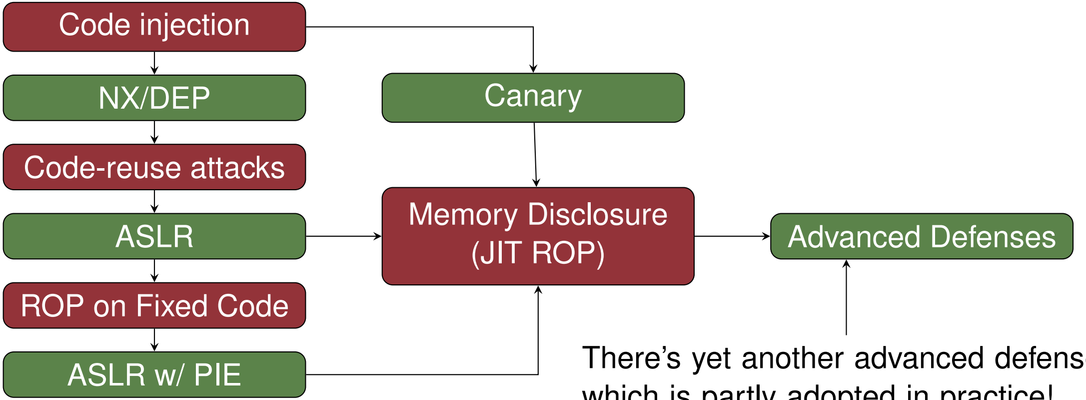
Lec 14: CFI

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

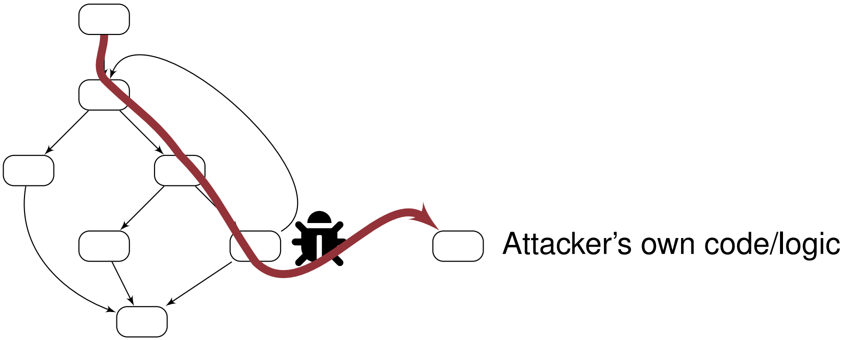
Attack and Defense So Far ...



There's yet another advanced defense, which is partly adopted in practice!

CFI

Control Flow Hijack Exploit



CFI: The Motivation

Can we Enforce the Integrity of Control Flows?

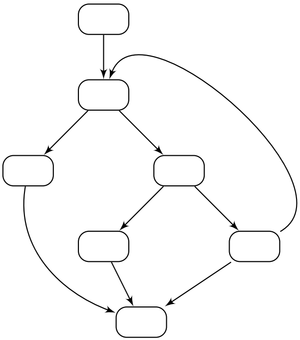
CFI Policy

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ***ahead of time***[1].

[1] Quote from Control Flow Integrity, ***CCS 2005***

CFG

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



Basic Block

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[2].

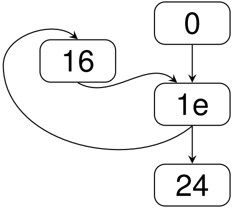
[2] Quote from Modern Compiler Implementation.

Exercise: CFG

```
 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
```

Exercise: CFG

```
 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
```



Key Idea of CFI

Any execution should follow control paths of the CFG.

If not, it is a control-flow hijack!

CFI Assumptions

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

Why?

Enforcing CFI

- Give a unique ID for each destination.
- For all branch instructions, check destination IDs before taking the branch.

CFI Instrumentation

	Source		Destination	
Original	FF E1	jmp ecx	8B 44 24 04	mov eax, [esp+4]
Version 1	81 39 78 56 34 12	cmp [ecx], 12345678h		
	75 13	jne error_label	78 56 34 12	.data 12345678h
	8D 49 04	lea ecx, [ecx + 4]	8B 44 24 04	mov eax, [esp+4]
	FF E1	jmp ecx		
Version 2	B8 77 56 34 12	mov eax, 12345677h		
	40	inc eax	3F 0F 18 05	prefetchnta
	39 41 04	cmp [ecx + 4], eax	78 56 34 12	[12345678h]
	75 13	jne error_label	8B 44 24 04	mov eax, [esp+4]
	FF E1	jmp ecx		

Why version 2 is more secure than version 1?

CFI Challenge

Indirect branches can have more than one jump target. In the previous example, `jmp ecx` can have multiple jump targets!

1. How about checking multiple IDs in a single branch?
2. How about assigning the same ID to different targets?

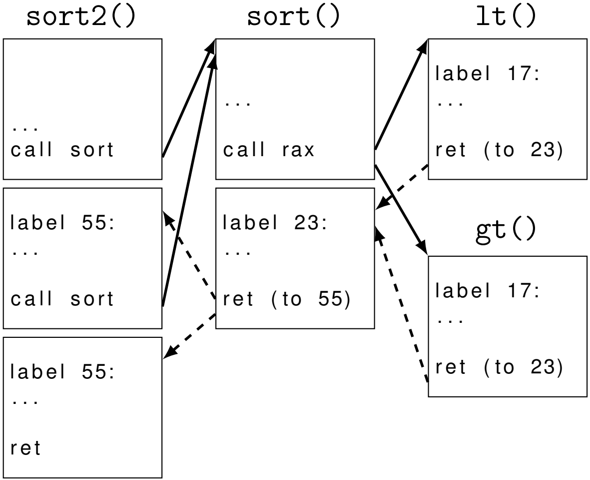
Checking Multiple IDs in a Single Branch

- How do we know which ID is correct in which context?
- We could consider it safe if one of the IDs matches, but then it is no different from using a single ID.

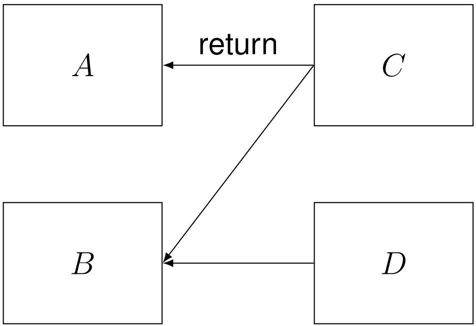
So, practical solution would be to use a single ID for multiple targets.

Single ID Multiple Targets

```
bool lt(int x, int y) { return x < y; }
bool gt(int x, int y) { return x > y; }
void sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

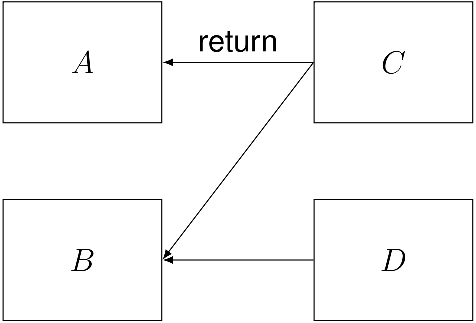


Problem: False Negatives



1. When D returns to A?
2. When C returns to A, even if it should return to B in the current context?
    - Context-insensitivity.

Problem: False Negatives



1. When D returns to A?
2. When C returns to A, even if it should return to B in the current context?
   - Context-insensitivity.

Will using multiple IDs help here?

Partial Solution: Shadow Call Stack[3]

- In each function prologue, store the return address in a separate memory area (shadow stack).
- In each function epilogue, check if we are returning to the proper address.

This is a context-sensitive solution ***only*** for backward (return) edges.

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

CFI with Shadow Call Stack

	Source	Destination
Original	`call eax ; call function ptr`	`...` `ret return`

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

CFI with Shadow Call Stack

	Source	Destination
Original	`call eax ; call function ptr`	`...` `ret return`

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

Why not just use a `ret` instruction?

Time of Check to Time of Use Problem

Example

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

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

Time of Check to Time of Use Problem

Example

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

TOC

TOU

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

TOCTOU

	Source	Destination
Original	`call eax ; call function ptr`	`...` `ret return`

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

TOCTOU can happen here if `ret` is used.

CFI Runtime Overhead

CFI + shadow stack overhead $\approx$ 20% on average[4].

[4] Control Flow Integrity, ***CCS 2005***

CFI Practical Implication

- Achieving CFI is infeasible in practice.
- There is a practical (but unsafe) solution with shadow call stack, but it incurs significant overhead still.
- CFI is a source-level solution, and achieving CFI on binary code is even more difficult.

CFI Limitation

- Hard to apply for legacy binary code.
- Cannot apply for JIT-compiled code.

CFI Research

- ***Coarse-grained*** CFI: make it more scalable and adoptable by making it less secure.
  - Practical Control Flow Integrity and Randomization for Binary Executables, ***Oakland 2013***.
  - Control Flow Integrity for COTS binaries, ***USENIX Security 2013***
  - ROPecker: A Generic and Practical Approach for Defending against ROP attacks, ***NDSS 2014***
  - Microsoft EMET (ROPGuard).
- H/W support: make it more scalable with H/W support.

(Source-Level) CFI is Now in Major Compilers

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, ***USENIX Security 2014***

- ***VTV*** (VTable Verification): checks the VTABLE hierarchy to check whether virtual function call is valid or not.
- ***IFCC*** (Indirect Function Call Checker) and ***FSAN*** (indirect Function call Sanitizer) dynamically check the types of each function to see if it has the same type as declared in the function pointer.

Coarse-grained CFI Example

Suppose we want to make CFI more scalable by removing the back edge protection (i.e., shadow call stack). What would be the security implication of this approach?

CFI without Shadow Call Stack

- ROP may be possible at a very limited way.
- However, return-to-LIBC is extremely easy! Why?[5]

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

CFI without Shadow Call Stack

- ROP may be possible at a very limited way.
- However, return-to-LIBC is extremely easy! Why?[5]
  - `system` internally calls `memcpy`.
  - If a vulnerable function calls `memcpy`, we can return to `system` under the coarse-grained CFI by modifying its own stack.

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

Idea: 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 could be a dispatcher function. For example, `memcpy`, `printf`, etc.
- `memcpy` (if the user can supply arbitrary input) can modify its own stack and return to any address.

Coarse-grained CFI could be extremely vulnerable!

Beyond CFI

CFI Recap

The idea is straightforward, but realizing it is ***not at all*** easy.

Fully Precise CFI?

Let's assume that we can realize fully precise CFI. Can we say that we are safe against memory corruption? What else can attackers do?

A New Exploitation

A new exploitation technique named ***printf-oriented programming*** can bypass fully precise CFI.

- A single call to `printf` allows an attacker to perform Turing-complete computation.
- Assuming that we can fully control the arguments to `printf`.

Printf-Oriented Programming

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

Conditionals in POP

Can we represent the conditional statement below in POP?

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

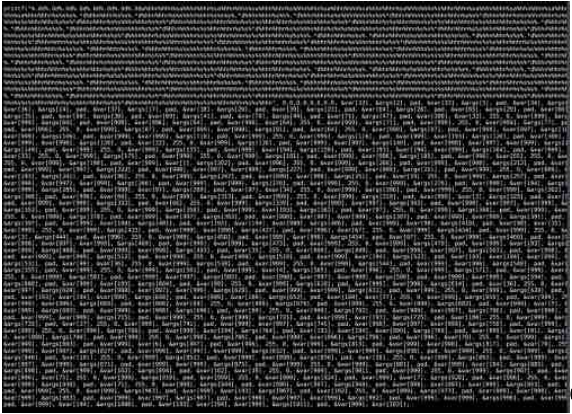
Single-byte write that overwrites `Q`. When a NULL byte is written to it, `printf` terminates.

Address of `Q`

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

Width specifier

POP is Turing-Complete!


[6]

[6] Ripple carry adder implementation in POP. Image taken from the slides of Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, ***USENIX Security 2015***

POP

- Single call to `printf` is enough to run any arbitrary code.
- No need to violate CFI.

Q: Do POP-based exploits hijack control flows?

Question?

Exercise

Read Clang's manual for CFI[7] and discuss how each different option implements different CFI policies.

[7] https://clang.llvm.org/docs/ControlFlowIntegrity.html