CS 380S

Memory Corruption Exploits

Vitaly Shmatikov

Slides on return-oriented programming courtesy of Hovav Shacham

Reading Assignment

- scut / team teso. "Exploiting format string vulnerabilities".
- Dowd. "Leveraging the ActionScript Virtual Machine".
- Chen et al. "Non-control-data attacks are realistic threats" (Usenix Security 2005).
- Roemer et al. "Return-oriented programming".
- Optional:
  - "Basic integer overflows", "w00w00 on heap overflows", "Once upon a free()..."

Variable Arguments in C

- In C, can define a function with a variable number of arguments
  - Example: void printf(const char* format, …)
- Examples of usage:

```
printf("hello, world");
printf("length of '%s' = %d\n", str, str.length());
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special %-encoded characters

- %d,%i,%o,%u,%x,%X – integer argument
- %s – string argument
- %p – pointer argument (void *)
- Several others

Implementation of Variable Args

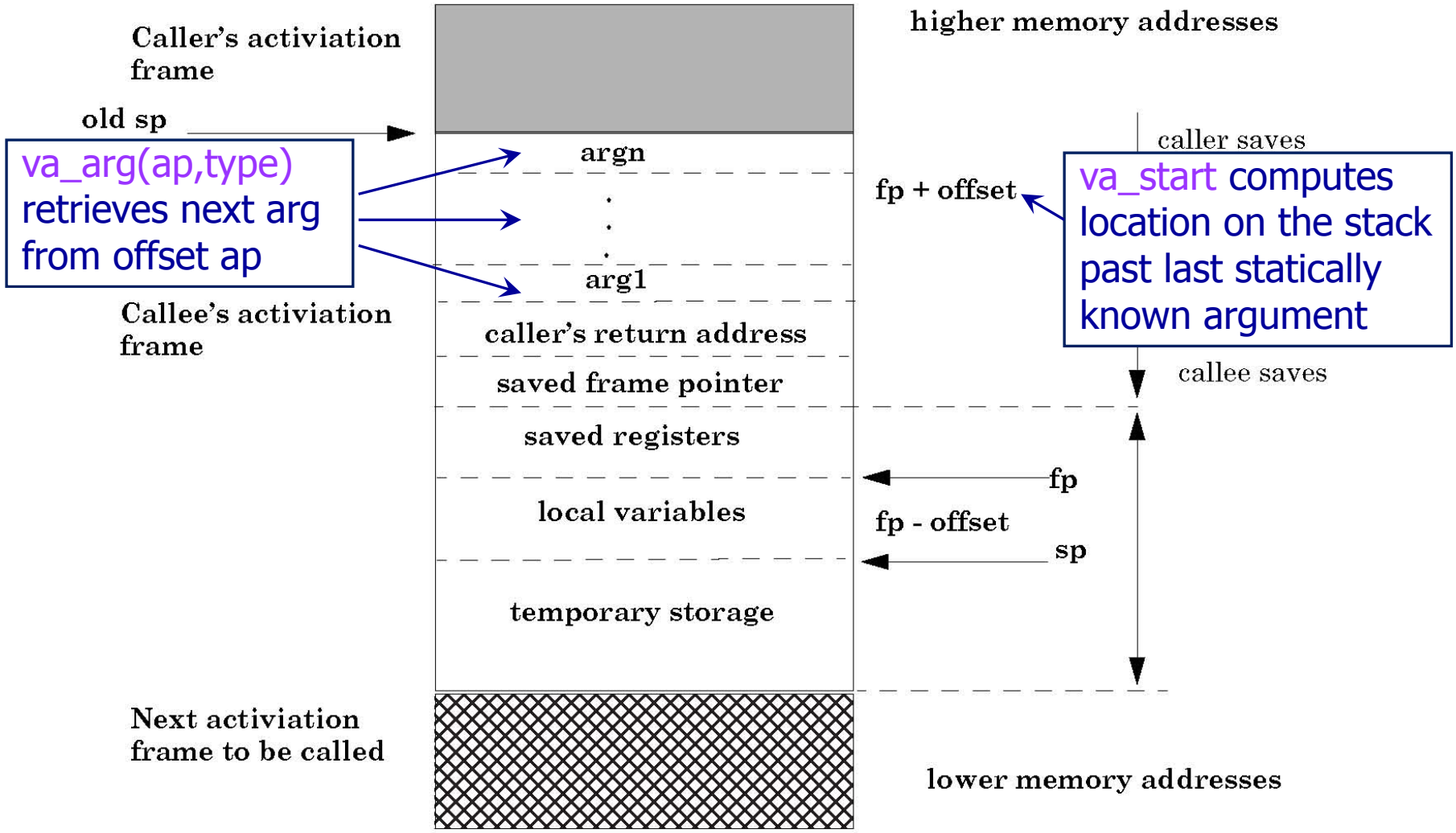
- Special functions va_start, va_arg, va_end compute arguments at run-time

```
void printf(const char* format, ...)
{
     int i; char c; char* s; double d;
     va_list ap;  /* declare an "argument pointer" to a variable arg list */
     va_start(ap, format);  /* initialize arg pointer using last known arg */

     for (char* p = format; *p != '\0'; p++) {
        if (*p == '%') {
           switch (*++p) {
              case 'd':
                 i = va_arg(ap, int); break;
              case 's':
                 s = va_arg(ap, char*); break;
              case 'c':
                 c = va_arg(ap, char); break;
              }
              ... /* etc. for each % specification */
           }
     }
     ...

     va_end(ap);  /* restore any special stack manipulations */
}
```

Frame with Variable Arguments



Format Strings in C

- Proper use of printf format string:

```
… int foo=1234;
  printf("foo = %d in decimal, %X in hex",foo,foo); …
```

  - This will print

```
foo = 1234 in decimal, 4D2 in hex
```

- Sloppy use of printf format string:

```
… char buf[13]="Hello, world!";
  printf(buf);
  // should've used printf("%s", buf); …
```

  - If buffer contains a format symbol starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer!

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
… printf("Overflow this!%n",&myVar); …
```

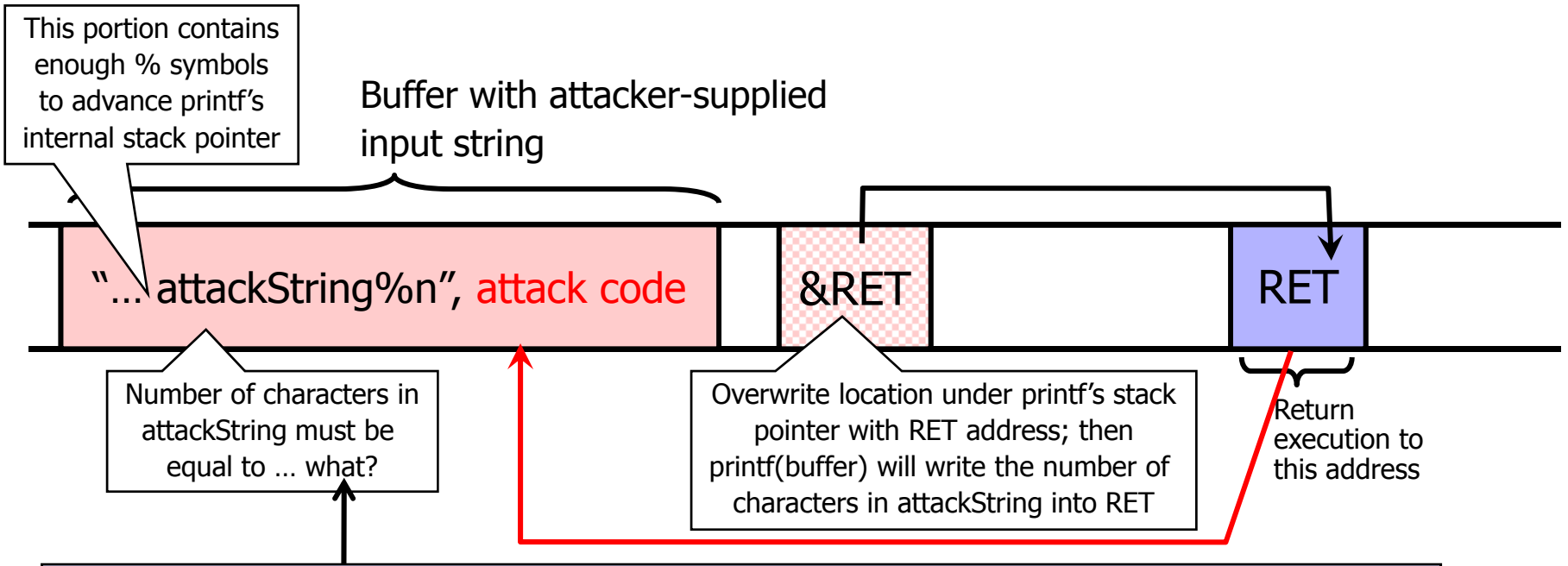
  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar

- What if printf does not have an argument?

```
… char buf[16]="Overflow this!%n";
  printf(buf); …
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as the address into which the number of characters will be written!

Using %n to Mung Return Address



C has a concise way of printing multiple symbols: %Mx will print exactly M bytes (taking them from the stack). If attackString contains enough "%Mx" so that its total length is equal to the most significant byte of the address of the attack code, this byte will be written into &RET.

Repeat three times (four "%n" in total) to write into &RET+1, &RET+2, &RET+3, replacing RET with the address of attack code.

- See "Exploiting Format String Vulnerabilities" for details

Bad Format Strings in the Wild

- Chen and Wagner study (2007)
  - "Large-scale analysis of format string vulnerabilities in Debian Linux"
- Analyzed a large fraction of the Debian Linux 3.1 distribution using CQual, a static taint analysis tool
  - 92 million lines of C and C++ code
  - Objective: find "tainted" format strings (controlled by user, yet used in printf and similar functions)
- Taint violations reported in 1533 packages
- Estimated 85% are real format string bugs

  (Why not 100%?)

Targets of Memory Corruption

- Configuration parameters
  - E.g., directory names that confine remotely invoked programs to a portion of the server's file system
- Pointers to names of system programs
  - E.g., replace the name of a harmless script with an interactive shell (not the same as return-to-libc)
  - System call interposition doesn't help unless it verifies call arguments and not just the name of the routine
- Branch conditions in input validation code

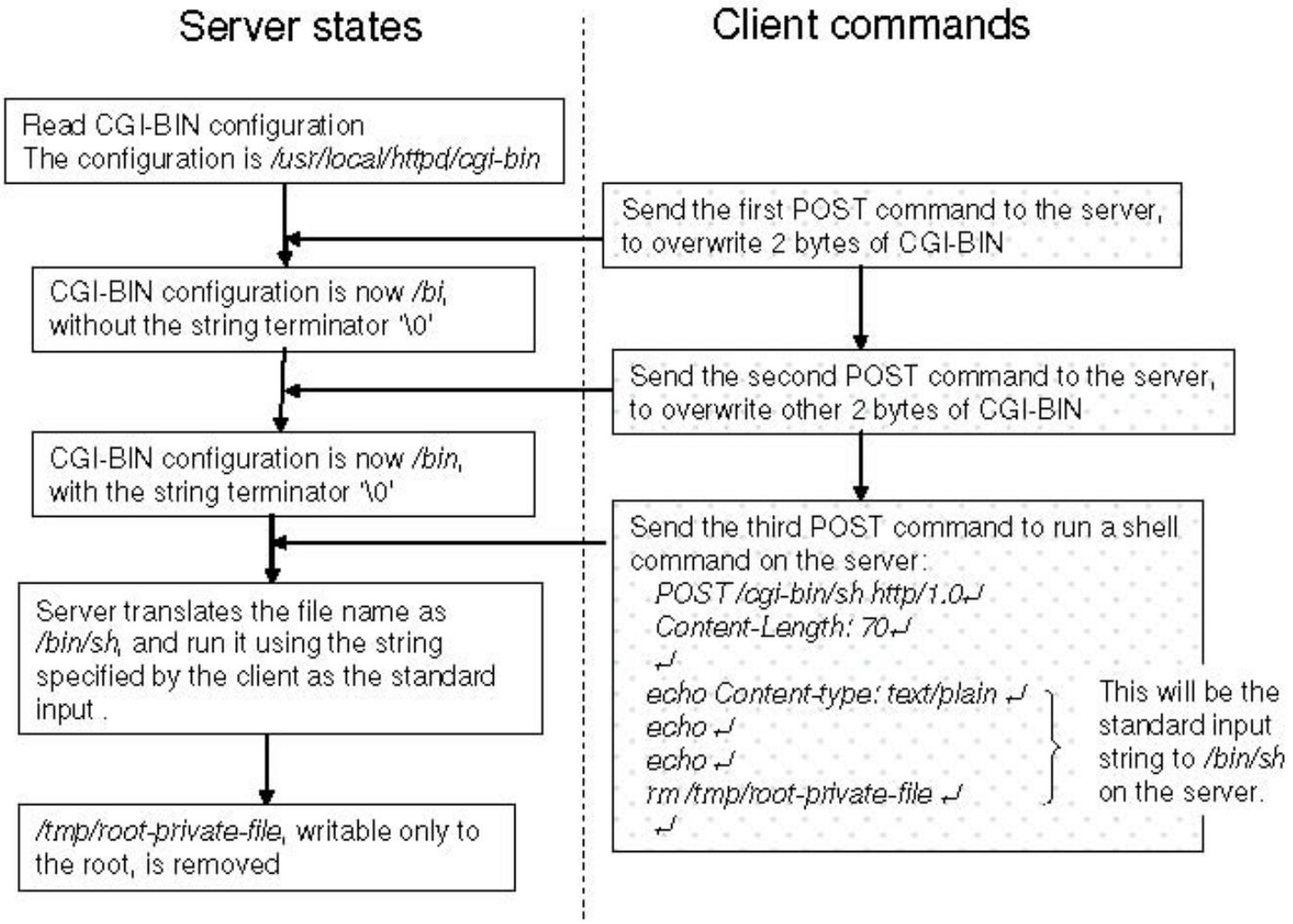
Example: Web Server Security

- CGI scripts are executables on the server that can be invoked by remote user via a special URL
  - http://www.server.com/cgi-bin/SomeProgram
- Don't want remote users executing arbitrary programs with Web server's privileges
  - Especially if the Web server runs with root privileges
  - Need to restrict which programs can be executed
- CGI-BIN is the directory name which is always prepended to the name of the CGI script
  - If CGI-BIN is /usr/local/httpd/cgi-bin, the above URL will execute /usr/local/httpd/cgi-bin/SomeProgram

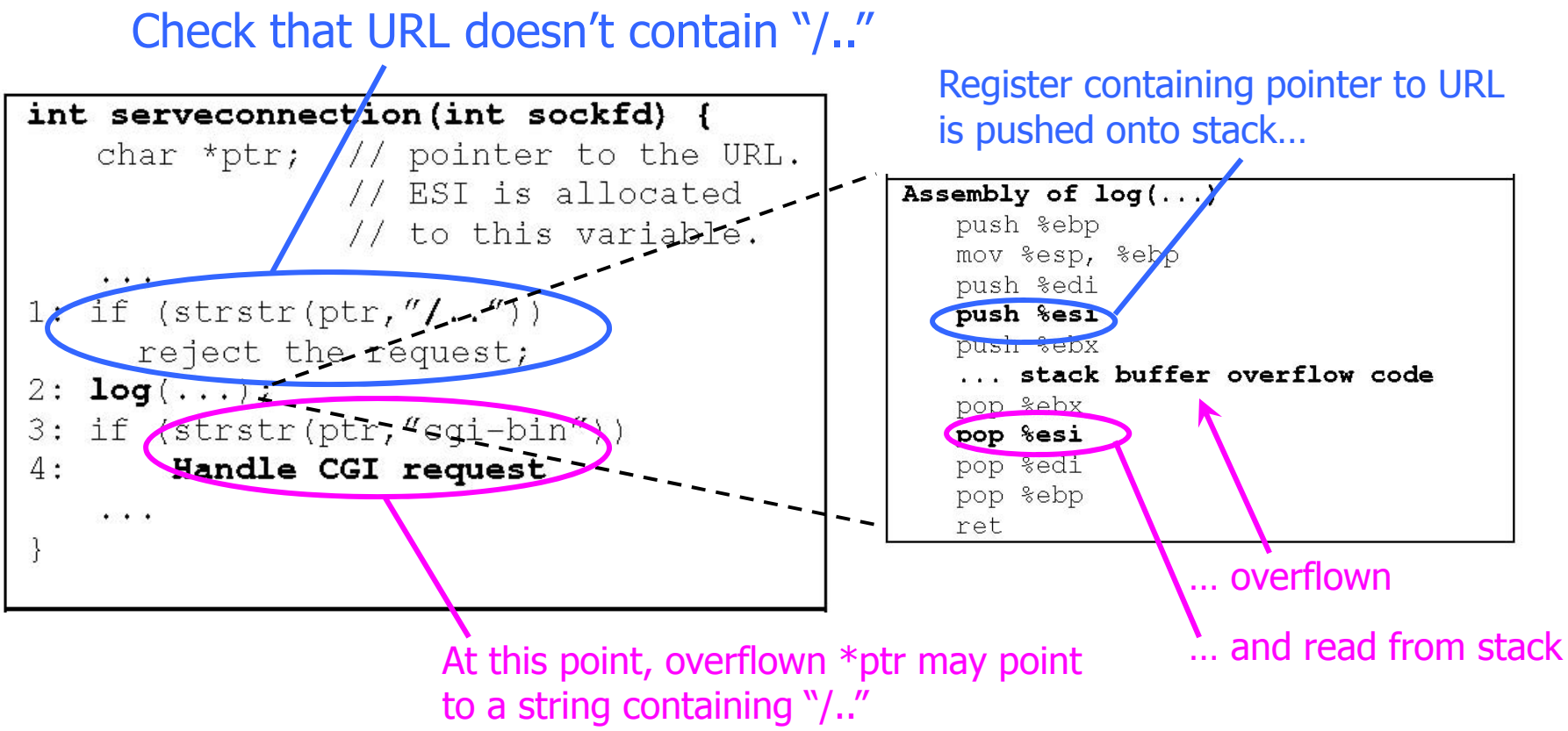
Exploiting Null HTTP Heap Overflow

- Null HTTPD had a heap overflow vulnerability
  - When corrupted buffer is freed, an overflown value is copied to a location whose address is read from an overflown memory area
  - This enables attacker to copy an arbitrary value into a memory location of his choice
- Standard exploit: copy address of attack code into the table containing addresses of library functions
  - Transfers control to attacker's code next time the library function is called
- Alternative: overwrite the value of CGI-BIN

Null HTTP CGI-BIN Exploit



Another Web Server: GHTTPD



Value at *ptr changes after it was checked
but before it was used! (This is a TOCTTOU attack)

SSH Authentication Code

```
void do_authentication(char *user, ...) {
1:   int authenticated = 0;
     ...
2:   while (!authenticated) {
       /* Get a packet from the client */
3:     type = packet_read();
       /* calls detect_attack() internally
4:     switch (type) {
       ...
5:     case SSH_CMSG_AUTH_PASSWORD:
6:      if (auth_password(user, password))
7:           authenticated =1;
       case ...
       }
8:     if (authenticated) break;
     }
     /* Perform session preparation. */
9:   do_authenticated(pw);
}
```

write 1 here

Loop until one of the authentication methods succeeds

detect_attack() prevents checksum attack on SSH1…

…and also contains an overflow bug which permits the attacker to put any value into any memory location

Break out of authentication loop without authenticating properly

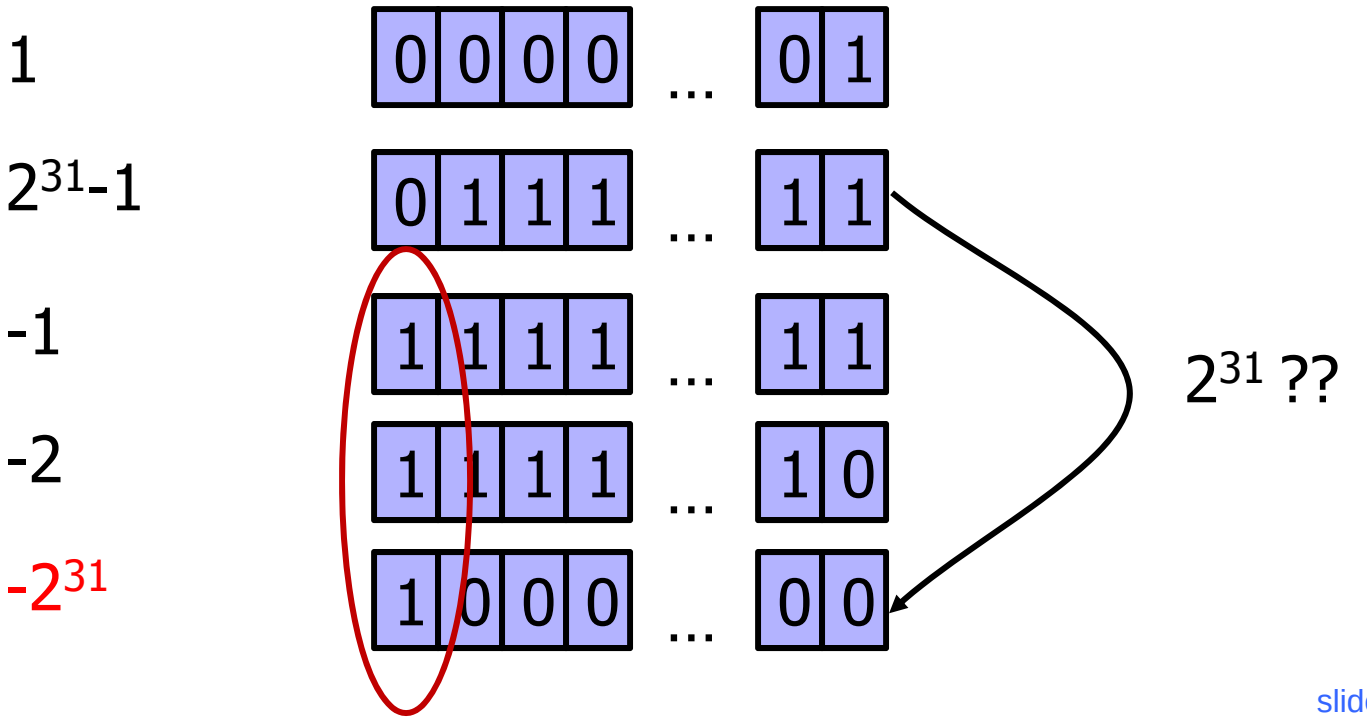
Reducing Lifetime of Critical Data

```
(B2)Modified SSHD do_authentication()
{ int authenticated = 0;
  while (!authenticated) {
L1:type = packet_read(); //vulnerable
   authenticated = 0;
   switch (type) {
    case SSH_CMSG_AUTH_PASSWORD:
     if (auth_password(user, passwd))
          authenticated = 1;
    case ...
   }
   if (authenticated) break;
  }
  do_authenticated(pw);
}
```

Reset flag here, right before doing the checks

Two's Complement

- Binary representation of negative integers
- Represent X (where X<0) as 2^N-|X|
  - N is word size (e.g., 32 bits on x86 architecture)



Integer Overflow

```
static int getpeername1(p, uap, compat) {
// In FreeBSD kernel, retrieves address of peer to which a socket is connected
    …
    struct sockaddr *sa;
     …
    len = MIN(len, sa->sa_len);
    … copyout(sa, (caddr_t)uap->asa, (u_int)len);
    …
}
```

Checks that "len" is not too big

Negative "len" will always pass this check…

… interpreted as a huge unsigned integer here

Copies "len" bytes from kernel memory to user space

… will copy up to 4G of kernel memory

ActionScript Exploit

[Dowd]

- ActionScript 3 is a scripting language for Flash
  - Basically, JavaScript for Flash animations
  - For performance, Flash 9 and higher compiles scripts into bytecode for ActionScript Virtual Machine (AVM2)
- Flash plugins are installed on millions of browsers, thus a perfect target for attack
  - Different Flash binaries are used for Internet Explorer and Firefox, but this turns out not to matter
- Exploit published in April 2008

Processing SWF Scene Records (1)

Code that allocates memory for scene records:

Supplied as part of SWF file from potentially malicious website

```
call   SWF_GetEncodedInteger ; Scene Count
mov  edi, [ebp+arg_0]
mov  [esi+4], eax      ← How much memory is neded to store scenes
mov  ecx, [ebx+8]      ← Total size of the buffer
sub   ecx, [ebx+4]     ← Offset into the buffer
cmp    eax, ecx        ← Is there enough memory in the buffer?
jg      loc_30087BB4   ← (signed comparison)
...
push  eax              ← Tell mem_Calloc how many bytes to allocate
call    mem_Calloc     ← Interprets its argument as unsigned integer
```

What if scene count is negative?

mem_Calloc fails (why?) and returns NULL

Processing SWF Scene Records (2)

- Scene records are copied as follows:
  - Start with pointer P returned by allocator
  - Loop through and copy scenes until count ≤ 0
  - Copy frame count into P + offset, where offset is determined by scene count
    - Frame count also comes from the SWF file
    - It is a "short" (16-bit) value, but written as a 32-bit DWORD
- Attacker gains the ability to write one value into any location in memory (why?)
  - … subject to some restrictions (see paper)
  - But this is not enough to hijack control directly (why?)

ActionScript Virtual Machine (AVM2)

- Register-based VM
  - Bytecode instructions write and read from "registers"
- "Registers", operand stack, scope stack allocated on the same runtime stack as used by Flash itself
  - "Registers" are mapped to locations on the stack and accessed by index (converted into memory offset)
  - This is potentially dangerous (why?)
- Malicious Flash script could hijack browser's host
  - Malicious bytecode can write into any location on the stack by supplying a fake register index
  - This would be enough to take control (how?)

AVM2 Verifier

- ActionScript code is verified before execution
- All bytecodes must be valid
  - Throw an exception if encountering an invalid bytecode
- All register accesses correspond to valid locations on the stack to which registers are mapped
- For every instruction, calculate the number of operands, ensure that operands of correct type will be on the stack when it is executed
- All values are stored with correct type information
  - Encoded in bottom 3 bits

Relevant Verifier Code

```
…
if(AS3_argmask[opCode] == 0xFF) {
   … throw exception …
}
…
opcode_getArgs(…)
…

void opcode_getArgs(…) {
   DWORD mask=AS3_argmask[opCode];
   …
   if(mask <=0) { … return … }
   … *arg_dword1 = SWF_GetEncodedInteger(&ptr);
   if(mask>1) *arg_dword2 = SWF_GetEncodedInteger(&ptr);
}
```

Invalid bytecode

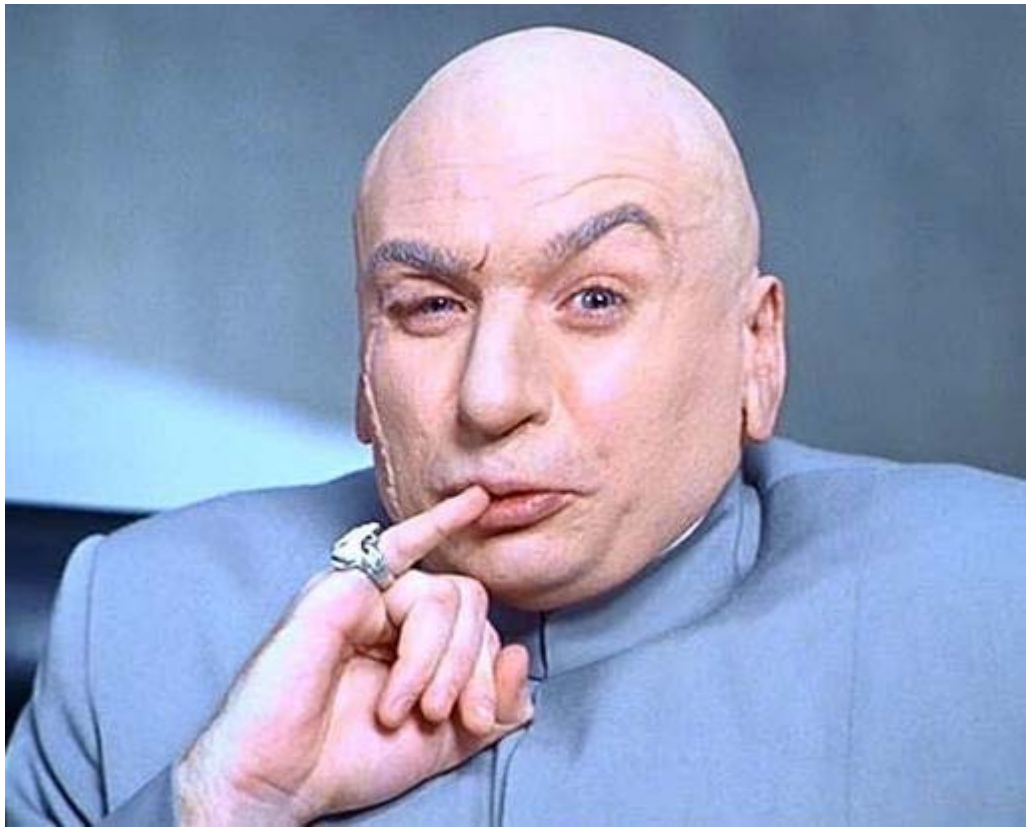
Number of operands for each opcode is defined in AS3_argmask array

Determine operands

Executing Invalid Opcodes

- If interpreter encounters an invalid opcode, it silently skips it and continues executing
  - Doesn't really matter because this can't happen
    - Famous last words…
  - AS3 code is executed only after it has been verified, and verifier throws an exception on invalid bytecode
- But if we could somehow trick the verifier…
  - Bytes after the opcode are treated as data (operands) by the verifier, but as executable code by interpreter
  - This is an example of a TOCTTOU (time-of-check-to-time-of-use) vulnerability

Breaking AVM2 Verifier



Breaking AVM2 Verifier

- Pick an invalid opcode
- Use the ability to write into arbitrary memory to change the AS3_argmask of that opcode from 0xFF to something else
- AVM2 verifier will treat it as normal opcode and skip subsequent bytes as operands
  - How many? This is also determined by AS3_argmask!
- AVM2 interpreter, however, will skip the invalid opcode and execute those bytes
- You can now execute unverified ActionScript code

Further Complications

- Can execute only a few unverified bytecodes at a time (why?)
  - Use multiple "marker" opcodes with overwritten masks
- Cannot directly overwrite saved EIP on the evaluation stack with the address of shellcode because 3 bits are clobbered by type information
  - Stack contains a pointer to current bytecode (codePtr)
  - Move it from one "register" to another, overwrite EIP
  - Bytecode stream pointed to by codePtr should contain a jump to the actual shellcode
- Read the paper

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it
    - Overwrite saved EIP, function callback pointer, etc.
- Idea: prevent execution of untrusted code
  - Make stack and other data areas non-executable
    - Note: messes up useful functionality (e.g., ActionScript)
  - Digitally sign all code
  - Ensure that all control transfers are into a trusted, approved code image

W⊕X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's DEP (Data Execution Prevention)
  - This blocks all code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows (since XP SP2), Linux (via PaX patches), OpenBSD, OS X (since 10.5)

What Does W⊕X Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap, but that's not important right now
- As long as "saved EIP" points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange memory to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code
  - … especially if system() is not available

return-to-libc on Steroids

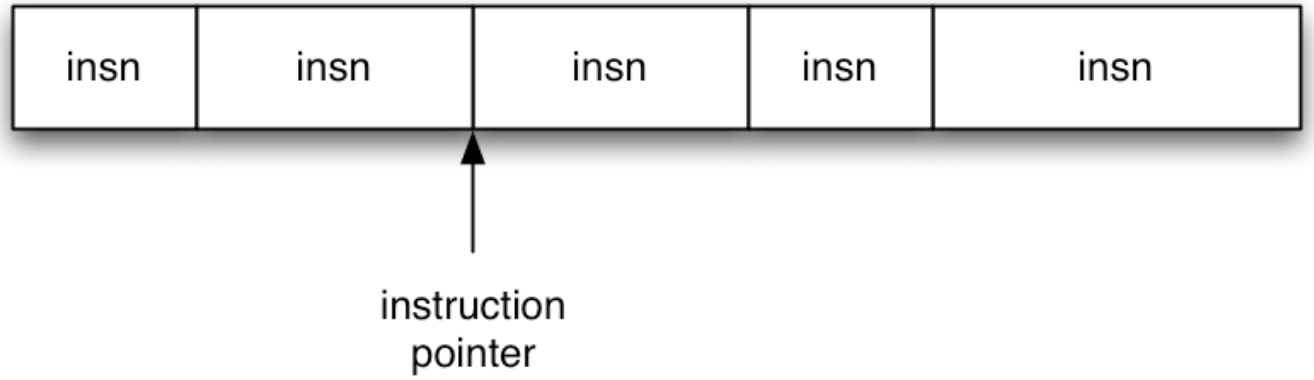
- Overwritten saved EIP need not point to the beginning of a library routine
- Any existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (ESP)
    - Guess what? Its value is under attacker's control! (why?)
  - Use it as the new value for EIP
    - Now control is transferred to an address of attacker's choice!
  - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

[Shacham et al]

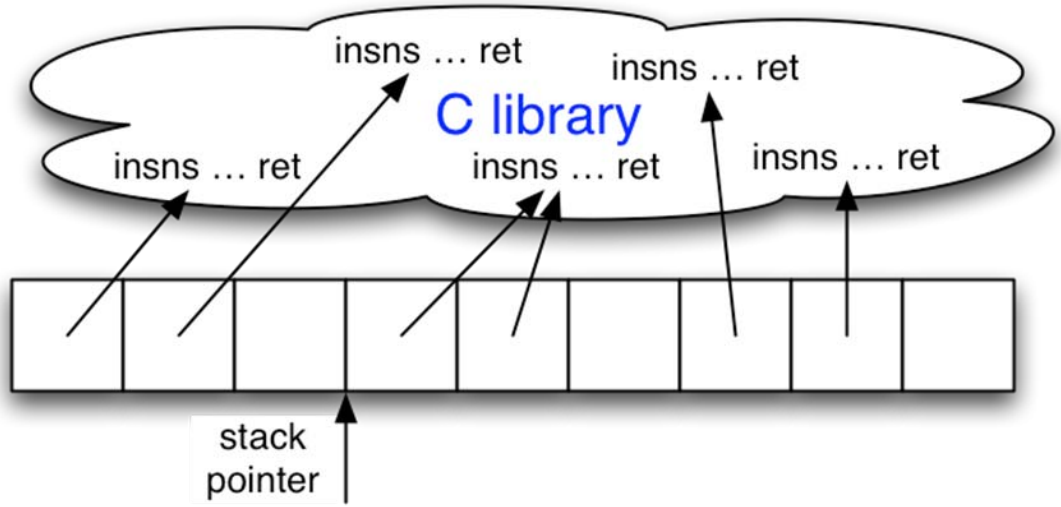
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all!

Ordinary Programming



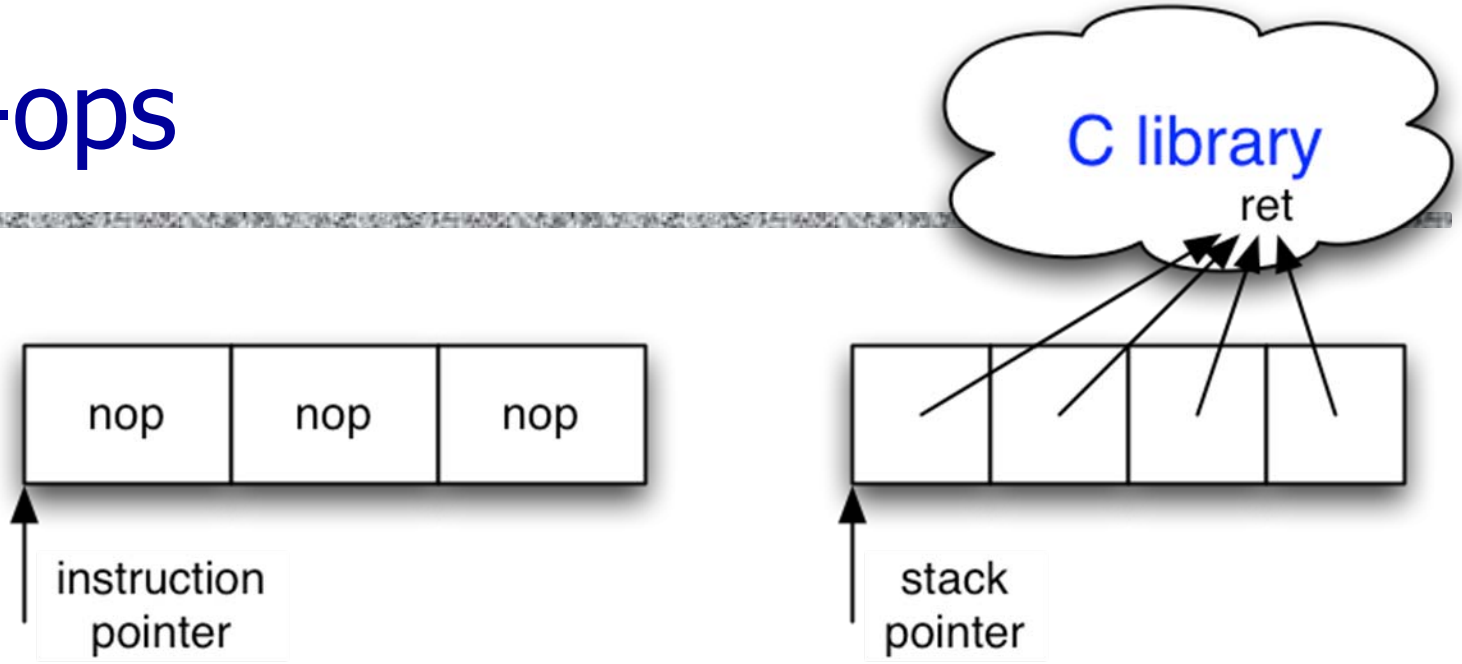
- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Return-Oriented Programming



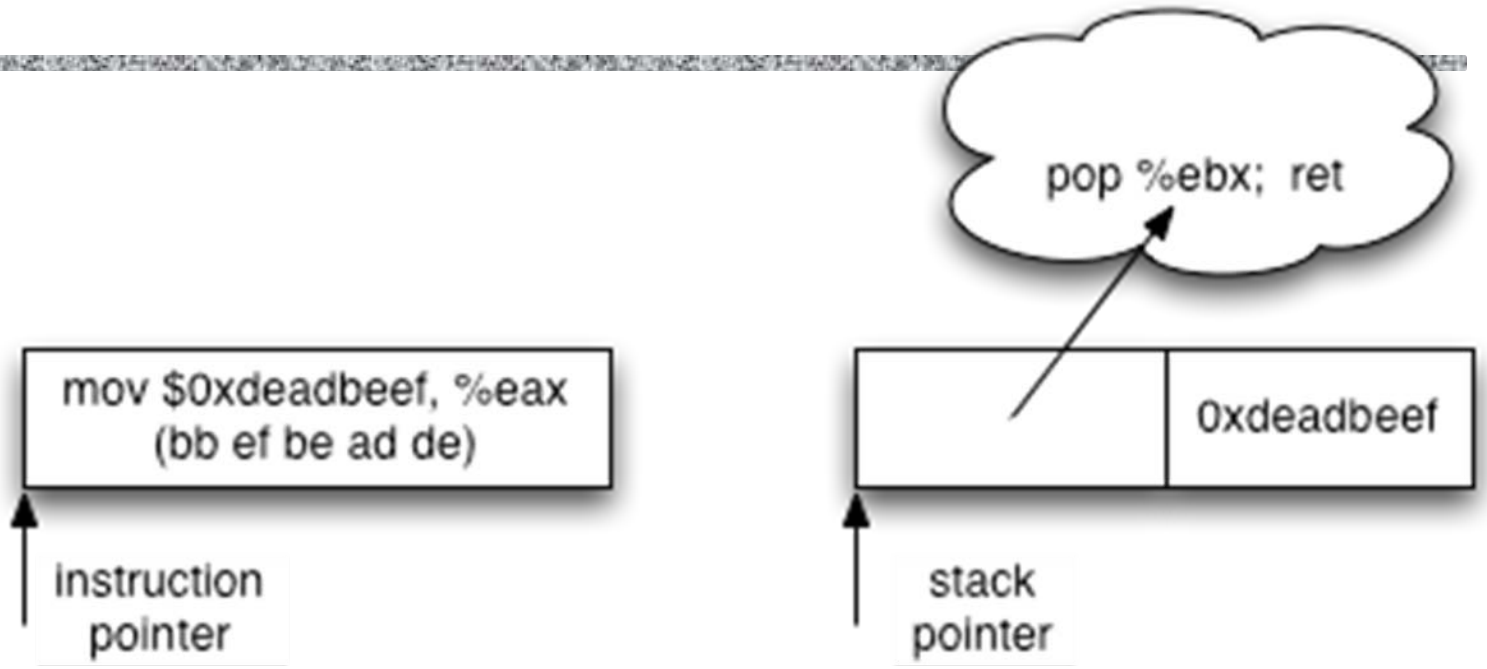
- Stack pointer (ESP) determines which instruction sequence to fetch and execute
- Processor doesn't automatically increment ESP
  - But the RET at end of each instruction sequence does

No-ops



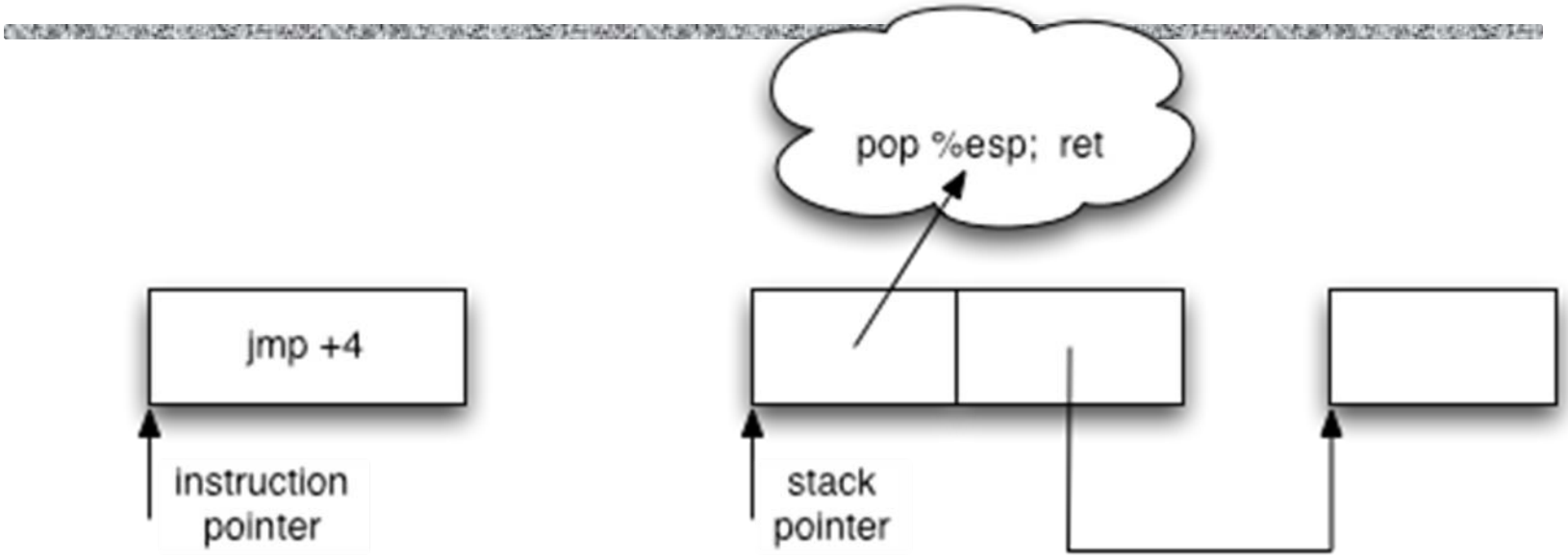
- No-op instruction does nothing but advance EIP
- Return-oriented equivalent
  - Point to return instruction
  - Advances ESP
- Useful in a NOP sled (what's that?)

Immediate Constants



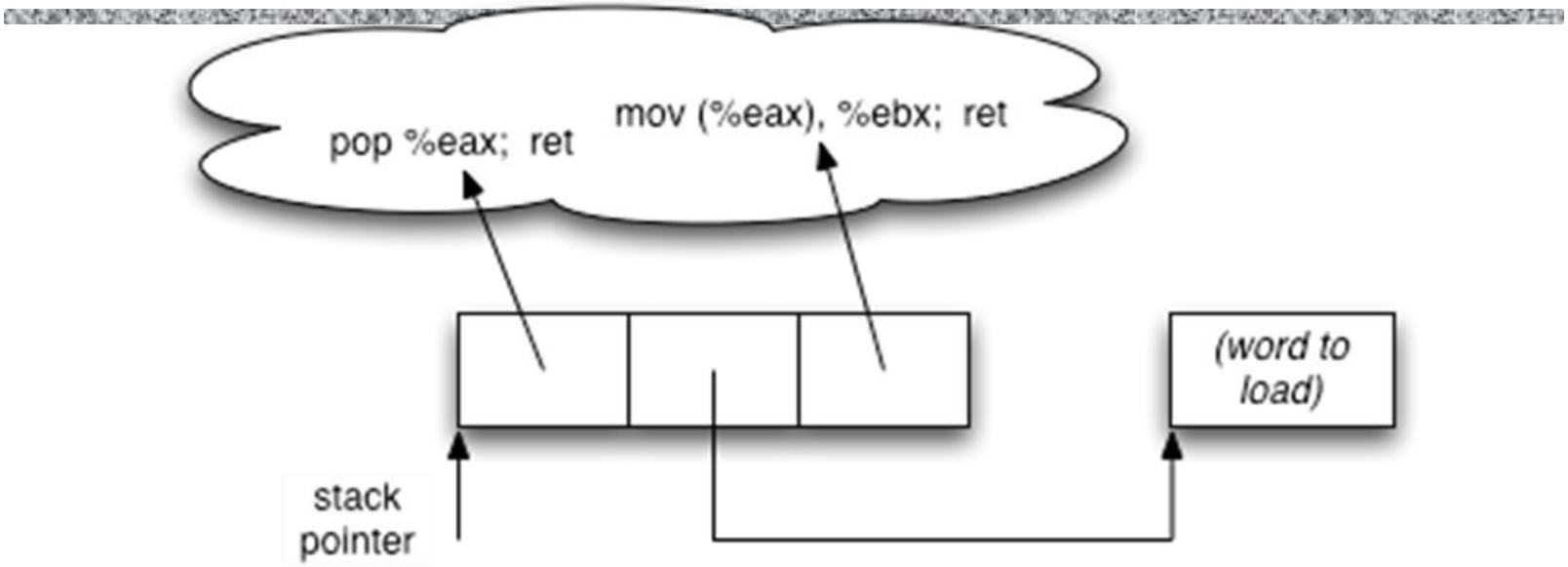
- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Control Flow



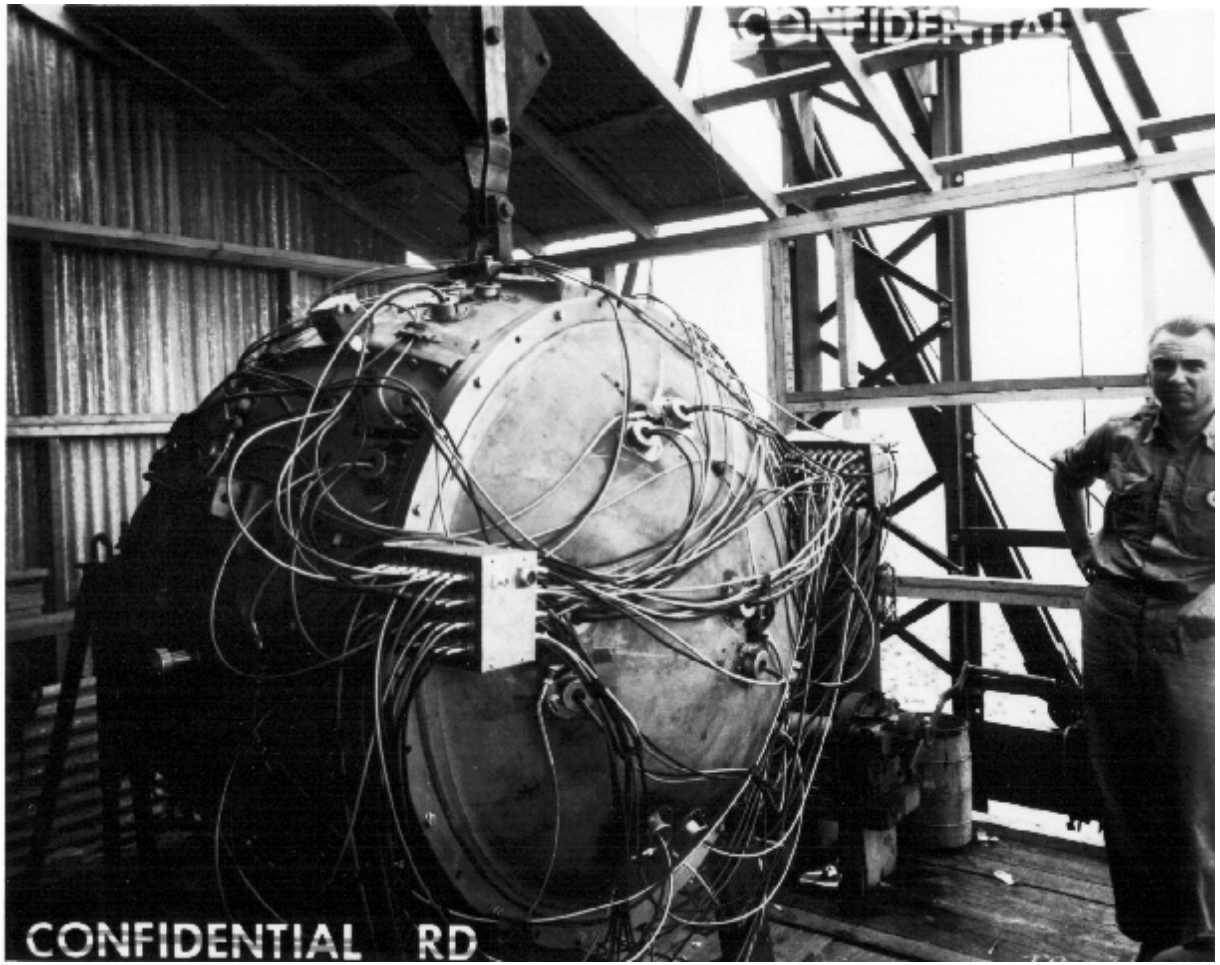
- Ordinary programming
  - (Conditionally) set EIP to new value
- Return-oriented equivalent
  - (Conditionally) set ESP to new value

Gadgets: Multi-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

“The Gadget”: July 1945



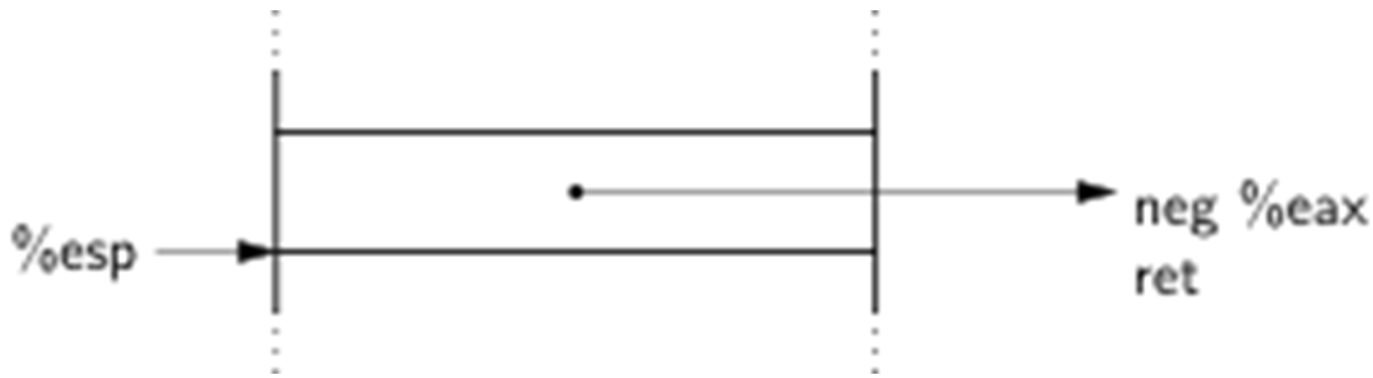
Gadget Design

- Testbed: libc-2.3.5.so, Fedora Core 4
- Gadgets built from found code sequences:
  - Load-store, arithmetic & logic, control flow, syscalls
- Found 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

Conditional Jumps

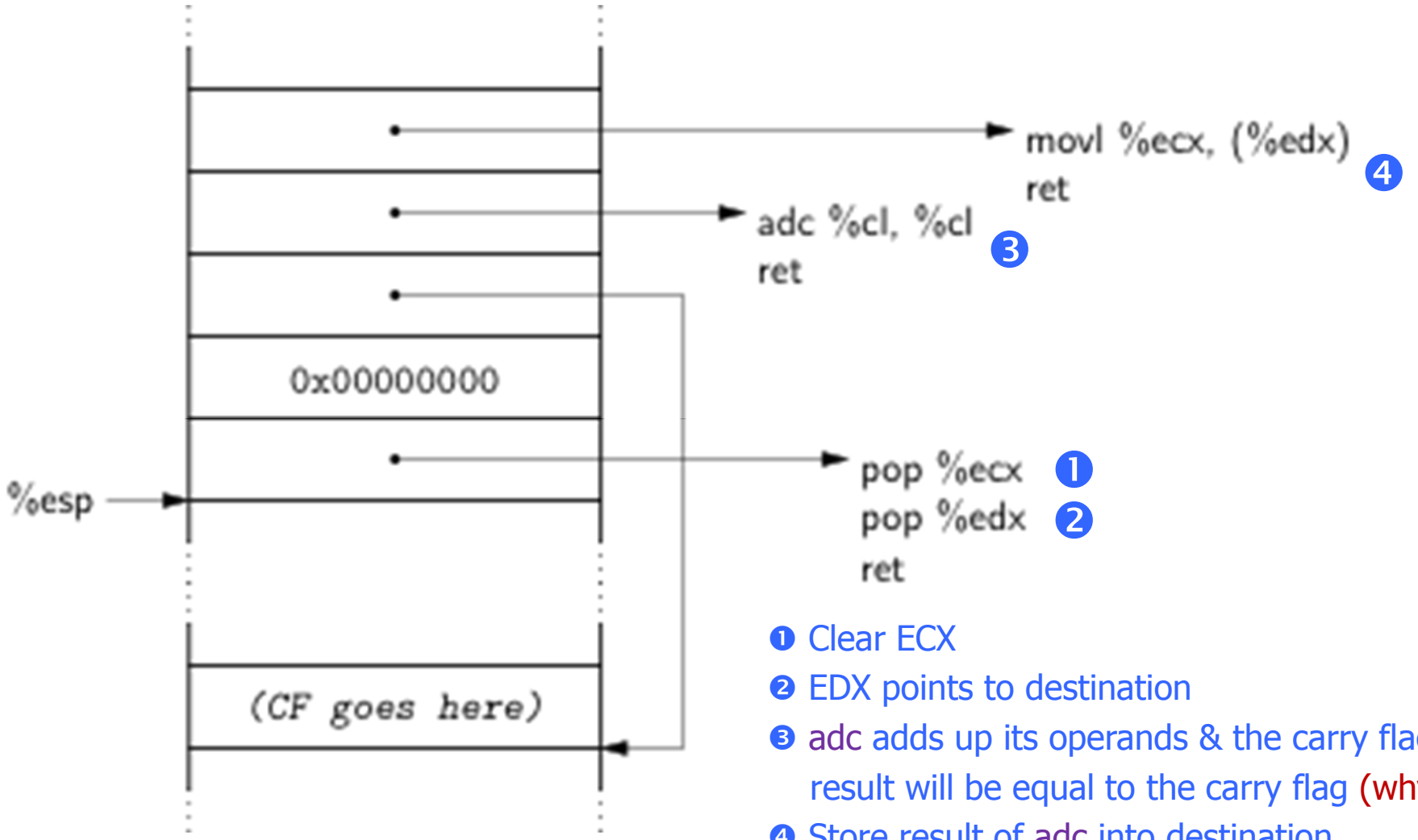
- cmp compares operands and sets a number of flags in the EFLAGS register
  - Luckily, many other ops set EFLAGS as a side effect
- jcc jumps when flags satisfy certain conditions
  - But this causes a change in EIP… not useful (why?)
- Need conditional change in stack pointer (ESP)
- Strategy:
  - Move flags to general-purpose register
  - Compute either delta (if flag is 1) or 0 (if flag is 0)
  - Perturb ESP by the computed delta

Phase 1: Perform Comparison



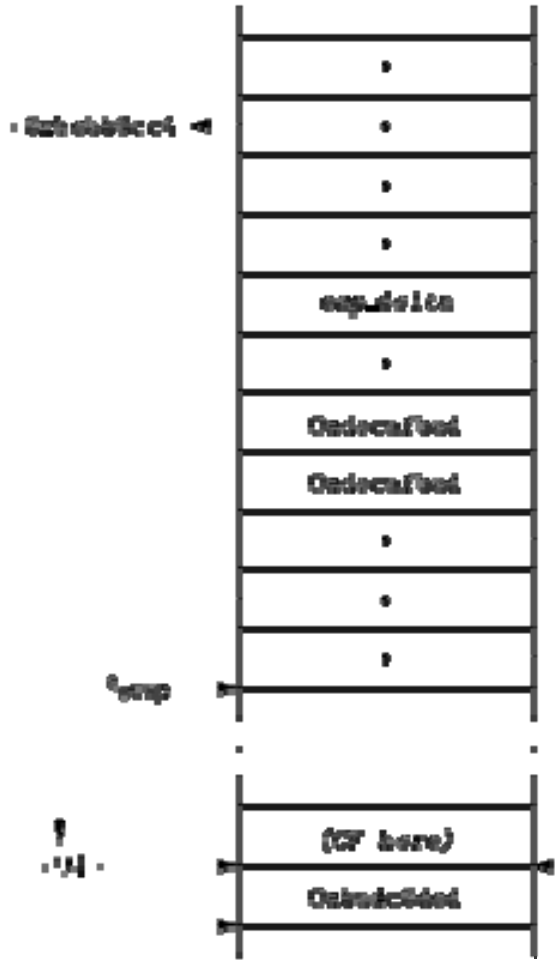
- neg calculates two's complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- sub is similar, use to test if one number is greater than another

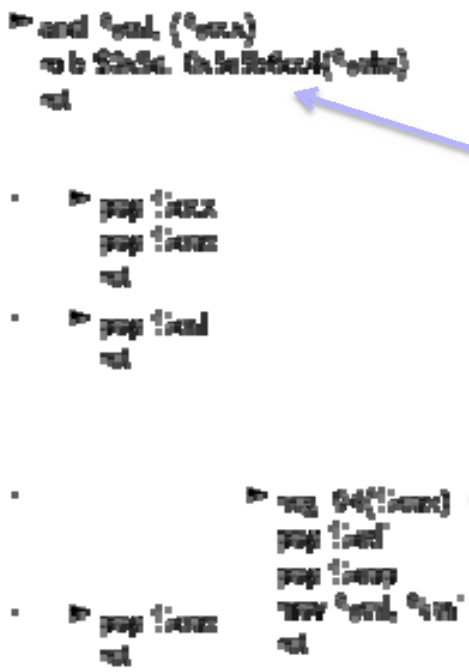
Phase 2: Store 1-or-0 to Memory



1. Clear ECX
2. EDX points to destination
3. adc adds up its operands & the carry flag; result will be equal to the carry flag (why?)
4. Store result of adc into destination

Phase 3: Compute Delta-or-Zero





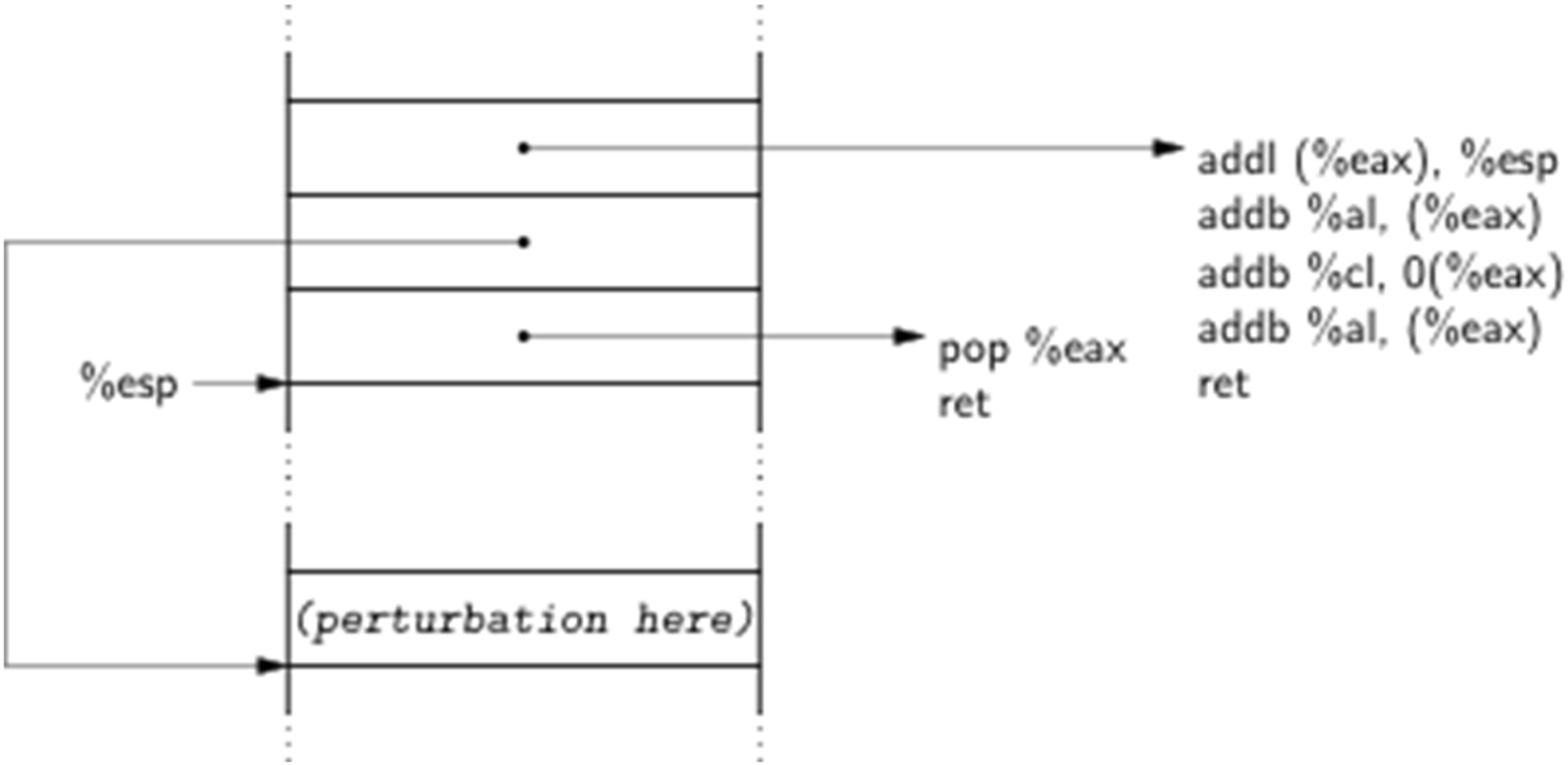
Bitwise AND with delta (in ESI)

Two's-complement negation:

0 becomes 0…0;

1 becomes 1…1

Phase 4: Perturb ESP by Delta

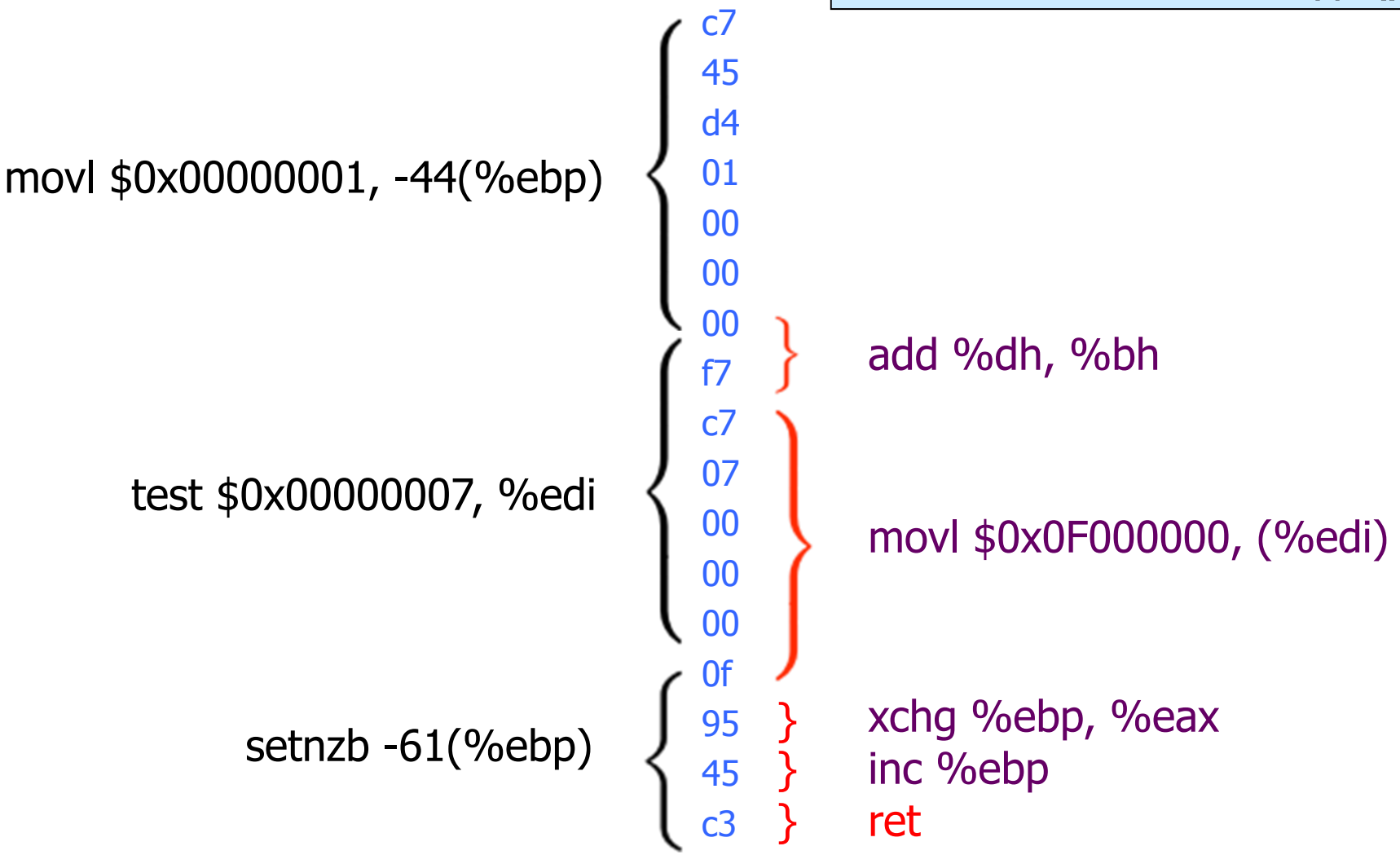


Finding Instruction Sequences

- Any instruction sequence ending in RET is useful
- Algorithmic problem: recover all sequences of valid instructions from libc that end in a RET
- At each RET (C3 byte), look back:
  - Are preceding i bytes a valid instruction?
  - Recur from found instructions
- Collect instruction sequences in a trie

Unintended Instructions

Actual code from ecb_crypt()



x86 Architecture Helps

- Register-memory machine
  - 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
  - Instruction types not issued by compiler may be available
- Unstructured call/ret ABI
  - Any sequence ending in a return is useful

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

ROP on SPARC

- Testbed: Solaris 10 libc (1.3 MB)
- Use instruction sequences that are suffixes of real functions
- Dataflow within a gadget
  - Structured dataflow to dovetail with calling convention
- Dataflow between gadgets
  - Each gadget is memory-memory
- Turing-complete computation!
- Read paper for details