Ingeniería Inversa
Clase 8

Exploit Writing I
Stack and Integer Overflow

martin.uy
Open by default.

Stack Overflow

- ¿Qué es un stack? (x86)
  - Área de memoria donde se almacenan variables locales, parámetros de funciones, registros salvados, direcciones de retorno (en llamadas a funciones) y memoria dinámicamente alocada en stack
  - Cada thread tiene 2 stacks:
    - Stack en espacio de usuario
    - Stack en espacio de kernel (cuando el thread ejecuta una *syscall*)
    - ¿Por qué?

Stack Overflow

- ¿Qué es un stack? (x86)
  - El stack no es compartido entre threads: no hay problemas de concurrencia para los datos allí almacenados
  - Los stacks de usuario generalmente están en memoria con direcciones virtuales altas y, en x86 / x86_64, crecen hacia direcciones más virtuales más bajas
  - El tope del stack es apuntado por el registro ESP (RSP en x86_64)
    - Que un stack crezca no significa necesariamente que se aloque más memoria: la memoria puede ya estar alocada y solamente se cambia el valor del registro que apunta a su tope
  - Los stacks tienen una capacidad máxima definida al momento de crear el thread (ej. 2MB para stacks de usuario)

Stack Overflow

Punto de entrada para syscalls (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 → Puntero al stack en espacio de usuario

(gdb) print $rsp
$2 = (void *) 0xffffc90000b40000 → Puntero al stack en espacio de kernel

Stack Overflow

- Stacks en Linux (kernel)
  - sys_clone (creación de un thread/proceso)
  - _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 en Linux (kernel)
  - struct task_struct {

```
    ...
    void *stack;
    ...
}
```

include/linux/sched.h

Stack Overflow

- Breakpoint en entrada a syscall (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

- Uso del stack
  - Instrucciones que implícitamente modifican el stack (x86 / x86_64)
    - PUSH, POP, PUSHAD, POPAD, CALL, LEAVE, RET, RET n
    - La cantidad de bytes afectados por cada una de estas operaciones está relacionada al tamaño natural de la arquitectura. Por ejemplo, en x86_64 un CALL va a pushear 8 bytes al stack con la dirección de retorno
  - Instrucciones que explícitamente modifican el stack
    - Ej. SUB ESP, 10h

Stack Overflow

- Ejemplos

```
; 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

- Ejemplos

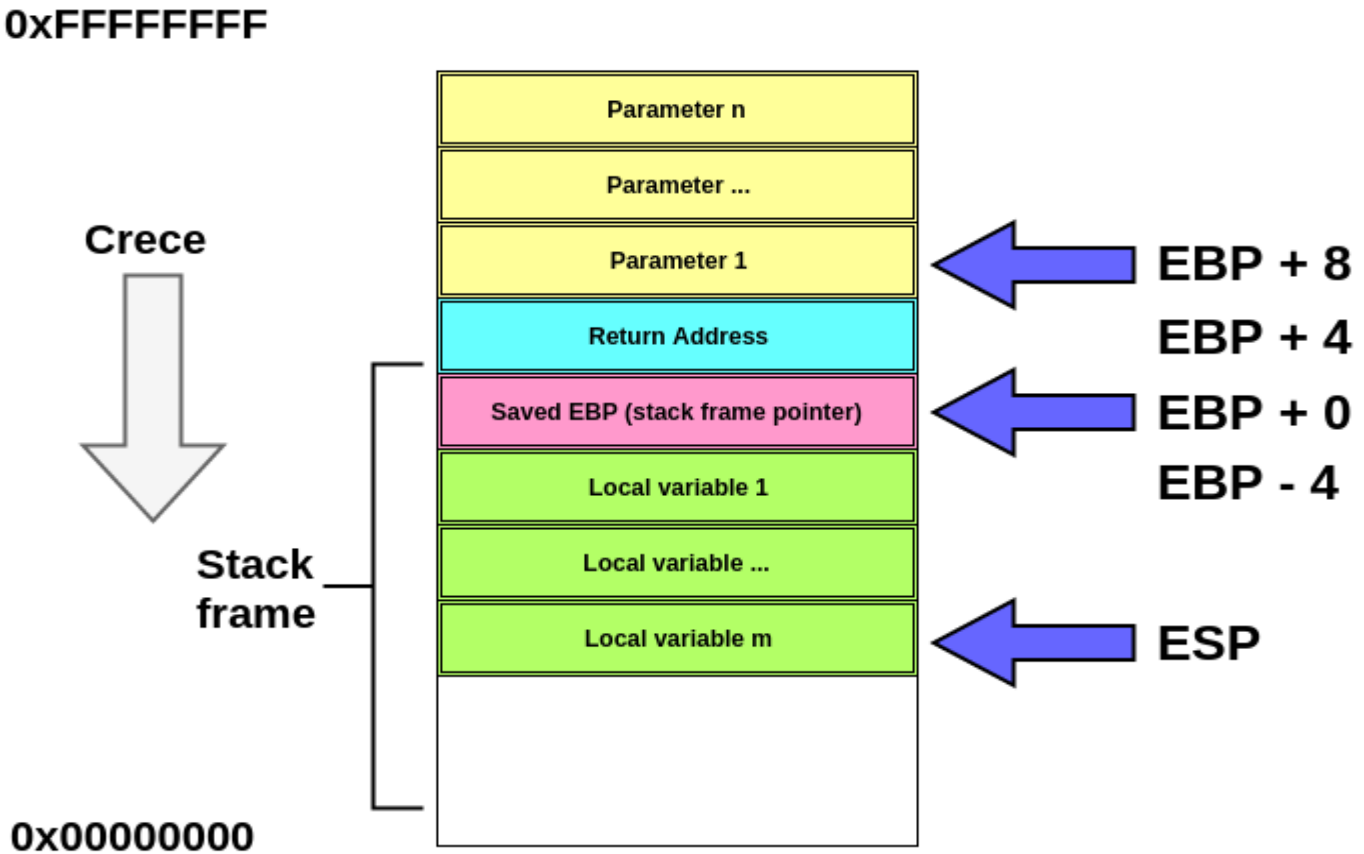
```
(gdb) x/1i $rip
=> 0x555555555670 <main>:       push   %r15
(gdb) set $r15 = 0x4141414141414141
(gdb) x/1xg $rsp
0x7fffffffdfd8: 0x00007ffff7a31401
(gdb) si
0x0000555555555672 in main ()
(gdb) x/2xg $rsp
0x7fffffffdfd0: 0x4141414141414141      0x00007ffff7a31401
(gdb)
```

Stack Overflow

- Stack overflow es un tipo de vulnerabilidad causada por corrupción de memoria
- Independiente del sistema operativo y puede darse en diferentes arquitecturas. Vamos a estudiarla en x86/x86_64
- Permite tomar el control del instruction pointer y/o modificar variables locales de una función (ataque sobre datos)
- Esto es posible porque los datos (escribibles) se entremezclan con punteros a código en un mismo stack:
  - direcciones de retorno
  - punteros a vtables (que contienen punteros a código)
  - punteros a handlers de excepciones
- Vulnerabilidad descripta en el paper “Smashing The Stack For Fun and Profit” en el año 1996, por Elias Levy

Stack Overflow

- Application Binary Interface para CALLs (x86)



Stack

1 en espacio de usuario por thread

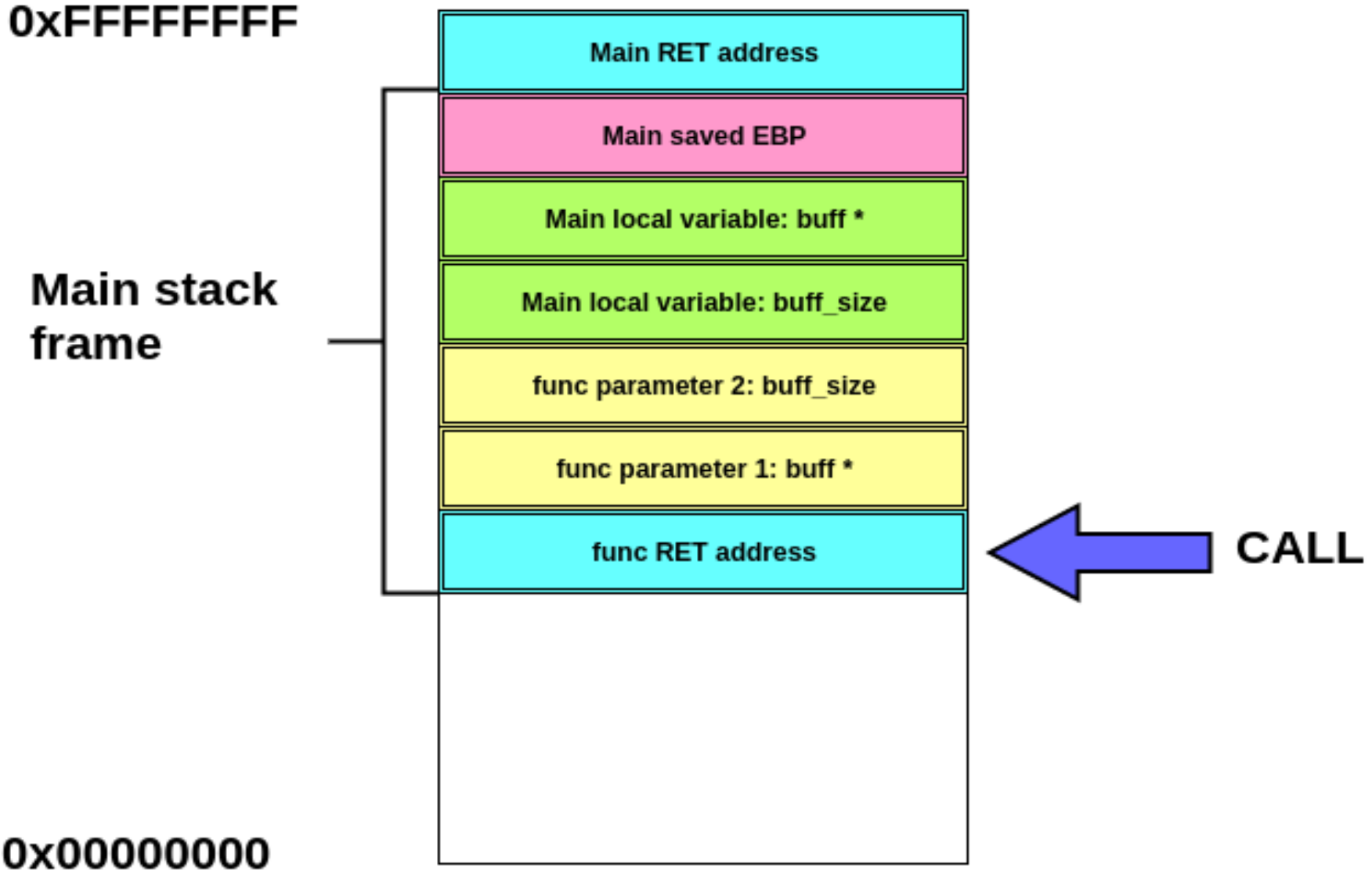
Stack Overflow

- ¿Dónde está la vulnerabilidad?

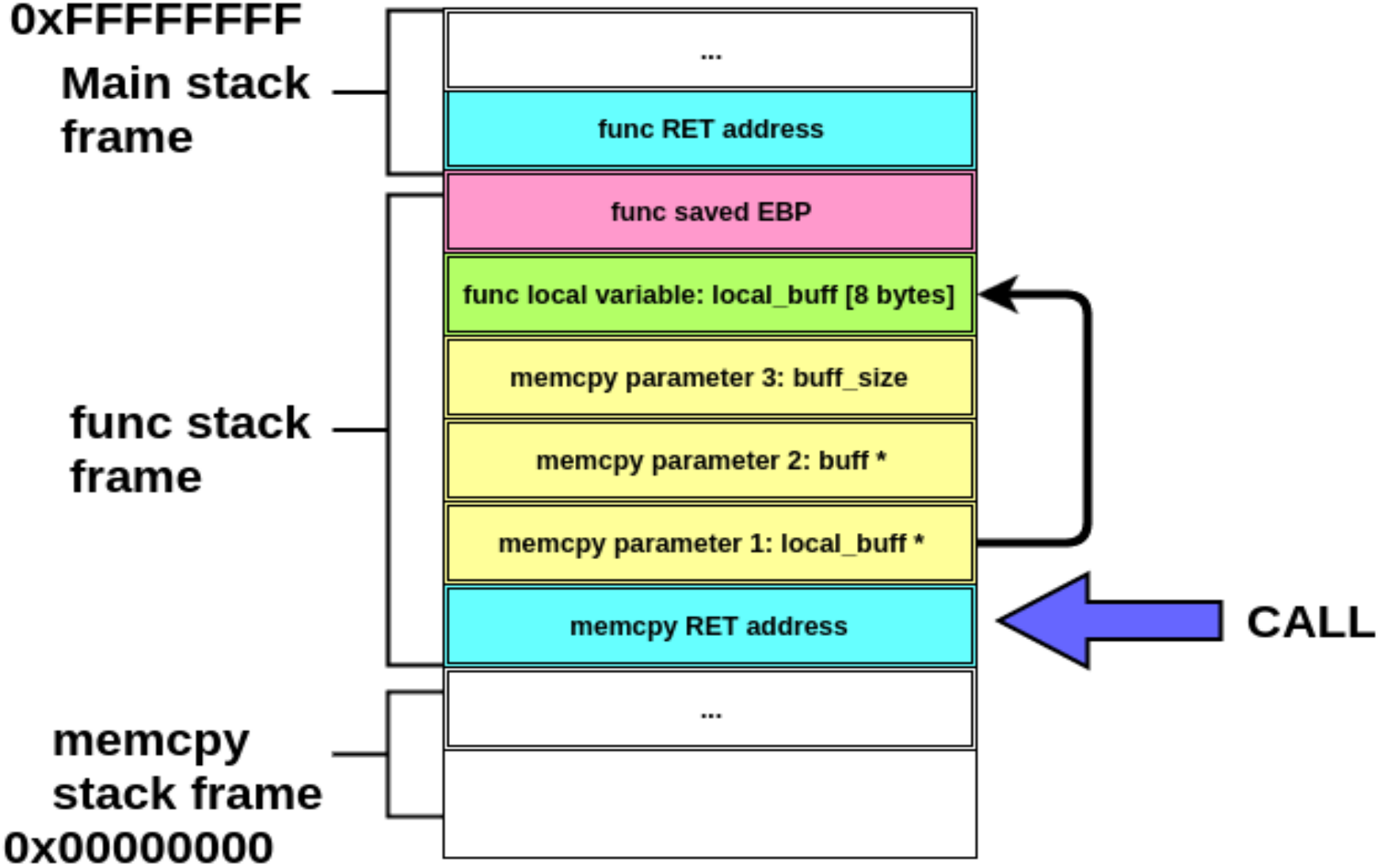
```
void main(){
    ...
    func(buff, buff_size);
}

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

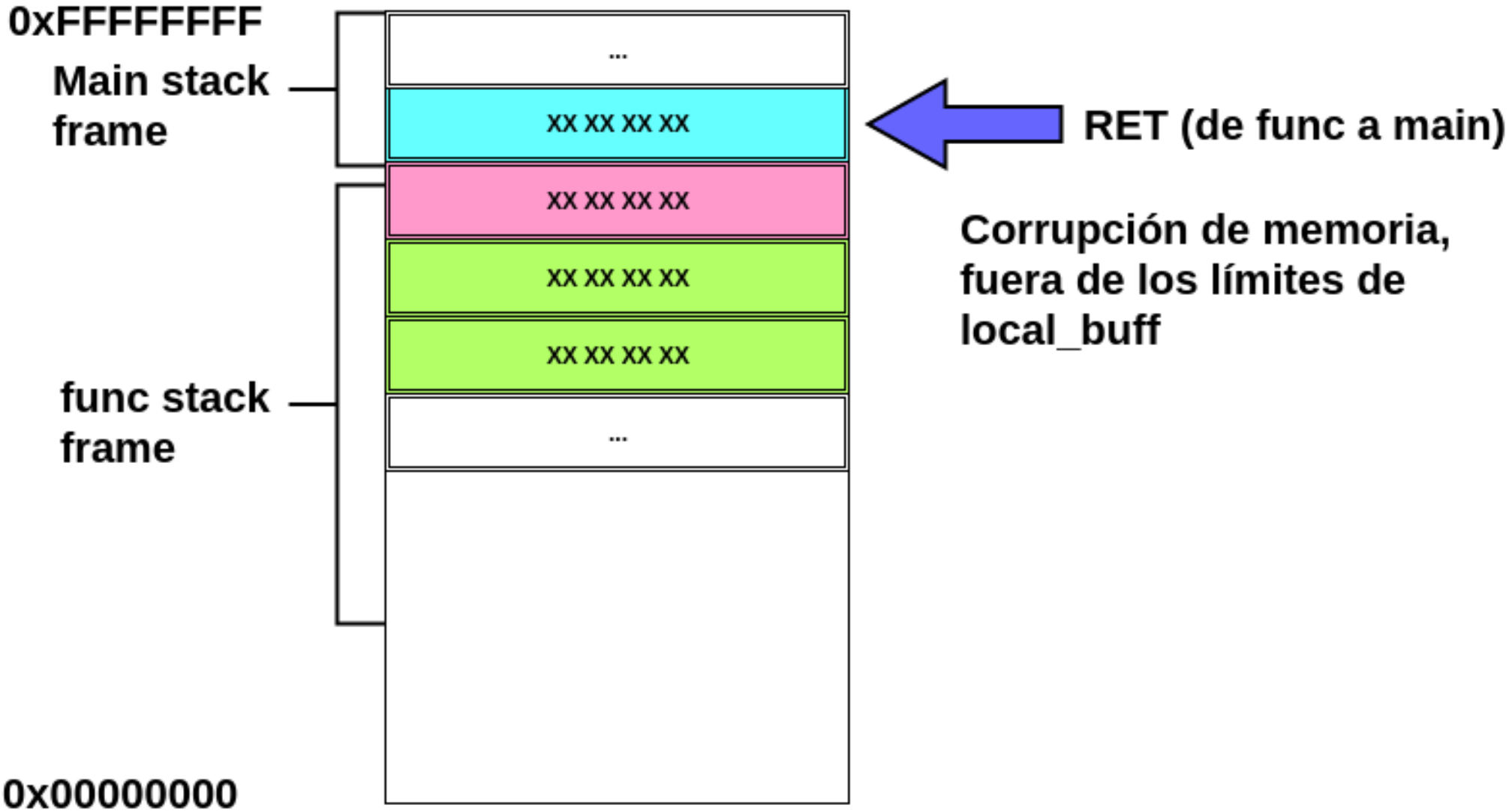
Stack Overflow



Stack Overflow



Stack Overflow



Stack Overflow

Puntero a buff (variable local de main, en 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
```

Parámetros para “func” (en stack)

buff en stack: 32 bytes, desde 0x00 a 0x1F

Stack Overflow

Dirección de retorno a main (en stack)

Parámetros para “func”: puntero a buff y buff_size (en 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

Buffer de destino para memcpy. Capacidad: 8 bytes. Variable local de func: local_buffer (stack)

Source buffer para memcpy (puntero a buff)

Cantidad de bytes a copiar (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
```

Buffer de destino de memcpy. 8 bytes (basura del stack en este momento)

Pushed ebp al entrar a func

Return address a main (al salir de func)

Parámetros al llamar a func

Stack Overflow

Ex dirección de retorno desde func a main. Ahora tiene bytes del buffer copiado (fuera de los límites de local_buffer)

Ex parámetros a func (sobreescritos)

Estos bytes se salvaron de la sobreescritura

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

Se retornó a ejecutar la dirección indicada por los bytes del buffer desbordado que se ubicaron donde estaba la dirección de retorno de func a main

Stack Overflow

- Análisis de la corrupción de memoria
  - La función **memcpy** (llamada desde **func**) copió bytes más allá de los límites del array de destino (**local_buffer**)
  - Al pasar los límites, se corrompe el stack. Las variables locales de **func**, EBP pusheado y la dirección de retorno de **func** son sobreescritos
  - Cuando se retorna desde **func** a **main**, se toma la dirección de retorno corrupta del stack para setear EIP

Stack Overflow

- ¿Es **memcpy** una función insegura?
- ¿Hay otras funciones que puedan generar un overflow?
- ¿Qué es un underflow?

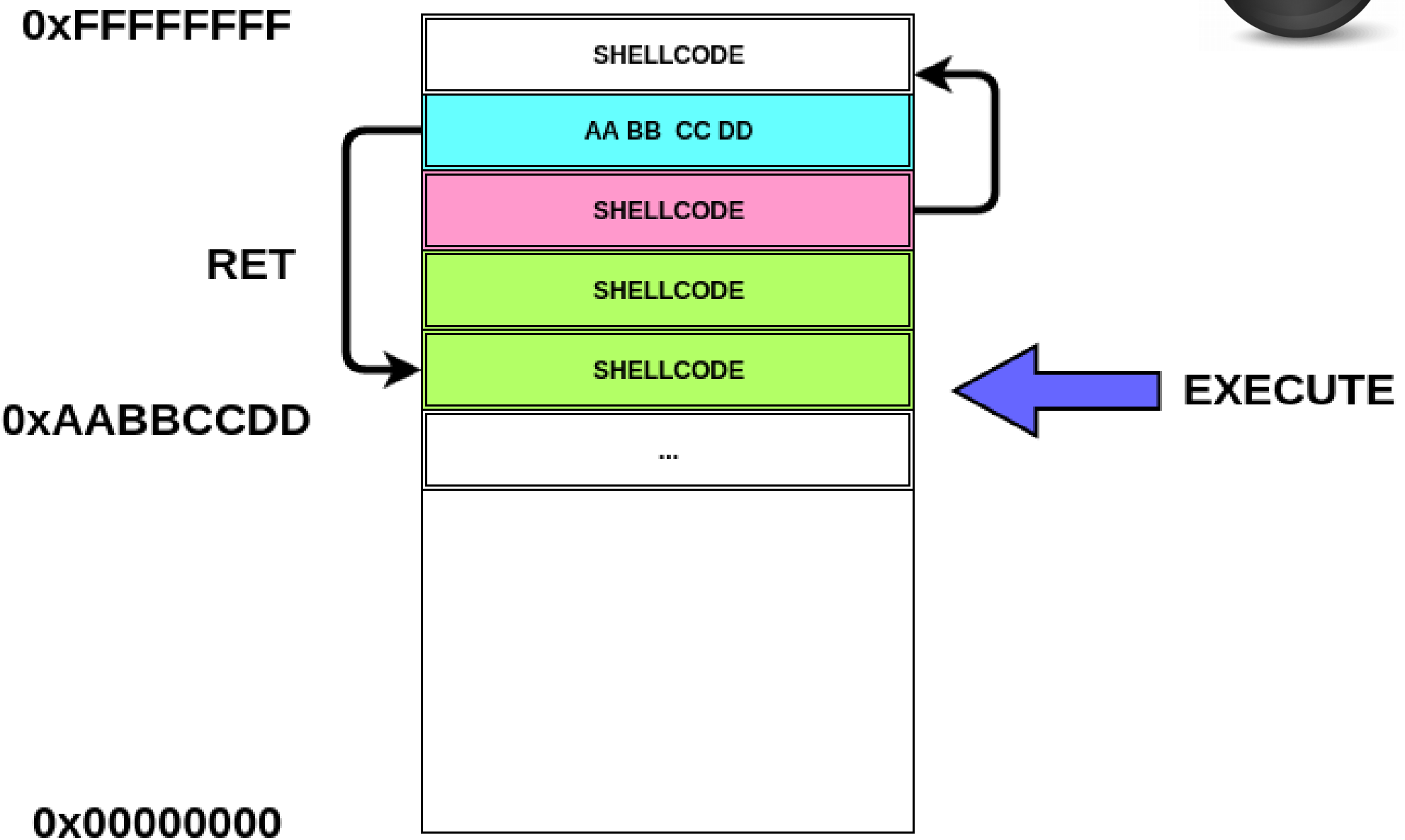
Stack Overflow

- ¿Es **memcpy** una función insegura?
  - No pero debemos asegurarnos que:
    - Haya espacio en el buffer de destino
    - Hayan bytes para copiar en el buffer de origen
- ¿Hay otras funciones que puedan generar un overflow?
  - Cualquier función que copie memoria (ej. strcpy)
- ¿Qué es un underflow?
  - Un desborde pero en la dirección opuesta

Stack Overflow

- Explotabilidad
  - El atacante controla EIP, ¿y ahora?
  - Si las direcciones virtuales del stack son predecibles (no-randomizadas) y el stack es ejecutable, el escenario es favorable al atacante
    - Saltar a ejecutar al stack
    - Esto ya no es posible en sistemas operativos modernos, pero aún puede encontrarse en sistemas embebidos

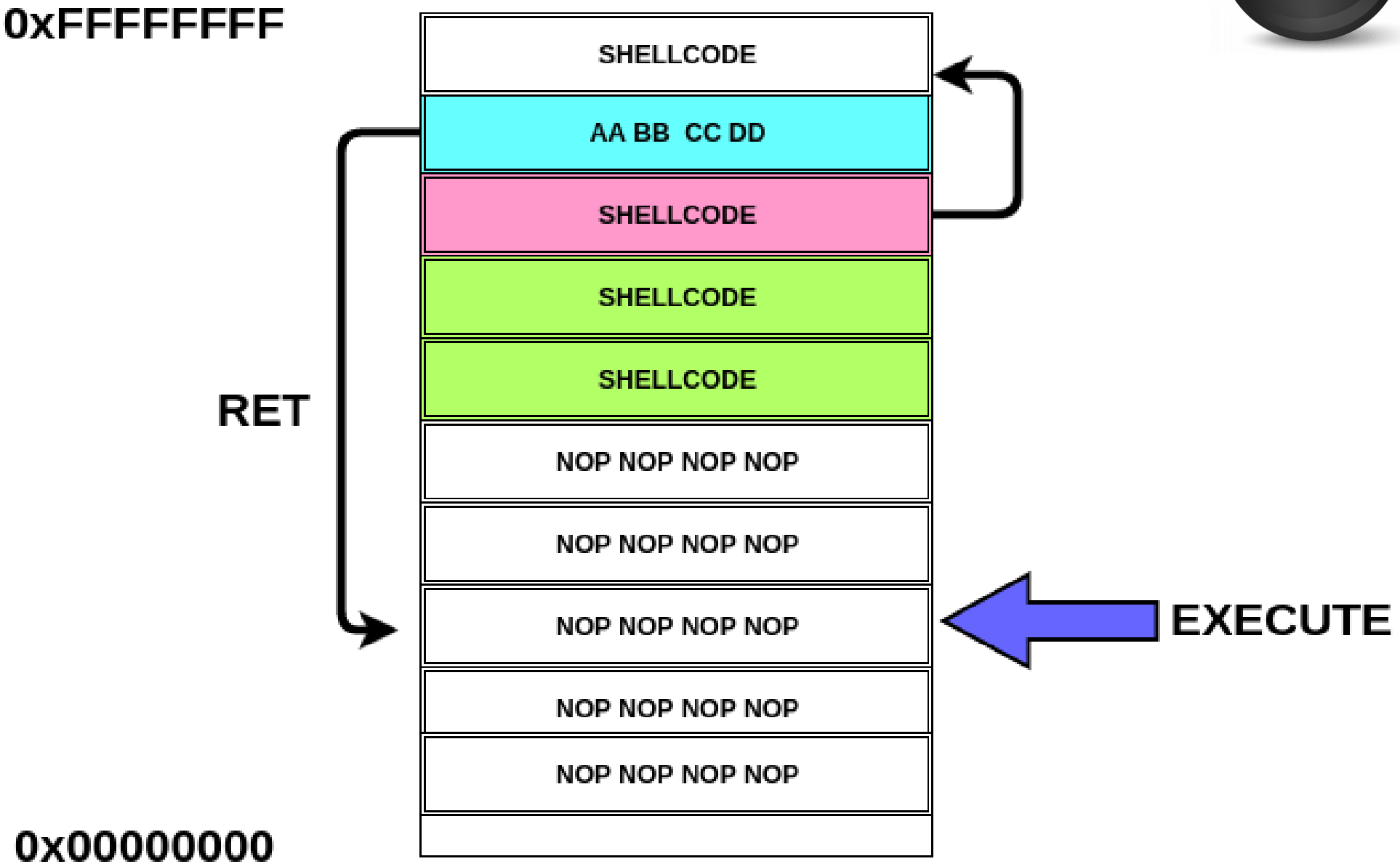
Stack Overflow



Stack Overflow

- Explotabilidad
  - Si el stack es predecible con un cierto margen, se utiliza la técnica de NOP sled para aumentar la probabilidad de controlar la ejecución

Stack Overflow



Stack Overflow

- Con stacks randomizados, sería necesario un pointer leak
- Con stacks no ejecutables, es necesario aplicar técnicas de explotación más sofisticadas como Return-Oriented-Programming (ROP)
- Además de controlar EIP, es posible en ciertos escenarios sacar ventaja de la corrupción de las variables locales u otros datos presentes en el stack. Ataques sobre datos
- Pueden haber overflows de lectura que a veces sirven para leakear información

Stack Overflow

- Mitigaciones
  - Compiladores: stack canary
  - Compiladores: reordenamiento de las variables locales. Los buffers van juntos a continuación del canary para evitar que se corrompan otras variables locales
    - No siempre es posible. Buffers en structs
  - OS: stack randomizado (direcciones impredecibles)
  - OS: stack no ejecutable (NX bit en x86)

Stack Overflow

- Cuando se entra a una función protegida por stack canary:

```
(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 (espacio de usuario)

```
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: 0x00007ffffffffe000
```

return address

Stack Overflow

- Stack canaries (espacio de usuario)
  - El selector %fs apunta a una estructura en thread-local-storage (tls.h): Thread Control Block

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

Stack Overflow

- Stack canaries (espacio de usuario)
  - En x86_64 el selector %fs es seteado durante la inicialización del dynamic loader (*init_tls*) con la syscall *arch_prctl*
  - Cada thread setea una base address para el selector %fs. Luego se utiliza junto a un índice
  - El stack canary es un número que cambia en cada ejecución
  - Es pusheado al stack al entrar a la función y su integridad se chequea al salir
  - Por lo tanto, para desbordar el buffer y retornar satisfactoriamente, deberíamos conocerlo previamente y reemplazarlo por si mismo. Se necesitaría explotar un leak de información primero

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 0x
_dl_start_final() at rtld.c:414 0x7ffff7dd8
_dl_start() at rtld.c:521 0x7ffff7dd8f68
_start() at 0x7ffff7dd80b8

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

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

```
(gdb) 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>
(gdb)
```

Se obtiene un valor aleatorio para el canary: _dl_random

elf/rtld.c (glibc)

Se limpia el byte más bajo del canary

Se almacena el canary en el área Thread Control Block

Stack Overflow

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

    ...

    unsigned long stack_canary;

    ...

  } **include/linux/sched.h**

Cargado en la función dup_task_struct (kernel/fork.c):

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

Stack Overflow

- Task stack canaries (Linux kernel)
  - En x86_64 GCC utiliza el selector %gs con offset 0x28, que corresponde al “percpu storage area” en kernel, para acceder al stack canary durante la ejecución
  - Al switchear de task, el kernel necesita actualizar el área %gs:0x28 con el stack canary de la nueva 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:     0x00201009a0d58f4       0x00000009a0d58f4
0xffffc90000ad7ca0:     0xffffc90000ad7d18      0xffffffff81002190
(gdb) print/x $rdx
$4 = 0x9a0d58f4
```

Canary

Return address

Demo 8.1

Stack overflow en espacio de kernel

Buffer Overflows

- Los overflows de memoria pueden ocurrir en el heap
  - Más difíciles de explotar
  - Se pueden corromper datos de objetos alocados en el heap (ataques sobre datos)
  - Se pueden sobreescribir punteros a funciones o a vtables (que tienen luego punteros a funciones)
  - Se pueden corromper las estructuras del alocador dinámico de memoria y lograr primitivas de tipo lectura/escritura en memoria

Integer Overflow

- Overflow en tipos de datos no signados (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 ]
```

El resultado de la operación es 0x3 y se modifica el registro de estado del CPU cuando ocurre un overflow de este tipo, activándose el flag de *carry*

Integer Overflow

- Overflow en tipo de datos signados (x86_64):
  - Char - 1 byte: **0** 0 0 0 0 0 0 0
    - Primer bit: signo
    - Se puede representar: -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 ]
```

El resultado de la operación es -9223372036854775808, y se modifica el registro de estado del CPU cuando ocurre un overflow de este tipo, activándose el flag de *overflow*

Integer Overflow

- Nota: el flag OF se activa cuando se modifica el bit de signo del registro. Si el compilador utiliza un registro más grande para operar, esto no sucede (pero el overflow sí). Ej.:

```
char a = 0x7F;

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

a = a + 0x1;

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

Integer Overflow

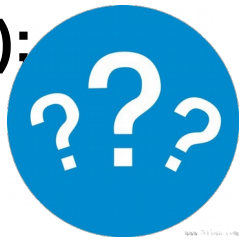
```
(gdb) x/2i $rip
=> 0x40051b <main+37>:  add    $0x1,%eax
   0x40051e <main+40>:  mov    %al,-0x1(%rbp)
(gdb) print $eflags
$1 = [ PF IF ]
(gdb) si
0x000000000040051e      9               a = a + 0x1;
(gdb) print $eflags
$2 = [ AF IF ]
```

El resultado de la operación es -128, y no se activa el flag de *overflow*

Integer Overflow

- ¿Por qué son relevantes los integer overflows desde el punto de vista de la seguridad?

```
#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;
```

- El usuario solicitó 127 bytes, que sumados al header son 142 bytes totales
- Sin embargo, ese valor genera un overflow al ser almacenado en una variable de tipo char (que solo almacena valores en el rango -128 … 127)
- El valor real almacenado en la variable es -114

Integer Overflow

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

- Esa comparación es falsa porque -114 < 127. Por lo tanto, se continúa la ejecución en lugar de fallar
- Ahora, al castear “total_data_requested” a unsigned tenemos un valor de 142 para operar sobre un buffer de 127.
  - Si se hace una copia, vamos a hacer un overflow de memoria.
  - Si se hace una lectura, vamos a leakear información
- Si combinamos esto con un cast a un tipo de dato mayor (con extensión de signo), el delta entre el tamaño del buffer y el valor a usar es aún mayor

Integer Overflow

- ¿Por qué son relevantes los integer overflows desde el punto de vista de la seguridad?

```
#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;

- Esa asignación genera un overflow porque buffer_size puede guardar hasta el valor 255. El valor 265 termina siendo en realidad 9
- Por lo tanto, se van a alocar 9 bytes de memoria siendo “user_data_size” 250. Eso va a generar un memory overflow
- En algunos escenarios, puede triggerearse un malloc que retorne 0 e intentar escribir la página con la dirección virtual 0. En sistemas operativos modernos, esa página no puede ser mapeada

Integer Overflow

- Operadores que pueden causar 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

Tabla extraída de “Secure Coding in C and C++”

Integer Overflow

- ¿Cómo se pueden prevenir?
  - Utilizar tipos de datos no signados para representar tamaños. size_t es un tipo de dato estandarizado para eso (que generalmente tiene un tamaño igual al de un puntero)
  - Evitar casteos implícitos y downcasting. El downcasting puede, además de trucar el valor, modificar el signo
  - En caso de upcasting, tener cuidado con la extensión de signo (seguida de un unsigned cast)

Integer Overflow

- ¿Cómo se pueden prevenir?
  - Utilizar tipos de datos mayores al valor máximo a representar. Ej. si sumamos 2 unsigned chars, podemos llegar al valor 510 como máximo. Un tipo de dato unsigned short permite guardar ese valor (y cualquier valor hasta 65535)
  - Utilizar chequeos antes o después de operar si ameritara. ¿El resultado de la suma es menor al de alguno de los sumandos? Se pueden utilizar constantes como INT_MAX, etc. definidas en “limits.h”
    - El código tiene que continuar legible
    - Evitar impacto de performance en modo release

Integer Overflow

- ¿Cómo se pueden prevenir?
  - Tener cuidado al generar código multiplataforma: diferentes plataformas pueden tener diferentes sizes para el mismo tipo de dato (ej.: long es 8 bytes en Linux x86_64 y 4 en Windows x86_64). Por lo tanto, utilizar tipos de datos estandarizados como los disponibles en “stdint.h”:
    - uint8_t
    - uint32_t
    - int32_t
    - …
- De la misma forma que pueden ocurrir overflows, pueden ocurrir underflows o wrap-arounds al revés

Integer Overflow

- Tamaños de tipos de datos para las plataformas más comunes:

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

Tabla extraída de “Secure Coding in C and C++”

Comparaciones signadas

- ¿Cuál es el problema de seguridad aquí?

```
#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;
}
```

Comparaciones signadas

- ¿Cuál es el problema de seguridad aquí?

```
(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
```

Comparación signada (jump-greater): estamos comparando dos enteros signados. Si fuera no signada, habría un jump-above

"malloc" va a tratar a este parámetro como no-signado

Comparaciones signadas

- Al intentar alocarse una cantidad muy grande de memoria (0xFF...FF), malloc falla retornando un puntero a NULL. Sucesivas operaciones pueden corromper memoria si la falla de malloc no está correctamente manejada
- Una alocación de memoria muy grande puede causar problemas de Denial Of Service y puede facilitar heap sprays
- ¿Cómo prevenir esto?
  - Evitar o analizar casteos implícitos
  - Observar el signo de la comparación (signada vs no-signada)
  - Utilizar valores no signados para cantidades o tamaños

Comparaciones signadas

- ¿Y ahora?

```
#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;
}
```

Comparaciones signadas

- ¿Y ahora?

```
(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>
```

Comparación no signada (jump-above).
Termina saltando

Integer Overflow

- ¿Por qué los compiladores no nos protegen de estas situaciones?
  - Para el estándar de C, los overflows y underflows son comportamiento no definido
  - Los compiladores optimizan por performance, y no agregan overhead de chequeos (innecesarios para la mayoría de los casos)
  - Evitar comportamientos no definidos es responsabilidad del desarrollador

Lab

8.1: Stack overflow en espacio de usuario

Referencias

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