CS 6431

Exploiting Memory

Vitaly Shmatikov

Famous Internet Worms

- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected (10% of existing Internet)
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- Sasser (2004): overflow in Windows LSASS
  - Around 500,000 machines infected

Responsible for user authentication in Windows

… And The Band Marches On

- Conficker (2008-09): overflow in Windows RPC
  - Around 10 million machines infected (estimates vary)
- Stuxnet (2009-10): several zero-day overflows + same Windows RPC overflow as Conficker
  - Windows print spooler service
  - Windows LNK shortcut display
  - Windows task scheduler
- Flame (2010-12): same print spooler and LNK overflows as Stuxnet
  - Targeted cyberespionage virus

Memory Exploits

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
  - Simplest exploit: supply executable code as “data”, trick victim’s machine into executing it
    - Code will self-propagate or give attacker control over machine
- Attack can exploit <u>any</u> memory operation and need not involve code injection or data execution
  - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables …

Stack Buffers

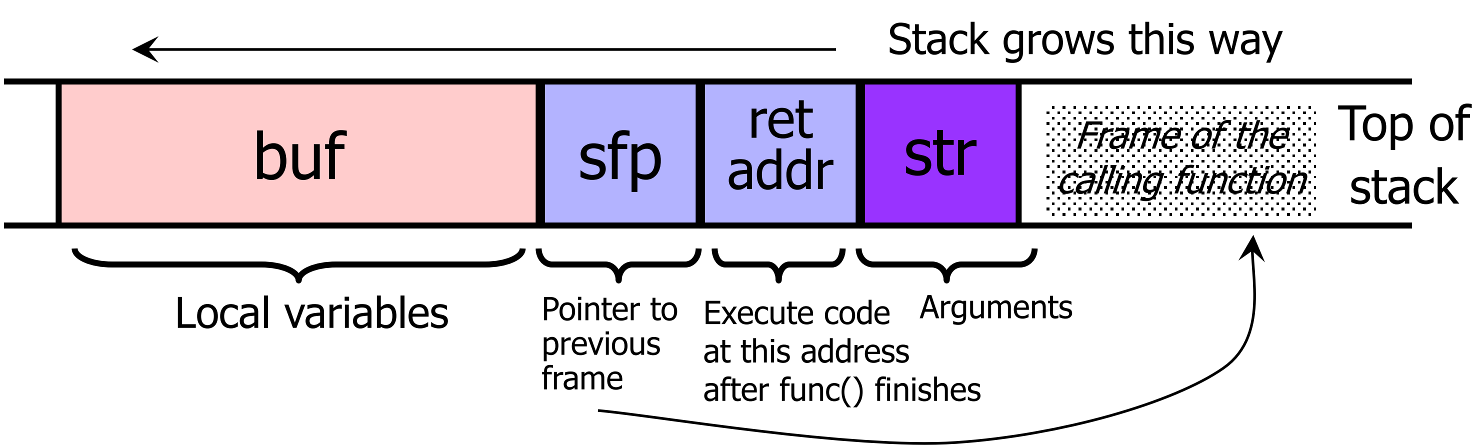
- Suppose Web server contains this function

```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack



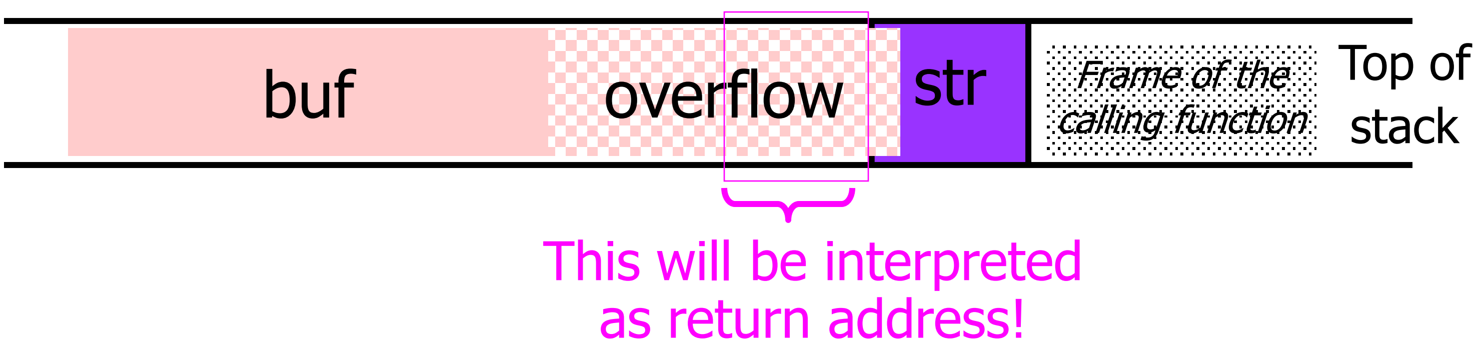
What If Buffer Is Overstuffed?

- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

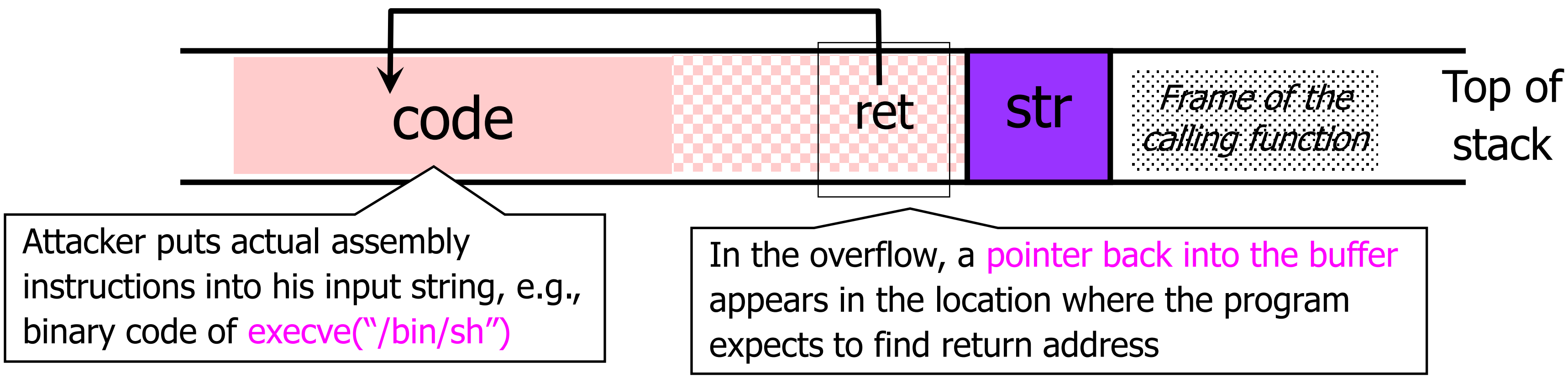
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

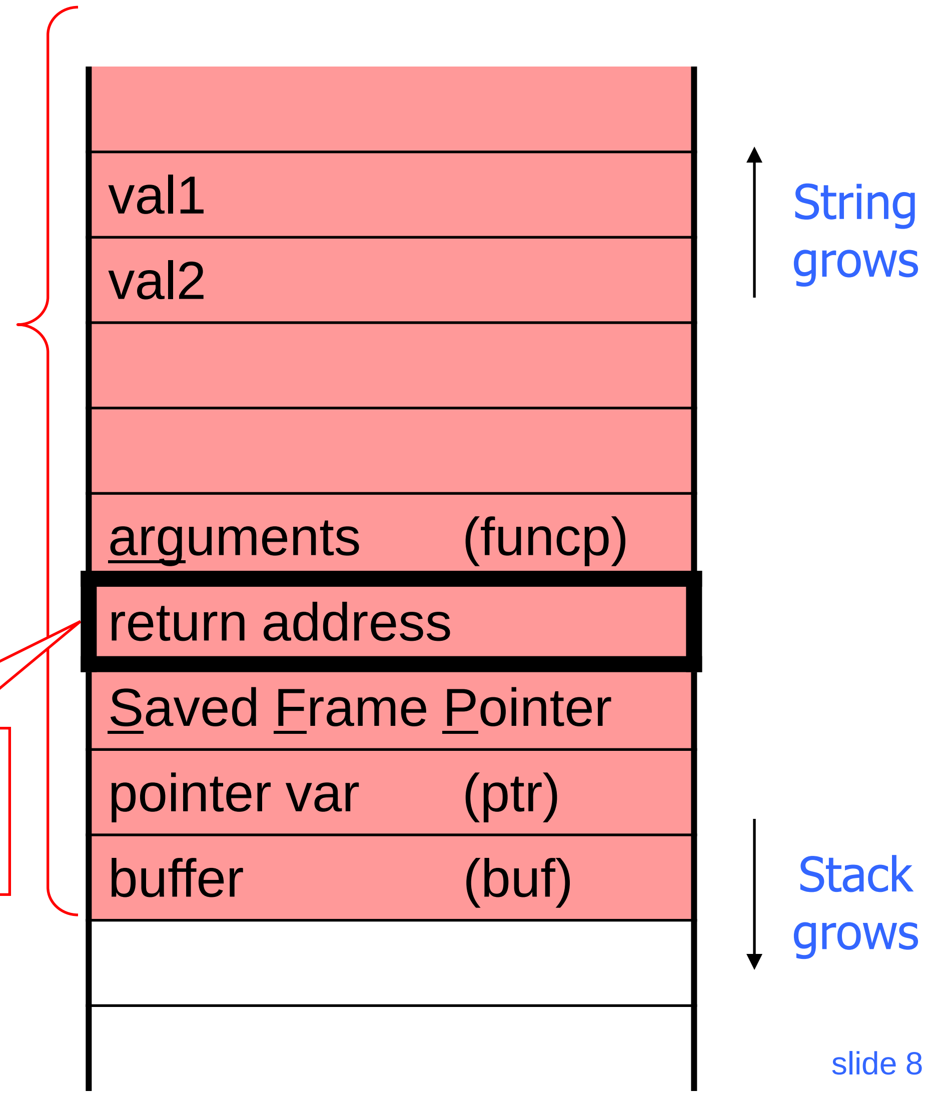
Stack Corruption: General View

```
int bar (int val1) {
   int  val2;
   foo (a_function_pointer);
}
```

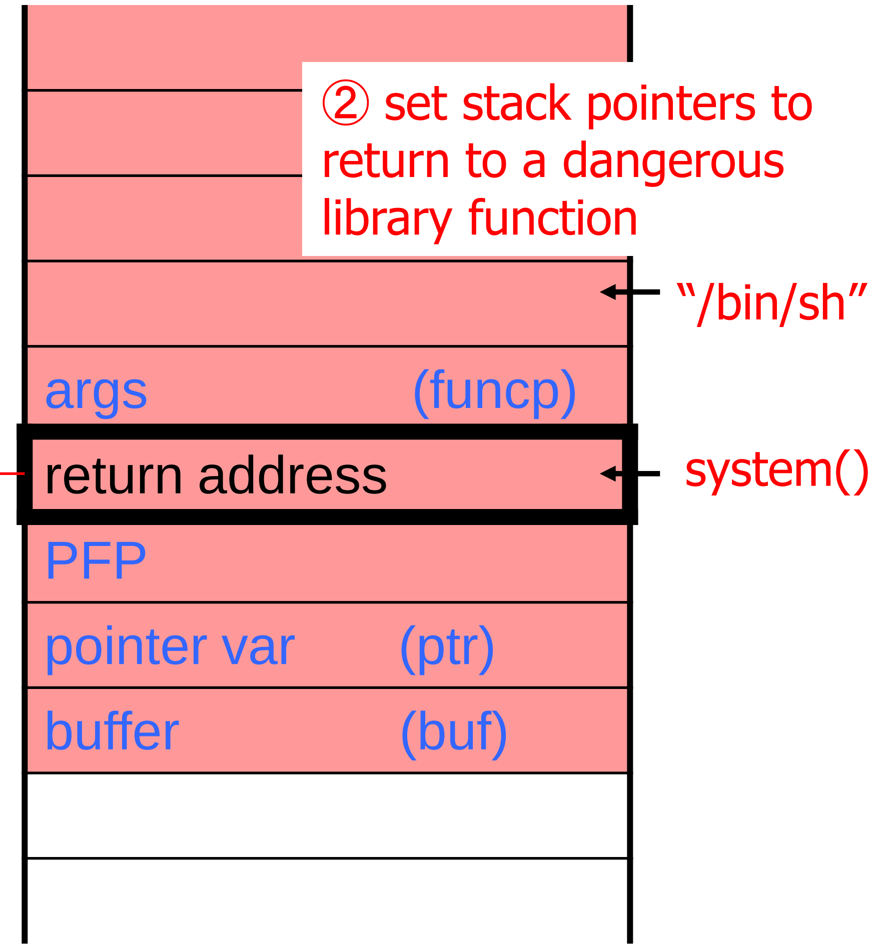
Attacker-controlled memory

```
int foo (void (*funcp)()) {
   char* ptr = point_to_an_array;
   char buf[128];
   gets (buf);
   strncpy(ptr, buf, 8);
   (*funcp)();
}
```

Most popular target



Attack #1: Return Address



① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

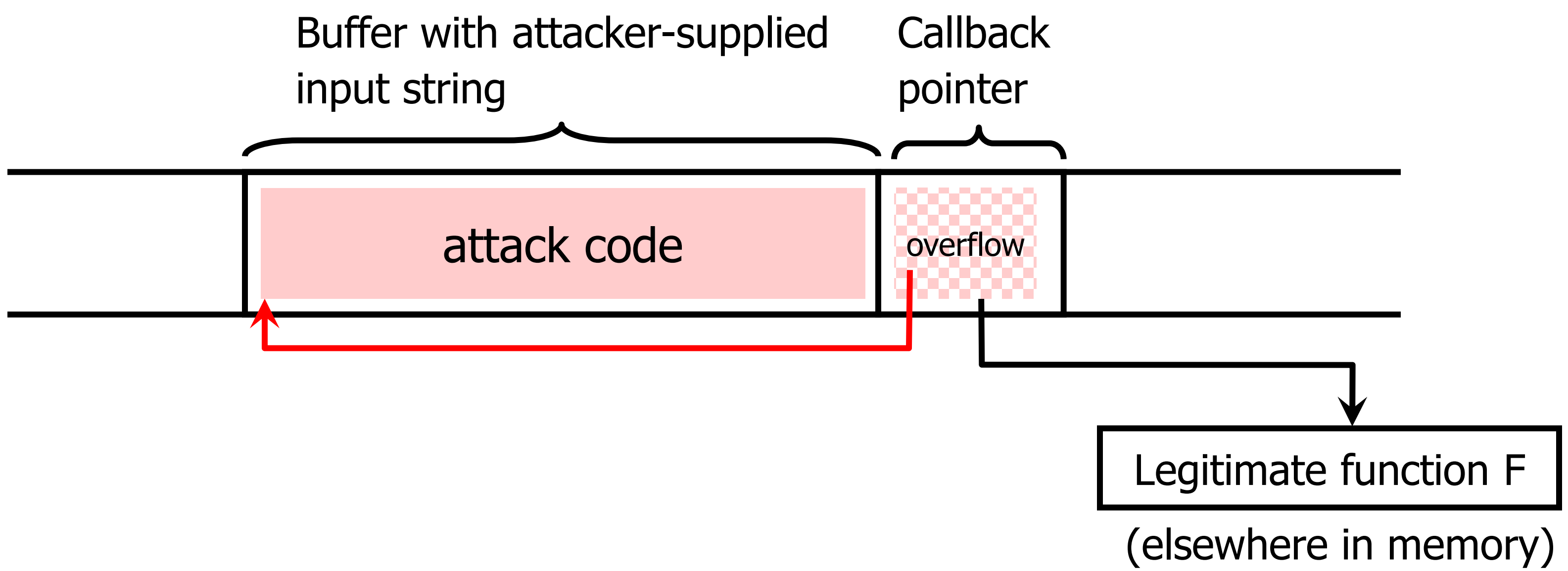
② … or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

Cause: No Range Checking

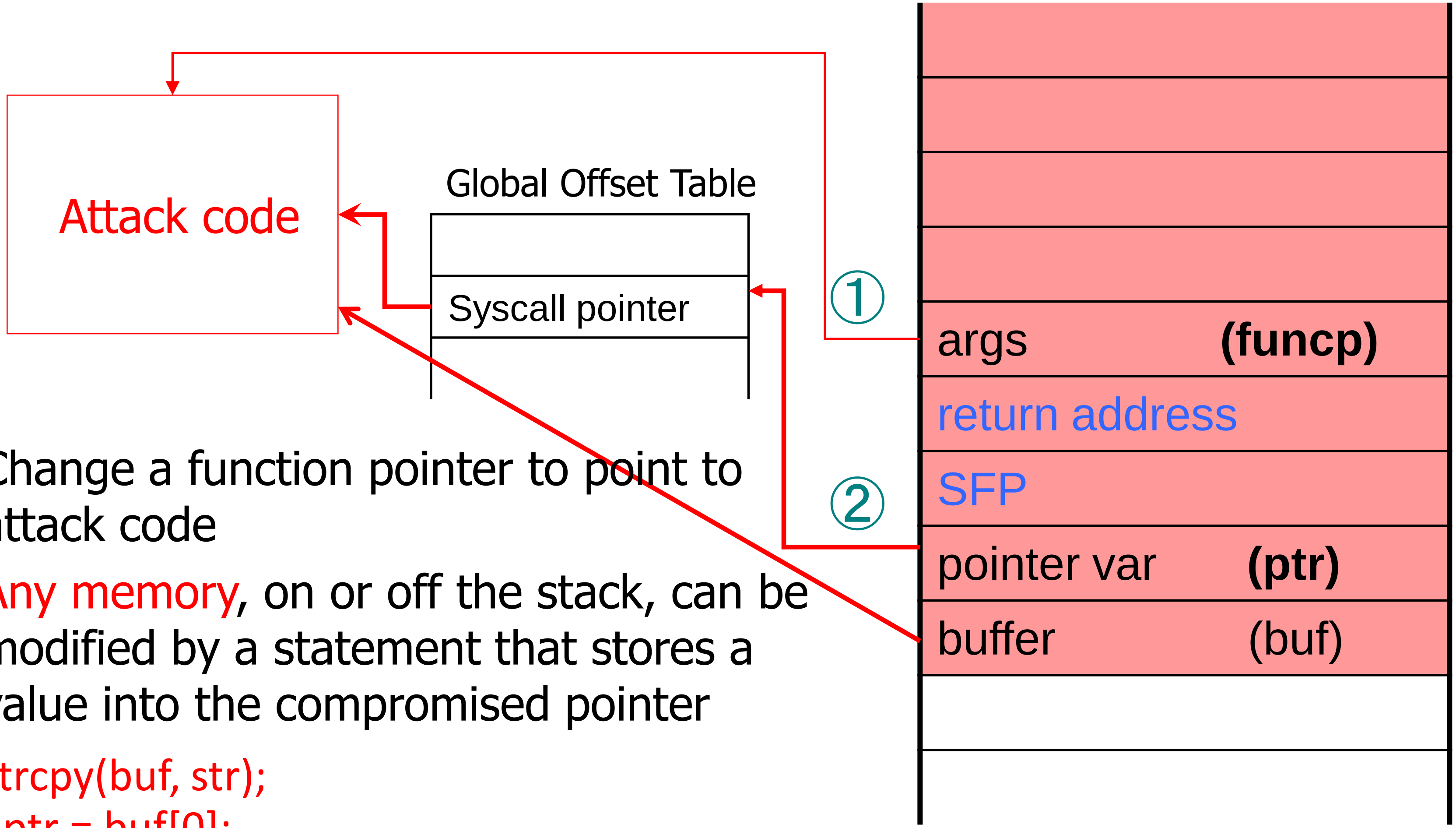
- strcpy does <u>not</u> check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(…)



Attack #2: Pointer Variables



① Change a function pointer to point to attack code

② Any memory, on or off the stack, can be modified by a statement that stores a value into the compromised pointer

```
strcpy(buf, str);
*ptr = buf[0];
```

Off-By-One Overflow

- Home-brewed range-checking string copy

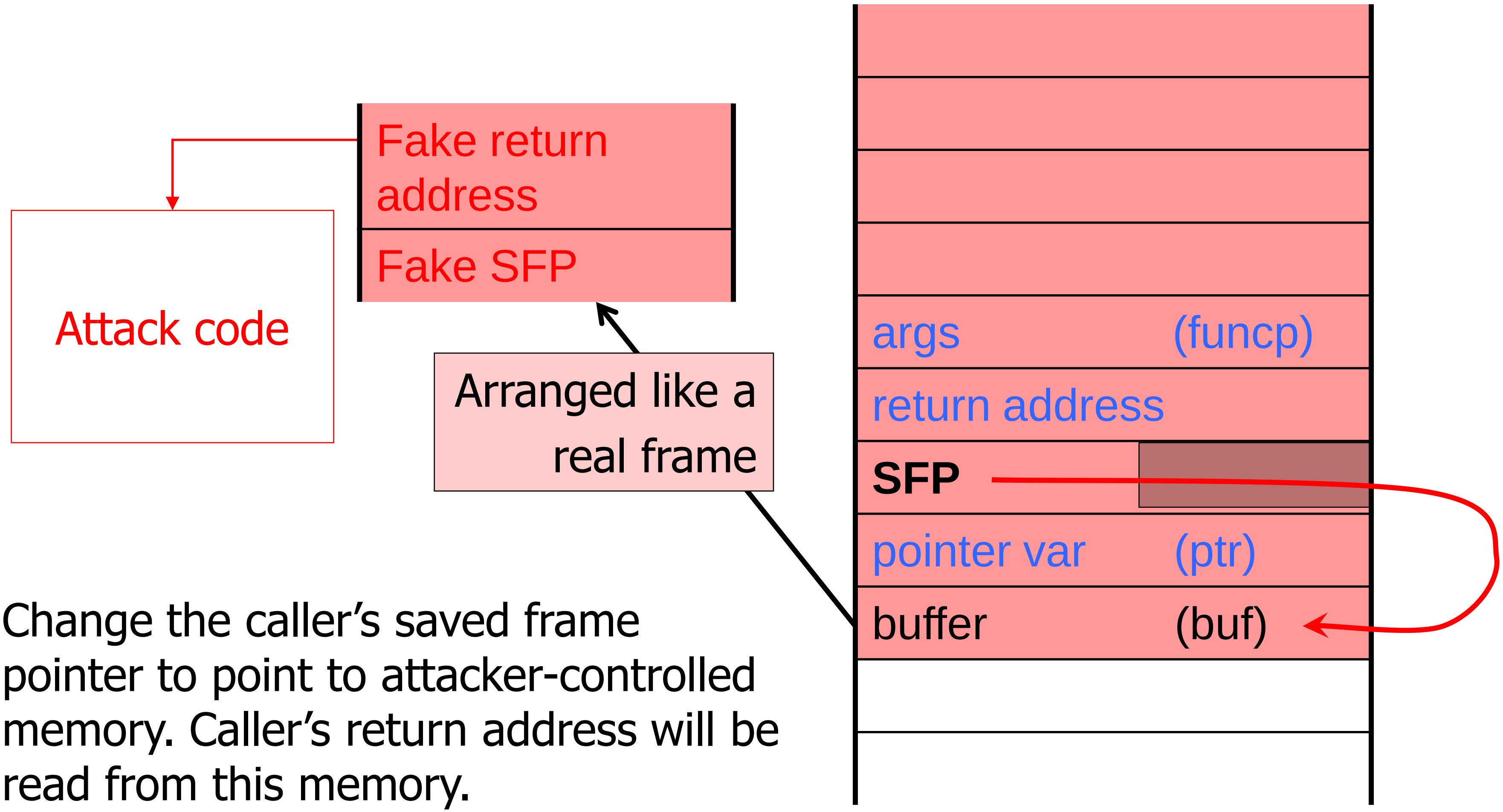
```
void notSoSafeCopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
          buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
          notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into the buffer. Oops!

- 1-byte overflow: can't change RET, but can change saved pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - Caller's RET will be read from buffer!

Attack #3: Frame Pointer



Change the caller's saved frame pointer to point to attacker-controlled memory. Caller's return address will be read from this memory.

Buffer Overflow: Causes and Cures

- "Classic" 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
  - 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 most (not all) code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - OS can make a memory page non-executable
- Widely deployed
  - Windows (since XP SP2), Linux (via PaX patches), OpenBSD, OS X (since 10.5)

Issues with W⊕X / DEP

- Some applications require executable stack
  - Example: JavaScript, Flash, Lisp, other interpreters
- JVM makes all its memory RWX – readable, writable, executable (why?)
  - Can spray attack code over memory containing Java objects (how?), pass control to them
- Some applications don't use DEP
  - For example, some Web browsers
- Attack can start by "returning" into a memory mapping routine and make the page containing attack code writeable

What Does W⊕X <u>Not</u> 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 the 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 the 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

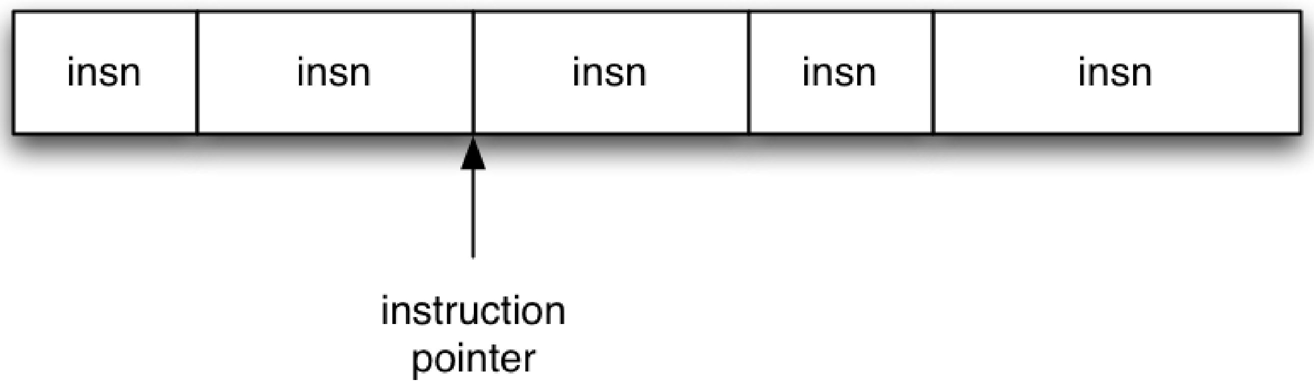
Chaining RETs for Fun and Profit

[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!

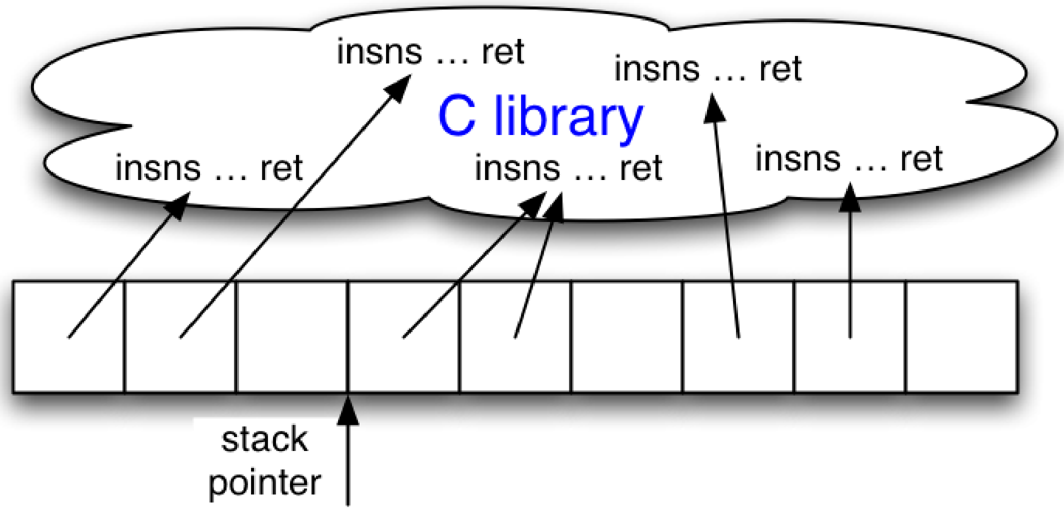
Image by Dino Dai Zovi

Ordinary Programming



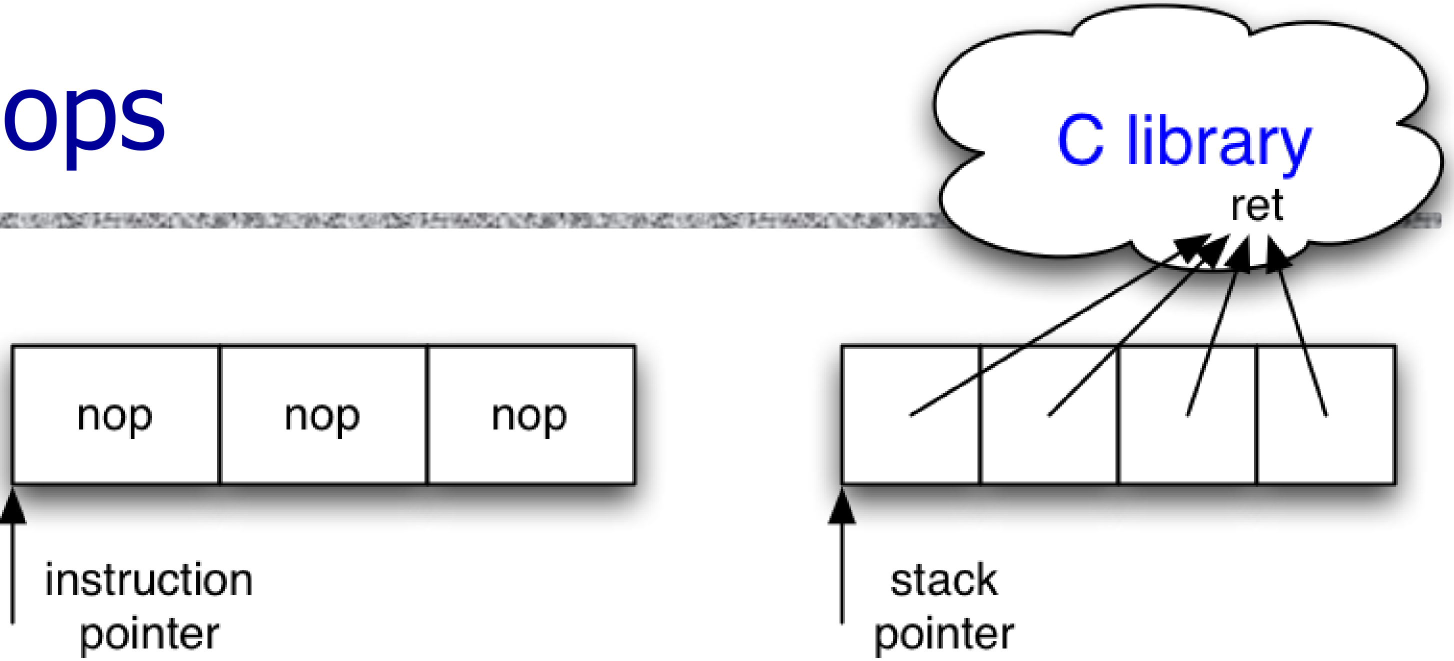
- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Return-Oriented Programming



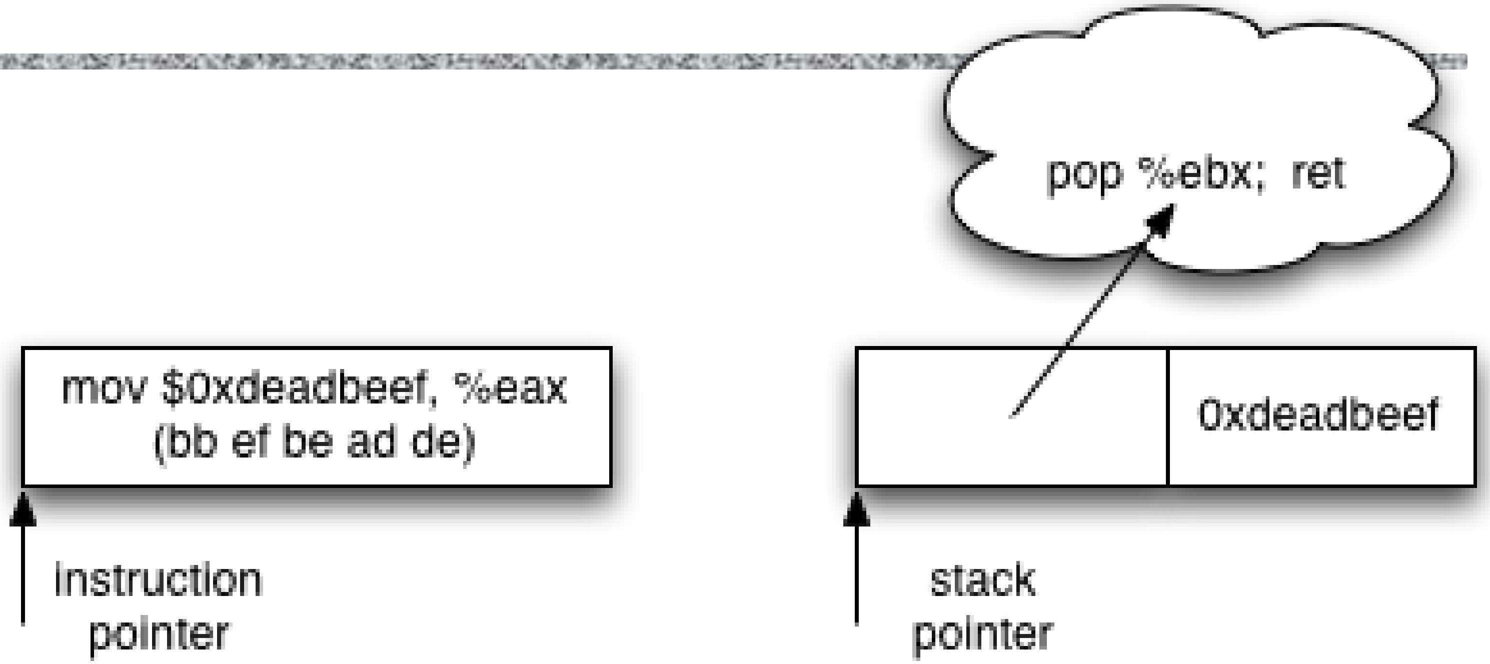
- Stack pointer (ESP) determines which instruction sequence to fetch and execute
- Processor doesn't automatically increment ESP
  - But the RET at end of each instruction sequence does

No-ops



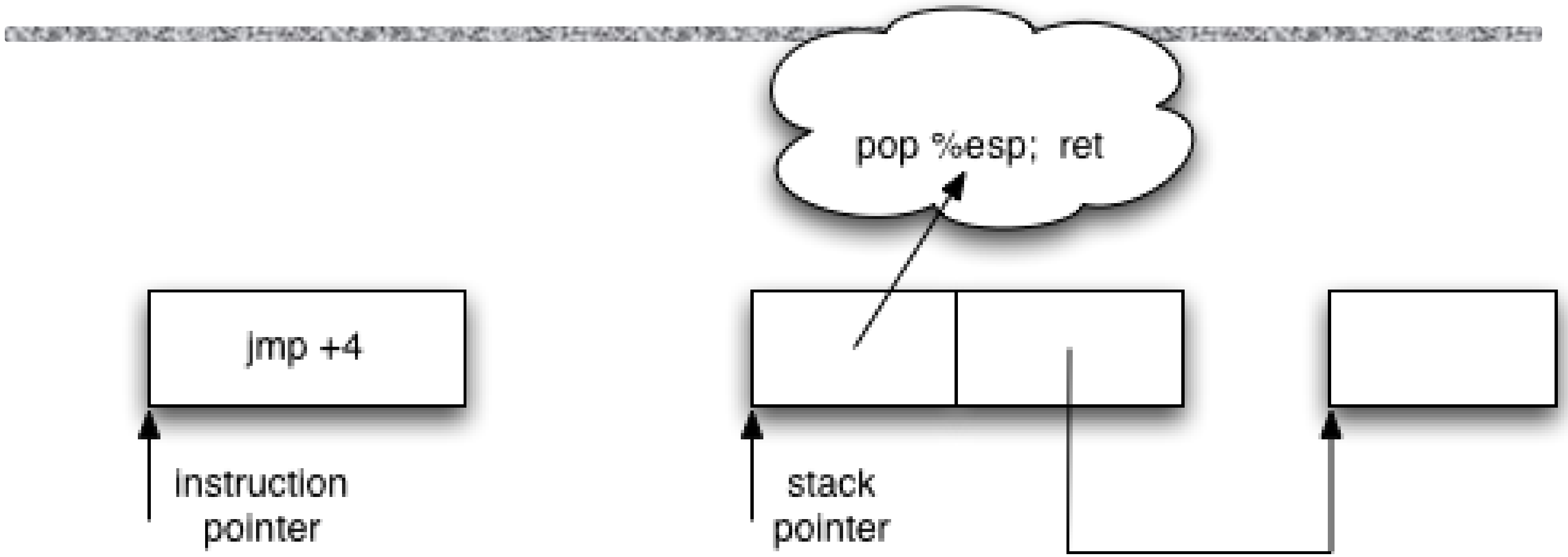
- No-op instruction does nothing but advance EIP
- Return-oriented equivalent
  - Point to return instruction
  - Advances ESP
- Useful in a NOP sled (what's that?)

Immediate Constants



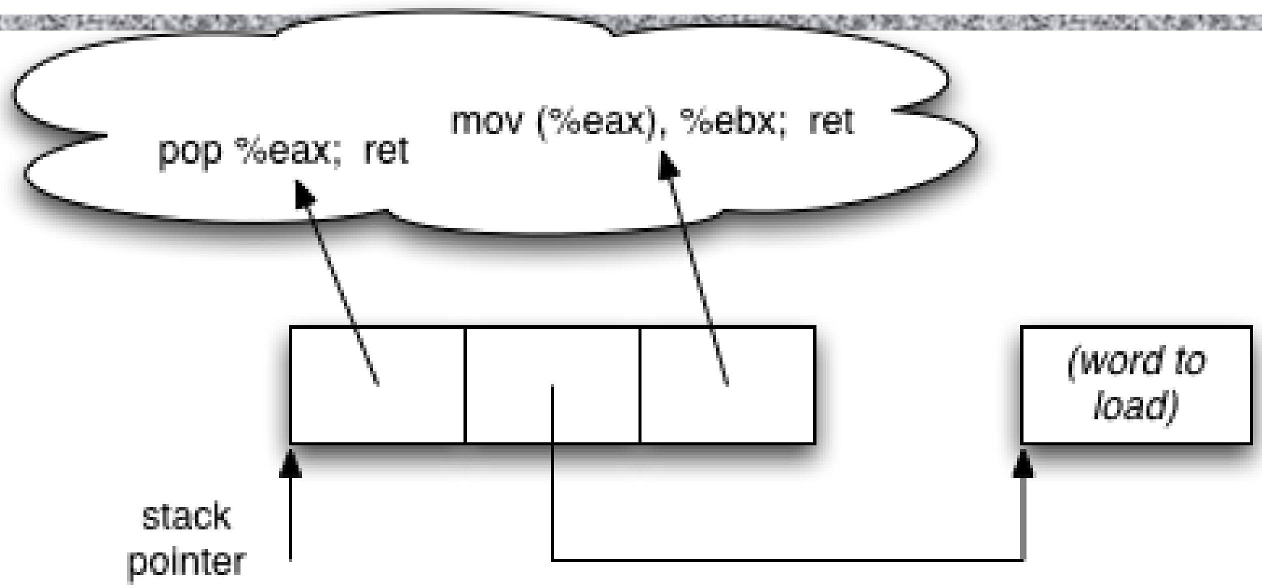
- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Control Flow



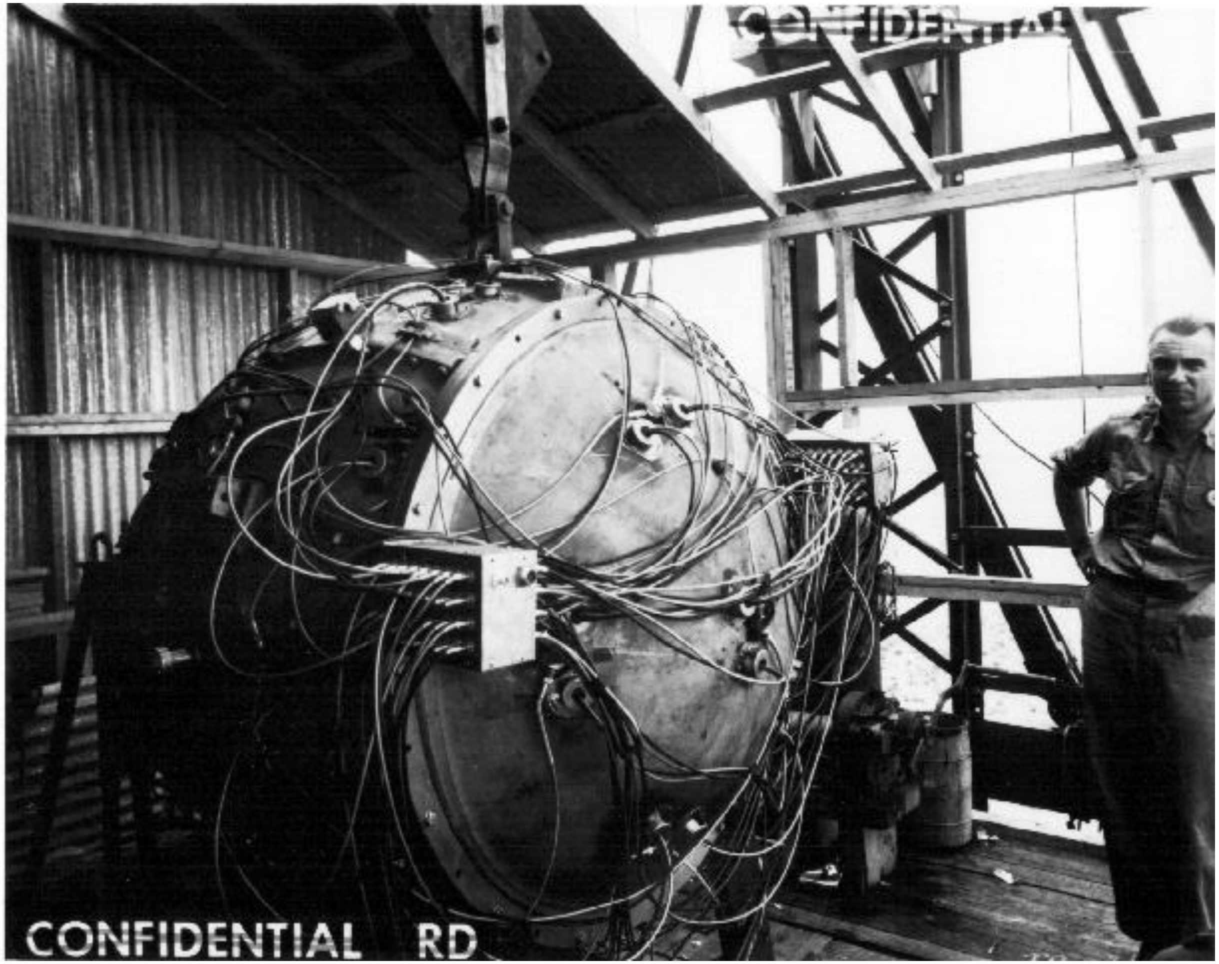
- Ordinary programming
  - (Conditionally) set EIP to new value
- Return-oriented equivalent
  - (Conditionally) set ESP to new value

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



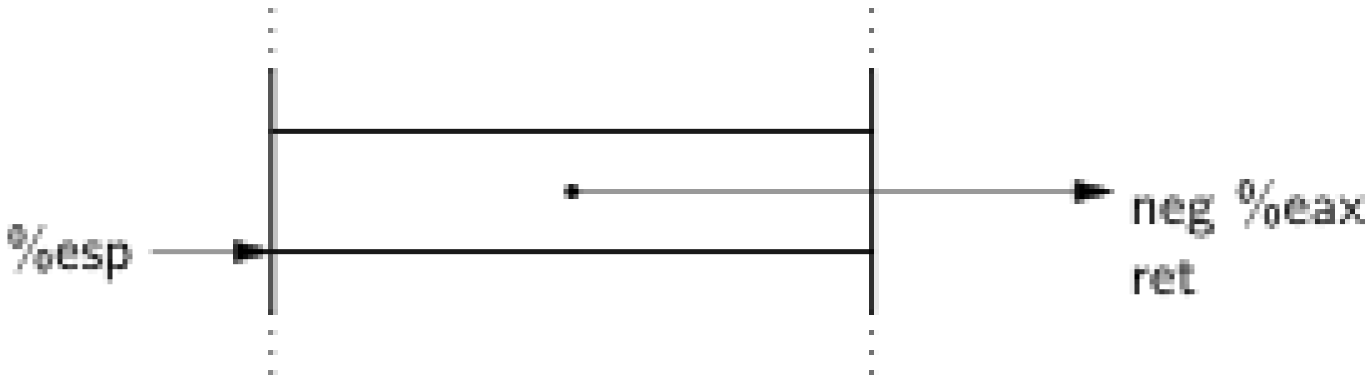
Gadget Design

- 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

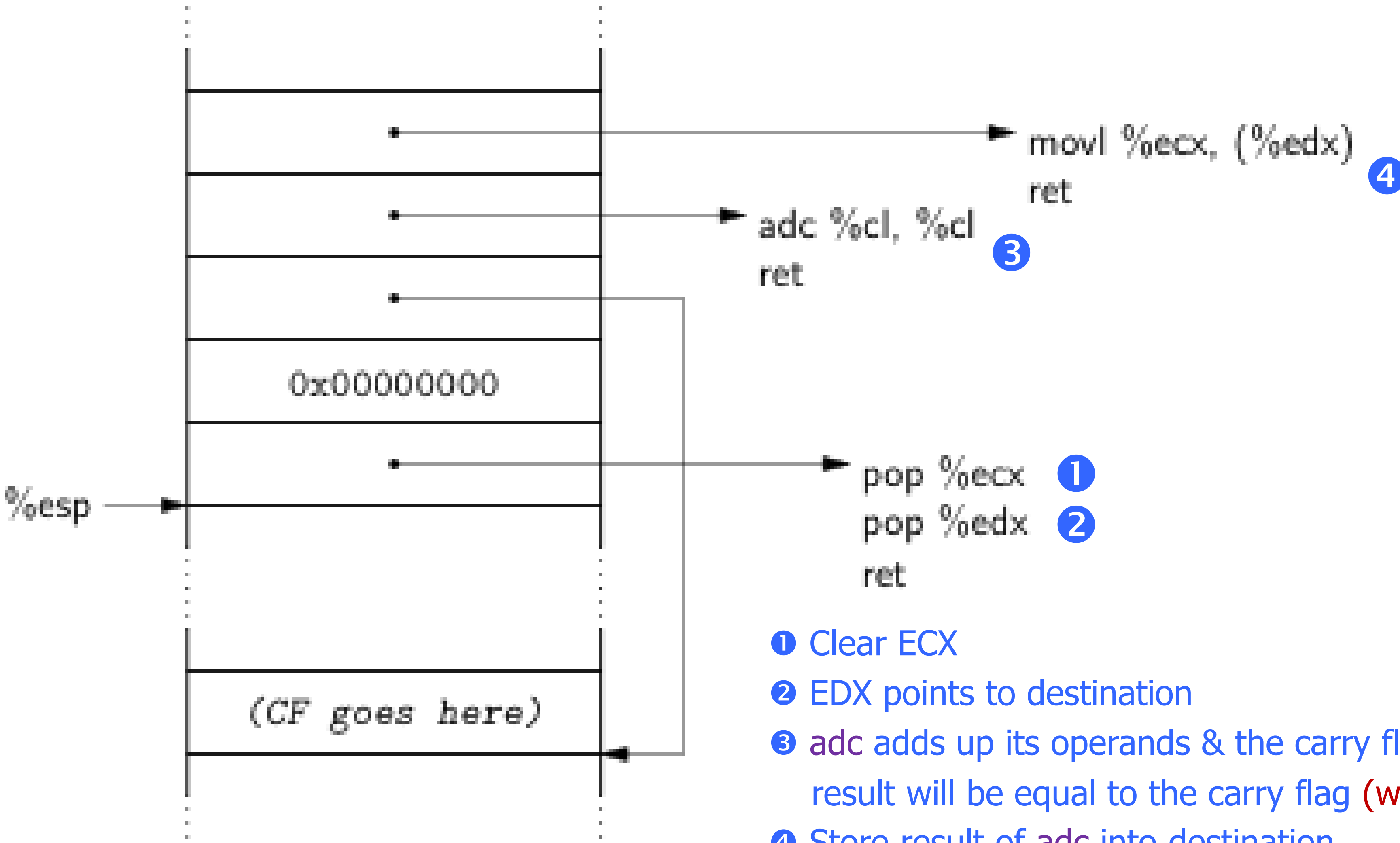
- cmp compares operands and sets a number of flags in the EFLAGS register
  - Luckily, many other ops set EFLAGS as a side effect
- jcc jumps when flags satisfy certain conditions
  - But this causes a change in EIP… not useful (why?)
- Need conditional change in stack pointer (ESP)
- Strategy:
  - Move 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



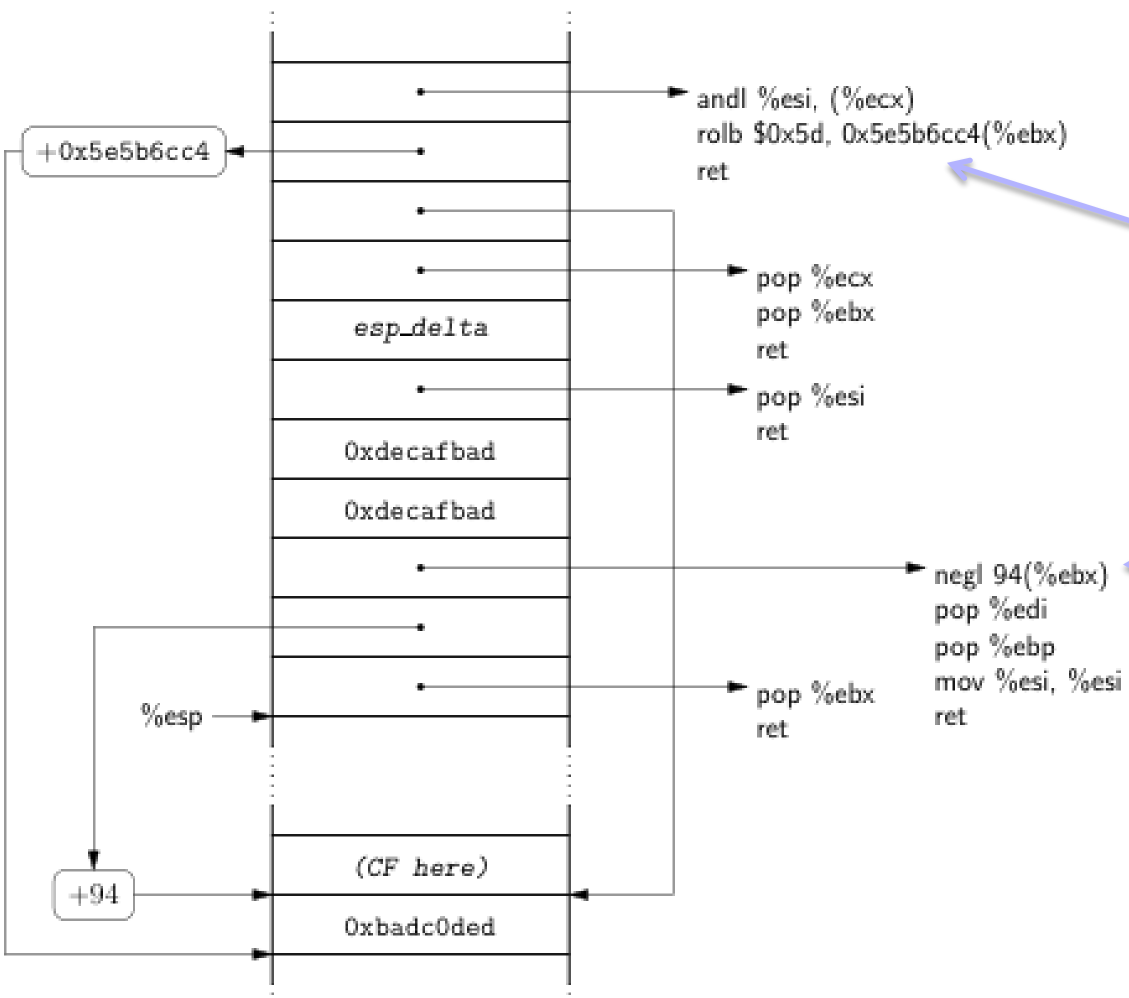
- neg calculates two's complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- sub is similar, use to test if one number is greater than another

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



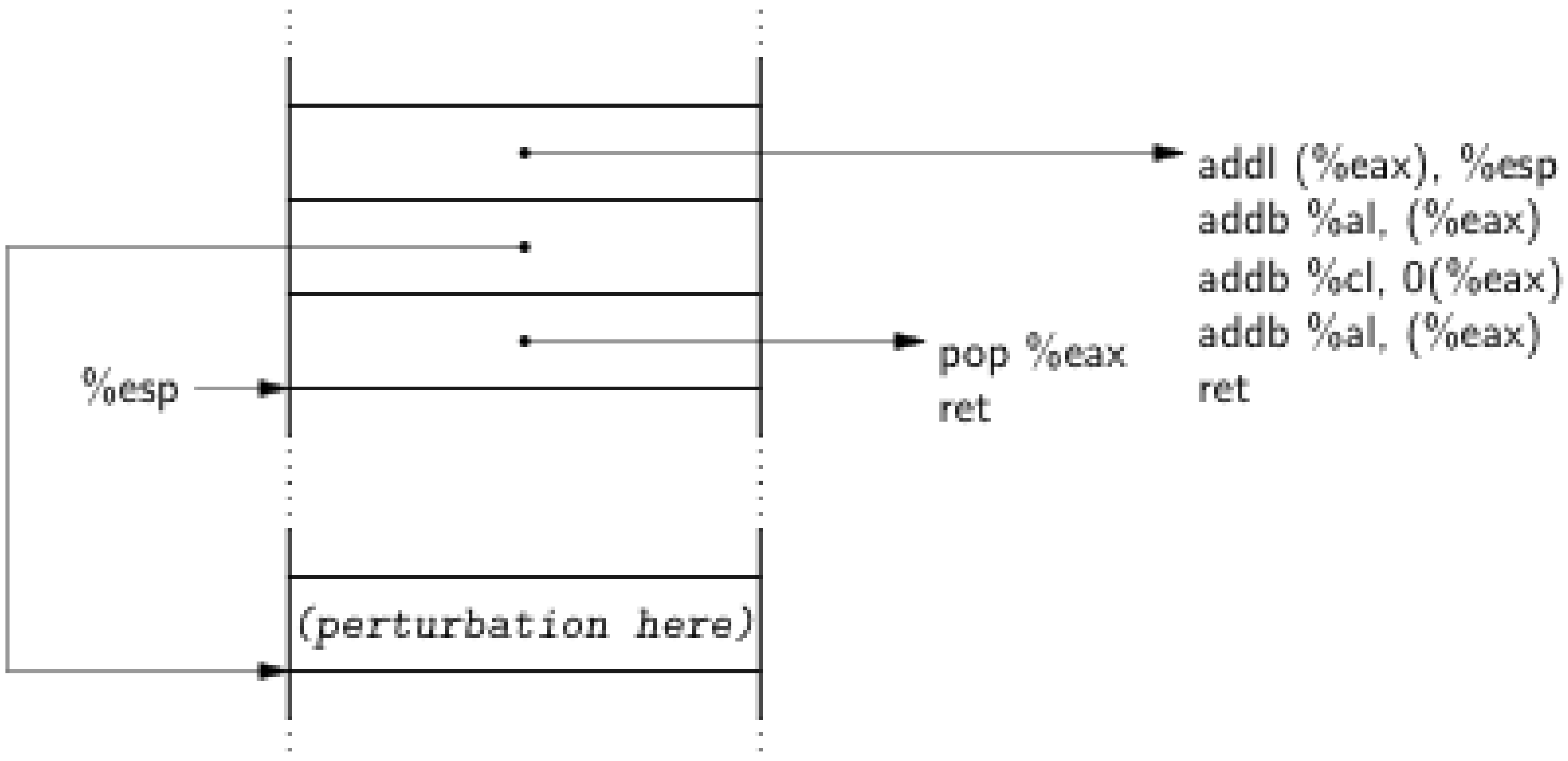
Bitwise AND with delta (in ESI)

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 found instruction sequences in a trie

Unintended Instructions

Actual code from ecb_crypt()

movl $0x00000001, -44(%ebp)

test $0x00000007, %edi

setnzb -61(%ebp)

c7
45
d4
01
00
00
00
f7
c7
07
00
00
00
0f
95
45
c3

add %dh, %bh

movl $0x0F000000, (%edi)

xchg %ebp, %eax

inc %ebp

ret

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

- Use instruction sequences that are <u>suffixes</u> 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!
  - "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC" (CCS 2008)

Proposed ROP Defenses

- Eliminate code sequences with RET
- Look for violations of LIFO call-return order
  - kBouncer - winner of 2012 MS BlueHat Prize ($200K)
  - Observation about legitimate RETs: they return to instructions right after CALLs
  - Modern Intel CPUs store sources and targets of last 4-16 branches in special registers
    - Direct hardware support, zero overhead
  - When application enters the kernel (system call), check that the target of every recorded RET follows a CALL
    - Why check only on kernel entry?

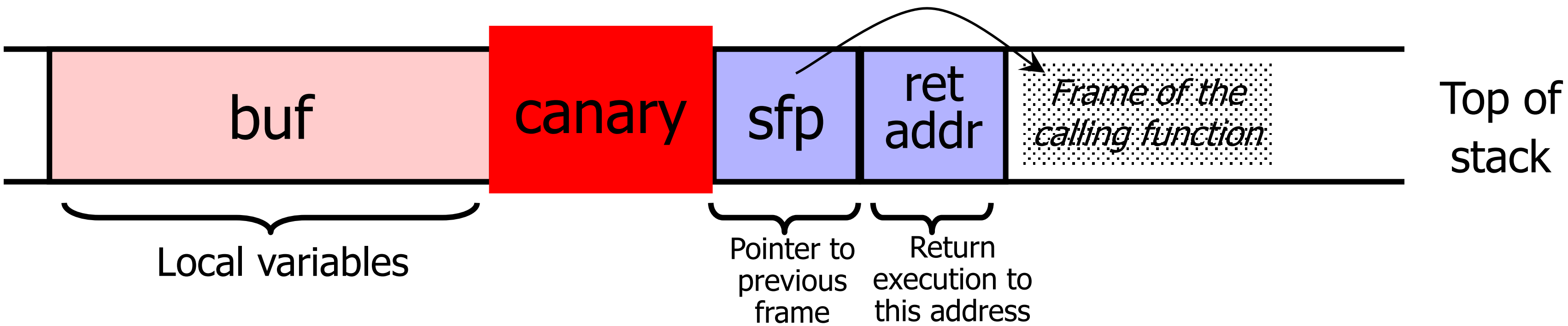
Defeating ROP Defenses

[Checkoway et al.]

- "Jump-oriented" programming
  - Use update-load-branch sequences instead of returns + a trampoline sequence to chain them together
  - "Return-oriented programming w/o returns" (CCS 2010)
- Craft a separate function call stack and call legitimate functions present in the program
  - Checkoway et al.'s attack on Sequoia AVC Advantage voting machine
  - Harvard architecture: code separate from data $\Rightarrow$ code injection is impossible, but ROP works fine
    - Similar issues on some ARM CPUs (think iPhone)

StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



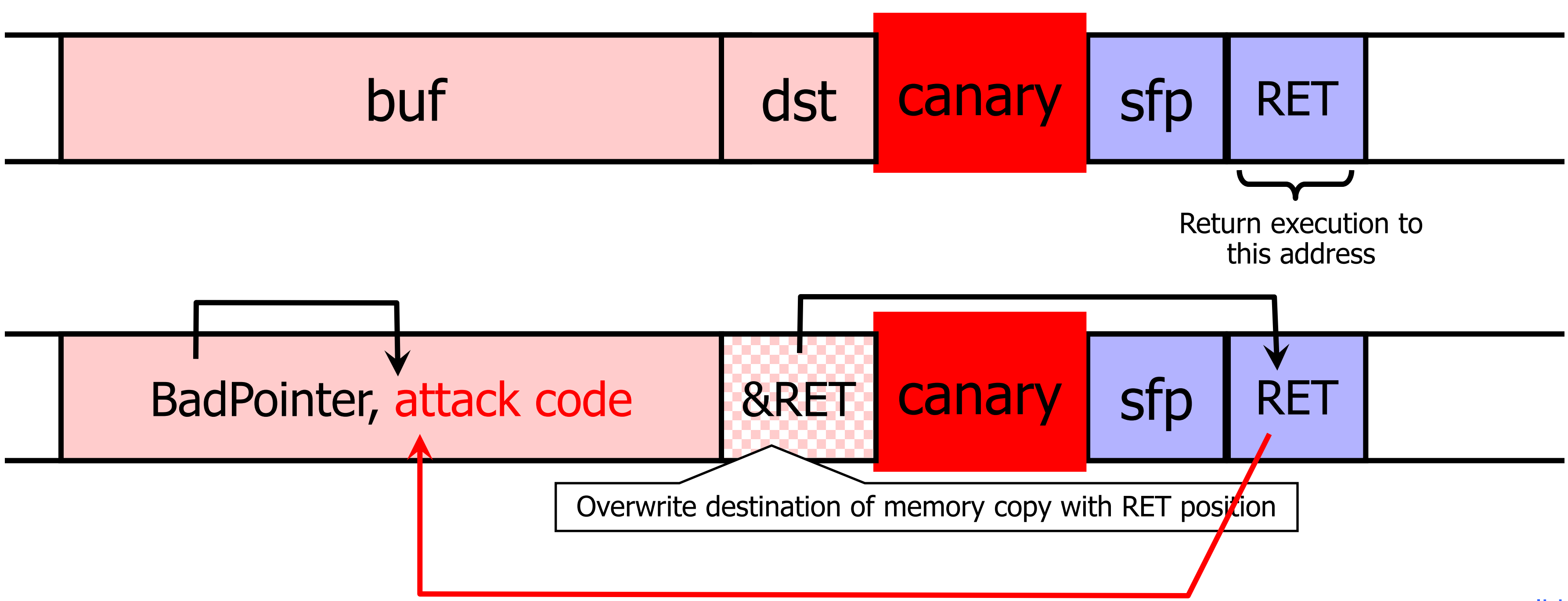
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory copy where the attacker controls both the source and the destination is sufficient

Defeating StackGuard

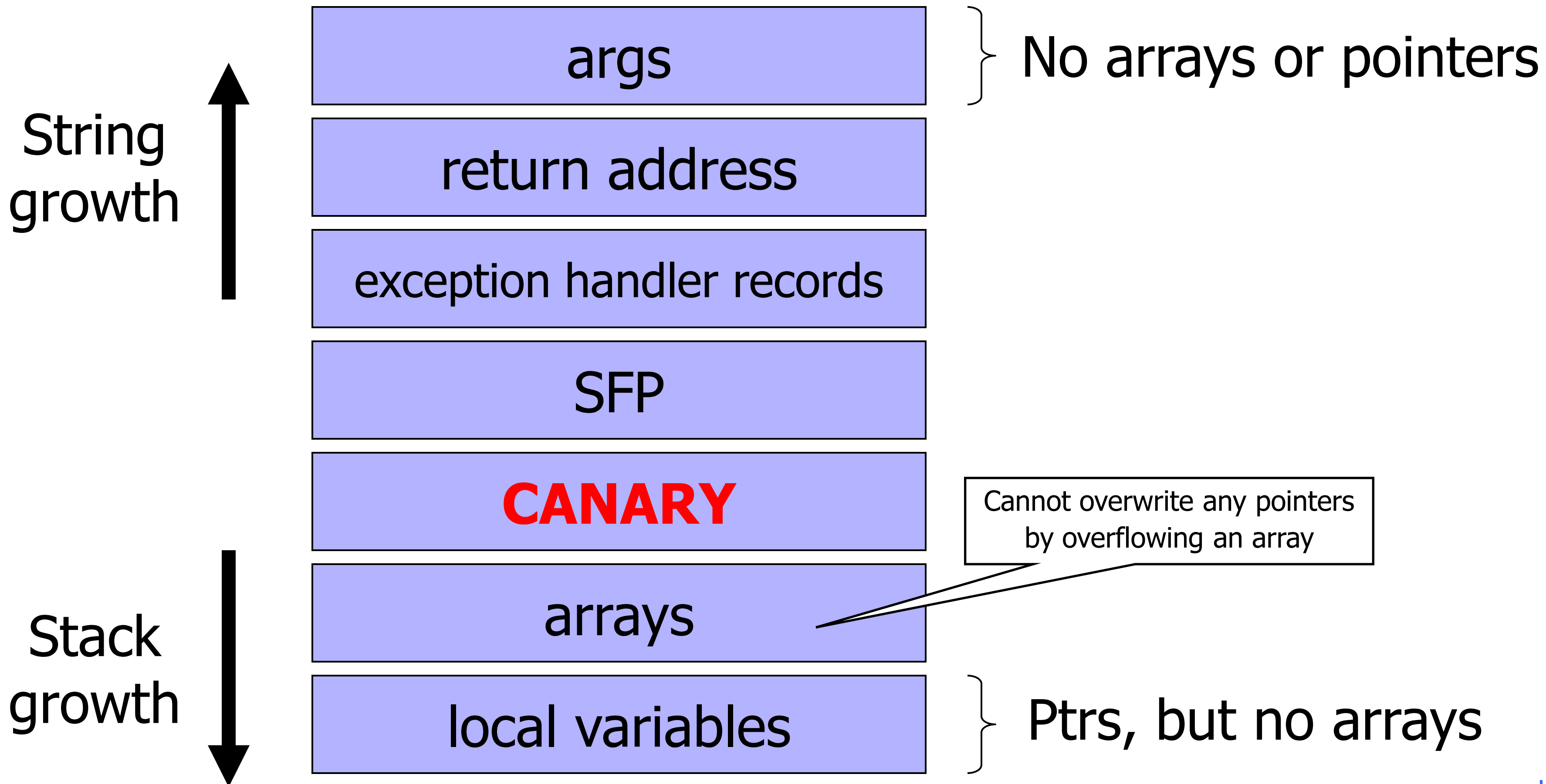
- Suppose program contains *dst=buf[0] where attacker controls both dst and buf
  - Example: dst is a local pointer variable



ProPolice / SSP

[IBM, used in gcc 3.4.1; also MS compilers]

- Rerrange stack layout (requires compiler mod)



What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Any data stored on the stack
  - Exception handling records
  - Pointers to virtual method tables
    - C++: call to a member function passes as an argument "this" pointer to an object on the stack
    - Stack overflow can overwrite this object's vtable pointer and make it point into an attacker-controlled area
    - When a virtual function is called (how?), control is transferred to attack code (why?)
    - Do canaries help in this case?

      (Hint: when is the integrity of the canary checked?)

Code Red Worm (2001)

[Chien and Szor, "Blended Attacks"]

- A malicious URL exploits buffer overflow in a rarely used URL decoding routine in MS-IIS …
- … the stack-guard routine notices the stack has been smashed, raises an exception, calls handler
- … pointer to exception handler located on the stack, has been overwritten to point to CALL EBX instruction inside the stack-guard routine
- … EBX is pointing into the overwritten buffer
- … the buffer contains the code that finds the worm's main body on the heap and executes it

Safe Exception Handling

- Exception handler record must be on the stack of the current thread
- Must point outside the stack (why?)
- Must point to a valid handler
  - Microsoft's /SafeSEH linker option: header of the binary lists all valid handlers
- Exception handler records must form a linked list, terminating in FinalExceptionHandler
  - Windows Server 2008: SEH chain validation
  - Address of FinalExceptionHandler is randomized (why?)

SEHOP

- SEHOP: Structured Exception Handling Overwrite Protection (since Win Vista SP1)
- Observation: SEH attacks typically corrupt the “next” entry in SEH list
- SEHOP adds a dummy record at top of SEH list
- When exception occurs, dispatcher walks up list and verifies dummy record is there; if not, terminates process

Non-Control Targets

[Chen et al. "Non-Control-Data Attacks Are Realistic Threats"]

- Configuration parameters
  - Example: directory names that confine remotely invoked programs to a portion of the file system
- Pointers to names of system programs
  - Example: replace the name of a harmless script with an interactive shell
  - This is not the same as return-to-libc (why?)
- Branch conditions in input validation code
- None of these exploits violate the integrity of the program's control flow
  - Only original program code is executed!

SSH Authentication Code

[Chen et al. "Non-Control-Data Attacks Are Realistic Threats"]

```
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

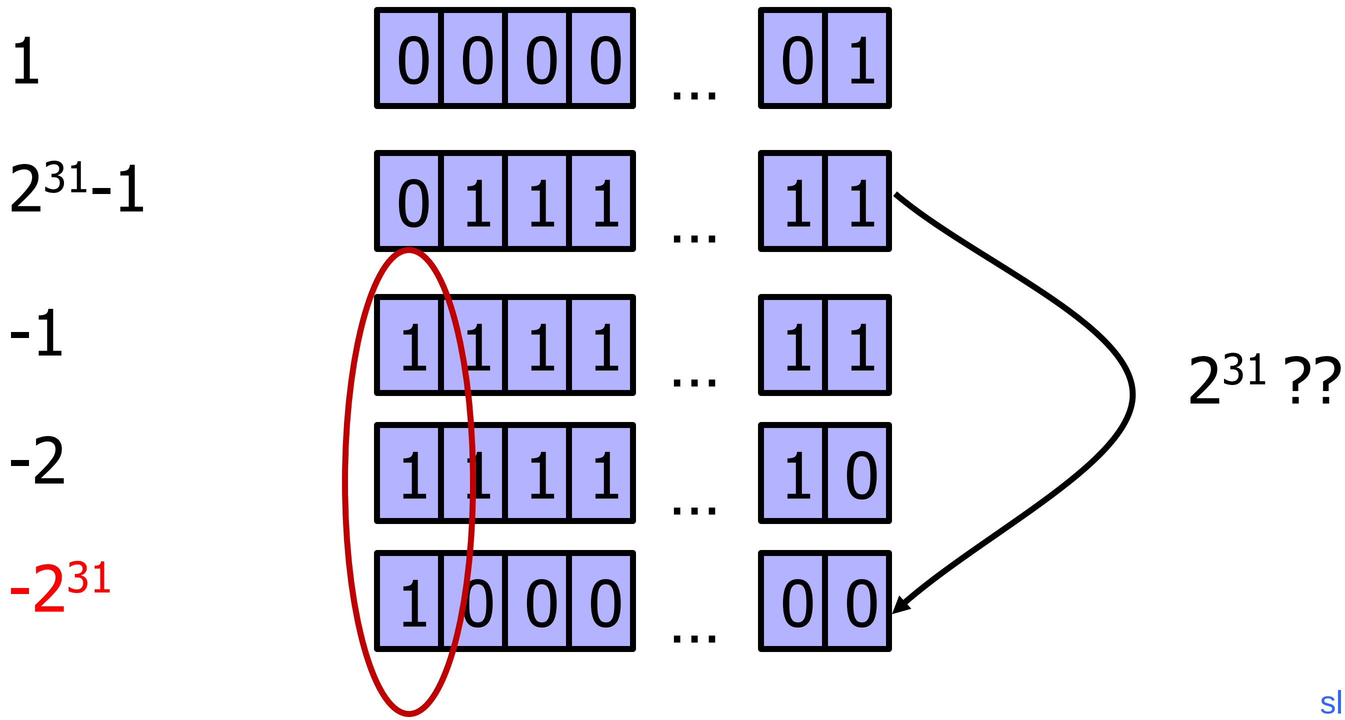
detect_attack() prevents checksum attack on SSH1…

…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

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
  - Internet Explorer and Firefox use different Flash binaries, but this turns out not to matter
- Exploit published in April 2008
  - “Leveraging the ActionScript Virtual Machine”

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 needed 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 short value into any location in memory (why?)
  - … subject to some restrictions (see paper)
  - But this is <u>not</u> 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

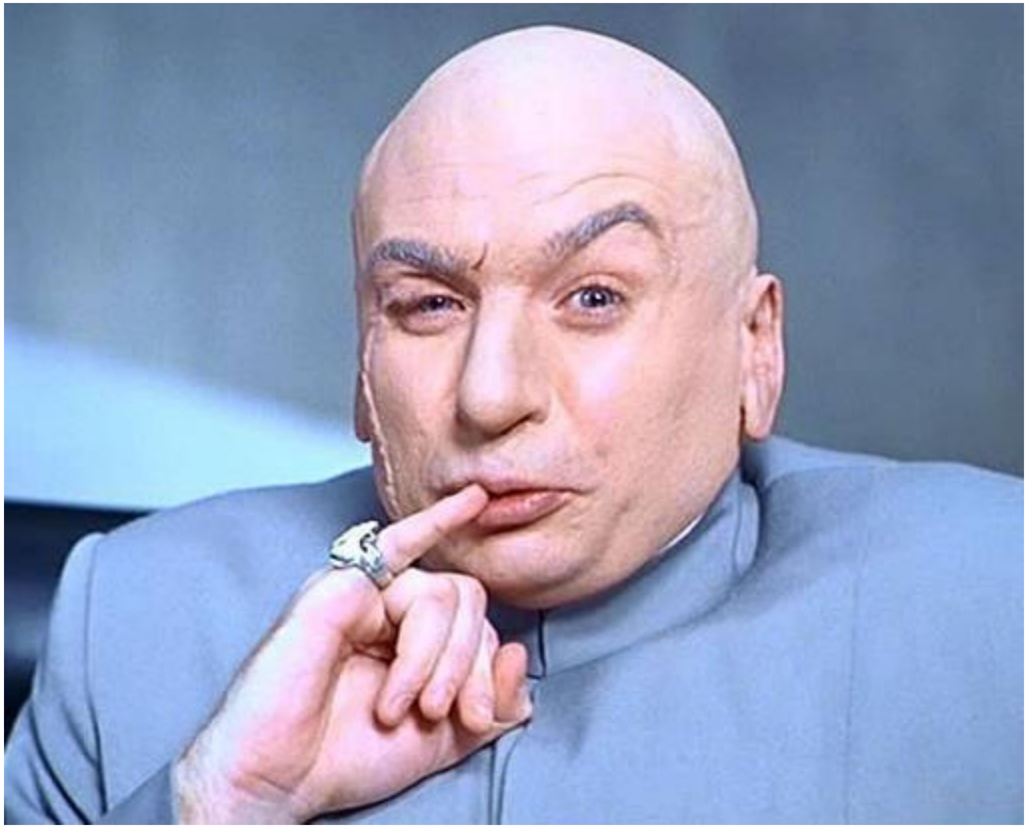
Number of operands for each opcode is defined in AS3_argmask array

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 <u>data</u> (operands) by the verifier, but as executable <u>code</u> 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
- 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 contains a jump to the actual shellcode
- Read the paper for more details