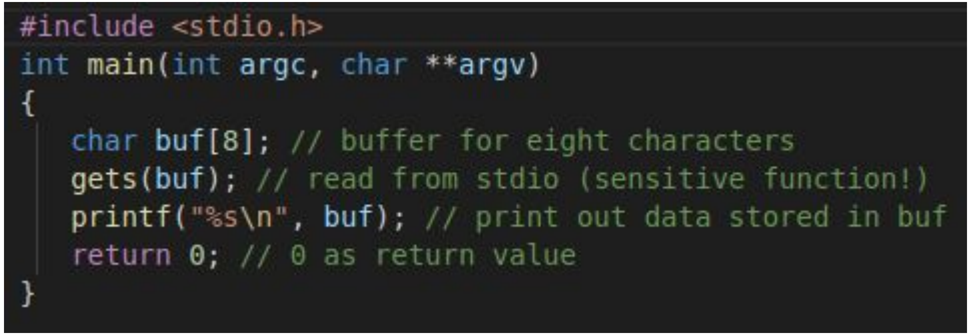
Binary analysis: Defense Mechanism

Yue Duan

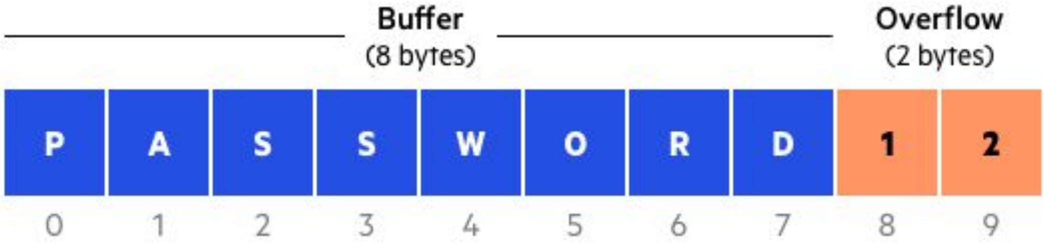
Outline

- Control-flow problems
- Simple defense mechanisms
- Research paper:
  - Control-Flow Integrity

Control-flow Violation



```
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read from stdio (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

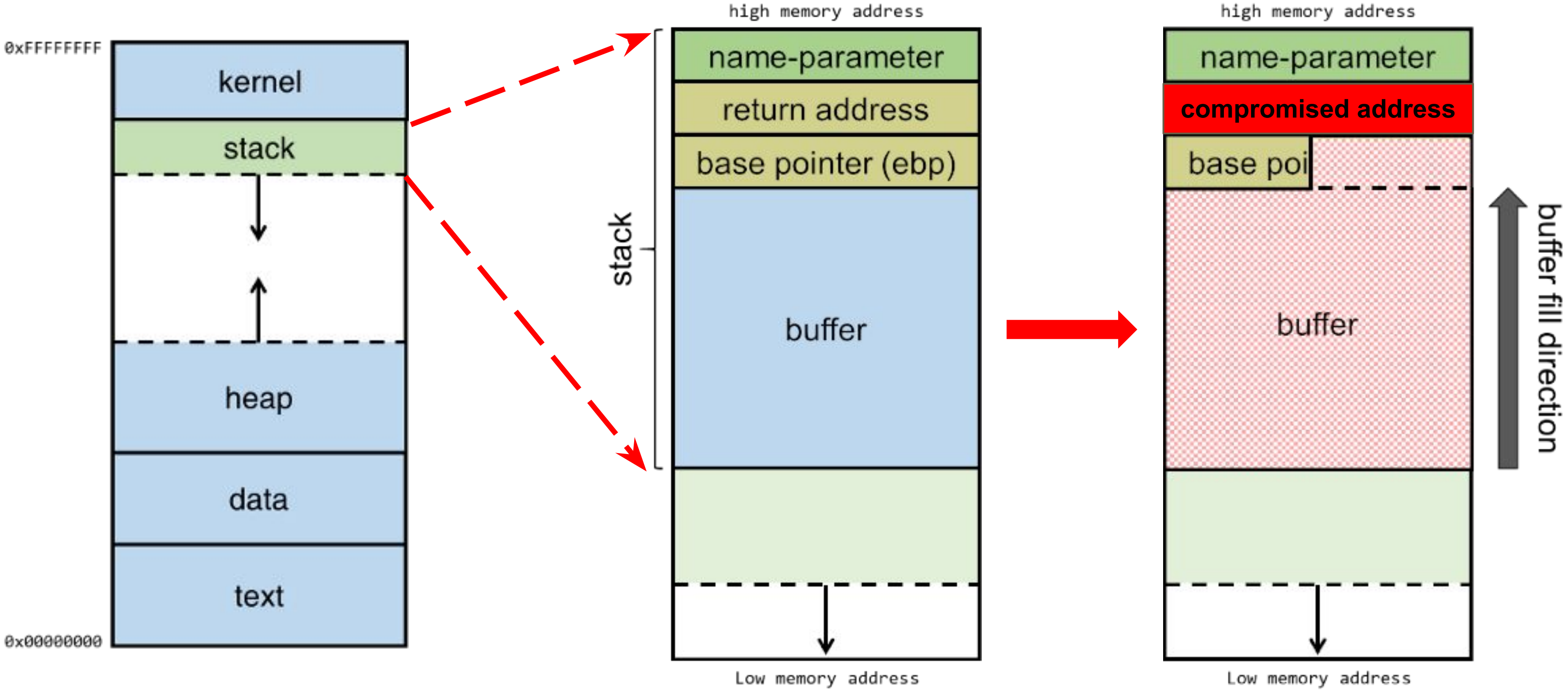


Buffer
(8 bytes)

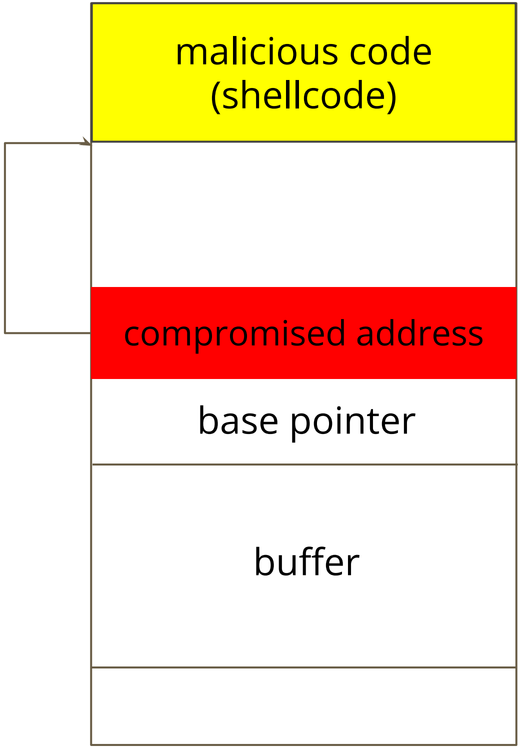
Overflow
(2 bytes)

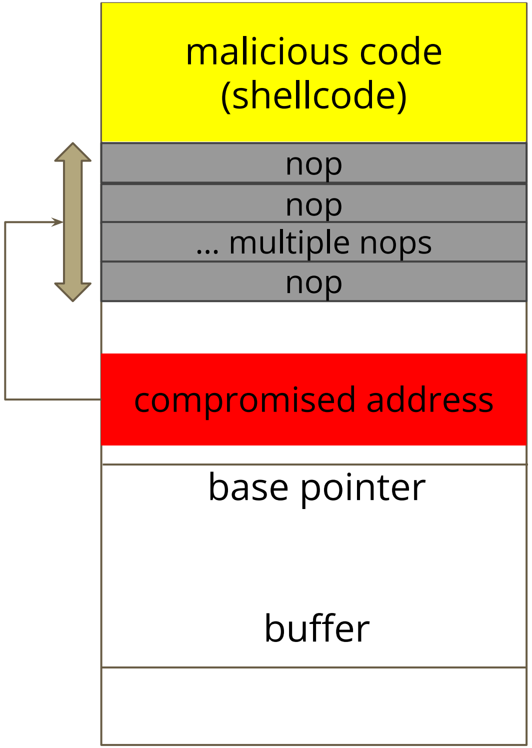
P	A	S	S	W	O	R	D	1	2
0	1	2	3	4	5	6	7	8	9

Control-flow Violation

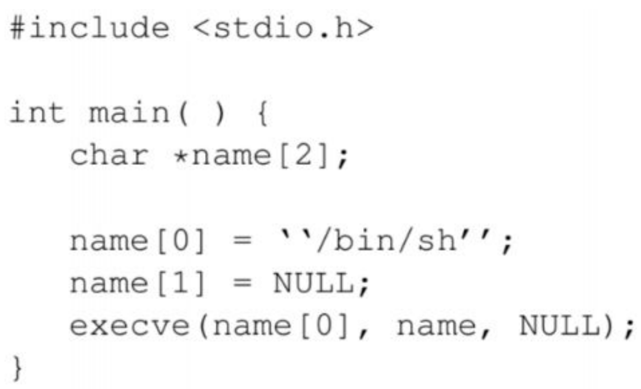


Control-flow Violation

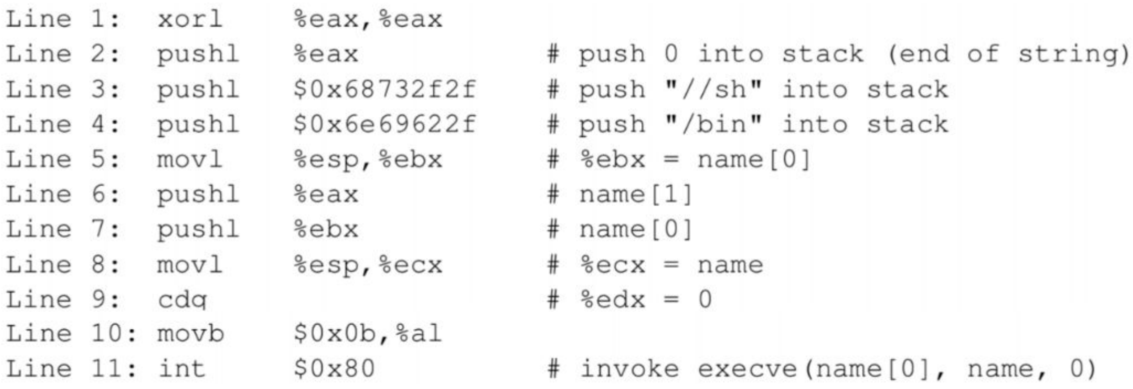




Shellcode



```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ‘‘/bin/sh’’;
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```



```
Line 1:  xorl    %eax,%eax
Line 2:  pushl   %eax            # push 0 into stack (end of string)
Line 3:  pushl   $0x68732f2f     # push "//sh" into stack
Line 4:  pushl   $0x6e69622f     # push "/bin" into stack
Line 5:  movl    %esp,%ebx       # %ebx = name[0]
Line 6:  pushl   %eax            # name[1]
Line 7:  pushl   %ebx            # name[0]
Line 8:  movl    %esp,%ecx       # %ecx = name
Line 9:  cdq                     # %edx = 0
Line 10: movb    $0x0b,%al
Line 11: int     $0x80           # invoke execve(name[0], name, 0)
```

Steps

- insert shellcode into somewhere in memory
- overwrite return address with shellcode’s address
- when function returns, shellcode will be executed
- a shell will be launched
- if the vulnerable program happens to be root-owned SetUID program

Fundamental Problems

- No distinguishment between code and data
    - data in the process can be interpreted as code
- Attacker can easily redirect control flow to the injected data (code)

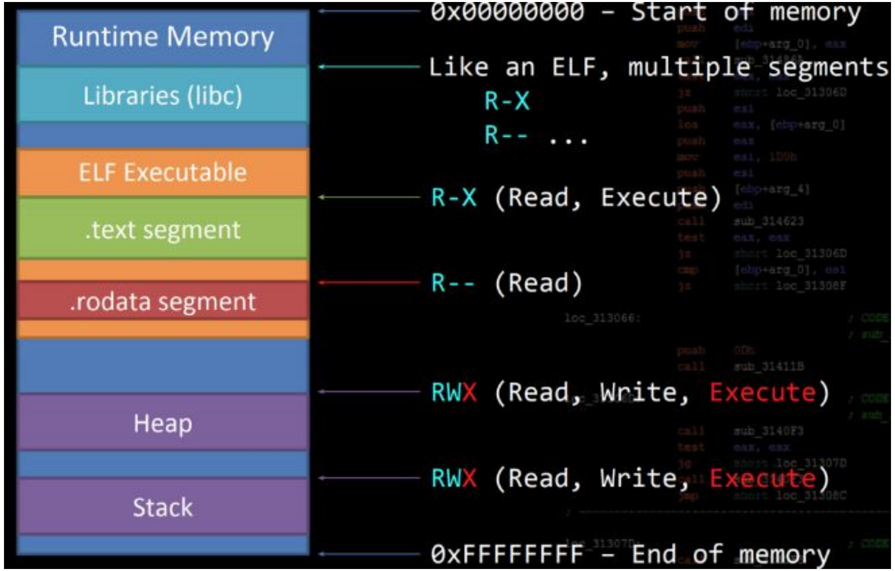
Simple Defense Mechanisms

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack Canary

Data Execution Prevention

- System-level memory protection feature
- DEP prevents code from being run from data pages
    - heap, stacks, etc
- If an application attempts to run
    - memory access violation exception is raised
    - process is terminated

Data Execution Prevention



Runtime Memory

Libraries (libc)

ELF Executable

.text segment

.rodata segment

Heap

Stack

0x00000000 - Start of memory

Like an ELF, multiple segments
R-X
R-- ...

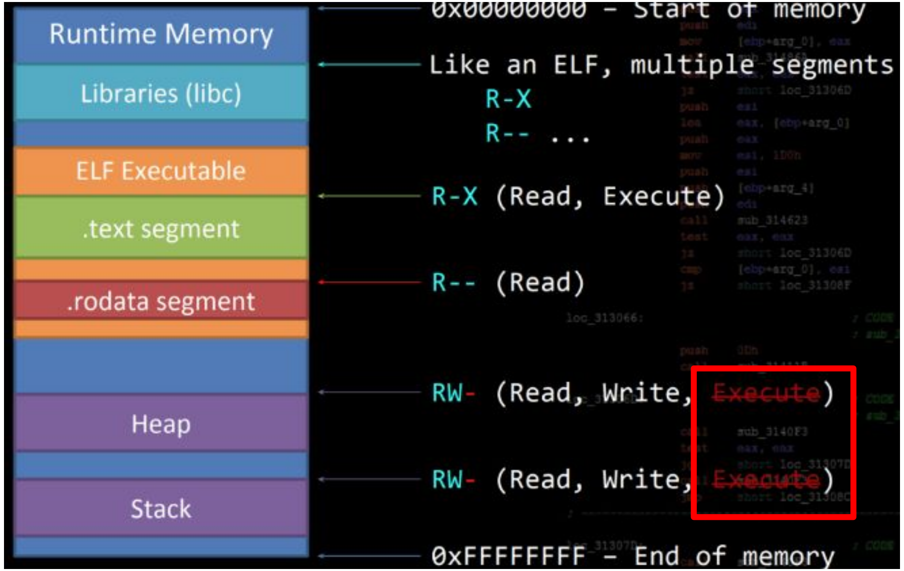
R-X (Read, Execute)

R-- (Read)

RWX (Read, Write, Execute)

RWX (Read, Write, Execute)

0xFFFFFFFF - End of memory



Runtime Memory

Libraries (libc)

ELF Executable

.text segment

.rodata segment

Heap

Stack

0x00000000 - Start of memory

Like an ELF, multiple segments
R-X
R-- ...

R-X (Read, Execute)

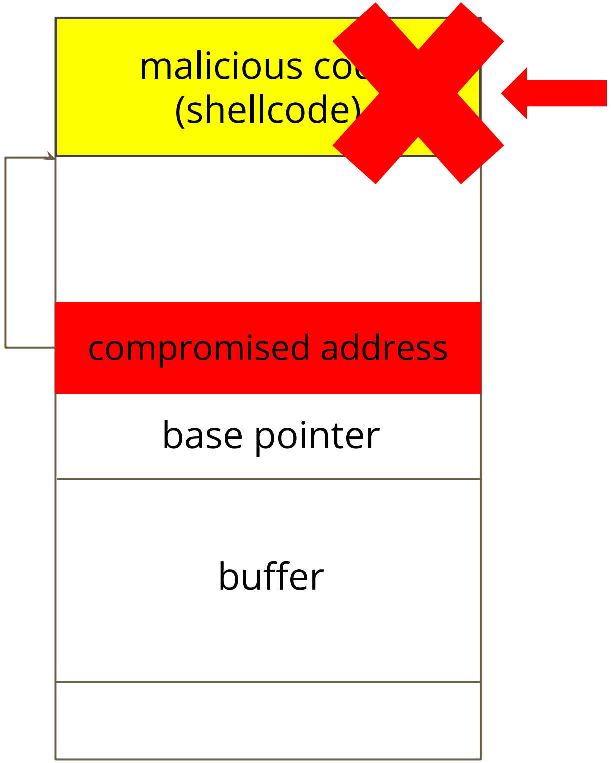
R-- (Read)

RW- (Read, Write, ~~Execute~~)

RW- (Read, Write, ~~Execute~~)

0xFFFFFFFF - End of memory

Data Execution Prevention

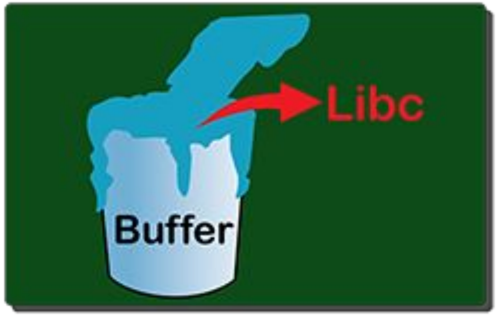


not executable
SEGFAULT

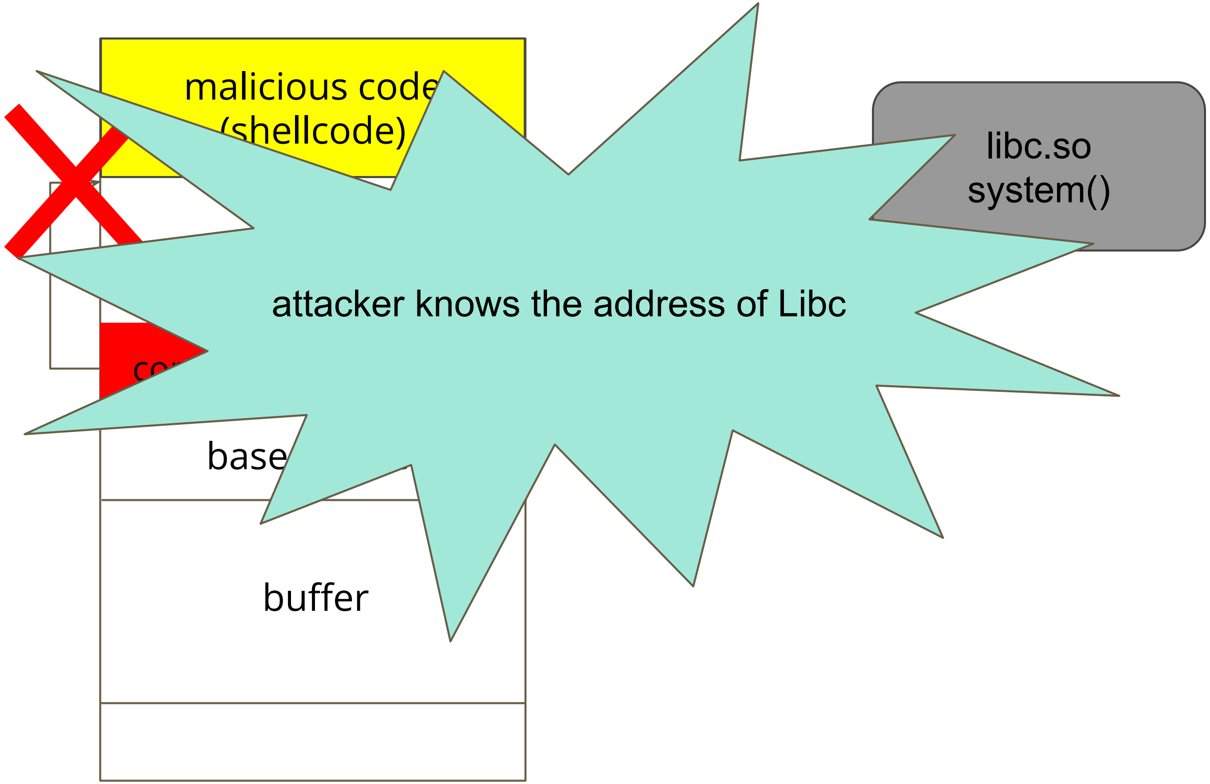
Evasion

- What if …
  - attacker does not inject code
  - in stead, he reuses existing code

Return-to-libc attack : reuse code in Libc



Return-to-Libc



malicious code (shellcode)

co

base

buffer

libc.so
system()

attacker knows the address of Libc

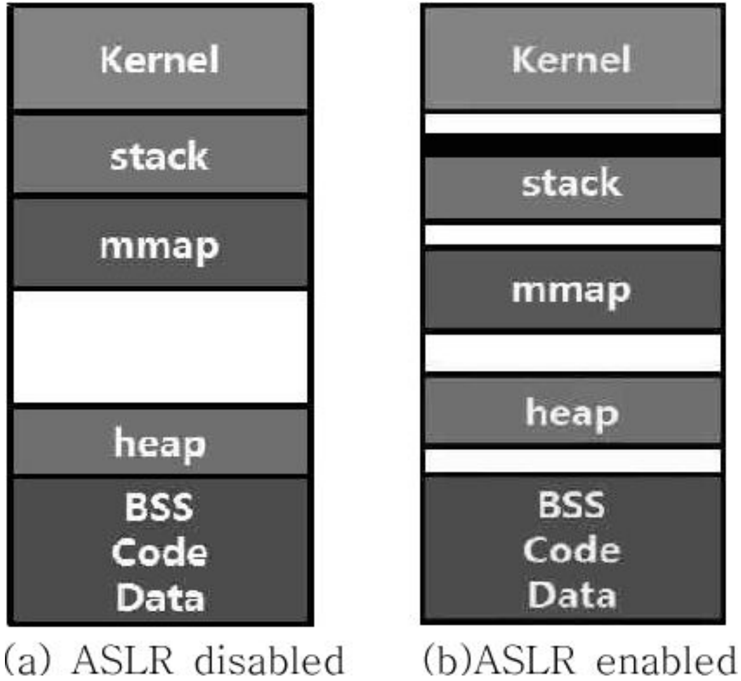
Address Space Layout Randomization (ASLR)

- Multiple attacks rely on guessing addresses
  - inject shellcode
  - return-to-libc

- Key idea:
  - Can we make the guessing practically impossible?

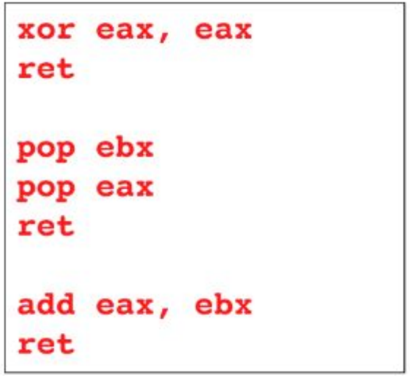
Address Space Layout Randomization (ASLR)

- Randomize the address space positions
  - base of the executable
  - stack
  - heap
  - libraries
- Possible evasions
  - bruteforce
  - information leakage
  - return-oriented programming (ROP)



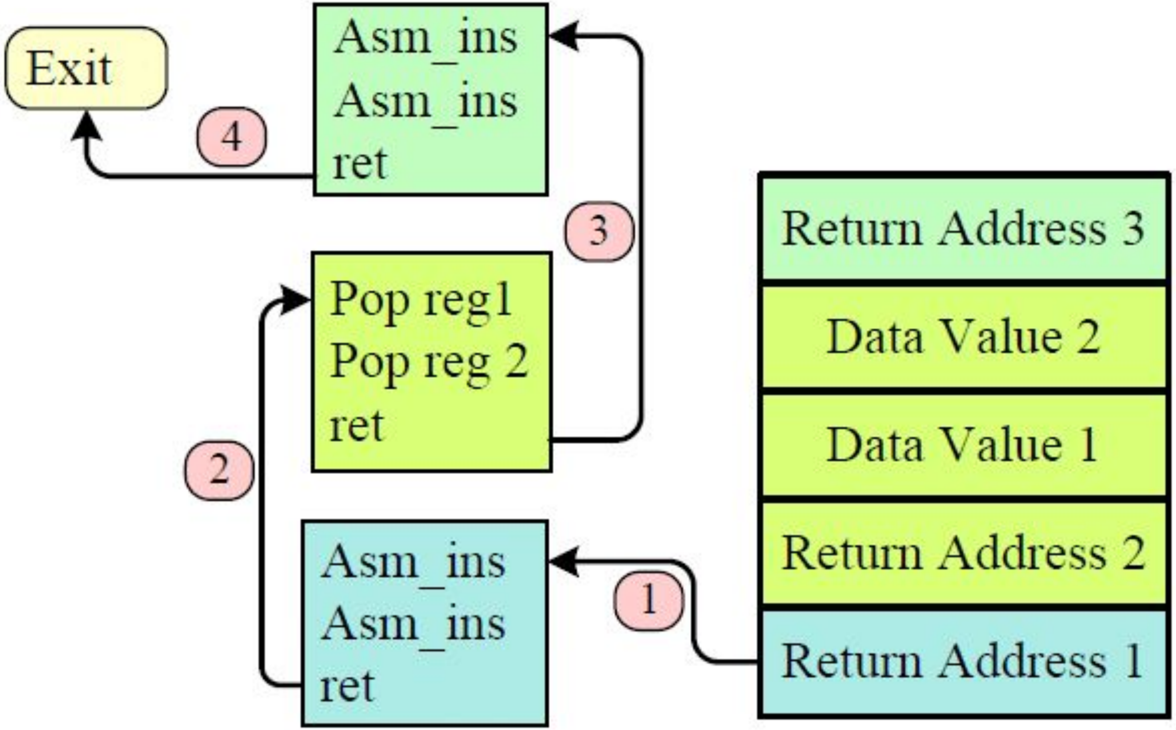
Return-Oriented Programming (ROP)

- Key idea:
  - reuse existing code without knowing the exact addresses
  - find meaningful code ‘gadgets’
  - chain ‘gadgets’ together with return to complete a malicious action



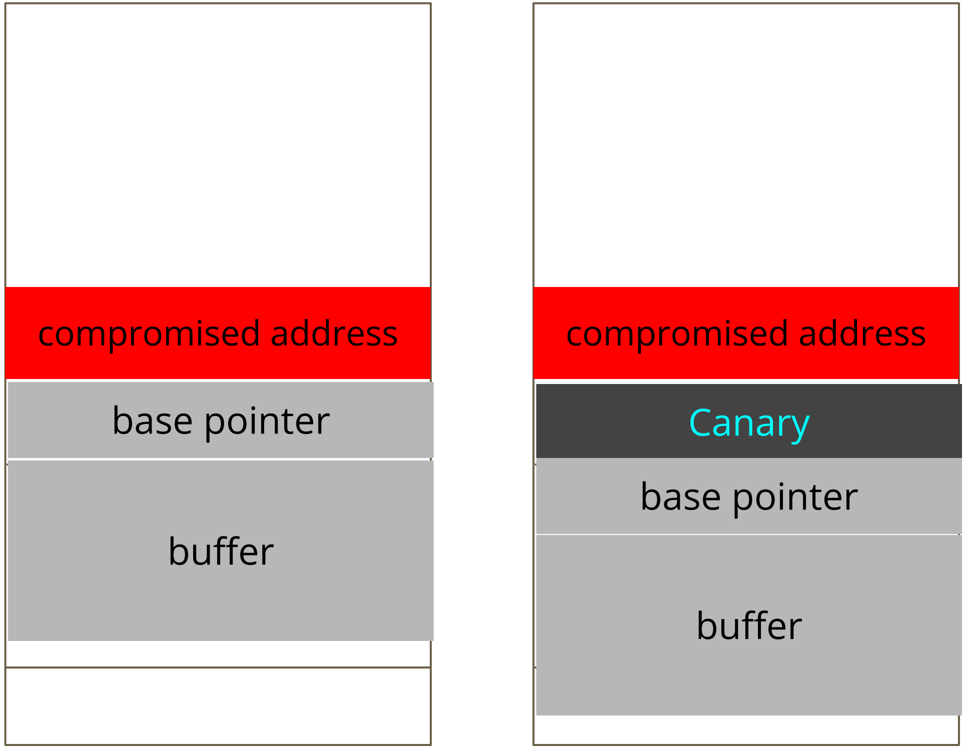
```
xor eax, eax
ret

pop ebx
pop eax
ret

add eax, ebx
ret
```

Return-Oriented Programming (ROP)



Stack Canary

- Also called StackGuard
- random integer
    - next to the return address
- before return, always check if
    - canary has been changed
- Built-in feature for most of the modern compilers
- Can be guessed



Control-Flow Integrity

M Abadi, M Budiu, Ú Erlingsson, J Ligatti

ACM CCS 2005

Motivation

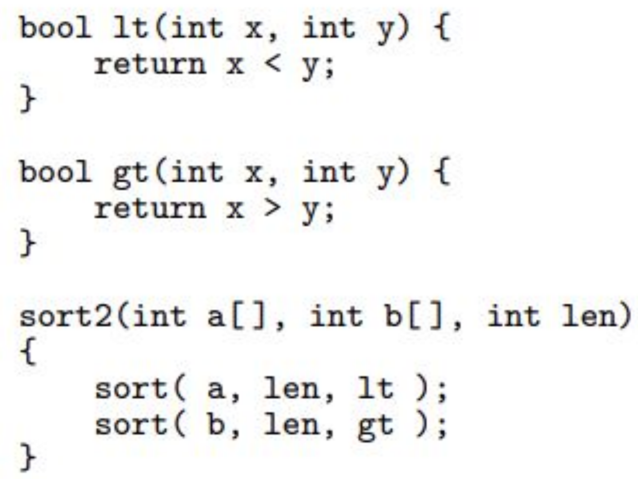
- Many attacks involve control-flow hijacking
- Existing defense mechanisms
  - either impractical
  - or need hardware support
- Fundamental limitations
  - lack of a realistic attack model
  - reliance on informal reasoning and hidden assumptions

- Major challenge: How to protect control-flow from being hijacked?

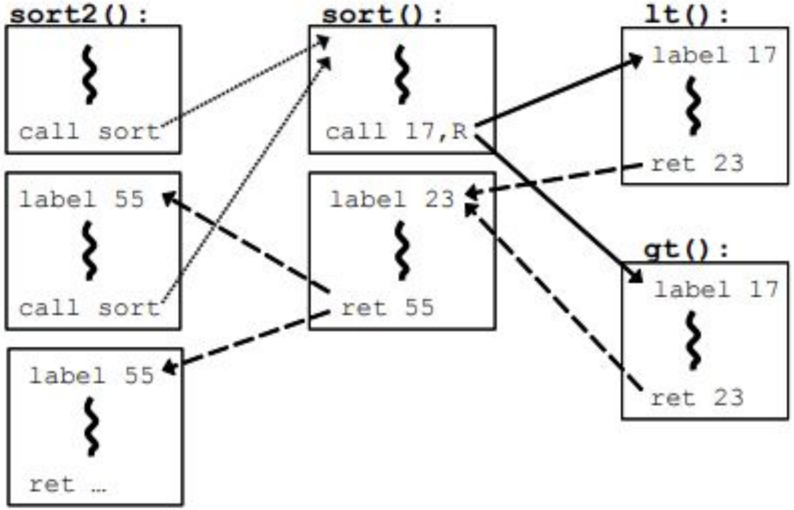
Control-Flow Integrity

- Key idea:
    - execution should always follow the pre-defined control flow (CFI security policy)!
- CFI policy
    - extracted from Control-flow graph (CFG)
    - enforced at runtime with checks

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 );
}
```

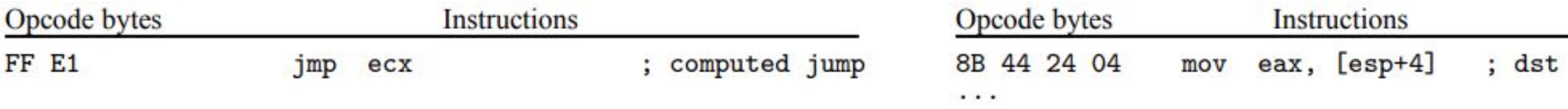


CFI Enforcement

- For each control transfer, such as a function call
  - statically determine its possible destinations
- Insert a unique bit pattern at every destination
  - two destinations are considered equivalent if CFG contains edges to each from the same source
- Insert codes that
  - enforce runtime check
  - whether the bit pattern of the target instruction matches the pattern of possible destinations
- Done by binary instrumentation technique

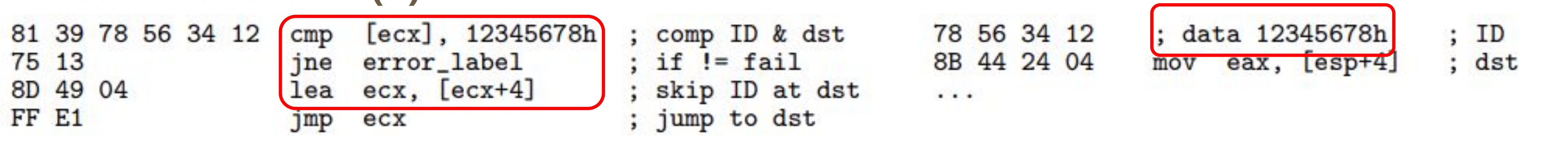
CFI Enforcement - jmp

original code



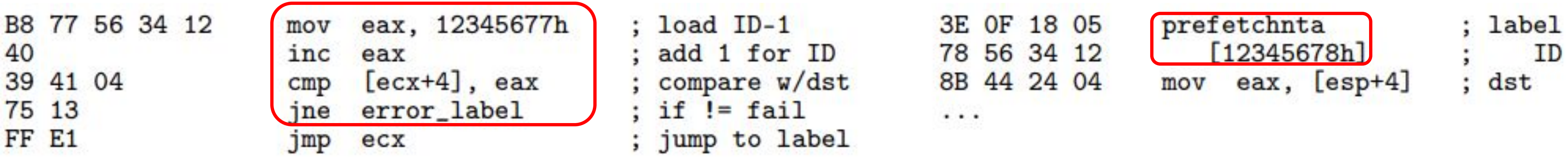
```
Opcode bytes          Instructions                              Opcode bytes          Instructions
FF E1                 jmp   ecx              ; computed jump    8B 44 24 04     mov   eax, [esp+4]     ; dst
                                                                ...
```

instrumentation (a)



```
81 39 78 56 34 12     cmp   [ecx], 12345678h  ; comp ID & dst   78 56 34 12     ; data 12345678h     ; ID
75 13                 jne   error_label       ; if != fail      8B 44 24 04     mov   eax, [esp+4]   ; dst
8D 49 04              lea   ecx, [ecx+4]      ; skip ID at dst  ...
FF E1                 jmp   ecx               ; jump to dst
```

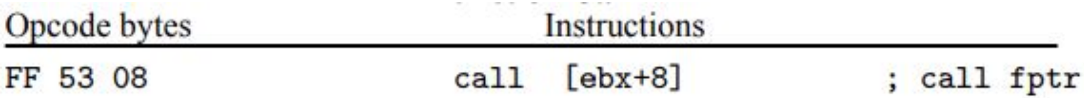
instrumentation (b)



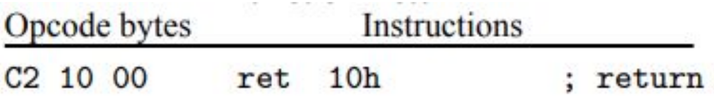
```
B8 77 56 34 12        mov   eax, 12345677h    ; load ID-1       3E 0F 18 05     prefetchnta          ; label
40                    inc   eax               ; add 1 for ID    78 56 34 12          [12345678h]     ;    ID
39 41 04              cmp   [ecx+4], eax      ; compare w/dst   8B 44 24 04   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

CFI Enforcement - ret

original code

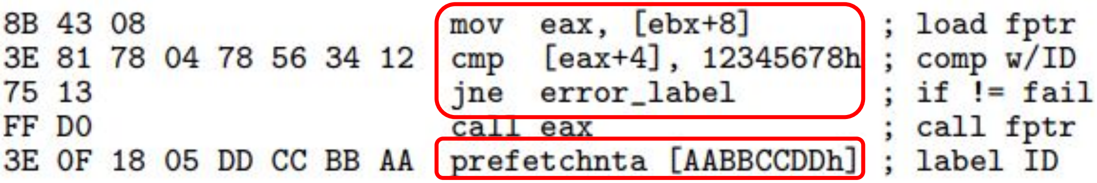


```
Opcode bytes                 Instructions
FF 53 08            call  [ebx+8]          ; call fptr
```

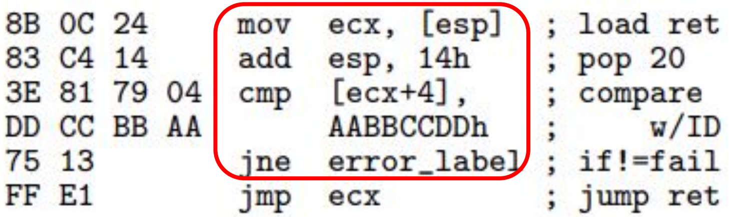


```
Opcode bytes          Instructions
C2 10 00      ret  10h               ; return
```

instrumentation



```
8B 43 08                  mov   eax, [ebx+8]       ; load fptr
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h ; comp w/ID
75 13                     jne   error_label        ; if != fail
FF D0                     call  eax                ; call fptr
3E 0F 18 05 DD CC BB AA   prefetchnta [AABBCCDDh]  ; label ID
```



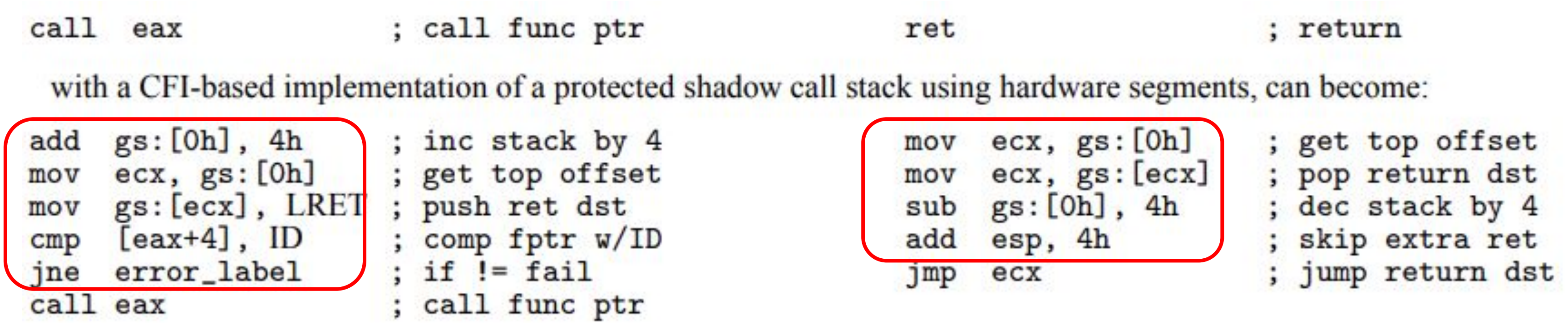
```
8B 0C 24      mov   ecx, [esp]      ; load ret
83 C4 14      add   esp, 14h        ; pop 20
3E 81 79 04   cmp   [ecx+4],        ; compare
DD CC BB AA         AABBCCDDh       ;      w/ID
75 13         jne   error_label     ; if!=fail
FF E1         jmp   ecx             ; jump ret
```

CFI Precision

- Assume that:
  - A() calls C()
  - B() calls C() or D() (When can this happen?)
- CFI will use the same tag for C and D
  - allow A to call D
- Possible solutions:
  - duplicate code or inlining
  - multiple tags

CFI Precision

- function F() is called twice from A() and B()
  - CFI uses the same tag for both call sites
  - allow F() to return to B() after being called from A()
- Solution: shadow call stack
  - always guarantee the return to the latest call site



```
call  eax              ; call func ptr          ret                  ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add   gs:[0h], 4h      ; inc stack by 4         mov   ecx, gs:[0h]   ; get top offset
mov   ecx, gs:[0h]     ; get top offset         mov   ecx, gs:[ecx]  ; pop return dst
mov   gs:[ecx], LRET   ; push ret dst           sub   gs:[0h], 4h    ; dec stack by 4
cmp   [eax+4], ID      ; comp fptr w/ID         add   esp, 4h        ; skip extra ret
jne   error_label      ; if != fail             jmp   ecx            ; jump return dst
call  eax              ; call func ptr
```

Evaluation: performance overhead

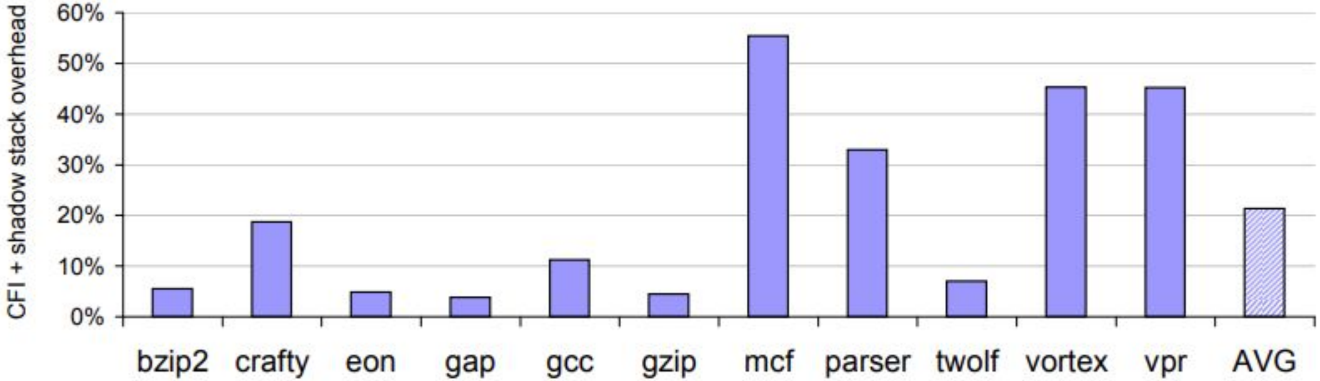


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

- modest performance overhead
  - on average 21%
  - 5% for gzip and 11% for gcc

Thank you!

Question?