



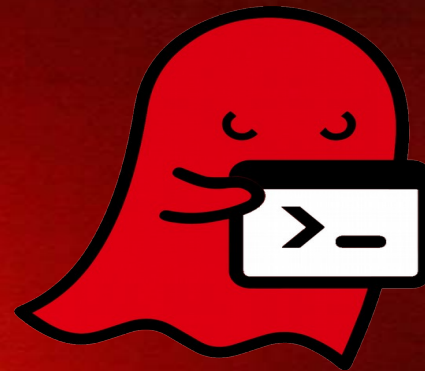
hexhive

New memory corruption attacks: why can't we have nice things?

Mathias Payer (@gannimo) and Nicholas Carlini
<http://hexhive.github.io>

Software is unsafe and insecure

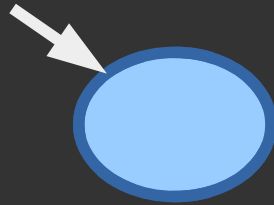
- Low-level languages (C/C++) trade type safety and memory safety for performance
 - Programmer responsible for all checks
- Large set of legacy and new applications written in C / C++ prone to memory bugs
- Too many bugs to find and fix manually
 - Protect integrity through safe runtime system



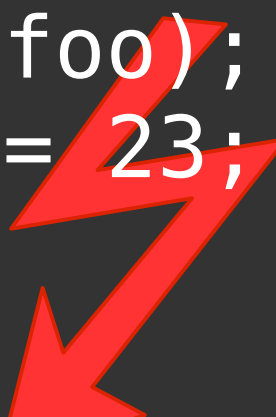
Memory (Un-)safety

Memory (un-)safety: invalid dereference

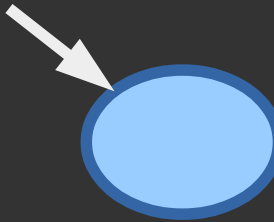
Dangling pointer:
(temporal)



```
free(foo);  
*foo = 23;
```



Out-of-bounds pointer:
(spatial)



```
char foo[40];  
foo[42] = 23;
```

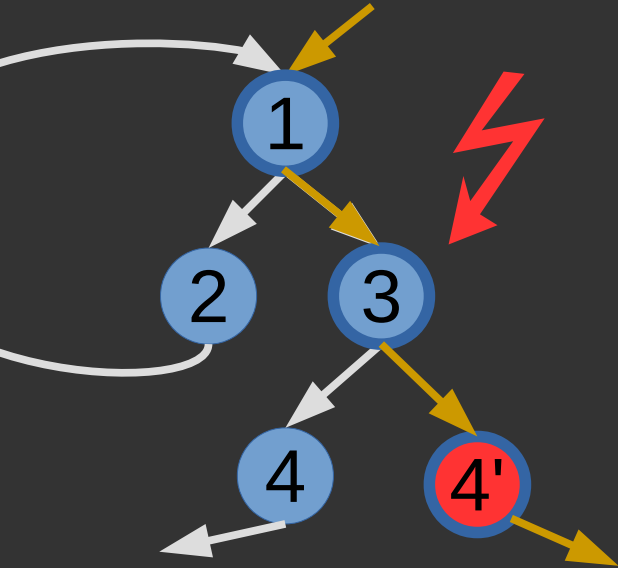
Violation iff: pointer is read, written, or freed

Two types of attack

- Control-flow hijack attack
 - Execute Code

**Today, we focus on
executing code**

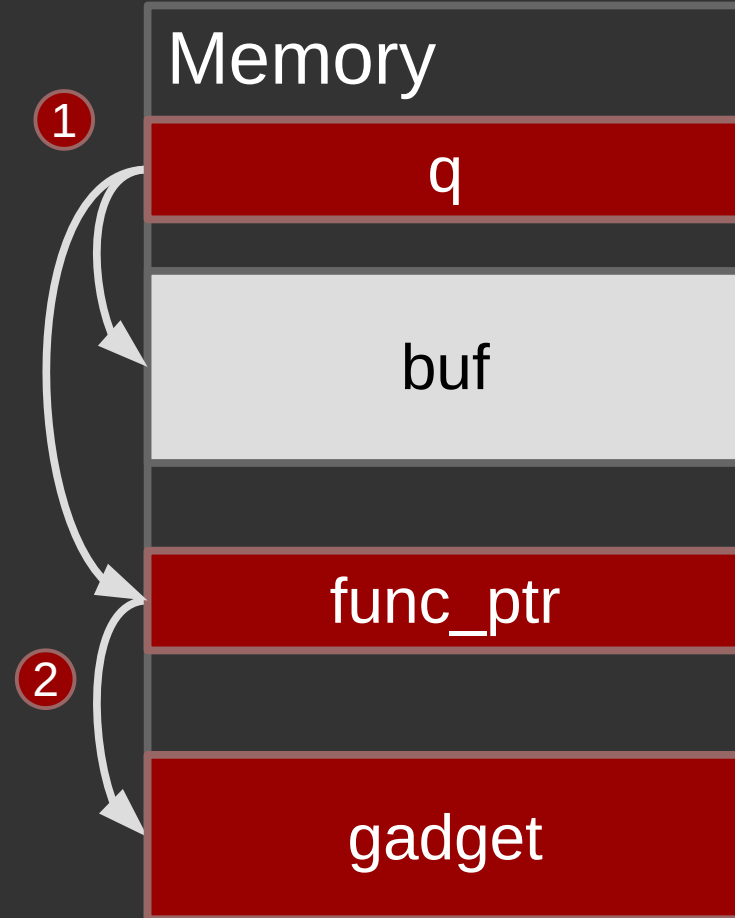

Control-flow hijack attack



- Attacker modifies *code pointer*
 - Function return
 - Indirect jump
 - Indirect call
- Control-flow leaves *valid graph*
- Reuse existing code
 - Return-oriented programming
 - Jump-oriented programming

Control-Flow Hijack Attack

```
int vuln(int usr, int usr2){  
    void *(func_ptr)();  
    ① int *q = buf + usr;  
    ...  
    func_ptr = &foo;  
    ...  
    ② *q = usr2;  
    ...  
    ③ (*func_ptr)();  
}
```



Status of deployed defenses

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack canaries
- Safe exception handlers

Memory

0x400 ~~R~~WX

text

0x800 ~~R~~WX

data

0xf?? ~~R~~WX



stack



Status of deployed defenses

- ASLR and DEP only effective in combination
- **Breaking** ASLR enables code reuse
 - On desktops, information leaks are common
 - On servers, code reuse attacks have decreased
 - For clouds: look at CAIN ASLR attack from WOOT'15

Stack Integrity and Control-Flow Integrity

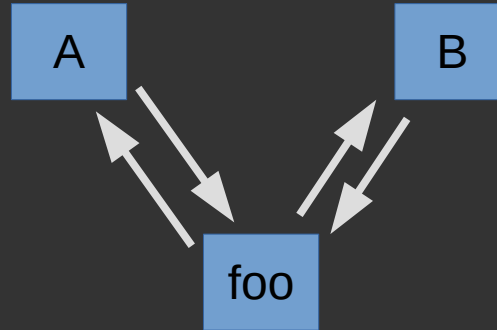
Stack integrity

- Enforce dynamic restrictions on return instructions
- Protect return instructions through shadow stack

```
void a() {  
    foo();  
}
```

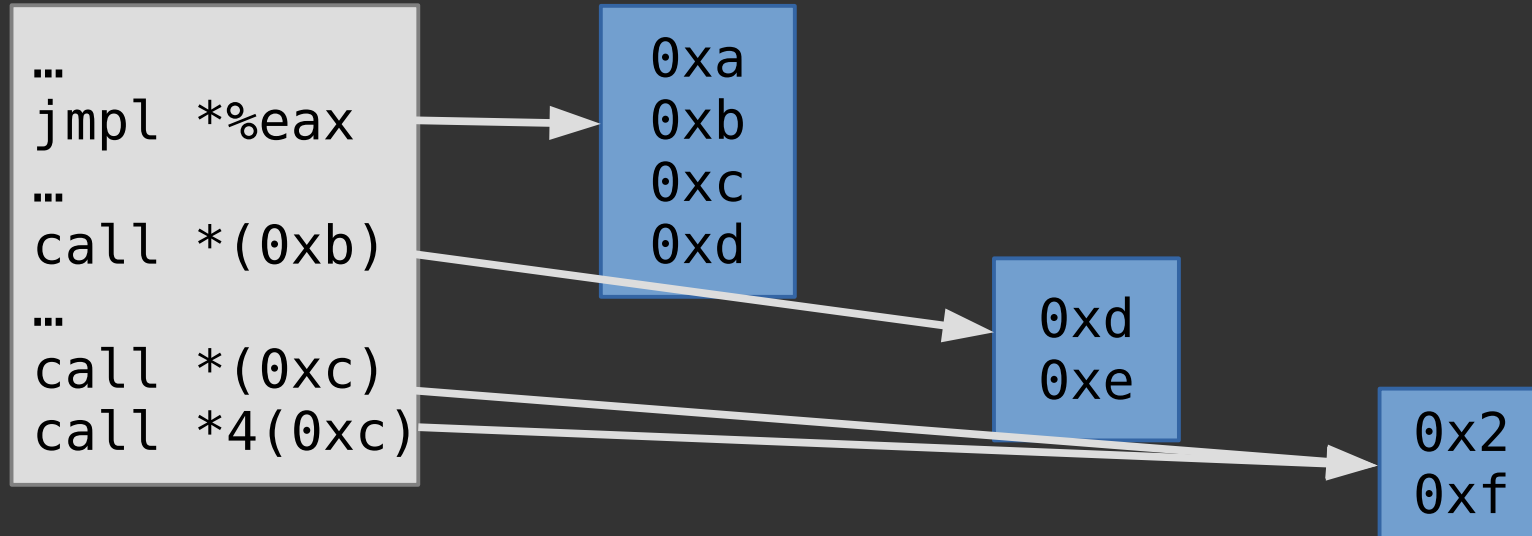
```
void b() {  
    foo();  
}
```

```
void foo();
```



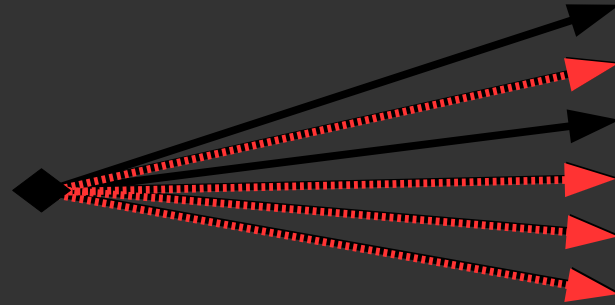
Control-Flow Integrity (CFI)

- Statically construct Control-Flow Graph
 - Find set of allowed targets for each location
- Online set check



Control-Flow Integrity (CFI)

```
CHECK(fn);  
(*fn)(x);
```



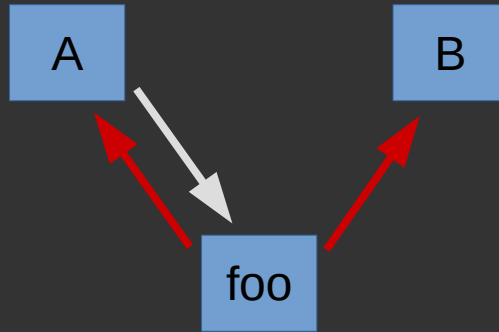
**Attacker may write to memory,
code ptrs. verified when used**

CFI on the stack

```
void a() {  
    foo();  
}
```

```
void b() {  
    foo();  
}
```

```
void foo();
```



Novel Code Reuse Attacks

Control-Flow Bending

- Attacker-controlled execution along “*valid*” CFG
 - Generalization of non-control-data attacks
- Each individual control-flow transfer is valid
 - Execution trace may not match non-exploit case
- Circumvents static, fully-precise CFI

CFI's limitation: statelessness

- Each state is verified without context
 - Unaware of constraints between states
- Bending CF along valid states undetectable
 - Search path in CFG that matches desired behavior

Weak CFI is broken

Microsoft's Control-Flow Guard is an instance of a weak CFI mechanism

- *Size does matter: why using gadget-chain length to prevent code-reuse is hard*

Goektas et al., Usenix SEC '14

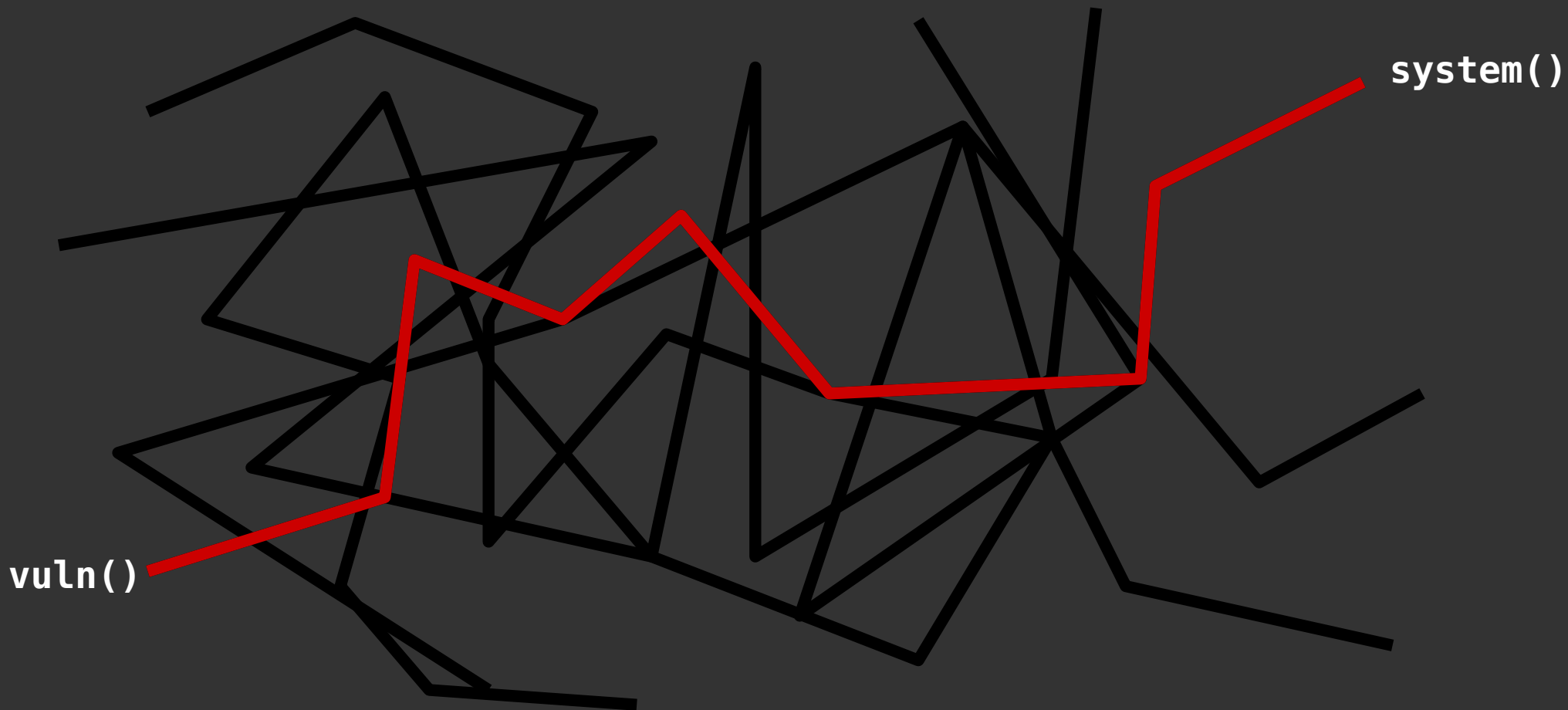
Strong CFI

- Precise CFG: no over-approximation
- Stack integrity (through shadow stack)
- Fully-precise static CFI: a transfer is only allowed if some benign execution uses it
- How secure is CFI?
 - With and without stack integrity

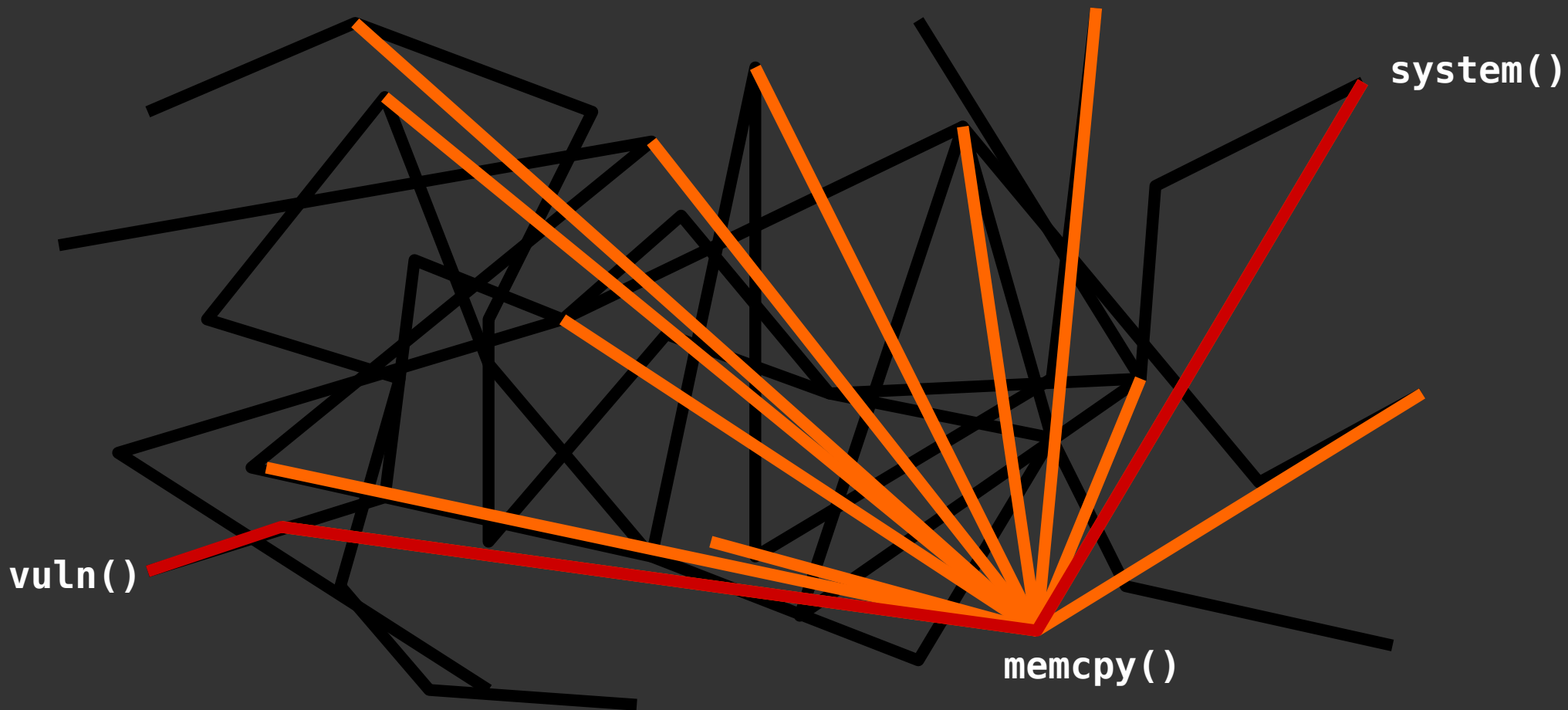
CFI, no stack integrity: ROP challenges

- Find path to `system()` in CFG.
- Divert control-flow along this path
 - Constrained through memory vulnerability
- Control arguments to `system()`

What does a CFG look like?



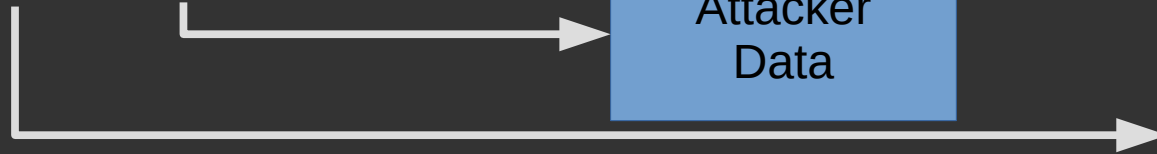
What does a CFG look like? Really?



Dispatcher functions

- Frequently called
- Arguments are under attacker's control
- May overwrite their own return address

`memcpy(dst, src, 8)`



Caller
Stack
Frame

Return
Address

Local
Data

Control-Flow Bending, no stack integrity

- CFI without stack integrity is broken
 - Stateless defenses insufficient for stack attacks
 - Arbitrary code execution in all cases
- Attack is program-dependent, harder than w/o CFI

Counterfeit Object-Oriented Programming

- A function can be a gadget too!

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

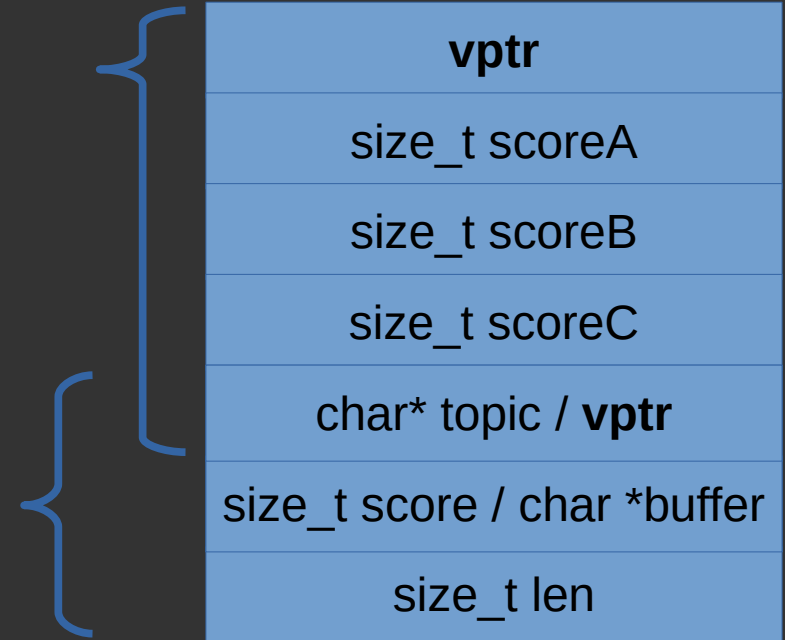
Keyword for ind. call

Control length of loop

Array with ptrs. to vtables

Counterfeit Object-Oriented Programming

```
class Exam {  
private:  
    size_t scoreA, scoreB, scoreC;  
public:  
    char *topic; Arithmetic  
    virtual void updateScore() {  
        score = scoreA + scoreB + scoreC;  
    }  
};  
  
struct SimpleString {  
    char *buffer; size_t len;  
    virtual void set(char *s) {  
        strncpy(buffer, s, len);  
    }  
}; "memcpy"
```



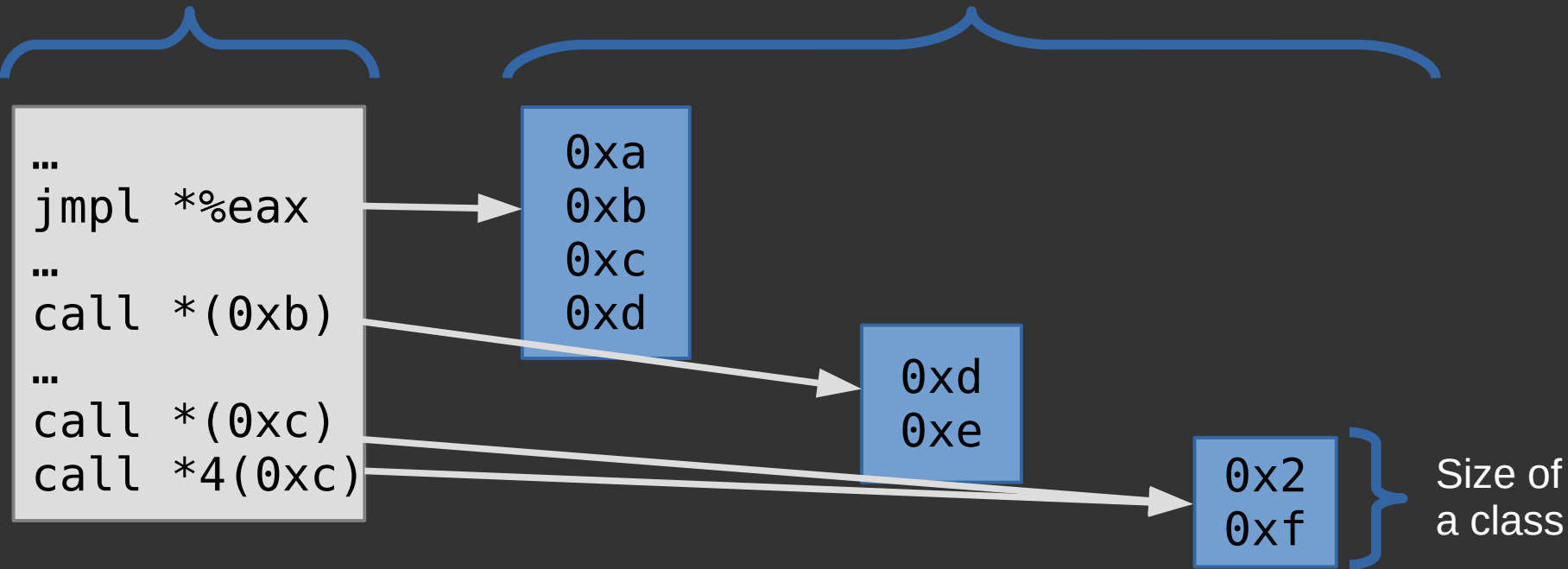
Existing CFI mechanisms

- Lockdown
- MCFI and piCFI
- Google LLVM-CFI
- Google IFCC
- MS Control-Flow Guard
- Many many others

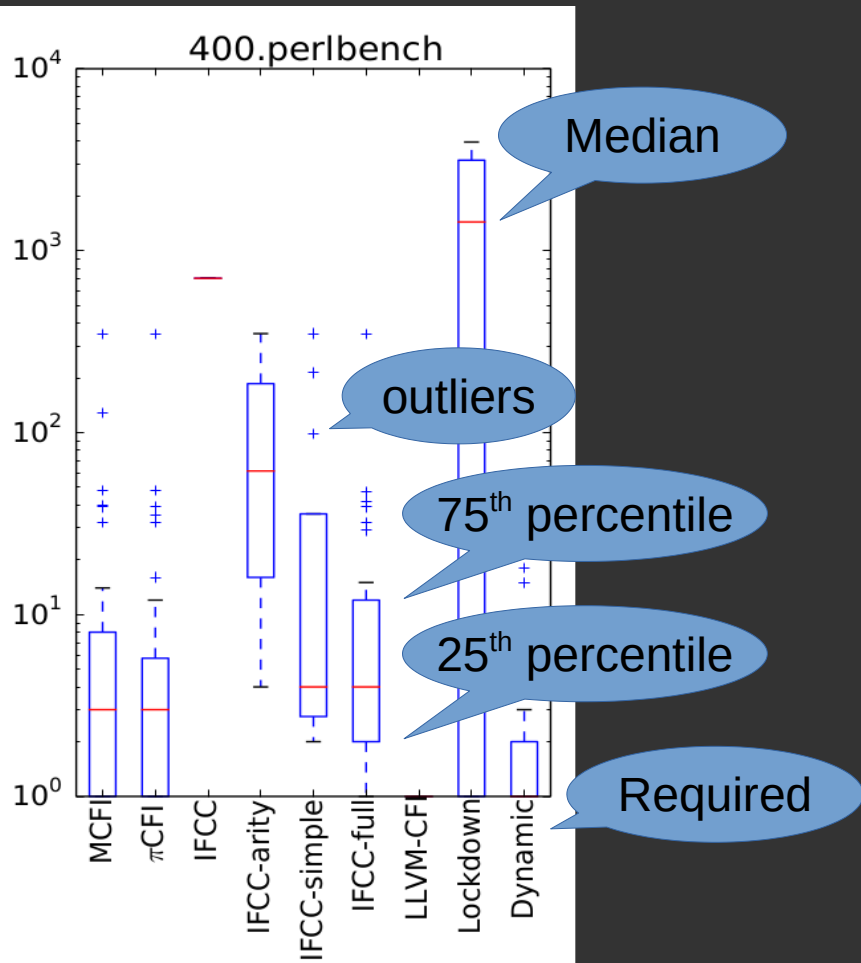
Remember CFI?

Indirect CF transfers






Equivalence classes



Forward edge precision: size of eqi classes



Existing CFI mechanisms

CFI mechanism	Forward Edge	Backward Edge	CFB
IFCC	~	□	
MS CFG	~	□	
LLVM-CFI	□	□	
MCFI/piCFI	□	~	
Lockdown	~+	□	

What if we have stack integrity?

- ROP no longer an option
- Attack becomes harder
 - Need to find a path through virtual calls
 - Resort to “restricted COOP”
- An interpreter would make attacks much simpler...

printf()-oriented programming

- Translate program to format string
 - Memory reads: %s
 - Memory writes: %n
 - Conditional: %.*d
- Program counter becomes format string counter
 - Loops? Overwrite the format specific counter
- Turing-complete domain-specific language

Ever heard of brainfuck?

- > == dataptr++
%1\$65535d%1\$.*1\$d%2\$hn
- < == dataptr--
%1\$.*1\$d %2\$hn
- + == (*dataptr)++
%3\$.*3\$d %4\$hhn
- - == (*dataptr)--
%3\$255d%3\$.*3\$d%4\$hhn
- . == putchar(*dataptr)
%3\$.*3\$d%5\$hn
- , == getchar(dataptr)
%13\$.*13\$d%4\$hn
- [== if (*dataptr == 0) goto ']'
%1\$.*1\$d%10\$.*10\$d%2\$hn
-] == if (*dataptr != 0) goto '['
%1\$.*1\$d%10\$.*10\$d%2\$hn

```

void loop() {
    char* last = output;
    int* rpc = &progn[pc];

    while (*rpc != 0) {
        // fetch -- decode next instruction
        sprintf(buf, "%1$.*1$d%1$.*1$d%1$.*1$d%1$.*1$d%1$.*1$d%1$.*1$d%1$.*1$d%2$hn",
            *rpc, (short*)(&real_syms));

        // execute -- execute instruction
        sprintf(buf, *real_syms,
            ((long long int)array)&0xFFFF, &array, // 1, 2
            *array, array, output, // 3, 4, 5
            ((long long int)output)&0xFFFF, &output, // 6, 7
            &cond, &bf_CGOTO_fmt3[0], // 8, 9
            rpc[1], &rpc, 0, *input, // 10, 11, 12, 13
            ((long long int)input)&0xFFFF, &input // 14, 15
        );

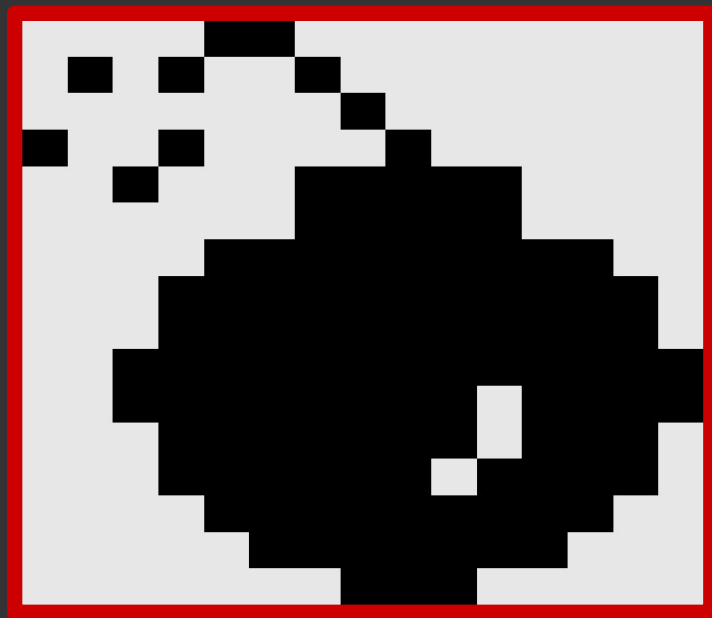
        // retire -- update PC
        sprintf(buf, "12345678%.*d%hn", (int)((((long long int)rpc)&0xFFFF), 0, (short*)&rpc);

        // for debug: do we need to print?
        if (output != last) { putchar(output[-1]); last = output; }
    }
}

```

Introducing: printbf

- Turing complete interpreter
- Relies on format strings
- Allows you to execute stuff



<http://github.com/HexHive/printbf>

Conclusion

Conclusion

- Low level languages are here to stay
 - ... and they are full of “potential”
- Without stack integrity, defenses are broken
- Even with stack integrity we can do fun stuff
 - Enjoy our Turing-complete printf interpreter



hexhive

Thank you!
Questions?

Mathias Payer (@gannimo) and Nicholas Carlini
<http://hexhive.github.io>