

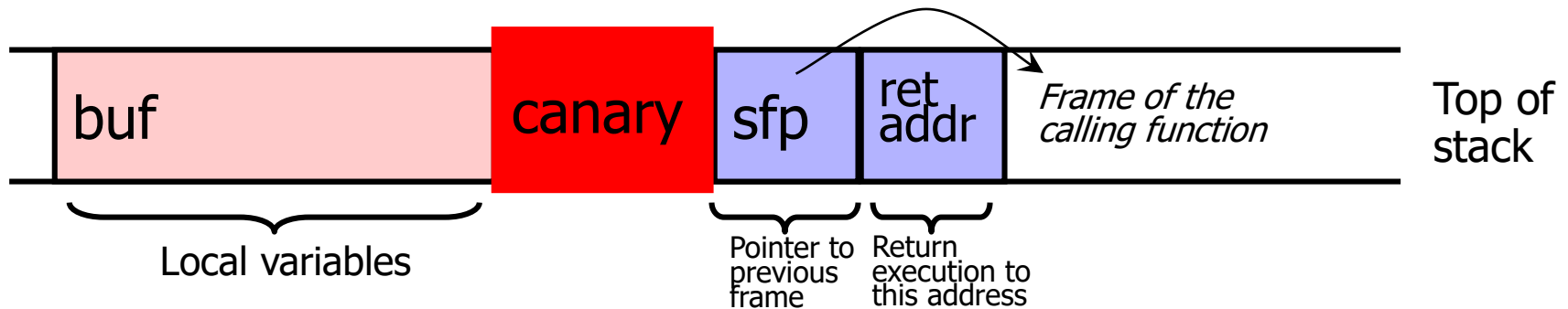
Defenses against Memory Attacks

Vitaly Shmatikov

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



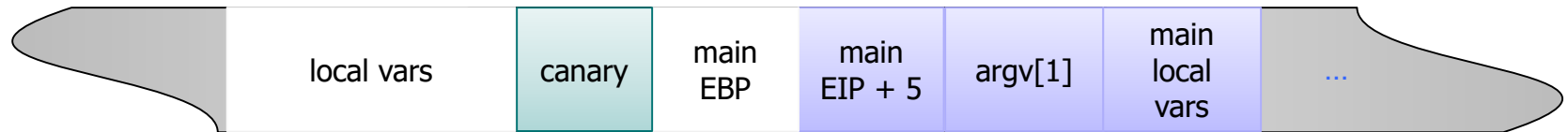
Choose random canary string on program start

- 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”

Modern Stack Canaries



Low memory
addresses

High memory
addresses

Contemporary gcc implementation

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

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

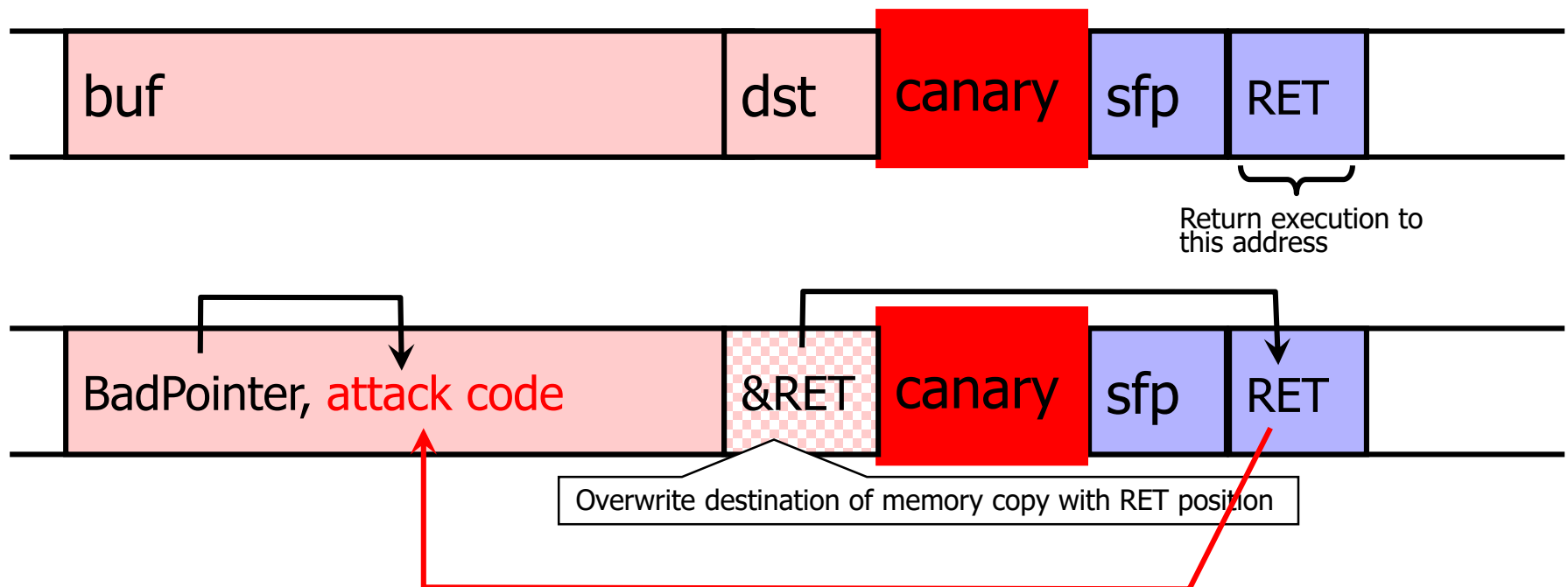
StackGuard can be defeated

- 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



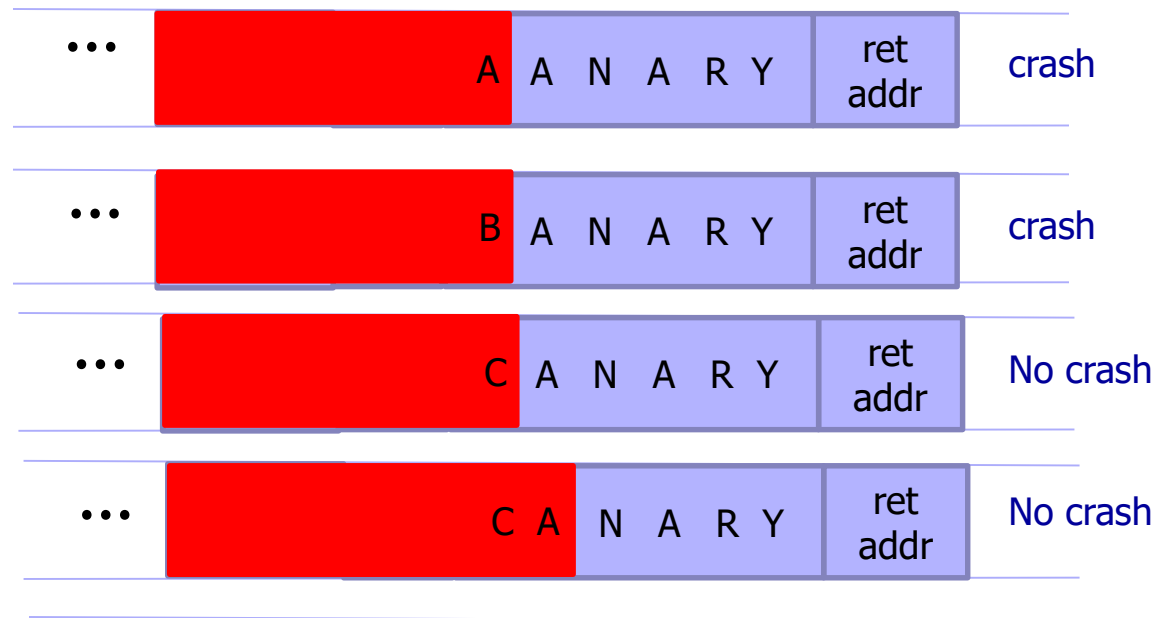
"Reading" the Stack for Canary

A common design for crash recovery:

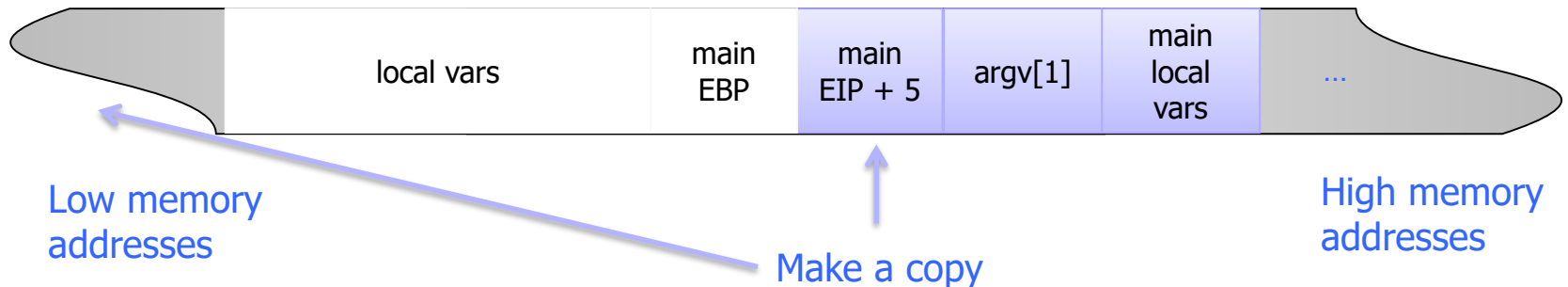
When process crashes, restart automatically (for availability)

If relaunched using fork, canary is unchanged

Attacker can
extract canary
byte by byte



StackShield



StackShield:

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

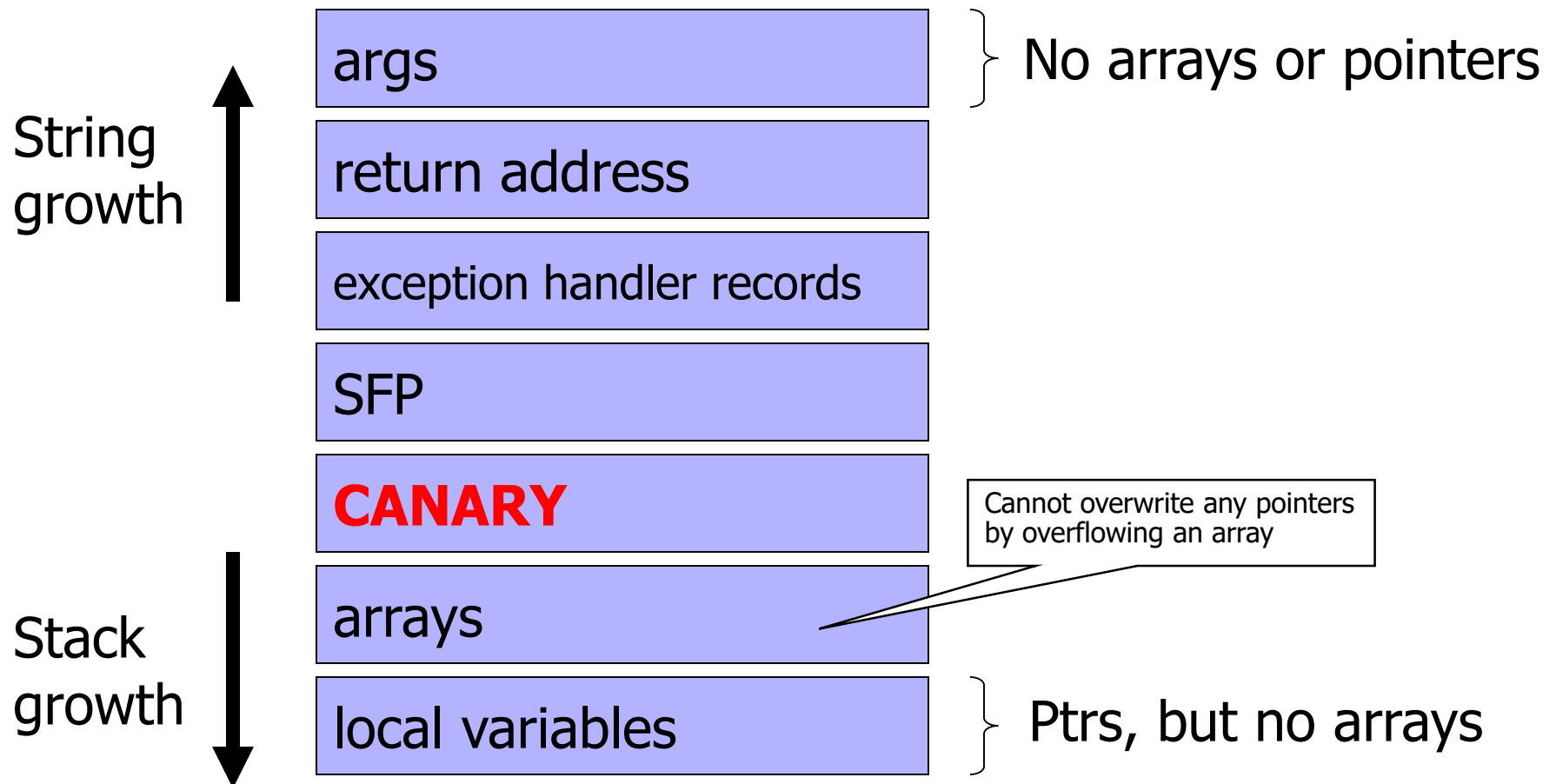
Circumvention:

- Write over saved copy & return address, if possible
- Hijack control flow without return address

ProPolice / SSP

Canary [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

~~CONFIDENTIAL~~ [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; write 1 here  
    ...  
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);  
}
```

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

GHTTPD Web Server

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
```

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

... overflown
... and read from stack

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

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 base address of stack, heap, code segment, location of 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

Base-Address Randomization

Only the base address is randomized

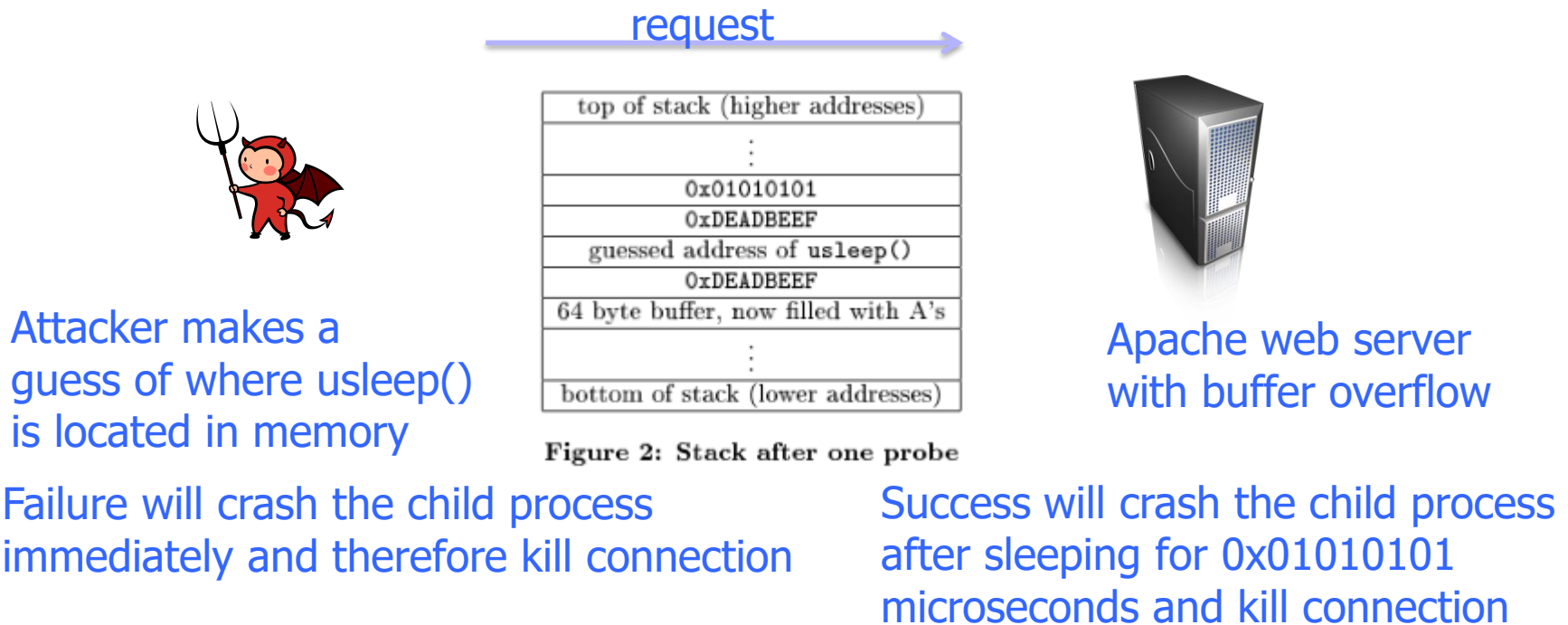
- **Layouts** of stack and library table remain the same
- Relative distances between memory objects are not changed by base address randomization

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 addr of known library function – how long does it take?
 - In “On the effectiveness of address-space randomization” (CCS 2004), Shacham et al. used `usleep()` for attack (why?)
- If address is wrong, target will simply crash

Brute-Force Guessing



If on 64-bit architecture, such brute-force attack unlikely to work

ASLR in Windows

Vista and Server 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

Example: ASLR in Windows Vista

Booting Vista twice loads libraries into different locations:

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

ASLR is only applied to images for which
the **dynamic-relocation** flag is set

Defeating ASLR (1)

Large NOP sled with classic buffer overflow

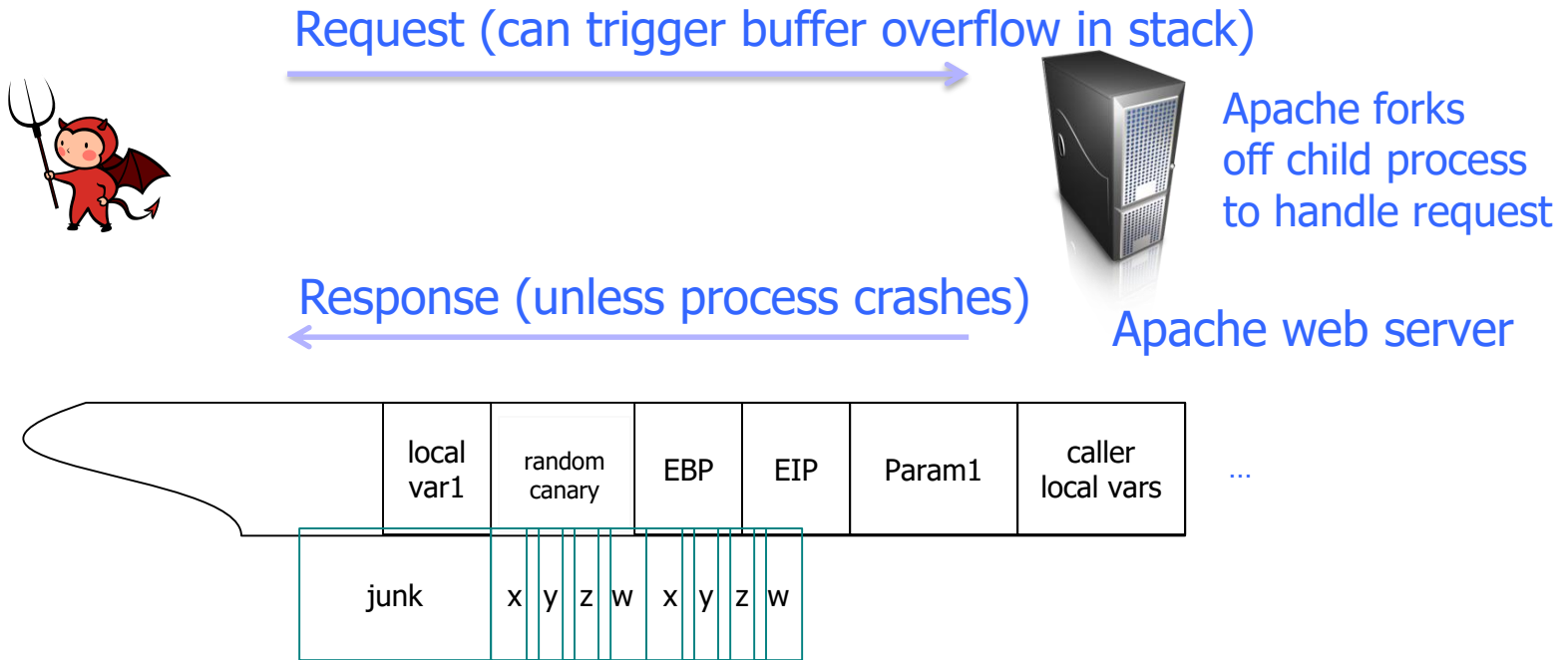
- Doesn't work with W^X

Use a vulnerability that can be used to leak address information

- E.g., printf arbitrary read

Brute force the address using forking process

Reading the Stack, Remotely



Reading stack for EBP/EIP can give approximate address offset

Defeating ASLR (2)

Implementation uses randomness improperly, thus distribution of heap bases is biased

- Ollie Whitehouse, Black Hat 2007
- Makes guessing a valid heap address easier

When attacking browsers, may be able to insert arbitrary objects into the victim's heap

- Executable JavaScript code, plugins, Flash, Java applets, ActiveX and .NET controls...

Heap spraying

- Stuff heap with multiple copies of attack code

Problem?

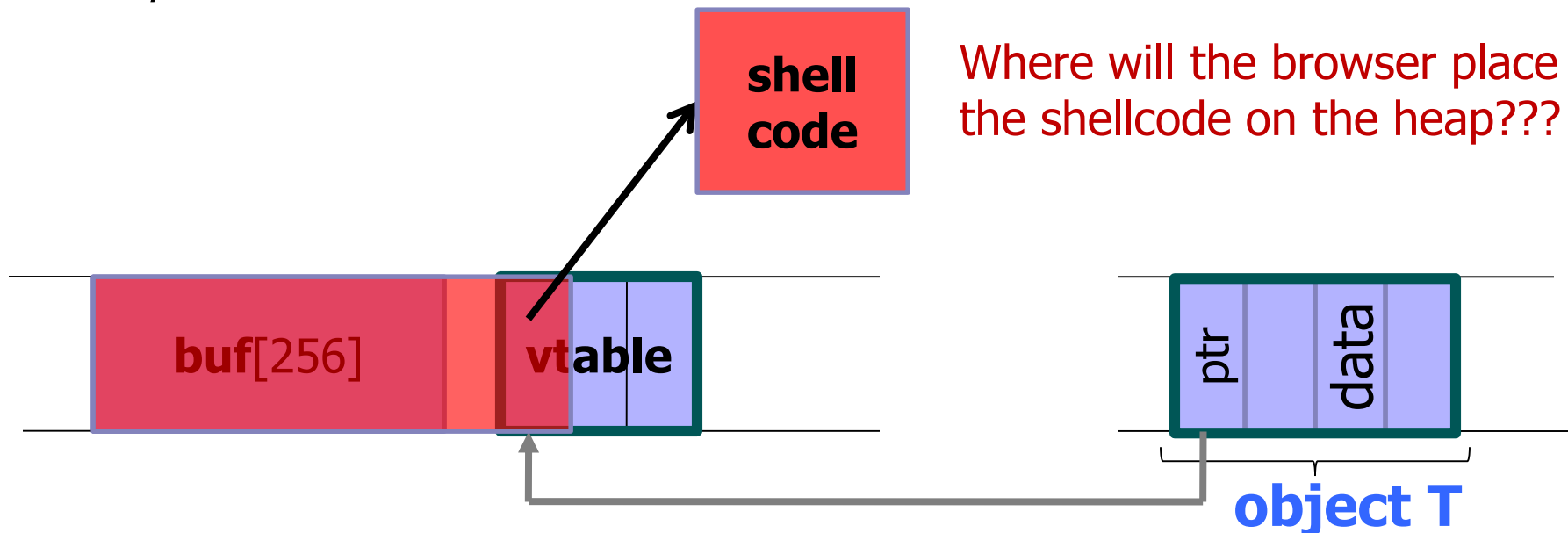
```
<SCRIPT language="text/javascript">
```

```
  shellcode = unescape("%u4343%u4343%...");
```

```
  overflow-string = unescape("%u2332%u4276%...");
```

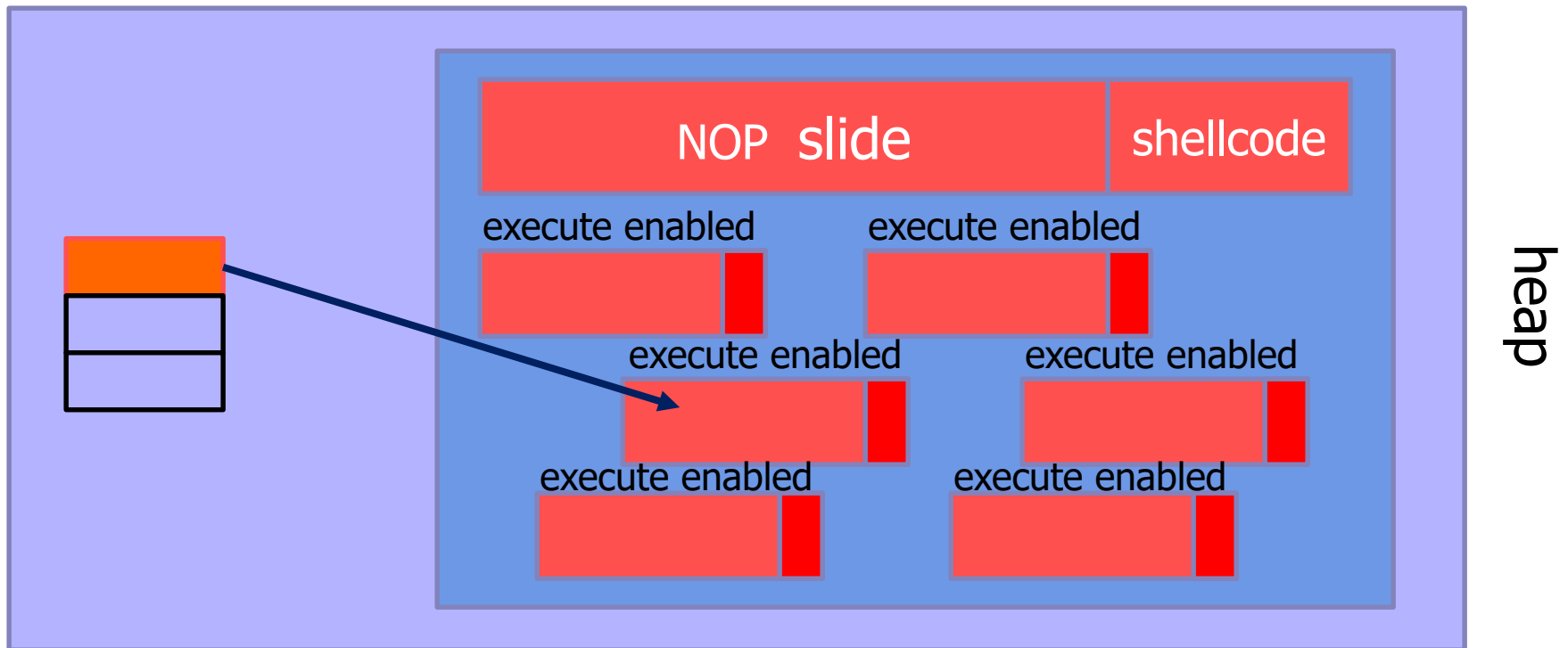
```
  cause-overflow( overflow-string );      // overflow buf[ ]
```

```
</SCRIPT>
```



Heap Spraying

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%...");
```

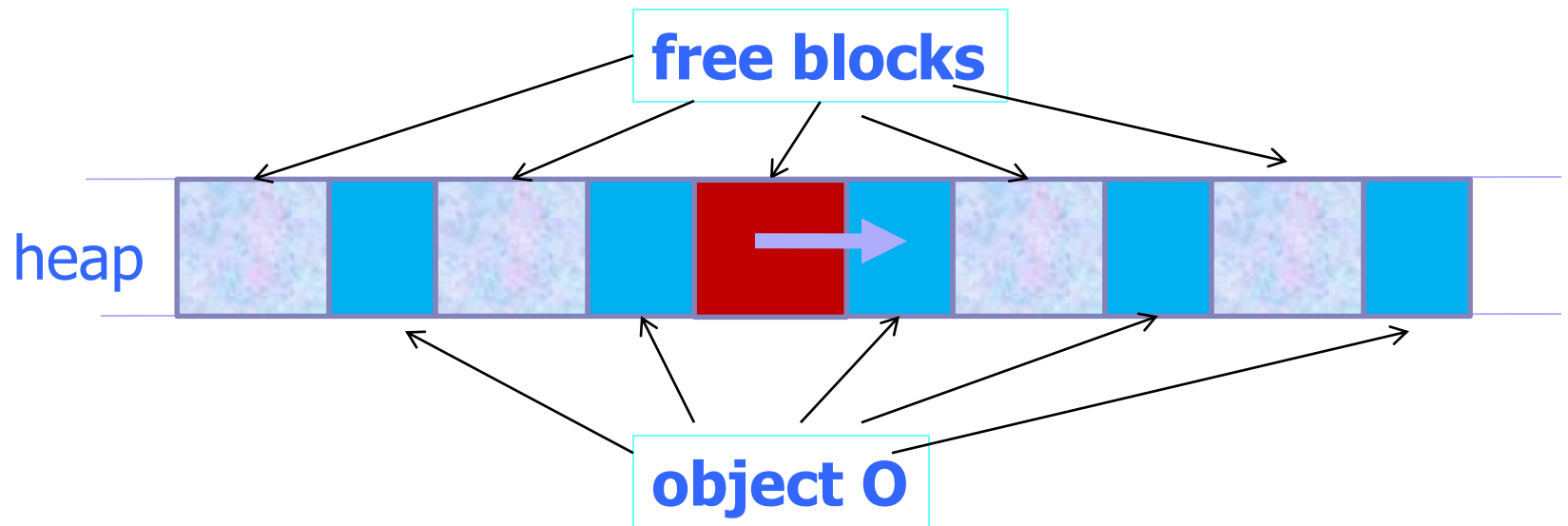
```
var x = new Array ()  
for (i=0; i<1000; i++) {  
    x[i] = nop + shellcode;  
}
```

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

Placing Vulnerable Buffer

[Safari PCRE exploit, 2008]

Use a sequence of JavaScript allocations and free's to make the heap look like this:



Allocate vulnerable buffer in JavaScript and cause overflow

Heap Spraying in EternalBlue

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

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.

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 \oplus 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 $W\oplus X$ / 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 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 writable

What Does $W\oplus X$ 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, $W\oplus X$ 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**

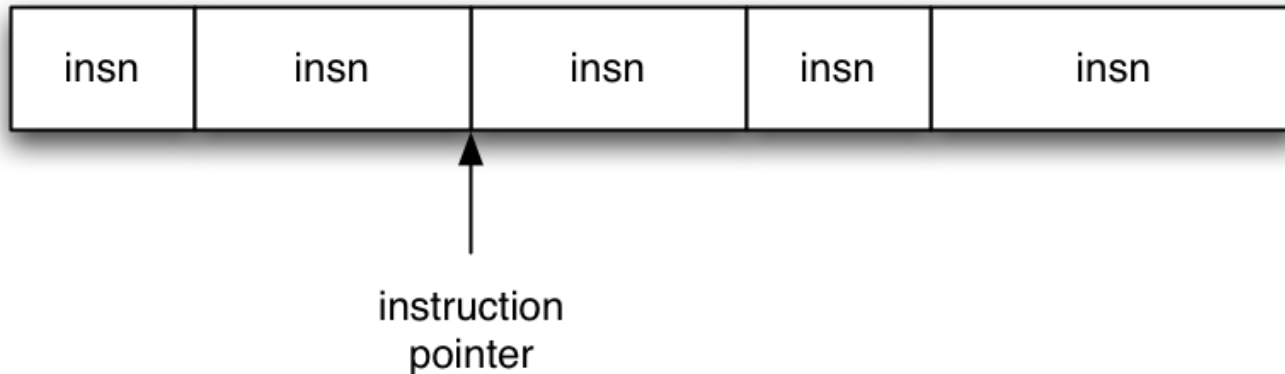
- Turing-complete language
- Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
- Attack can perform arbitrary computation using no injected code at all!



Return-Oriented Programming

is A lot like a ransom
note, BUT instead of cutting
cut letters from magazines,
YOU ARE cutting out
instructions from text
segments

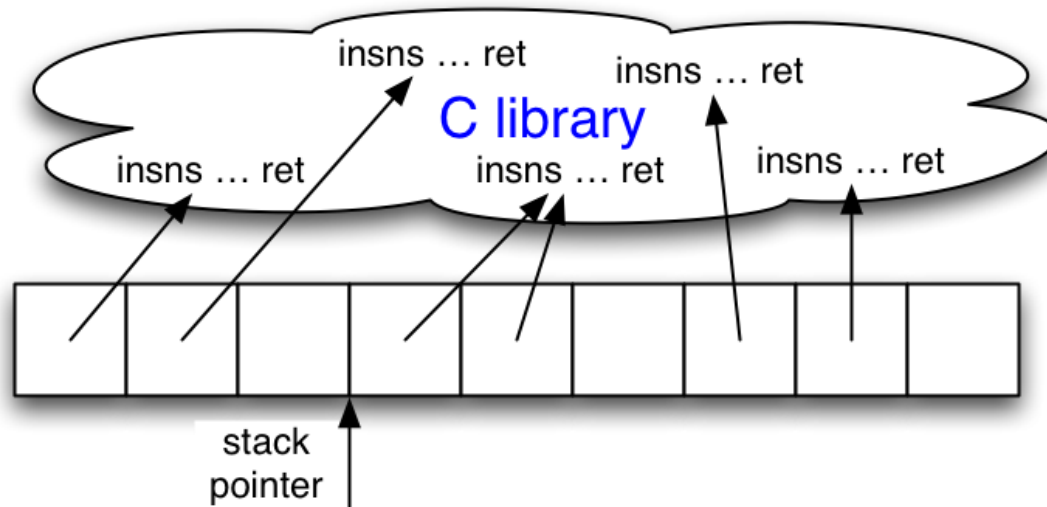
Ordinary Programming



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

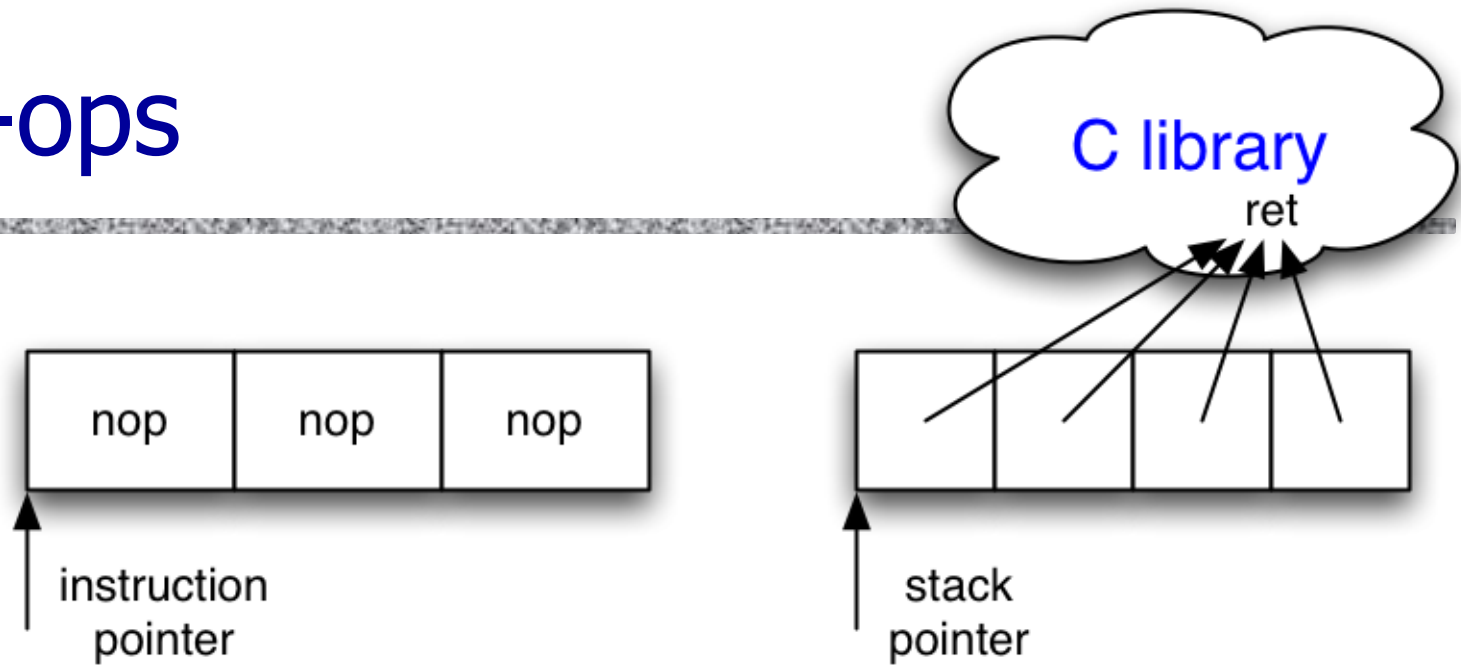


Stack pointer (ESP) determines which instruction sequence to fetch and execute

Processor doesn't automatically increment ESP

- But the RET at end of each instruction sequence does

No-ops

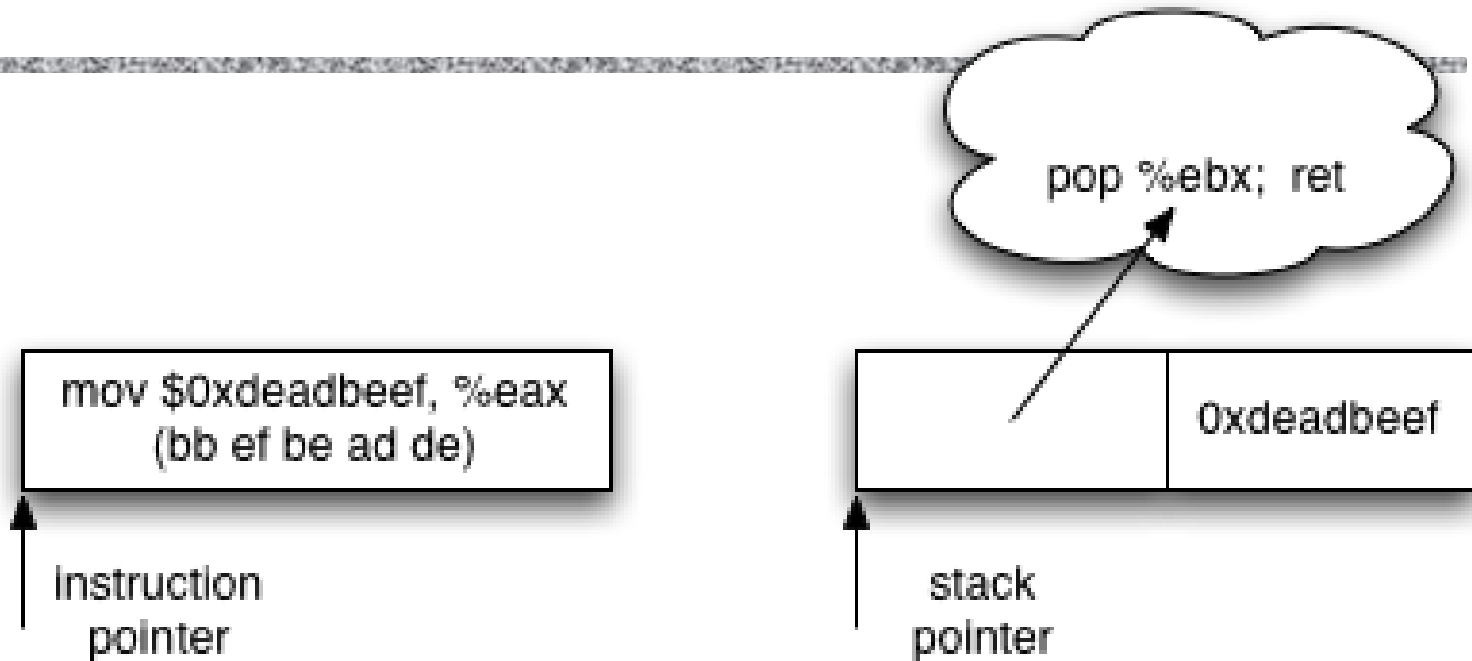


No-op instruction does nothing but advance EIP
Return-oriented equivalent

- Point to return instruction
- Advances ESP

Useful in a NOP sled (what's that?)

Immediate Constants

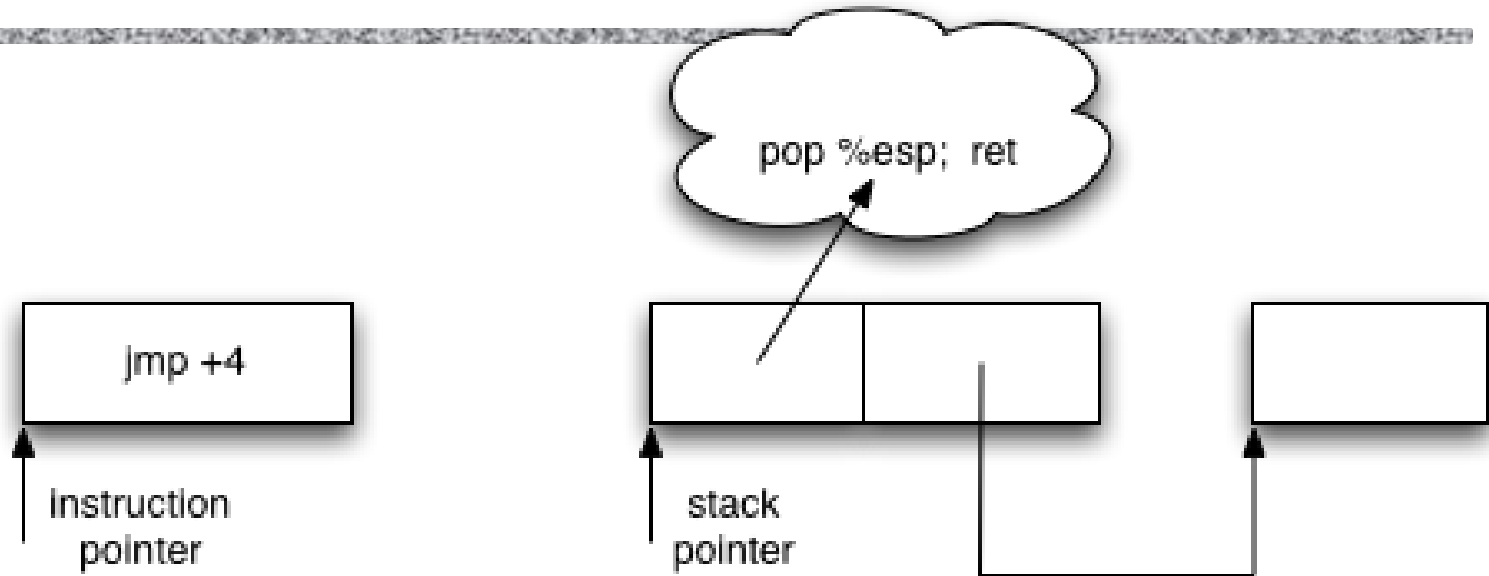


Instructions can encode constants

Return-oriented equivalent

- Store on the stack
- Pop into register to use

Control Flow



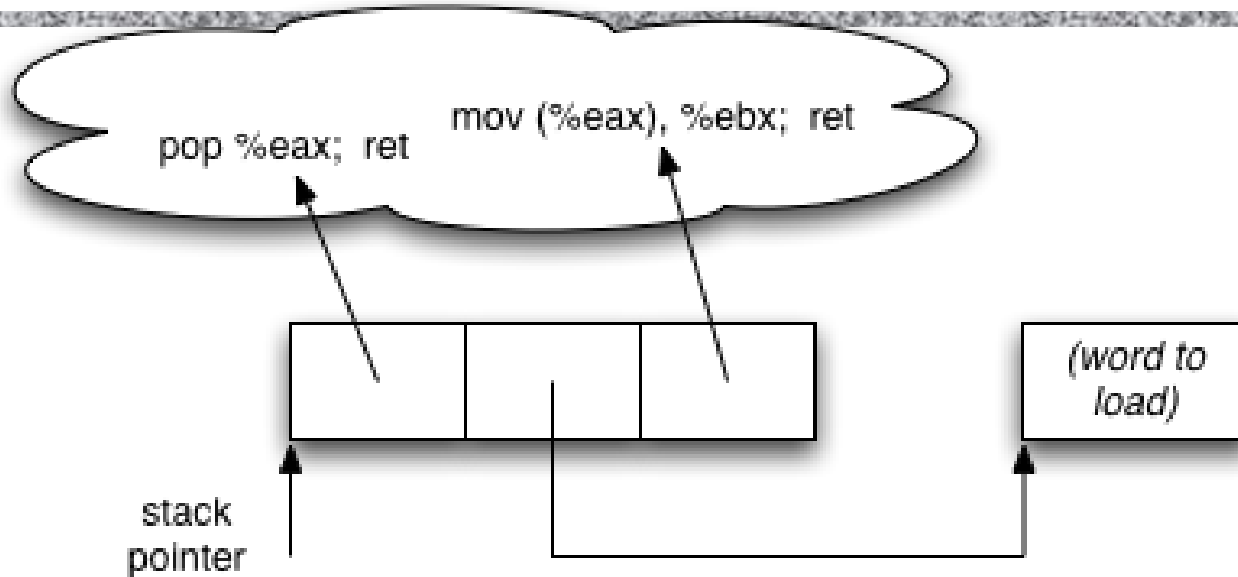
Ordinary programming

- (Conditionally) set EIP to new value

Return-oriented equivalent

- (Conditionally) set ESP to new value

Gadgets: Multi-instruction Sequences

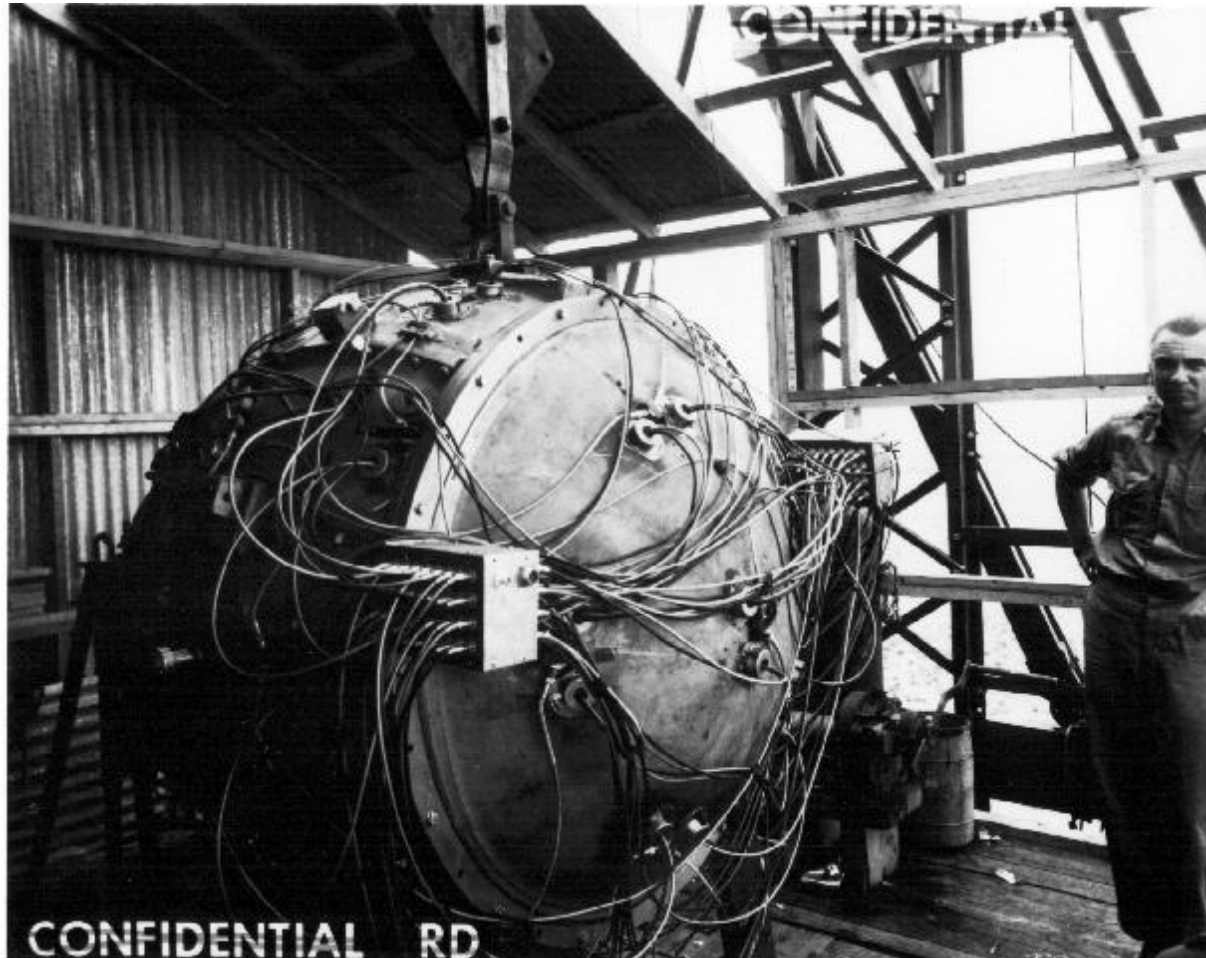


Sometimes more than one instruction sequence needed to encode logical unit

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

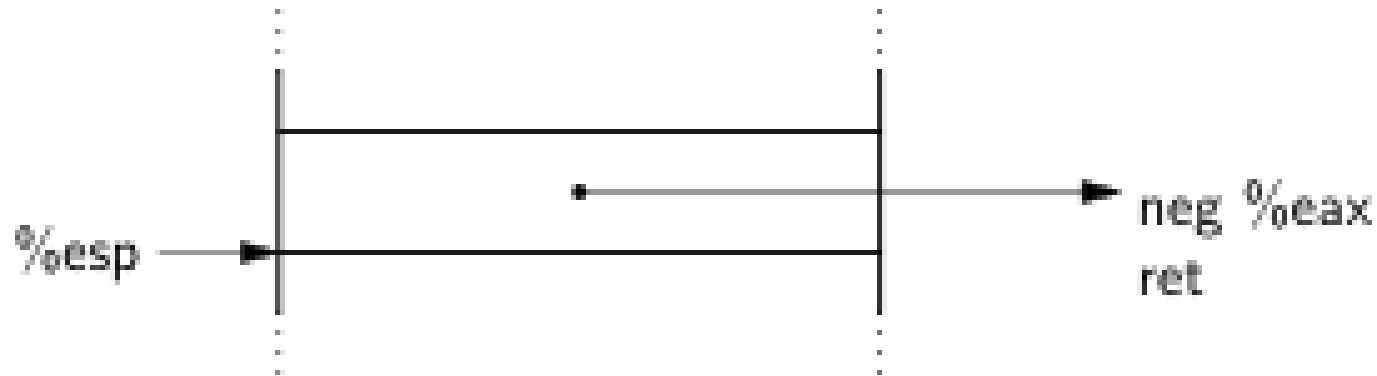
- But this causes a change in EIP... not useful (why?)

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



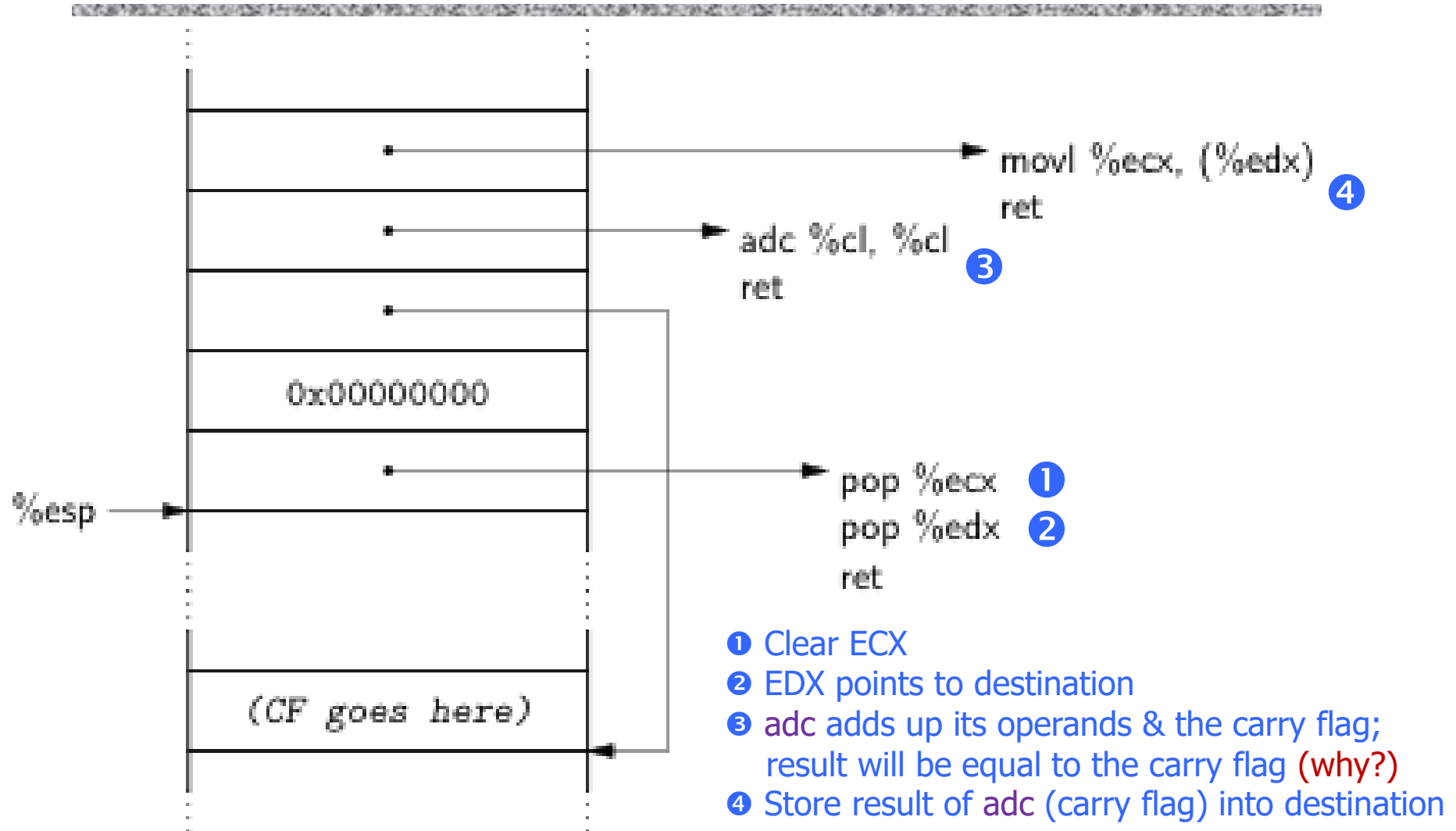
`neg` calculates two's complement

- As a side effect, sets carry flag (CF) if the argument is nonzero

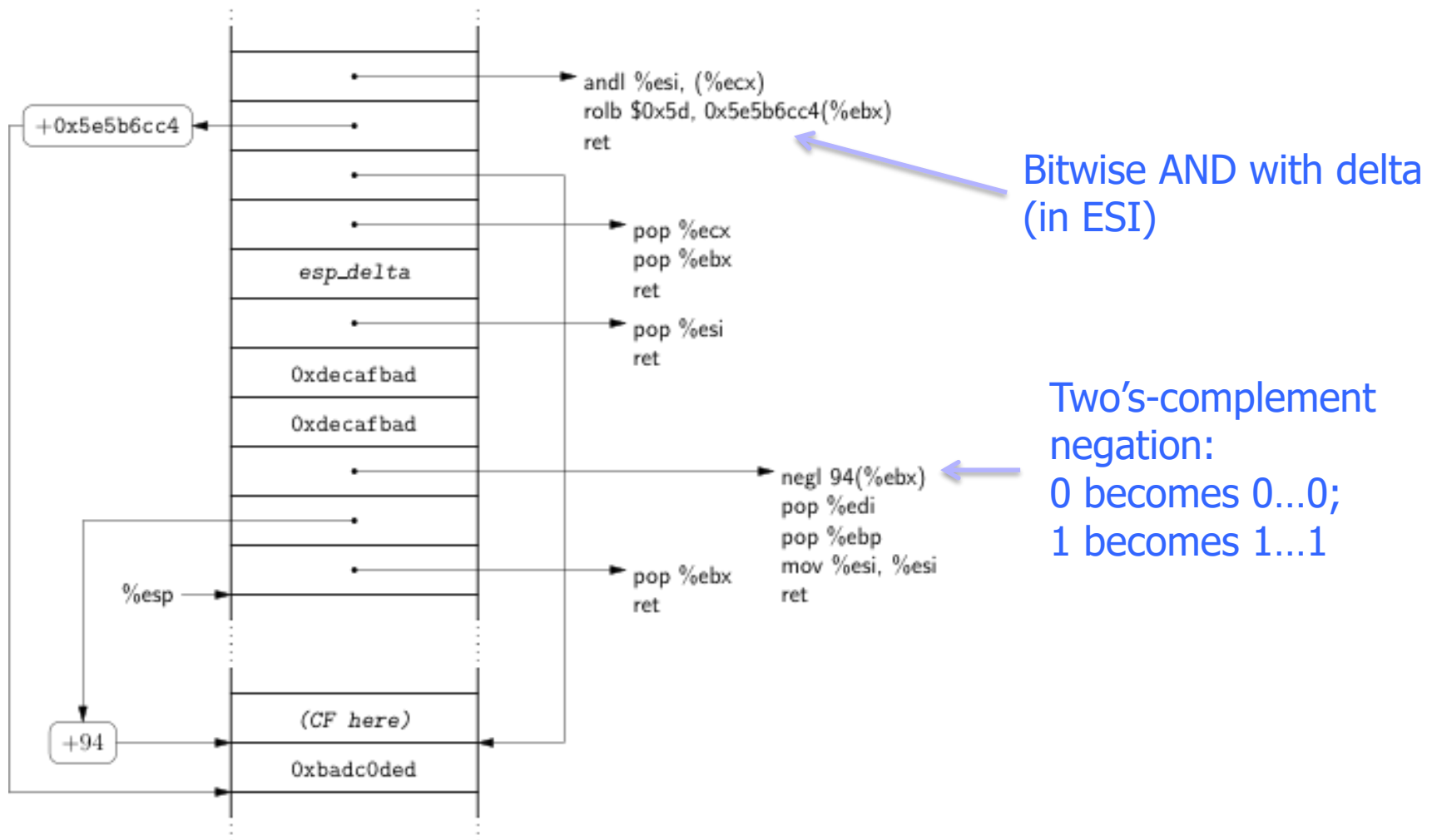
Use this to test for equality

`sub` is similar, use to test if one number is greater than another

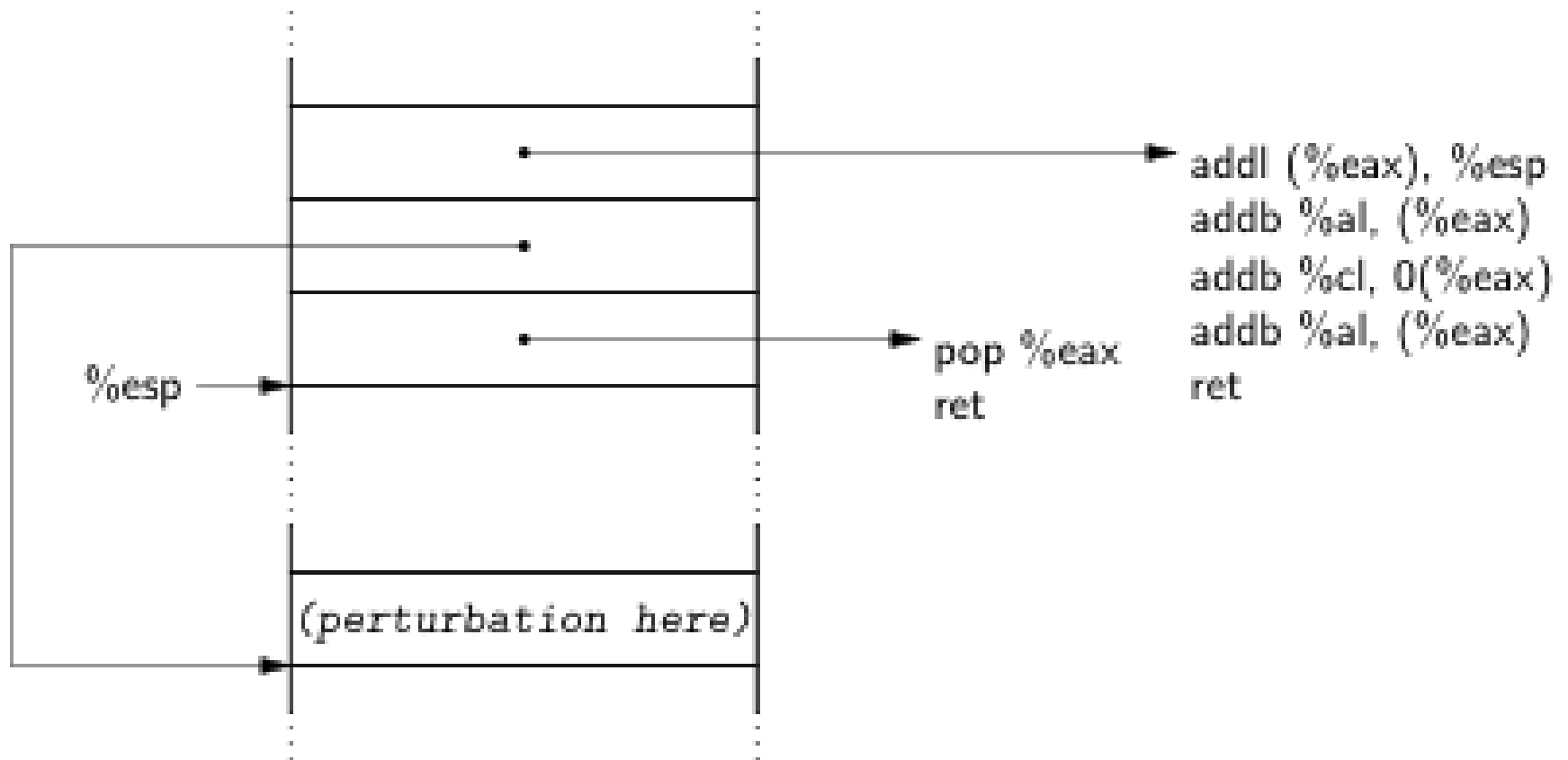
Phase 2: Store 1-or-0 to Memory



Phase 3: Compute Delta-or-Zero



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()



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 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!

- “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC” (CCS 2008)

kBouncer

winner of 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**

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

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

kBouncer



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

- store 16 last executed branches in a set of on-chip registers (MSR)
- 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 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)

Unintended Instructions Redux

English shellcode - Mason et al. (CCS 2009)

- 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 to make combinations of characters look like English text

English Shellcode: Example

[Mason et al., CCS 2009]

	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	Skip	2	Skip
There is a major center of economic activity, such as Star Trek, including The Ed			
Skip	3	Skip	
Sullivan Show. The former Soviet Union. International organization participation			
Skip		4	Skip
Asian Development Bank, established in the United States Drug Enforcement			
Skip			
Administration, and the Palestinian territories, the International Telecommunication			
Skip	5		
Union, the first ma...			

In-Place Code Randomization

[Pappas et al., Oakland 2012]

Instruction reordering

```
MOV EAX, &p1  
MOV EBX, &p2
```



```
MOV EBX, &p2  
MOV EAX, &p1
```

Instruction substitution

```
MOV EBX, $0
```



```
XOR EBX, EBX
```

Register re-allocation

```
MOV EAX, &p  
CALL *EAX
```

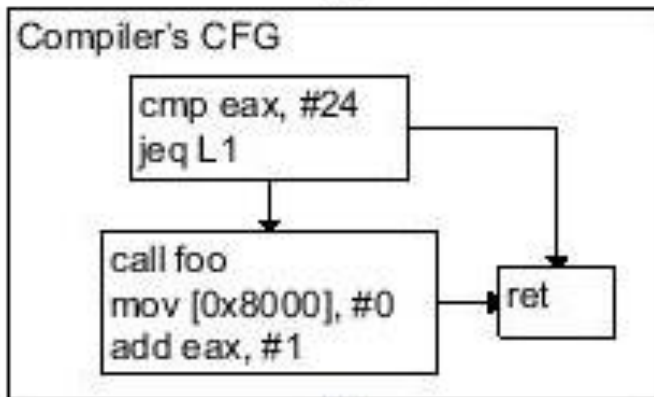


```
MOV EBX, &p  
CALL *EBX
```

Instruction Location Randomization

[Hiser et al., Oakland 2012]

Traditional Program Creation



7000	cmp eax, #24
7001	jeq 7005
7002	call 7500
7003	mov [0x8000], #0
7004	add eax, #1
7005	ret

ILR-protected Program

Fallthrough Map:

39bd->d27e
d27f->cb20
cb21->67f3
67f4->224a
224b->a96b

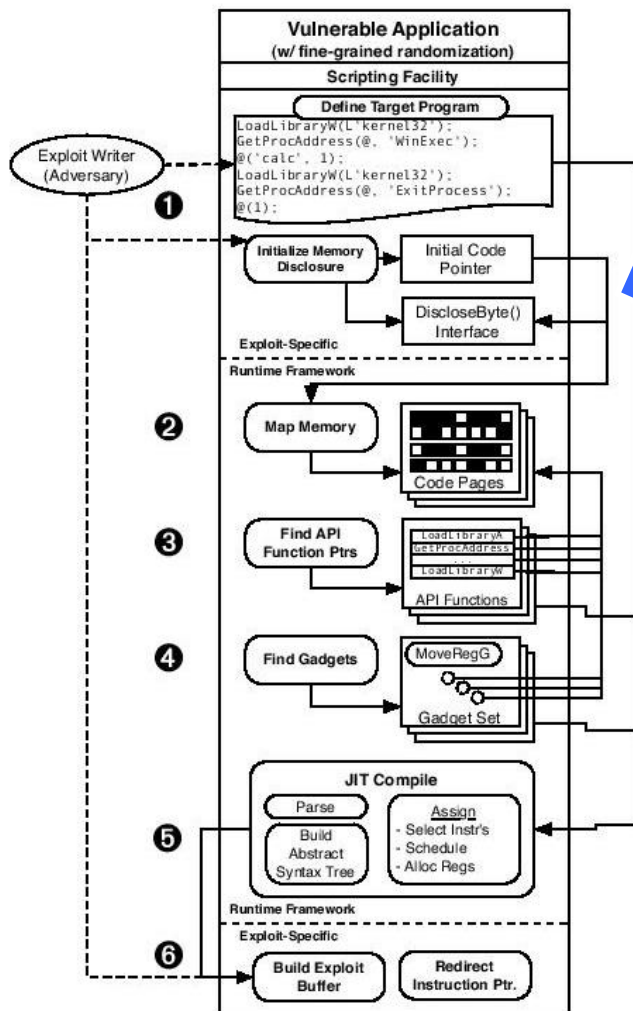
224a	add eax, #1
39bc	cmp eax, #24
67f3	mov [0x8000], #0
a96b	ret
cb20	call 5f32
d27e	jeq a96b

Every instruction is in a random location and has an explicit successor

ROP solved?

Just-in-Time Code Reuse (1)

[Snow et al., Oakland 2013]



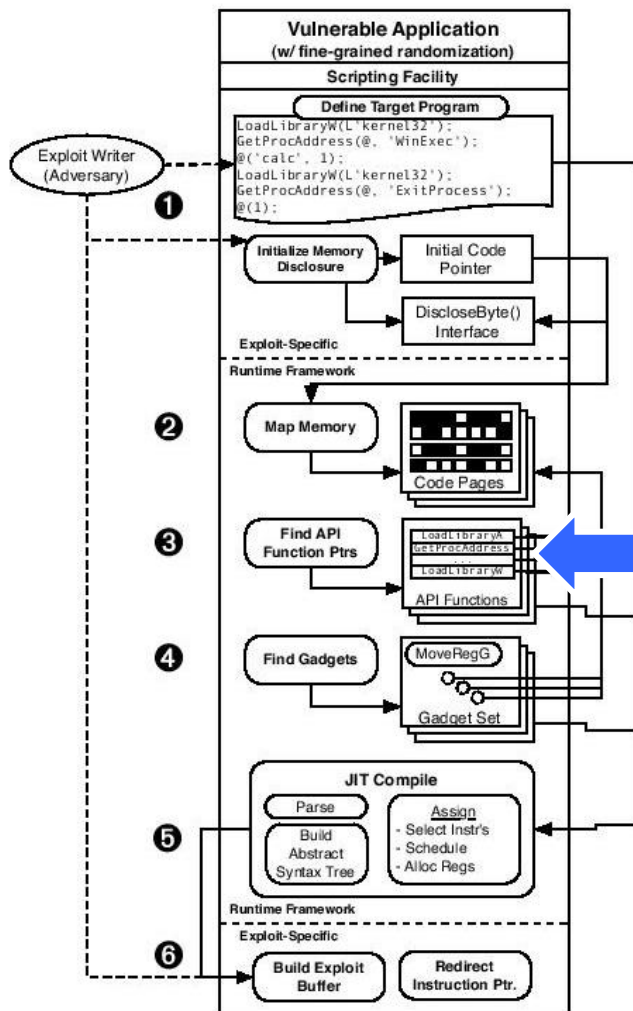
Find one code pointer
(using any disclosure vulnerability)

The entire page must be code...
Analyze the instructions to find
jumps and calls to other code pages...

Map out a big portion of
the application's code pages

Just-in-Time Code Reuse (2)

[Snow et al., Oakland 2013]

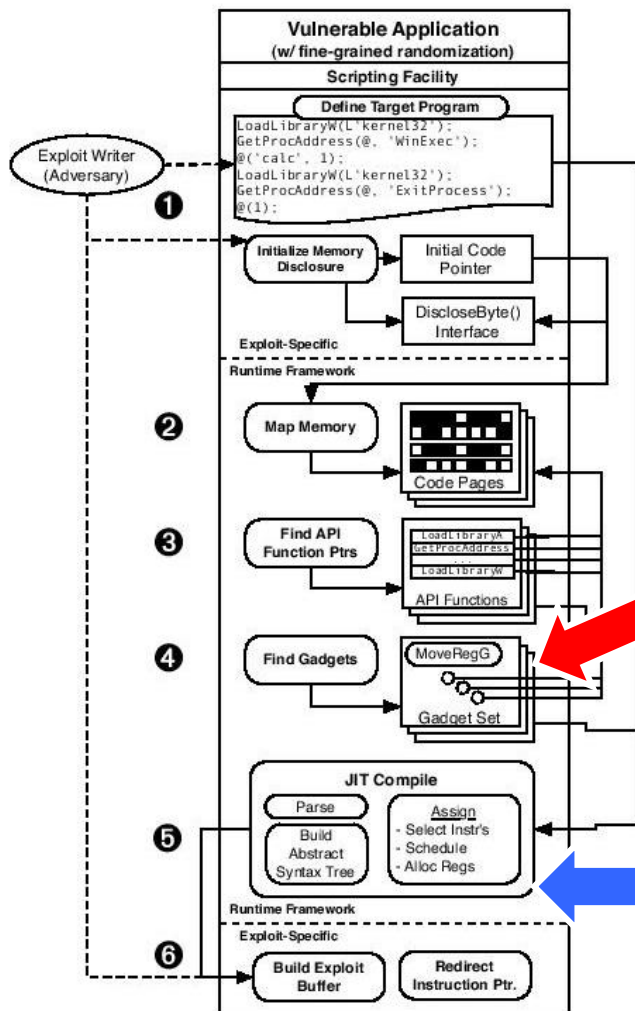


Use typical opcode sequences to find calls to `LoadLibrary()` and `GetProcAddress()`...

These can be used to invoke any library function by supplying the right arguments - don't need to discover the function's address!

Just-in-Time Code Reuse (3)

[Snow et al., Oakland 2013]



Collect gadgets in runtime by analyzing the discovered code pages (dynamic version of Shacham's "Galileo" algorithm)

Compile on the fly into shellcode

CFI: Control-Flow Integrity

[Abadi et al. "Control-Flow Integrity". CCS 2005]

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

CFI: Binary Instrumentation

Use binary rewriting to instrument code with runtime checks (similar to SFI)

Inserted checks ensure that the execution always stays within the statically determined CFG

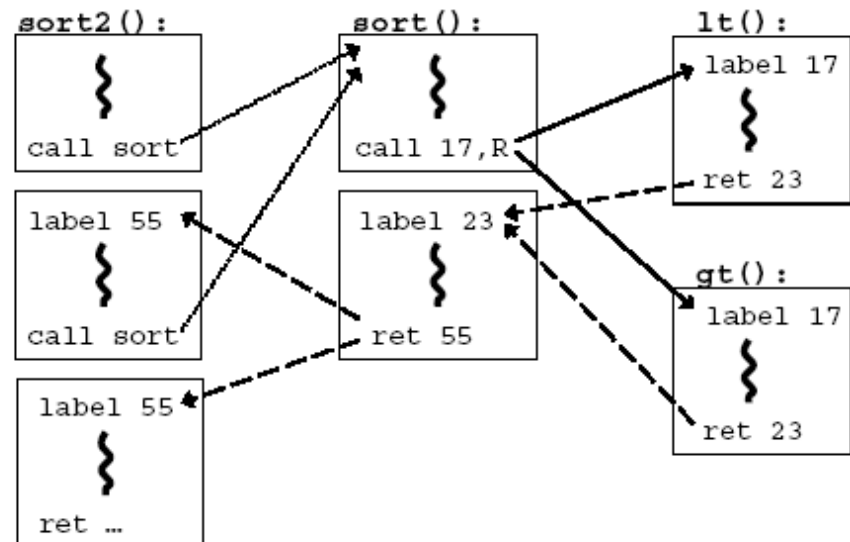
- Whenever an instruction transfers control, destination must be valid according to the CFG

Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc)

- Secure 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 );  
}
```



CFI: Control Flow Enforcement

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
 - This is imprecise (why?)
- 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

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

Instrumented code

B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

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

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

CFI: Preventing Circumvention

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

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 allows an “invalid” call from A to D
- Possible solution: duplicate code or inline
- Possible solution: 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

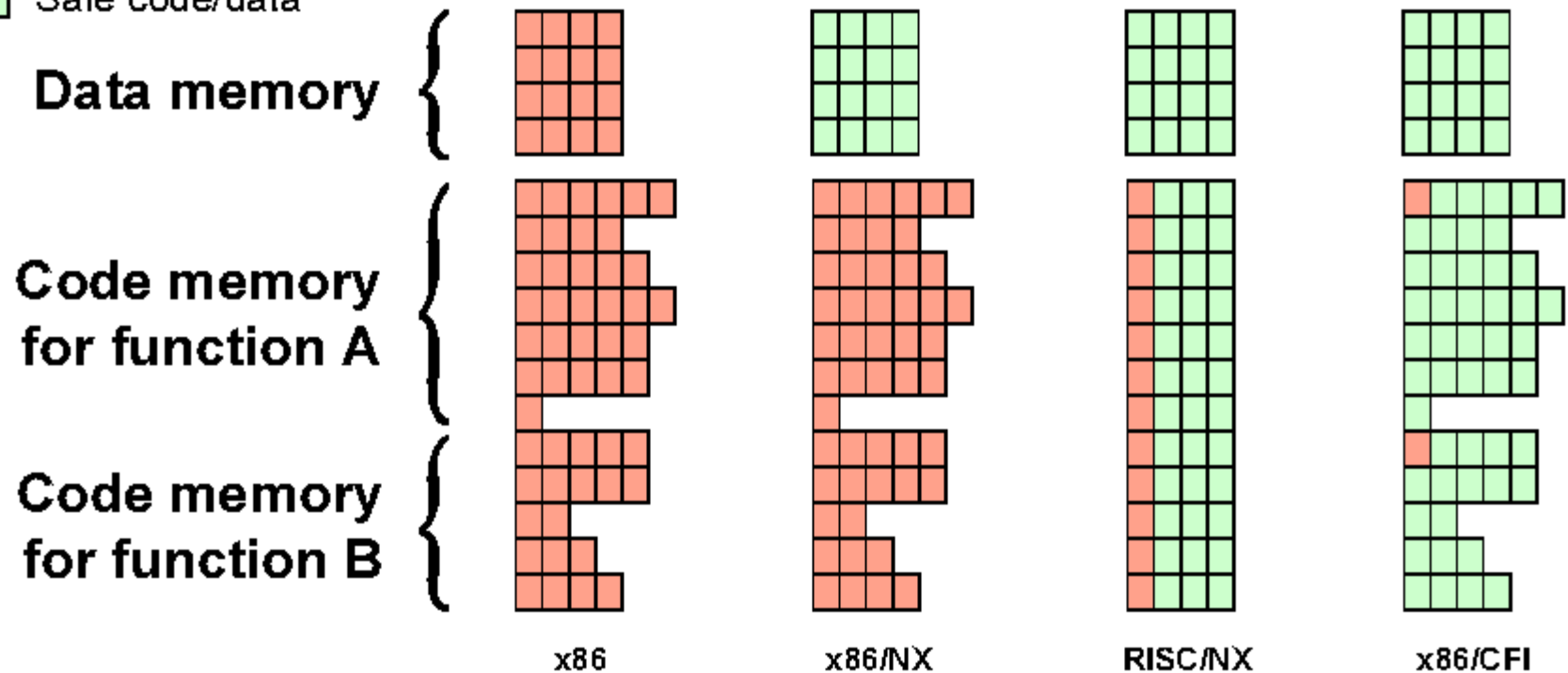
- Incorrect arguments to system calls
- Substitution of file names
- Other data-only attacks

Possible Execution of Memory

[Erlingsson]

-  Possible control flow destination
-  Safe code/data

Possible Execution of Memory



Control-Flow Guard (CFG)

Windows 10

Poor man's version of CFI:

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

```
rep stosd  
mov     esi, [esi]  
mov     ecx, esi           ; Target  
push    1  
call    @_guard_check_icall@4 ; _guard_check_icall(x)  
call    esi  
add     esp, 4  
xor     eax, eax
```

ensures target is
the entry point of a
function

CFI Using Intel's CET

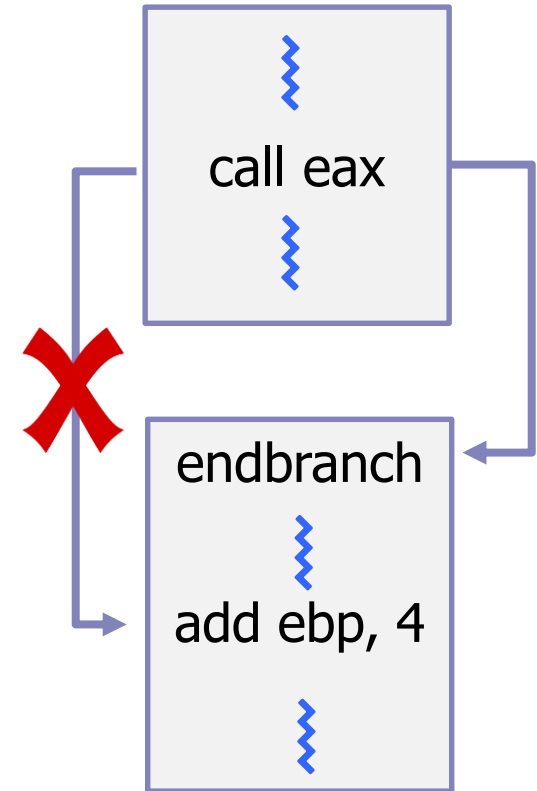
Deployed in Intel Tiger Lake (2020)

New **EndBranch** (ENDBR64) instruction:

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

Ensures an indirect JMP or CALL can only go
to a valid target address \Rightarrow no func. ptr. hijack
(compiler inserts EndBranch at valid locations)



CFI and CET

Windows 10

Poor man's version of CFI:

Practical

- Does not prevent attacker from causing a jump to a valid **wrong** function

- Hard to build accurate control flow graph statically

```
rep s  
mov  
mov  
push  
call  
call  
add    esp, 4  
xor    eax, eax
```

et is
nt of a

Example of a Bypass

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



Attacker
controls
handler

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

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

static CFI permits
attacker to call
DataHandler to
bypass
authentication

Cryptographic CFI (CCFI)

Attacker can read/write **anywhere** in memory,
Program should not deviate from its control flow graph

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 || source-address))**

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

Before following address, verify AES-MAC and crash if invalid

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

also ARM pointer authentication