Lec 14: CFI

CS492E: Introduction to Software Security

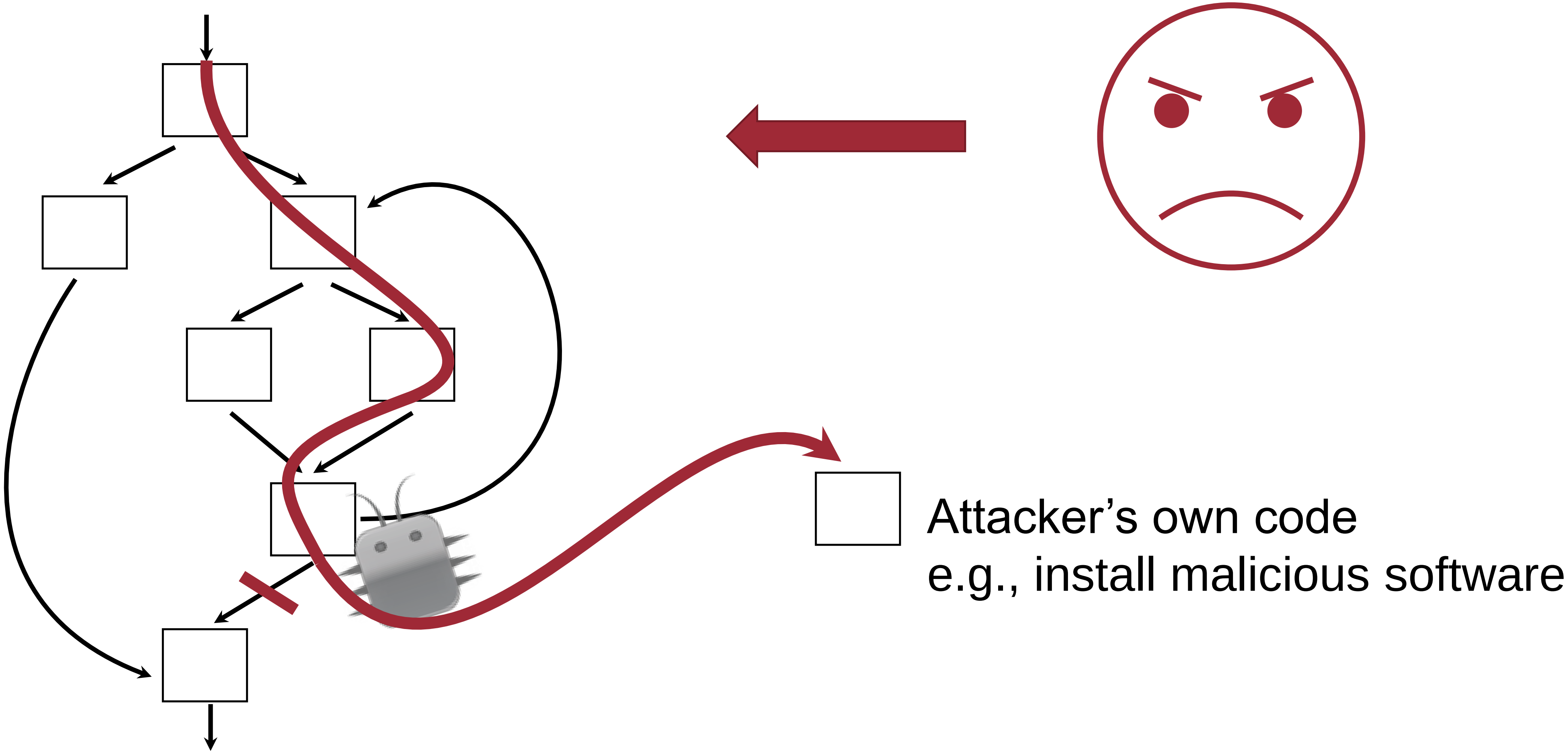
Sang Kil Cha

Defense Techniques So Far …

- DEP
- ASLR
- Canary

Problem: control-flow hijacking still possible

Control Flow Hijack Exploit



Can we enforce control-flow integrity?

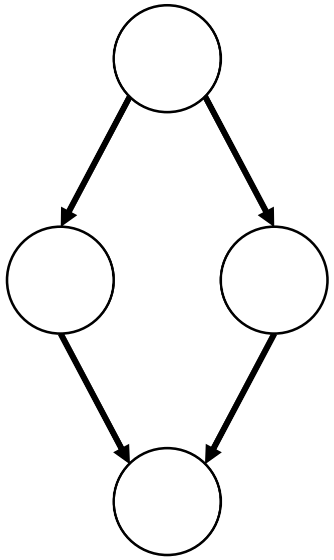
CFI Policy

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ***ahead of time***.

Quote from control flow integrity, CCS 2005

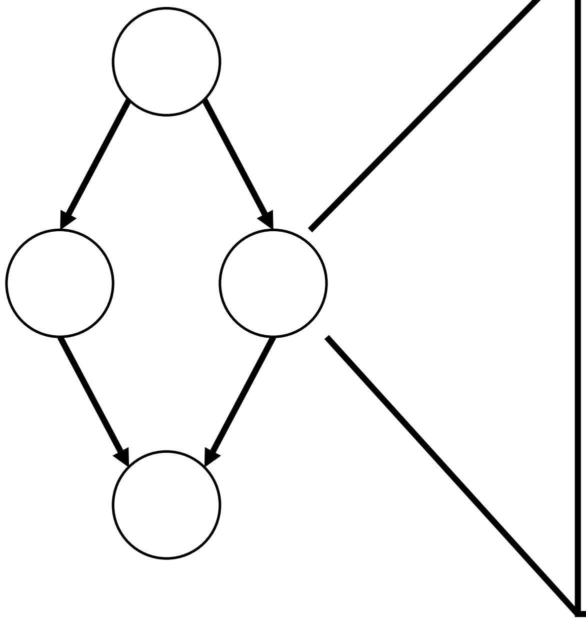
CFG (Control Flow Graph)

A CFG is a graph that represents all paths that might be traversed through a program execution.



CFG (Control Flow Graph)

Each node in a CFG represents a ***basic block***



Basic Block:

A sequence of statements that is always entered at the beginning and exited at the end*

* Quote from Modern Compiler Implementation

Basic Block

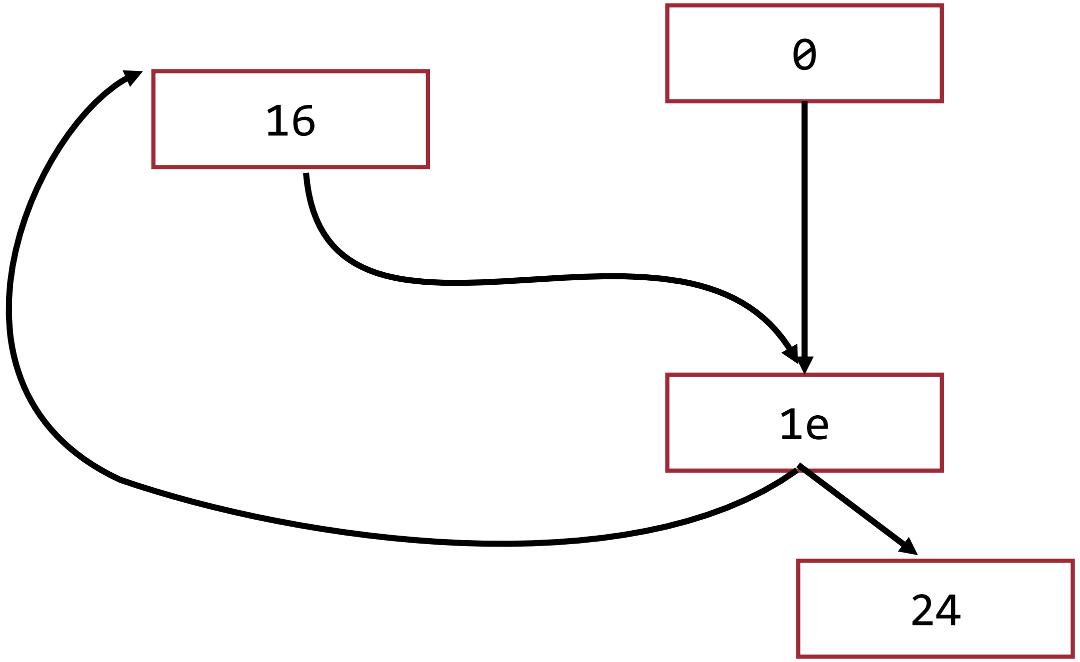
```
 0:   55                      push   ebp
 1:   89 e5                   mov    ebp,esp
 3:   83 ec 10                sub    esp,0x10
 6:   c7 45 f8 00 00 00 00    mov    DWORD PTR [ebp-0x8],0x0
 d:   c7 45 fc 0a 00 00 00    mov    DWORD PTR [ebp-0x4],0xa
14:   eb 08                   jmp    1e <v+0x1e>

16:   83 45 f8 01             add    DWORD PTR [ebp-0x8],0x1
1a:   83 6d fc 01             sub    DWORD PTR [ebp-0x4],0x1

1e:   83 7d fc 00             cmp    DWORD PTR [ebp-0x4],0x0
22:   7f f2                   jg     16 <v+0x16>

24:   8b 45 f8                mov    eax,DWORD PTR [ebp-0x8]
27:   c9                      leave
28:   c3                      ret
```

CFI = Any Execution Should Follow Control Paths of This CFG



CFI Assumptions

- Attackers cannot execute data (DEP is enabled)
- Programs cannot change themselves (no self-modifying code)

How to Enforce CFI?

- Give ***unique*** IDs at destinations
- For all branch instructions, check destination IDs before taking the branch

How to Instrument?

```
                          Source                                                  Destination
Opcode bytes                  Instructions                          Opcode bytes           Instructions
FF E1              jmp  ecx                 ; computed jump        8B 44 24 04     mov  eax, [esp+4]    ; dst
                                                                   ...

                              can be instrumented as (a):

81 39 78 56 34 12  cmp  [ecx], 12345678h   ; comp ID & dst        78 56 34 12     ; data 12345678h     ; ID
75 13              jne  error_label        ; if != fail           8B 44 24 04     mov  eax, [esp+4]    ; dst
8D 49 04           lea  ecx, [ecx+4]       ; skip ID at dst       ...
FF E1              jmp  ecx                ; jump to dst

                         or, alternatively, instrumented as (b):

B8 77 56 34 12     mov  eax, 12345677h     ; load ID-1            3E 0F 18 05     prefetchnta          ; label
40                 inc  eax                ; add 1 for ID         78 56 34 12        [12345678h]       ;    ID
39 41 04           cmp  [ecx+4], eax       ; compare w/dst        8B 44 24 04     mov  eax, [esp+4]    ; dst
75 13              jne  error_label        ; if != fail           ...
FF E1              jmp  ecx                ; jump to label
```

Image from control flow integrity, CCS 2005

CFI Challenge

What if a single branch instruction can jump to multiple addresses? (e.g., `call eax`)

Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

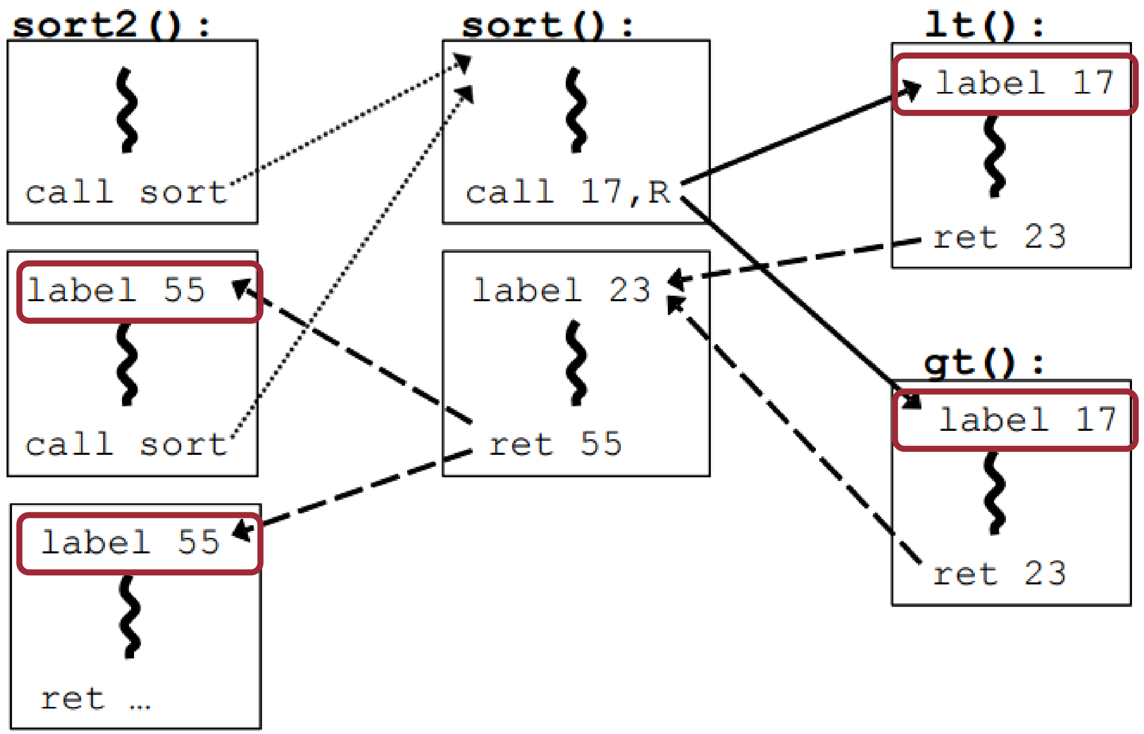


Image from control flow integrity, CCS 2005

Can you spot labeling problems?

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

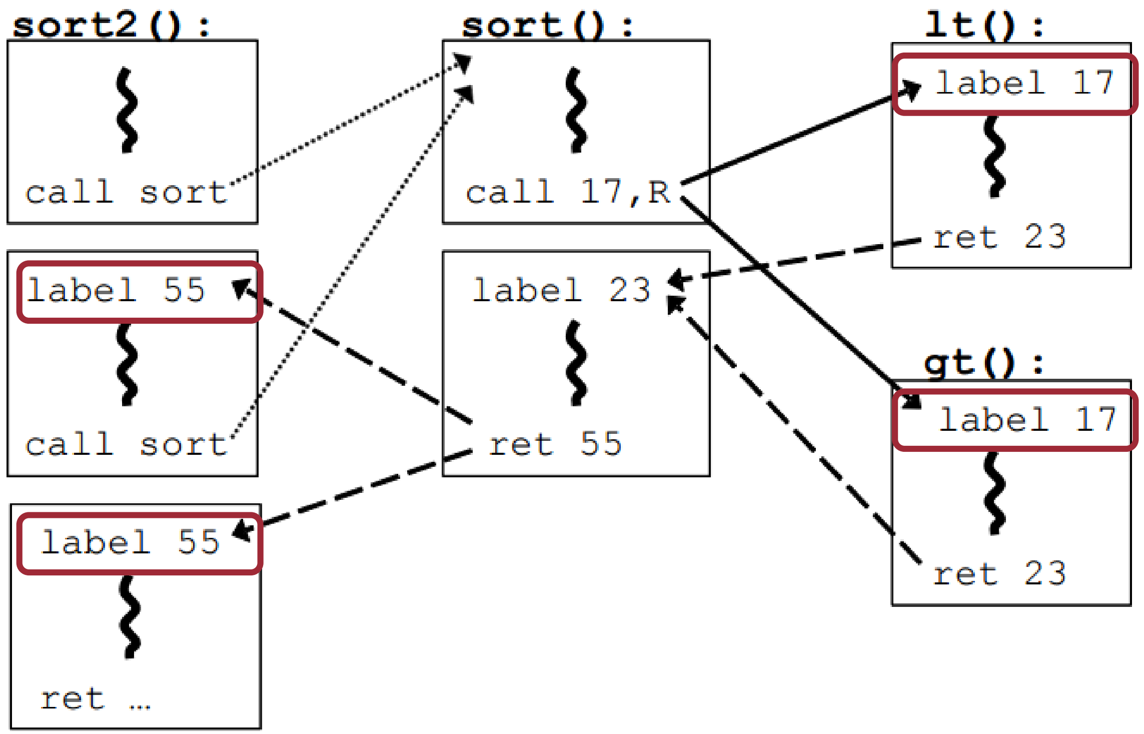
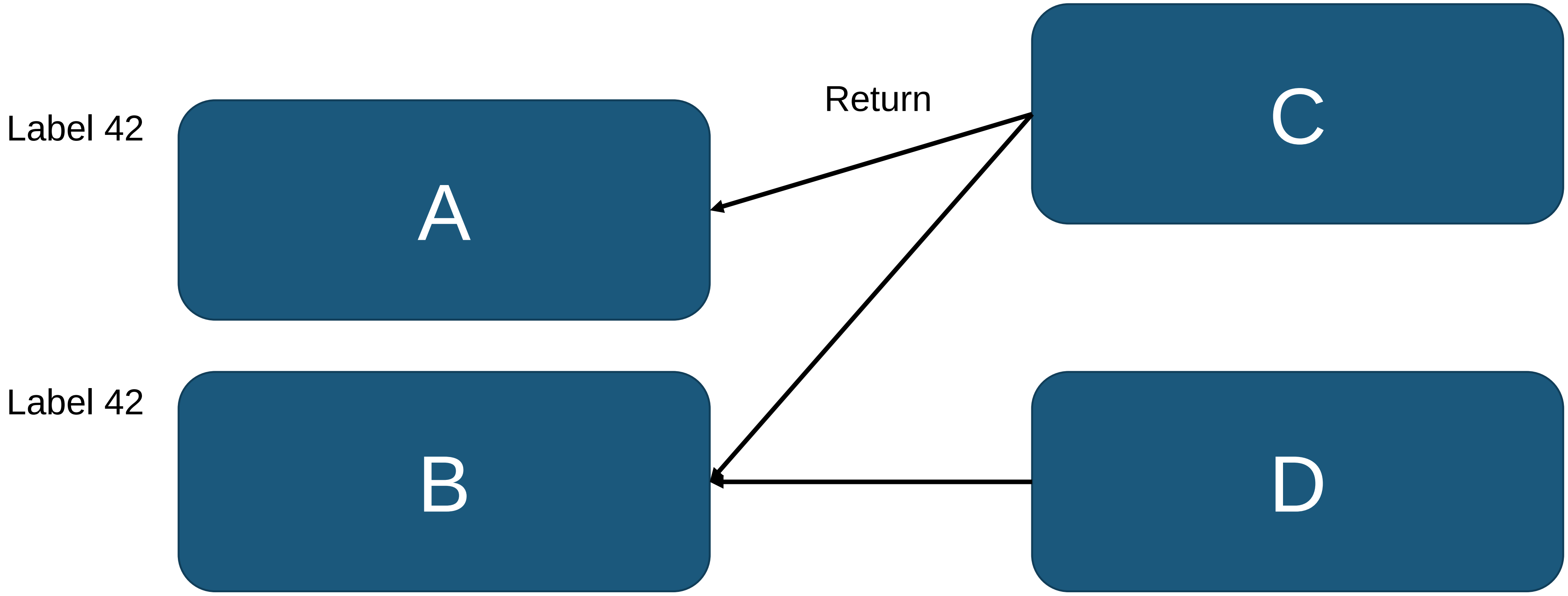


Image from control flow integrity, CCS 2005

Problem: What if D returns to A?



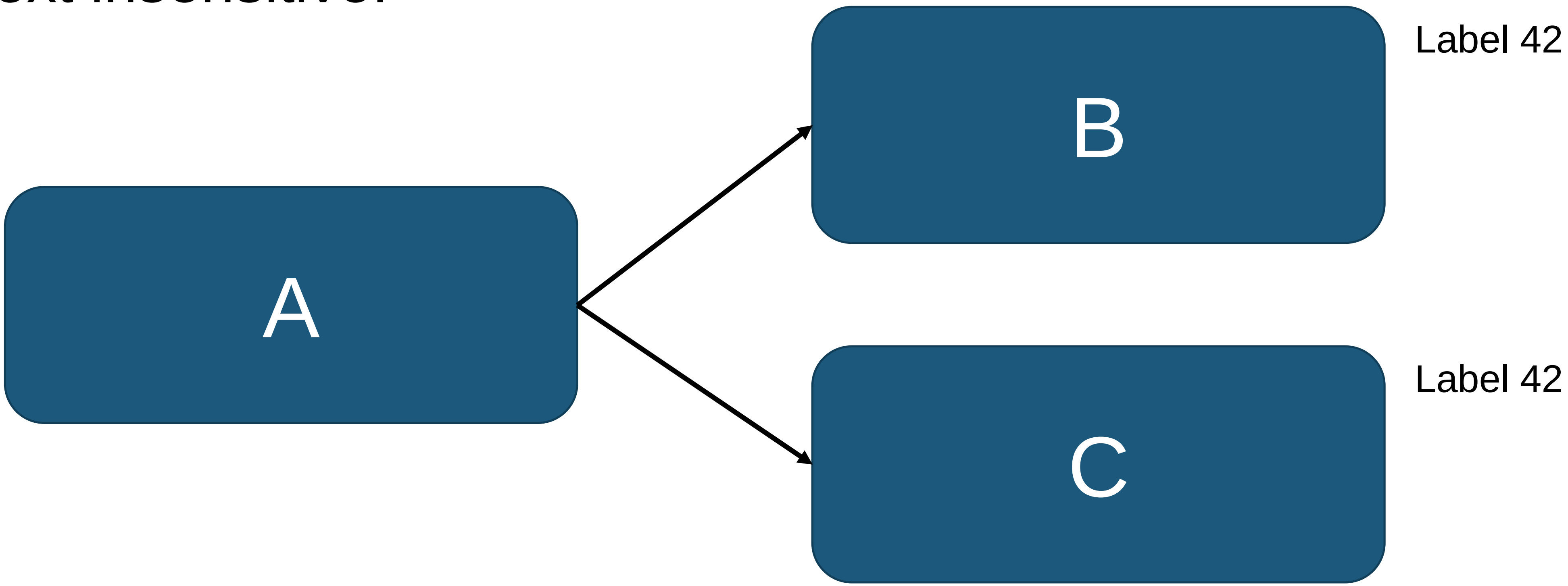
Potential Solutions

- Multiple tags
- Shadow call stack

What's the problem?

Another Problem

Context insensitive!



Shadow Call Stack

- In function prologues, store the return address in another area of memory

- In function epilogues, check if we are returning to the proper address

A Binary Rewriting Defense against Stack based Buffer Overflow Attacks, ***USENIX ATC 2003***

CFI with Shadow Call Stack

```
call  eax              ; call func ptr          ret                  ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add   gs:[0h], 4h      ; inc stack by 4         mov   ecx, gs:[0h]    ; get top offset
mov   ecx, gs:[0h]     ; get top offset         mov   ecx, gs:[ecx]   ; pop return dst
mov   gs:[ecx], LRET   ; push ret dst           sub   gs:[0h], 4h     ; dec stack by 4
cmp   [eax+4], ID      ; comp fptr w/ID         add   esp, 4h         ; skip extra ret
jne   error_label      ; if != fail             jmp   ecx             ; jump return dst
call  eax              ; call func ptr
LRET: ...
```

Why not just use a `ret` instruction?

Image from control flow integrity, CCS 2005

Time of Check to Time of Use

```
if (access("file", W_OK) != 0) {
   exit(1); // exit if not writable
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

TOC

Attacker can manipulate the file system

TOU

Example taken from Wikipedia (https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use)

TOCTTOU

```
call  eax             ; call func ptr
```

```
ret                   ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
      add  gs:[0h], 4h      ; inc stack by 4
      mov  ecx, gs:[0h]     ; get top offset
      mov  gs:[ecx], LRET   ; push ret dst
      cmp  [eax+4], ID      ; comp fptr w/ID
      jne  error_label      ; if != fail
      call eax              ; call func ptr
LRET: ...
```

```
mov  ecx, gs:[0h]     ; get top offset
mov  ecx, gs:[ecx]    ; pop return dst
sub  gs:[0h], 4h      ; dec stack by 4
add  esp, 4h          ; skip extra ret
jmp  ecx              ; jump return dst
```

TOCTTOU can happen here if `ret` is used

Image from control flow integrity, CCS 2005

Runtime Overhead

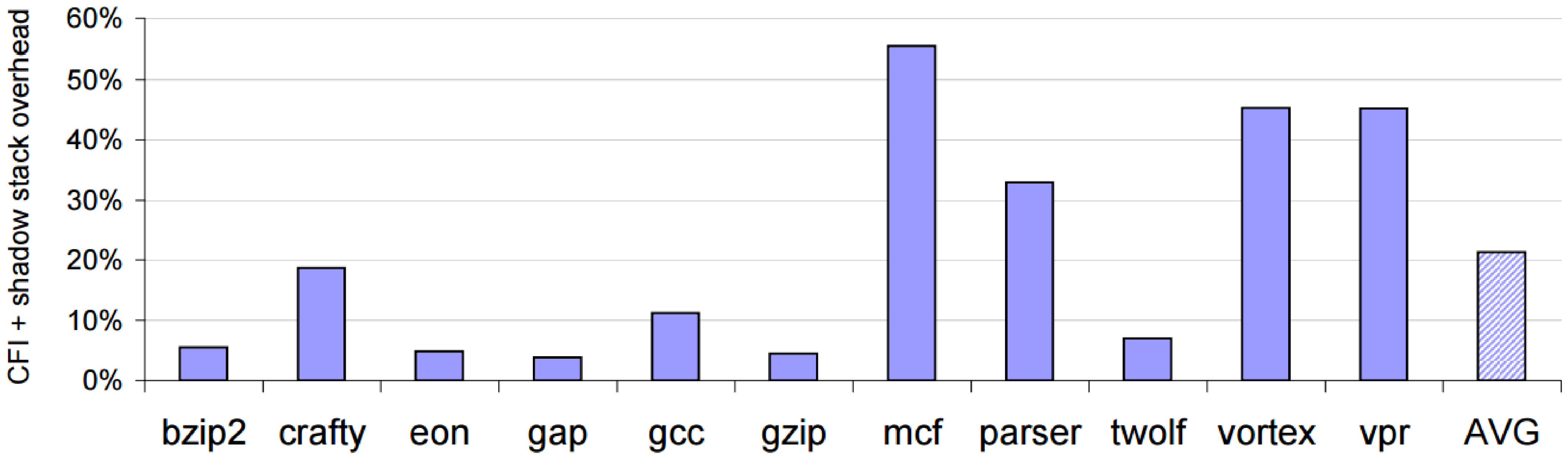


Image from control flow integrity, CCS 2005

CFI Practical Implication?

- CFI on binary code is difficult
  - Subtlety of Vulcan

- CFI is slow

CFI on Binary: Legacy Code

- CFG reconstruction from binary is difficult
- Indirect jumps?

CFI on Binary: Bypassing CFI

- Dynamically generated code
  - Self modifying code (e.g., packing)
  - JIT compiled code

- CFI is not perfect anyways

CFI Practicality: Coarse-Grained CFI

- Practical Control Flow Integrity and Randomization for Binary Executables, ***Oakland 2013***
- Control Flow Integrity for COTS binaries, ***USENIX Security 2013***
- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, ***USENIX Security 2013***
- ROPecker: A Generic and Practical Approach for Defending against ROP attacks, ***NDSS 2014***

CFI Practicality: Coarse-Grained CFI

- P

- Reduce the # of labels to check
  (e.g., checks if a function returns to a call-preceded instruction)

- Employ behavioral heuristics to quickly check integrity (e.g., detect gadget-like sequences)

Attacking Coarse-Grained CFI

- Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, ***USENIX Security 2014***
- Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard, ***USENIX Security 2014***
- Out of Control: Overcoming Control-Flow Integrity, ***Oakland 2014***

CFI is Now in Major Compilers

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM,
USENIX Security 201

Protect forward edges with
VTV (VTable Verification)
IFCC (Indirect Function Call Checker)
FSAN (Indirect Function Call Sanitizer)

Performance vs. Security

Still not solved ☹

Implication of Shadow Call Stack

What if we have a perfect CFI, but without shadow call stack?

We can return to some functions that are not in the CFG

CFI Without Shadow Call Stack

- ROP may be possible, but not easy

- Return-into-libc is much easier though
  - `system` calls `memcpy`
  - If a vulnerable function can call `memcpy`, then we can jump back to system (with a dispatcher function)

Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, ***USENIX Security 2015***

Dispatcher Function

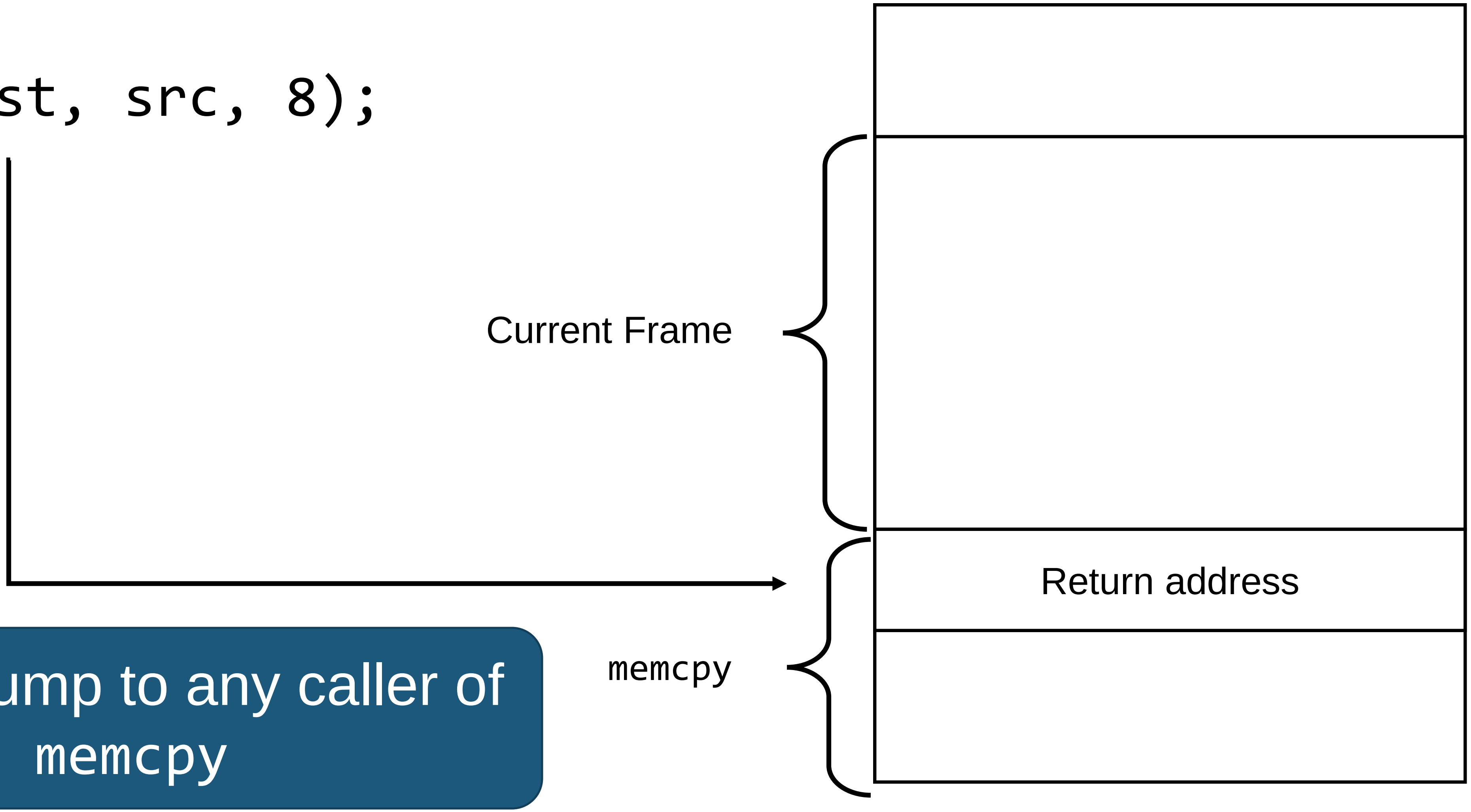
A function that can overwrite its own return address when given arguments supplied by an attacker.

Any function that has a "write-what-where" primitive

E.g. memcpy, printf, fputs, etc.

memcpy

```
memcpy(dst, src, 8);
```



We can jump to any caller of `memcpy`

Eval: CFI Without Shadow Call Stack

- Anaylzed 6 apps.

- Successfully exploited 5 apps. assuming fully precise static CFI without shadow call stack

What about Fully Precise CFI?

- We now assume we use shadow call stack
- We cannot use dispatcher functions any more
- Are we secure now?

Printf-Oriented Programming

- A single call to `printf` allows an attacker to perform Turing-complete computation!
- Assume we can fully control the arguments to `printf`
- Can bypass fully precise CFI

Printf-Oriented Programming

- Memory read: %s
- Memory write: %n
- Conditional?

Conditional

```
if ( *c ) {
  *t = x;
}
```

Single byte write that overwrite Q
If NULL byte is written, printf terminates

Address of Q

```
"%s%hhnQ%*d%n", c, s, x-2, 0, t
```

Width specifier

Turing Complete!

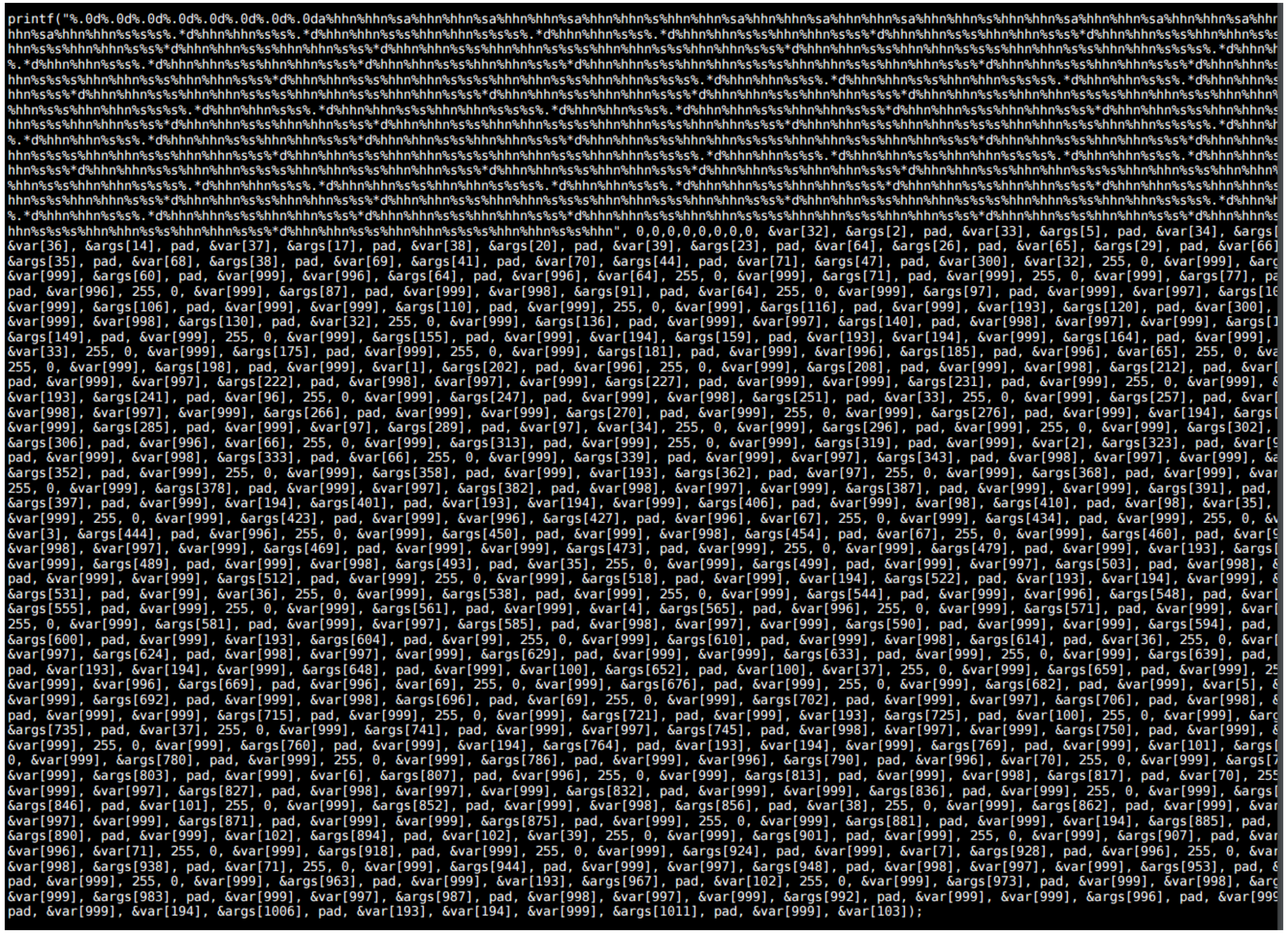


Image from the slides of Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, USENIX Security 2015

Printf-Oriented Programming

- Single call to printf is enough to run any arbitrary code

- No need to violate CFI

Question

Do you think printf-oriented-programming-based attacks hijack control flow?

Questions?