Unwinding The Stack For Fun and Profit

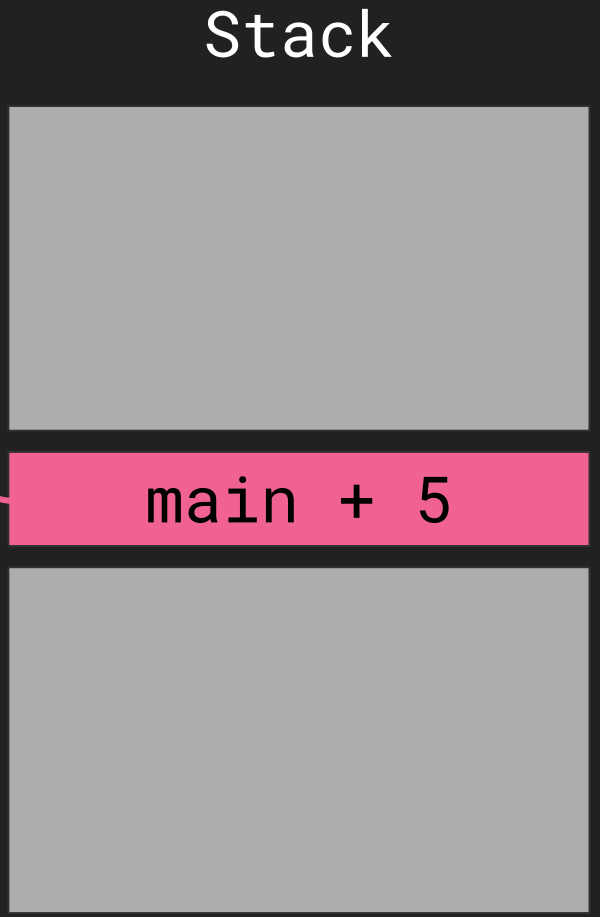
Victor M. Duta Fabian Freyer

Fabio Pagani Marius Muench Cristiano Giuffrida

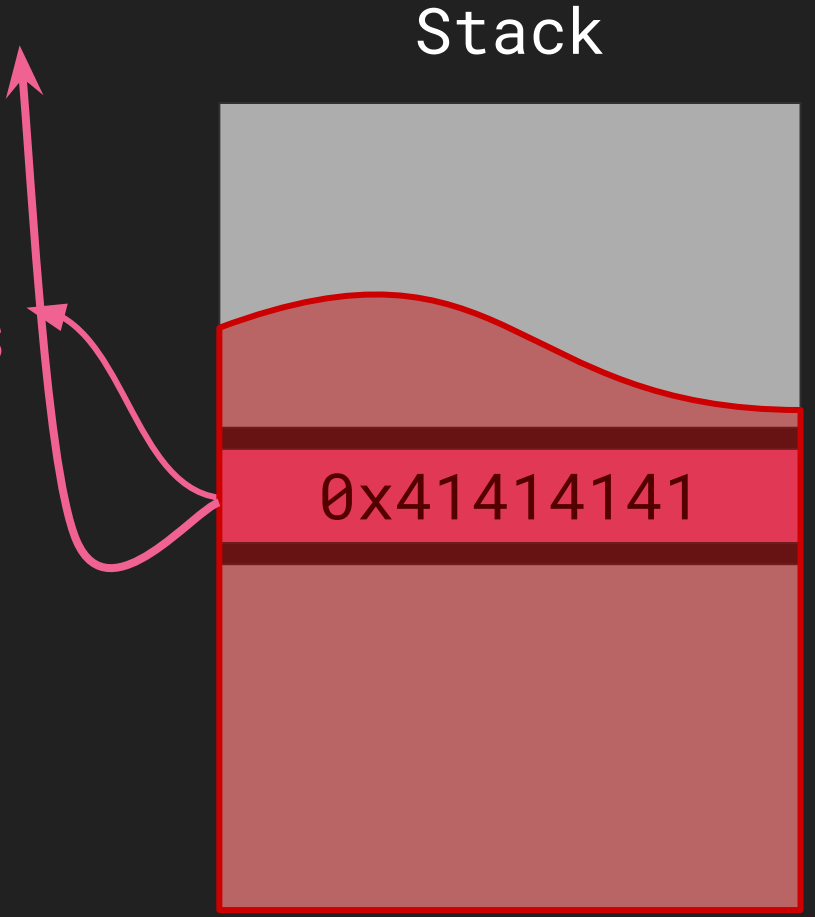
Binary Exploitation

Control Flow Hijacking

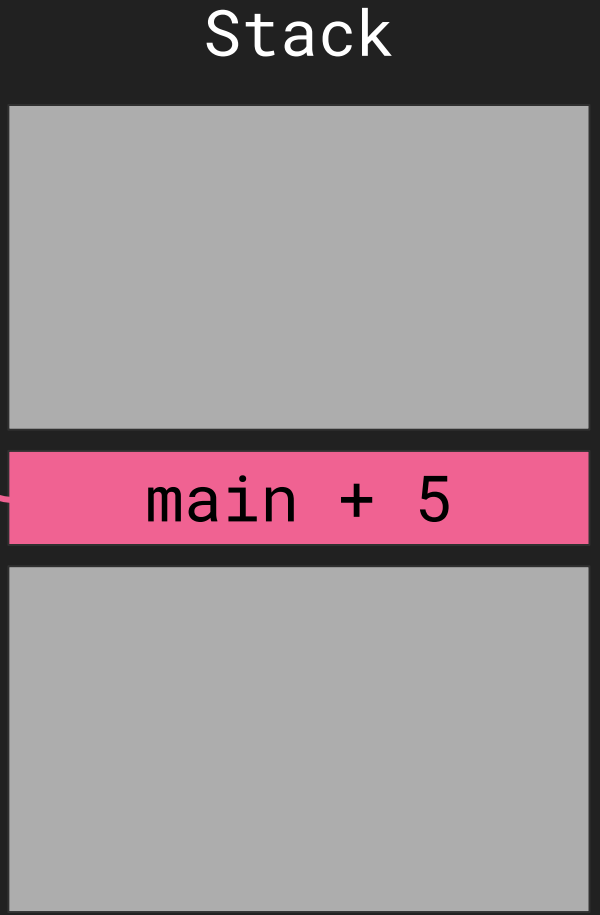
```
int main() {
    callme();
    // more things
}

void callme() {
    // do things
}
```



```c
int main() {
    callme();
    // more things
}

void callme() {
    // overflow
}
```

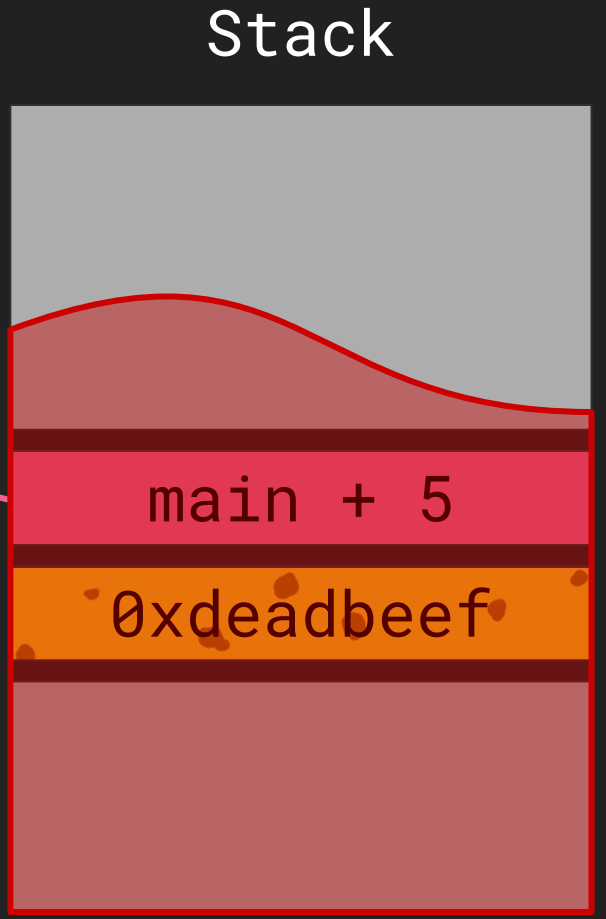


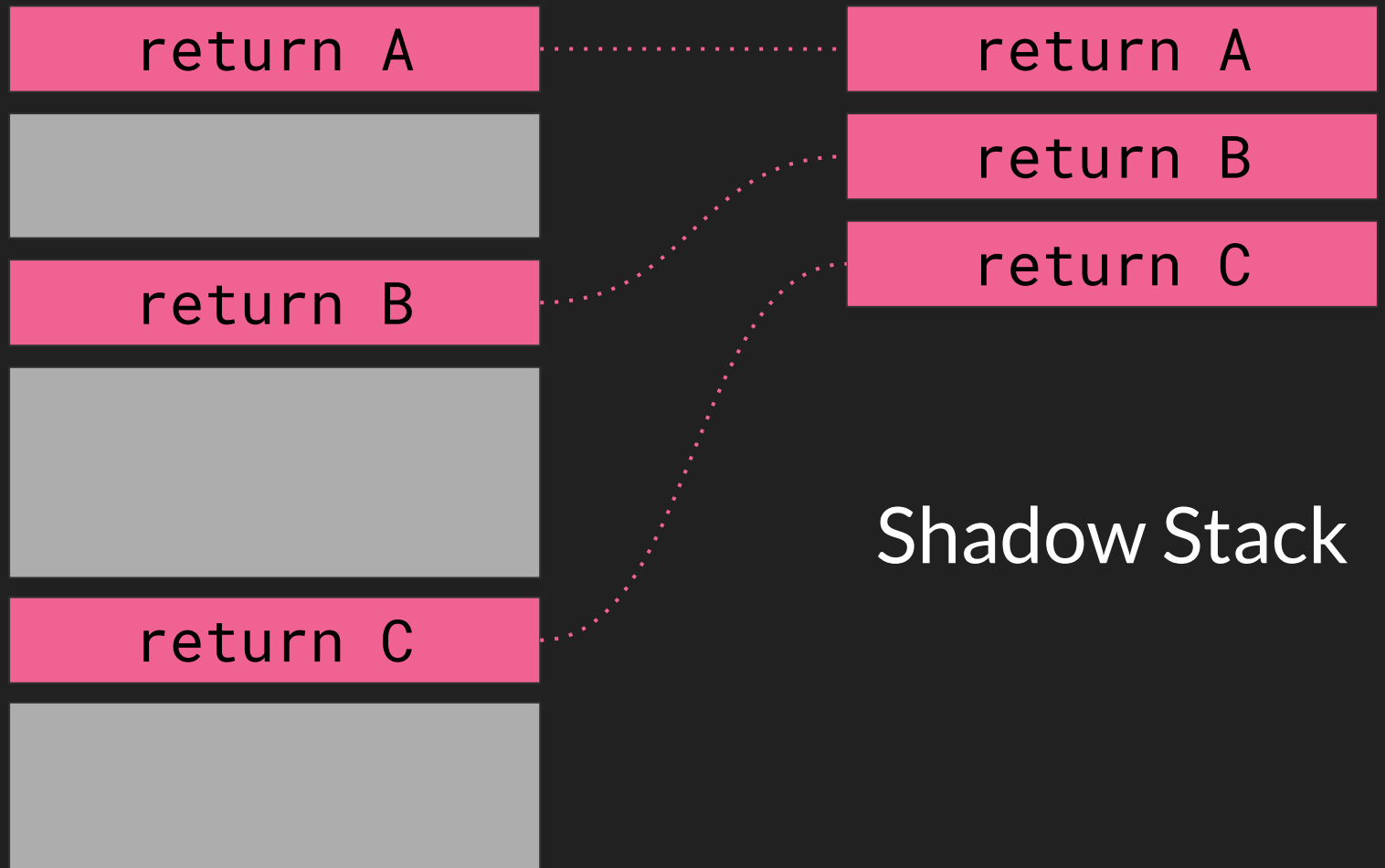
```
int main() {
    callme();
    // more things
}

void callme() {
    // overflow

}
```



```
int main() {
    callme();
    // more things
}

void callme() {
    // overflow
    // check canary
}
```

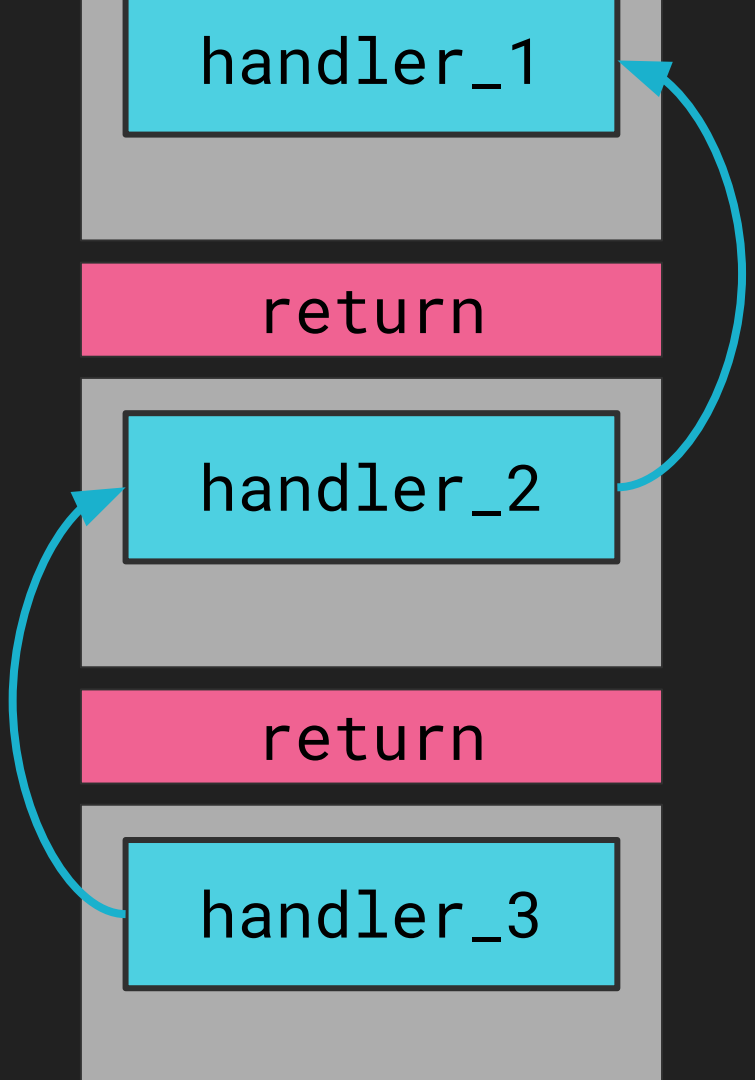
```
int main() {
    callme();
    // more things
}

void callme() {
    // overflow
    // check canary
}
```




return A
return B
return C
return A
return B
return C
Shadow Stack

Backward edge Control-flow Hijack

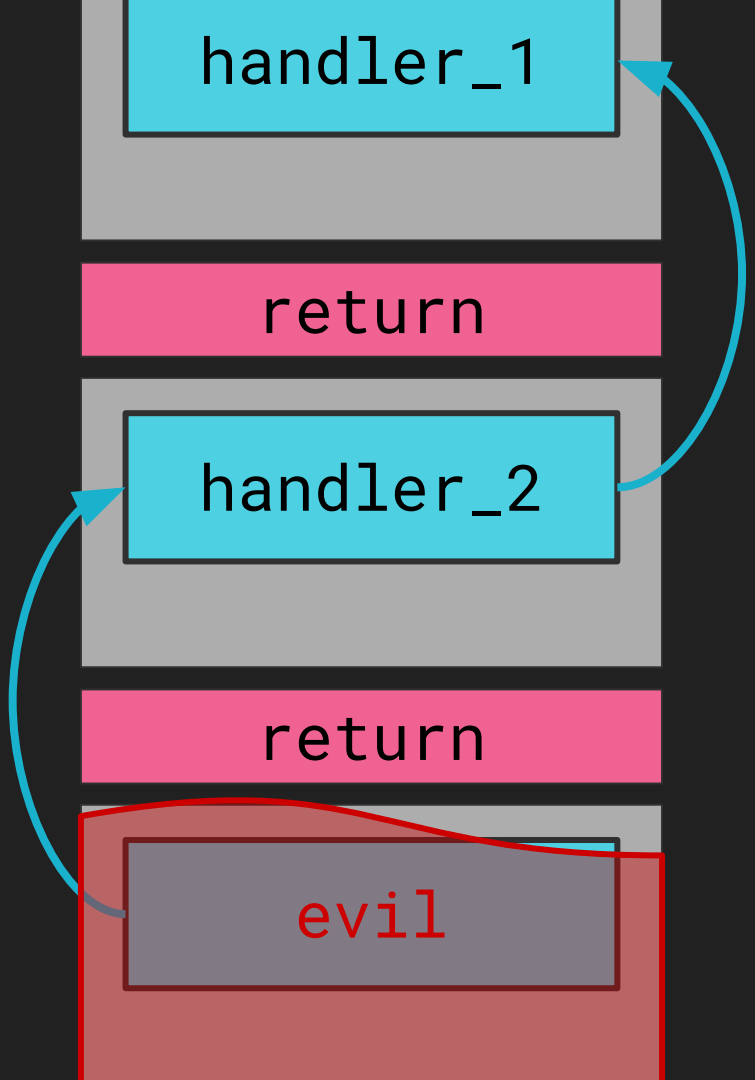
Forward edge Control-flow Hijack



SEH (Windows)

Single Linked List
of pointers to handlers

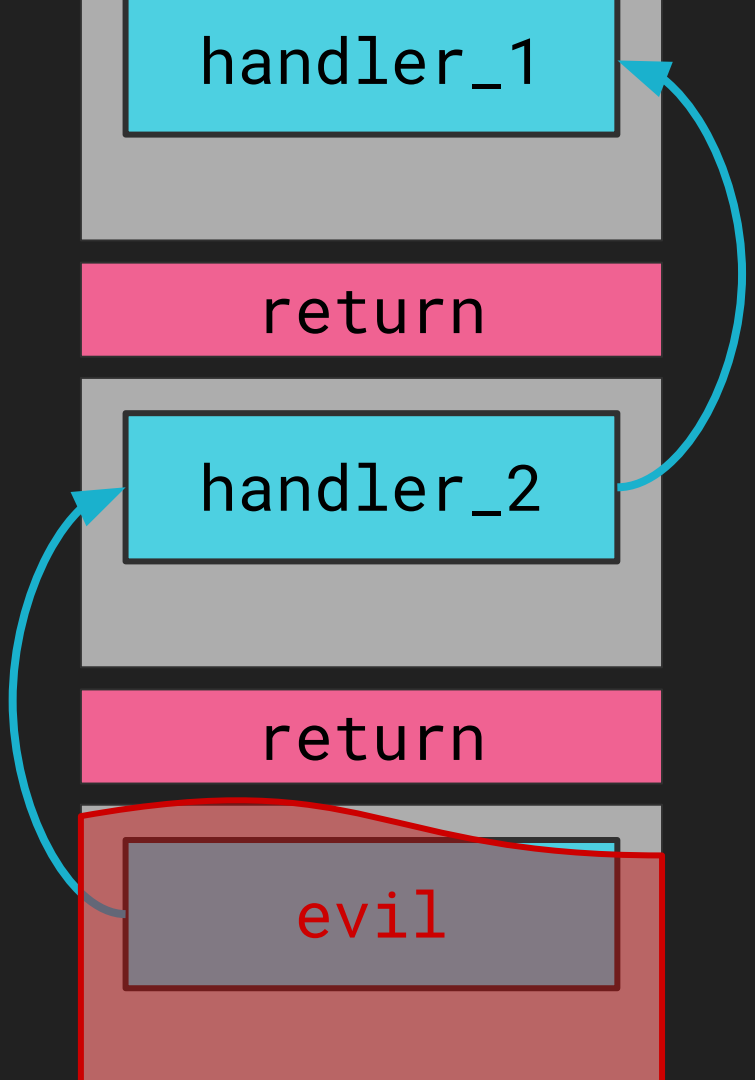
Called when Exception is thrown



SEH (Windows)

Single Linked List
of pointers to handlers

Called when Exception is thrown



SafeSEH

Allowed List of Handlers

- `handler_1`
- `handler_2`
- `handler_3`

`evil` not in list!

Stack Buffer Overflows are a solved problem!

Thank you for listening!

Any questions?

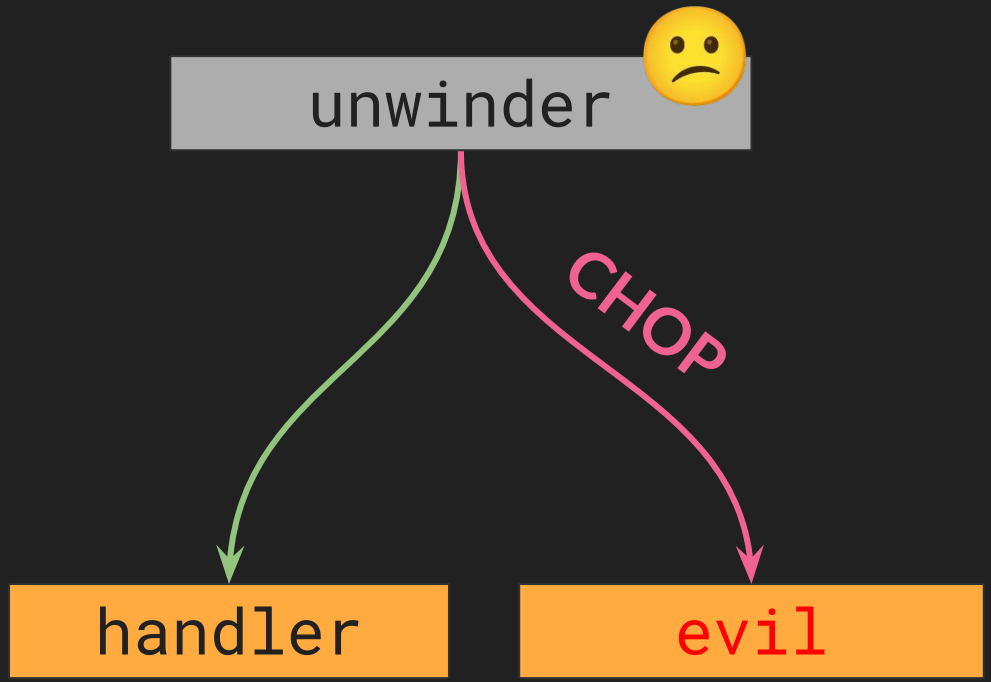
Are there any exceptions to these mitigations?

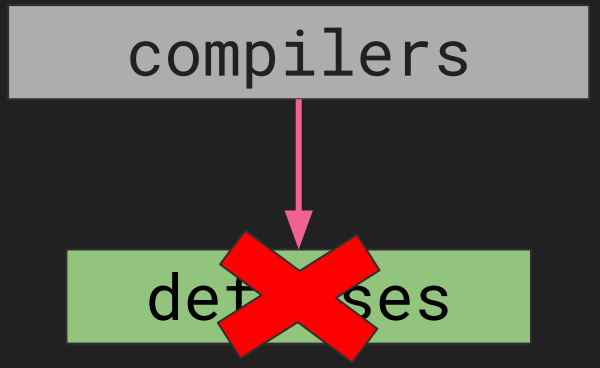
Yes!

CHOP

Control Flow Hijacking by confusing the unwinder

In a nutshell





NDSS 2023

Let Me Unwind That For You: Exceptions to Backward-Edge Protection

Confidential Preprint - Do not redistribute.

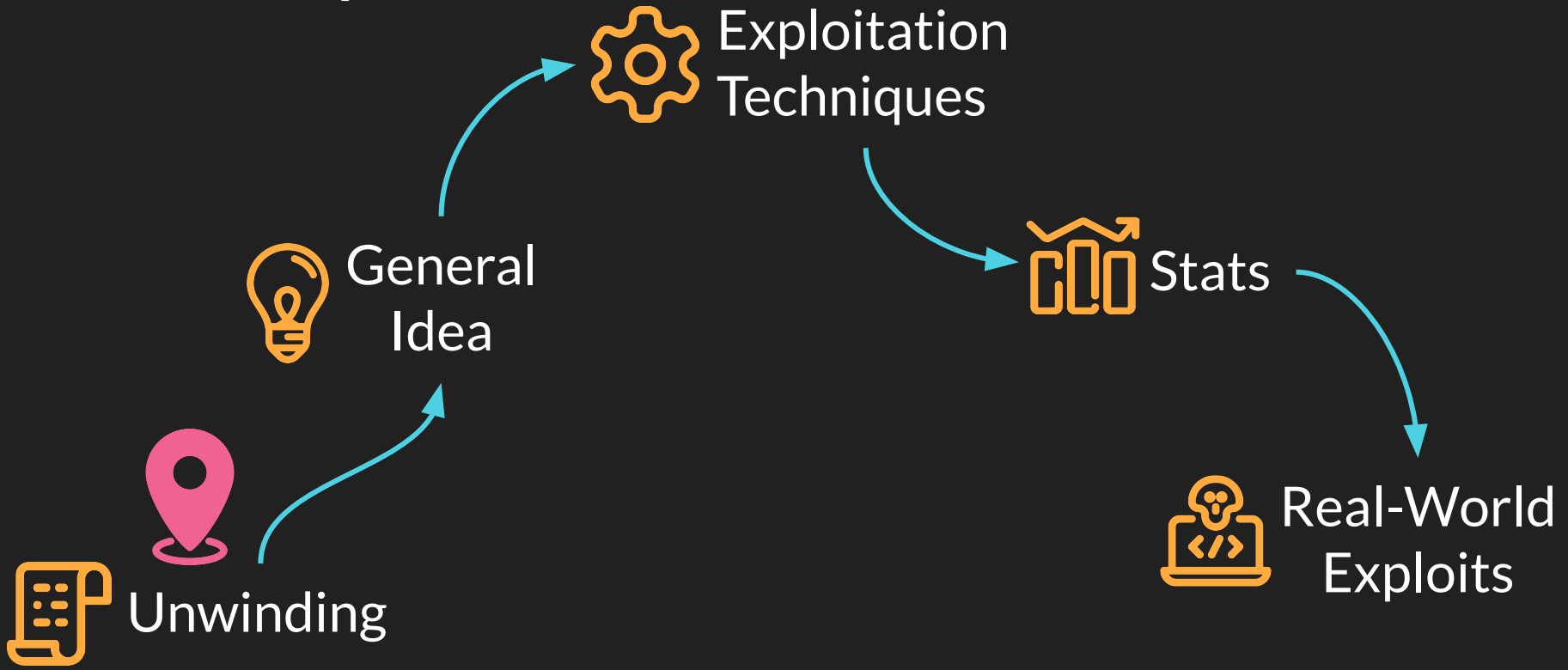
Victor Duta*§, Fabian Freyer†§, Fabio Pagan[illegible]ius [illegible]ch [illegible]stiano Giuffrida*
*Vrije Universiteit Amsterdam, {v.m[illegible]@cs.vu.nl

***Abstract*—Backwa[illegible]control-f[illegible]ija[illegible]fia[illegible]buffer overflow is th[illegible]oly gra[illegible]e[illegible]ability to directly cont[illegible]ca[illegible]an[illegible]target makes this explo[illegible]on s[illegible]ula[illegible]caling for attackers. As a result[illegible]on[illegible]h[illegible]eployed backward-edge protection[illegible]ch a[illegible]ha[illegible]w stac[illegible]ack canaries, forcing attackers t[illegible]o[illegible]ss ide[illegible]eap-based exploitation strategies. Howe[illegible]ch mi[illegible]s commonly rely on one key assumption, namely[illegible]ker relying on return address corruption to *directly*[illegible]control flow upon function return.**

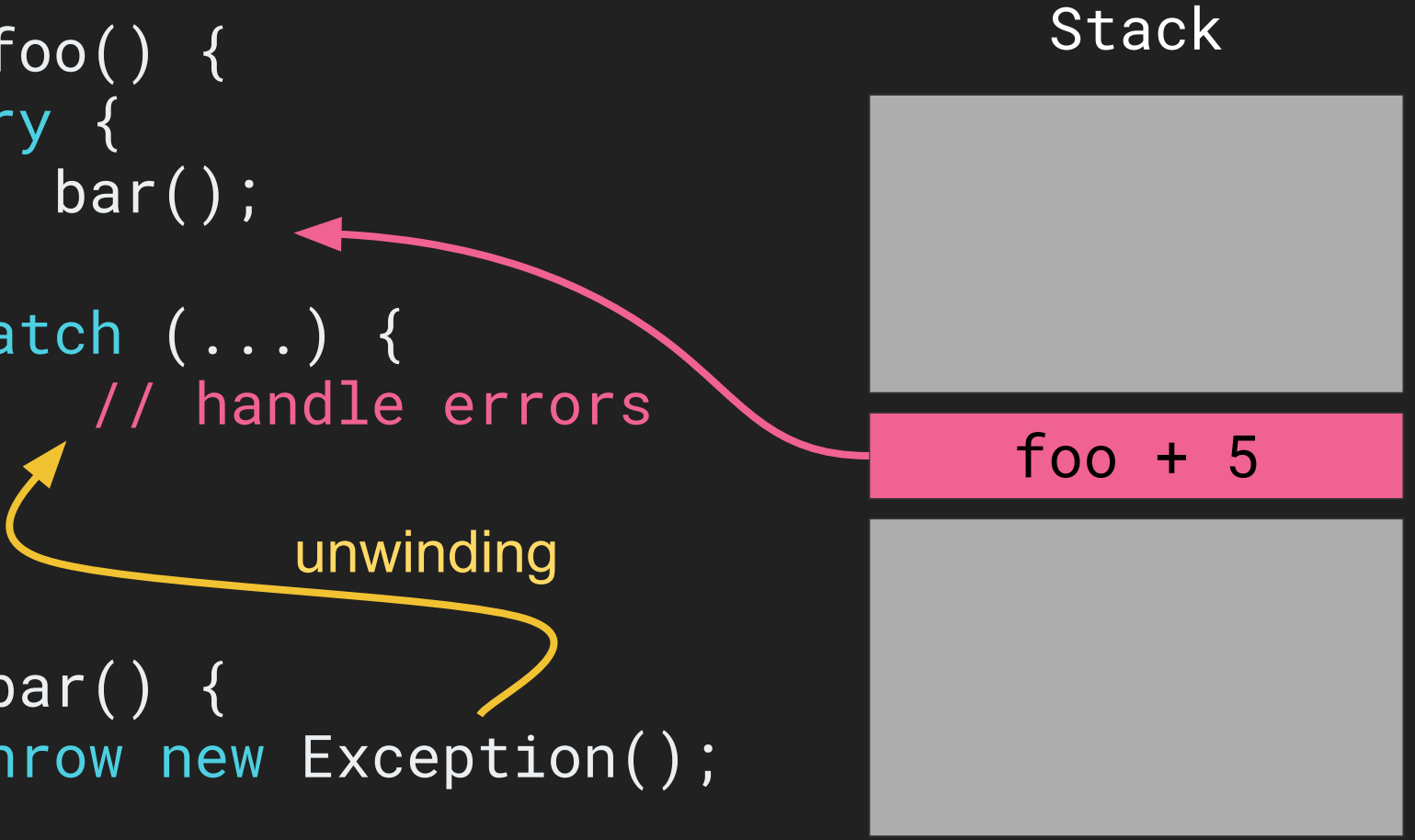
[illegible]bilities are still painfully [illegible]mo[illegible]d co[illegible]nt both developers and end [illegible]gh-va[illegible]rget[illegible]in this space are stack buffer overflow [illegible]nerabi[illegible]es, [illegible]h have historically enabled "convenient" ba[illegible]ge control-flow hijacking attacks corrupting critical [illegible]rol (i.e. return address) and non-control (e.g., pointers in locals and saved registers) data on the stack.

In response, researchers have devised strong mitigations such as stack canaries [3] and shadow stacks [4] to protect backward-edge integrity and cripple exploits. This effort has

Roadmap

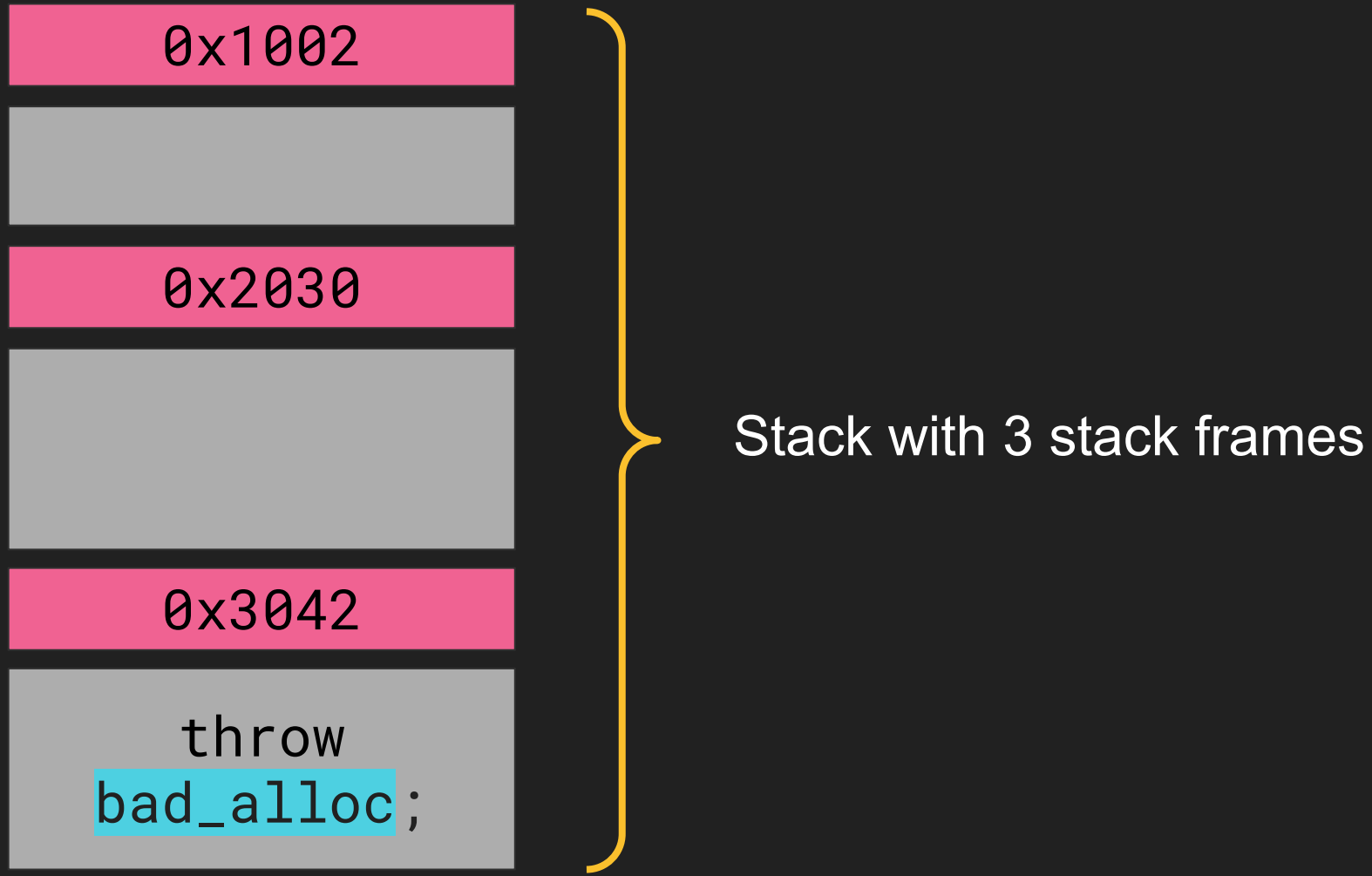

Exploitation Techniques
Stats
Real-World Exploits
General Idea
Unwinding

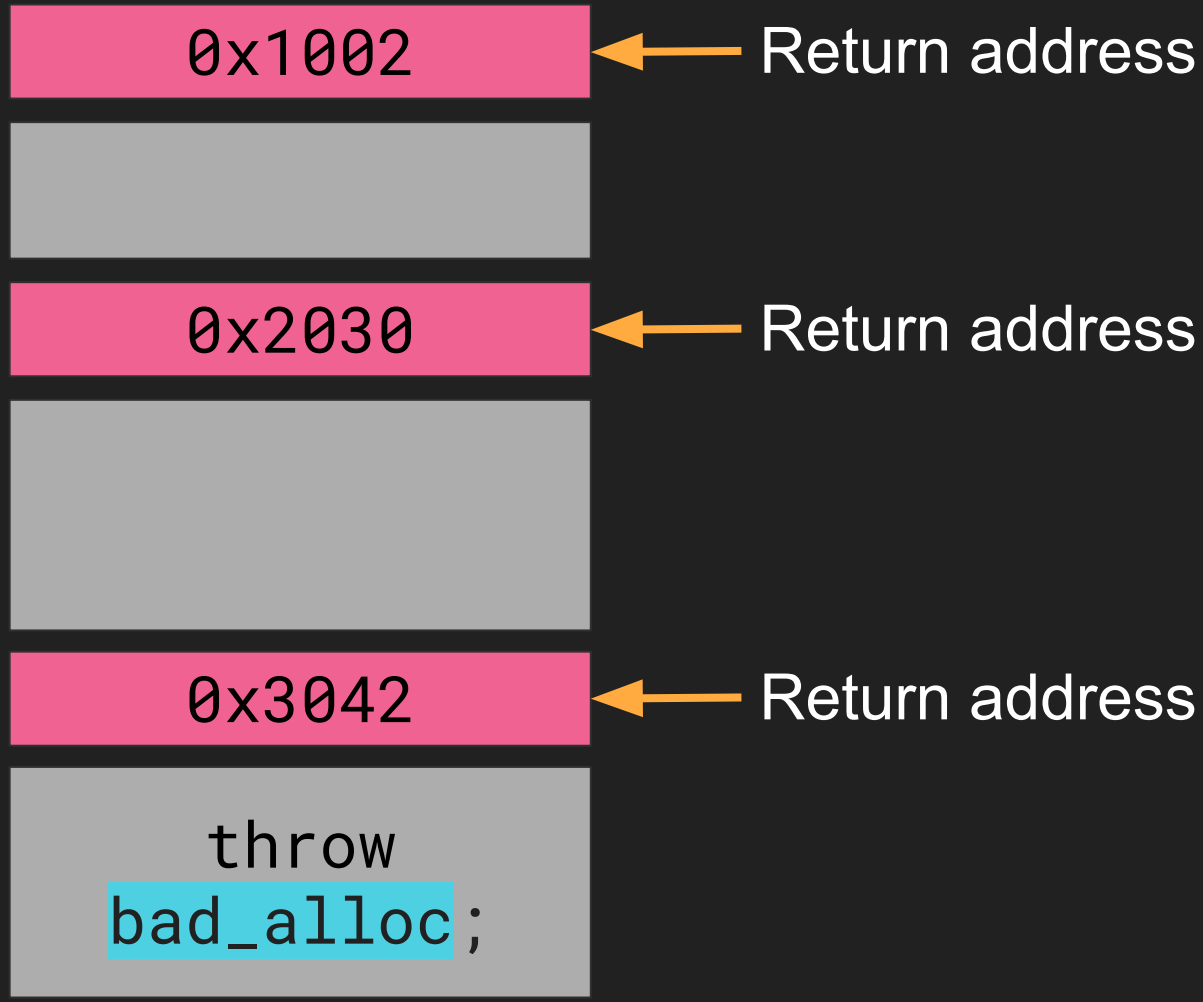
```
void foo() {
    try {
        bar();
    }
    catch (...) {
        // handle errors
    }
}

void bar() {
    throw new Exception();
}
```



Exception Tables

Call Site	Exception Metadata	Landing Pad
`0x1000..0x1004`	`*`	`0x1042`
`0x2000..0x2040`	`runtime_error`	`0x2080`


0x1002
0x2030
0x3042
throw
bad_alloc;
Stack with 3 stack frames

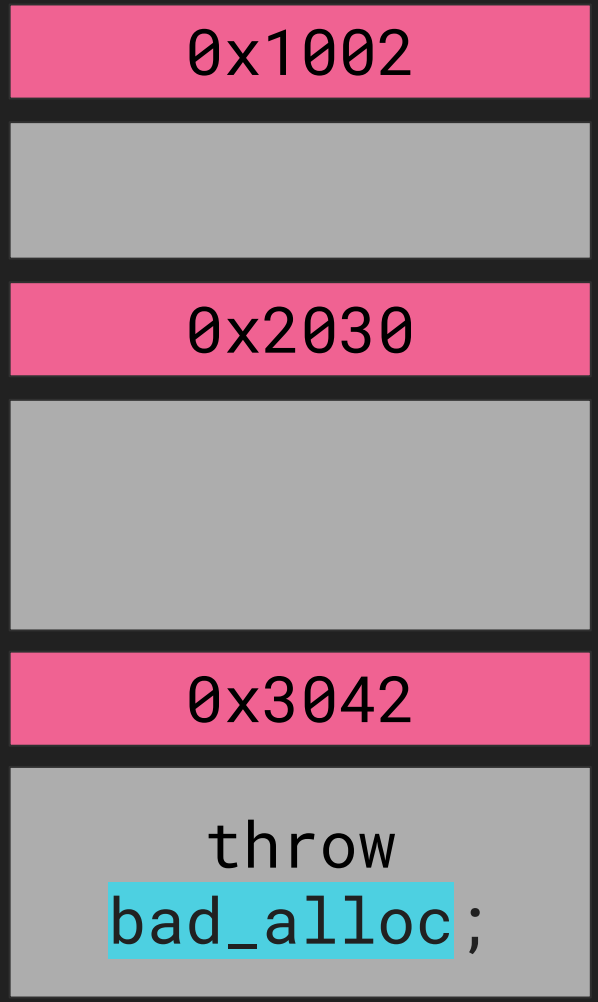

0x1002
Return address
0x2030
Return address
0x3042
Return address
throw
bad_alloc;

0x1002

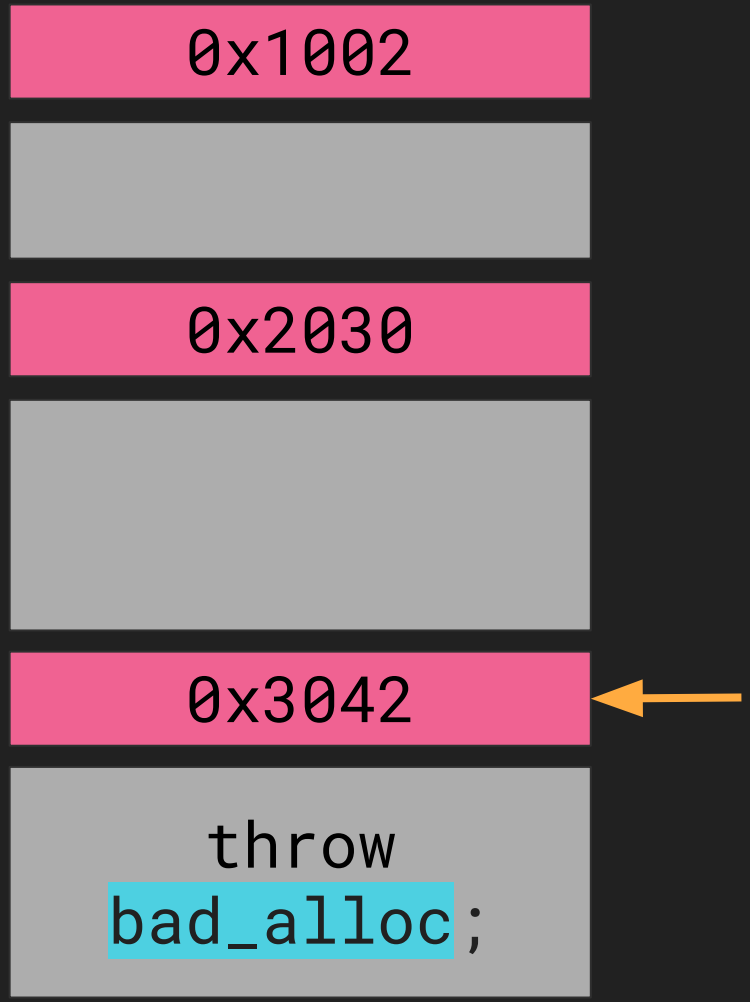
0x2030

0x3042

```
throw
bad_alloc;
```

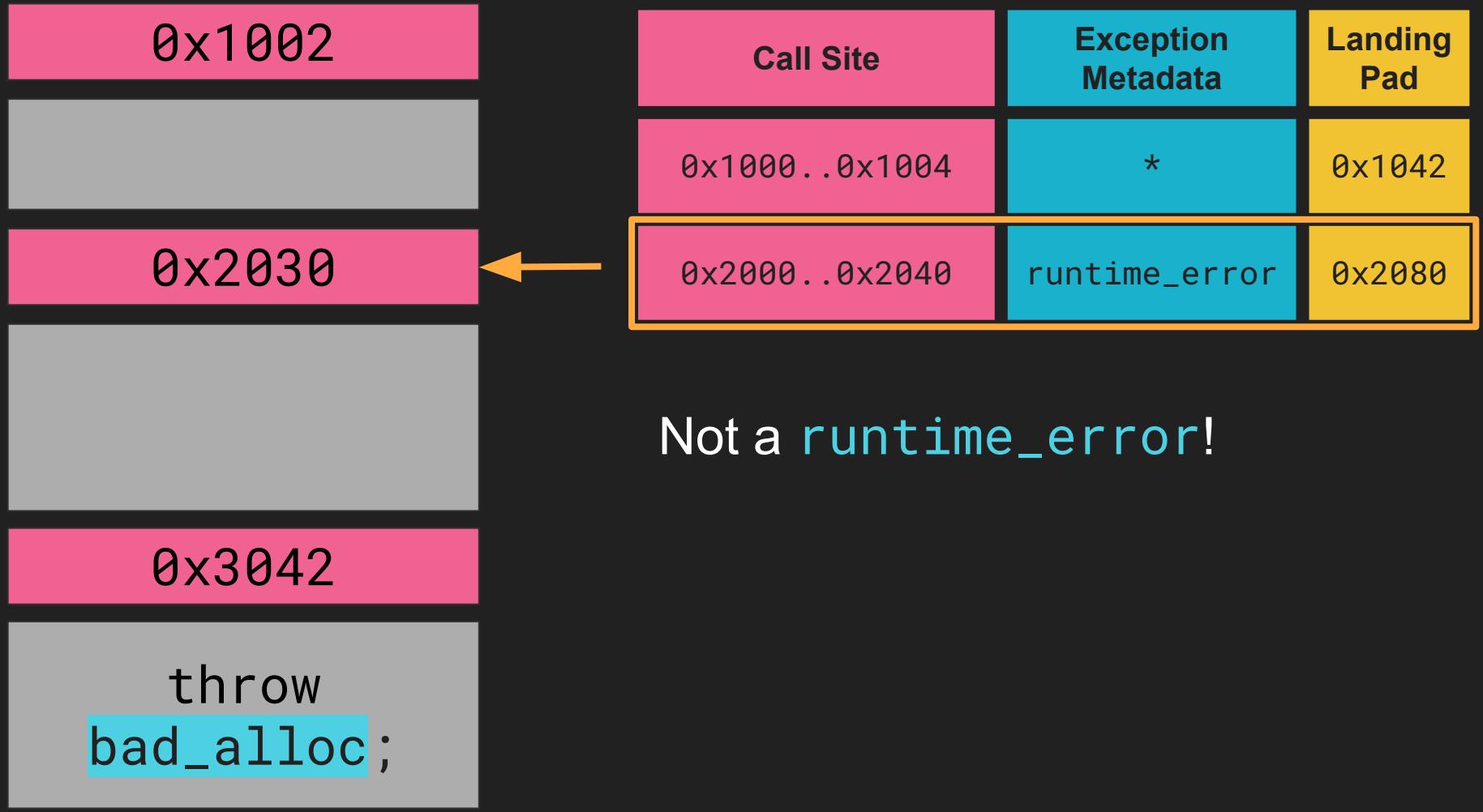


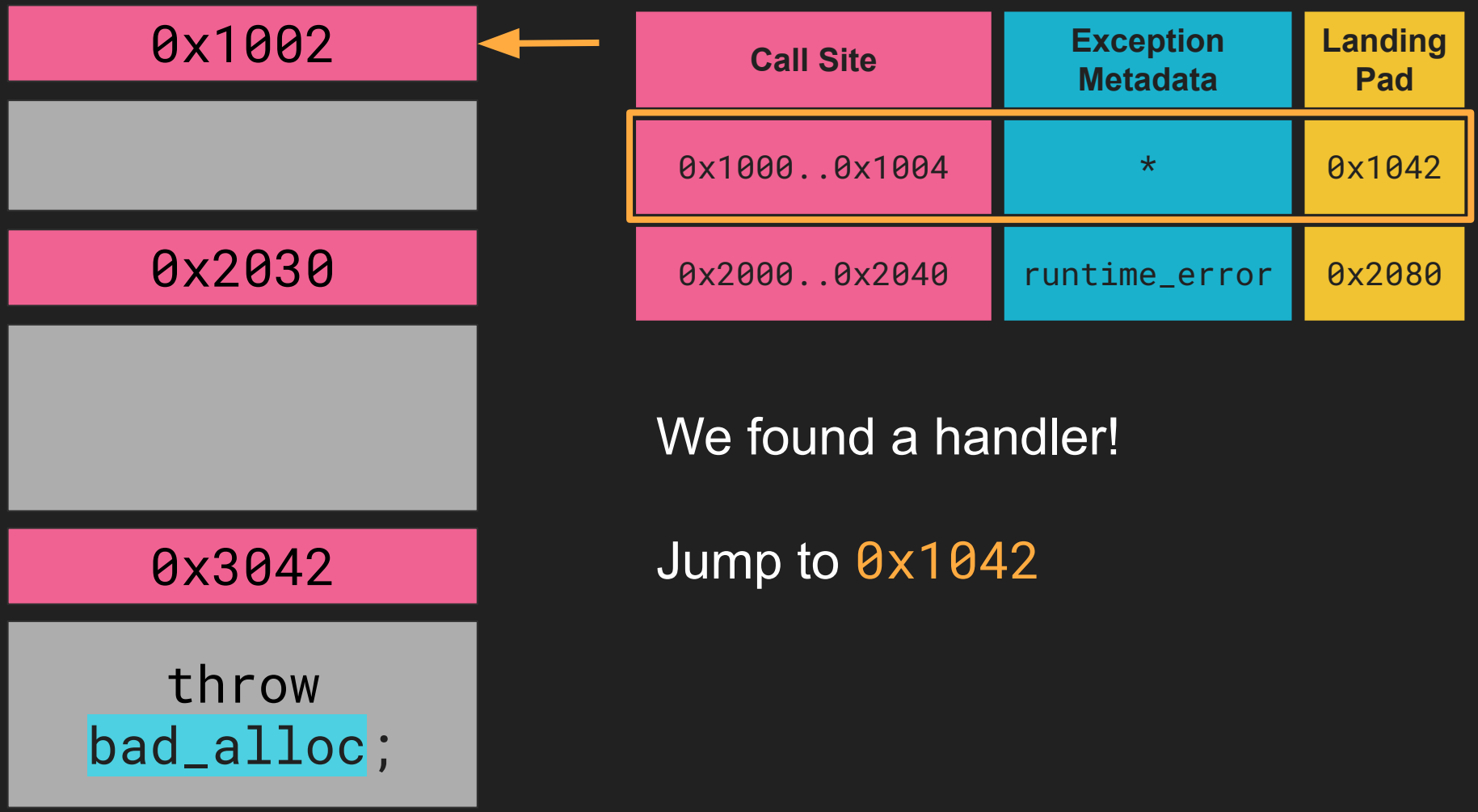
Call Site	Exception Metadata	Landing Pad
0x1000..0x1004	*	0x1042
0x2000..0x2040	runtime_error	0x2080



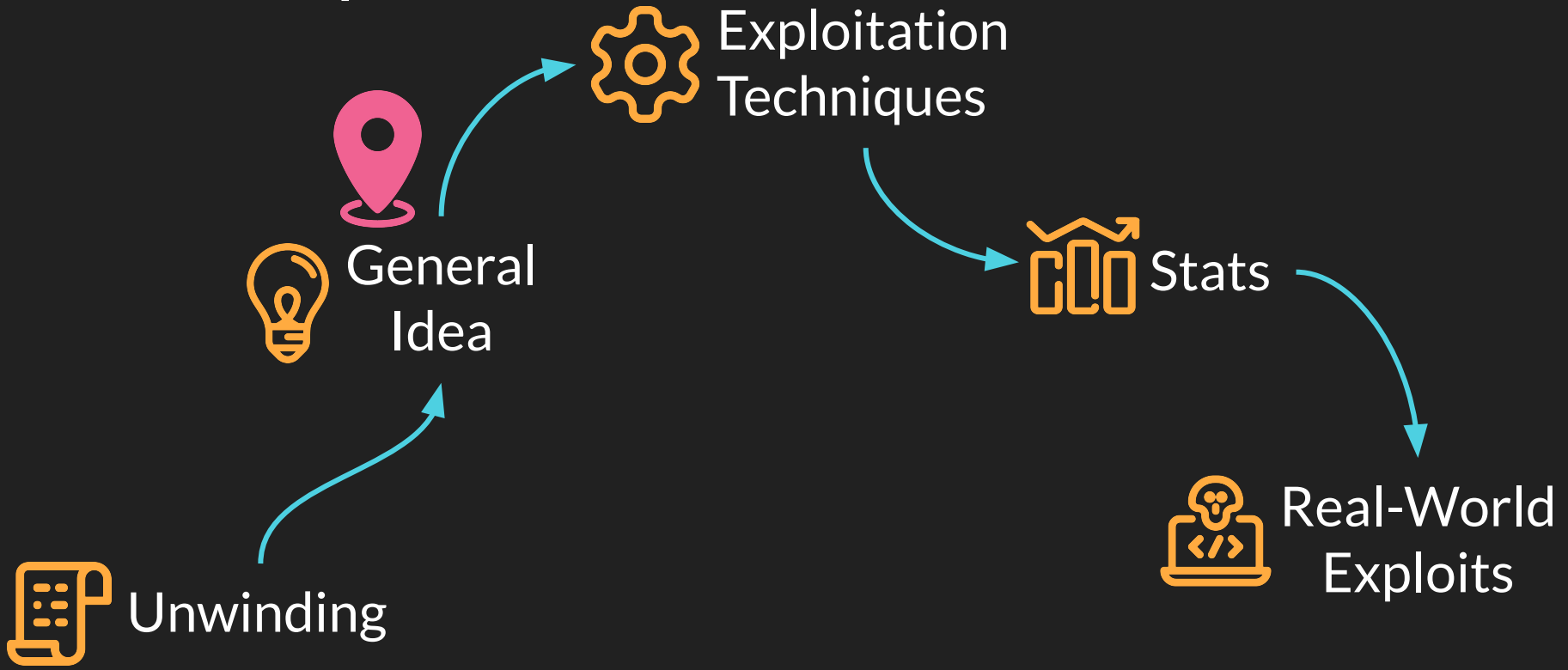
Call Site	Exception Metadata	Landing Pad
0x1000..0x1004	*	0x1042
0x2000..0x2040	runtime_error	0x2080

No Handler in Table

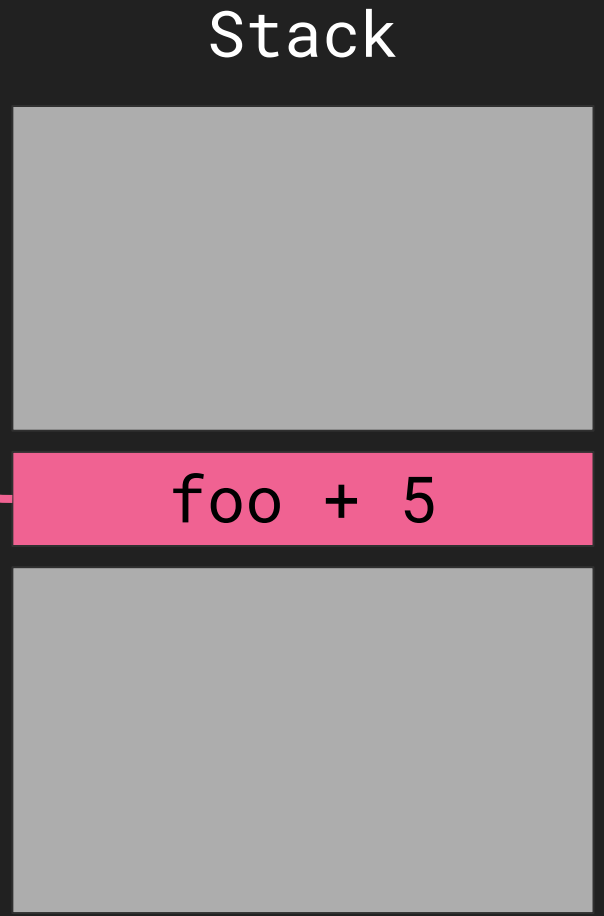

0x1002
0x2030
0x3042
throw
bad_alloc;
Call Site
Exception Metadata
Landing Pad
0x1000..0x1004
*
0x1042
0x2000..0x2040
runtime_error
0x2080
Not a runtime_error!


0x1002
0x2030
0x3042
throw
bad_alloc;
Call Site
Exception Metadata
Landing Pad
0x1000..0x1004
*
0x1042
0x2000..0x2040
runtime_error
0x2080
We found a handler!
Jump to 0x1042

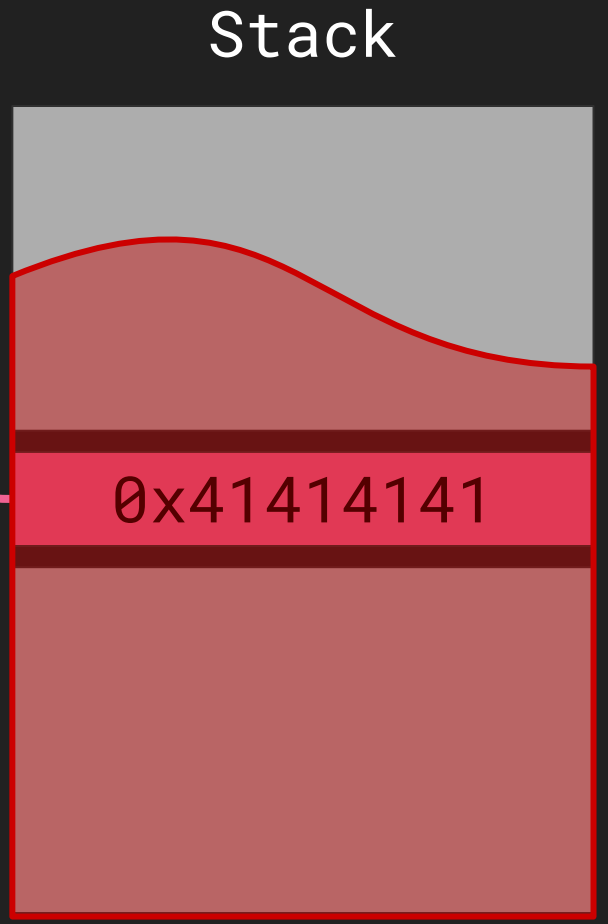
Roadmap


Exploitation Techniques
General Idea
Stats
Unwinding
Real-World Exploits

```
void foo() {
    try {
        bar();
    }
    catch (...) {
        // handle errors
    }
}

void bar() {

    throw new Exception();
}
```



```
void foo() {
    try {
        bar();
    }
    catch (...) {
        // handle errors
    }
}

void bar() {
    // overflow!
    throw new Exception();
}
```



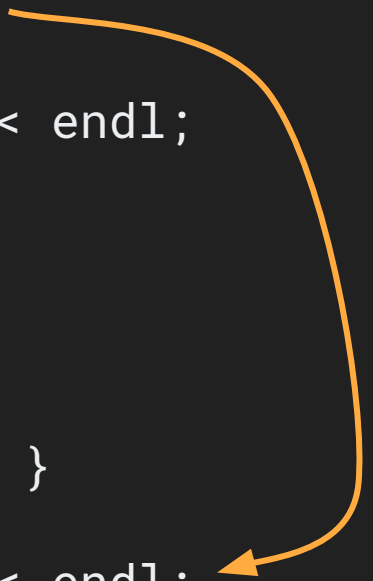
Catch Handler Confusion

```
void foo() {
    try {
        vuln();
    }
    catch (...) { lose(); }
}

void vuln() {
    // overflow
    throw new Exception();
}

void bar() {
    try { /* ... */ }
    catch (...) { win(); }
}
```

Overwrite `return_address` to call site range of `bar()`

What do we control?

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {
        cout << x << endl;
    }
}

void baz() {
    int x = 42;
    try { /* ... */ }
    catch (...) {
        cout << x << endl;
    }
}
```

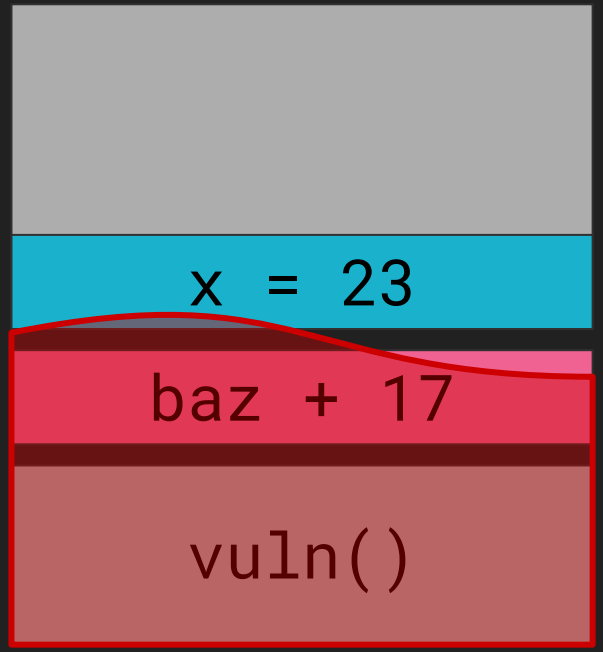


Prints 23

What does it print now?

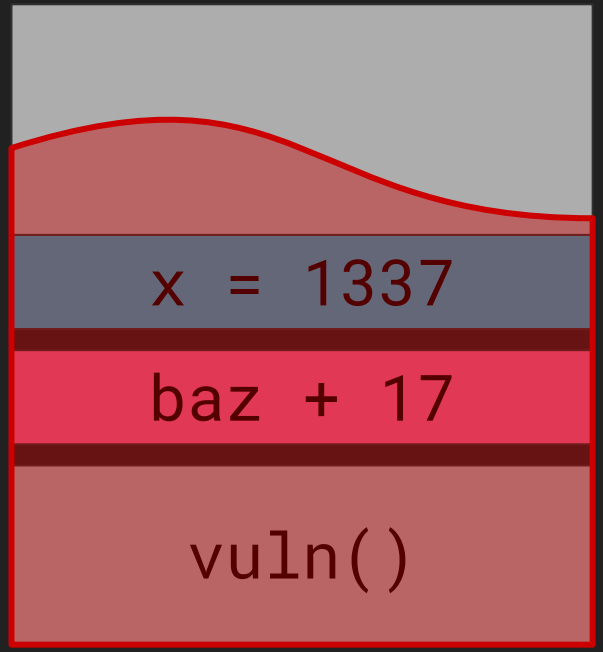
... still 23

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {
        cout << x << endl;
    }
}

void baz() {
    int x = 42;
    try { /* ... */ }
    catch (...) {
        cout << x << endl;
    }
}
```



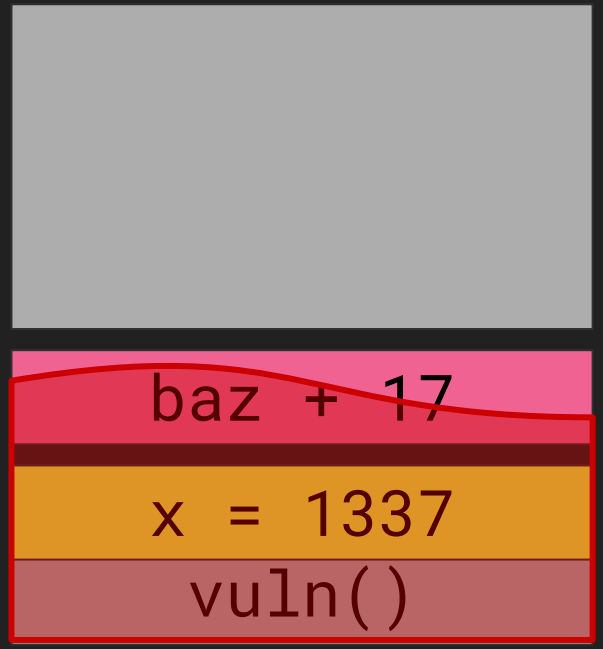
x gets restored from the stack!

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {
        cout << x << endl;
    }
}

void baz() {
    int x = 42;
    try { /* ... */ }
    catch (...) {
        cout << x << endl;
    }
}
```



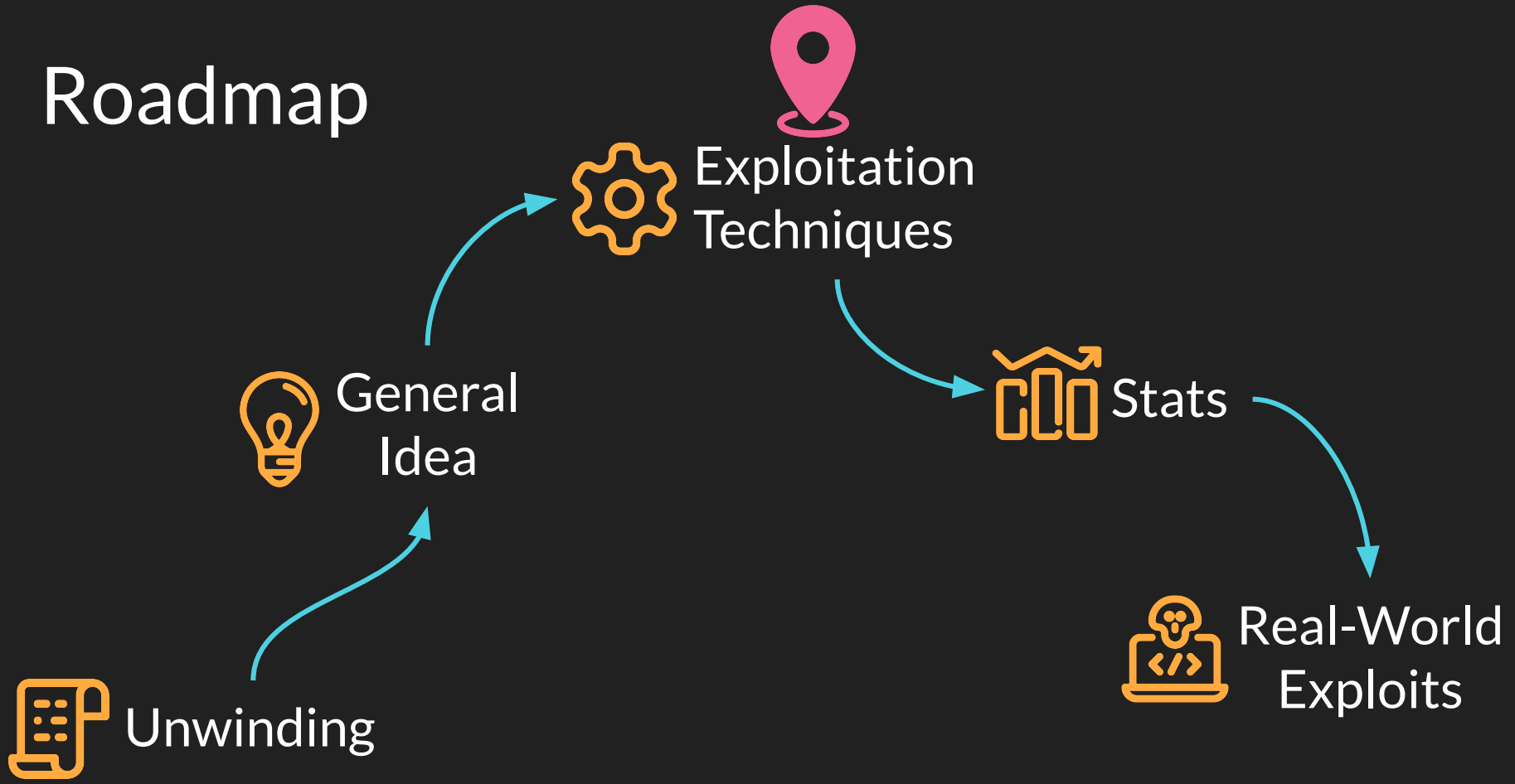
Attackers control the target handler's local variables!

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {
        cout << x << endl;
    }
}

void baz() {
    int x = 42;
    try { /* ... */ }
    catch (...) {
        cout << x << endl;
    }
}
```

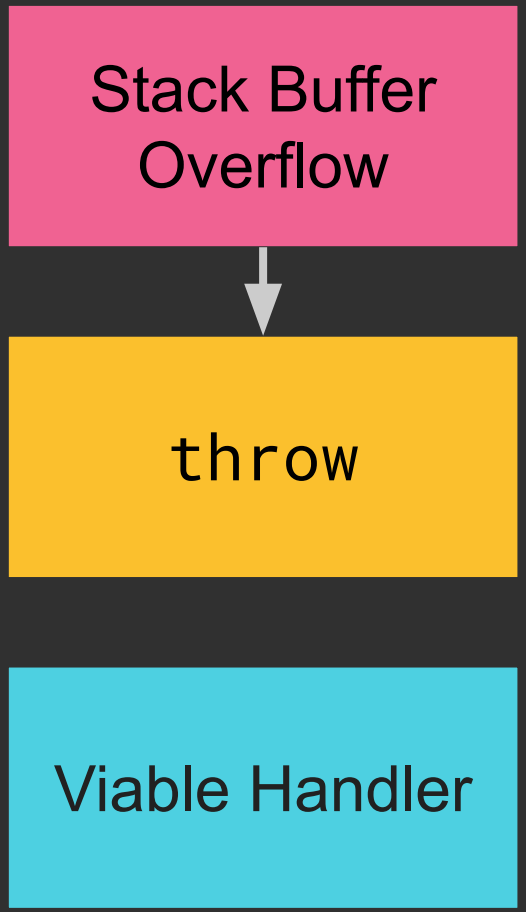


Sometimes they are even stored in callee-saved regs!

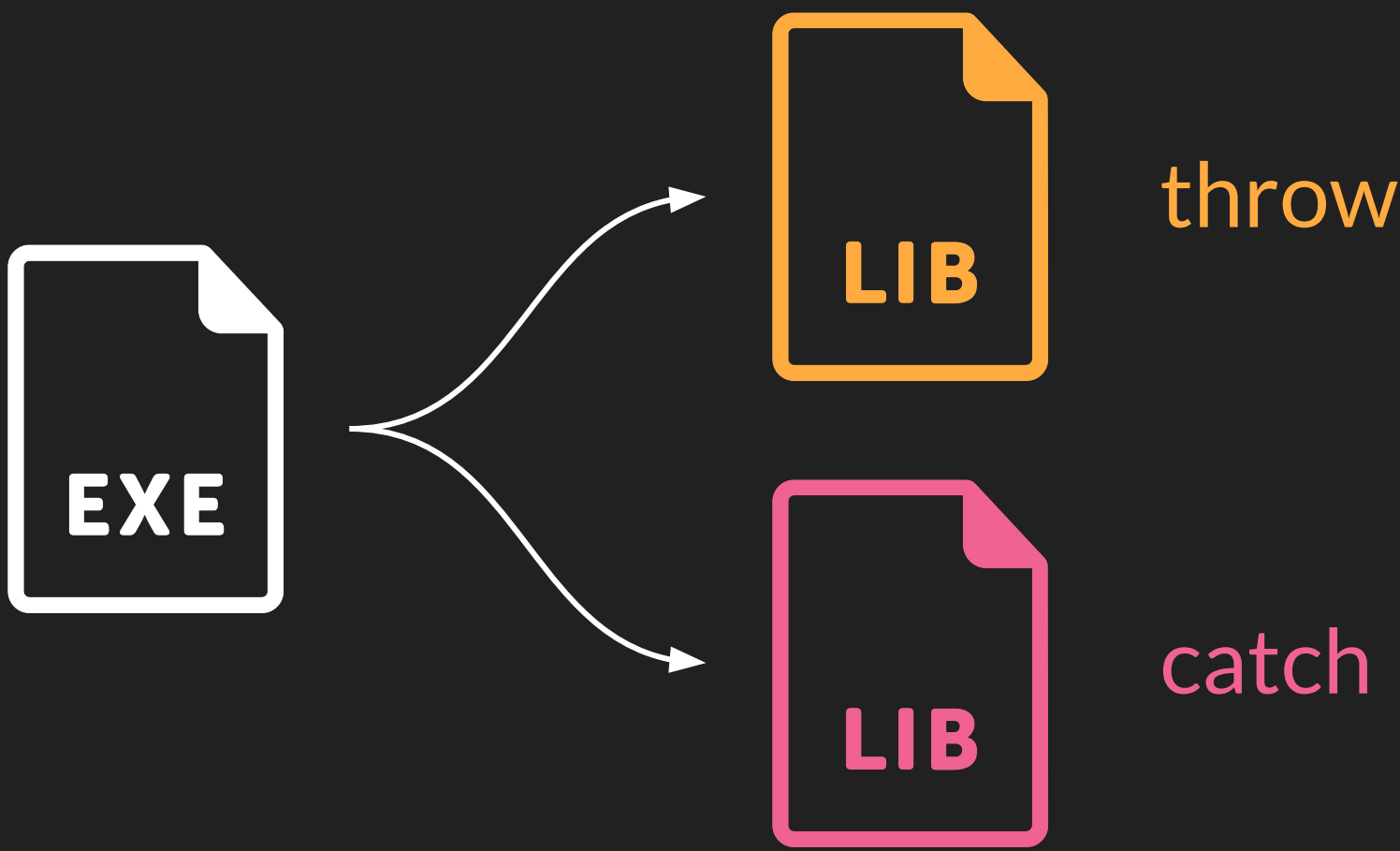
Roadmap


Exploitation Techniques
Stats
Real-World Exploits
General Idea
Unwinding

The Attack



Are there interesting handlers we can reach?


EXE
LIB
LIB
throw
catch

Golden Gadget in `libstdc++`

```
void __cxa_call_unexpected (void *exc_obj_in) {
    xh_terminate_handler = xh->terminateHandler;
    try { /* ... */ }
    catch (...) {
        __terminate(xh_terminate_handler);
    }
}

void __terminate (void (*handler)()) throw () {
    /* ... */
    handler();
    std::abort();
}
```

Restored from stack

And then called!

```
bool StackProtector::runOnFunction(Function &Fn) {
    // ...

    if (!RequiresStackProtector())
        return false;

    // ...

    return InsertStackProtectors();
}
```

llvm::StackProtector

```
/// Check whether or not this function needs a stack protector based
/// upon the stack protector level.
///
/// We use two heuristics: a standard (ssp) and strong (sspstrong).
/// The standard heuristic which will add a guard variable to functions that
/// call alloca with a either a variable size or a size >= SSPBufferSize,
/// functions with character buffers larger than SSPBufferSize, and functions
/// with aggregates containing character buffers larger than SSPBufferSize.
The
/// strong heuristic will add guard variables to functions that call alloca
/// regardless of size, functions with any buffer regardless of type and size,
/// functions with aggregates that contain any buffer regardless of type and
/// size, and functions that contain stack-based variables that have had their
/// address taken.
bool StackProtector::RequiresStackProtector() {
```

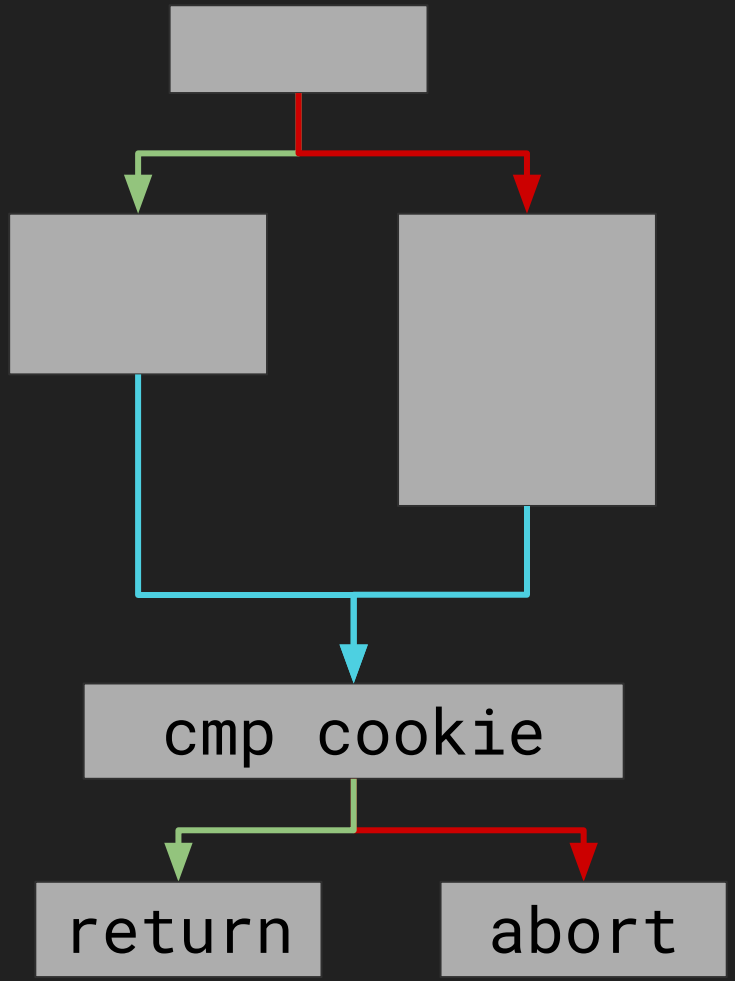
TL;DR

llvm::StackProtector

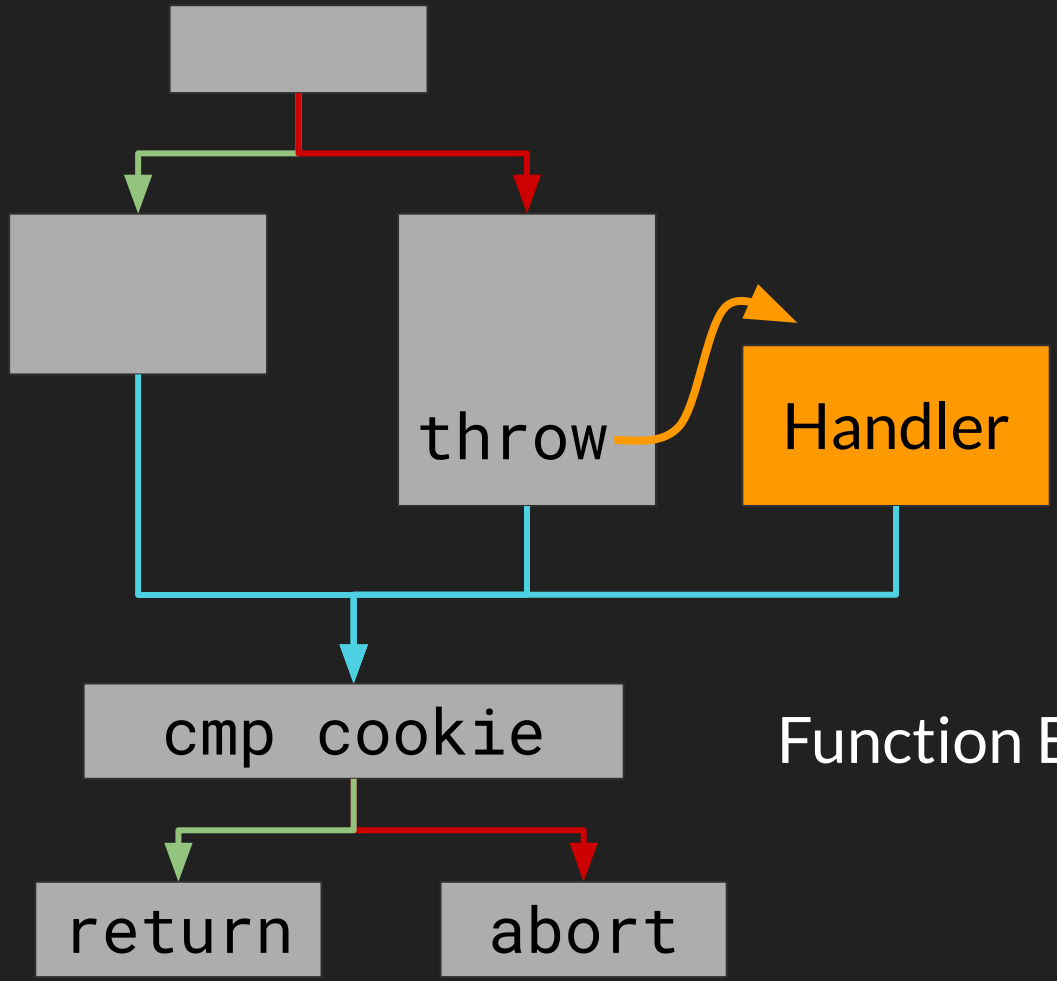
- ➔ Functions that call `alloca()`
- ➔ Functions with any buffer
- ➔ Functions that contain stack-based variables that have had their address taken.

Other functions don’t have stack cookies!

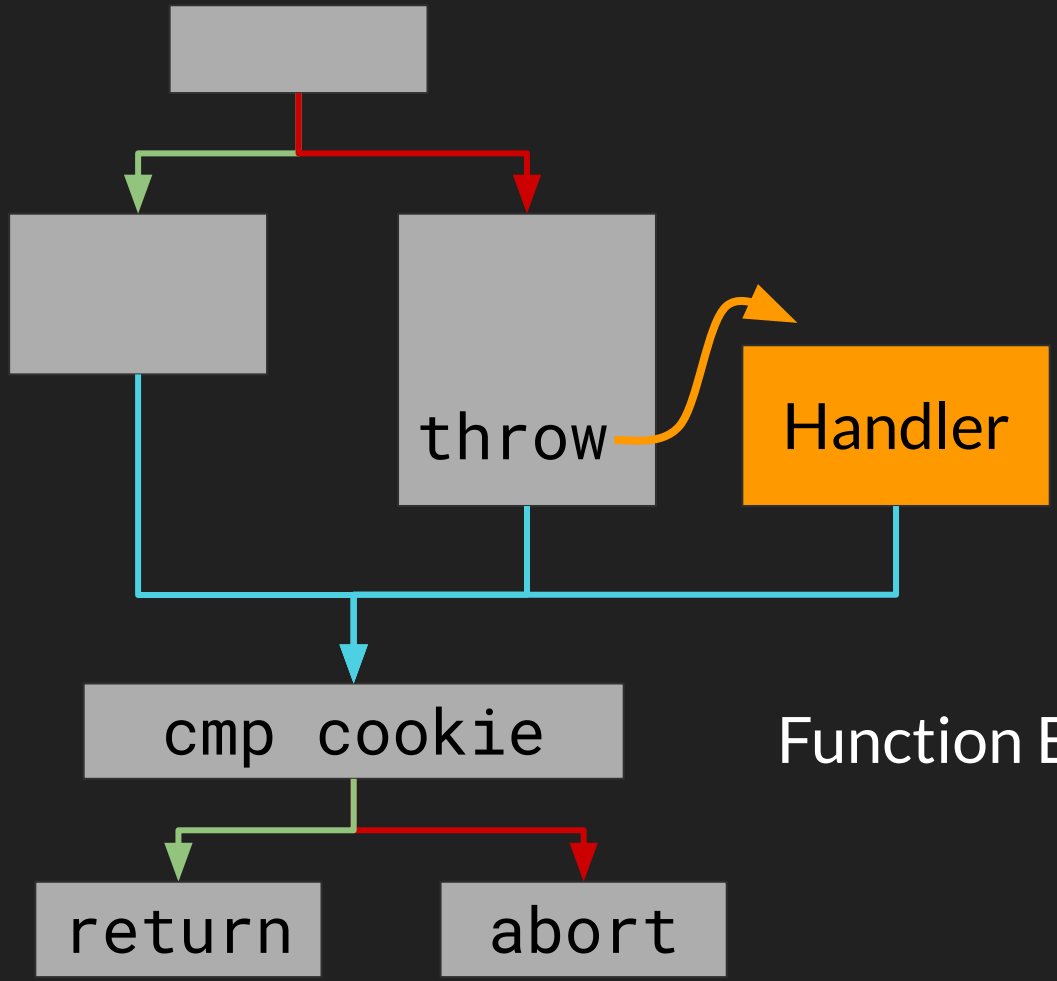
Let’s look at an Example



Function Epilog

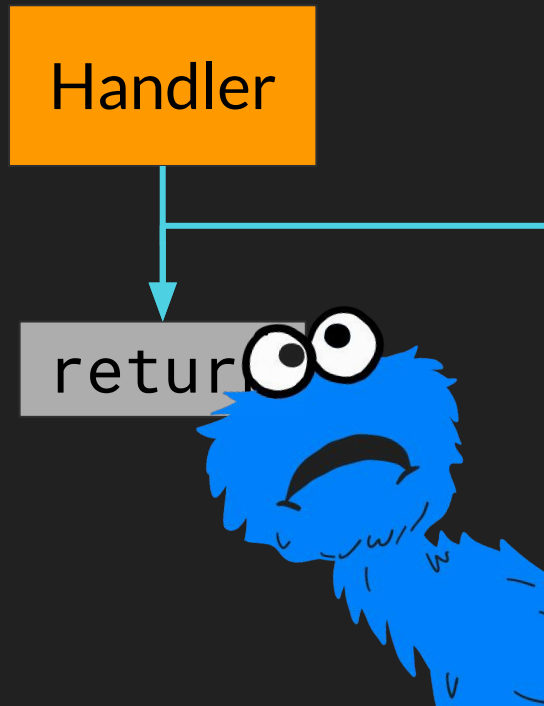


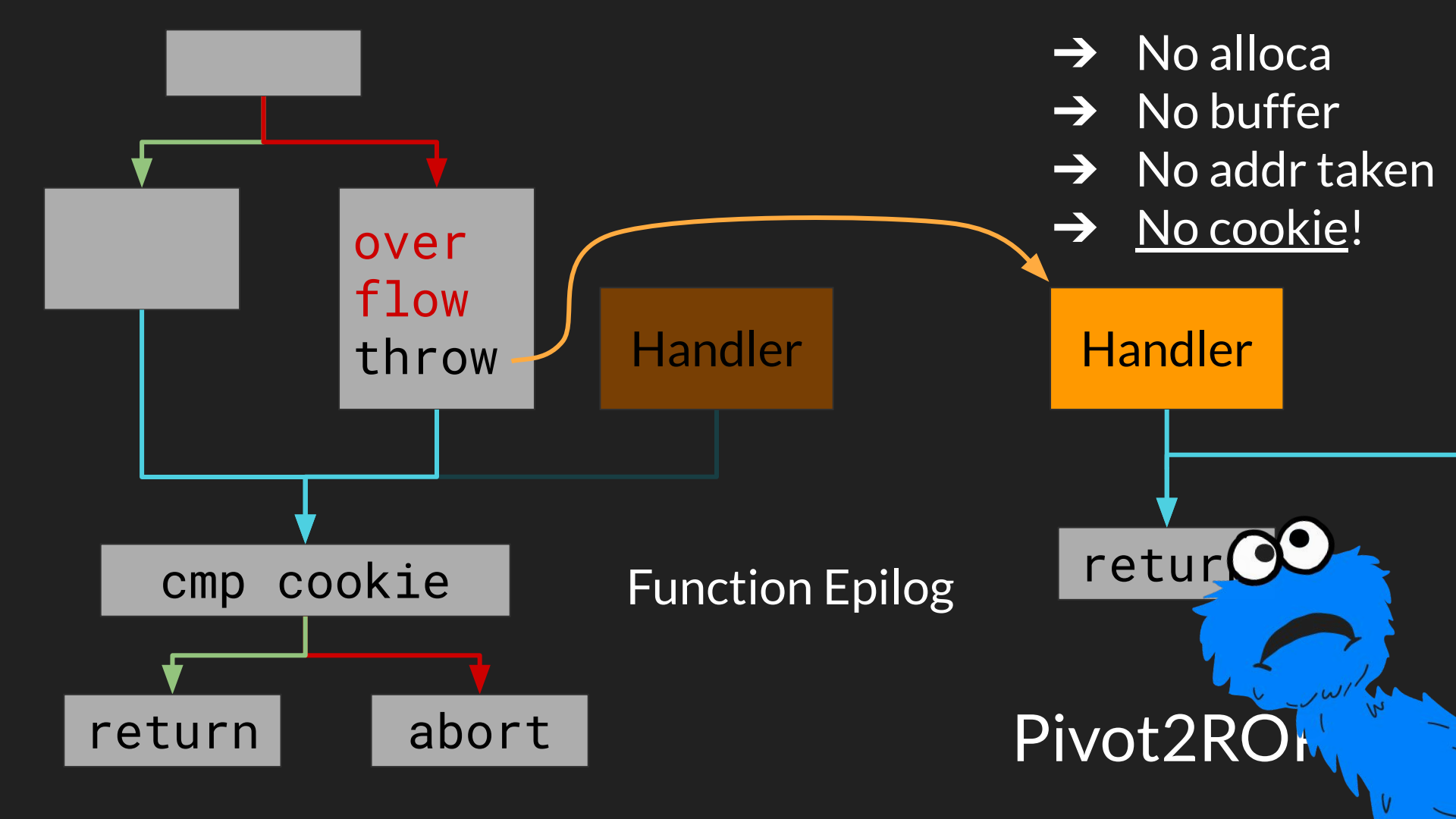
Function Epilog

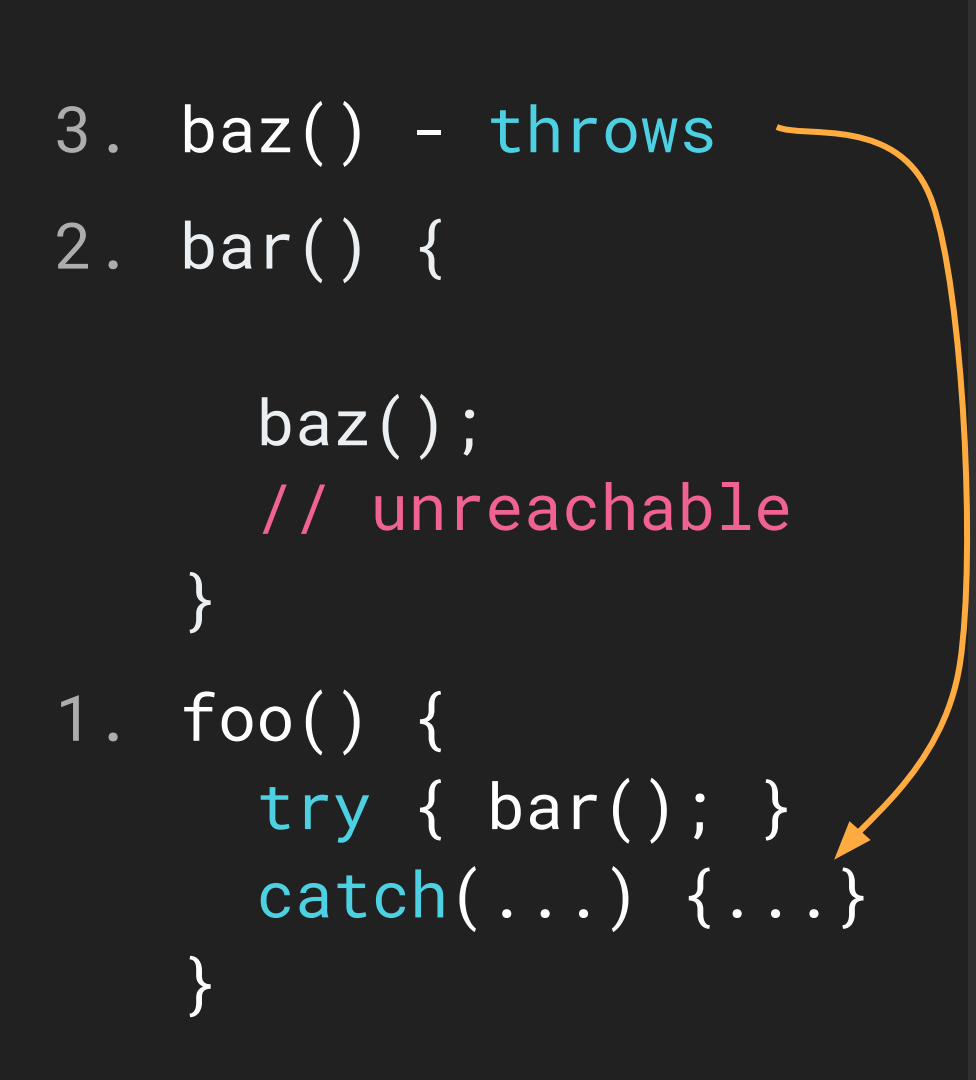


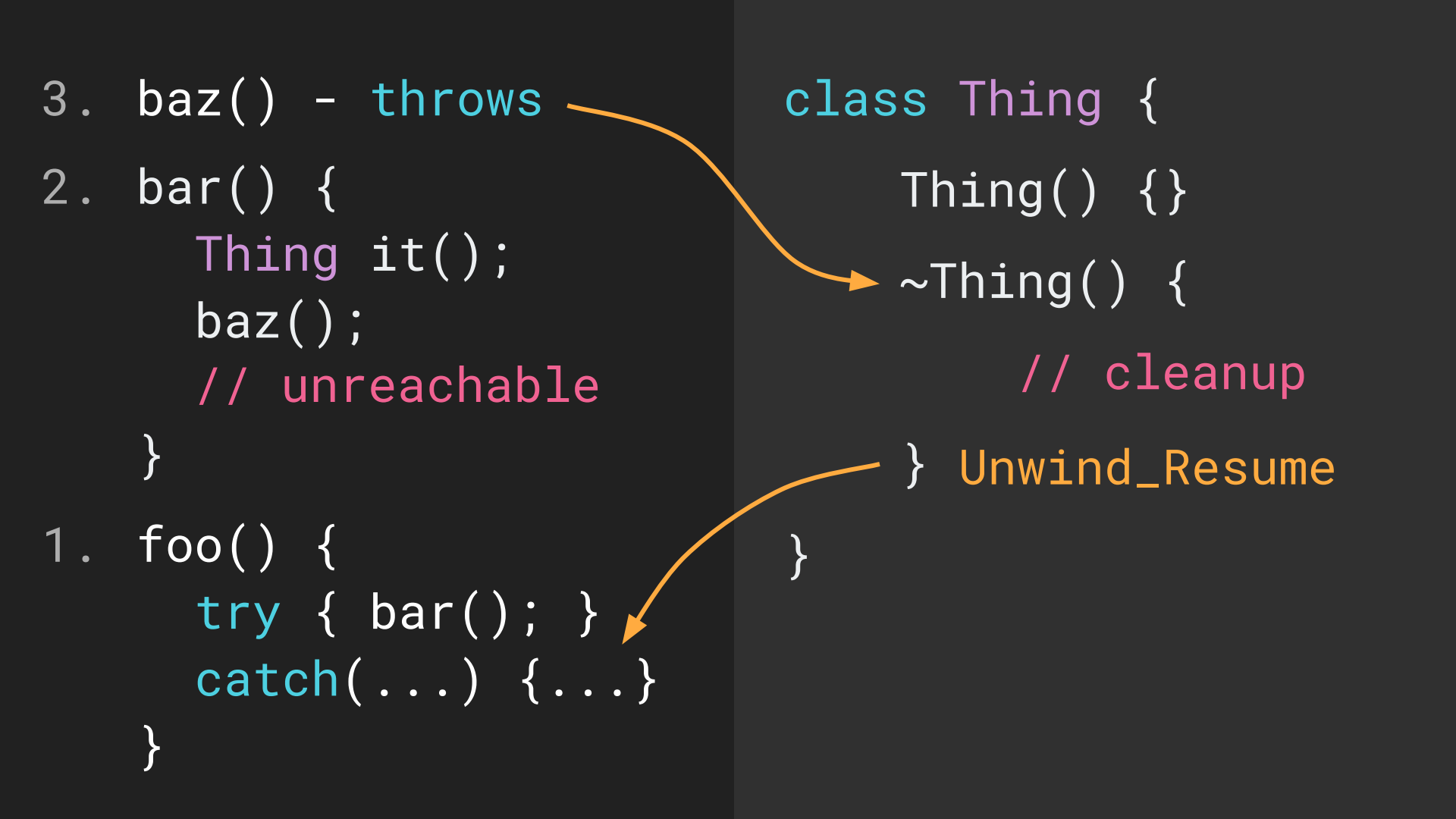
Function Epilog

- ➔ No alloca
- ➔ No buffer
- ➔ No addr taken
- ➔ No cookie!




➔ No alloca
➔ No buffer
➔ No addr taken
➔ No cookie!
over
flow
throw
Handler
Handler
cmp cookie
Function Epilog
return
abort
Pivot2ROP


3. baz() - throws
2. bar() {
baz();
// unreachable
}
1. foo() {
try { bar(); }
catch(...) {...}
}


3. baz() - throws
2. bar() {
Thing it();
baz();
// unreachable
}
1. foo() {
try { bar(); }
catch(...) {...}
}
class Thing {
Thing() {}
~Thing() {
// cleanup
} Unwind_Resume
}

Cleanup Handlers

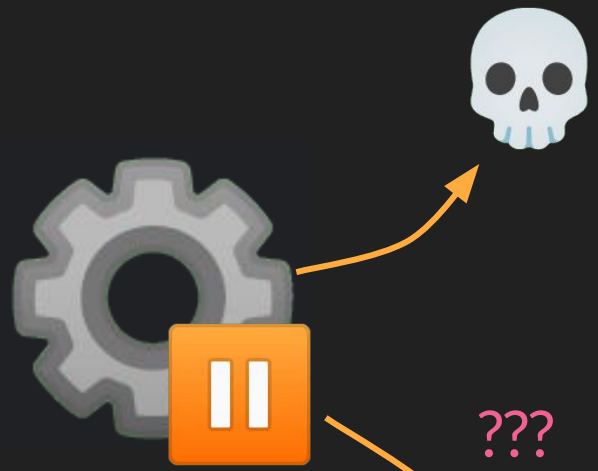
- Often call `free()`
- Can be chained (similar to ROP)

```
// Cleanup exception handler for function
// std::codecvt<char, char, __mbstate_t>::~codecvt()
// @0xbb9e0
// LSDA: 0x2103f1
// callsite: 0xbb9ed - 0xbb9f2

int64_t sub_bba00(struct _Unwind_Exception* arg1 @ rax,
    void* arg2 @ rbx) __noreturn
```

```
operator delete(arg2)
_Unwind_Resume(arg1)
noreturn
```

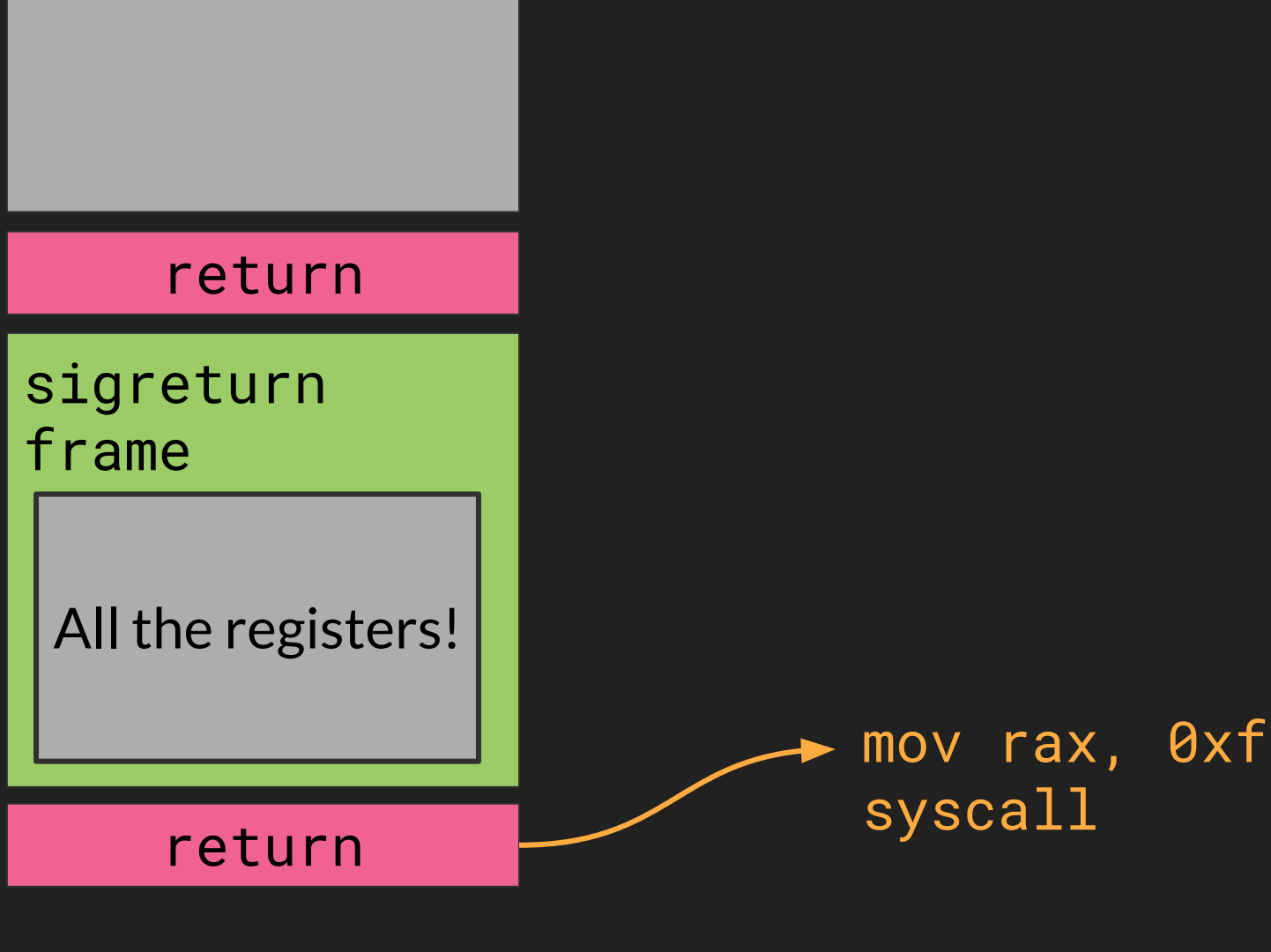
libstdc++

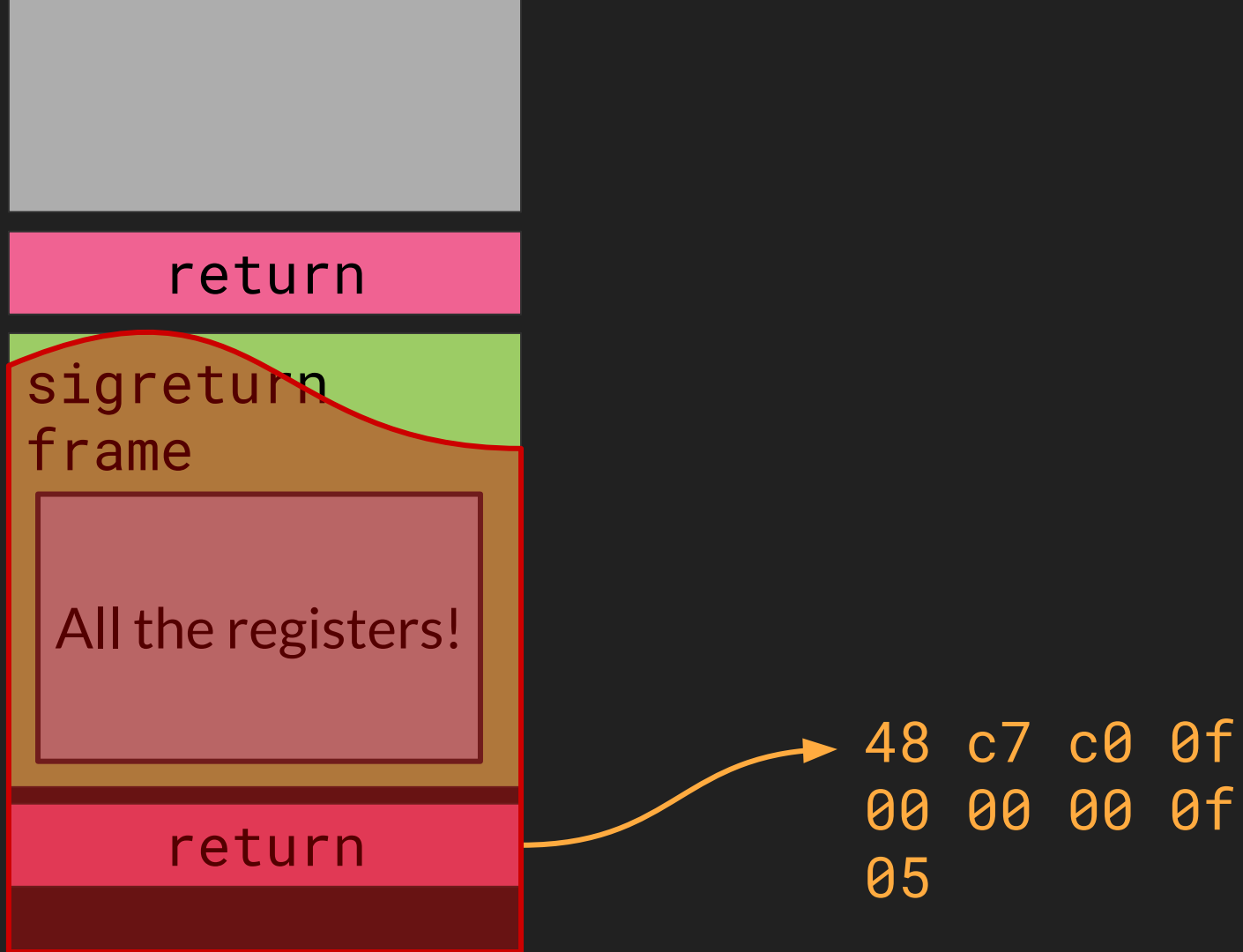
```
kill -SIGUSR1 $pid
```

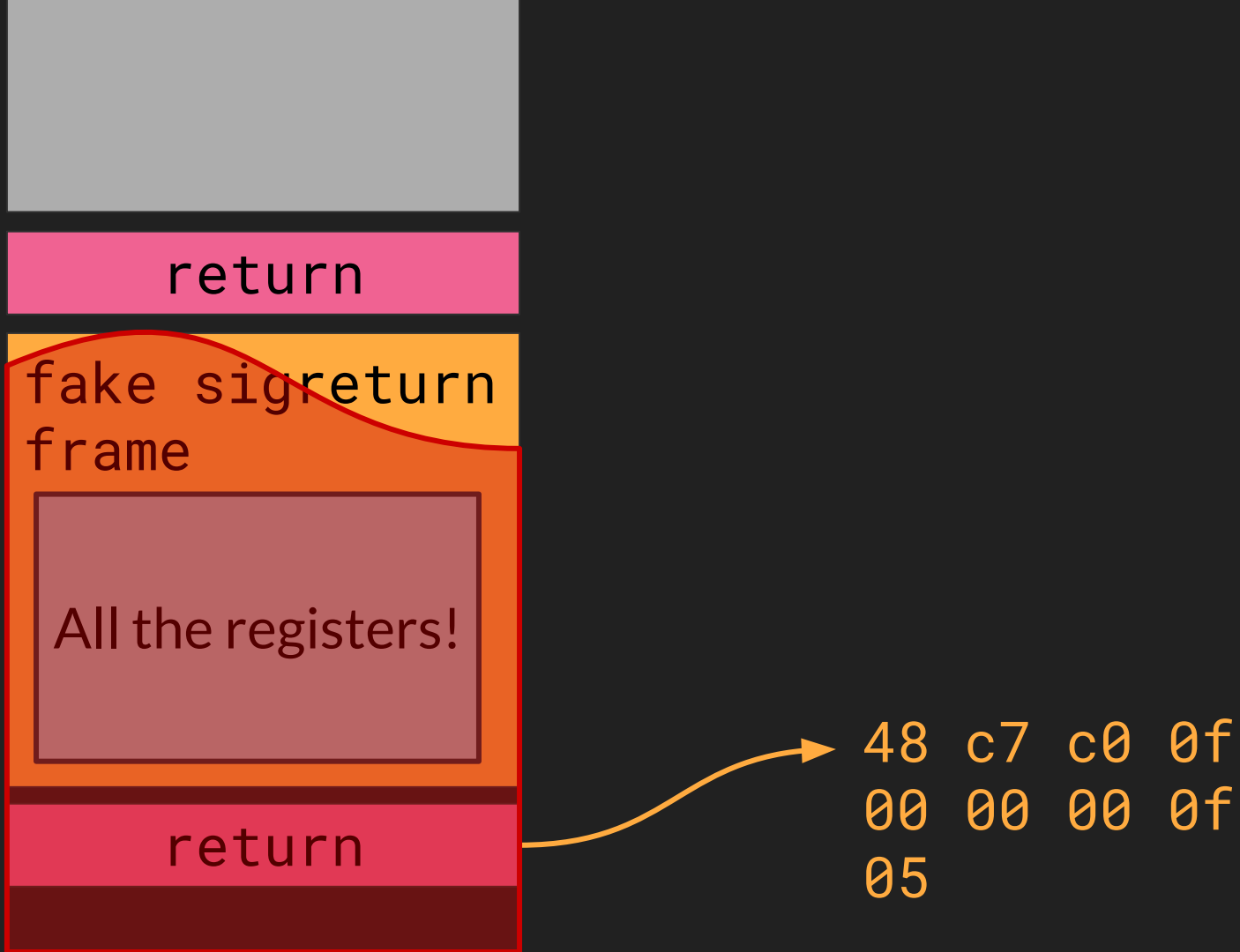


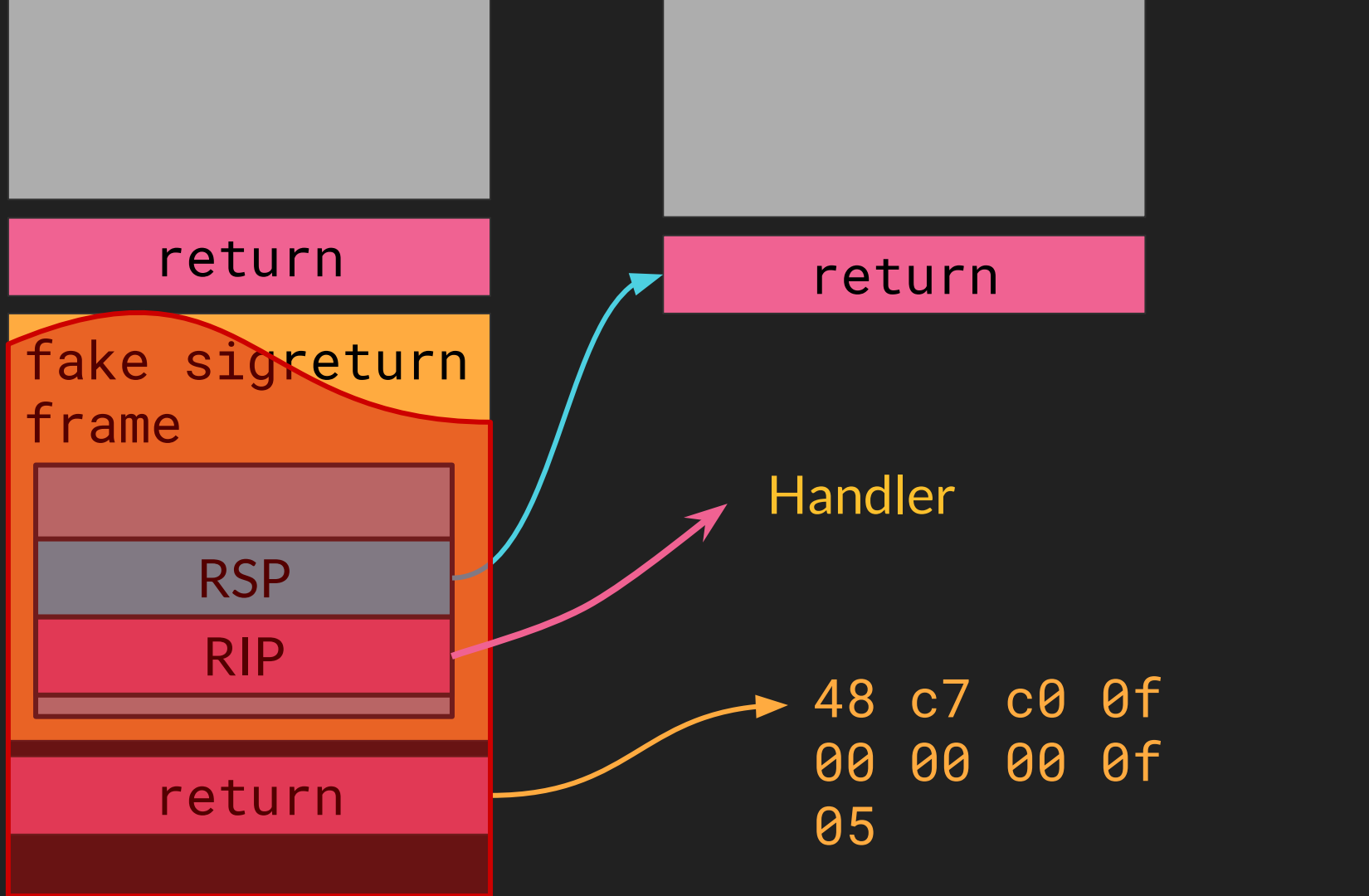
???

Signal Handler


return
sigreturn
frame
All the registers!
return
mov rax, 0xf
syscall


return
sigreturn
frame
All the registers!
return
48 c7 c0 0f
00 00 00 0f
05


return
fake sigreturn frame
All the registers!
return
48 c7 c0 0f
00 00 00 0f
05

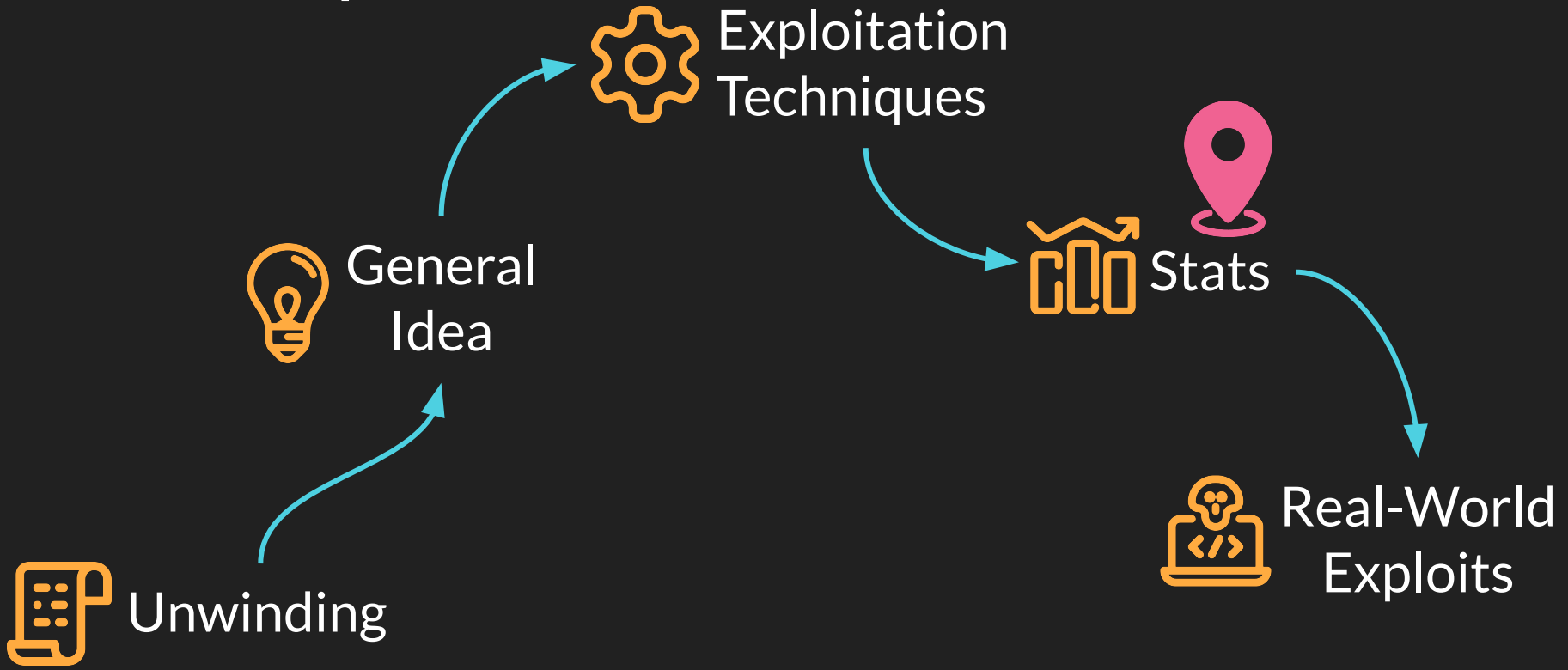

return
return
fake sigreturn frame
RSP
RIP
Handler
return
48 c7 c0 0f
00 00 00 0f
05

Recap: Techniques

- ➔ Divert control flow by confusing Catch Handlers
- ➔ Evade Stack cookies and Pivot to ROP
- ➔ Groom the heap by chaining Cleanup Handlers
- ➔ Craft signal frames to pivot the stack

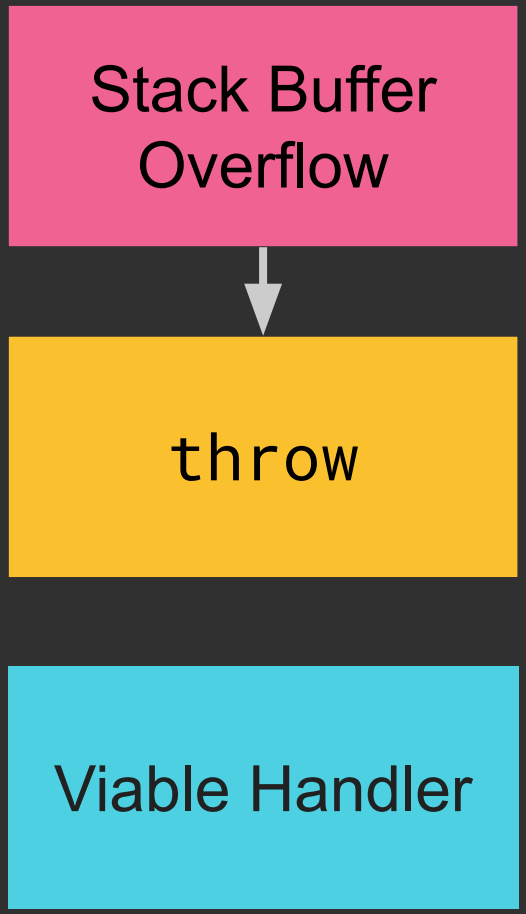
Does this even happen?

Roadmap

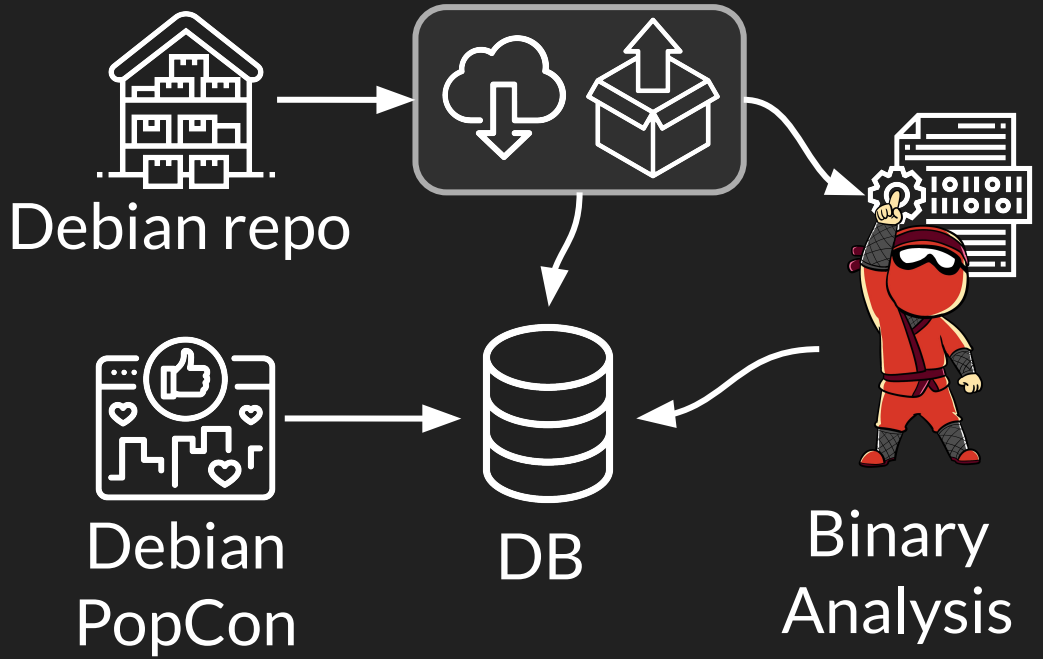

Exploitation Techniques
Stats
Real-World Exploits
General Idea
Unwinding

Preconditions

Let’s recap



Select top 1000 popular packages (~3.3k binaries)

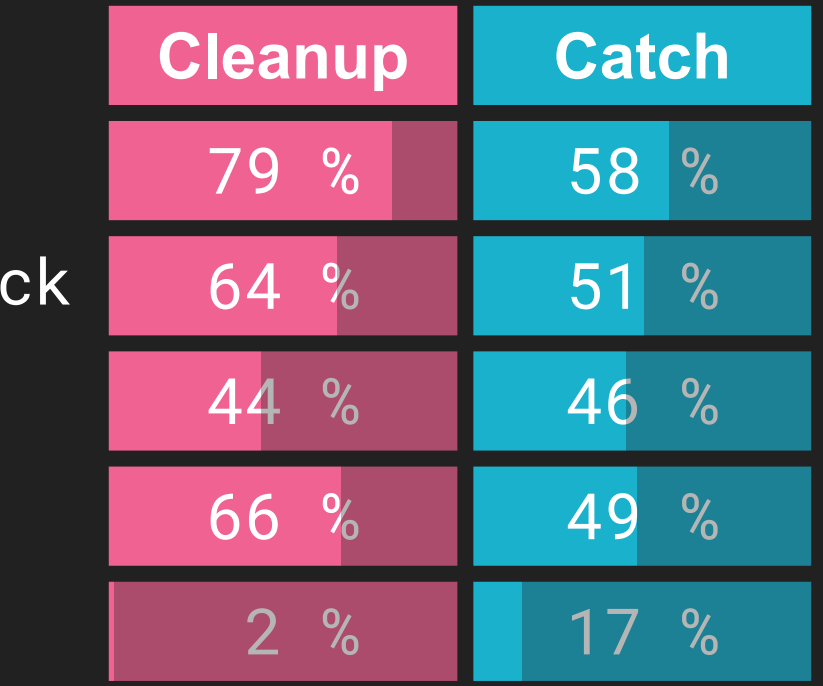


~10% of Binaries
use exception handling

Half contain at least 40%
throwing functions

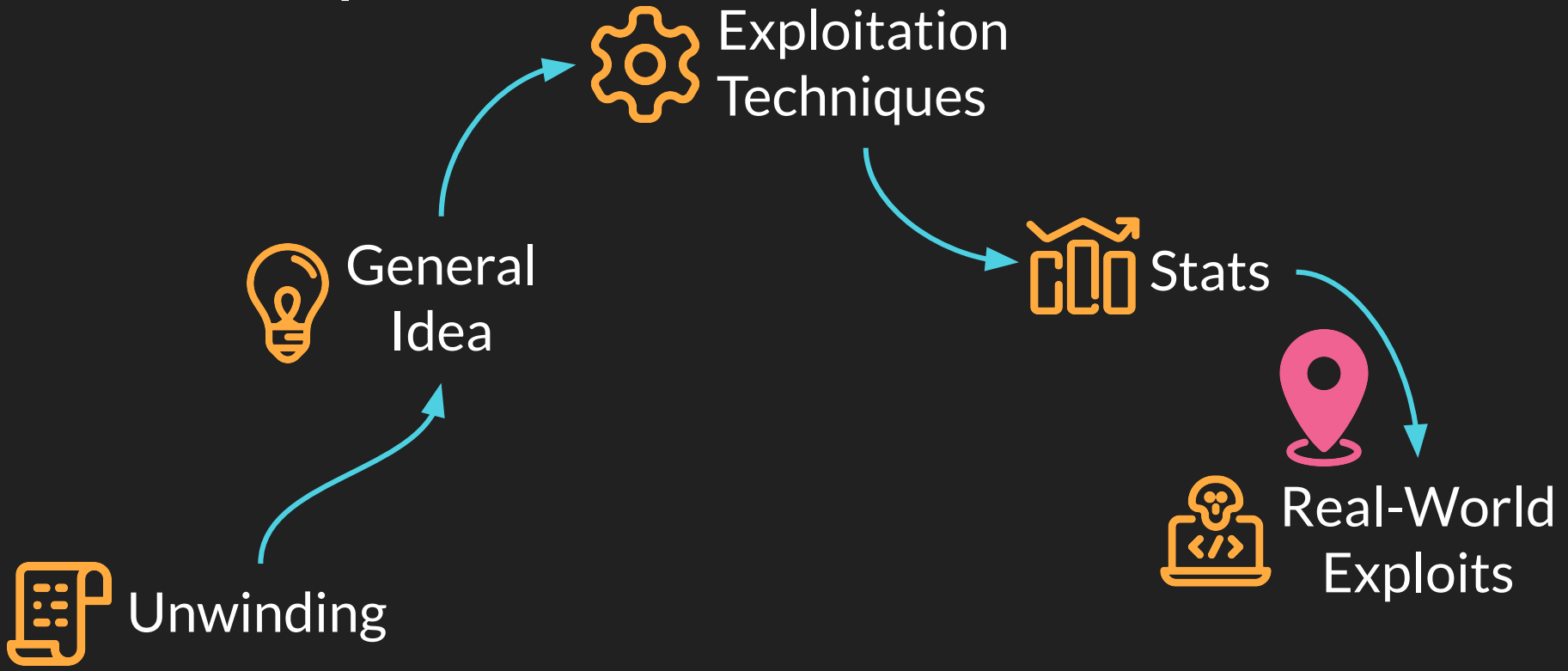
Interesting gadgets

- ➔ `Arbitrary Free`
- ➔ `Control-flow Hijack`
- ➔ `Write-What-Where`
- ➔ `Write-Where`
- ➔ `Write-What`



% of binaries containing at least one gadget

Roadmap


Exploitation Techniques
Stats
Real-World Exploits
General Idea
Unwinding

Snap Back to Reality

Exploiting
Known Bugs

- PowerDNS
  CVE-2009-4009
- SmartCardServices
  CVE-2018-4300
- LibRaw
  CVE-2018-5809

Snap Back to Reality

Exploiting
Known Bugs

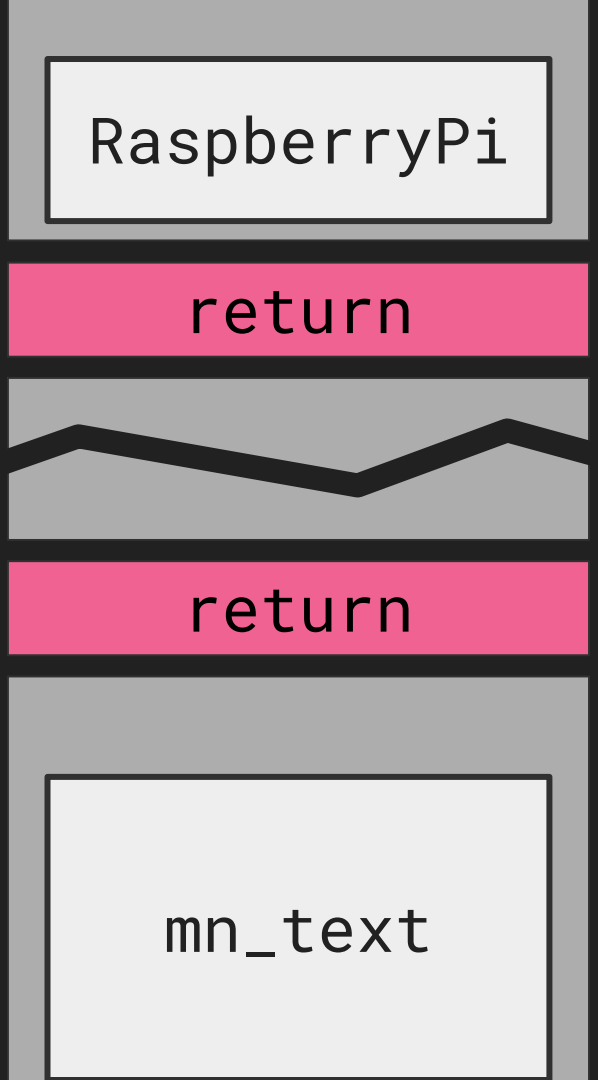
- PowerDNS
  CVE-2009-4009
- SmartCardServices
  CVE-2018-4300
- LibRaw
  CVE-2018-5809

CVE-2018-5809

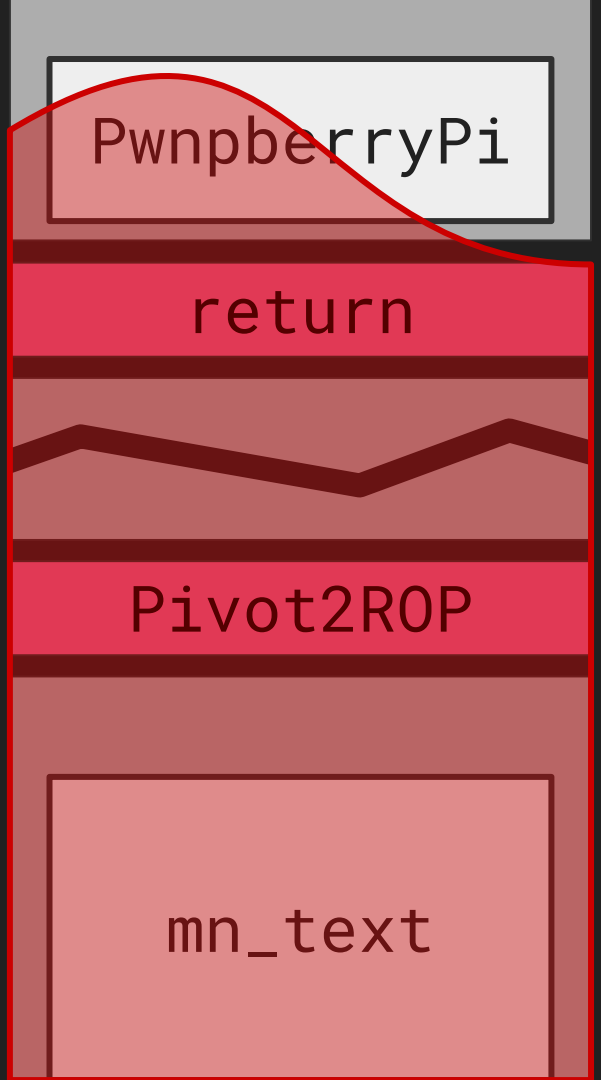
```
parse_exif (int base) {
  // ...
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    // ...
    switch (tag) {
    // ...
    case 37500:                                  // tag 0x927c
      if ((!strncmp(make, "RaspberryPi",11)) &&
          (!strncmp(model, "RP_imx219",9))))  // and others
        char mn_text[512];
        // ...
        fgets(mn_text, len, ifp);
        // no throw :(
      }
```

CVE-2018-5809

```
parse_exif (int base) {
  // ...
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    // ...
    switch (tag) {
    // ...
    case 37500:                                  // tag 0x927c
      if ((!strncmp(make, "RaspberryPi",11)) &&
          (!strncmp(model, "RP_imx219",9))))  // and others
        char mn_text[512];
        // ...
        fgets(mn_text, len, ifp);
        // no throw :(
      }
      else
        parse_makernote (base, 0);
```



parse_exif



Now we can trigger `throw`!

`one_gadget`
`/bin/sh`

`parse_exif`

```
// Exception handler for function LibRaw::x3f_thumb_size() @0x71970
// LSDA: 0x8b608
// callsite: 0x71984 - 0x719b0
// catch clause types
//     0x8b614

000719c1  int64_t sub_719c1(void* arg1 @ rax)

0001c9b6  __cxa_begin_catch(arg1)
0001c9bb  __cxa_end_catch()
000719a4  return -1
```

Demo Time

CVE-2018-5809

```
parse_exif (int base) {
  // ...
  while (entries--) {
    tiff_get (base, &tag, &type, &len, &save);
    // ...
    switch (tag) {
    // ...
    case 37500:                                  // tag 0x927c
      if ((!strncmp(make, "RaspberryPi",11)) &&
          (!strncmp(model, "RP_imx219",9))))  // and others
        char mn_text[512];
        // ...
        fgets(mn_text, len, ifp);
        // no throw :(
      }
      else
        parse_makernote (base, 0);
```

If we corrupt the stack,
we can modify data
to reach `throw`
via unexpected paths!

Recap


Exploitation Techniques
Stats
Real-World Exploits
Recap
General Idea
Unwinding

Thanks!

- The entire research team
- Tasteless
- Gregor Kopf
- Jiska Classen
- The VUSec group

Black Hat Sound Bytes

Key Takeaways

github.com/chop-project/chop

- ➔ Exception Handling is an overlooked attack surface
- ➔ Mitigations are often incomplete
- ➔ Even when preconditions seem unlikely, exploitation is often possible