Let Me Unwind That For You:

Exceptions to Backward-Edge Protection

Victor Duta Fabian Freyer

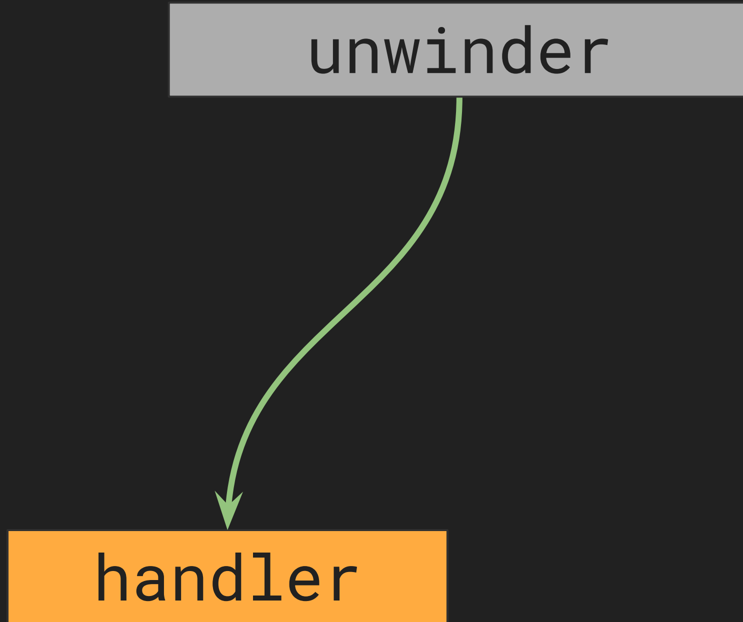
Fabio Pagani Marius Muench Cristiano Giuffrida

Catch handler oriented programming

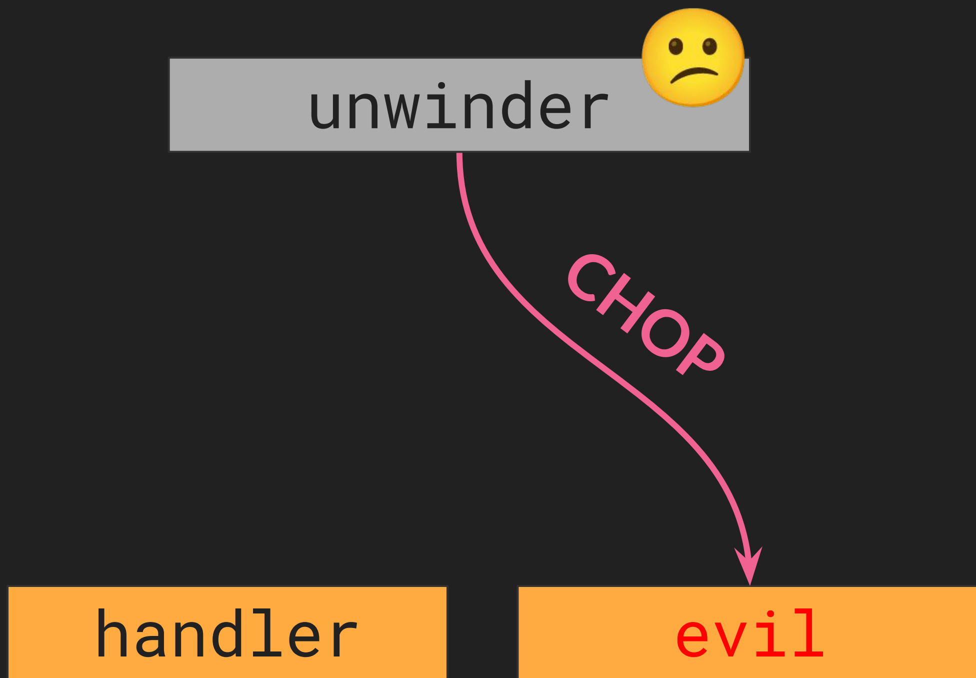
CHOP in a nutshell

unwinder

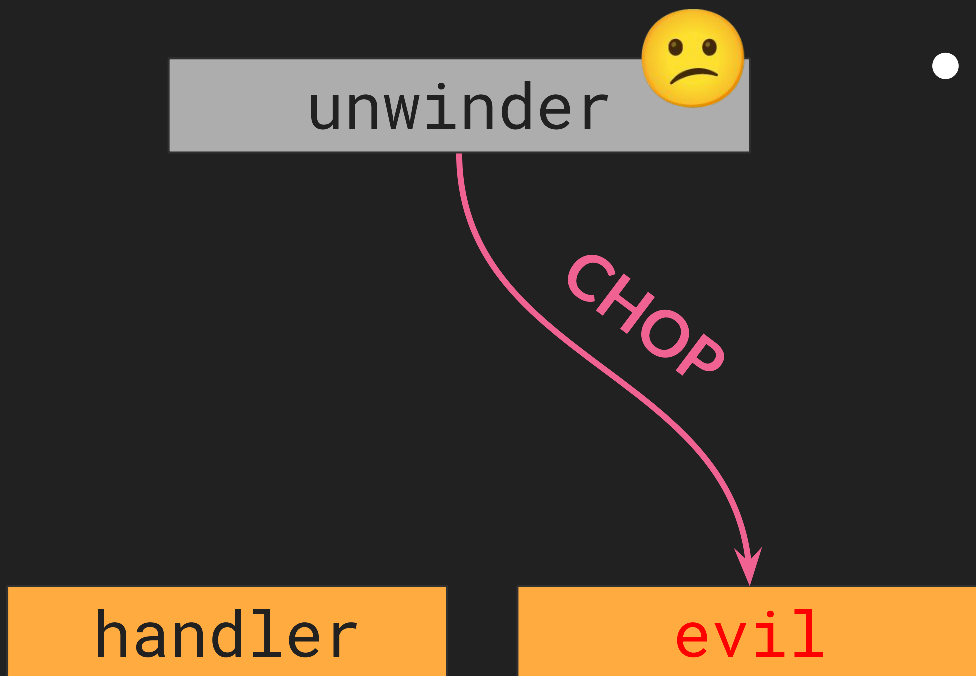
CHOP in a nutshell



CHOP in a nutshell

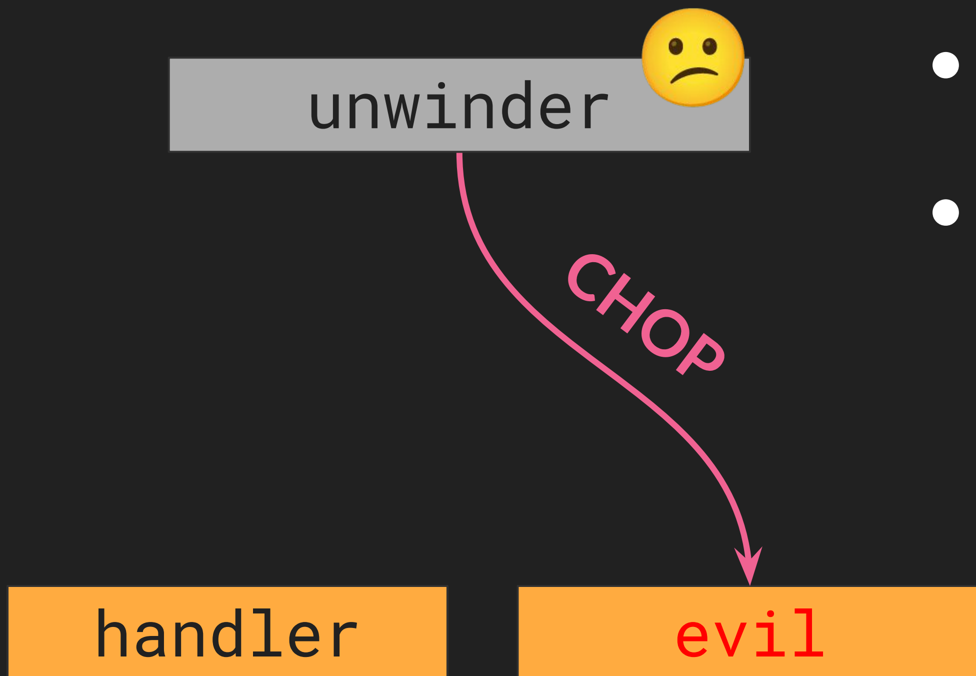


CHOP in a nutshell



- Likelihood of CHOP attacks

CHOP in a nutshell



- Likelihood of CHOP attacks
- Are CHOP attacks a serious issue ?

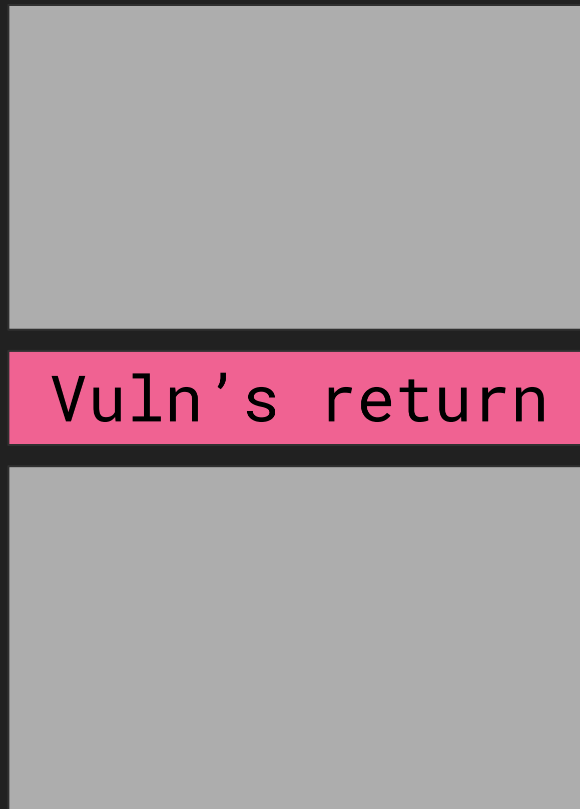
The basic recipe…

The basic recipe...

```
void vuln() {
    // overflow
    // some other code

    return;
}
```

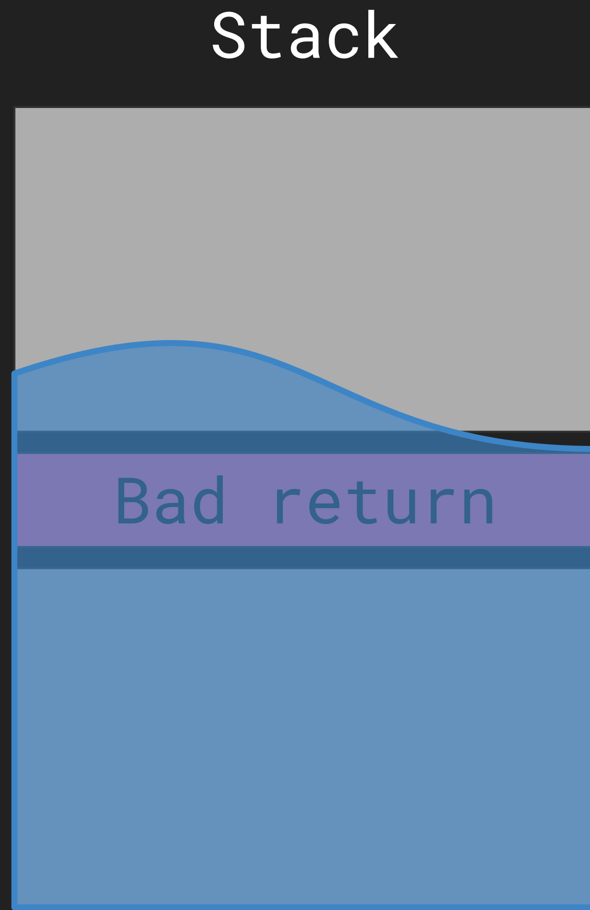
The basic recipe...

```
void vuln() {
    // overflow
    // some other code

    return;
}
```



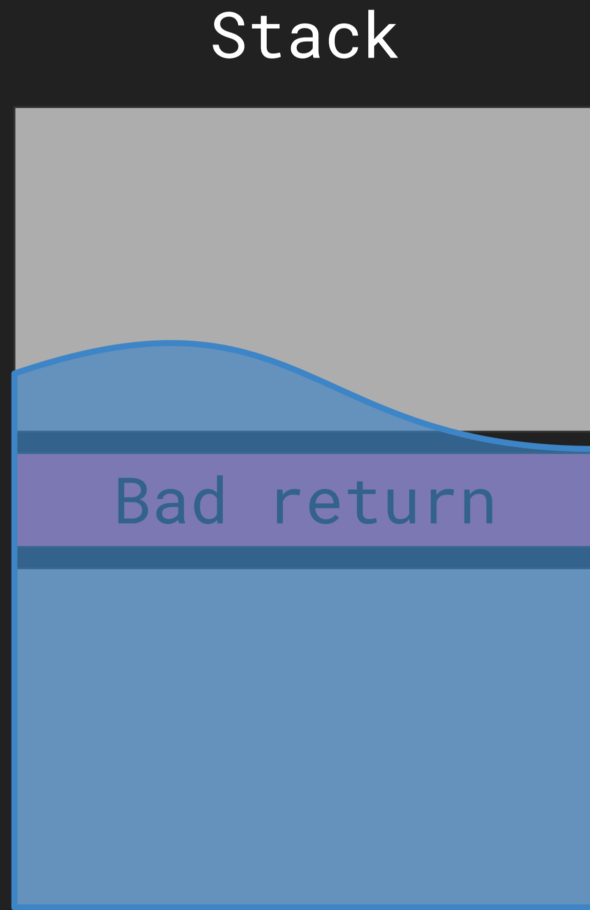
The basic recipe...

```
void vuln() {
    // overflow
    // some other code

    return;
}
```



The basic recipe...

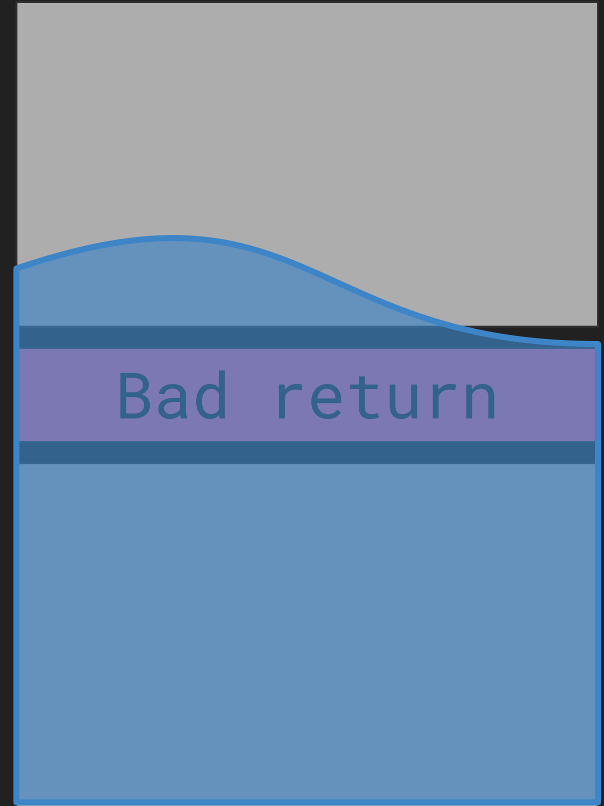
```
void vuln() {
    // overflow
    // some other code
    throw new Exception();
    return;
}
```



The basic recipe...

```
void vuln() {
    // overflow
    // some other code
    throw new Exception();
    return;
}
```

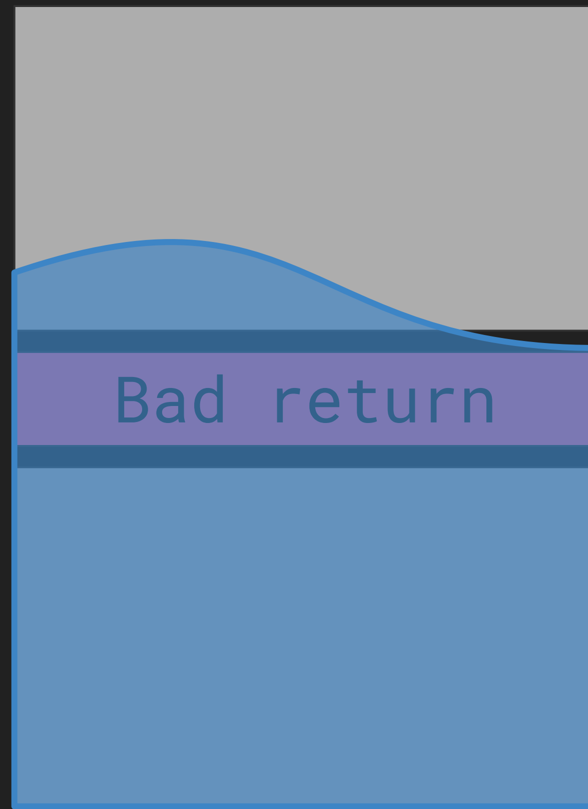
handler



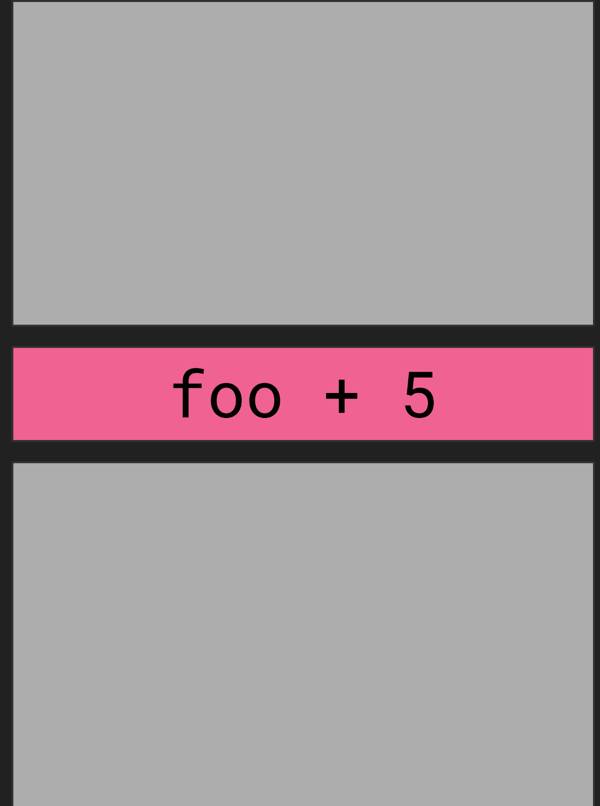
The basic recipe...

```
void vuln() {
    // overflow
    // some other code
    throw new Exception();
    return;
}
```

evil



Catch Handler Confusion


Stack
foo + 5

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

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

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

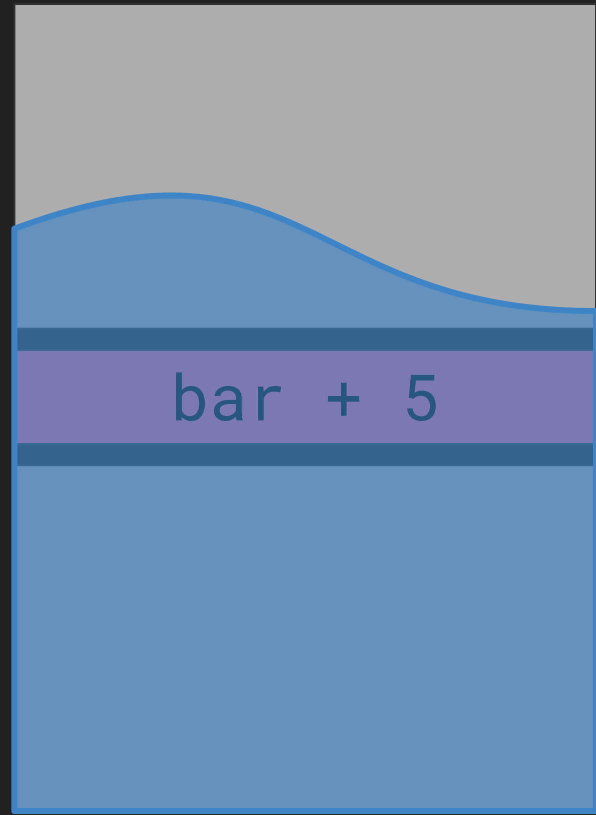
```
void foo() {
    try {
        vuln();
        // more code
    }
    catch (...) { lose(); }
}

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

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

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

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

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

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

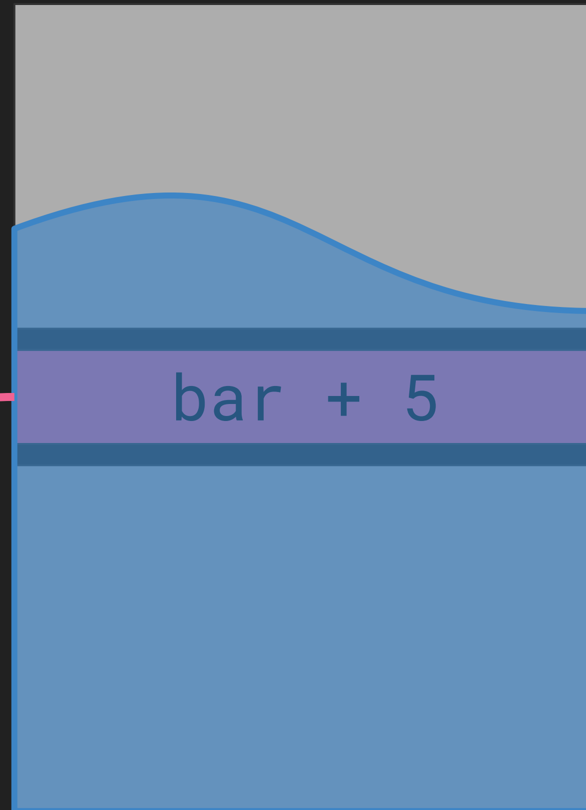
return

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

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

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



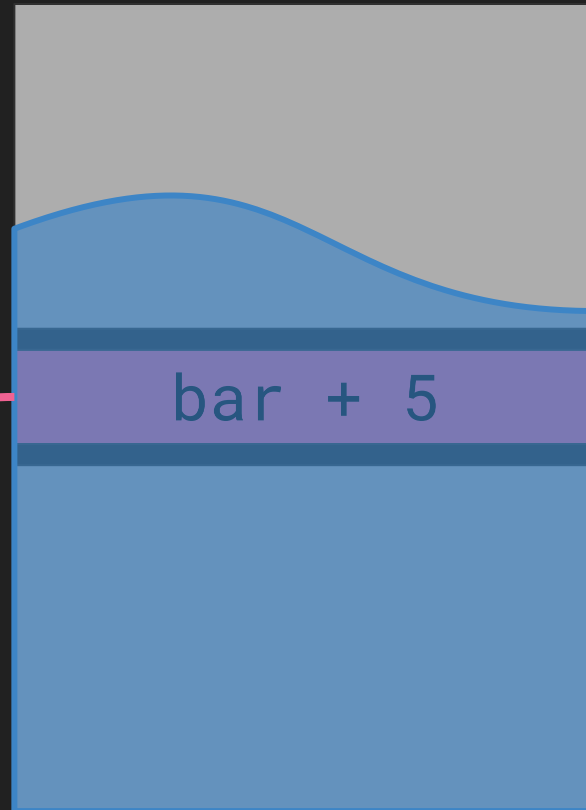
```
void foo() {
    try {
        vuln();
        // more code
    }
    catch (...) { lose(); }
}

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

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

return



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

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

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

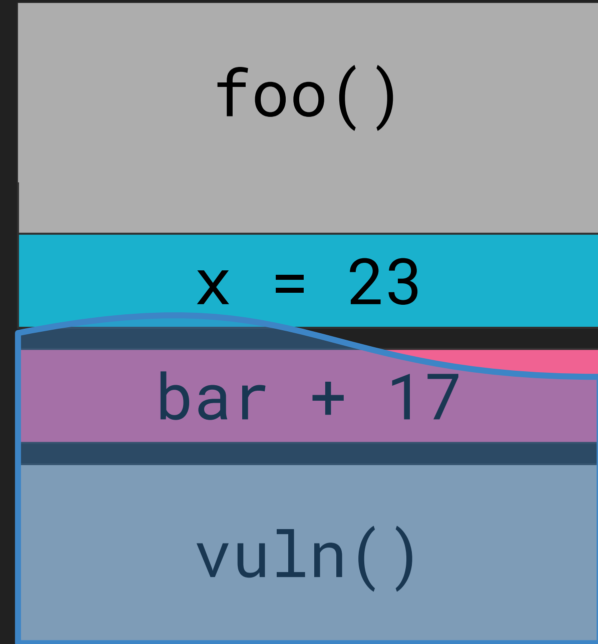
return



What do we control ?

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {/* ... */}
}

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

CHOP

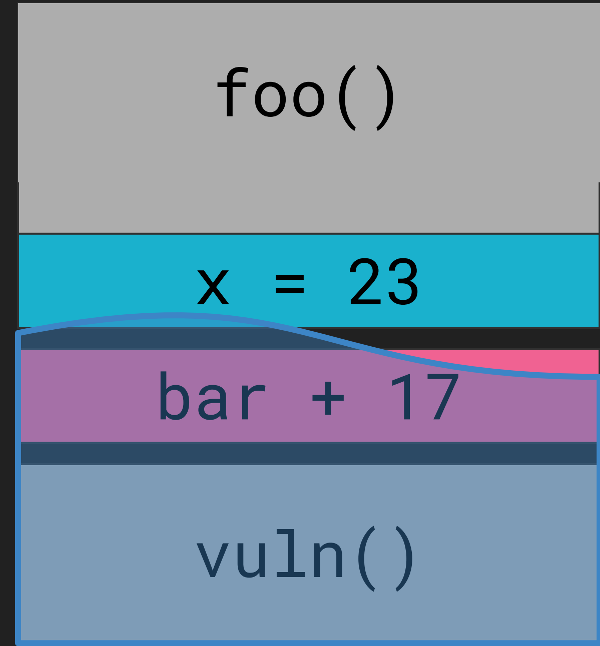


x gets restored from foo's stack frame!

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {/* ... */}
}

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

CHOP

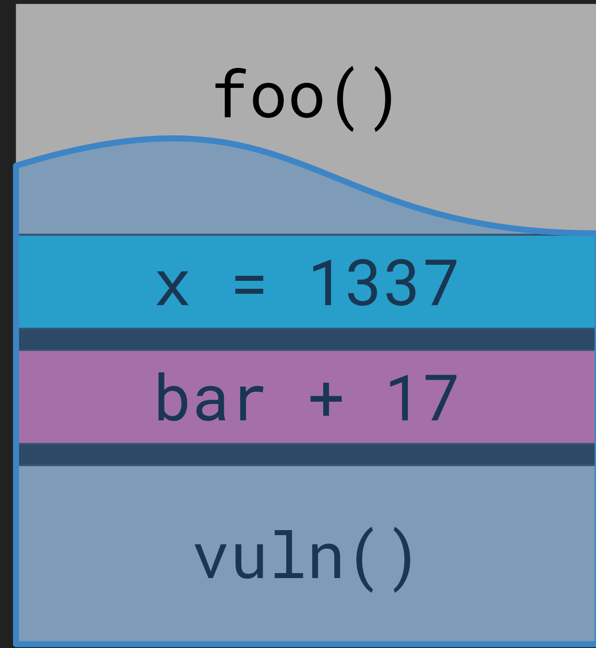
prints 23



x gets restored from foo's stack frame!

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {/* ... */}
}

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

CHOP

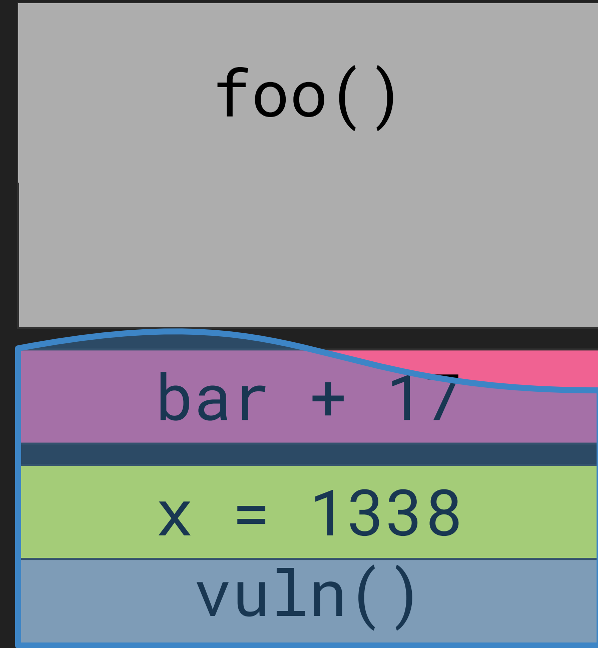


prints 1337

Attackers control the target handler's local variables!

```
void foo() {
    int x = 23;
    try { vuln(); }
    catch (...) {/* ... */}
}

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

CHOP



prints 1338

Sometimes they are even stored in callee-saved regs!

The Attack

The Attack

Stack Buffer Overflow

The Attack

throw

The Attack

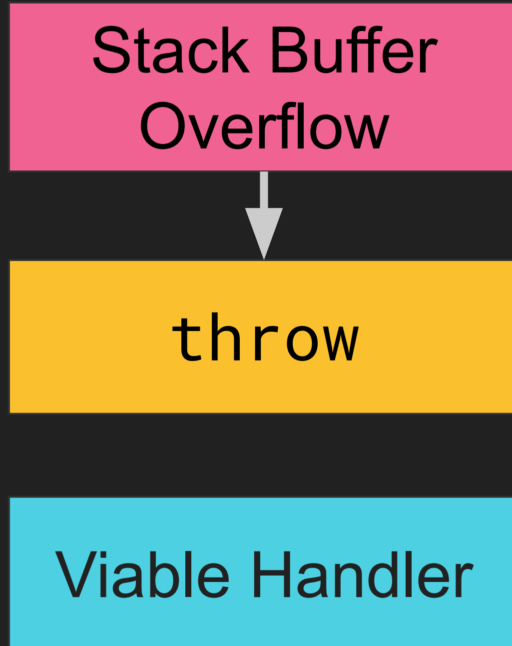
Stack Buffer Overflow

↓

`throw`

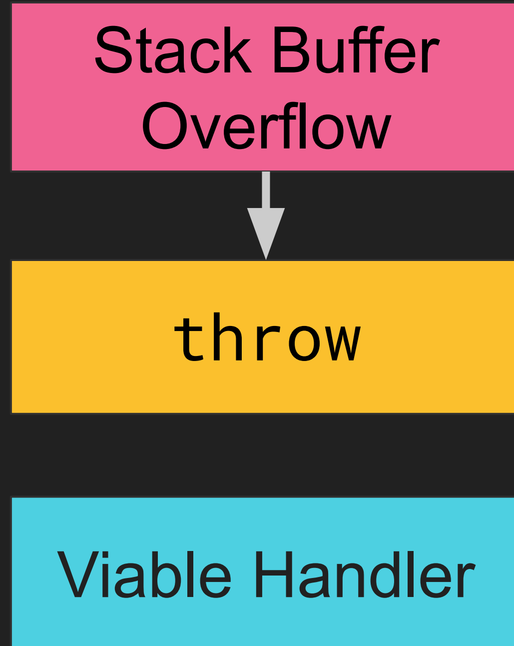
Viable Handler

The Attack



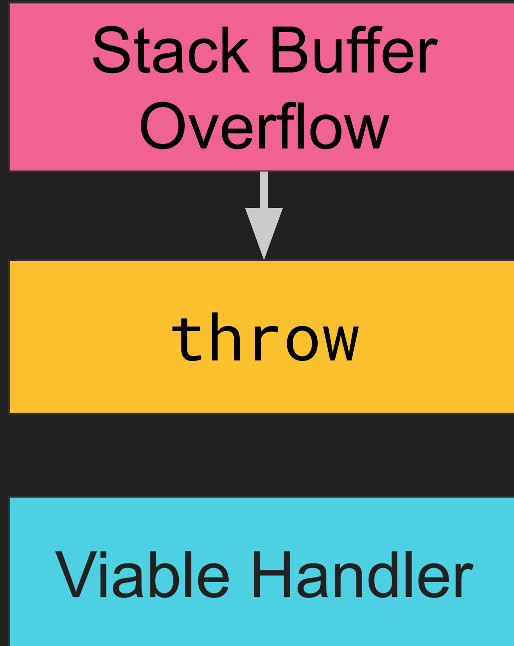
- Prevalence of exception handling code

The Attack

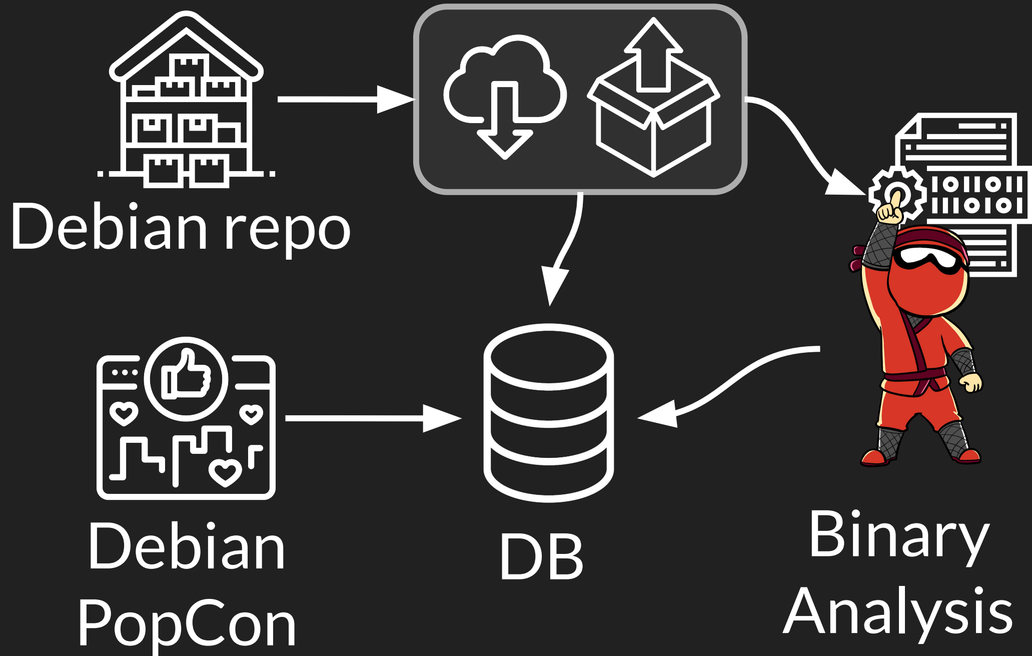


- Prevalence of exception handling code
- Likelihood of throwing functions

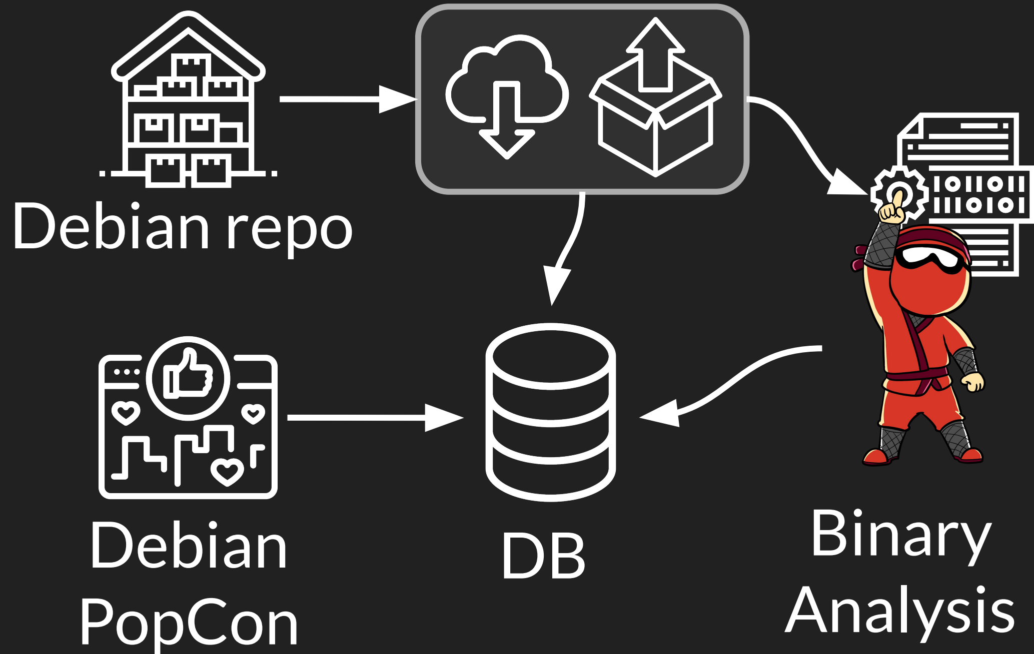
The Attack



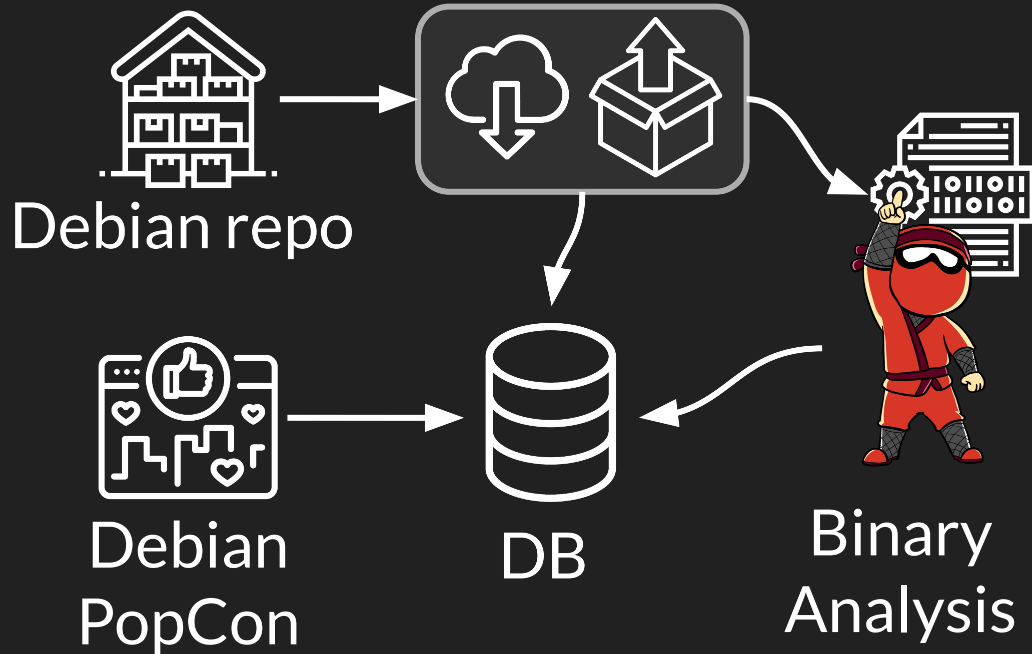
- Prevalence of exception handling code
- Likelihood of throwing functions
- Handlers with meaningful attacker gadgets


Debian repo
Debian
PopCon
DB
Binary
Analysis

Select top 1000 popular packages (~3.3k binaries)

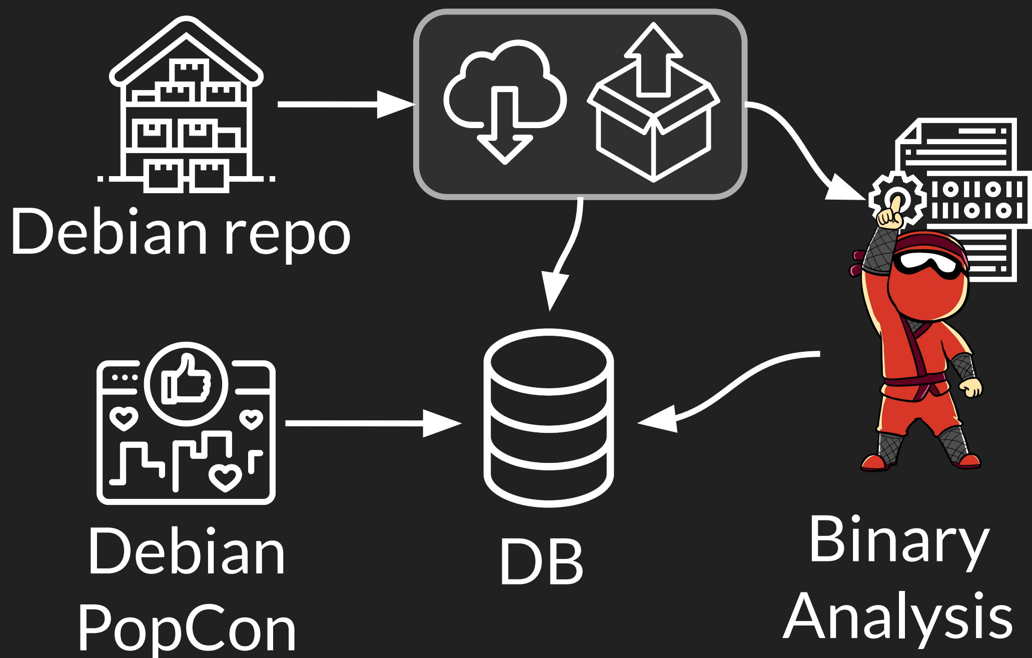


Select top 1000 popular packages (~3.3k binaries)



Not restricted to C++ binaries

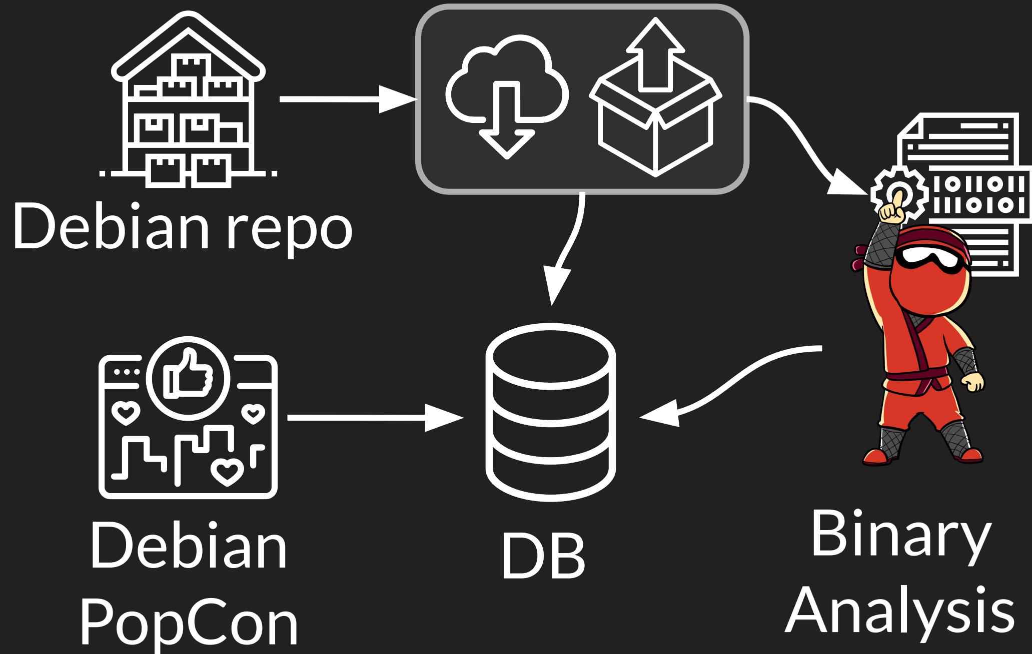
Select top 1000 popular packages (~3.3k binaries)



Not restricted to C++ binaries

~10% of binaries use exception handling

Select top 1000 popular packages (~3.3k binaries)



Not restricted to C++ binaries

~10% of binaries use exception handling

Half contain at least 40% throwing functions

Taint analysis

Taint analysis

- Extract exception handlers from our data set

Taint analysis

- Extract exception handlers from our data set
- Model forms of control as the taint source

Taint analysis

- Extract exception handlers from our data set
- Model forms of control as the taint source
- Search for tainted gadgets in the exception handlers

Interesting gadgets

- Arbitrary Free
- Control-flow Hijack
- Write-What-Where
- Write-Where-Only
- Write-What-Only

Interesting gadgets

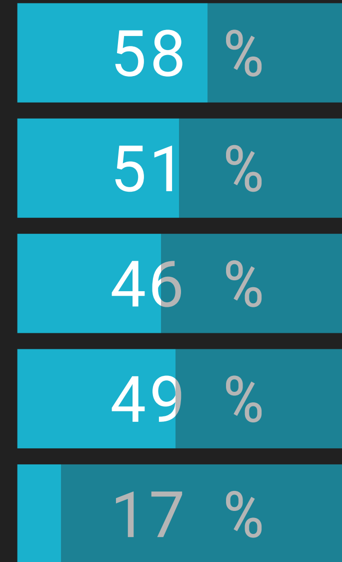
→ Arbitrary Free

→ Control-flow Hijack

→ Write-What-Where

→ Write-Where-Only

→ Write-What-Only



% of binaries containing at least one gadget

Real-World impact

Real-World impact

- Stack cookies

Real-World impact

- Stack cookies
- Intel CET, ShadowCallStack and more…

Real-World impact

- Stack cookies
- Intel CET, ShadowCallStack and more…
- Apple, Google, Intel, ARM, Microsoft, gcc and llvm

Conclusion

- CHOP attacks are possible
- … and dangerous

More in the paper...

- are feasible in practice (3 CVEs)
    - unexploitable with modern defences

Paper QR