Lecture 6 : Control Flow Integrity (CFI)

Alexandre Bartel

2019

Previously...

Previously... in Lecture 1 (Introduction)

- Software Development Life-cycle
- Vulnerability Life-cycle
- Vulnerability Disclosure

Previously... in Lecture 2 (Buffer Overflow)

- A buffer on the stack
- Return address on the stack
- Overwrite return address
- Jump to shellcode on the stack

Previously... in Lecture 3 (ROP)

- NX bit (stack non-executable)
- Gadgets in already loaded code
- Chain gadgets (addresses of gadgets and data on the stack)
- Only data on the stack

Previously... in Lecture 4 (ASLR)

- Randomize code segment at program start
- Breaks gadget chains
- Bypass with information leak (e.g, vulnerability)

Feedback on lab1 and lab2.

CFI [1] [2]

[1] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC), 13(1), 4.

[2] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J. (2005, November). A theory of secure control flow. In International Conference on Formal Engineering Methods (pp. 111-124). Springer, Berlin, Heidelberg.

What is Control Flow Integrity?

- A mechanism to ensure that control flows executed are the ones intended by the original program.

What is Control Flow?

- A control flow graph (CFG) [1] is a graph representation of all paths that might be traversed through a program during its execution.
- Nodes are statements / basic blocks
- Directed edges represent jumps
- Usually a CFG is for a single function
- Inter-procedural CFG is for the whole program
- Call-graph represents relationships between functions
- Call-site
- Return-site

[1] Allen, Frances E. "Control flow analysis." ACM Sigplan Notices. Vol. 5. No. 7. ACM, 1970.

Why CFI?

Without CFI an attacker can use any gadget.

Why CFI? (cont)

With CFI, the set of available gadgets is greatly reduced.

How to compute the CFG?

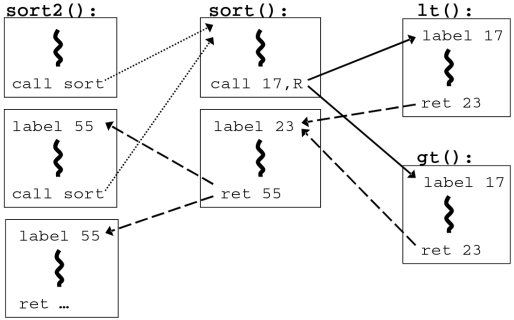
- By analyzing source code
- By execution profiling
- By analyzing the binary ←

How to compute the CFG?

```
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 );
}
```



Binary Instrumentation

New machine-code instructions, with an immediate operand ID

- An effect-free label ID instruction
- A call instruction call ID, DST that transfers control to the code at the address contained in register DST only if that code starts with label ID
- A corresponding return instruction ret ID

Binary Instrumentation (cont)

When to instrument.

- CFI instrumentation does not affect direct function calls.
- Only indirect calls require an ID-check.
- Only functions called indirectly (such as virtual methods) require the addition of an ID.
- An ID must be inserted after each function callsite, whether that function is called indirectly or not.

Binary `jmp` Instrumentation

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				

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

Binary `call`/`ret` Instrumentation

Function Call

Opcode bytes	Instructions	
FF 53 08	call [ebx+8]	; call fptr

are instrumented using `prefetchnta` destination IDs, to become

8B 43 08	mov eax, [ebx+8]	; load fptr
3E 81 78 04 78 56 34 12	cmp [eax+4], 12345678h	; comp w/ID
75 13	jne error_label	; if != fail
FF D0	call eax	; call fptr
3E 0F 18 05 DD CC BB AA	prefetchnta [AABBCCDDh]	; label ID

Function Return

Opcode bytes	Instructions	
C2 10 00	ret 10h	; return

8B 0C 24	mov ecx, [esp]	; load ret
83 C4 14	add esp, 14h	; pop 20
3E 81 79 04	cmp [ecx+4],	; compare
DD CC BB AA	AABBCCDDh	; w/ID
75 13	jne error_label	; if!=fail
FF E1	jmp ecx	; jump ret

Figure 3: The CFI instrumentation of x86 `call` and `ret` used in our implementation.

Imprecision

Naive CFI may be imprecise.

- Same ID is used for virtual calls and returns

Imprecision: Solution (very briefly)

Shadow Stack.

- Stack to keep information about the current call stack
- This stack must be protected (to prevent the attacker from modifying it)
- For instance, protected by using an isolated code segment for CFI code (x86 specific)

Evaluation

Target Applications for the Evaluation

SPEC [1] 2000

"SPEC CPU2000 is the next-generation industry-standardized CPU-intensive benchmark suite. SPEC designed CPU2000 to provide a comparative measure of compute intensive performance across the widest practical range of hardware. The implementation resulted in source code benchmarks developed from real user applications. These benchmarks measure the performance of the processor, memory and compiler on the tested system."

[1] Standard Performance Evaluation Corporation

CFI Overhead

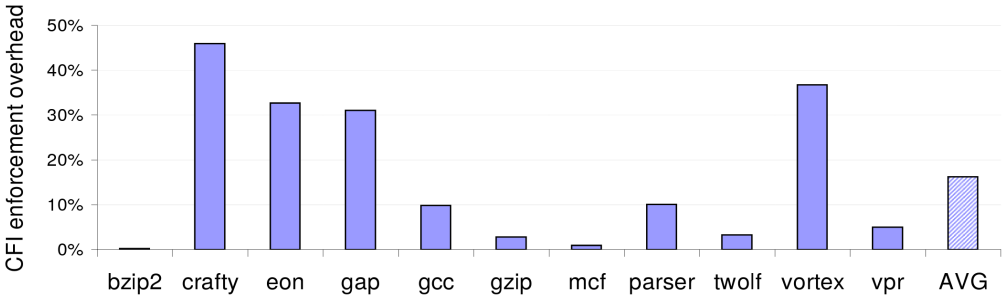


Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

CFI (shadow stack) Overhead

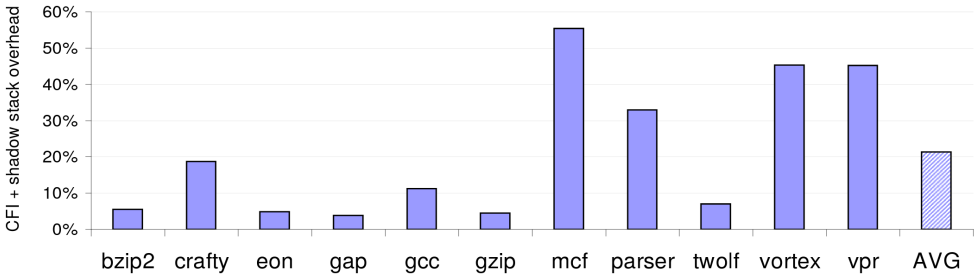


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

Conclusion

- CFI "greatly" reduces number of gadgets
- Software implementation brings an overhead too high to be used in practice
- Other solutions are currently being developed

Question?