Reverse Engineering
Class 8

Exploit Writing I
Stack and Integer Overflow

martin.uy
Open by default.

Stack Overflow

- What's a stack? (x86)
  - Memory area used to store local variables, function parameters, saved registers, return addresses (in function calls) and stack dynamically allocated memory
  - Each thread has 2 stacks:
    - Stack in user space
    - Stack in kernel space (when thread executes a *syscall*)
    - Why? 

Stack Overflow

- What’s a stack? (x86)
  - Stack is not shared between threads: no concurrency issues for data stored there
  - User space stacks are generally in high virtual memory addresses and, in x86 / x86_64, grow towards lower virtual memory addresses
  - Top of stack is pointed by ESP register (RSP in x86_64)
    - A stack growing does not necessarily implies memory allocation: memory may be already allocated and only the register that points to the top of the stack is modified
  - Stacks have a maximum capacity defined when the thread is created (I.e. 2MB for user stacks)

Stack Overflow

Syscalls entry point (x86_64, Linux kernel)

```
ENTRY(entry_SYSCALL_64)

...

movq   %rsp, PER_CPU_VAR(rsp_scratch)
movq   PER_CPU_VAR(cpu_current_top_of_stack), %rsp

...
```

arch/x86/entry/entry_64.S

(gdb) print $rsp
$1 = (void *) 0x7ffcf152c368 → User-space stack pointer

(gdb) print $rsp
$2 = (void *) 0xffffc90000b40000 → Kernel-space stack pointer

Stack Overflow

- Stacks in Linux (kernel)
  - sys_clone (thread/process creation)
  - _do_fork (fork.c)
  - copy_process (fork.c)
  - dup_task_struct (fork.c)
  - alloc_thread_stack_node (fork.c)
  - __vmalloc_node_range (vmalloc.c)

Stack Overflow

- Stack in Linux (kernel)
  - struct task_struct {

```
    …
    void *stack;
    …
}
```

include/linux/sched.h

Stack Overflow

- Breakpoint in syscall entry (x86_64)

PID	Stack top	Stack bottom (current->stack)	Size
768	0xffffc90000**bd8000**	0xffffc90000**bd4000**	16384
725	0xffffc90000**694000**	0xffffc90000**690000**	16384
731	0xffffc90000**6d4000**	0xffffc90000**6d0000**	16384
768	0xffffc90000**bd8000**	0xffffc90000**bd4000**	16384
731	0xffffc90000**6d4000**	0xffffc90000**6d0000**	16384

Stack Overflow

- Stack use
  - Instructions that implicitly modify the stack (x86 / x86_64)
    - PUSH, POP, PUSHAD, POPAD, CALL, LEAVE, RET, RET n
    - The number of bytes affected in each of this operations is related to the architecture natural size. In example, in x86_64 a CALL will push 8 bytes to the stack containing the return address
  - Instructions that explicitly modify the stack
    - I.e. SUB ESP, 10h

Stack Overflow

- Examples

```
; int __cdecl main(int, char **, char **)
main proc near

var_205C= dword ptr -205Ch
src= qword ptr -2058h
size= qword ptr -2050h
dest= byte ptr -2048h
var_40= qword ptr -40h

push    r15
push    r14
mov     r15, rsi
push    r13
push    r12
push    rbp
push    rbx
movsxd  rbx, edi
sub     rsp, 2038h
```

Stack Overflow

- Examples

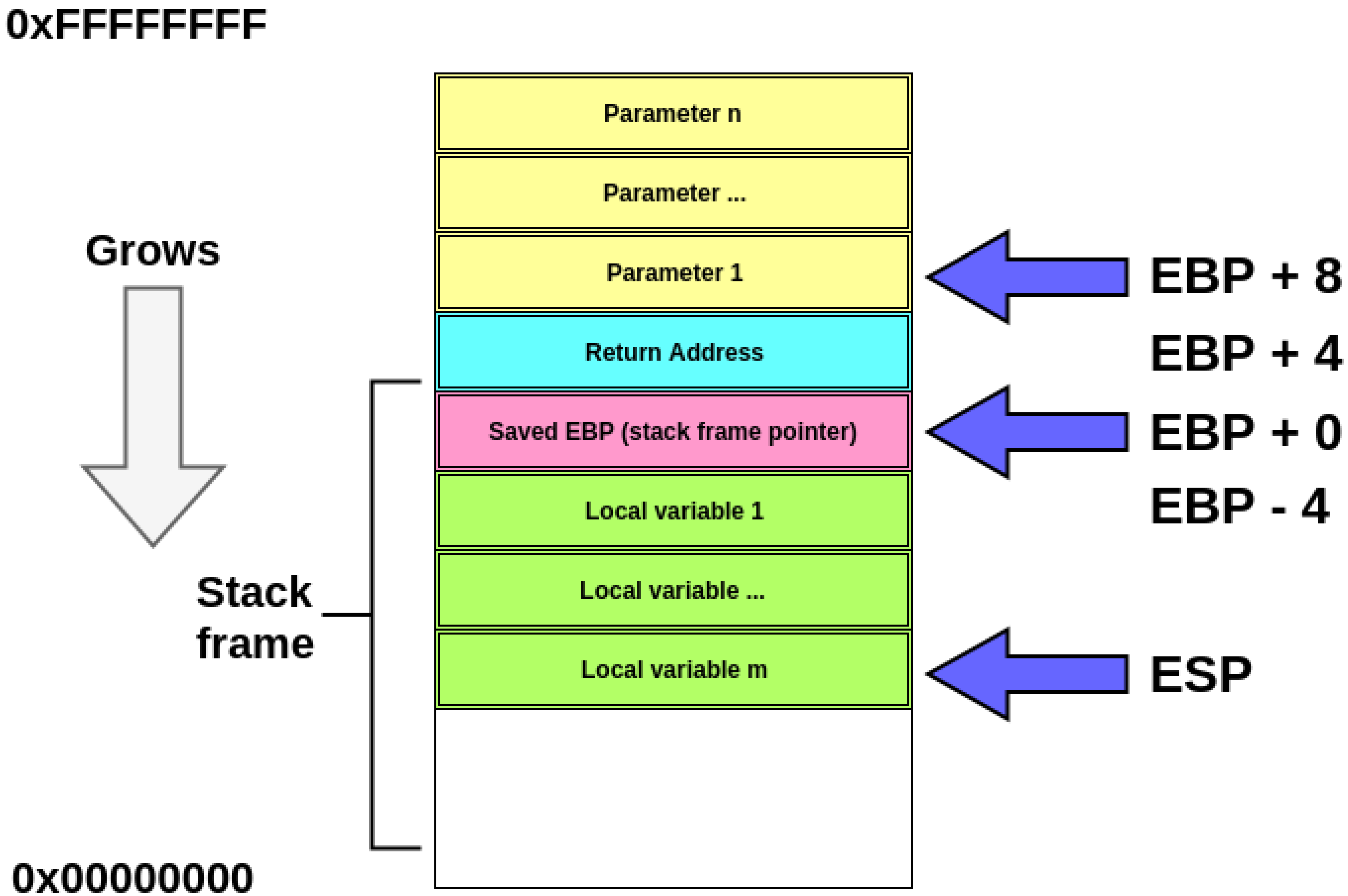
```
(gdb) x/1i $rip
=> 0x555555555670 <main>:       push   %r15
(gdb) set $r15 = 0x4141414141414141
(gdb) x/1xg $rsp
0x7fffffffdfd8: 0x00007ffff7a31401
(gdb) si
0x0000555555555672 in main ()
(gdb) x/2xg $rsp
0x7fffffffdfd0: 0x4141414141414141      0x00007ffff7a31401
(gdb) 
```

Stack Overflow

- Stack overflow is a type of vulnerability caused by a memory corruption
- Independent from the operating system and may apply to different architectures. We will study it in x86/x86_64
- Allows to take control of the instruction pointer and/or modify local variables in a function (data attacks)
- This is possible because data (writable) is mixed with pointers to code within the same stack:
  - return addresses
  - pointers to vtables (that contain pointers to code)
  - pointers to exception handlers
- Vulnerability described in "Smashing The Stack For Fun and Profit" paper in 1996, by Elias Levy

Stack Overflow

- Application Binary Interface for CALLs (x86)



Stack

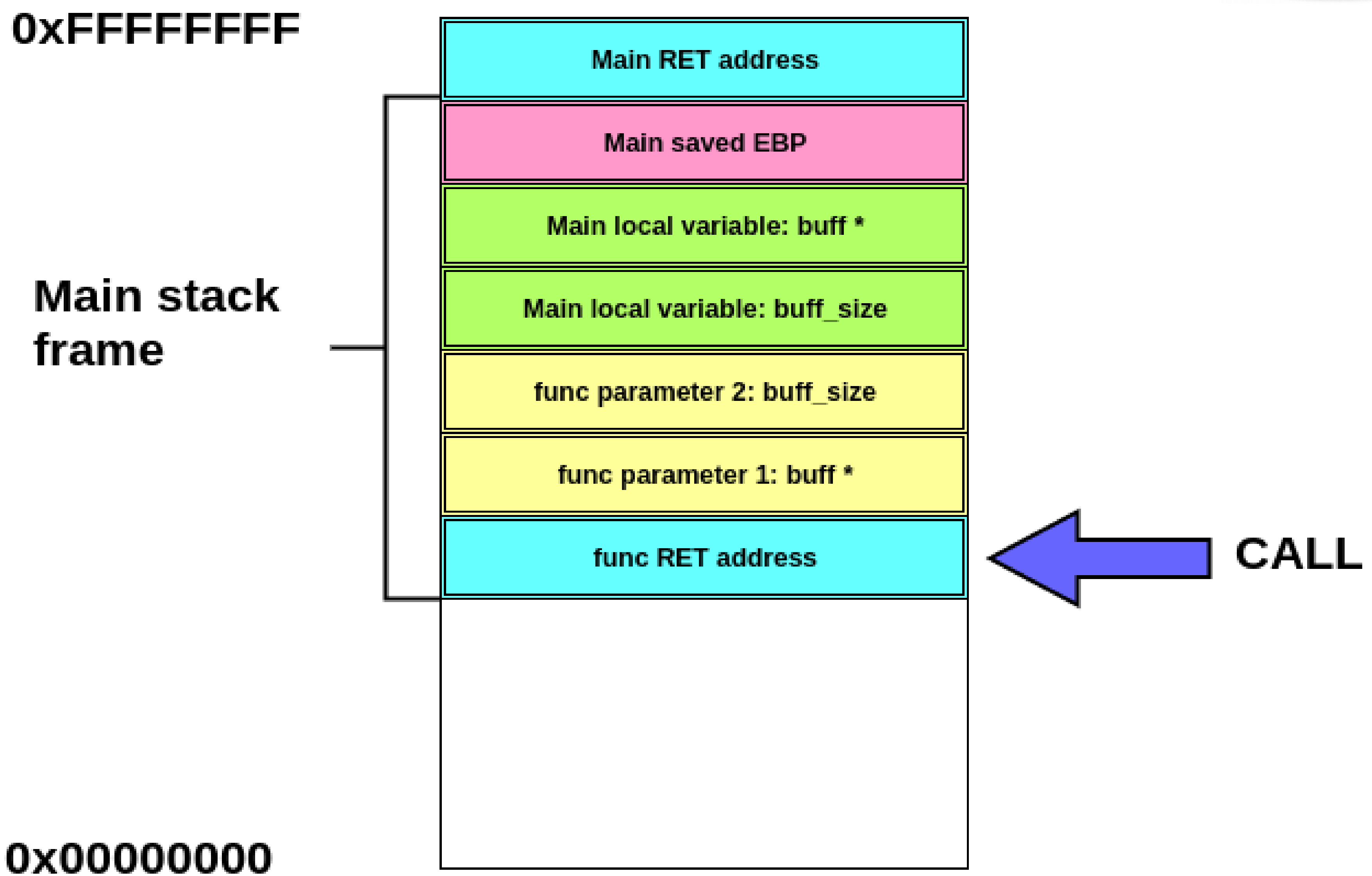
1 stack in user-space per main thread

Stack Overflow

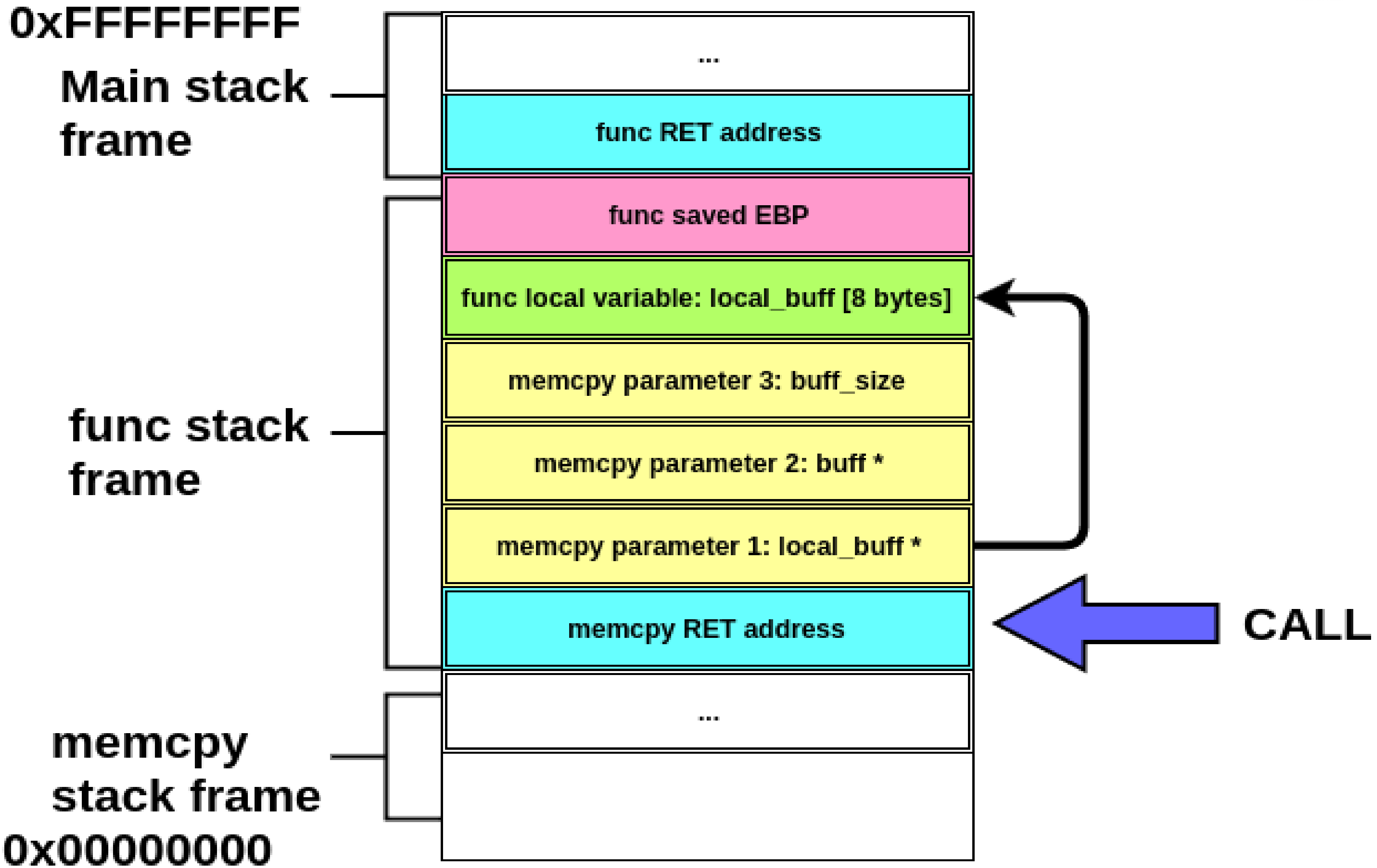
- Where is the vulnerability?

```
void main(){
    ...
    func(buff, buff_size);
}

void func (const char* buff, size_t buff_size) {
    char local_buffer[8];
    memcpy((void*)local_buffer, (const void*)buff,
buff_size);
}
```

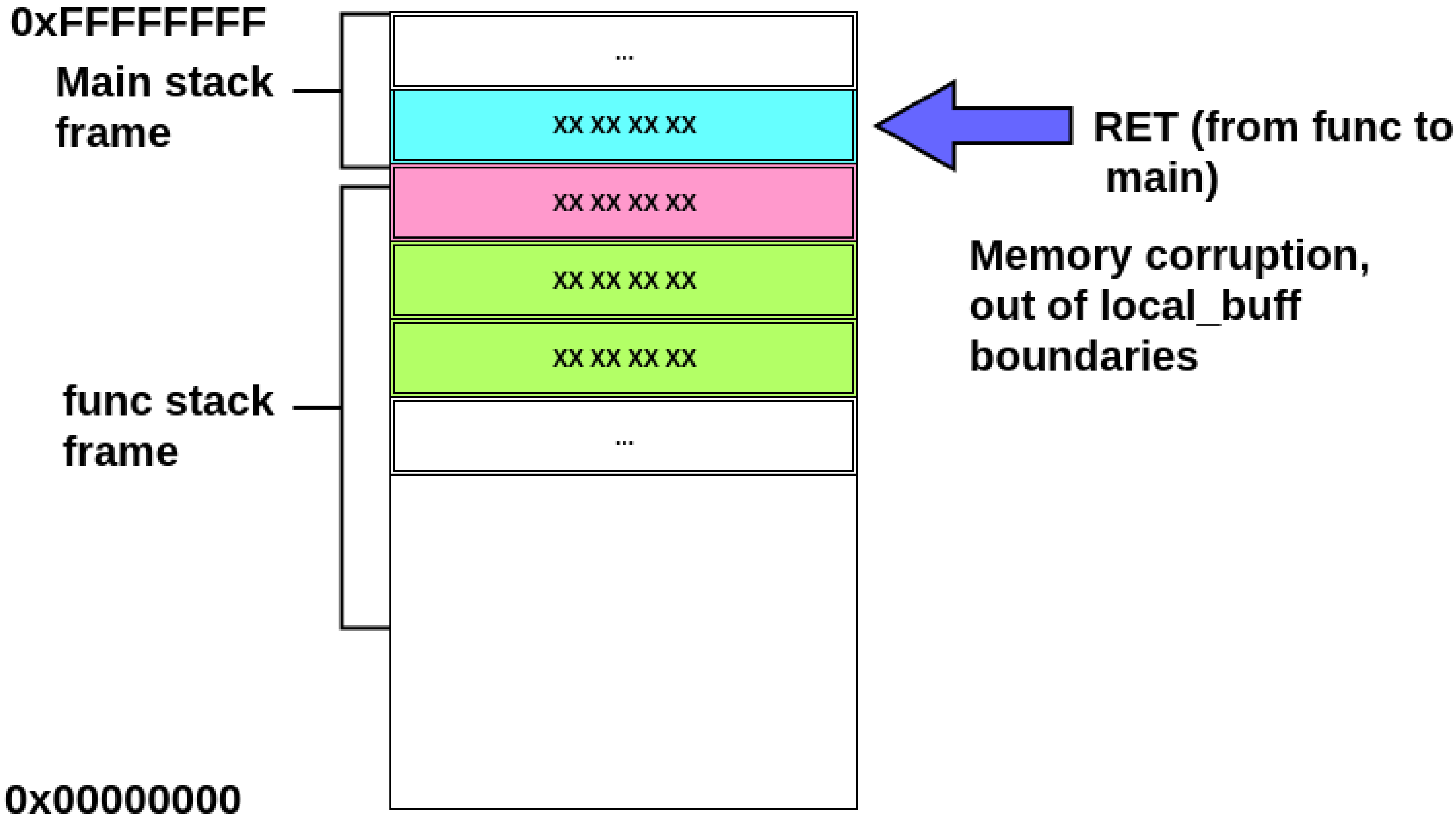
Stack Overflow



Stack Overflow



Stack Overflow



Stack Overflow

Pointer to buff (main local variable, in stack)

buff_size: 32 bytes

```
(gdb) x/1i $eip
=> 0x80484a5 <main+154>:       call   0x80484ba <func>
(gdb) x/2xw $esp
0xffffce90:     0xffffcea0      0x00000020
(gdb) x/32xb 0xffffcea0
0xffffcea0:     0x00    0x01    0x02    0x03    0x04    0x05    0x06    0x07
0xffffcea8:     0x08    0x09    0x0a    0x0b    0x0c    0x0d    0x0e    0x0f
0xffffceb0:     0x10    0x11    0x12    0x13    0x14    0x15    0x16    0x17
0xffffceb8:     0x18    0x19    0x1a    0x1b    0x1c    0x1d    0x1e    0x1f
```

Parameters to "func" (in stack)

buff in stack: 32 bytes, from 0x00 to 0x1F

Stack Overflow

Return address to main (in stack)

Parameters to “func”: pointer to buff and buff_size (in stack)

```
(gdb) x/1i $eip
=> 0x80484ba <func>:     push   %ebp
(gdb) x/3xw $esp
0xffffce8c:      0x080484aa      0xffffcea0      0x00000020
(gdb) x/6i 0x080484aa
   0x80484aa <main+159>:         add    $0x10,%esp
   0x80484ad <main+162>:         mov    $0x0,%eax
   0x80484b2 <main+167>:         mov    -0x4(%ebp),%ecx
   0x80484b5 <main+170>:         leave
   0x80484b6 <main+171>:         lea    -0x4(%ecx),%esp
   0x80484b9 <main+174>:         ret
```

Stack Overflow

memcpy destination buffer. Capacity: 8 bytes. func local variable: local_buffer (stack)

memcpy source buffer (pointer to buff)

Number of bytes to copy (32)

```
(gdb) x/1i $eip
=> 0x80484cd <func+19>: call   0x80482e0 <memcpy@plt>
(gdb) x/3xw $esp
0xffffce60:     0xffffce78      0xffffcea0      0x00000020
(gdb) x/8xw 0xffffce78
0xffffce78:     0xf7e6c1a9      0x00000016      0xffffffff      0xf7fa6000
0xffffce88:     0xffffcec8      0x080484aa      0xffffcea0      0x00000020
```

memcpy destination buffer. 8 bytes (stack garbage by now)

Pushed ebp when entering func

Return address to main (when exiting func)

Parameters to call func

Stack Overflow

Ex return address from func to main. Now it has bytes from copied buffer (out of local_buffer boundaries)

Ex parameters to func (overwritten)

These bytes were not overwritten

```
(gdb) x/1i $eip
=> 0x80484d7 <func+29>: ret
(gdb) x/5xw $esp
0xffffce8c:     0x17161514      0x1b1a1918      0x1f1e1d1c      0xf7ffcce0
0xffffce9c:     0x00040000
(gdb) si
0x17161514 in ?? ()
```

Returned to execute an address indicated by those bytes from the overflown buffer located where the return address from func to main was present

Stack Overflow

- Memory corruption analysis
  - **memcpy** function (called from **func**) copied bytes beyond destination array boundaries (**local_buffer**)
  - When overflowing boundaries, stack is corrupted. Local variables from **func**, pushed EBP and **func** return address are overwritten
  - When returning from **func** to **main**, a corrupted return address from the stack is used to set EIP

Stack Overflow

- Is **memcpy** an insecure function?
- Are there any other functions that may cause an overflow?
- What is an underflow?

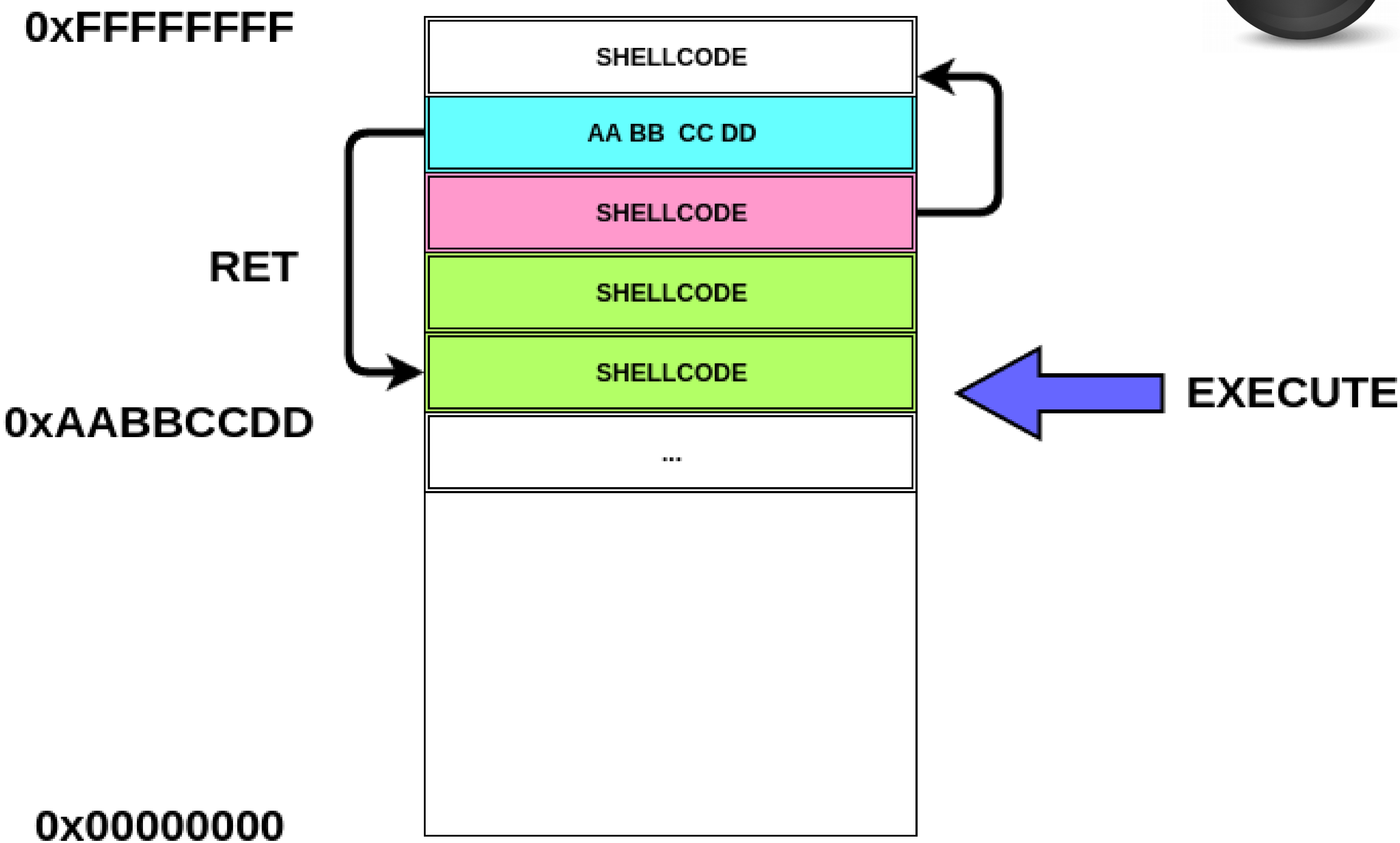
Stack Overflow

- Is **memcpy** an insecure function?
  - No but we need to make sure that:
    - There is enough space in destination buffer
    - There are enough bytes to copy in source buffer
- Are there any other functions that may cause an overflow?
  - Any function that copies memory (I.e. strcpy)
- What is an underflow?
  - An overflow but in the opposite direction

Stack Overflow

- Exploitability
  - Attacker controls EIP, and now?
  - If stack addresses were predictable (not-randomized) and stack executable, scenario is favorable to the attacker
    - Jump to execute in the stack
    - This is not possible anymore in modern operating systems, but may be in some embedded systems

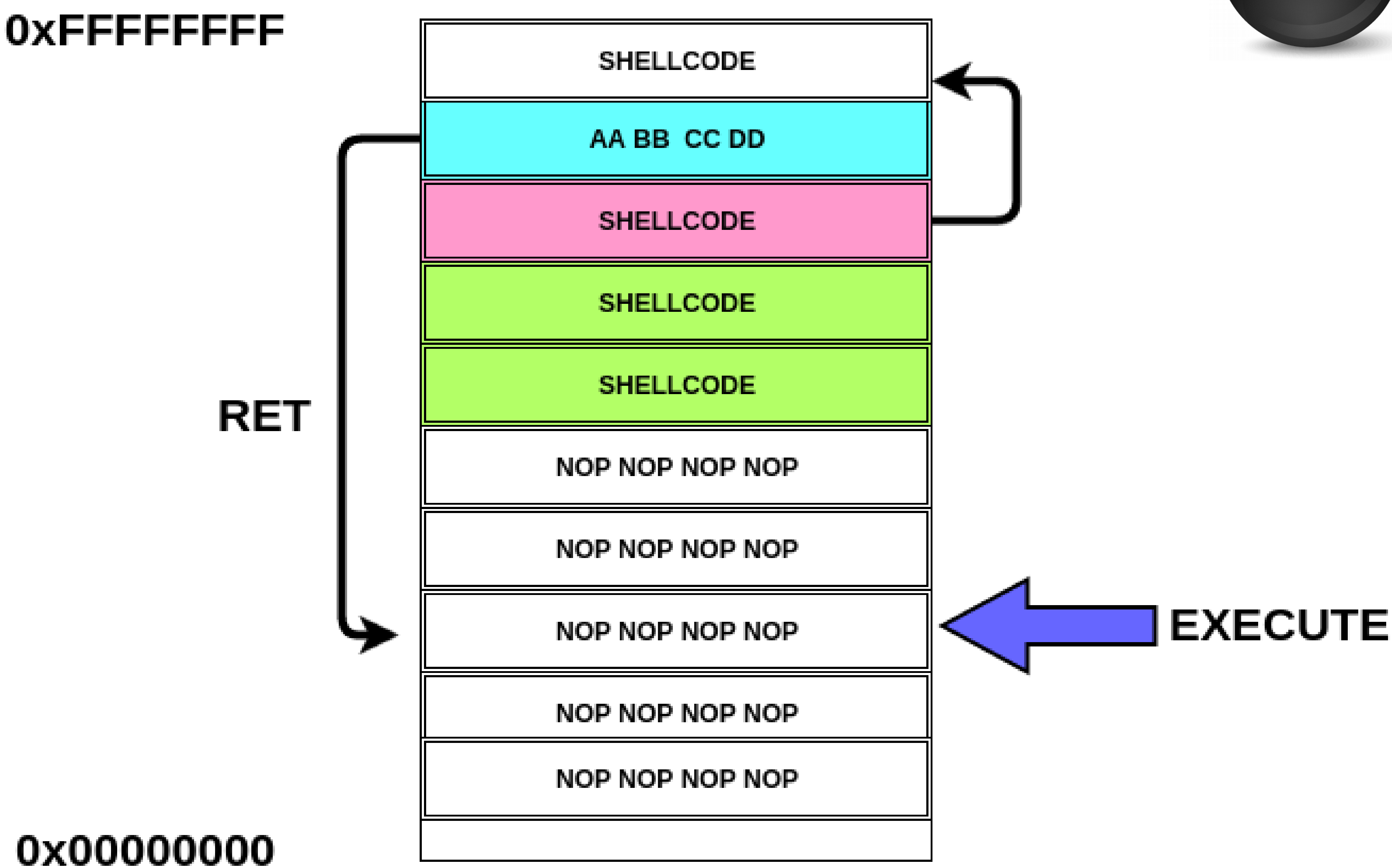
Stack Overflow



Stack Overflow

- Exploitability
  - If stack addresses were predictable within a certain range, a technique called NOP sled can be used to increase the probability of taking control of the execution

Stack Overflow



Stack Overflow

- With randomized stacks, a pointer leak is necessary
- With non-executable stacks, it’s necessary to use more advanced exploitation techniques like Return-Oriented-Programming (ROP)
- In addition to controlling EIP, it’s possible on some scenarios to take advantage of the corruption of local variables or other data present in the stack. Data attacks
- There can be read overflows useful to leak information

Stack Overflow

- Mitigations
  - Compilers: stack canary
  - Compilers: local variables reordering. Buffers are put together previous to canaries to avoid overflows that corrupt local variables
    - It’s not always possible. Buffers in structs
  - OS: randomized stack (unpredictable addresses)
  - OS: non-executable stacks (NX bit in x86)

Stack Overflow

- When a function protected by a stack canary is entered:

```
(gdb) x/3i $rip
=> 0x4005c5 <main+15>:  mov    %fs:0x28,%rax
   0x4005ce <main+24>:  mov    %rax,-0x8(%rbp)
   0x4005d2 <main+28>:  xor    %eax,%eax


(gdb) print/x $rax
$1 = 0xb998a401c0724300

(gdb) x/1xg ($rbp-0x8)
0x7fffffffdef8: 0xb998a401c0724300
```

Stack Overflow

- Stack canaries (user space)

```
4005c4:       48 8b 45 f8             mov    -0x8(%rbp),%rax
4005c8:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
4005cf:       00 00
4005d1:       74 05                   je     4005d8 <f+0x36>
4005d3:       e8 88 fe ff ff          callq  400460 <__stack_chk_fail@plt>
4005d8:       c9                      leaveq
4005d9:       c3                      retq
```

stack canary

```
(gdb) x/5xg $rsp
0x7fffffffdef0: 0x0000000000000000      0xb998a401c0724300
0x7fffffffdf00: 0x00007fffffffdf20      0x0000000000400587
0x7fffffffdf10: 0x00007fffffffe000
```

return address

Stack Overflow

- Stack canaries (user space)
  - %fs selector points to a structure in thread-local-storage (tls.h): Thread Control Block

```
typedef struct
{
  ...
  uintptr_t stack_guard;
  ...
} tcbhead_t;
```

Stack Overflow

- Stack canaries (user space)
  - In x86_64 %fs selector is set during initialization of the dynamic loader (*init_tls*) with syscall *arch_prctl*
  - Each thread sets a base address for the %fs selector. Then it’s used with an index
  - Stack canary is a number that changes in each execution
  - It’s pushed to the stack at the beginning of the function, and its integrity checked before returning
  - Thus, to overflow a buffer and return successfully, we have to know it -and replace it by itself-. It’s necessary to exploit an information leak vulnerability first

Stack Overflow

```
static void
security_init (void)
{
  /* Set up the stack checker's canary.  */
  uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard (_dl_random);
#ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
```

```
main [1384] [cores: 0]
  Thread #1 [main] 1384 [core: 0] (Suspend
    dl_main() at rtld.c:1,804 0x7ffff7ddad14
    _dl_sysdep_start() at dl-sysdep.c:253 0x7
    _dl_start_final() at rtld.c:414 0x7ffff7dd8
    _dl_start() at rtld.c:521 0x7ffff7dd8f68
    _start() at 0x7ffff7dd80b8
```

onsole | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

_remote [C/C++ Attach to Application] gdb (7.12.1)

```
lb) x/7i $rip
 0x7ffff7ddad14 <dl_main+6516>:      mov    0x222135(%rip),%rdx        # 0x7ffff7ffce50 <_dl_random>
 0x7ffff7ddad1b <dl_main+6523>:      mov    (%rdx),%rax
 0x7ffff7ddad1e <dl_main+6526>:      xor    %al,%al
 0x7ffff7ddad20 <dl_main+6528>:      mov    %rax,%fs:0x28
 0x7ffff7ddad29 <dl_main+6537>:      mov    0x8(%rdx),%rax
 0x7ffff7ddad2d <dl_main+6541>:      mov    %rax,%fs:0x30
 0x7ffff7ddad36 <dl_main+6550>:      movq   $0x0,0x22210f(%rip)        # 0x7ffff7ffce50 <_dl_random>
lb)
```

A random value is obtained for the canary: _dl_random

Canary lower byte is cleared

Canary is stored in Thread Control Block area.

elf/rtld.c (glibc)

Stack Overflow

- Task stack canary in Linux (kernel)
  - struct task_struct {

```
    ...
    unsigned long stack_canary;
    ...
}
```

include/linux/sched.h

Loaded in dup_task_struct function (kernel/fork.c):

```
tsk->stack_canary = get_random_long();
```

Stack Overflow

- Task stack canaries (Linux kernel)
  - In x86_64 GCC uses %gs selector with offset 0x28, that corresponds to “percpu storage area” in kernel, to read the stack canary in run time
  - When switching tasks, kernel has to update %gs:0x28 area with the stack canary from the new task

Stack Overflow

```
/*
 * %rdi: prev task
 * %rsi: next task
 */
ENTRY(__switch_to_asm)

   ...

#ifdef CONFIG_CC_STACKPROTECTOR
   movq TASK_stack_canary(%rsi), %rbx
   movq %rbx, PER_CPU_VAR(irq_stack_union)
+stack_canary_offset
#endif
```

arch/x86/entry/entry_64.S

Stack Overflow

- Stack canaries (Linux kernel)

```
ffffffffa0008033:   mov     $0xffffffffa0551028,%rdi
ffffffffa000803a:   callq   0xffffffff811bf2b2 <printk>
ffffffffa000803f:   mov     -0x8(%rbp),%rdx
ffffffffa0008043:   xor     %gs:0x28,%rdx
ffffffffa000804c:   je      0xffffffffa0008053
ffffffffa000804e:   callq   0xffffffff810a1d90 <__stack_chk_fail>
ffffffffa0008053:   xor     %eax,%eax
ffffffffa0008055:   leaveq
ffffffffa0008056:   retq
```

Tasks | Problems | Executables | Memory | Debugger Console

kernel_dev Default [C/C++ Attach to Application] gdb (7.12.1)

```
(gdb) x/4xg $rbp-0x10
0xffffc90000ad7c90:    0x000201009a0d58f4    0x000000009a0d58f4
0xffffc90000ad7ca0:    0xffffc90000ad7d18    0xffffffff81002190
(gdb) print/x $rdx
$4 = 0x9a0d58f4
```

Canary

Return address

Demo 8.1

Stack overflow in kernel space

Buffer Overflows

- Memory overflows can occur in the heap
  - More difficult to exploit
  - Object data allocated in the heap can be corrupted (data attacks)
  - Pointers to functions or vtables (that contain pointers to functions) can be overwritten
  - Dynamic memory allocator structures can be corrupted, leading to memory read/write primitives

Integer Overflow

- Overflow in unsigned data types (Linux x86_64):
  - unsigned char: 1 byte (0x00… 0xFF)
  - unsigned short: 2 bytes (0x00 … 0xFFFF)
  - unsigned int: 4 bytes (0x00 … 0xFFFFFFFF)

```
unsigned long a = 0xFFFFFFFFFFFFFFFE;

a = a + 0x5;

printf("a: %lu\n", a);
```

Integer Overflow

```
(gdb) x/1i $rip
=> 0x400506 <main+16>:  addq   $0x5,-0x8(%rbp)
(gdb) print $eflags
$1 = [ PF IF ]
(gdb) si
9               printf("a: %lu\n", a);
(gdb) print $eflags
$2 = [ CF PF AF IF ]
```

Operation result is 0x3 and CPU state register is modified when this type of overflow occurs, turning on the *carry* flag

Integer Overflow

- Overflow in signed data types (x86_64):
  - Char - 1 byte: **0** 0 0 0 0 0 0 0
    - First bit: sign
    - Can represent: -128 … -1, 0, 1 … 127

```
long a = 0x7FFFFFFFFFFFFFFF;

printf("a (before): %ld\n", a);

a = a + 0x1;

printf("a (after): %ld\n", a);
```

Integer Overflow

```
(gdb) x/1i $rip
=> 0x400522 <main+44>:  addq   $0x1,-0x8(%rbp)
(gdb) print $eflags
$1 = [ PF IF ]
(gdb) si
11          printf("a (after): %ld\n", a);
(gdb) print $eflags
$2 = [ PF AF SF IF OF ]
```

Operation result is -9223372036854775808, and CPU state register is modified when this type of overflow occurs, turning on the *overflow* flag

Integer Overflow

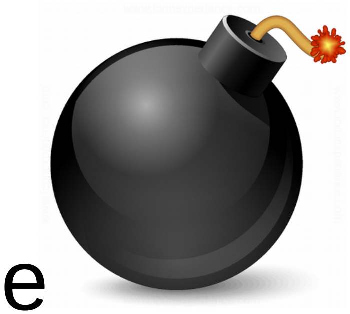
- Note: OF flag is turned on when the sign bit is modified in the register. If the compiler uses a larger register to operate, this does not happen (but the overflow yes). I.e.:

```
char a = 0x7F;

printf("a (before): %d\n", a);

a = a + 0x1;

printf("a (after): %d\n", a);
```

Integer Overflow

```
(gdb) x/2i $rip
=> 0x40051b <main+37>:  add    $0x1,%eax
   0x40051e <main+40>:  mov    %al,-0x1(%rbp)
(gdb) print $eflags
$1 = [ PF IF ]
(gdb) si
0x000000000040051e      9           a = a + 0x1;
(gdb) print $eflags
$2 = [ AF IF ]
```

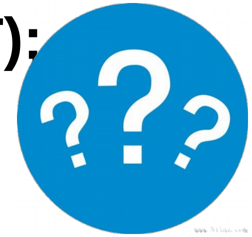
Operation result is -128, and the *overflow* flag is not turned on



Integer Overflow

- Why are integer overflows relevant from the security point of view?

```
#define HEADER_LENGTH 15
#define MAX_BUFFER_LIMIT (112 + HEADER_LENGTH)
const char global_buffer[MAX_BUFFER_LIMIT] = { 0x0 };
int main(void) {
   char user_data_bytes_requested = 127; // User input: 127 data bytes
   char total_data_requested = user_data_bytes_requested +
HEADER_LENGTH;
   if (total_data_requested > MAX_BUFFER_LIMIT) {
      goto fail;
   }
   printf("total_data_requested: %u - buffer size: %u\n",
         (unsigned int)total_data_requested, MAX_BUFFER_LIMIT);
   return 0;
fail:
   return -1;
}
```



Integer Overflow

```
char total_data_requested =
user_data_bytes_requested + HEADER_LENGTH;
```

- User requested 127 bytes, that when added to the header length are 142 bytes in total
- However, that value generates an overflow when stored in a variable of char type (that can only store values in the range -128 … 127)
- Real stored value in the variable is -114

Integer Overflow

```
if (total_data_requested > MAX_BUFFER_LIMIT) {
    goto fail;
}
```

- Comparison returns false because -114 < 127. Thus, execution continues instead of failing
- Now, then casting “total_data_requested” to unsigned we have a value of 142 to operate on a buffer of 127
  - If a copy is made, a memory overflow will occur
  - If a read is made, information will be leaked
- If this is combined with a cast to a larger data type with sign extension, delta between the size of the buffer and the value to be used would be even larger

Integer Overflow

- Why are integer overflows relevant from the security point of view?

```c
#define HEADER_SIZE 15U

int main(void) {

    unsigned char user_data_size = 250U;

    unsigned char buffer_size = user_data_size + HEADER_SIZE;

    char* buffer = (char*)malloc(buffer_size);

    printf("buffer_size: %u\n", buffer_size);

    return 0;
}
```

Integer Overflow

unsigned char buffer_size = user_data_size + HEADER_SIZE;

- That assignment generates an overflow because buffer_size can store up to value 255. Value 265 ends up being 9
- Thus, 9 bytes of memory will be allocated, being “user_data_size” 250. That will generate a memory overflow
- In some scenarios, a malloc that returns 0 can be used to write the page that starts with virtual address 0. In modern operating systems, this page cannot be mapped

Integer Overflow

- Operators that can cause overflows:

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
+	Yes	-=	Yes	<<	Yes	<	No
-	Yes	*=	Yes	>>	No	>	No
*	Yes	/=	Yes	&	No	>=	No
/	Yes	%=	Yes	|	No	<=	No
%	Yes	<<=	Yes	^	No	==	No
++	Yes	>>=	No	~	No	!=	No
--	Yes	&=	No	!	No	&&	No
=	No	|=	No	un +	No	||	No
+=	Yes	^=	No	un -	Yes	?:	No

Table from “Secure Coding in C and C++”

Integer Overflow

- How can it be prevented?
  - Use unsigned data types to represent sizes. size_t is as a standard data type for that (generally with a size equal to the size of a pointer)
  - Avoid implicit casting and downcasting. Downcasting can, in addition to data truncation, modify the sign value
  - In case of upcasting, be careful with sign extension (followed by an unsigned cast)

Integer Overflow

- How can it be prevented?
  - Use data types larger than the maximum value to be represented. I.e. if 2 unsigned chars are added, 510 is the maximum value that can be represented. An unsigned short data type can store that value (and any value up to 65535)
  - Include checks before of after operation if applies. Is the addition result less than any of the addends? Constants like INT_MAX, etc. defined in “limits.h” can be used
    - Code has to remain legible
    - Avoid performance impact in release mode

Integer Overflow

- How can it be prevented?
  - Be careful with multiplatform code: different platforms may have different sizes for the same data type (I.e.: long is 8 bytes in Linux x86_64 and 4 in Windows x86_64). Thus, use standard data types as those available in “stdint.h”:
    - uint8_t
    - uint32_t
    - int32_t
    - …
- In addition to overflows, there can be underflows or reverse wrap-arounds

Integer Overflow

- Data type sizes for most common platforms:

Data Type	8086	x86-32	64-Bit Windows	SPARC-64	ARM-32	Alpha	64-Bit Linux, FreeBSD, NetBSD, and OpenBSD
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32
long	32	32	32	64	32	64	64
long long	N/A	64	64	64	64	64	64
pointer	16/32	32	64	64	32	64	64

Table from “Secure Coding in C and C++”

Signed comparisons

- What’s the security problem here?

```
#define MAX_ALLOCATION_SIZE 0xFF
int main(void) {
    // User input.
    int user_requested_buffer_size = -1;
    if (user_requested_buffer_size > MAX_ALLOCATION_SIZE) {
        goto fail;
    }
    char* buff = (char*)malloc(user_requested_buffer_size);
    printf("user_requested_buffer_size: %u\n",
user_requested_buffer_size);
    printf("buff: %p\n", buff);

    return 0;
fail:
    return -1;
}
```

Signed comparisons

- What's the security problem here?

```
(gdb) x/20i $rip
=> 0x40054e <main+8>:    movl   $0xffffffff,-0x4(%rbp)
   0x400555 <main+15>:   cmpl   $0xff,-0x4(%rbp)
   0x40055c <main+22>:   jg     0x4005a0 <main+90>
   0x40055e <main+24>:   mov    -0x4(%rbp),%eax
   0x400561 <main+27>:   cltq
   0x400563 <main+29>:   mov    %rax,%rdi
   0x400566 <main+32>:   callq  0x400440 <malloc@plt>
   0x40056b <main+37>:   mov    %rax,-0x10(%rbp)
   0x40056f <main+41>:   mov    -0x4(%rbp),%eax
```

Signed comparison (jump-greater): 2 signed integers are being compared. If it were unsigned, there would be a jump-above

"malloc" will consider this parameter as unsigned

Signed comparisons

- When trying to allocate a huge amount of memory (0xFF...FF), malloc fails returning a NULL pointer. If malloc failure were not properly handled, subsequent operations may corrupt memory
- A huge memory allocation may cause a Denial Of Service and can facilitate heap sprays
- How can this be prevented?
  - Avoid or analyze implicit casting
  - Analyze the comparison sign (signed vs unsigned)
  - Use unsigned values to represent quantities or sizes

Signed comparisons

- And now?

```
#define MAX_ALLOCATION_SIZE 0xFFU
int main(void) {
   // User input.
   unsigned int user_requested_buffer_size = -1;
   if (user_requested_buffer_size > MAX_ALLOCATION_SIZE) {
      goto fail;
   }
   char* buff = (char*)malloc(user_requested_buffer_size);
   printf("user_requested_buffer_size: %u\n",
user_requested_buffer_size);
   printf("buff: %p\n", buff);

   return 0;
fail:
   return -1;
}
```

Signed comparisons

- And now?

```
(gdb) x/10i $rip
=> 0x40054e <main+8>:   movl   $0xffffffff,-0x4(%rbp)
   0x400555 <main+15>:  cmpl   $0xff,-0x4(%rbp)
   0x40055c <main+22>:  ja     0x40059e <main+88>
   0x40055e <main+24>:  mov    -0x4(%rbp),%eax
   0x400561 <main+27>:  mov    %rax,%rdi
   0x400564 <main+30>:  callq  0x400440 <malloc@plt>
```

Unsigned comparison (jump-above). Ends up jumping

Integer Overflow

- Why compilers do not protect the developer from this scenarios?
  - In the C standard, overflows and underflows are undefined behavior
  - Compilers optimize for performance, and do not add checks overhead (unnecessary for most cases)
  - Avoiding undefined behaviors is a responsibility of the developer

Lab

8.1: Stack overflow in user space

References

- Secure Coding in C and C++. Robert C. Seacord.