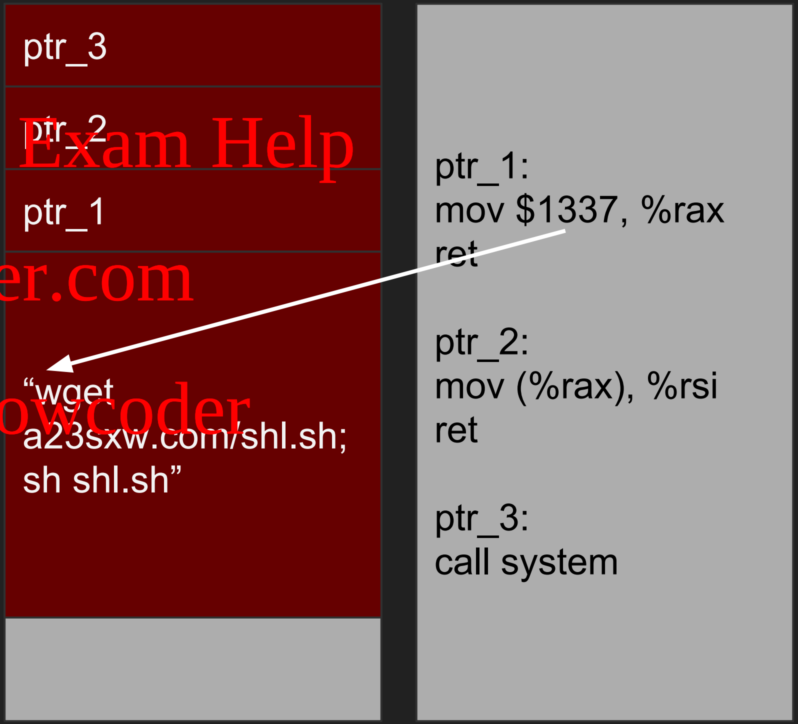
Return Oriented Programming

- Chains together “gadgets”, which is a sequence of a few instructions followed by ‘ret’.
- Smash stack with lots of addresses to gadgets.
- Choose gadgets to execute shellcode we want.
- If clever, we hide strings in other places we have write access to. Use the strings in the exploit.



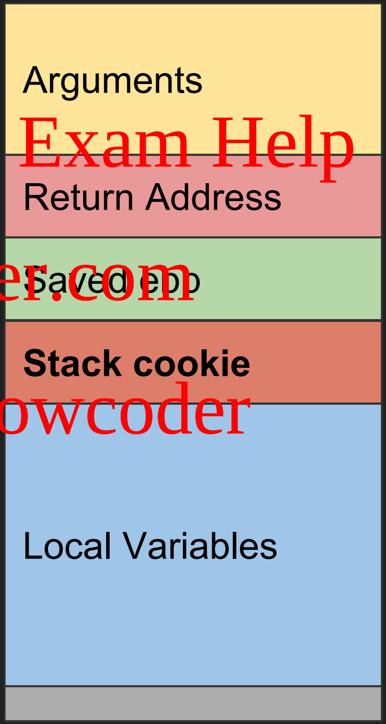
Return Oriented Programming

- Chains together "gadgets", which is a sequence of a few instructions followed by 'ret'.
- Smash stack with lots of addresses to gadgets.
- Choose gadgets to execute shellcode we want.
- If clever, we hide strings in other places we have write access to. Use the strings in the exploit.



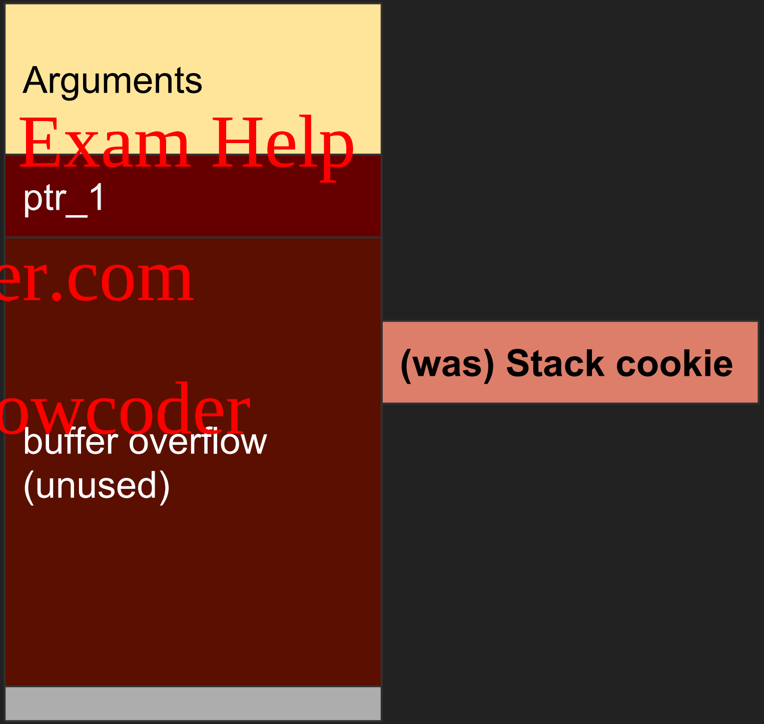
Stack Cookies

- -fstack-protector
- Adds “cookie” or “canary” to stack on function entry.
- Checks it on function exit.
- If the cookie fails, kill the program.



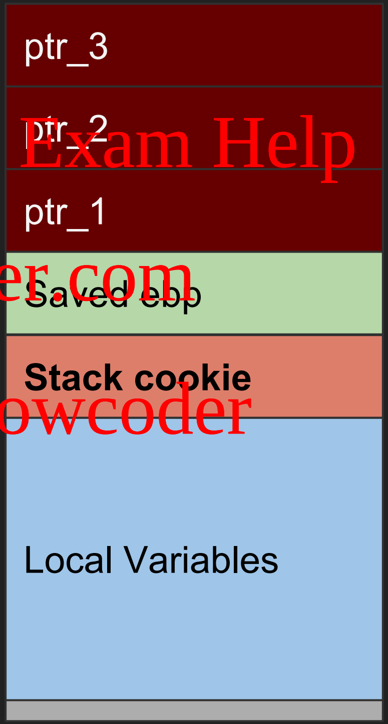
Stack Cookies

```
movq %fs:40, %rax       ; grab cookie
movq %rax, -8(%rbp)     ; save to stack
xorl %rax, %rax         ; hide the cookie

<normal function code>

movq -8(%rbp), %rax     ; get from stack
xorq %fs:40, %rax       ; compare
jnz __stack_chk_fail
ret
```



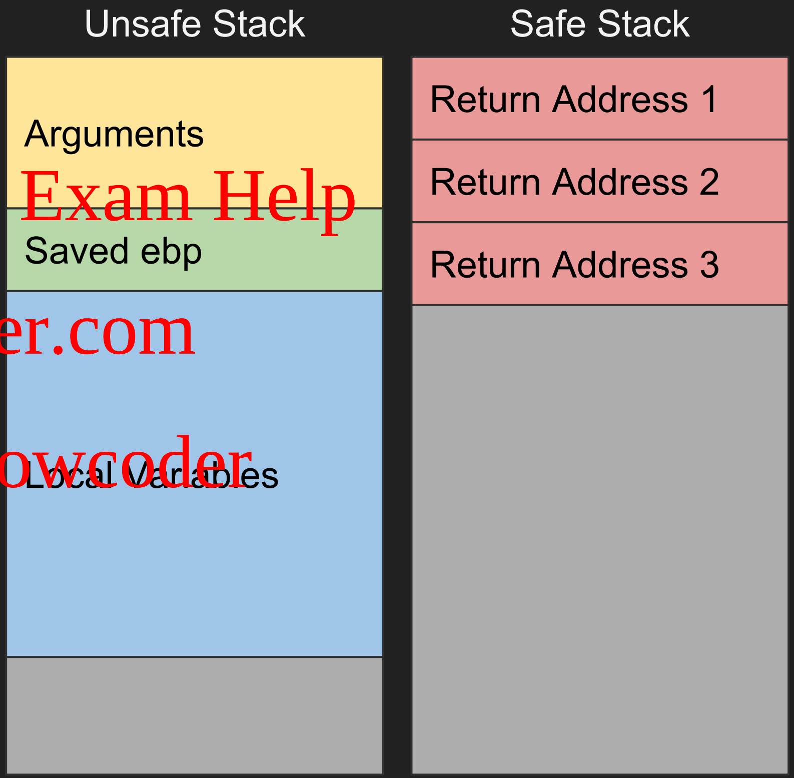
Stack Cookies

1. Overflowing the return address must write over stack cookie.
2. Cookie is hidden outside of normal memory.
3. 5.4210 * 10^(-20) chance of guessing correct cookie.
4. Stops sequential write bugs, what about arbitrary write?



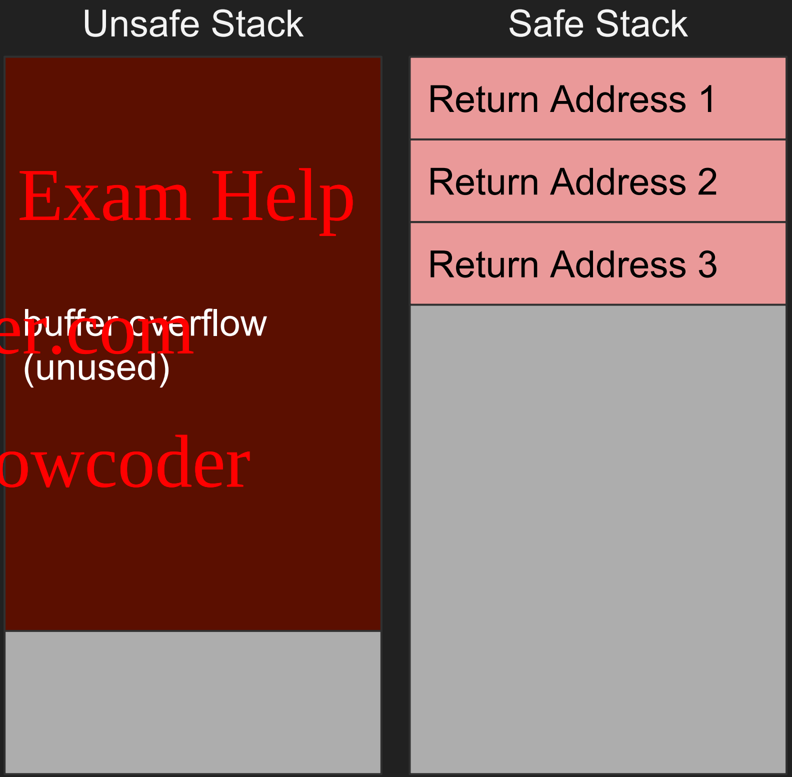
Shadow Call Stack

- -fsanitize=shadow-call-stack
- Another ROP defense
- Separate stacks into *safe* and *unsafe*.
- Safe contains return pointers.
- Unsafe contains everything else.



Shadow Call Stack

- Arbitrary writes are still safe as long as safe stack pointer is secret.
- Safe stack pointer is hidden:
  - Reserved register (x18) on ARM.
  - Reserved segment (%gs) on x86.



TOCTOU Vulnerabilities

- Time of Check to Time of Use (TOCTOU)
- Thread-based security race between call and prologue finishing.
- Prologue #3, save return address into register at top of function entry.
- Requires stack location disclosure.

ShadowCallStack Prologue:

1. Normal function entry.
2. Allocate space on shadow stack.
3. Add ptr to shadow stack.

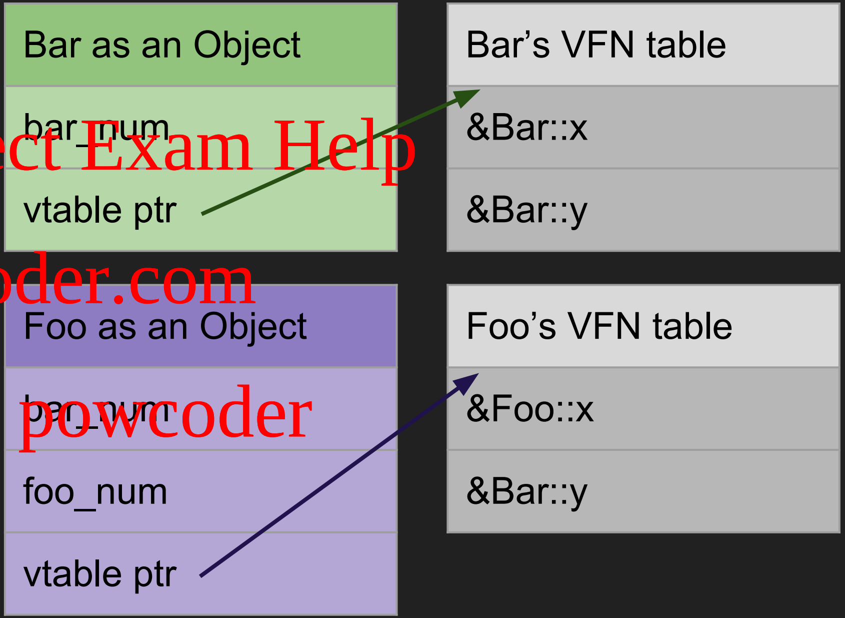
ShadowCallStack Epilogue:

1. Pop from shadow stack.
2. Compare to return address off regular stack.
3. Fail if return address has been compromised.

C++ Virtual Function Tables

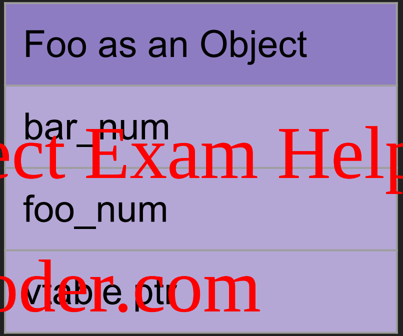
- Used to implement polymorphism.

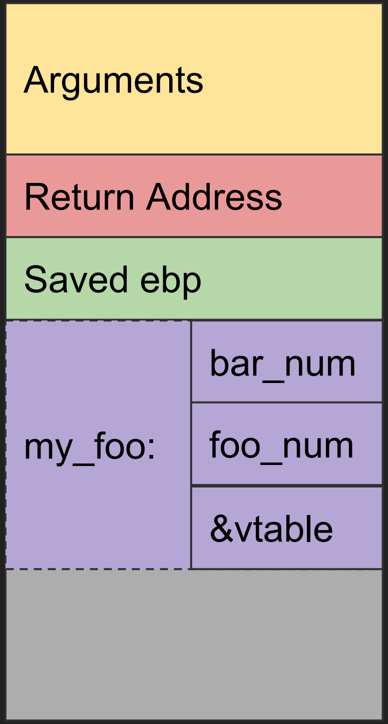
```
struct Bar {
	virtual void x() { … }
	virtual void y() { … }
	int bar_num = 0;
}

class Foo : Bar {
	void x() override { … }
	int foo_num = 0;
}
```



C++ Virtual Function Tables

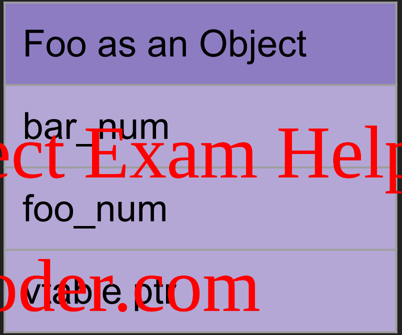
```
int x() {
        Foo my_foo;
        foo.x();
}
```

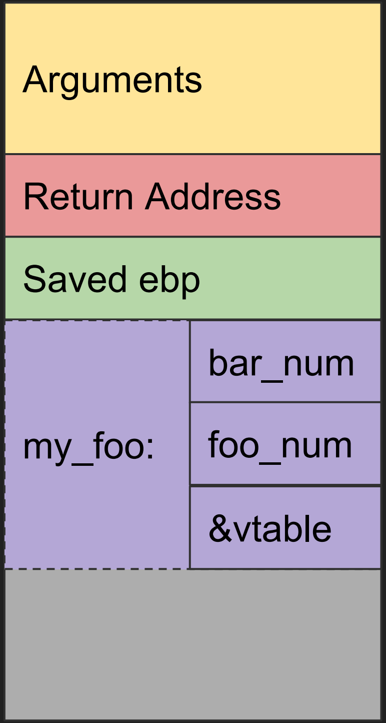




C++ Virtual Function Tables

```
int x() {
        Foo my_foo;
        foo.x();
}

x:
<create my_foo on stack>
movq -8(%rsp), %rdi    ; get vtable ptr
movq $0, %rsi          ; index of &Foo:x
movq %rsi(%rdi), %rdi  ; get &Foo:x
callq *(%rdi)          ; call &Foo:x
```

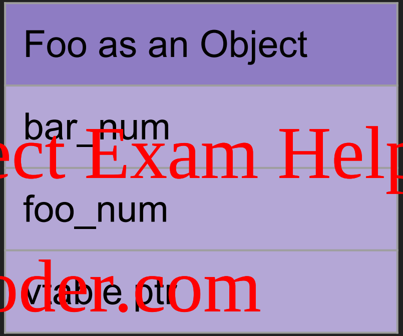


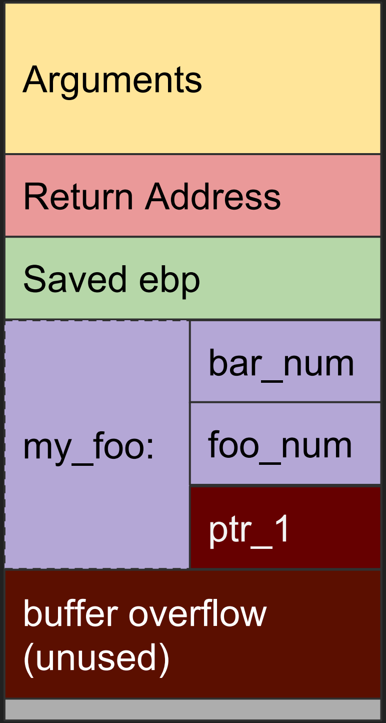


C++ Virtual Function Tables

- Add some dangerous code and...

```
int x() {
        Foo my_foo;
        char buffer[255];
        fgets("%s", buffer);
        foo.x();
}
```





Control Flow Integrity (CFI)

- -fsanitize=cfi
- Adds checks to ensure correct vtable before call.
- Stops smashing vtable pointers on stack/heap.
- Kills the program on sanity check failures.

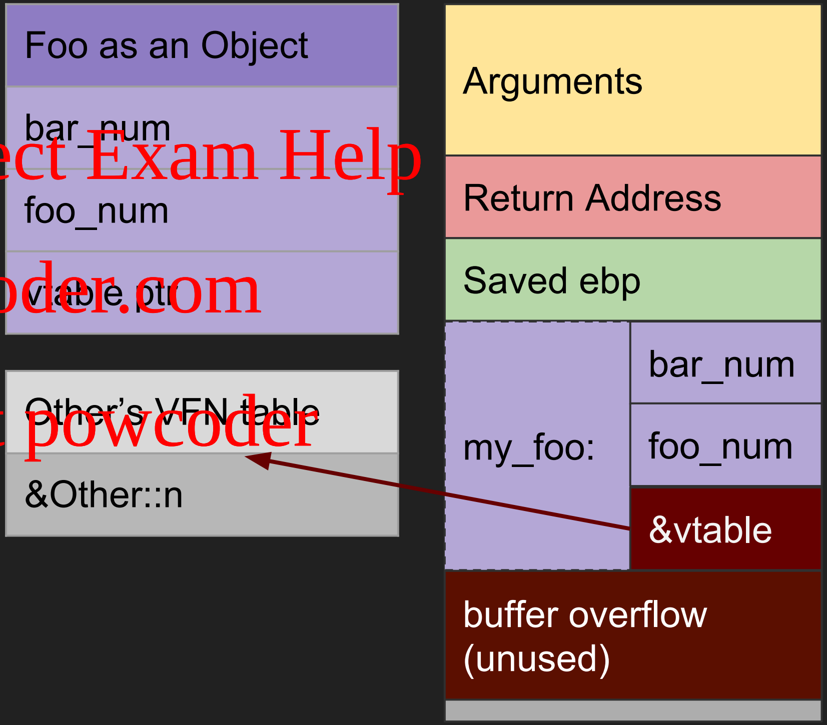
```
movq -8(%rsp), %rsi    ; get vtable ptr
movq $0, %rsi          ; index of &Foo:x
call cfi_check
movq (%rsi), %rsi      ; get &Foo:x
callq *(%rsi)          ; call &Foo:x

cfi_check:
; ensure table is in range and aligned
; ensure index (%rsi) is valid
```

CFI cont.

- More advanced attack: Run a different virtual function from a different class.

```
class Other {
    virtual void n();
}
```

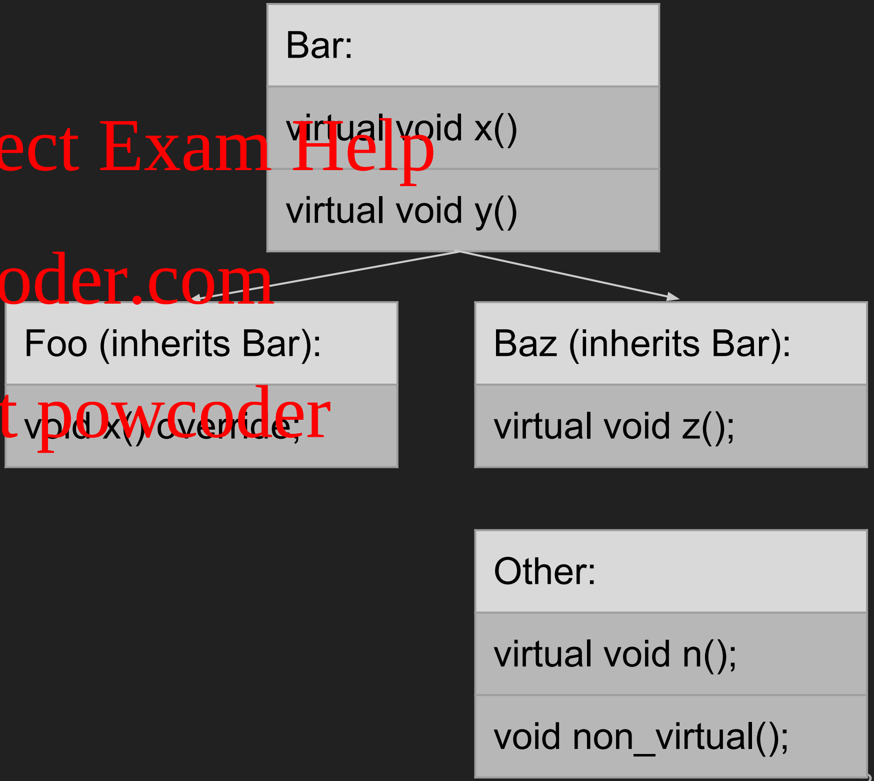
- CFI protects against these as well.



CFI Cast Checking

- CFI adds checks to all types of cast checking.

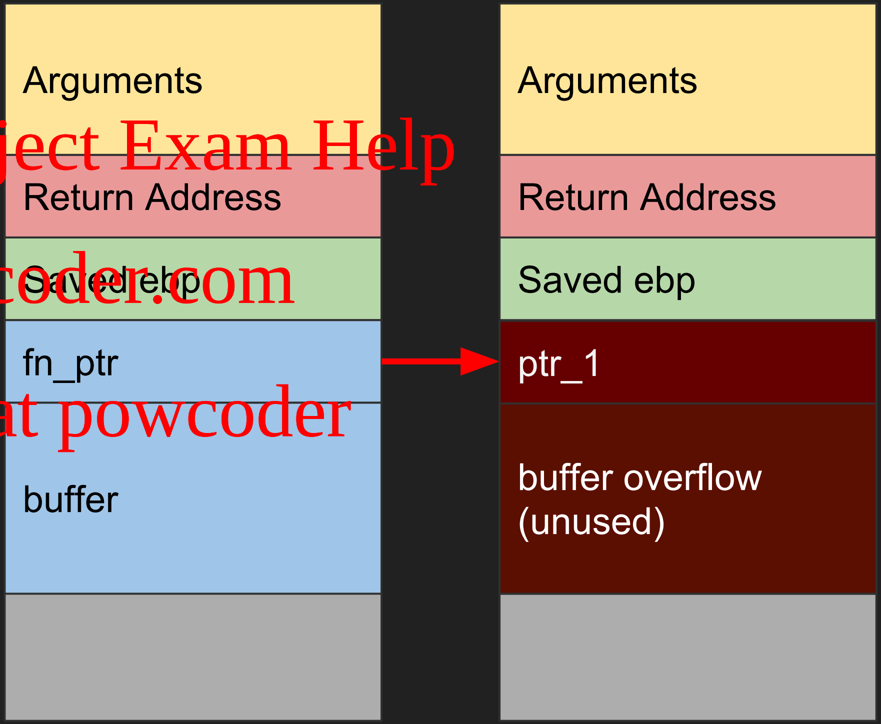
```
Baz* b1 = new Foo();          // wrong
Baz* b2 = new Bar();          // wrong
Bar* b3 = new Foo();          // OK
Baz* b4 = b3;                 // wrong
(Other* b3)->non_virtual(); // wrong
Bar* b6 = new Other();        // wrong
void* o = new Other();        // OK
(Bar* o)->y();                // wrong
```



CFI Indirect Call

- Also protects against similar tomfoolery with indirect function calls.

```
void my_function() { … }

int main() {
        void (*fn_ptr)() = &my_function;
        char buffer[255];
        fgets("%s", buffer);
        fn_ptr();
}
```

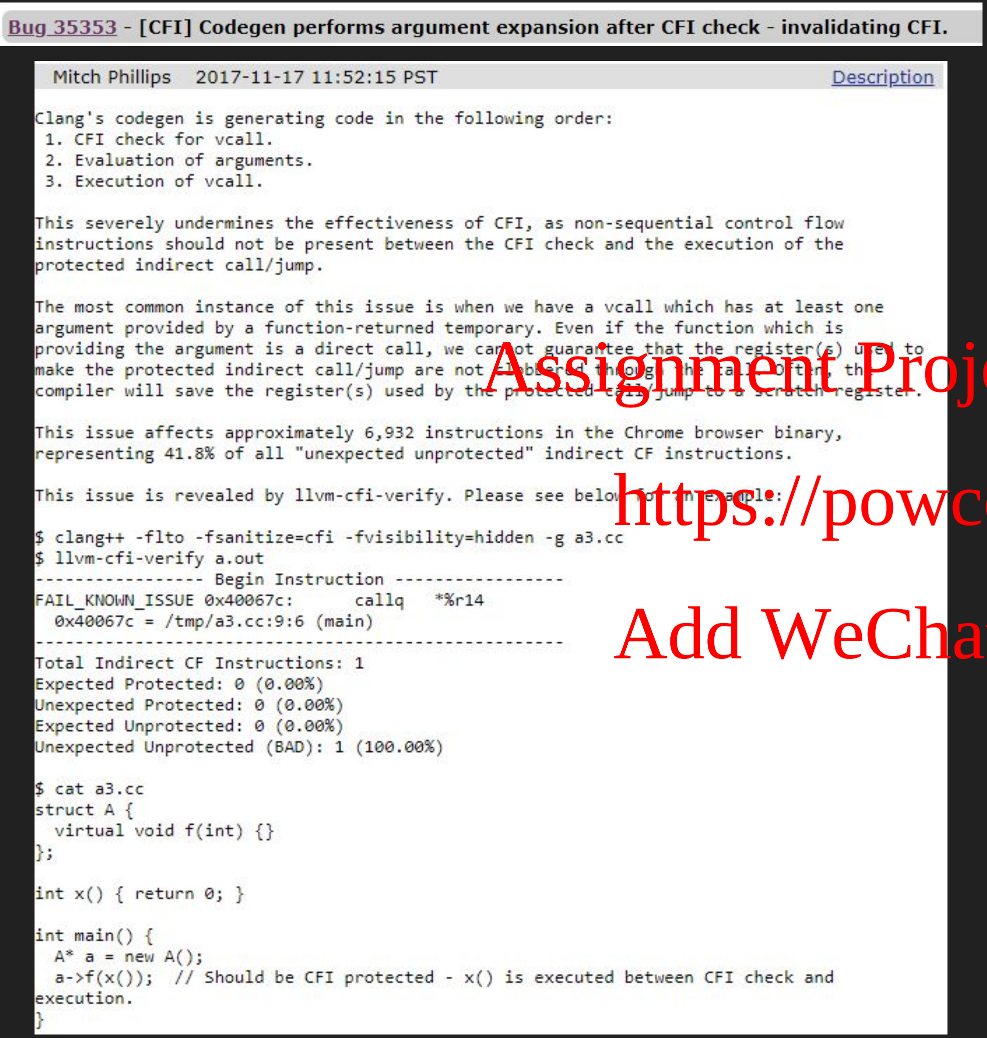


CFI Issues

- Only forward-edge protection.
  - rCFI is implemented, but very expensive and requires significant metadata.
- Cross-DSO is computationally expensive.
- Checks can get quite complicated:
  - e.g. Base class vcalls are legal for derived classes

```
Derived::x() {
        return Base::x() + Base::y();
}
```

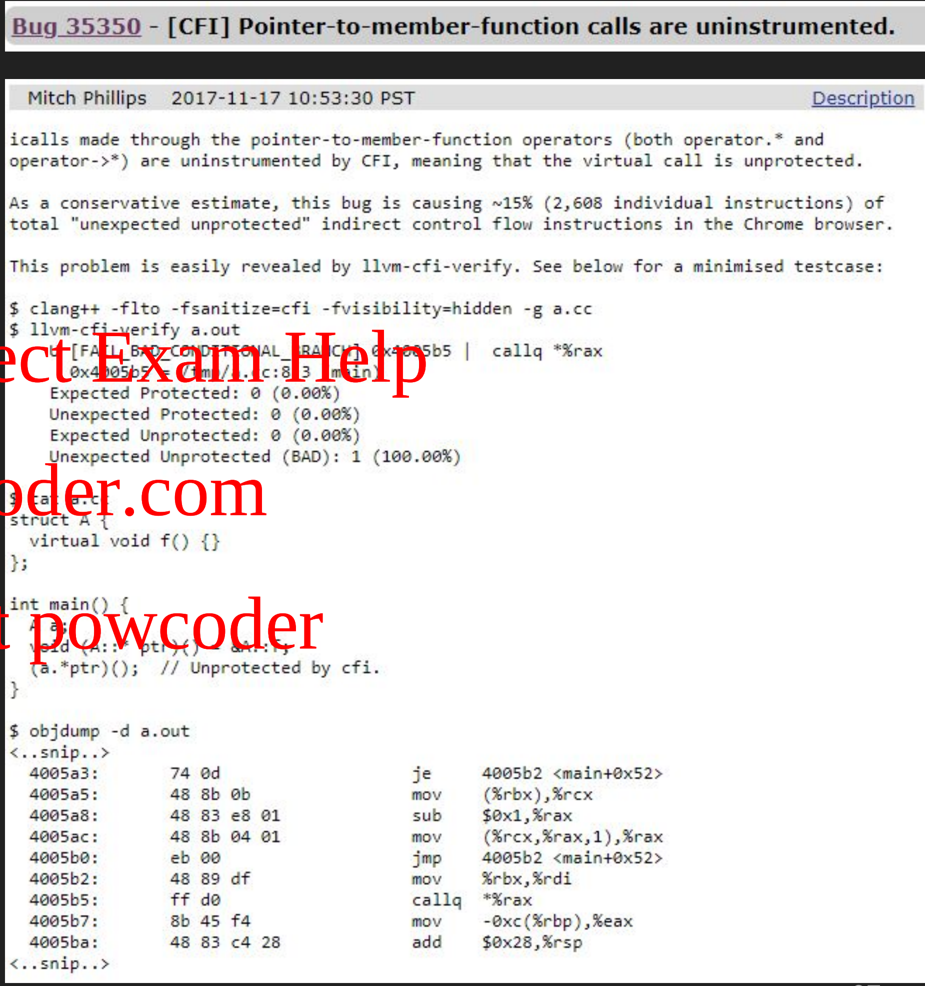
- also...



Bug 35353 - [CFI] Codegen performs argument expansion after CFI check - invalidating CFI.

Mitch Phillips 2017-11-17 11:52:15 PST Description

```
Clang's codegen is generating code in the following order:
 1. CFI check for vcall.
 2. Evaluation of arguments.
 3. Execution of vcall.

This severely undermines the effectiveness of CFI, as non-sequential control flow
instructions should not be present between the CFI check and the execution of the
protected indirect call/jump.

The most common instance of this issue is when we have a vcall which has at least one
argument provided by a function-returned temporary. Even if the function which is
providing the argument is a direct call, we cannot guarantee that the register(s) used to
make the protected indirect call/jump are not clobbered through the call. Often, the
compiler will save the register(s) used by the protected call/jump to a scratch register.

This issue affects approximately 6,932 instructions in the Chrome browser binary,
representing 41.8% of all "unexpected unprotected" indirect CF instructions.

This issue is revealed by llvm-cfi-verify. Please see below for an example:

$ clang++ -flto -fsanitize=cfi -fvisibility=hidden -g a3.cc
$ llvm-cfi-verify a.out
----------------- Begin Instruction -----------------
FAIL_KNOWN_ISSUE 0x40067c:      callq   *%r14
  0x40067c = /tmp/a3.cc:9:6 (main)
-----------------------------------------------------
Total Indirect CF Instructions: 1
Expected Protected: 0 (0.00%)
Unexpected Protected: 0 (0.00%)
Expected Unprotected: 0 (0.00%)
Unexpected Unprotected (BAD): 1 (100.00%)

$ cat a3.cc
struct A {
  virtual void f(int) {}
};

int x() { return 0; }

int main() {
  A* a = new A();
  a->f(x());  // Should be CFI protected - x() is executed between CFI check and
execution.
}
```



Bug 35350 - [CFI] Pointer-to-member-function calls are uninstrumented.

Mitch Phillips 2017-11-17 10:53:30 PST Description

```
icalls made through the pointer-to-member-function operators (both operator.* and
operator->*) are uninstrumented by CFI, meaning that the virtual call is unprotected.

As a conservative estimate, this bug is causing ~15% (2,608 individual instructions) of
total "unexpected unprotected" indirect control flow instructions in the Chrome browser.

This problem is easily revealed by llvm-cfi-verify. See below for a minimised testcase:

$ clang++ -flto -fsanitize=cfi -fvisibility=hidden -g a.cc
$ llvm-cfi-verify a.out
  U [FAIL_BAD_CONDITIONAL_BRANCH] 0x4005b5 |  callq *%rax
    0x4005b5 = /tmp/a.cc:8:3 (main)
    Expected Protected: 0 (0.00%)
    Unexpected Protected: 0 (0.00%)
    Expected Unprotected: 0 (0.00%)
    Unexpected Unprotected (BAD): 1 (100.00%)

$ cat a.cc
struct A {
  virtual void f() {}
};

int main() {
  A a;
  void (A::* ptr)() = &A::f;
  (a.*ptr)();  // Unprotected by cfi.
}

$ objdump -d a.out
<..snip..>
  4005a3:       74 0d                   je     4005b2 <main+0x52>
  4005a5:       48 8b 0b                mov    (%rbx),%rcx
  4005a8:       48 83 e8 01             sub    $0x1,%rax
  4005ac:       48 8b 04 01             mov    (%rcx,%rax,1),%rax
  4005b0:       eb 00                   jmp    4005b2 <main+0x52>
  4005b2:       48 89 df                mov    %rbx,%rdi
  4005b5:       ff d0                   callq  *%rax
  4005b7:       8b 45 f4                mov    -0xc(%rbp),%eax
  4005ba:       48 83 c4 28             add    $0x28,%rsp
<..snip..>
```

Other CFI

- Microsoft Control Flow Guard (CFG)
    - Near-precise. Isn't perfect.
- Intel Control Enforcement Technology (CET)
    - Hardware enforced safe stack.
    - Also ENDBRANCH, landing pad for indirect branches. Near-precise.
- ARM Pointer Authentication Keys.

Final Notes

- Protection mechanisms mentioned **do not fix the underlying bug**.
- Can still deny service by crashing process.
- No protection is holistic.
- Compilers can only help you if you enable them!
  - -fstack-protector (-fstack-protector-all)
  - -fsanitize=cfi
  - -shadow-call-stack
- Compilers are made by humans. Any security critical code should always be inspected by hand.