What has my compiler done for me lately��

unSeegmentat Speech)

Agenda

1. Actual shellcode attacks.
2. Stack/Heap based exploits
   a. W^X memory.
3. Return Oriented Pro
   a. Stack Cookies
   b. Shadow Call Stack
4. Indirect control flow & c++ vtable attack
   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
c3              ; ret
90              ; nop
```

Shellcode Attacks

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

```
48 89 ec        ; movq
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
    - Targets?

Shellcode Attack Types

- Remote attacks.
  - We have no ability to run unmanaged code.
  - Goal?
    - Get native cod
  - Targets?
    - Browsers, net
- Local attacks?

ine.

Shellcode Attack Types

- Remote attacks.
  - We have no ability to run unmanaged code.
  - Goal?
    - Get native cod
  - Targets?
    - Browsers, net
- Local attacks.
  - We have shellcode access on the machine.
  - Goal?

ine.

Shellcode Attack Types

- Remote attacks.
  - We have no ability to run unmanaged code.
  - Goal?
    - Get native cod
  - Targets?
    - Browsers, net
- Local attacks.
  - We have shellcode access on the machine.
  - Goal?
    - Privilege escalation.
  - Targets?

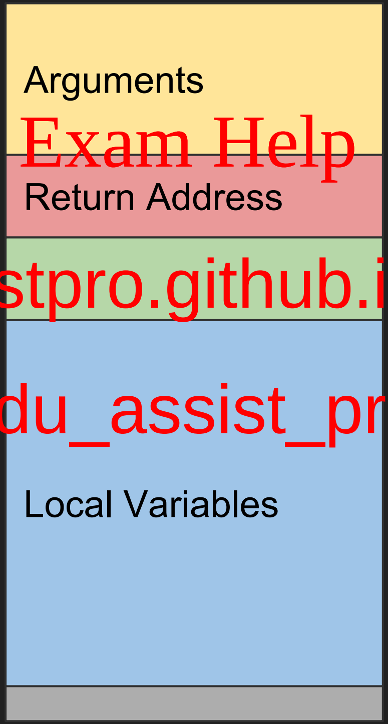
ine.

Shellcode Attack Types

ine.

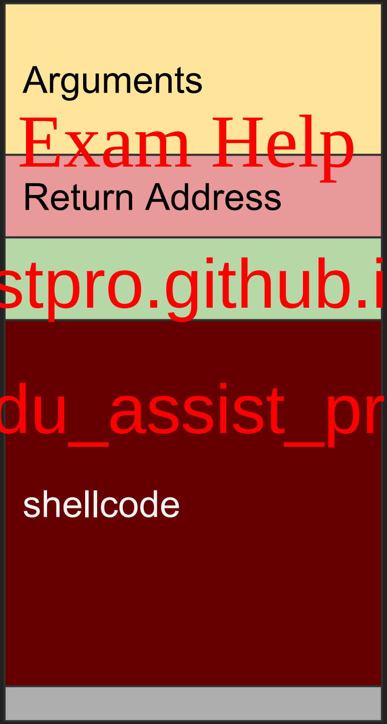
- Remote attacks.
    - We have no ability to run unmanaged code.
    - Goal?
        - Get native cod
    - Targets?
        - Browsers, net
- Local attacks.
    - We have shellcode access on the machine.
    - Goal?
        - Privilege escalation.
    - Targets?
        - Kernel, hypervisor, any process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

Stack based exploits



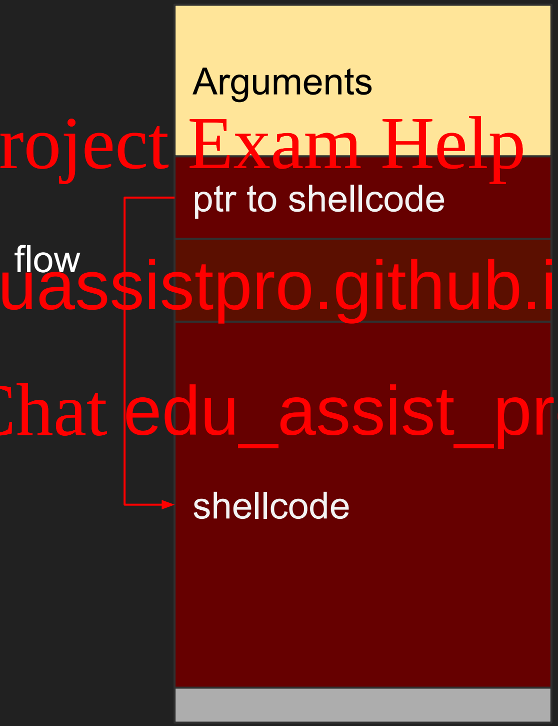
Stack based exploits

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



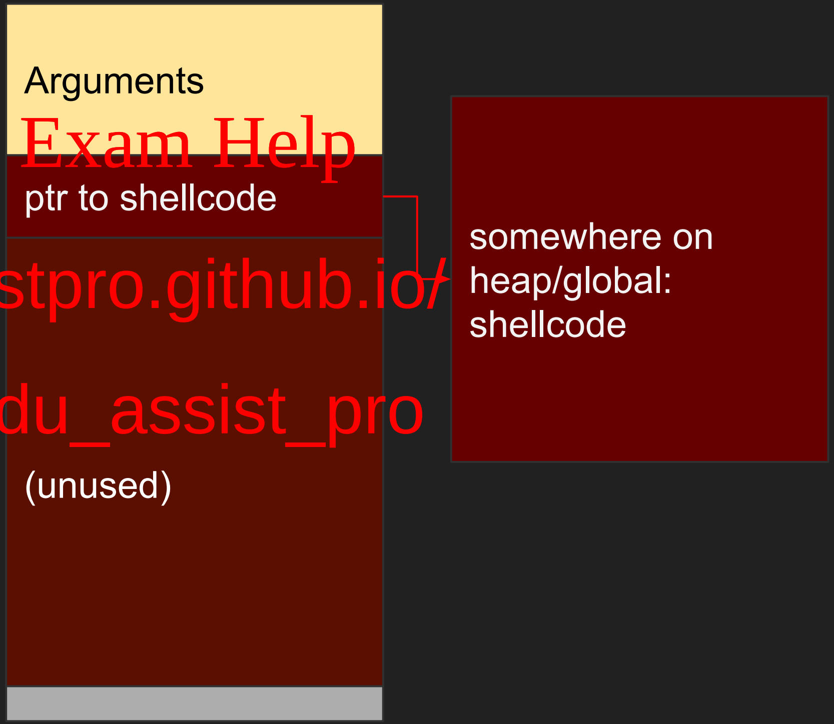
Stack based exploits

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



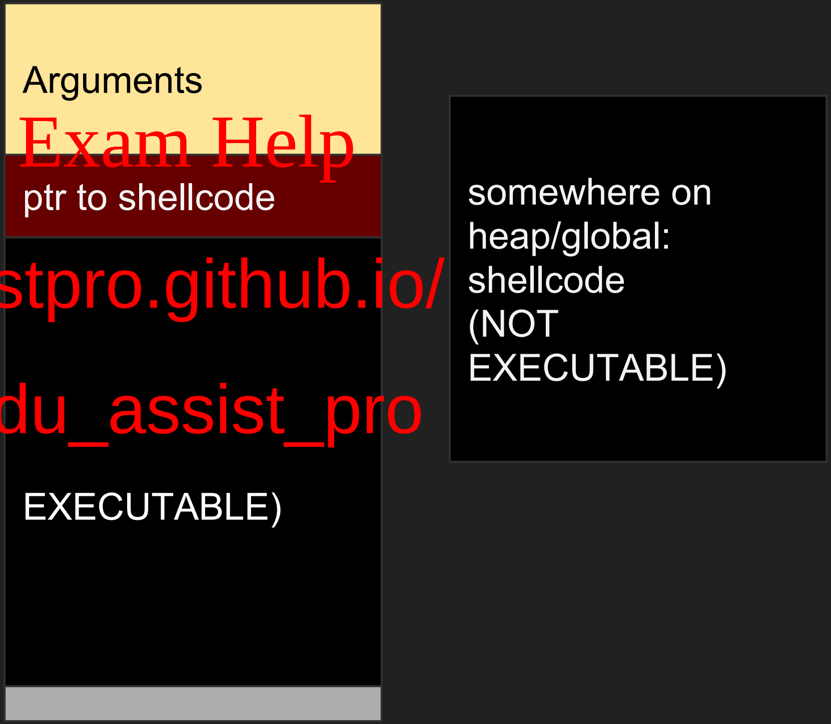
Heap based exploits

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



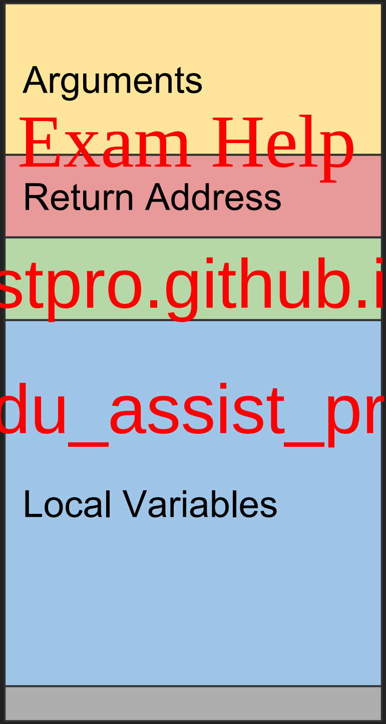
W^X Memory

- Write XOR Execute
- Can still write shellcode.
- Can't execute shellcode.
- Compiler sets sectio stack, heap, etc) me no-execute.
- Done by default on all compilers.
- Problem solved, right…?
  - Chrome still RWX v8 pages.



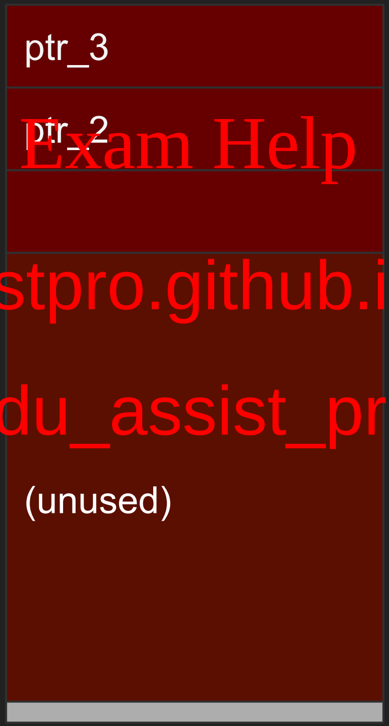
Return Oriented Programming

- Chains together “gadgets”, which is a sequence of a few instructions followed by ‘ret’.
- Smash stack with lo to gadgets.



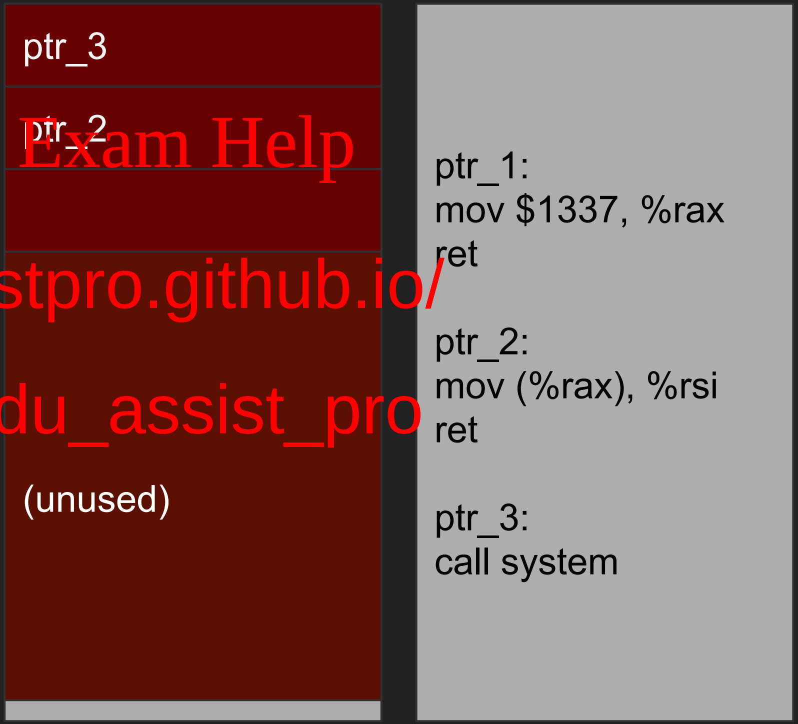
Return Oriented Programming

- Chains together "gadgets", which is a sequence of a few instructions followed by 'ret'.
- Smash stack with lo [illegible] 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 lo 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.



ptr_3

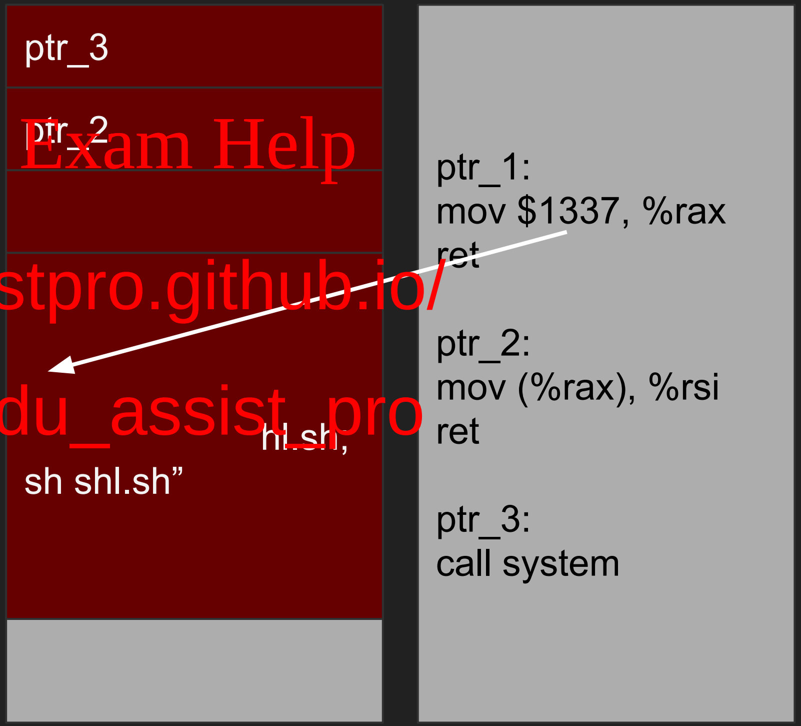
ptr_2

(unused)

```
ptr_1:
mov $1337, %rax
ret

ptr_2:
mov (%rax), %rsi
ret

ptr_3:
call system
```

Return Oriented Programming

- Chains together “gadgets”, which is a sequence of a few instructions followed by ‘ret’.
- Smash stack with lo 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.



ptr_3

ptr_2

hl.sh;
sh shl.sh”

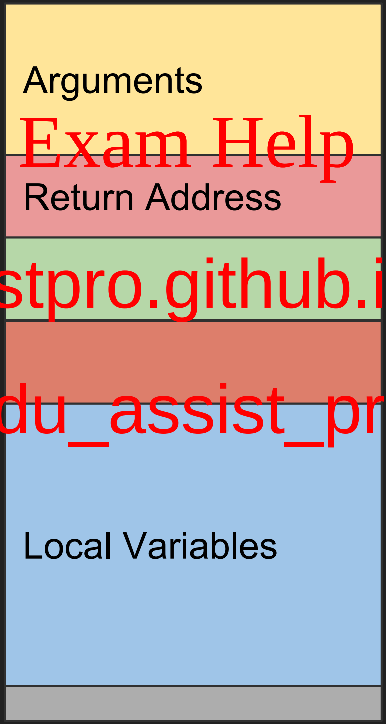
ptr_1:
mov $1337, %rax
ret

ptr_2:
mov (%rax), %rsi
ret

ptr_3:
call system

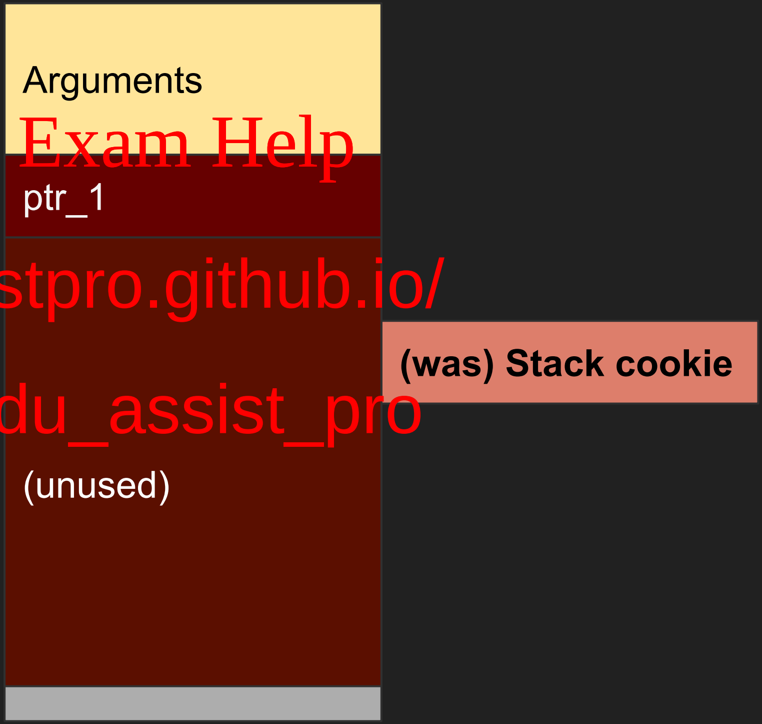
Stack Cookies

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



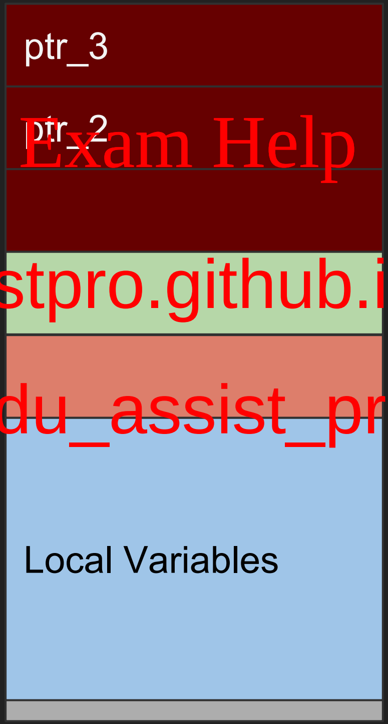
Stack Cookies

```
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
```



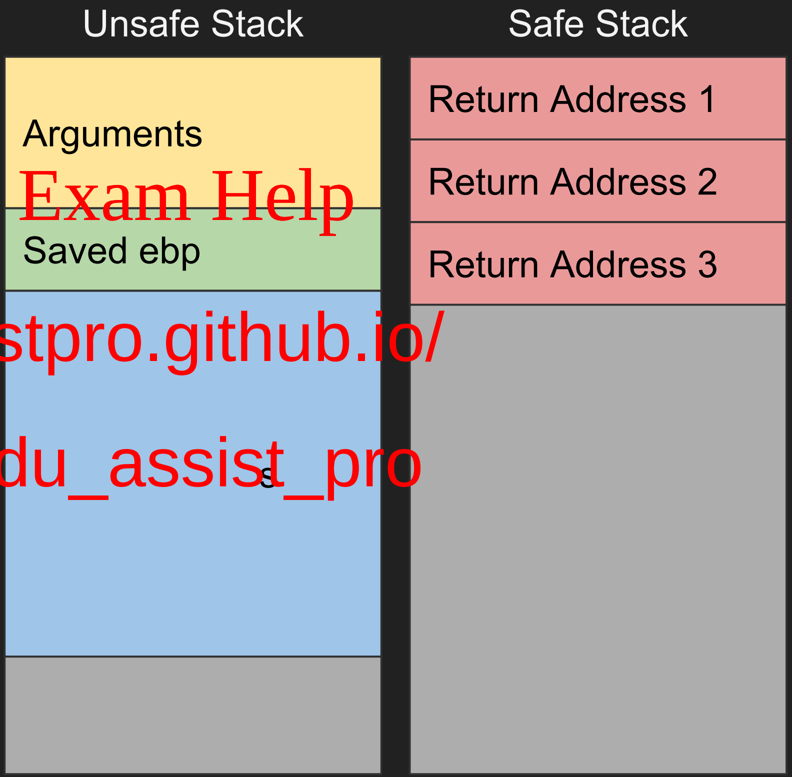
Stack Cookies

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



Shadow Call Stack

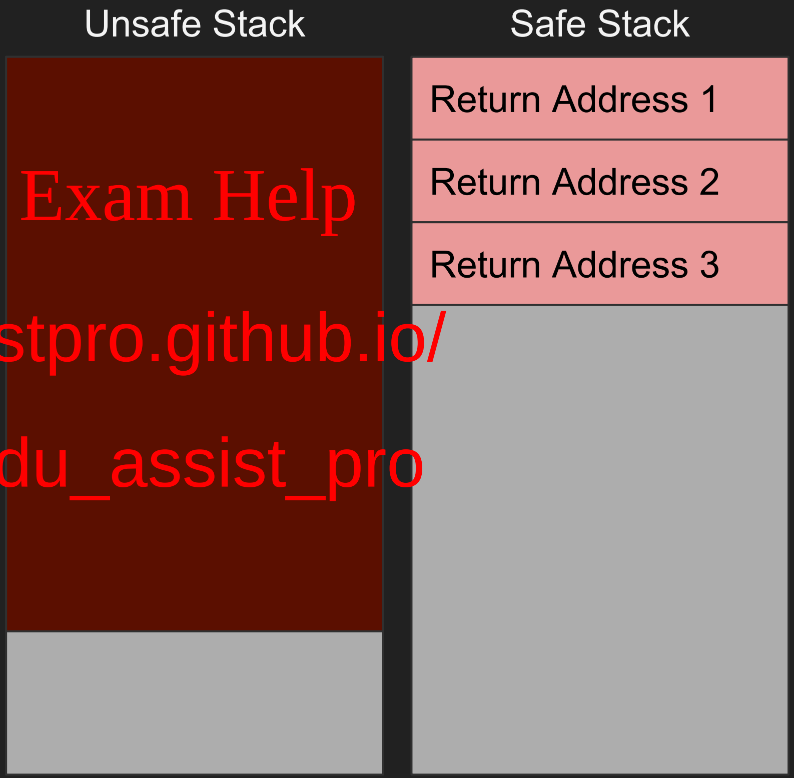
- -fsanitize=shadow-call-stack
- Another ROP defense
- Separate stacks into *safe* and *unsafe*.
- Safe contains return
- Unsafe contains everything else.



Shadow Call Stack

- Arbitrary writes are still safe as long as safe stack pointer is secret.
- Safe stack pointer is hidden:
  - Reserved register (x
  - Reserved segment (

flow



TOCTOU Vulnerabilities

- Time of Check to Time of Use (TOCTOU)
- Thread-based security race between call and pr 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.

shadow stack.

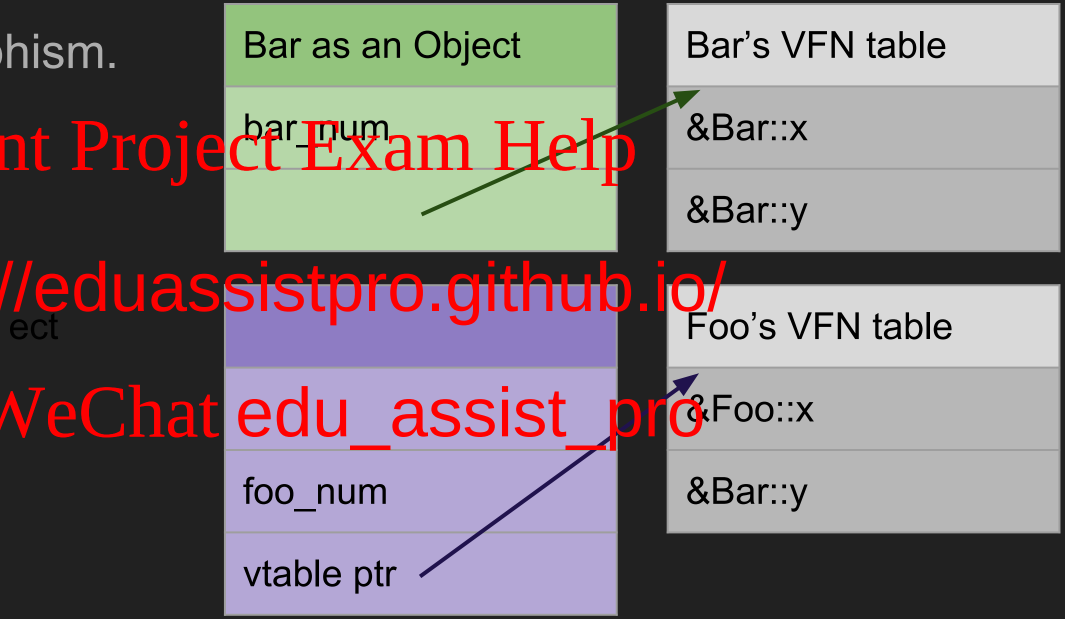
S k Epilogue:

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

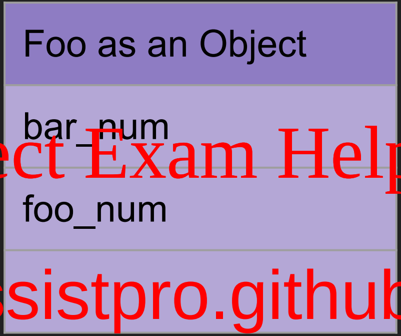
- Used to implement polymorphism.

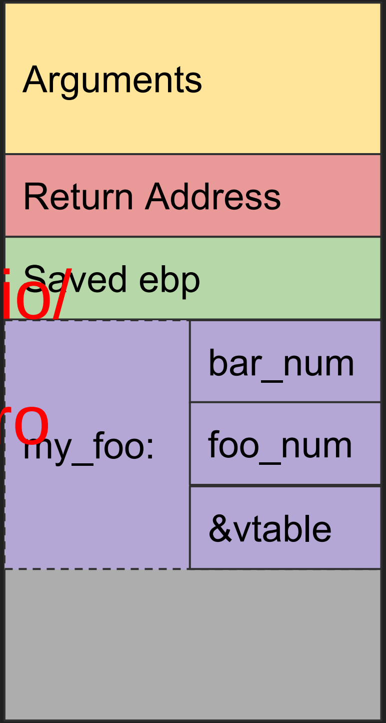
```
struct Bar {
    virtual void x() { … }
    virtual void y() { … }
    int bar_num = 0;
}

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



C++ Virtual Function Tables

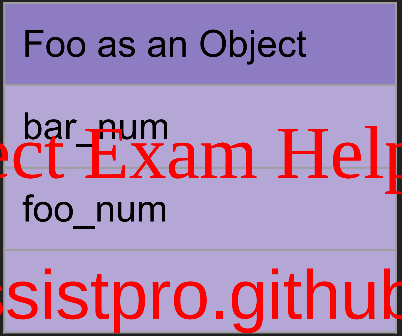
```
int x() {
        Foo my_foo;
        foo.x();
}
```





C++ Virtual Function Tables

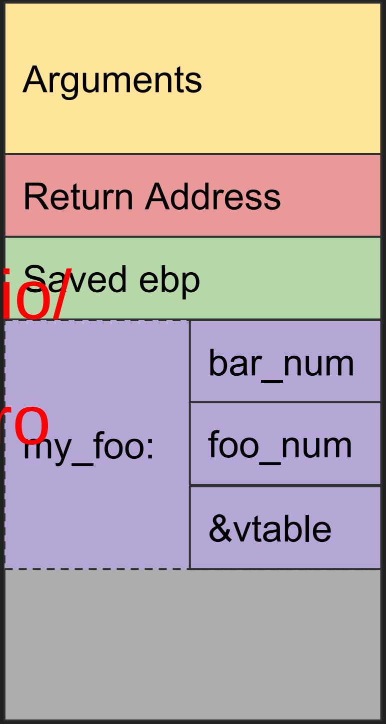
```
int x() {
        Foo my_foo;
        foo.x();
}


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
```



Foo as an Object

bar_num

foo_num



Arguments

Return Address

Saved ebp

my_foo:

bar_num

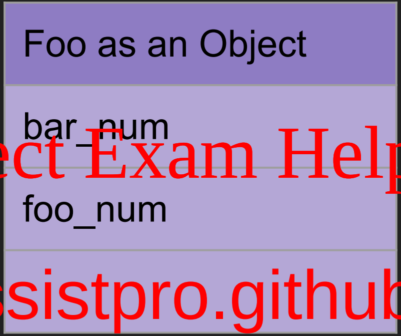
foo_num

&vtable

C++ Virtual Function Tables

- Add some dangerous code and...

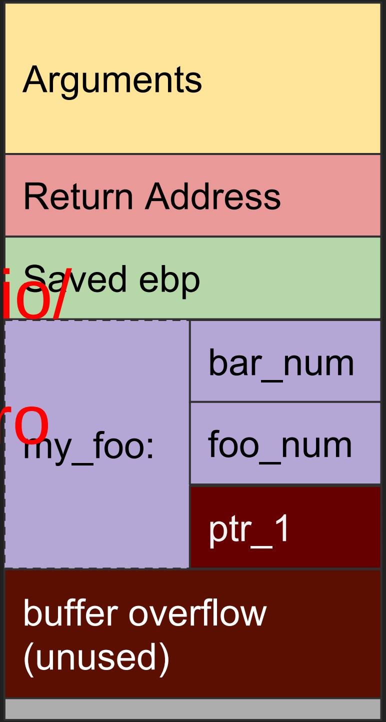
```
int x() {
    Foo my_foo;
    char buffer[255];
    fgets("%s", buffer);
    foo.x();
}
```



Foo as an Object

bar_num

foo_num



Arguments

Return Address

Saved ebp

my_foo:

bar_num

foo_num

ptr_1

buffer overflow (unused)

Control Flow Integrity (CFI)

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

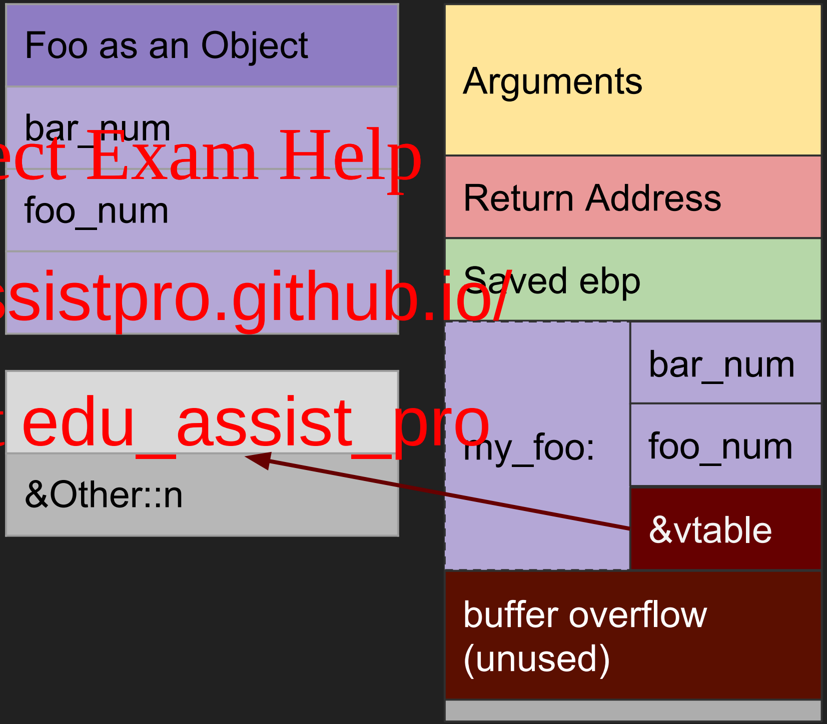
```
movq -8(%rsp), %rsi    ; get vtable ptr
movq $0, %rsi          ; index of &Foo:x
call cfi_check
            %rsi
                       ; get &Foo:x
                       ; call &Foo:x

cf
; 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.



Foo as an Object

bar_num

foo_num

&Other::n

Arguments

Return Address

Saved ebp

my_foo:

bar_num

foo_num

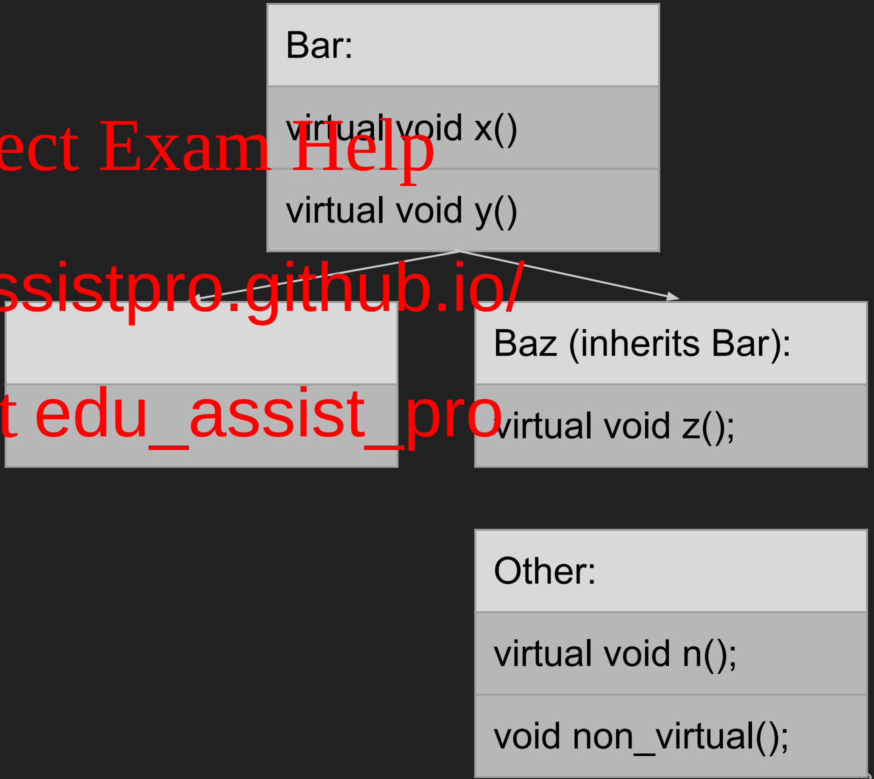
&vtable

buffer overflow (unused)

CFI Cast Checking

- CFI adds checks to all types of cast checking.

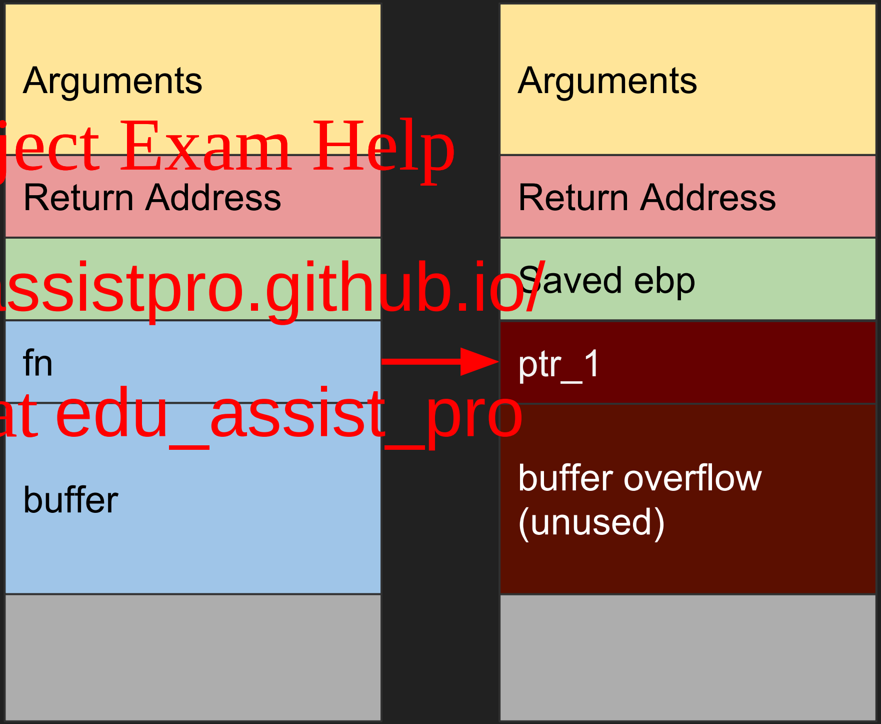
```
Baz* b1 = new Foo();
Baz* b2 = new Bar();
Bar* b3 = new Foo();           // OK
Baz* b4 = b3;                  // wrong
(Other* b3)->non_virtual(); // wrong
Bar* b6 = new Other();         // wrong
void* o = new Other();         // OK
(Bar* o)->y();                 // wrong
```



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();
}
```

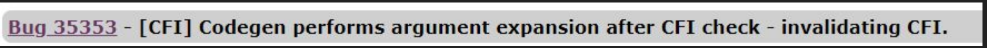


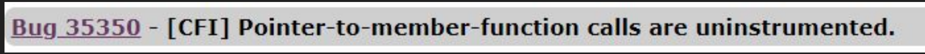
CFI Issues

- Only forward-edge protection.
  - rCFI is implemented, but very expensive and requires significant metadata.
- Cross-DSO is computationally expensive.
- Checks can get quit
  - e.g. Base class vcall

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

- also...


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


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

Other CFI

- Microsoft Control Flow Guard (CFG)
    - Near-precise. Isn't perfect.
- Intel Control Enforcement Technology (CET)
    - Hardware enforced
    - Also ENDBRANCH,
- ARM Pointer Authe

cise.

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
  - -fstack-protector (-fs
  - -fsanitize=cfi
  - -shadow-call-stack
- Compilers are made by humans. Any s ode should always be inspected by hand.