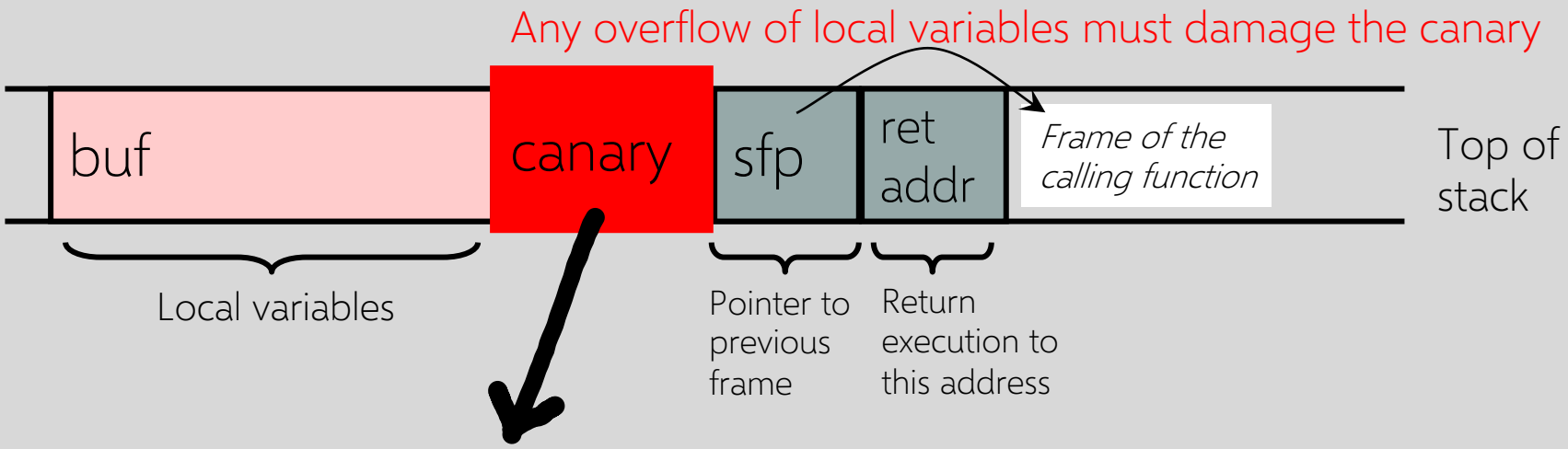
MEMORY PROTECTION TECHNIQUES

VITALY SHMATIKOV

StackGuard

Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to returning from the function



Choose random value on program start
(attacker can't guess what this value will be)

or

terminator canary: "\0", newline, linefeed, EOF
because strcpy, etc. won't copy beyond "\0"

StackGuard / Canary Implementation

StackGuard requires code recompilation

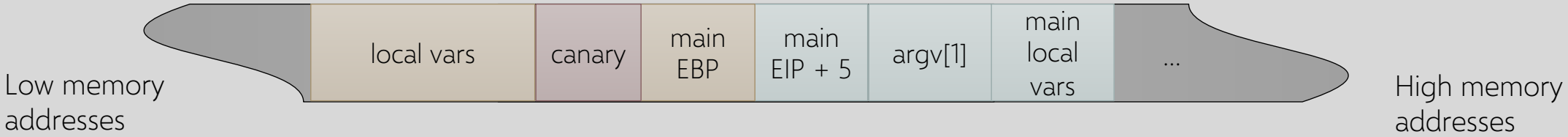
Checking canary integrity prior to every function return causes a performance penalty

- 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
- Or the attacker can infer the value of the canary

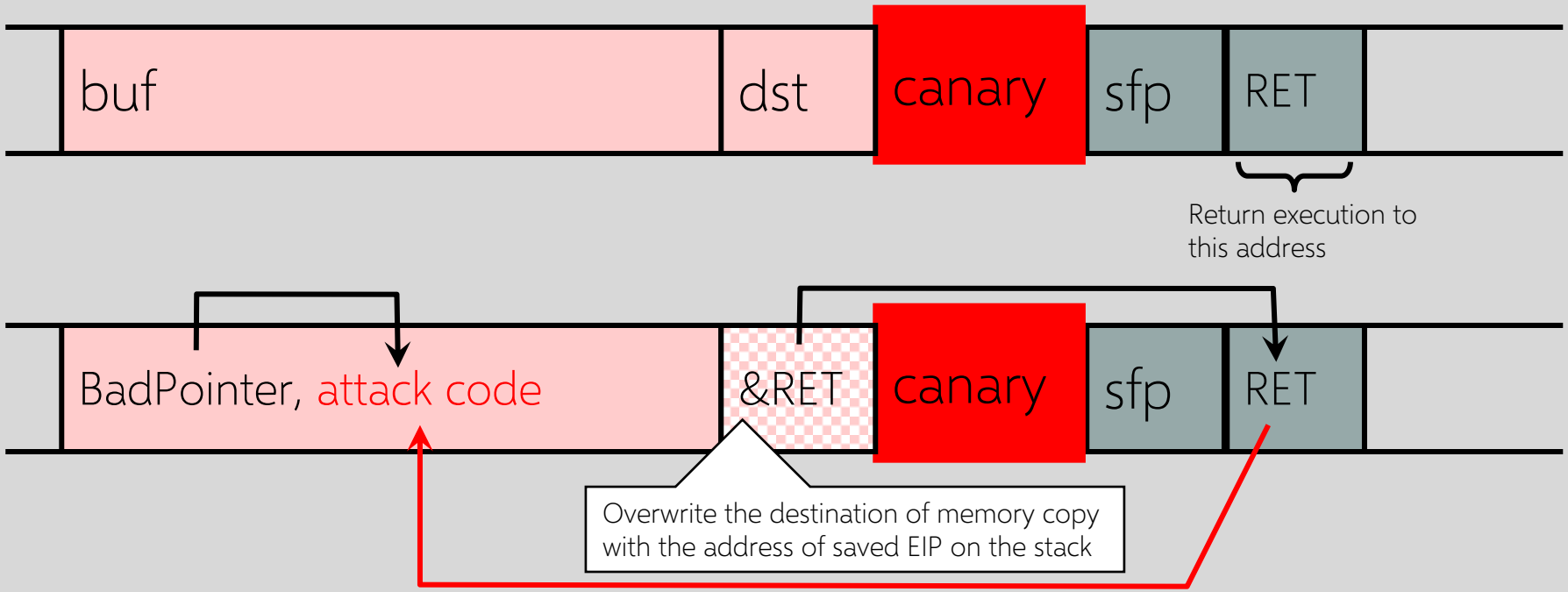
Stack Canaries in gcc



Flag	Default?	Notes
-fno-stack-protector	No	Turns off protections
-fstack-protector	Yes	Adds to funcs that call alloca() & w/ arrays larger than 8 chars (--param=ssp-buffer-size changes 8)
-fstack-protector-strong	No	Also funcs w/ any arrays & refs to local frame addresses
-fstack-protector-all	No	All funcs

Defeating StackGuard

Suppose the program contains *dst=buf[0] where the attacker controls both dst and buf

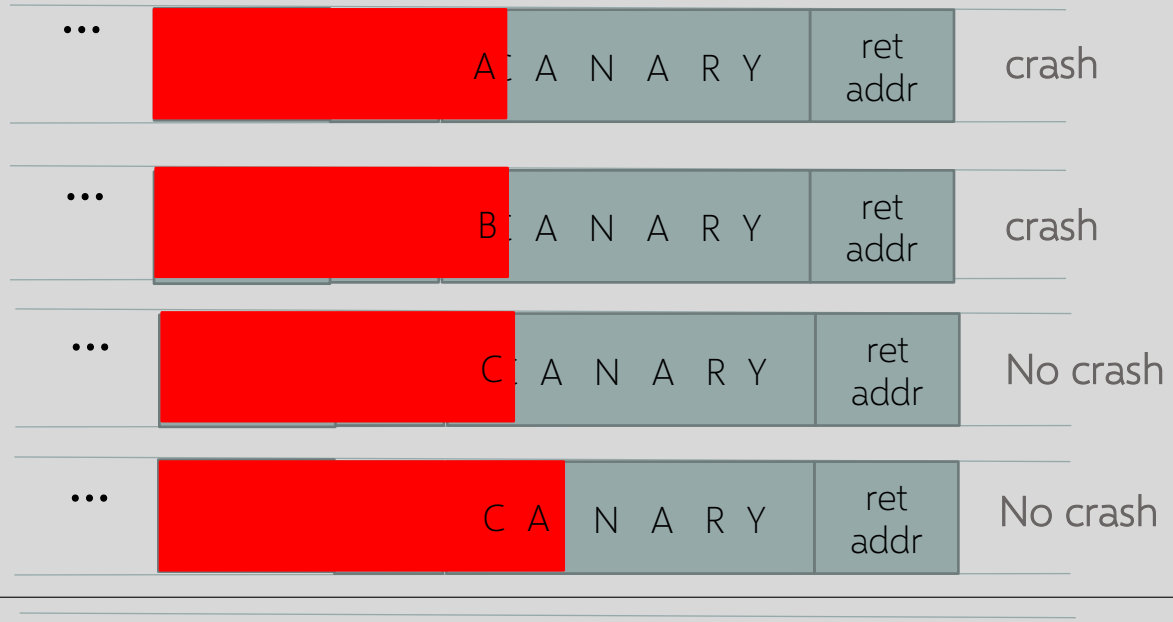


“Reading” the Stack for the Canary Value

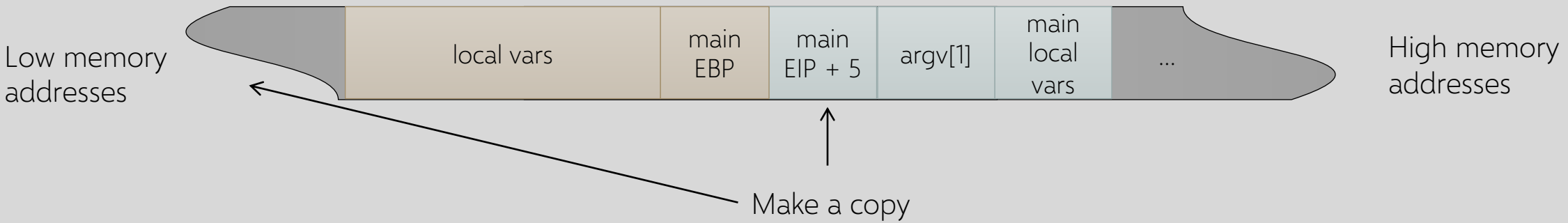
A common design for crash recovery:

- When process crashes, restart automatically (for availability)
- If relaunched using fork, canary is unchanged

Attacker can now extract the canary byte by byte



StackShield



StackShield:

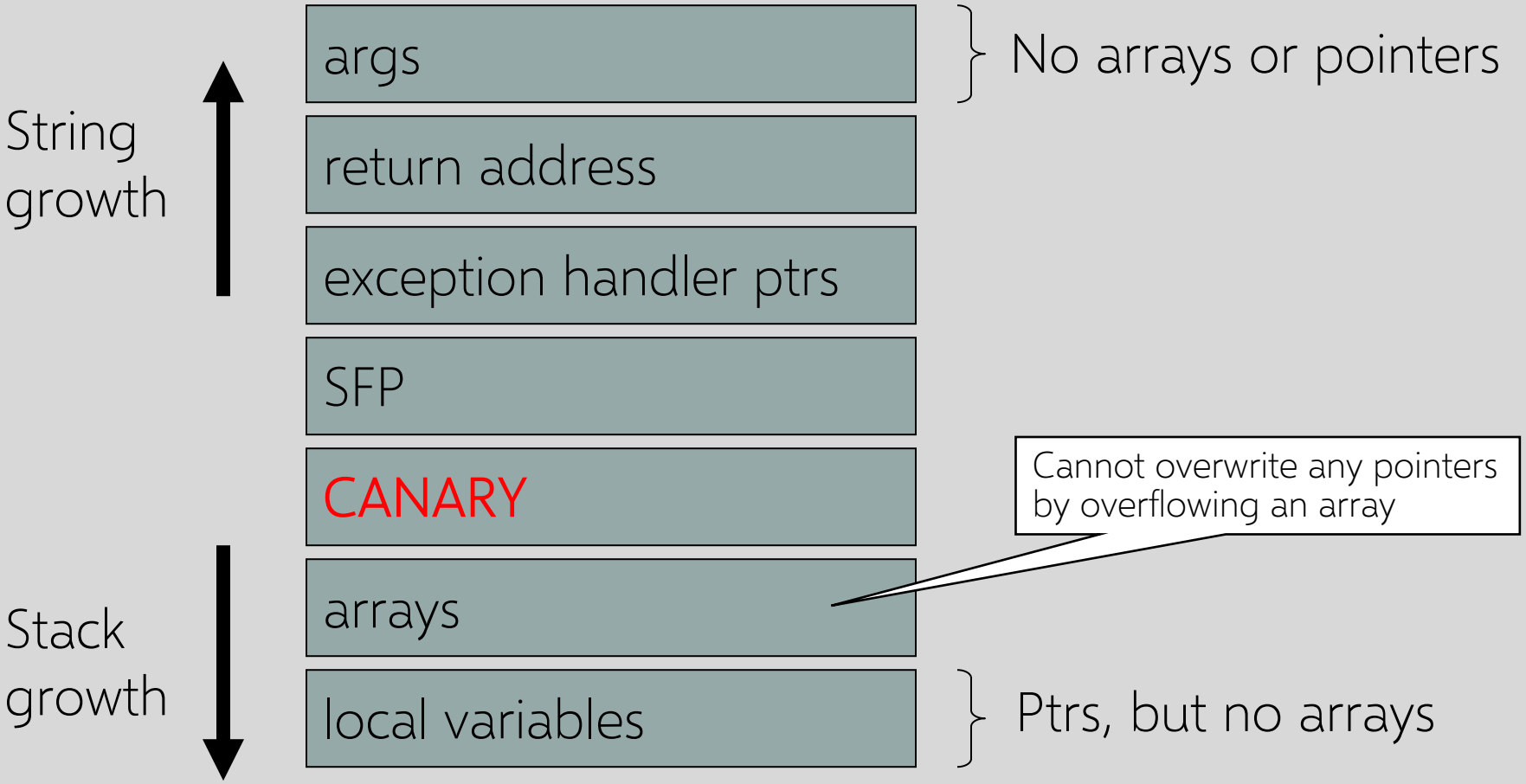
- Function call: copy return address to safer location (beginning of .data)
- Function exit: check if stack value is different on function exit

Circumvention:

- Overwrite both the return address & saved copy, if possible
- Hijack control flow without overwriting the return address

IBM, used in gcc 3.4.1; also MS compilers

ProPolice / SSP: Safer Stack Layout



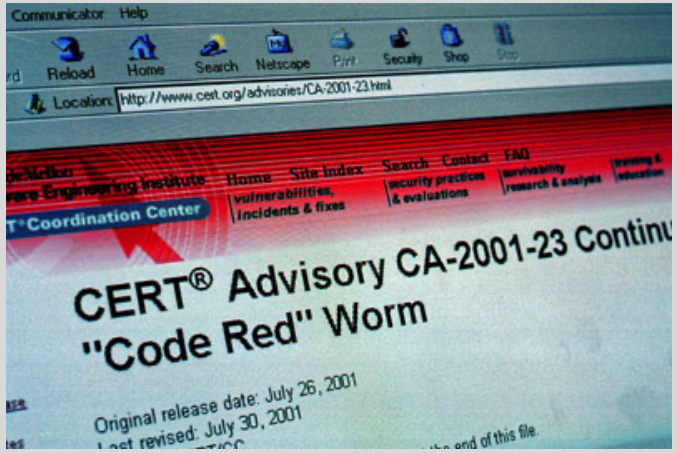
What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Exception handling records (stored on the stack!)
- 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 memory
  - 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)



exception handler

...

CANARY

stack smashed

buffer

Overwritten to point to a CALL EBX instruction inside the stack-guard routine

...
CALL EBX
...

stack guard routine notices, calls the exception handler

Finds the worm's main body and executes it

URL decoding routine in Microsoft Web server

EBX register

Worm's main body on the heap

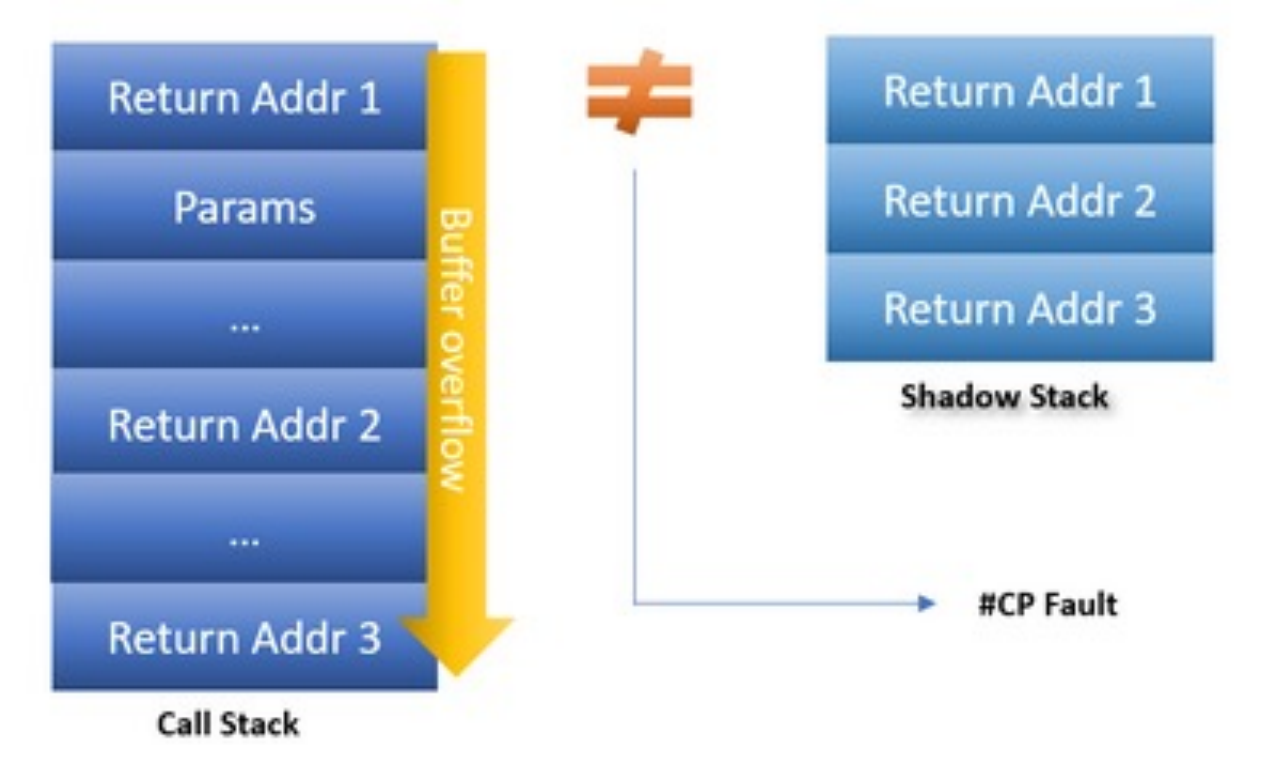
Safe Exception Handling

- 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

Shadow Stack



- On every CALL instruction, return addresses are pushed onto both the call stack and shadow stack
- On RET instructions, a comparison is made. If addresses don't match, CPU raises an exception that traps into the kernel.

On chipsets with Intel's Control-Flow Enforcement Technology (CET) instructions

https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815

Non-Control Targets

- 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 the original program code is executed!

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

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 the authentication loop without authenticating properly

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

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

GHTTPD Web Server

ptr changes after it was checked
but before it was used!
(Time-Of-Check-To-Time-Of-Use attack)

Check that URL doesn't contain "/.."

```
int serveconnection(int sockfd) {
    char *ptr;  // pointer to the URL.
                // ESI is allocated
                // to this variable.
    ...
1:  if (strstr(ptr,"/..")
      reject the request;
2:  log(...);
3:  if (strstr(ptr,"cgi-bin"))
4:      Handle CGI request
    ...
}
```

Register containing pointer to URL
is pushed onto stack...

```
Assembly of log(...)
    push %ebp
    mov %esp, %ebp
    push %edi
    push %esi
    push %ebx
    ... stack buffer overflow code
    pop %ebx
    pop %esi
    pop %edi
    pop %ebp
    ret
```

... overflown

... and read from stack

At this point, overflown ptr may point
to a string containing "/.."

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

Predictability is a fatal flaw in any defensive system

Problem: Lack of Diversity

Classic memory exploits need to know the (virtual) address to hijack control

- Address of attack code in the buffer
- Address of a standard kernel library routine

Same address is used on many machines

- Slammer infected 75,000 MS-SQL servers in 10 minutes using identical code on every machine

Idea: introduce artificial diversity

- Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

ASLR

Address Space Layout Randomization

Randomly choose the base address of stack, heap, code segment, the location of the Global Offset Table

- Randomization can be done at compile- or link-time, or by rewriting existing binaries

Randomly pad stack frames and malloc'ed areas

Other randomization methods: randomize system call ids or even instruction set

ASLR in Action

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Booting twice loads libraries in different locations

Base-Address Randomization

Only the base address is randomized

- Layouts of stack and library table remain the same
- Relative distances between memory objects are not changed by base address randomization

To attack, enough to guess the base shift

A 16-bit value can be guessed by brute force

- Try 2^{15} (on average) overflows with different values for the address of a known library function – how long does it take?
- If guess is wrong, target will simply crash & restart

usleep() is a good candidate (why?)

Brute-Force Guessing

Attacker makes a guess of where usleep() is located in memory

request

top of stack (higher addresses)
⋮
0x01010101
0xDEADBEEF
guessed address of `usleep()`
0xDEADBEEF
64 byte buffer, now filled with A's
⋮
bottom of stack (lower addresses)

Figure 2: Stack after one probe

Web server with a buffer overflow

Failure crashes the child process immediately and therefore kills connection

Success crashes the child process after sleeping for 0x01010101 microseconds and kills connection

Unlikely to work on a 64-bit architecture

ASLR in Windows (since 2008)

Stack randomization

- Find N^{th} hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)

Heap randomization: 5 bits

- Linear search for base + random 64K-aligned offset

EXE randomization: 8 bits

- Preferred base + random 64K-aligned offset

DLL randomization: 8 bits

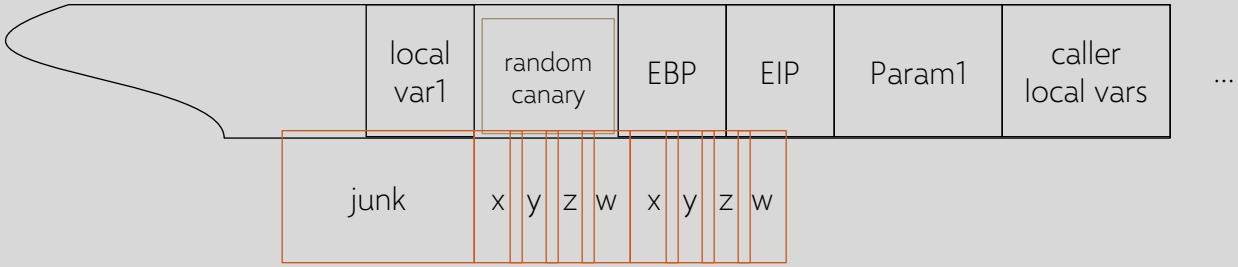
- Random offset in DLL area; random loading order

Defeating ASLR by Reading the Stack

Request (can trigger buffer overflow in stack)

Apache forks
off child process
to handle request

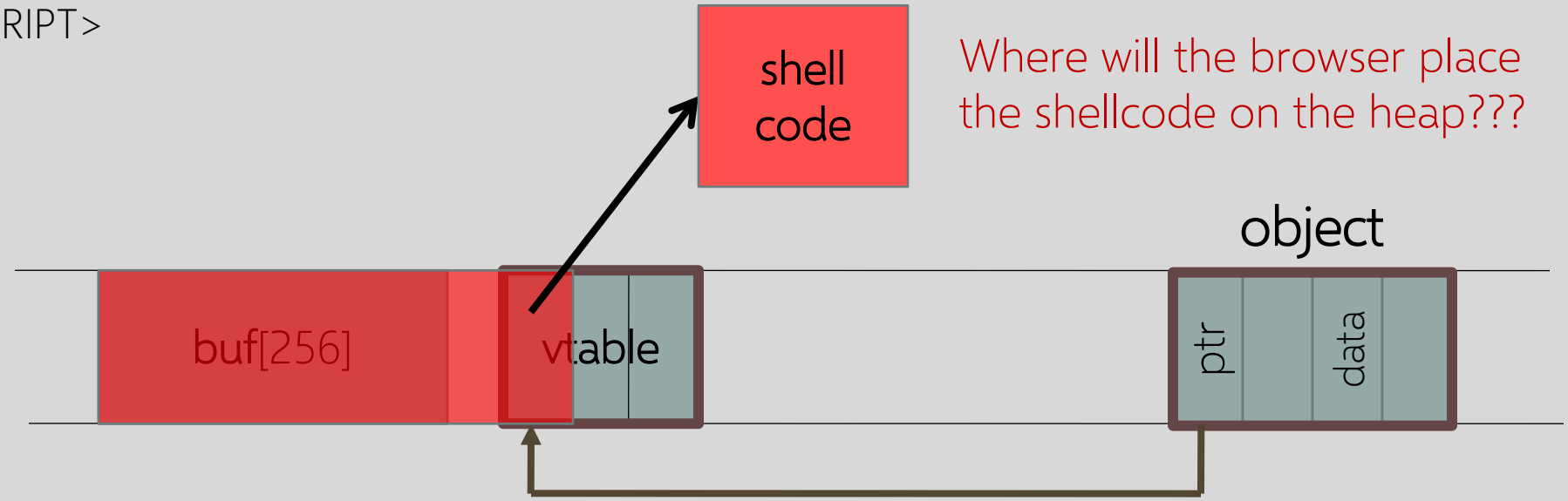
Response (unless process crashes)



Reading the stack for EBP/EIP can give approximate address offset

ASLR Against Heap Attacks

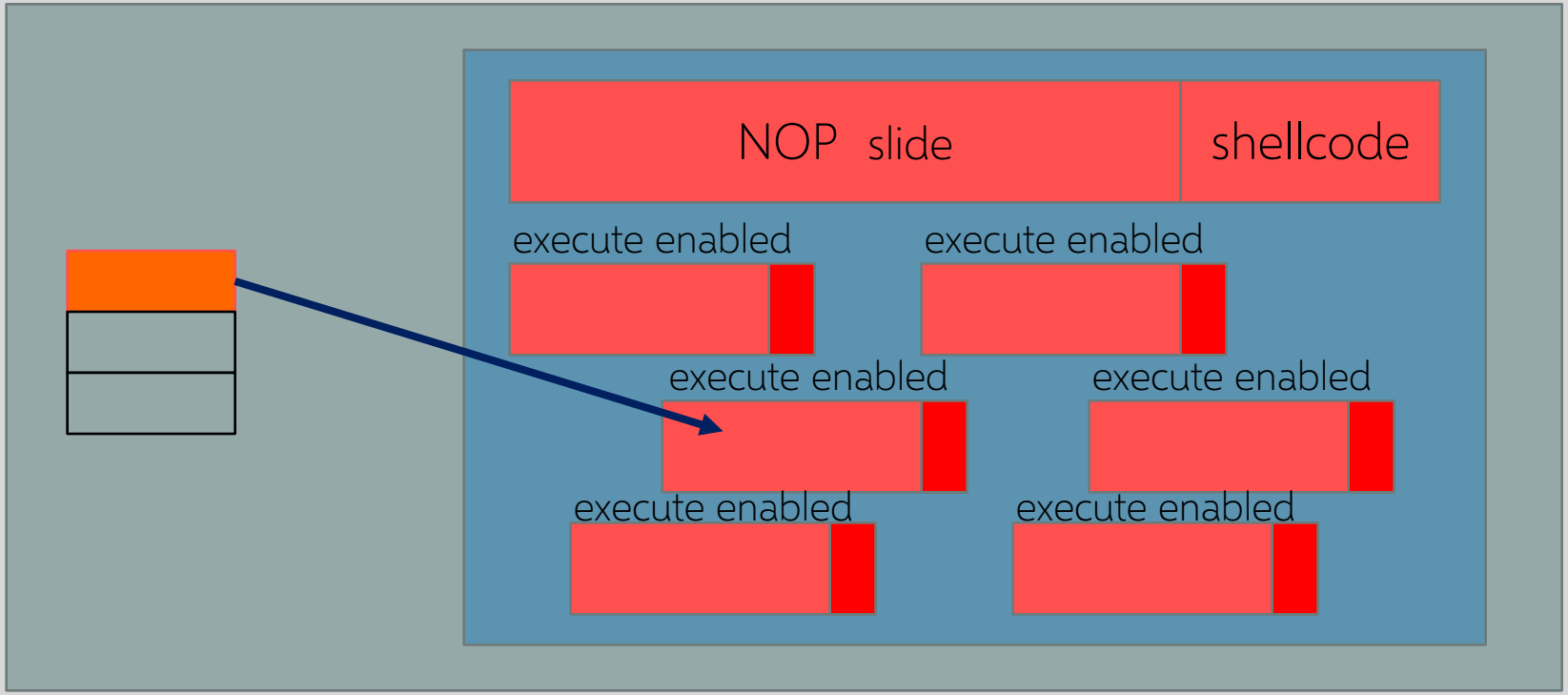
```
<SCRIPT language="text/javascript">
        shellcode = unescape("%u4343%u4343%...");
        overflow-string = unescape("%u2332%u4276%...");
        cause-overflow( overflow-string );  // overflow  buf[ ]
</SCRIPT>
```



Heap Spraying

JIT is a great target: it creates code on the fly, thus its memory must be writeable, readable, executable

Force JavaScript JiT ("just-in-time" compiler) to fill heap with executable shellcode, then point SFP or vtable ptr anywhere in the spray area



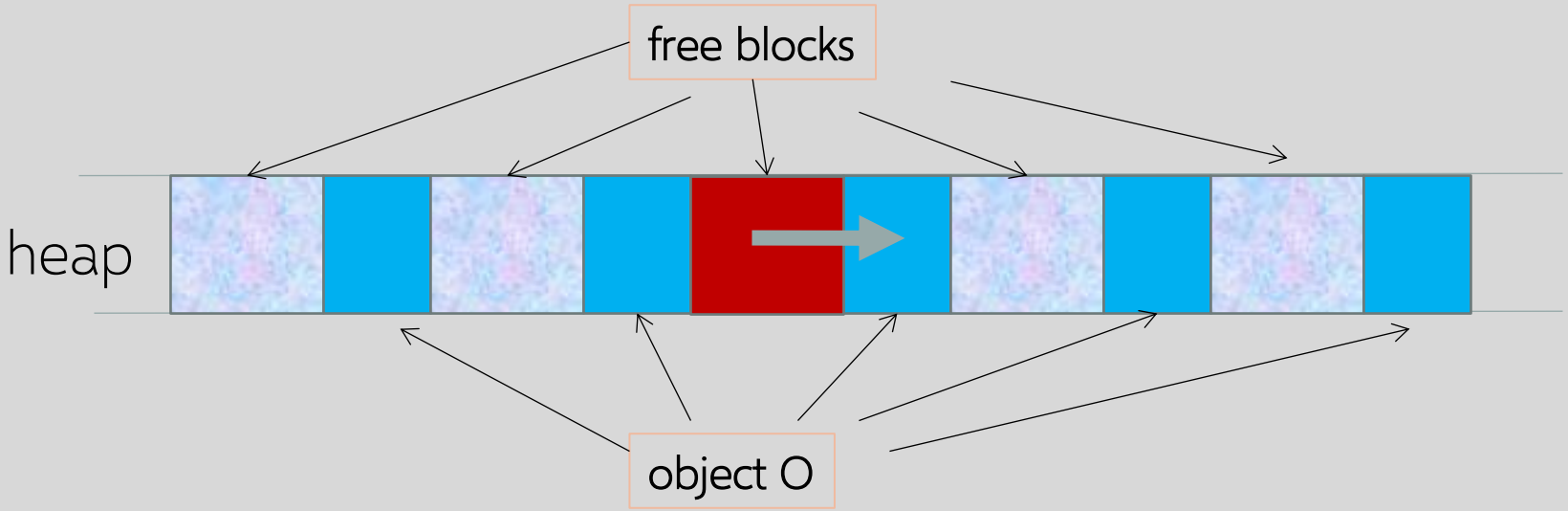
JavaScript Heap Spraying

```
var  nop = unescape(“%u9090%u9090”)
while (nop.length < 0x100000)  nop += nop
var shellcode = unescape("%u4343%u4343%...");
var x = new Array ()
for (i=0;  i<1000;  i++) {
        x[i] = nop + shellcode;
}
```

Pointing a function pointer anywhere in the heap will cause shellcode to execute

Heap Feng Shui (aka Heap Grooming)

Achieve the desired heap layout by a sequence of crafted memory allocations and deallocations



Safari PCRE exploit (2008): placing the vulnerable object next to attacker-controlled memory

Heap Spraying in EternalBlue

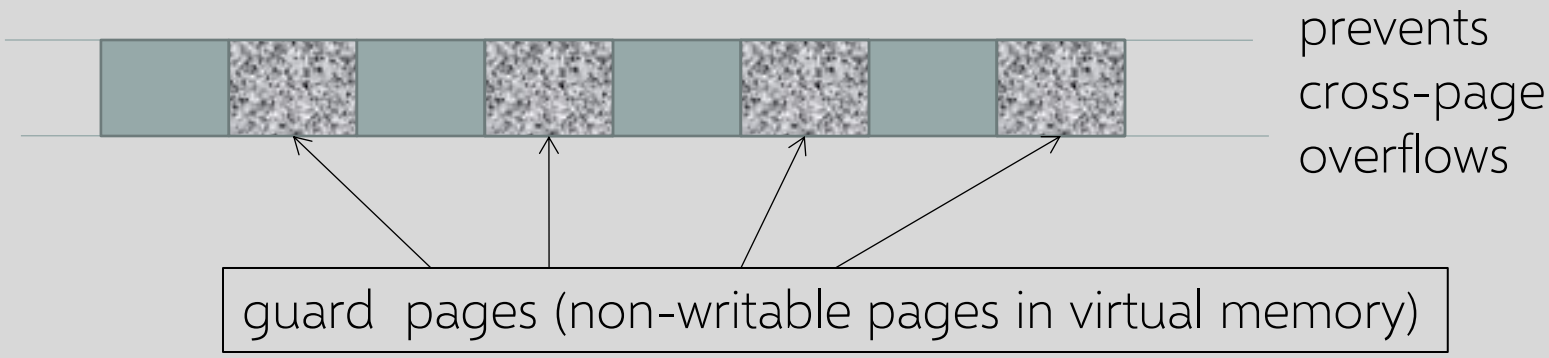
The exploit opens several bare minimum connections, added by a variable *NumGrooms* amount. Grooms are used to perform a type of heap spray attack of kernel pool memory, so that memory lines up correctly and overflow is controlled to a correct location. SMB drivers use large non-paged memory with its own structures for memory management of packets [24]. By adjusting the amount of grooms against a highly-fragmented pool, it is more likely to enter a known state and end up with a successful overwrite of desired structures.

https://risksense.com/wp-content/uploads/2018/05/White-Paper_Eternal-Blue.pdf

Heap Overflow Mitigations

Better browser architecture: store JavaScript strings in a separate heap from browser heap

Better memory layout (OpenBSD and Windows 8):



In theory, could allocate every object on its own page, but too wasteful in physical memory

Memory Attacks: 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, IA-64 "XD" bit, ARMv6 "XN" bit
- 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
- GCC stack trampolines (calling conventions, nested functions)

JVM makes all its memory RWX – readable, writable, executable (why?)

Some applications (eg, browsers) don't use DEP

Attack can start by "returning" into a memory mapping routine and make the page containing attack code writeable

What Does W⊕X Not Prevent?

Can still corrupt stack!

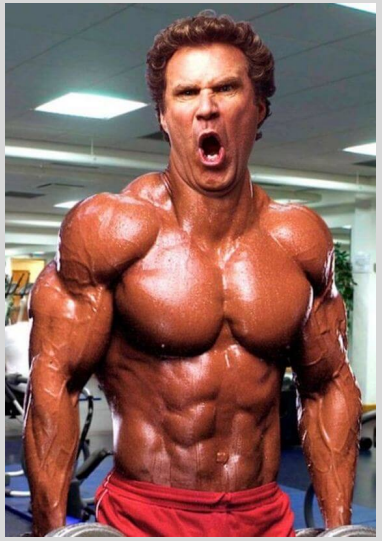
- … or function pointers or critical data on the heap, but that's not important right now

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 the address of any library routine, arrange memory to look like arguments

Does not look like a huge threat, since attacker cannot execute arbitrary code, especially if system() is not available

return-to-libc on Steroids

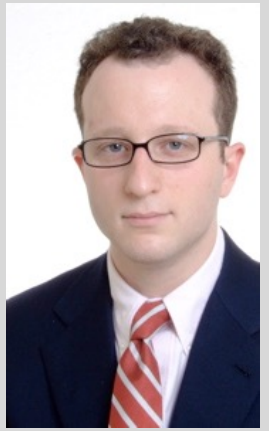


Overwritten saved EIP need not point to the beginning of a library routine. Any existing instruction in the code image is fine… processor will execute the sequence starting from that instruction

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



Hovav Shacham

Can chain together sequences ending in RET

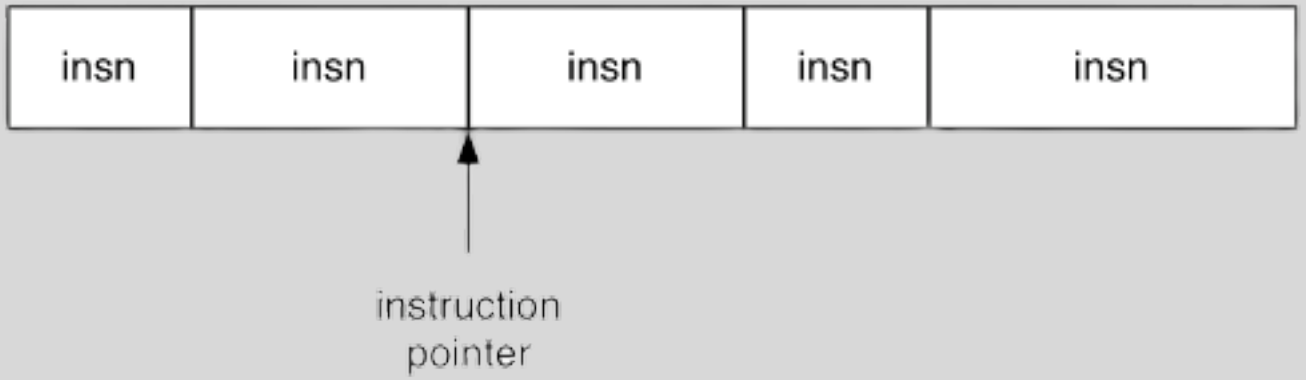
- Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)

What is this good for? 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

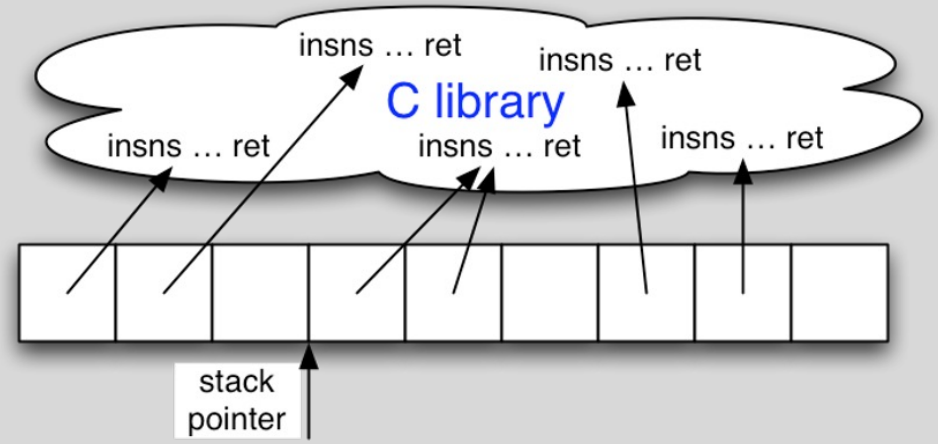
Ordinary Programming



- 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 the value of EIP

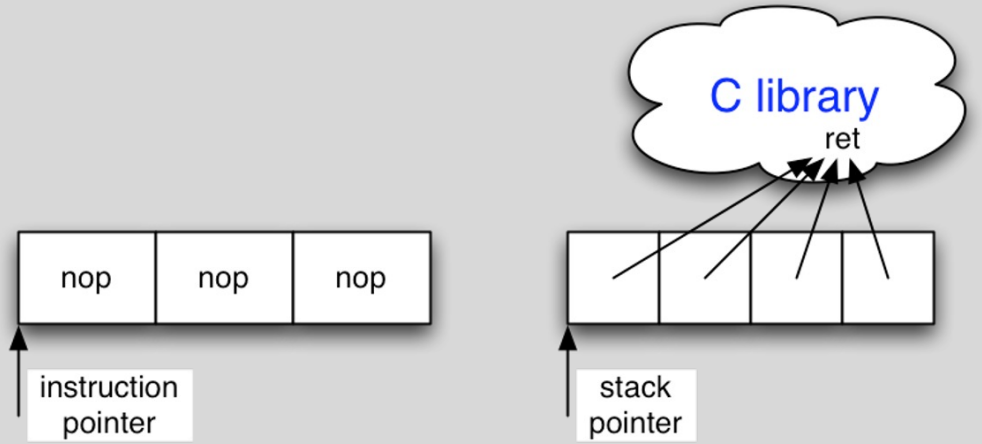
Return-Oriented Programming

Key idea: build arbitrary computations from existing code sequences ending in RET



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

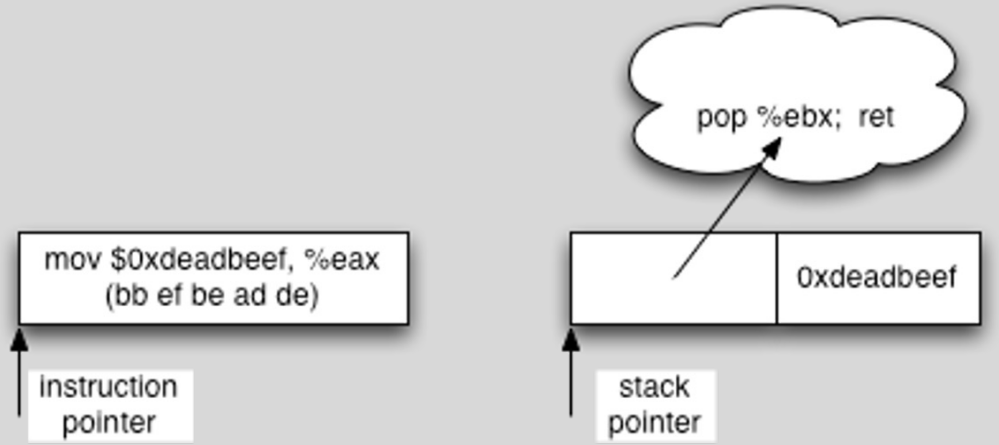
No-Ops (Useful for No-Op Sleds)



No-op instruction does nothing but advance EIP

Return-oriented equivalent: pointer to RET instruction (advances ESP)

Immediate Constants

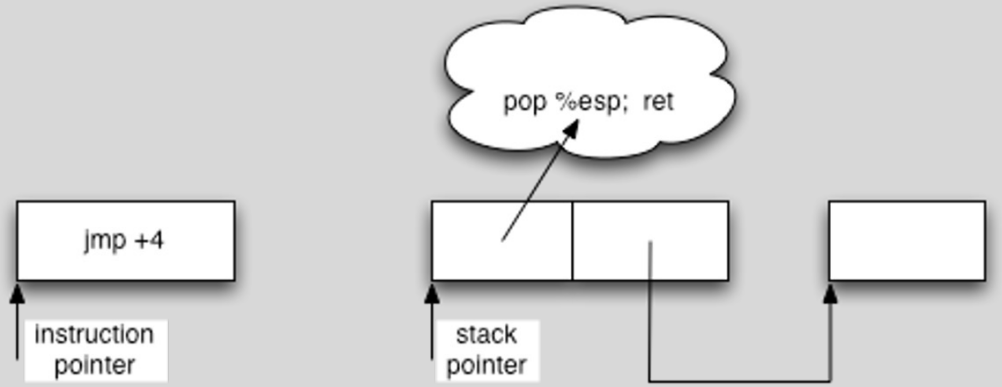


Instructions can encode constants

Return-oriented equivalent:

- Store value on the stack
- Pop into register to use

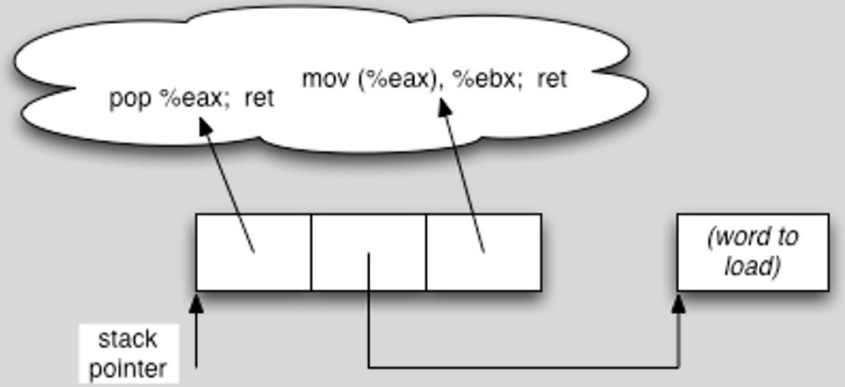
Control Flow



(Conditionally) set EIP to a new value

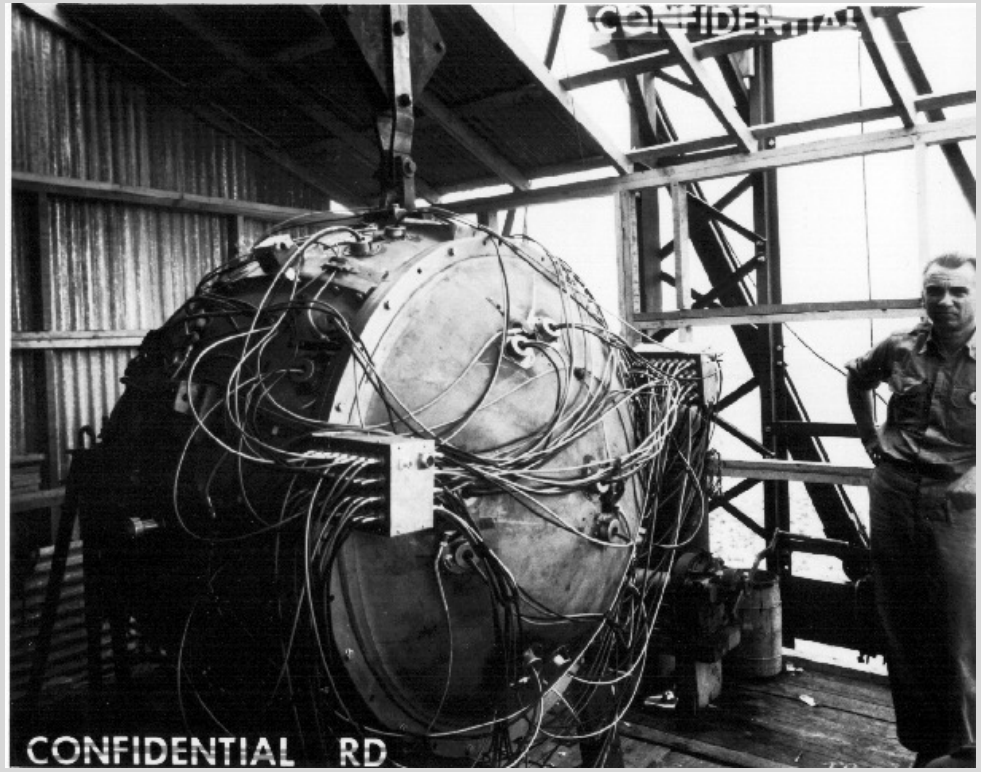
Return-oriented equivalent: (Conditionally) set ESP to a new value

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)



ROP gadgets

- 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

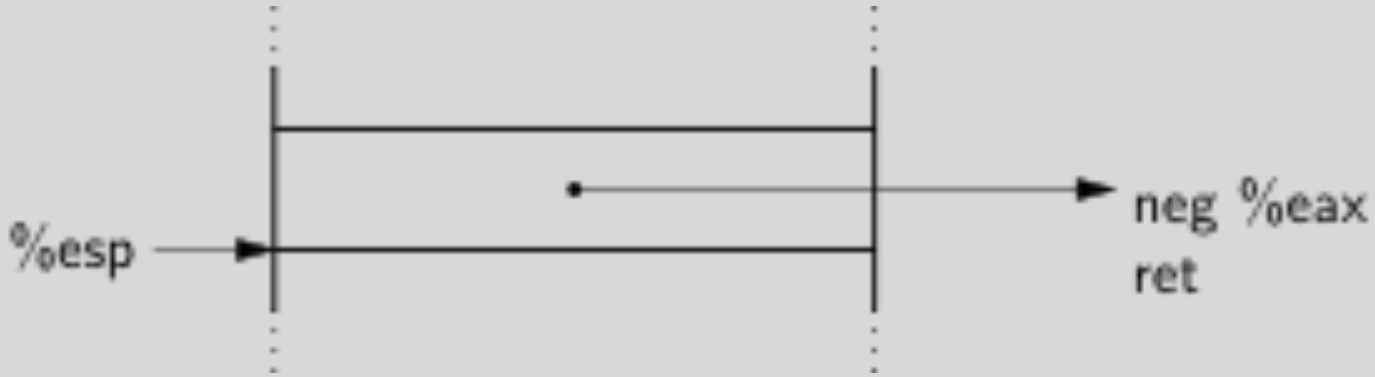
Conventional programming

- cmp compares operands and sets several 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?)

Return-oriented programming

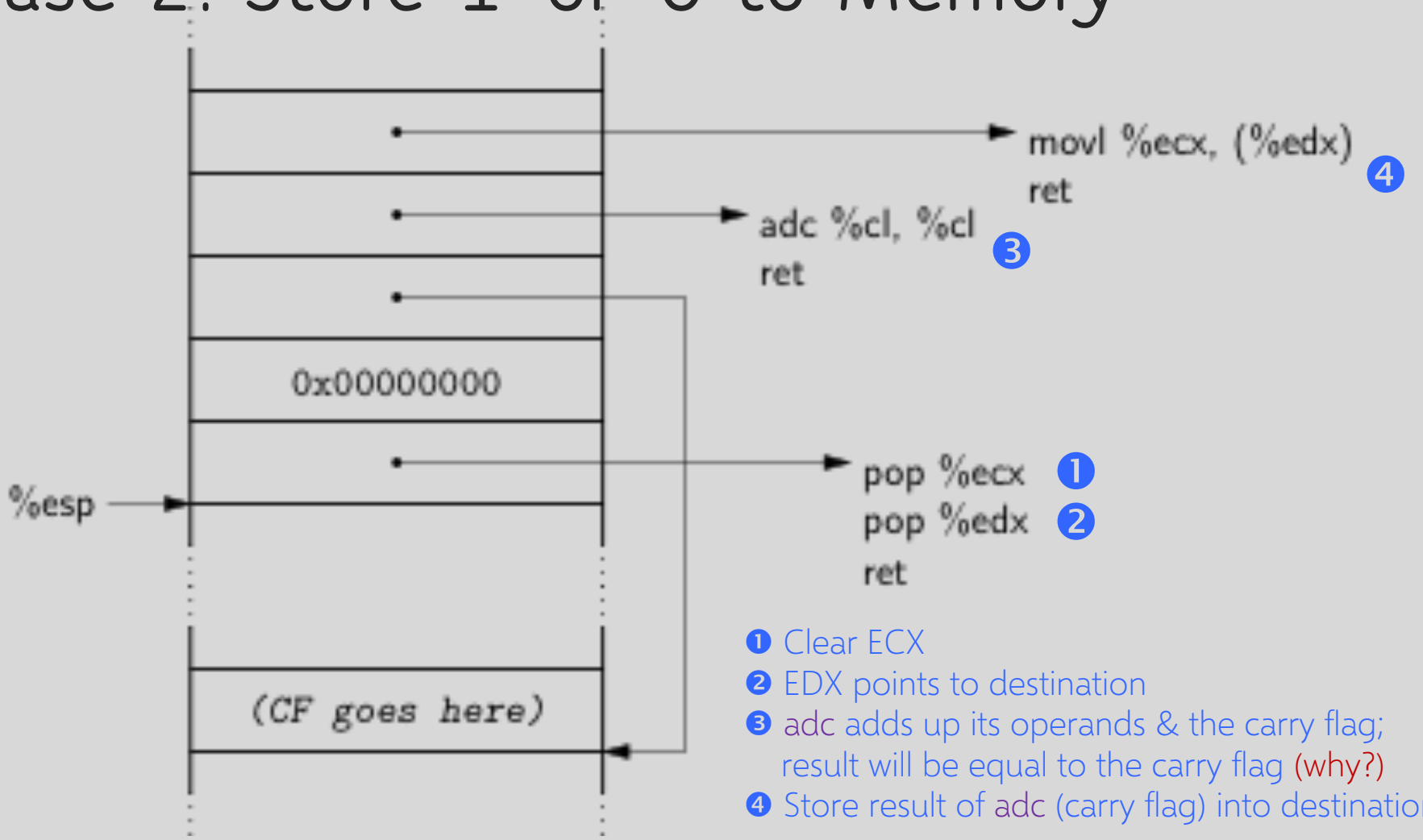
- Need conditional change in stack pointer (ESP)
- Strategy:
  - Move flags to a general-purpose register
  - Compute either delta (if flag is 1) or 0 (if flag is 0)
  - Perturb ESP by the computed delta

Phase 1: Perform Comparison



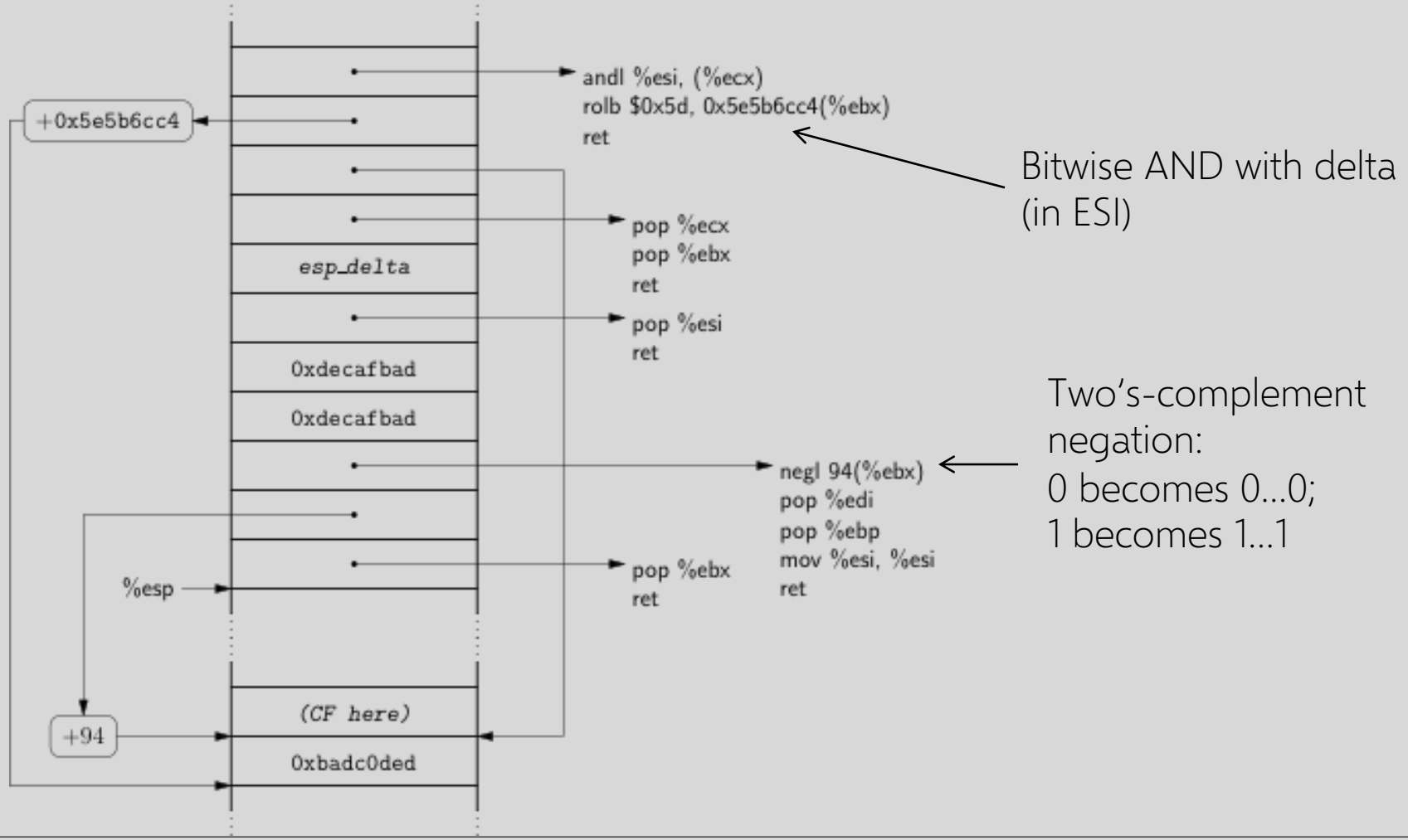
- neg calculates two's complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- sub is similar, use to test if one number is greater than another

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 (carry flag) into destination

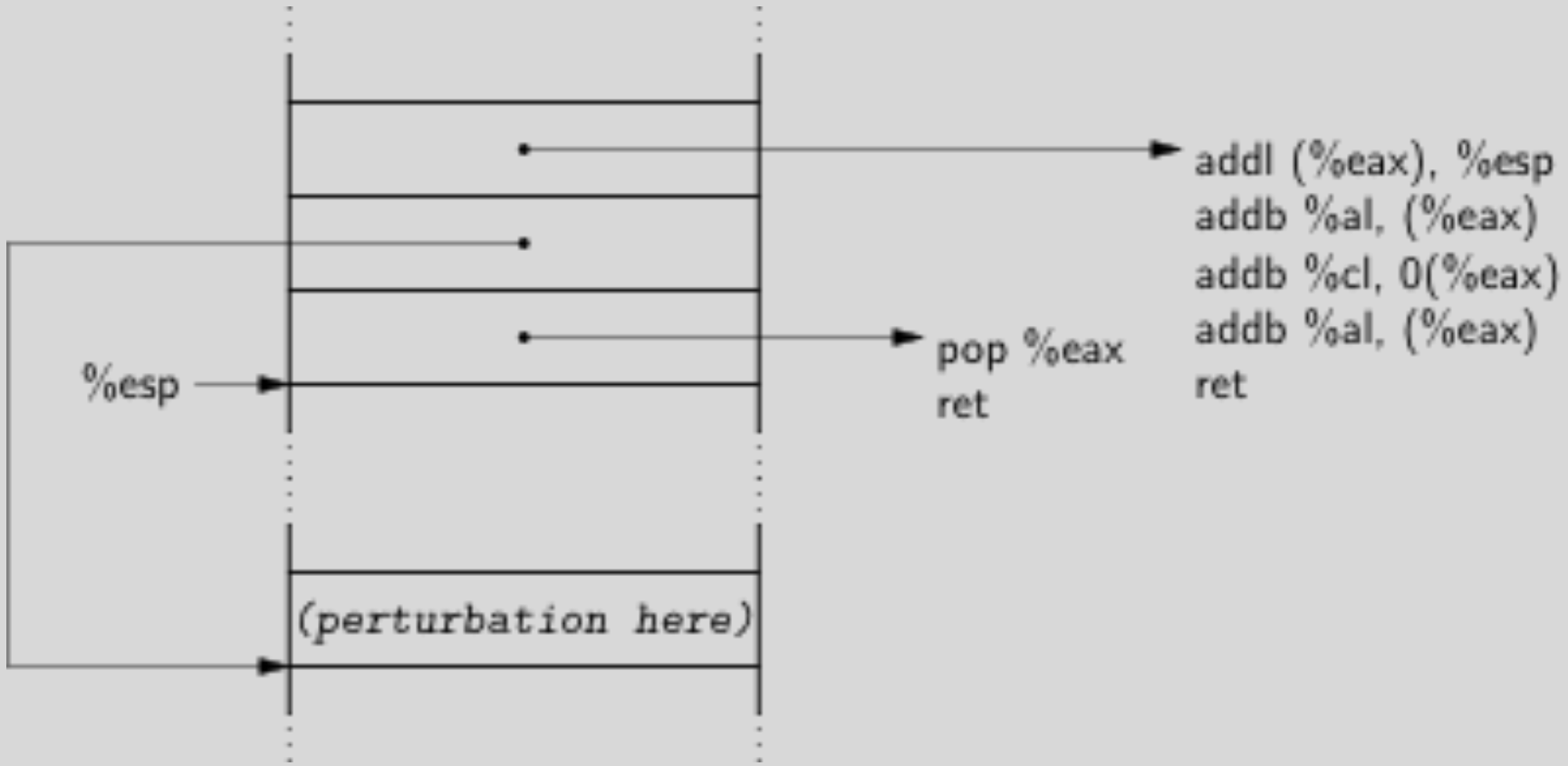
Phase 3: Compute Delta-or-Zero



Bitwise AND with delta (in ESI)

Two's-complement negation:
0 becomes 0…0;
1 becomes 1…1

Phase 4: Perturb ESP by Delta

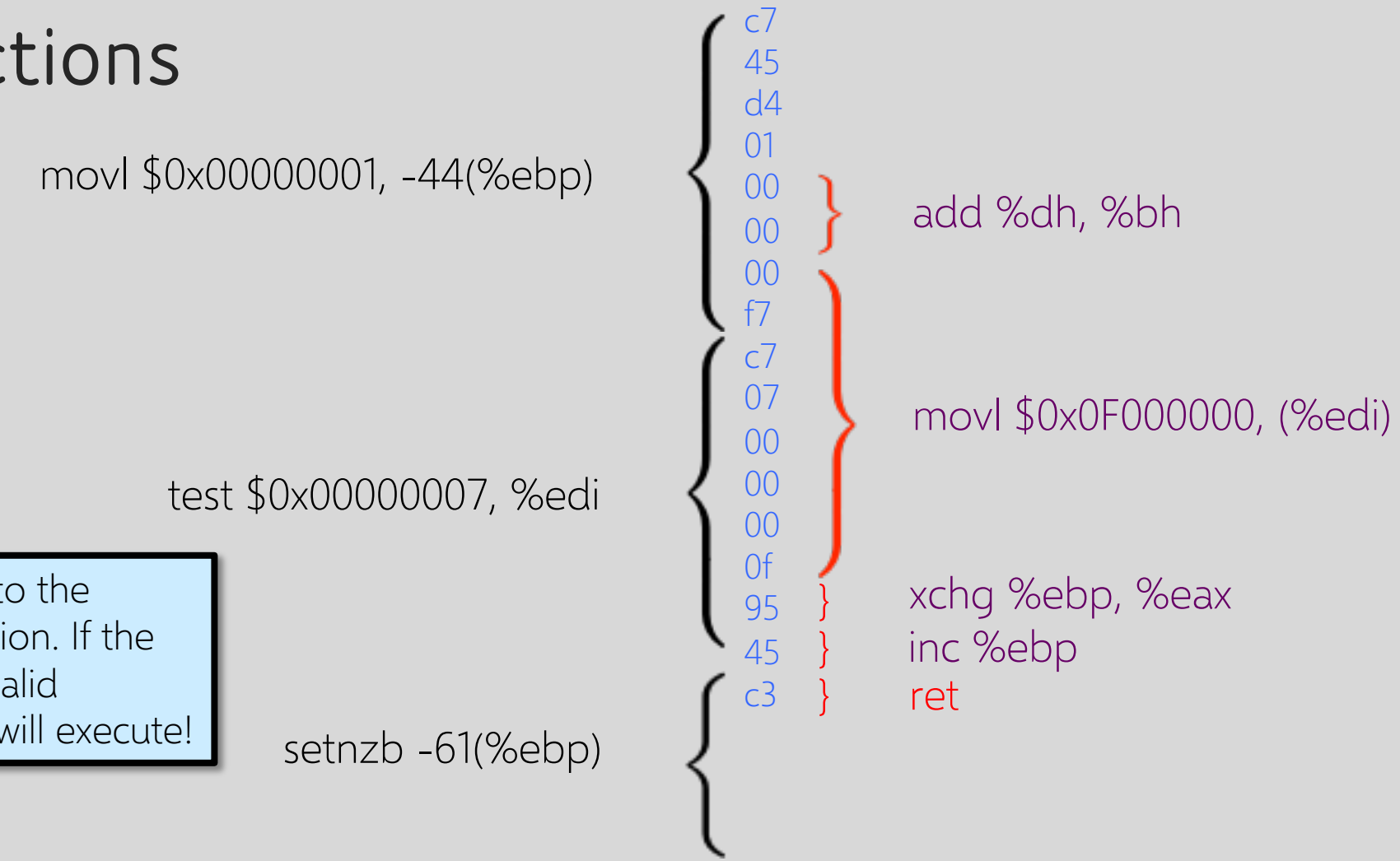


Finding Instruction Sequences

- 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 instruction (C3 byte), look back:
  - Are the preceding i bytes a valid instruction?
  - Recur from found instructions
- Collect found instruction sequences in a trie

Actual code from ecb_crypt()

Unintended Instructions



On x86, can jump into the middle of an instruction. If the following bytes are valid opcodes, processor will execute!

The Joy of x86

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 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!

kBouncer

winner of the 2012 Microsoft BlueHat Prize ($200K)

Observation: abnormal execution sequence violates LIFO call-return order

ret returns to an address that does not follow a **call**

Defense: before a system call, check that every prior ret is not abnormal

kBouncer

winner of the 2012 Microsoft BlueHat Prize ($200K)

Intel's **Last Branch Recording** (LBR):

- Stores 16 last executed branches in a set of on-chip registers (MSR)
- Can read using **rdmsr** instruction from privileged mode

kBouncer: before entering kernel, verify that last 16 **ret**'s are normal

- Requires no application code changes, minimal overhead
- Limitations: attacker can ensure 16 calls prior to syscall are valid

Defeating Other ROP Defenses

"Jump-oriented" programming

- Use update-load-branch sequences instead of returns
  + a trampoline sequence to chain them together

"Return-oriented programming without returns" (2010)

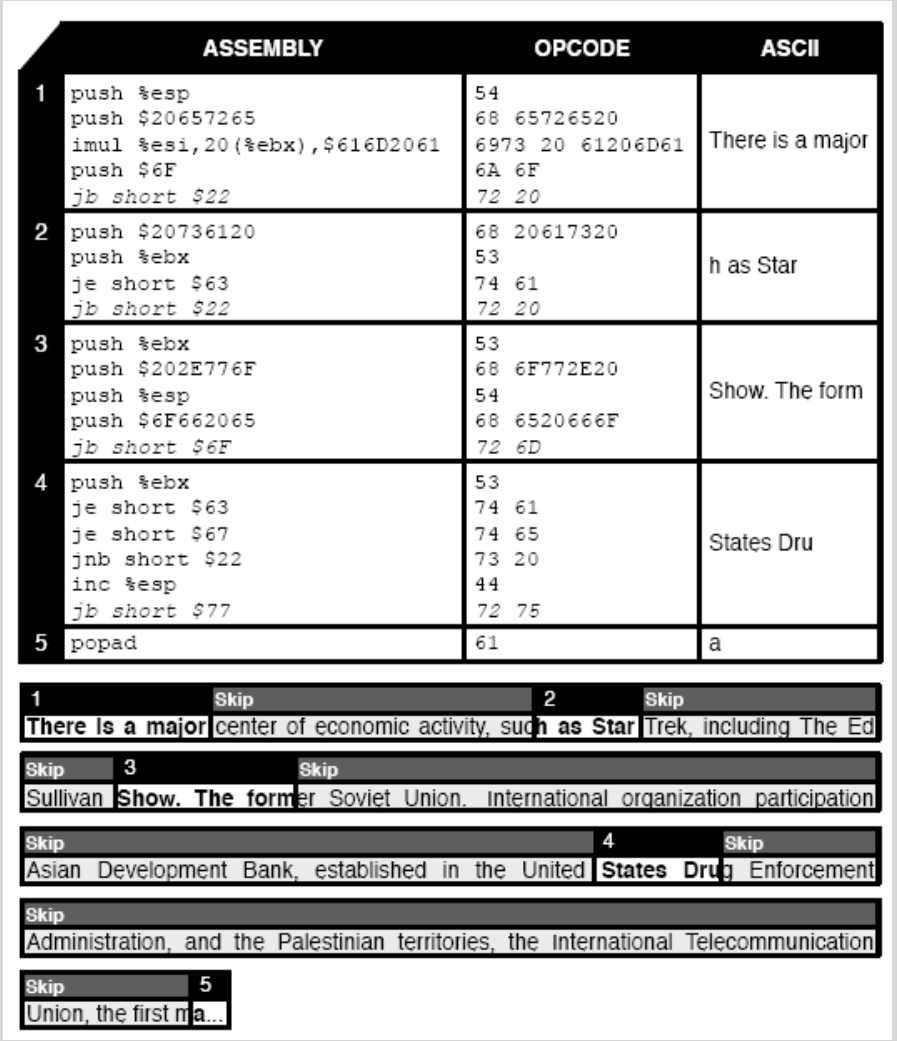
Craft a separate function call stack and call legitimate functions present in the program

Checkoway et al.'s attack on the Sequoia AVC Advantage voting machine

Harvard architecture: code is separate from data, thus code injection is impossible. ROP works fine!

English Shellcode

- Convert any shellcode into an English-looking text
- Encoded payload, decoder uses only a subset of x86 instructions (those whose binary representation corresponds to English ASCII characters)
  - Example: popa – “a”, push %eax – “p”
- Additional processing and padding



	ASSEMBLY	OPCODE	ASCII
1	push %esp push $20657265 imul %esi,20(%ebx),$616D2061 push $6F *jb short $22*	54 68 65726520 6973 20 61206D61 6A 6F *72 20*	There is a major
2	push $20736120 push %ebx je short $63 *jb short $22*	68 20617320 53 74 61 *72 20*	h as Star
3	push %ebx push $202E776F push %esp push $6F662065 *jb short $6F*	53 68 6F772E20 54 68 6520666F *72 6D*	Show. The form
4	push %ebx je short $63 je short $67 jnb short $22 inc %esp *jb short $77*	53 74 61 74 65 73 20 44 *72 75*	States Dru
5	popad	61	a

1 There is a major Skip center of economic activity, such 2 as Star Skip Trek, including The Ed

Skip Sullivan 3 Show. The form Skip er Soviet Union. International organization participation

Skip Asian Development Bank, established in the United 4 States Dru Skip g Enforcement

Skip Administration, and the Palestinian territories, the International Telecommunication

Skip Union, the first m 5 a...

Mason et al. (2009)

CFI: Control-Flow Integrity

Main idea: self-protecting code

Pre-determine control flow graph (CFG) of an application

- Static analysis of source code
- Static binary analysis ← CFI
- Execution profiling
- Explicit specification of security policy

Insert checks to ensure that execution follows the pre-determined control flow graph

Abadi et al. "Control-Flow Integrity" (2005)

CFI via Binary Instrumentation

- Use binary rewriting to instrument code with runtime checks
- Inserted checks ensure that whenever an instruction transfers control, destination is valid according to the CFG
  - Therefore, the execution always stays within the statically determined CFG

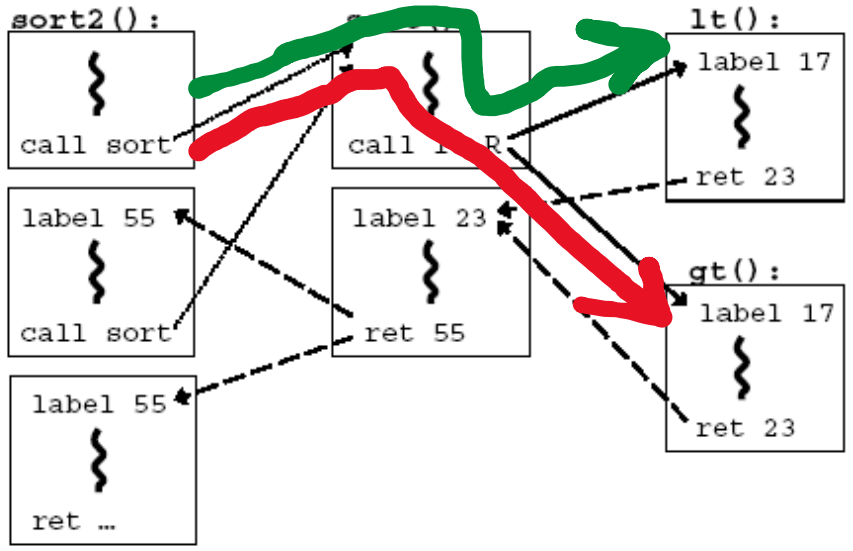
Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc) even if the attacker has complete control over the thread's address space

CFG Example

```
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 );
}
```



This CFG permits more executions than are actually possible (why?)

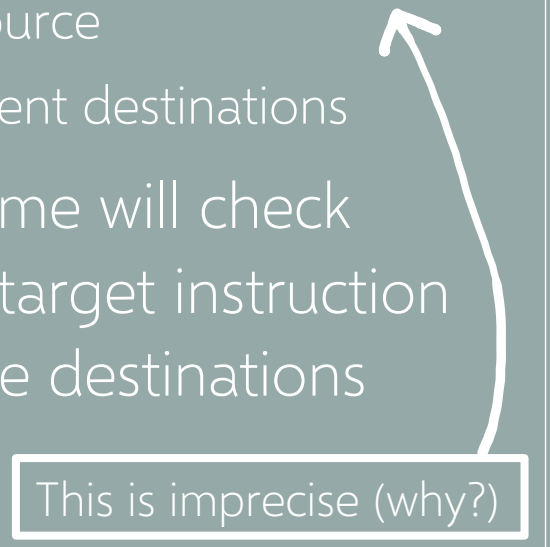
Control Flow Enforcement in CFI

For each control transfer, determine statically its possible destination(s)

Insert a unique bit pattern at every destination

- Two destinations are equivalent if static CFG contains edges to each from the same source
- Use same bit pattern for equivalent destinations

Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations



CFI: Example of Instrumentation

Original code

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	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

Preventing Circumvention

What about run-time code generation and self-modification?

Unique IDs

- Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks

Non-writable code

- Program should not modify code memory at runtime

Non-executable data

- Program should not execute data as if it were code

Enforcement: hardware support + prohibit system calls that change protection state + verification at load-time

Improving CFI Precision

Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)

- CFI will use the same tag for C and D, but this permits an invalid call from A to D
- Possible solutions: (1) duplicate code or inline; (2) multiple tags

Function F is called first from A, then from B. What's a valid destination for its return?

- CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
- Solution: shadow call stack

CFI Security Guarantees

Effective against attacks based on illegitimate control-flow transfer

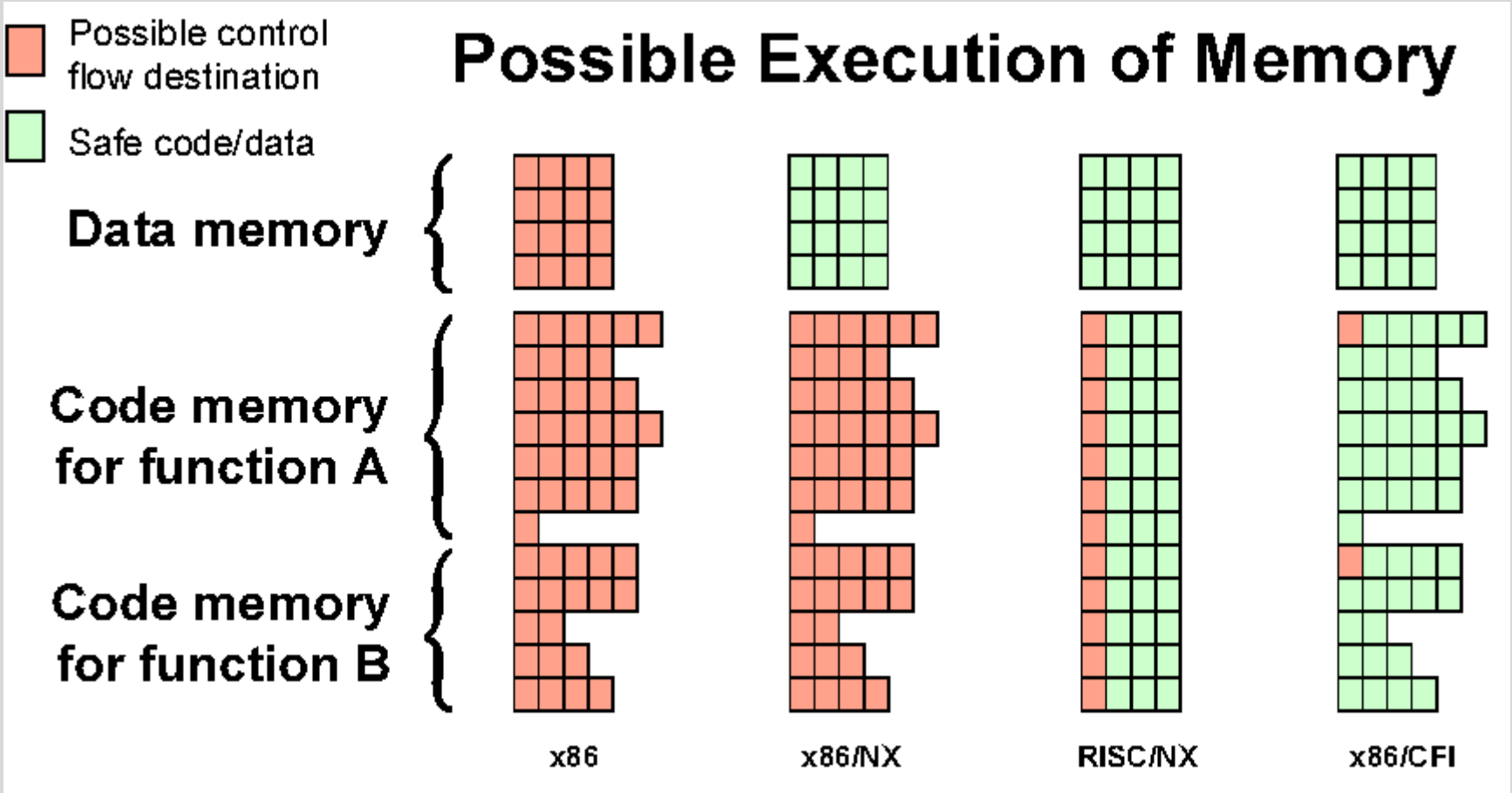
- Stack-based buffer overflow, return-to-libc exploits, function pointer overwrite

Does not protect against attacks that do not violate the program's original CFG

- Incorrect arguments to system calls
- Substitution of file names
- Other data-only attacks

Credit: Ulfar Erlingsson

Possible Execution of Memory



Poor man's CFI, sort of…

Control-Flow Guard (CFG) in Windows 10

Protects indirect calls by checking against a bitmask of **all valid function entry points** in the executable

ensures the target is the entry point of a function

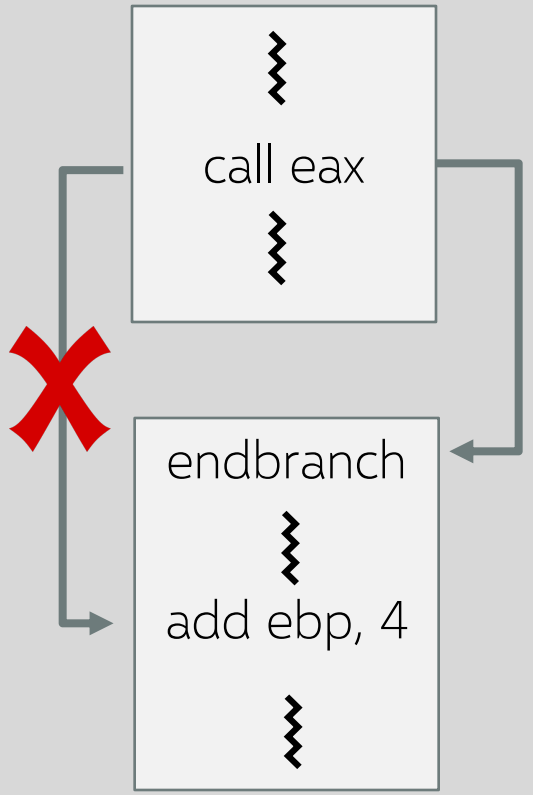
- Does not prevent attacker from causing a jump to a valid wrong function
- Hard to build accurate control flow graph statically

Deployed in Intel Tiger Lake (2020)

Intel's CET

New **EndBranch** (ENDBR64) instruction

- Compiler inserts **EndBranch** at all valid destinations of control transfers
- After an indirect **JMP** or **CALL**, the next instruction in the instruction stream must be **EndBranch**
- If not, then trigger a #CP fault and halt execution



- Does not prevent attacker from causing a jump to a valid wrong function
- Hard to build accurate control flow graph statically

Bypassing CFG and CET

```
void HandshakeHandler(Session *s, char *pkt) {
    s->hdlr = &LoginHandler;
    ... Buffer overflow over Session struct ...
}

void LoginHandler(Session *s, char *pkt) {
    bool auth = CheckCredentials(pkt);
    s->dhandler = &DataHandler;
}

void DataHandler(Session *s, char *pkt);
```

Attacker overwrites function pointer to call **DataHandler**, bypassing authentication

Call to **DataHandler** is permitted by the static control-flow graph

ARM Memory Tagging Extension (MTE)

Every 64-bit **memory pointer** P has a 4-bit "tag" in top byte

Every 16-byte user **memory region** R has a 4-bit "tag"

New hardware instructions
LDG, STG: load and store tag to memory region (used by malloc and free)
ADDG, SUBG: pointer arithmetic on an address preserving tags

Processor ensures:
if P is used to read R then tags are equal,
otherwise hardware exception

This prevents overflow and use-after-free

Note: does not prevent inter-object overflow in C++ (why?)

also ARM pointer authentication

Cryptographic CFI (CCFI)

Every time a jump address is written/copied anywhere in memory, compute 64-bit AES-MAC and append to address

On heap: tag = AES(k, (jump-address, 0 ll source-address))

On stack: tag = AES(k, (jump-address, 1 ll stack-frame))

Before control transfer, verify AES-MAC and crash if invalid

Where to store key k? In xmm registers (not memory)