

CSE 227 Computer Security

Stefan Savage, Fall 2021

Vulnerability Finding

Defenses: lost of kinds...

- Eliminate violation of runtime model
 - Better languages, code analysis
- Don't allow bad input
 - Input validation
- Detect overflow/overwrite of key data structures
 - Stack validation
 - Run-time bounds checking, pointer validation, etc
 - Reference monitors
 - Data-oriented anomaly detection
- Don't allow untrusted code to execute
 - Hardware protection, code signing
- Minimize invariants for making repeatable exploits
 - ASLR, code randomization, encrypted pointers
 - Detect payload insertion before it can be invoked
- Minimize impact of untrusted code running (sandboxing)

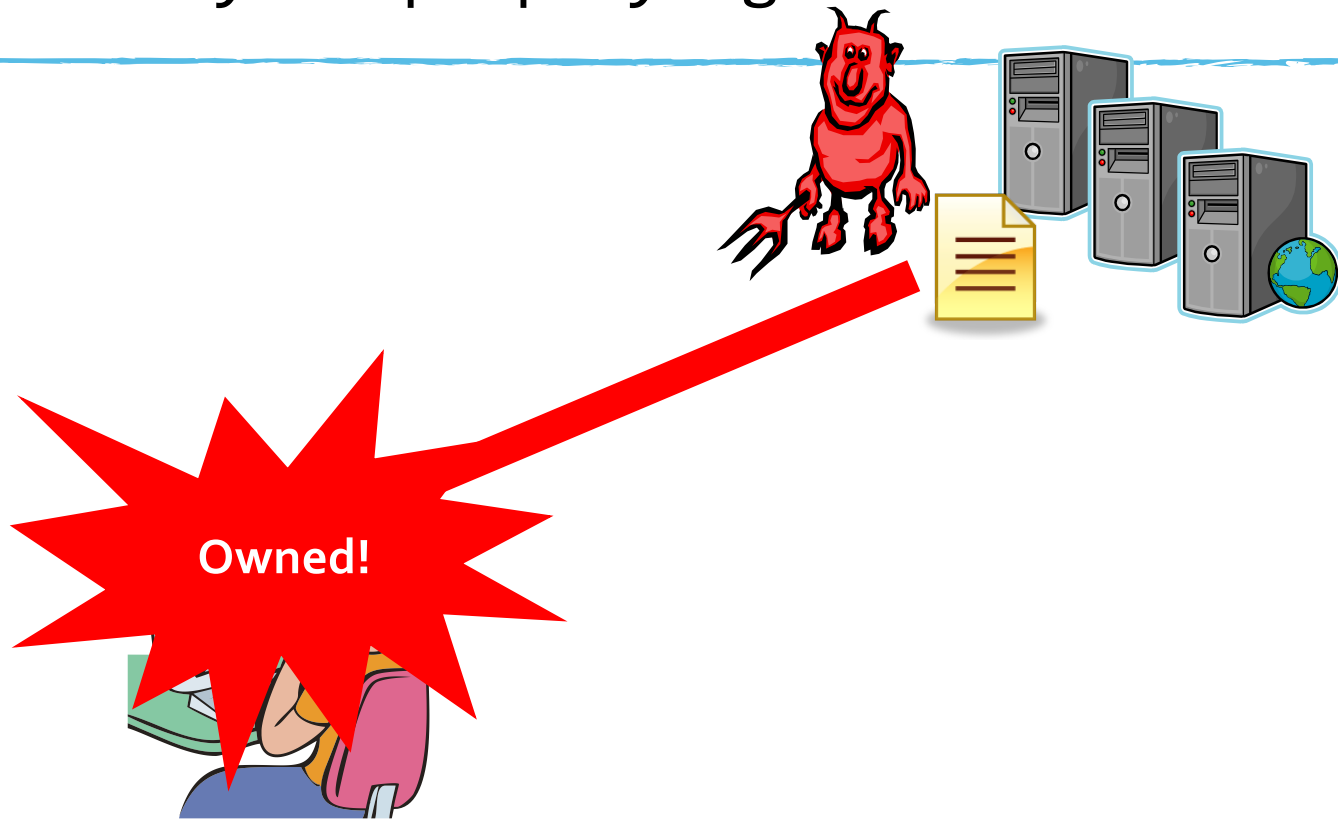
The game

- Assumptions
- What assumptions made by the system can the attacker undermine?
- What assumptions made by the attacker can the defender undermine?

Nozzle

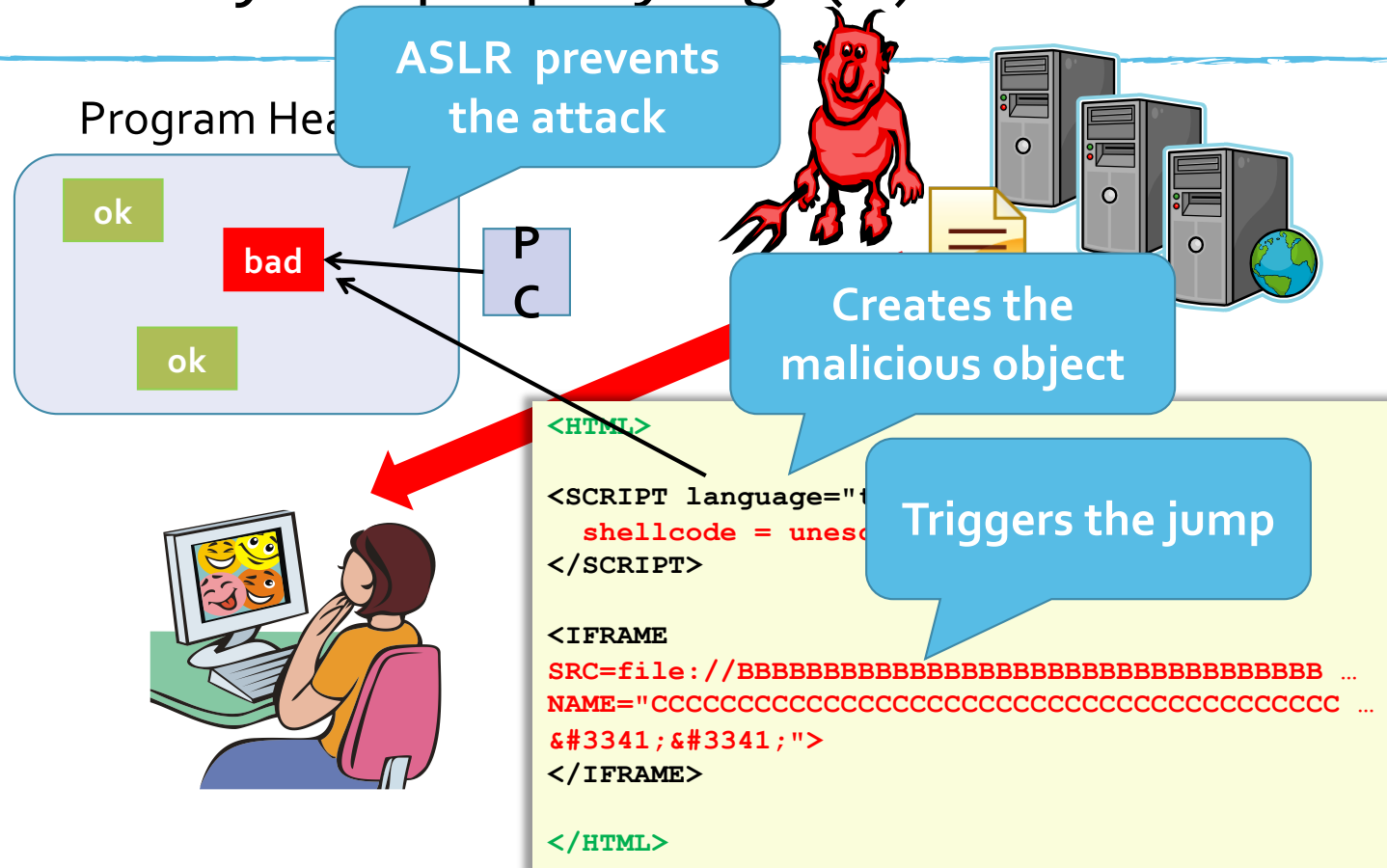
- Paruj Ratanaworabhan, Benjamin Livshits, Benjamin Zorn, **NOZZLE: A Defense against Heap-Spraying Code Injection Attacks**, USENIX Security 2009.
- What problem are they attacking? Why now?
- Who are these guys and why are they attacking this problem in this way?
- What is the goal? (i.e., how should you evaluate the paper?)

Drive-By Heap Spraying

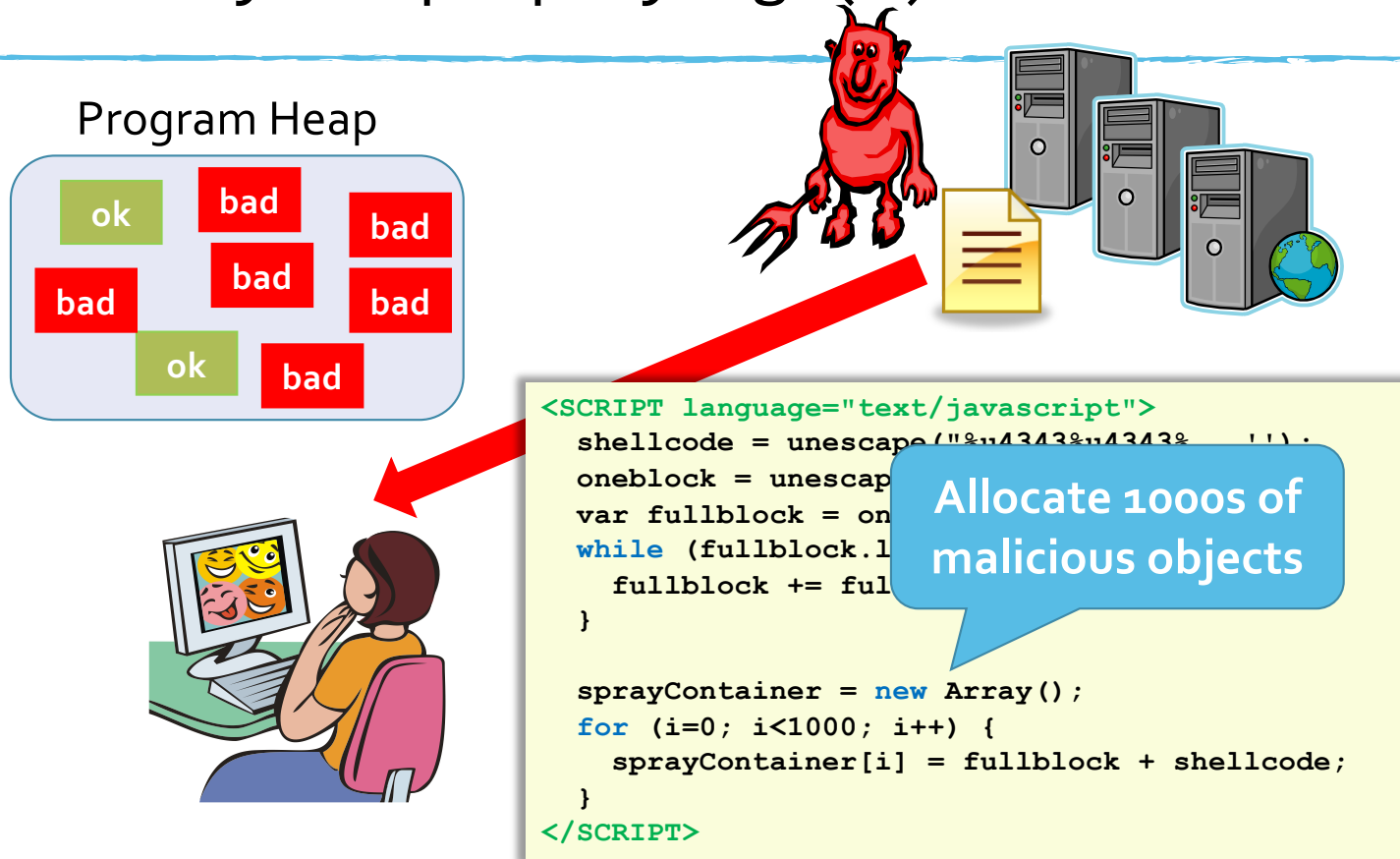


Slides courtesy Ratanaworabhan, Livshits, and Zorn

Drive-By Heap Spraying (2)

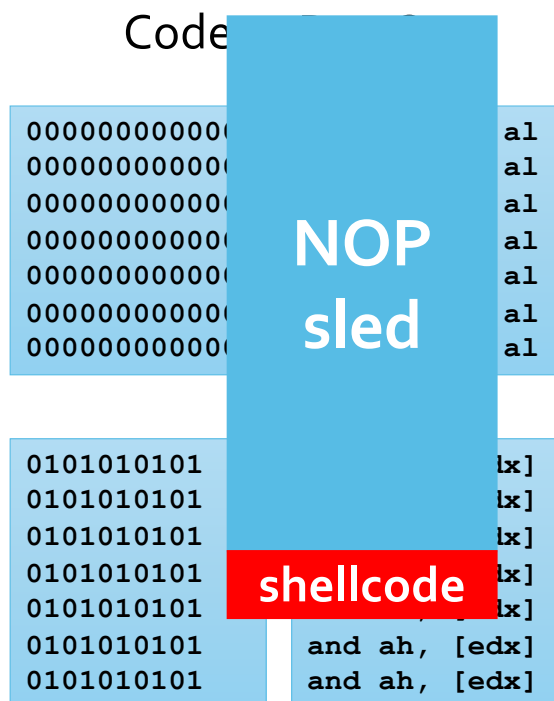


Drive-By Heap Spraying (3)



Local Malicious Object Detection

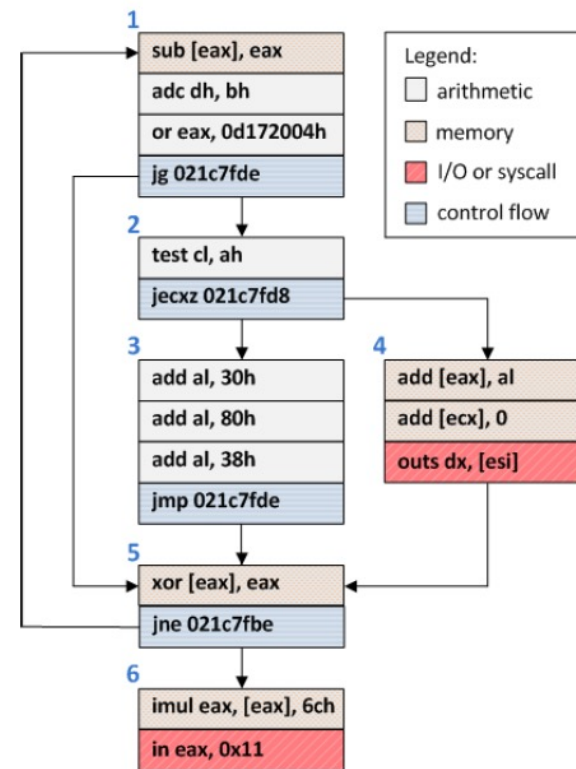
Is this object dangerous?



- Is this object code?
 - Code and data look the same on x86
- **Focus on sled detection**
 - Majority of object is sled
 - Spraying scripts build simple sleds
- Is this code a NOP sled?
 - Previous techniques did not look at heap
 - Many heap objects look like NOP sleds
 - 80% false positive rates using previous techniques
- Need stronger local techniques

Object Surface Area Calculation (1)

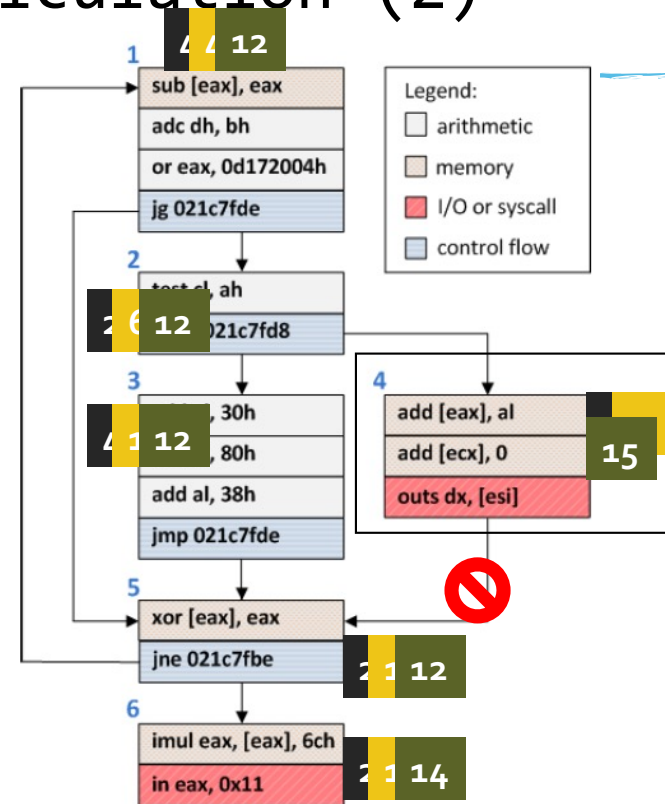
- Assume: attacker wants to reach shell code from jump to any point in object
- Goal: find blocks that are likely to be reached via control flow
- Strategy: use dataflow analysis to compute “surface area” of each block



An example object from visiting google.com

Object Surface Area Calculation (2)

- Each block starts with its own size as weight
- Weights are propagated forward with flow
- Invalid blocks don't propagate
- Iterate until a fixpoint is reached
- Compute block with highest weight



An example object from visiting google.com

Nozzle Global Heap Metric

Normalize to (approx):
 $P(\text{jump will cause exploit})$
 $NSA(H)$

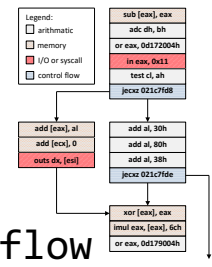
obj



build CFG

B_i

dataflow

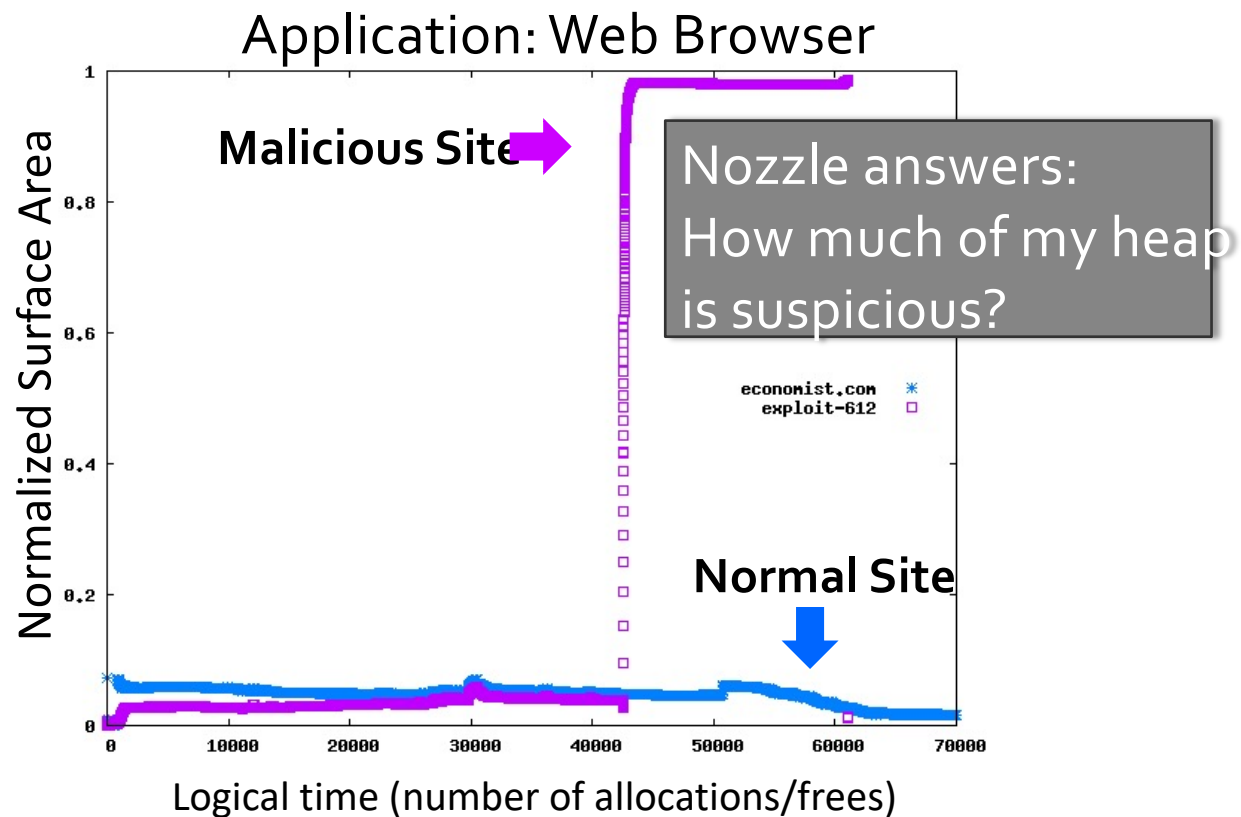


$SA(H)$
Compute threat
of entire heap

$SA(o)$
Compute threat of
single object

$SA(B_i)$
Compute threat of
single block

Nozzle - Runtime Heap Spraying Detection



Thoughts about Nozzle

- Assumptions?
- Limitations?
- Costs?

Follow-on work

- Zozzle (USENIX Sec 2011)
 - Nozzle has high overhead, scales badly with number of pages.
 - Use Nozzle to train ML classifier on JS; fast 1MB/sec per server
- Rozzle (IEEE Security and Privacy, 2012)
 - Challenge: vulnerability not expressed (e.g., your browser isn't using Acrobat version 8.23)
 - Huge number of combinations
 - Automatically explore multiple execution paths (similar to guided model checking)

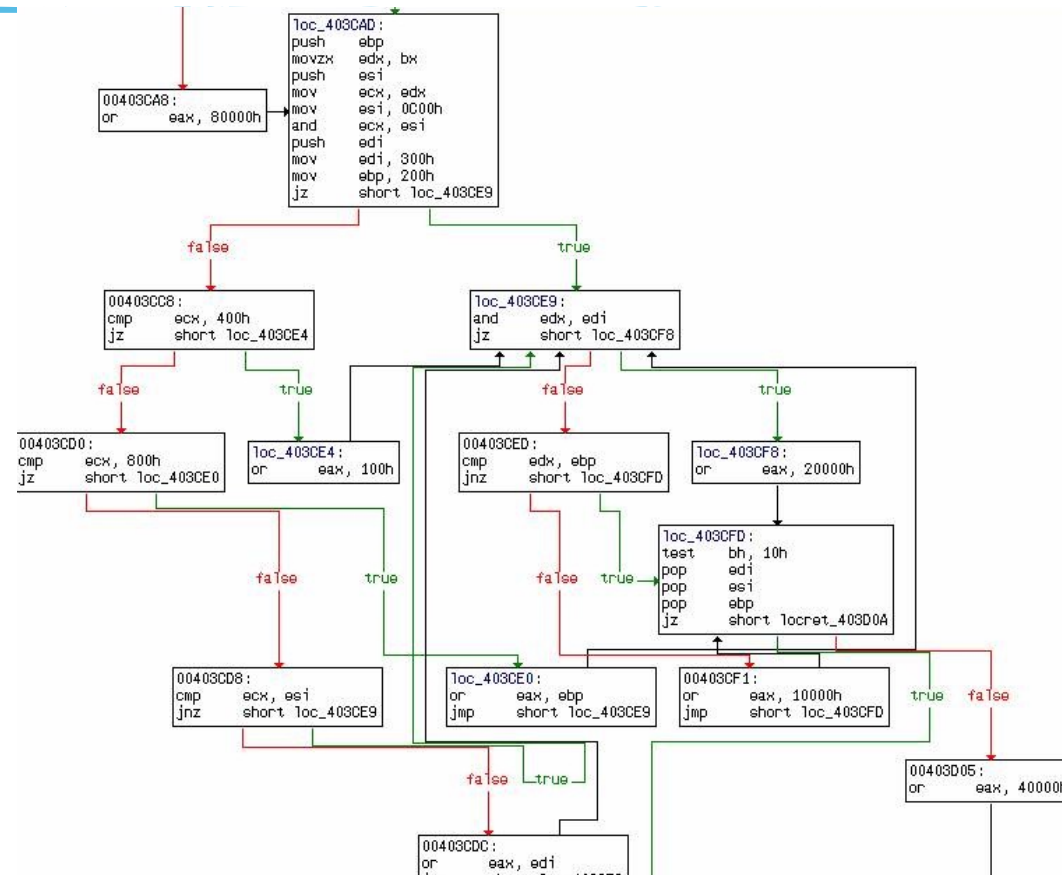
CFI

- Martin Abadi, Mihai Budiu, Ulfar Erlingsson and Jay Ligatti, **Control Flow Integrity: Principles, Implementations and Applications**, ACM CCS 2005.
- What problem are they attacking? Why now?
- Who are these guys and why are they attacking this problem in this way?
- What is the goal? (i.e., how should you evaluate the paper?)

Goal

- Fail-stop detection of any exploit that subverts a programs intended control flow
 - Stack overflow, heap overflow, format string, integer overflow, return-oriented programming, etc
- What defines the intended control flow of a program?

CFI: Code as Set of Basic Blocks



Control Flow Integrity

- Focus is on protecting indirect transfer of control flow instructions.
- **Direct** control flow transfer:
 - Advancing to next sequential instruction
 - Jumping to (or calling a function at) an address hard-coded in the instruction
 - These are static in code, so assume attacker can't control (if they can overwrite code segment its game over anyway)
- **Indirect** control flow transfer
 - Jumping to (or calling a function at) an address in register or memory
 - **Forward path**: indirect calls and branches (e.g., a function you are calling)
 - **Reverse path**: return addresses on the stack (returning from a called function)

Foundation: Control-Flow Integrity

- Need to know what instruction may execute to verify code as safe, but x86 makes this difficult

// ecx == 40001
jmp ecx

Some jump targets not known until runtime—may even point to the middle of an instruction

40000: 05 **CF** 01 00 00 **add eax, 1CFh**

Processor executes code as:

CF **iretd**

A blue curved arrow originates from the **ecx** operand of the **jmp** instruction and points to the **CF** byte of the **add** instruction, illustrating a jump to the middle of an instruction.

Control Flow Integrity: basic idea

- Computer control flow graph (CFG) for program
- Restrict all control transfers to the control flow graph
 - State control flow, no problem
 - Dynamic control flow
 - Assign **labels** to all indirect jumps and their targets
 - Before taking an indirect jump, validate that target label matches jump site
 - Like stack canaries, but for control flow targets
- If you're about to transfer to an address that can't provide evidence that it's a valid edge in the CFG then crash

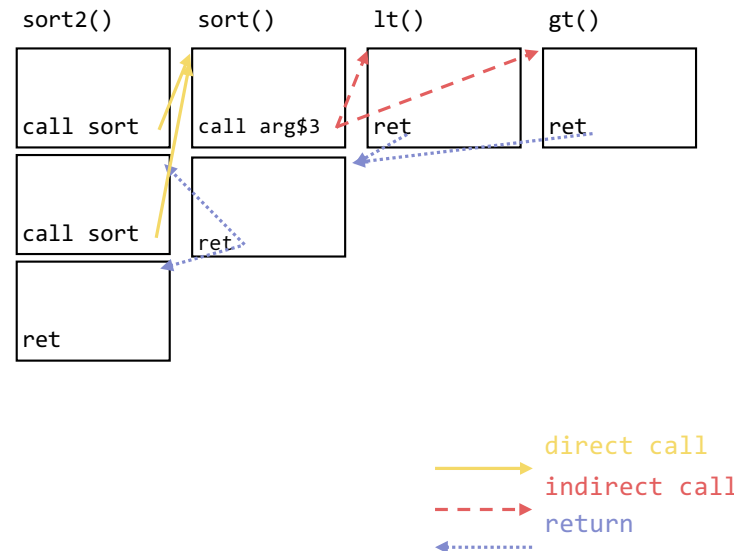
What's a legitimate target?

Look at the program control-flow graph (CFG)!

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

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

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

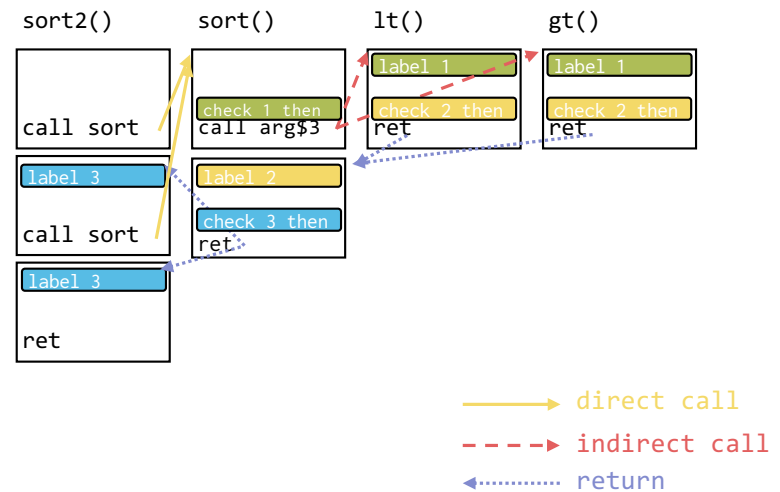


Fine grained CFI (Abadi et al.)

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

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

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

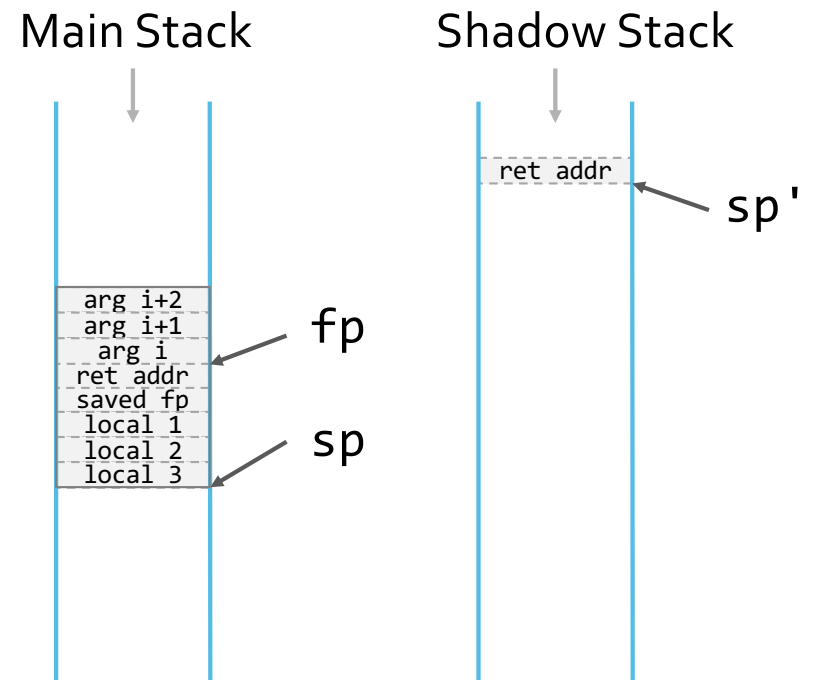


Some tricky details: ambiguity

- Forward path:
What if A calls C and B calls C or D?
 - If we use same tag for C and D then allow A to call D
 - Options:
 - Code duplication
 - Multiple tags per target
- Similar problem on return path:
 - C is called by A and B... what tag should be written?
 - If a static tag, this allows C to return to B after being called by A
 - Is this a real problem?

Shadow Stack

- Shadow Stack
 - On function entry, save a [shadow] copy of function call control flow data (return addresses and frame pointers) into another location
 - On function exit, compare the version on the stack to the shadow copy
- Requires compiler support
- Requires hardware support to be fast
 - Intel CET Shadow Stack starting with Tiger Lake



Threat model & Assumptions

- Attacker can arbitrarily read and modify any *data* in memory
- Can change registers, except PC
(and, during checks, registers used for address checks)
- Unique IDs
 - We can create unique bit patterns in the code that do not appear anywhere else
- Non-writable code
 - Can't modify code segment
- Non-executable data
 - Can't execute data (e.g., bss, heap, etc)
- Perfect knowledge of the CFG

CFI: *Example* of Instrumentation

Original code

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

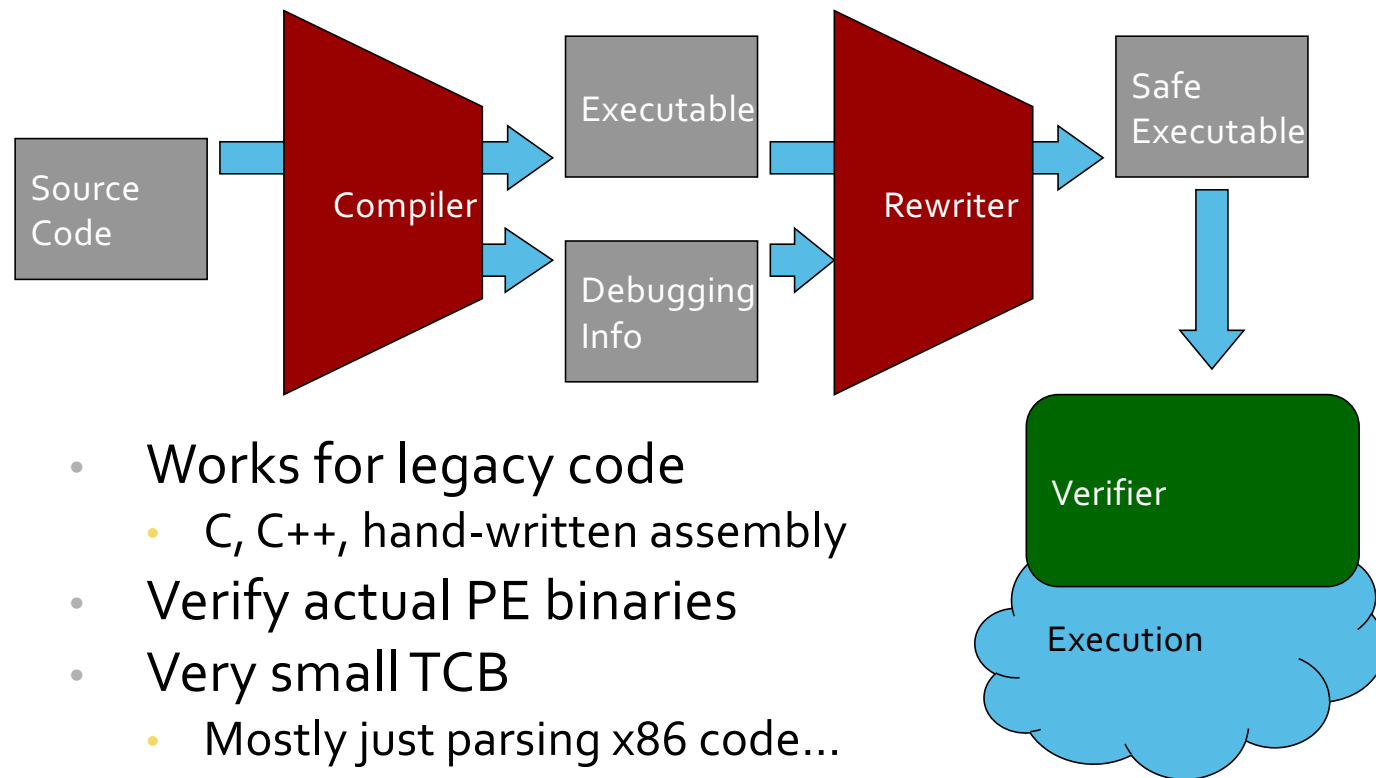
Instrumented code

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			

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

Abadi et al. Implementation



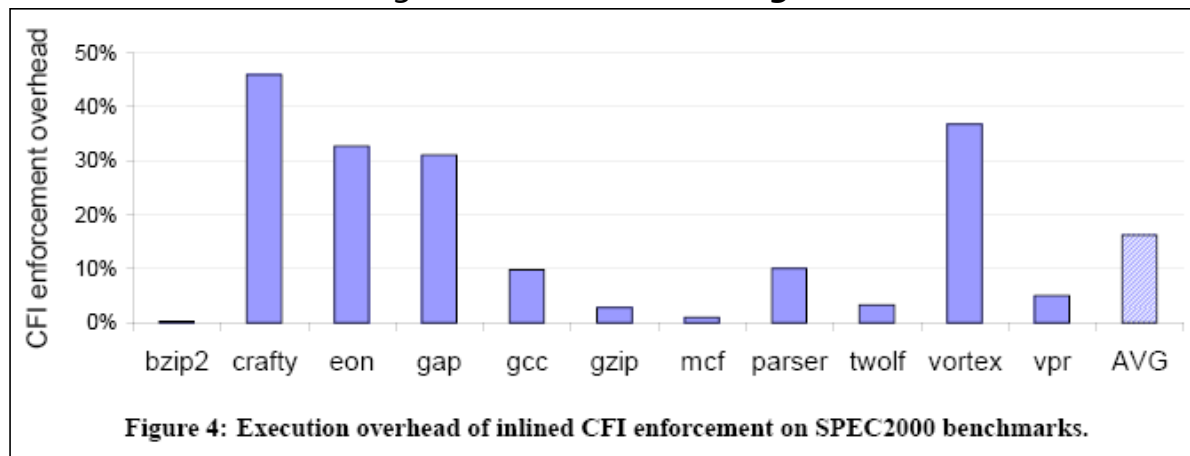
Verification

- Generally simple
 - Validate control flow instructions
 - Validate proper insertion of IDs and checking of IDs
 - Validate unique ID property

Implementation performance

- Measurements

- SPEC2000 benchmark
- CFG construction + CFI Instrumentation took about 10 seconds
- Binary size increased by an average 8%
- Benchmarks took 16% longer to execute **on average**



Thoughts about CFI

- Assumptions?
- Limitations?
- Costs?

Follow-on work

- Tradeoff strictness for performance (so-called “coarse CFI”)
 - MIP, CCFIR, binCFI, MoCFI: collapse labels (i.e. allow more control flow transfer edges)
 - Typically function entry points and return points
 - Jizillions of versions... probabilistic enforcement, crypto-based, etc
- Impact in practice
 - Software
 - Forward-edge CFI rolled into LLVM, gcc and Microsoft compilers
 - Used by Google for Android Linux kernel
 - Used by Microsoft in Windows 10 (Control Flow Guard)
 - Hardware
 - Intel CET
 - ARM PAC (not strictly for CFI, but used in some forward CFI implementations on ARM)

But...

- Carlini and Wager
 - Without stack integrity (i.e., a real shadow stack) can bypass everything
- Schuster et al – Counterfeit OO Programming
 - C++ semantics allows ROP-like attacks entirely out of existing C++ virtual function tables
 - Need forward edge precision that understands dynamic call graph and hence understands C++ semantics
- Overview for those who are interested at:
 - Payer and Carlini: New Memory Corruption Attacks: Why We Can't Have Nice things

For next time...

Two papers on software vulnerabilities at a higher level

- Does vuln finding improve security over time?
- Zero days