Memory corruption: why we can’t have nice things

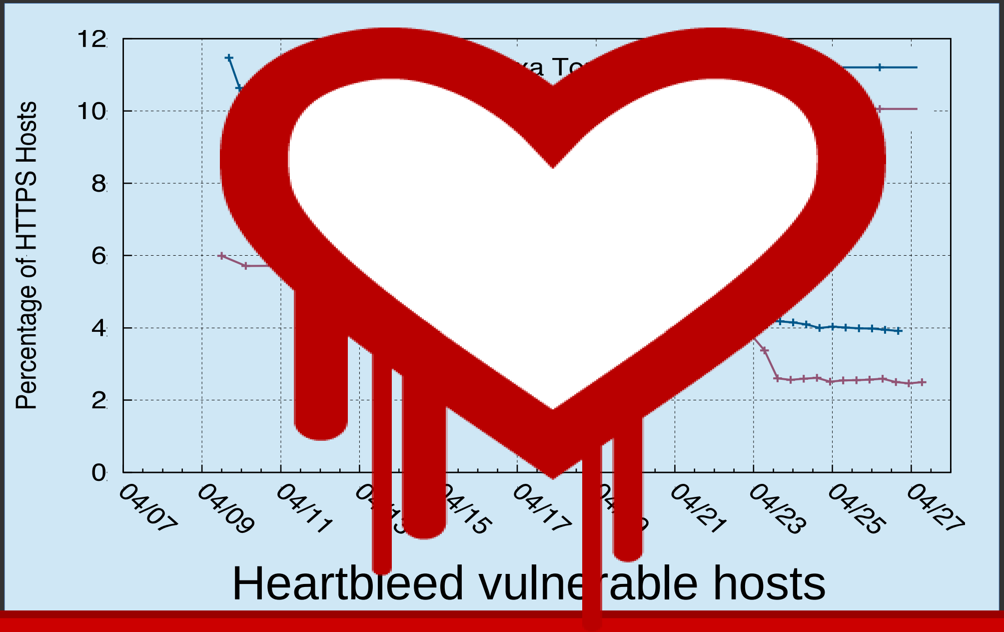
Mathias Payer (@gannimo)
http://hexhive.github.io

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

Heartbleed: patching observations

- 11% of servers remained vulnerable after 48 hours
- Patching plateaued at 4%
- Only 10% of vulnerable sites replaced certificates
- 15% of replaced cert's used vulnerable cryptographic keys

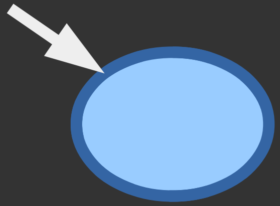


Update process is slow, incomplete, and incorrect

Memory (Un-)safety

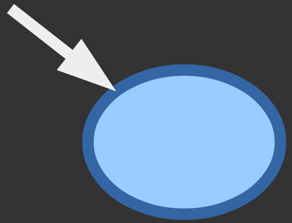
Memory (un-)safety: invalid dereference

Dangling pointer:
(temporal)



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

Out-of-bounds pointer:
(spatial)

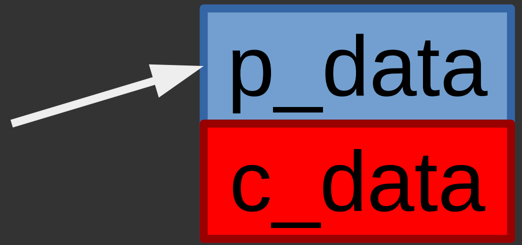


```
char foo[40];
foo[42] = 23;
```

Violation iff: pointer is read, written, or freed

Memory (un-)safety: type confusion

```
class P {
  int p_data;
};
class C: public P {
  int c_data;
};
P *Pptr = new P;
C *Cptr = static_cast<C*>(Pptr);
Cptr->c_data; // Type confusion!
```

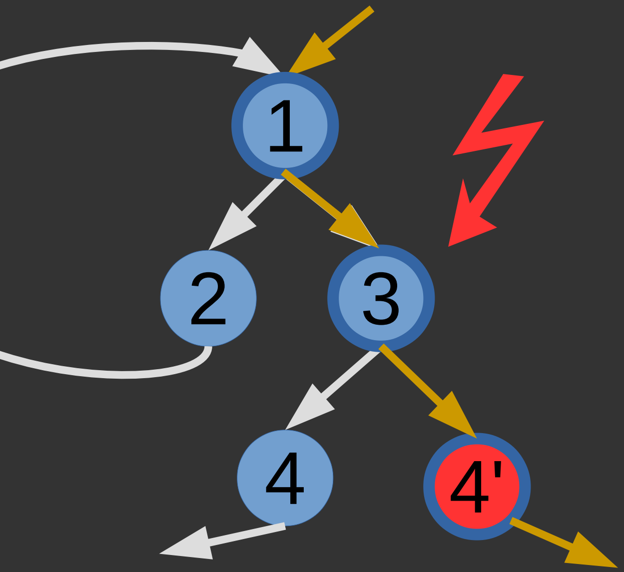


Two types of attack

- Control-flow hijack attack
  - Execute Code

Let’s focus on executing code

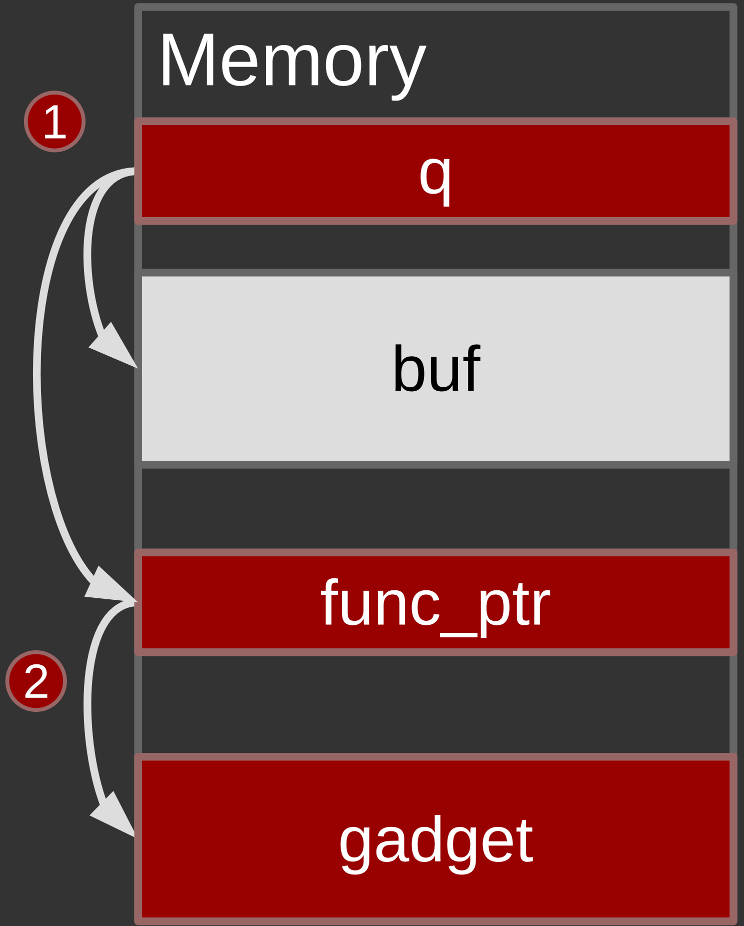
Control-flow hijack attack



- 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

Status of deployed defenses

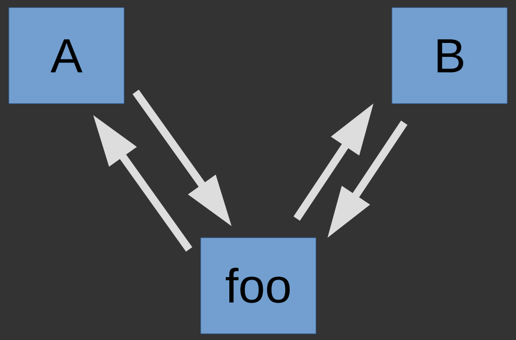
- ASLR and DEP only effective in combination
- *Breaking* ASLR enables code reuse
  - On desktops, information leaks are common
  - On servers, code reuse attacks have decreased
  - For clouds: CAIN attack at WOOT'15
  - For OS: Dedup Est Machine at S&P’16
  - For browsers: Flip Feng Shui at SEC’16

Stack Integrity
and
Control-Flow Integrity

Stack integrity

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

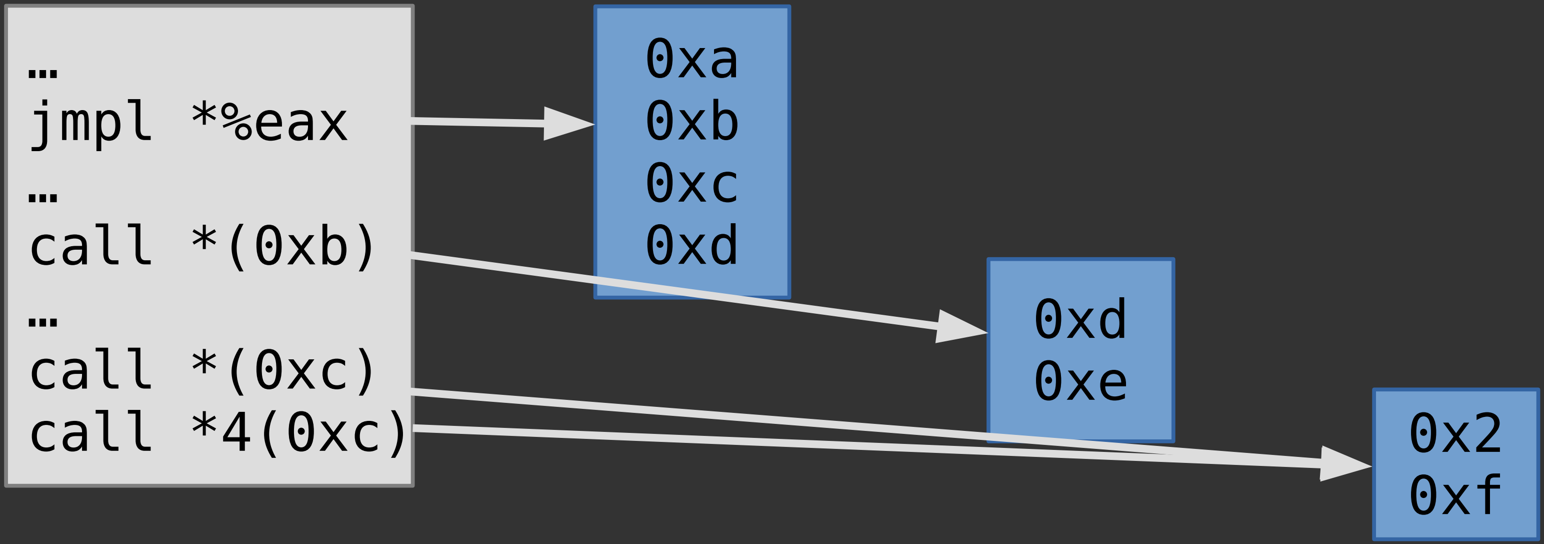
```
void a() {
  foo();
}

void b() {
  foo();
}

void foo();
```



Control-Flow Integrity (CFI)

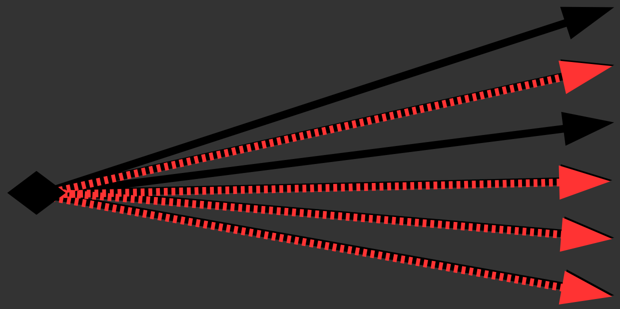
- Statically construct Control-Flow Graph
  - Find set of allowed targets for each location

- Online set check



Control-Flow Integrity (CFI)

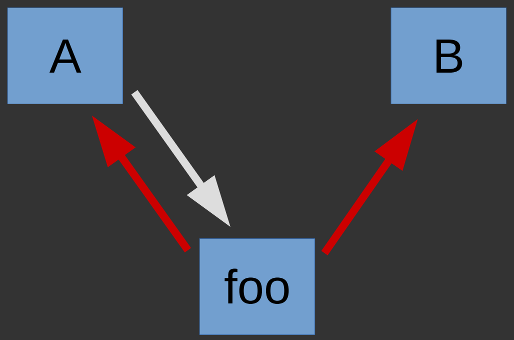
```
CHECK(fn);
(*fn)(x);
```



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

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross
“Control-Flow Bending”, Usenix SEC'15

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

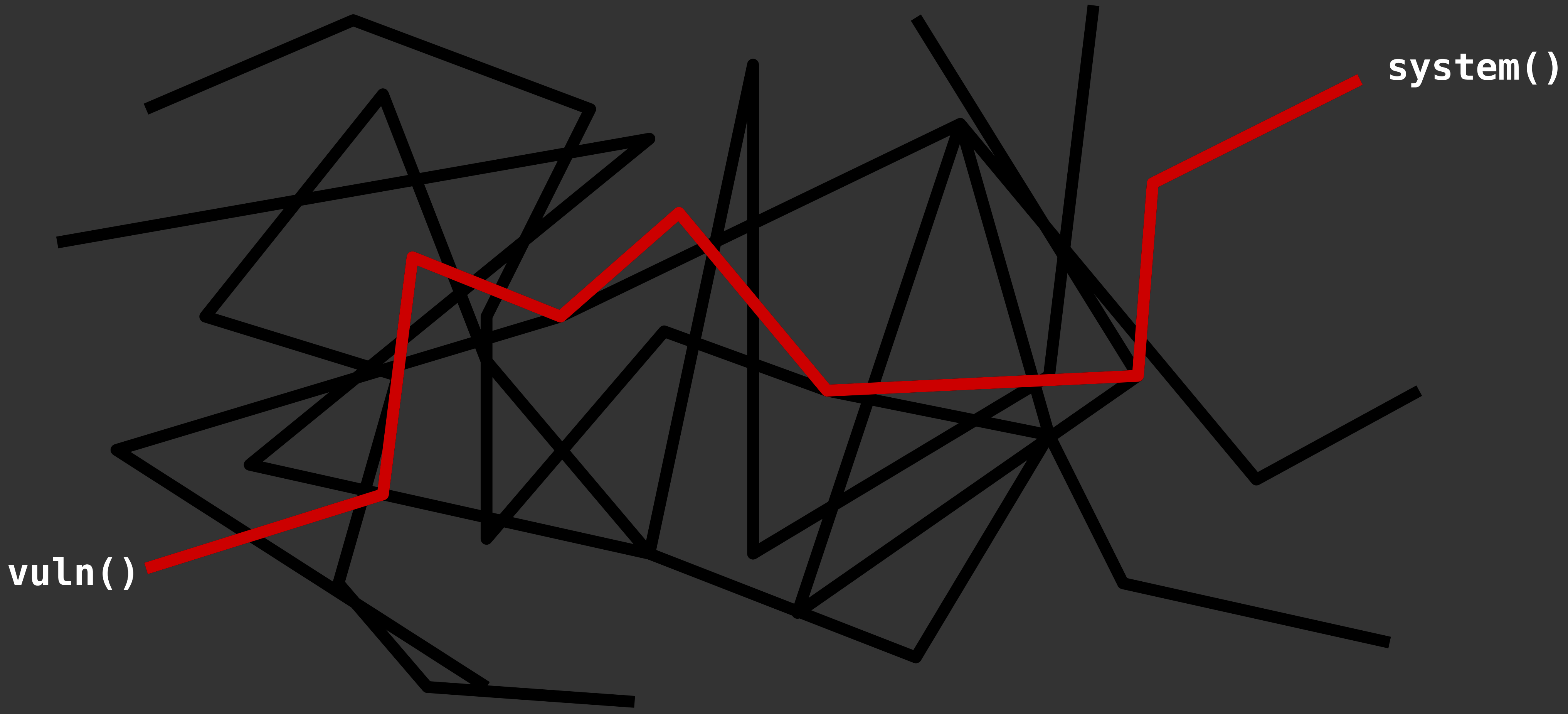
- Precise CFG: no over-approximation
- Stack integrity (through shadow stack)
- Fully-precise static CFI: a transfer is only allowed if some benign execution uses it

- How secure is CFI?
  - With and without stack integrity

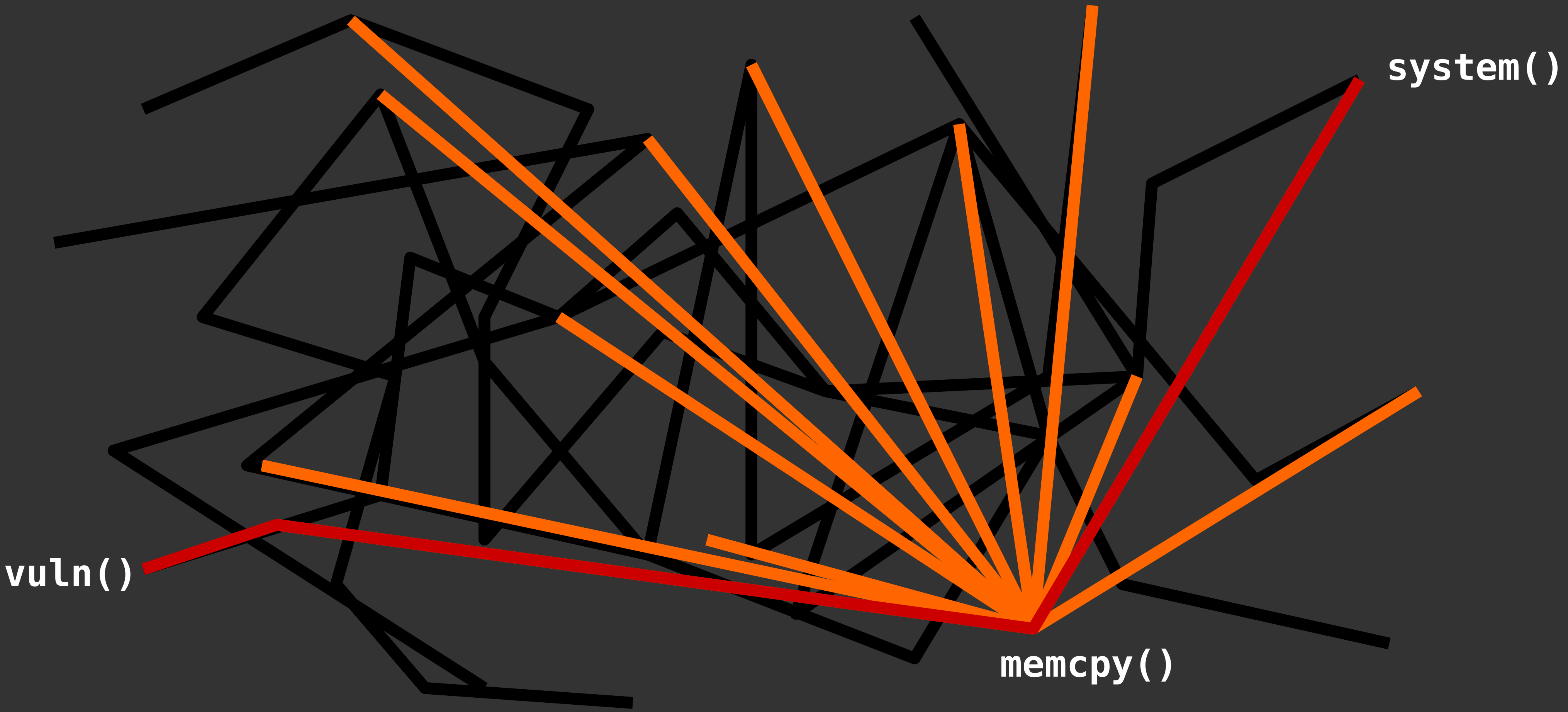
CFI, no stack integrity: ROP challenges

- 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?

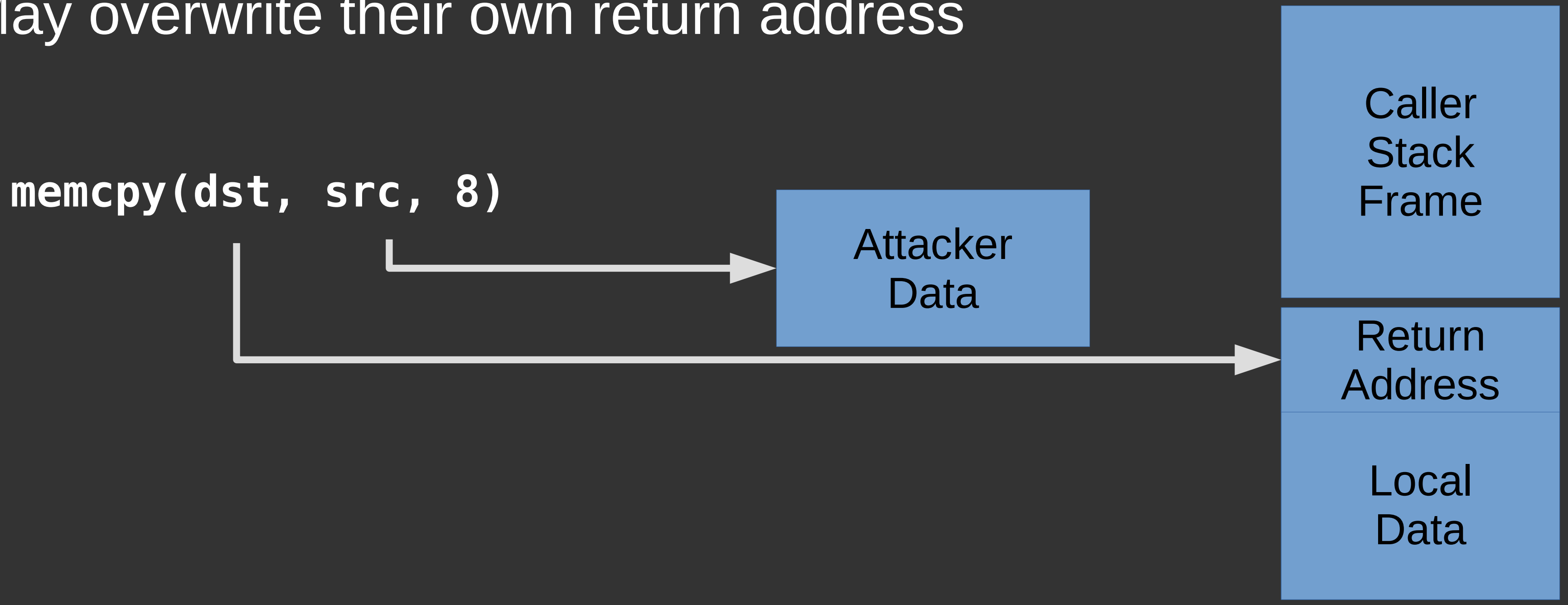


What does a CFG look like? Really?



Dispatcher functions

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

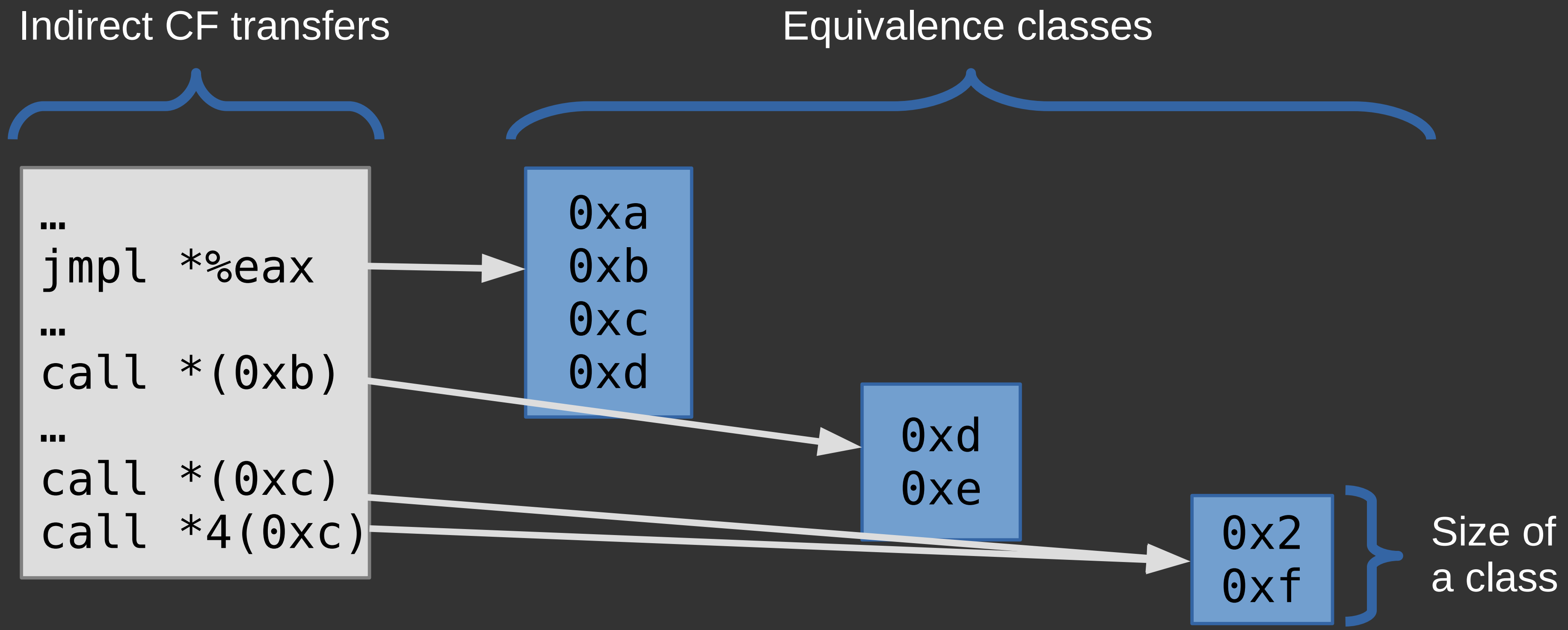


Control-Flow Bending, no stack integrity

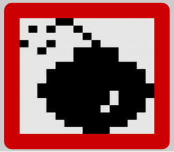
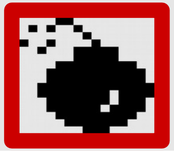
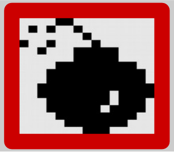
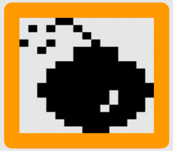
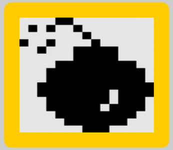
- CFI without stack integrity is broken
  - Stateless defenses insufficient for stack attacks
  - Arbitrary code execution in all cases

- Attack is program-dependent, harder than w/o CFI

Remember CFI?



Existing CFI mechanisms

CFI mechanism	Forward Edge	Backward Edge	CFB
Google 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

* Direct fame to Nicholas Carlini, blame to me

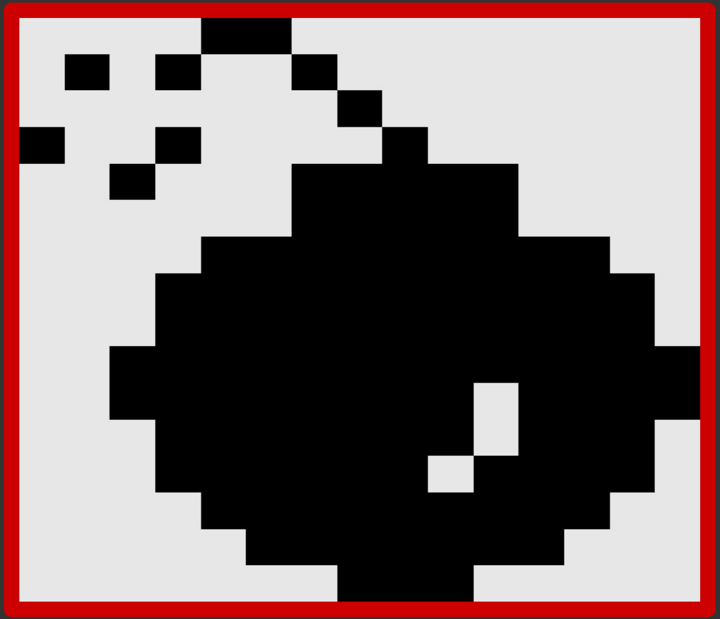
Ever heard of brainfuck?

>	== dataptr++	%1$65535d%1$.*1$d%2$hn
<	== dataptr--	%1$.*1$d %2$hn
+	== (*dataptr)++	%3$.*3$d %4$hhn
-	== (*datapr)--	%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%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

* Direct fame to Nicholas Carlini, blame to me

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 printbf interpreter