Ingeniería Inversa
Clase 9

Exploit Writing II
Use After Free

martin.uy
Open by default.

Use After Free

Polimorfismo y métodos virtuales

```
class A {
public:
    void m1();
    virtual void m2();
};

class B : public A {
public:
    void m1();
    void m2();
};

A* a = new A();
A* a2 = new B();
A a3;
B* b = new B();
B* b2 = new A();
B b3;
```

¿Qué método se ejecuta?

```
a->m1();
b->m1();

a->m2();
b->m2();

a2->m1();
a2->m2();

a3.m1();
b3.m1();

a3.m2();
b3.m2();
```

```
class A {
public:
    void m1();
    virtual void m2();
};

class B : public A {
public:
    void m1();
    void m2();
};

A* a = new A();
A* a2 = new B();
A a3;
B* b = new B();
B* b2 = new A();
B b3;
```

¿Qué método se ejecuta?

```
a->m1(); // A::m1
b->m1(); // B::m1

a->m2(); // A::m2
b->m2(); // B::m2

a2->m1(); // A::m1
a2->m2(); // B::m2

a3.m1(); // A::m1
b3.m1(); // B::m1

a3.m2(); // A::m2
b3.m2(); // B::m2
```

Use After Free

- Métodos virtuales
  - Se decide qué se ejecuta en tiempo de ejecución

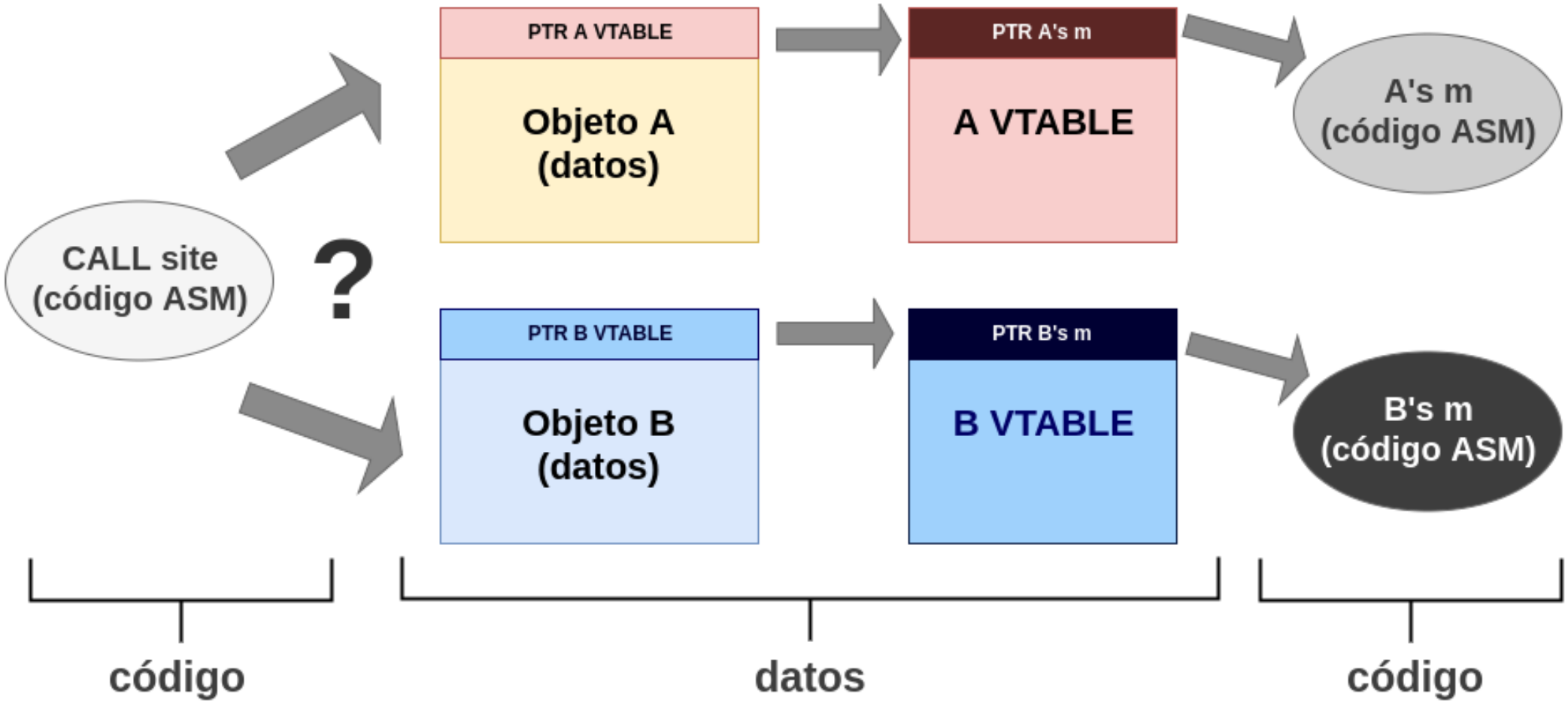
```
class A {
public:
   virtual void m();
};

class B : public A {
public:
   void m();
};
```

```
A* a;
if (rand() % 2) {
   a = new A();
} else {
   a = new B();
}
a->m();
```

Use After Free

- **Métodos virtuales**
  - No hay un destino único posible para generar un CALL directo en tiempo de compilación
    - CALL indirecto: depende de datos en tiempo de ejecución
  - Tienen un costo de performance
    - En C++ un método no es virtual a menos que se declare explícitamente como tal
    - En Java los métodos son virtuales por defecto. Sin embargo, se realizan optimizaciones para evitar la penalidad de performance cuando no es necesaria
- **Métodos no-virtuales**
  - Se conoce el destino en tiempo de compilación y es único
  - Son más eficientes

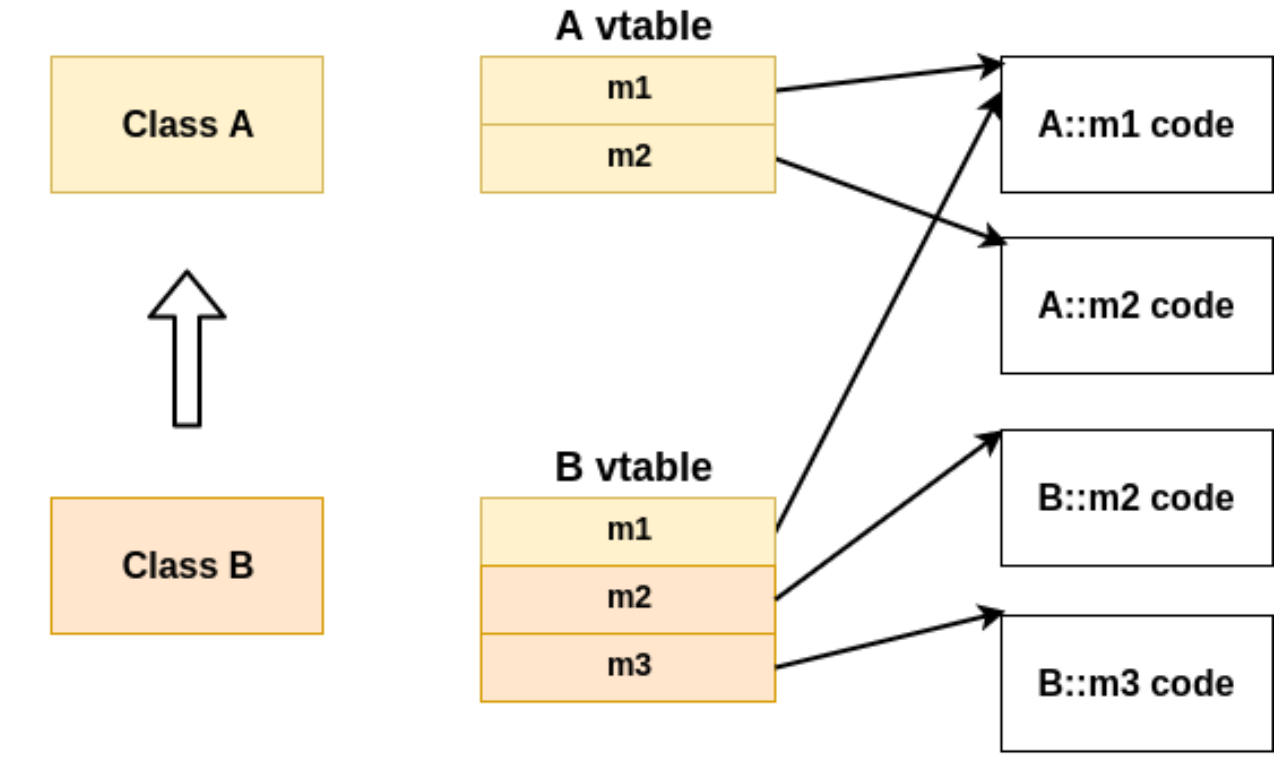
Use After Free



Use After Free

- Tabla de métodos virtuales (vtable)
  - Si la clase tiene métodos virtuales, existe en el objeto un puntero a una tabla con punteros a los métodos virtuales
    - En caso contrario, este puntero no existe y se ahorra memoria en el objeto (el objecto se parece a un struct de C)
  - Cuando una clase hereda de otras clases, la vtable de la clase incluye las vtables de las clases arriba en la jerarquía

Use After Free



A pointer to m2 implementation is in position #2 of both vtables. Which vtable will be used in run time is unknown, but position will be #2 for m2.

Use After Free

```
(gdb) x/6i $rip
=> 0x400a6c <main()+278>:       mov    -0x28(%rbp),%rax
   0x400a70 <main()+282>:       mov    (%rax),%rax
   0x400a73 <main()+285>:       mov    (%rax),%rax
   0x400a76 <main()+288>:       mov    -0x28(%rbp),%rdx
   0x400a7a <main()+292>:       mov    %rdx,%rdi
   0x400a7d <main()+295>:       callq  *%rax
```

Call site de un método virtual

Use After Free

- %rax = *(%rbp – 0x28)
  - Leer puntero al objeto desde una variable local y almacenar el valor en el registro %rax
    - Ej: variable “a”
  - El objeto puede ser de tipo A o B, según qué fue asignado a la variable “a” en tiempo de ejecución
- %rax = *(%rax)
  - %rax ahora apunta a la vtable de la clase A o B

Use After Free

- %rax = *(%rax)
  - %rax ahora apunta al método “m” (ubicado en la posición 0 de la vtable)
  - El método “m” está en la misma posición 0 de las vtables de las clases A y B
    - El código hace la desreferenciación para el método “m”, sin saber de qué vtable lo va a obtener en tiempo de ejecución pero sí sabiendo que está en la primer entrada de la vtable

Use After Free

- %rdx = *(%rbp – 0x28)
- %rdi = %rdx
  - En %rdi va el primer parámetro de la función llamada (x86_64 SystemV ABI)
  - El primer parámetro es un puntero al objeto (“this” en C++)
- CALL *%rax
  - Call indirecto al método “m”. La dirección de “m” fue anteriormente cargada en %rax

Use After Free

- Lo interesante, desde la perspectiva de la explotación, es la mezcla entre datos y código: en áreas de datos tenemos punteros a código
  - El objeto (y, por lo tanto, el puntero a la vtable) puede estar en el stack, heap o en .data
  - Las vtables están en la sección .rodata
  - Las entradas de la vtable apuntan a métodos que están en la sección .text

Use After Free

```
(gdb) x/1xg $rax
0x614c20:       0x0000000000400e98
(gdb) x/1xg *$rax
0x400e98 <_ZTV1A+16>:   0x0000000000400ca2
(gdb) x/1i **$rax
   0x400ca2 <A::m2()>:  push   %rbp
```

Vtable de la clase A

```
(gdb) x/1xg $rax
0x614c60:       0x0000000000400e80
(gdb) x/1xg *$rax
0x400e80 <_ZTV1B+16>:   0x0000000000400cce
(gdb) x/1i **$rax
   0x400cce <B::m2()>:  push   %rbp
```

Vtable de la clase B

Use After Free

¿Hay polimorfismo en C?

Use After Free

```
typedef struct _super_t {
        void(*m)(void); // virtual method
} super_t;

((super_t*)a)->m = mA;

(*(a->m))();
```

```
(gdb) x/3i $rip
=> 0x40059e <main+24>:  mov    -0x10(%rbp),%rax
   0x4005a2 <main+28>:  mov    (%rax),%rax
   0x4005a5 <main+31>:  callq  *%rax
```

Demo 9.1

Ejemplo de polimorfismo en C

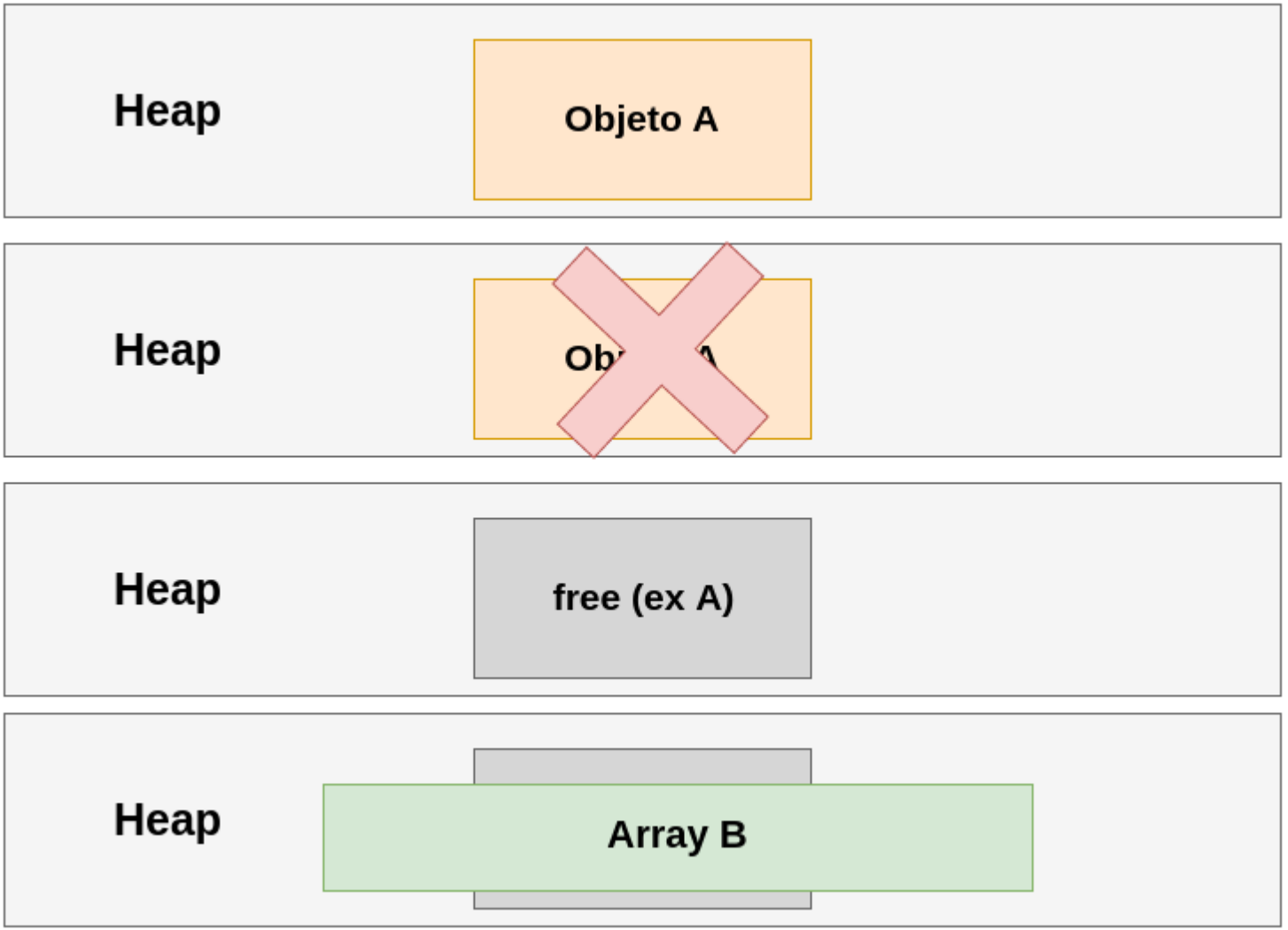
Use After Free

```
class A {
public:
   virtual void m();
};
```

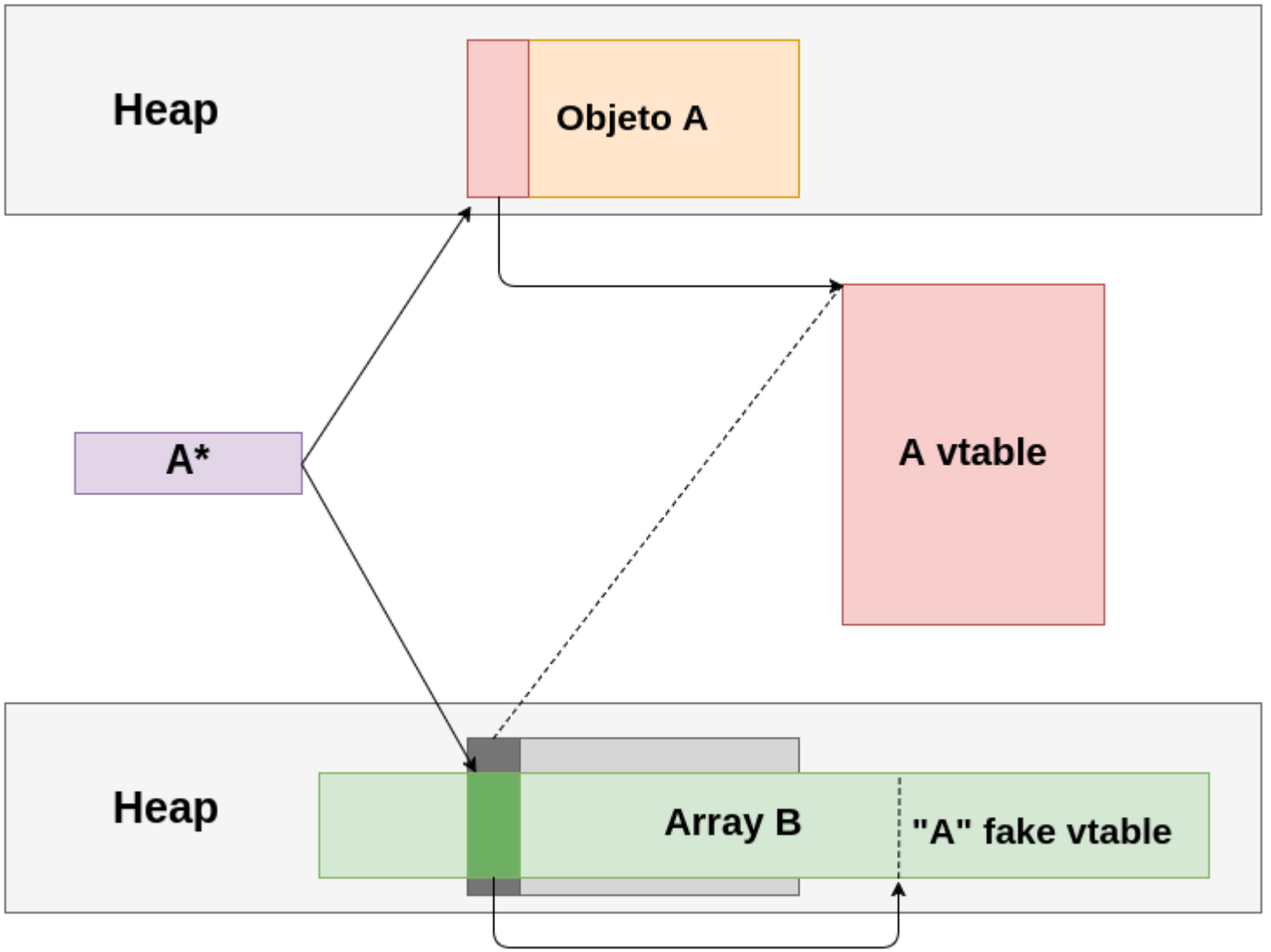
```
int main() {
   A* a = new A();
   ...
   delete a;
   ...
   a->m();
   return 0;
}
```

¿Cuál es el problema?

Use After Free



Use After Free



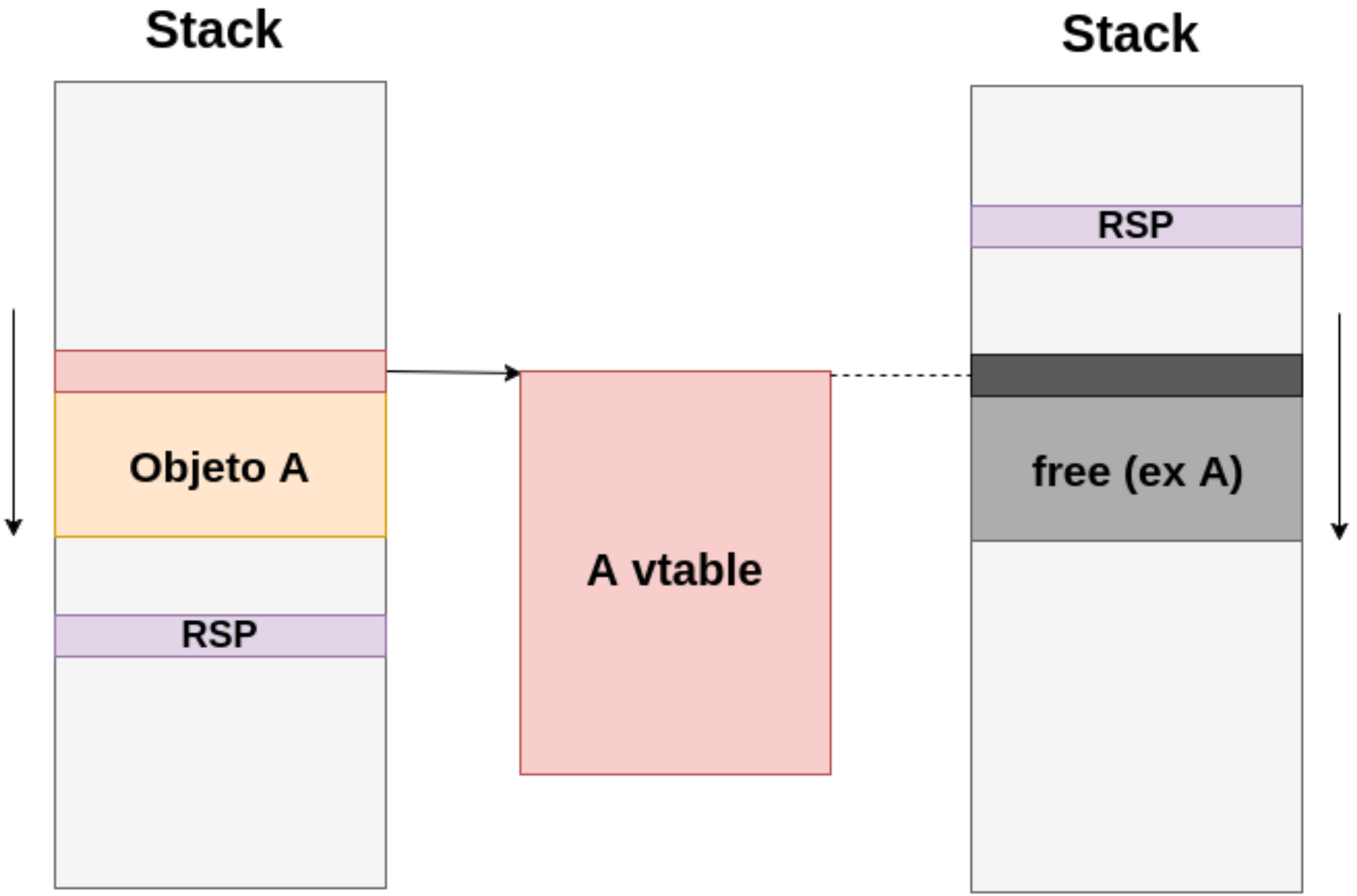
Use After Free

```
class A {
public:
   virtual void m();
};

A* f(void){
   A a;
   ...
   return &a;
}
```

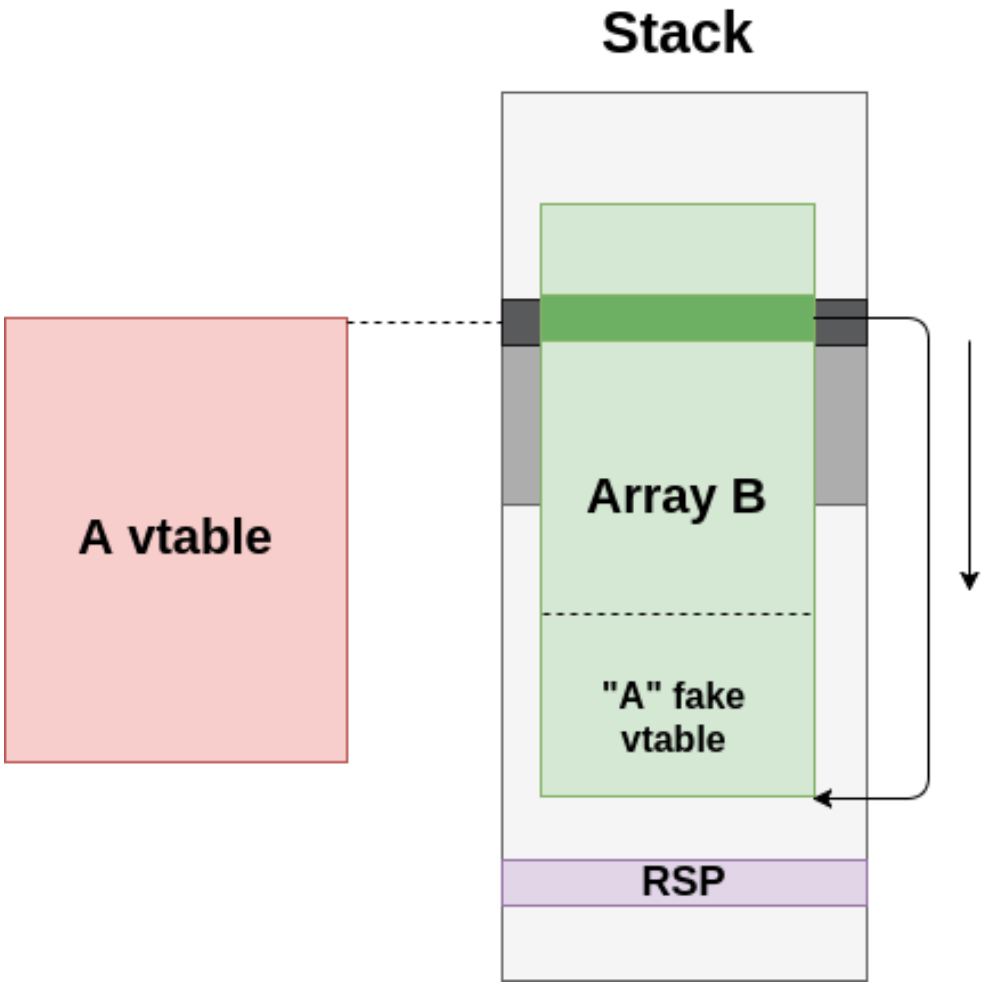
```
int main() {
   A* a = f();
   ...
   a->m();
   return 0;
}
```

¿Cuál es el problema?

Use After Free



Use After Free



Use After Free

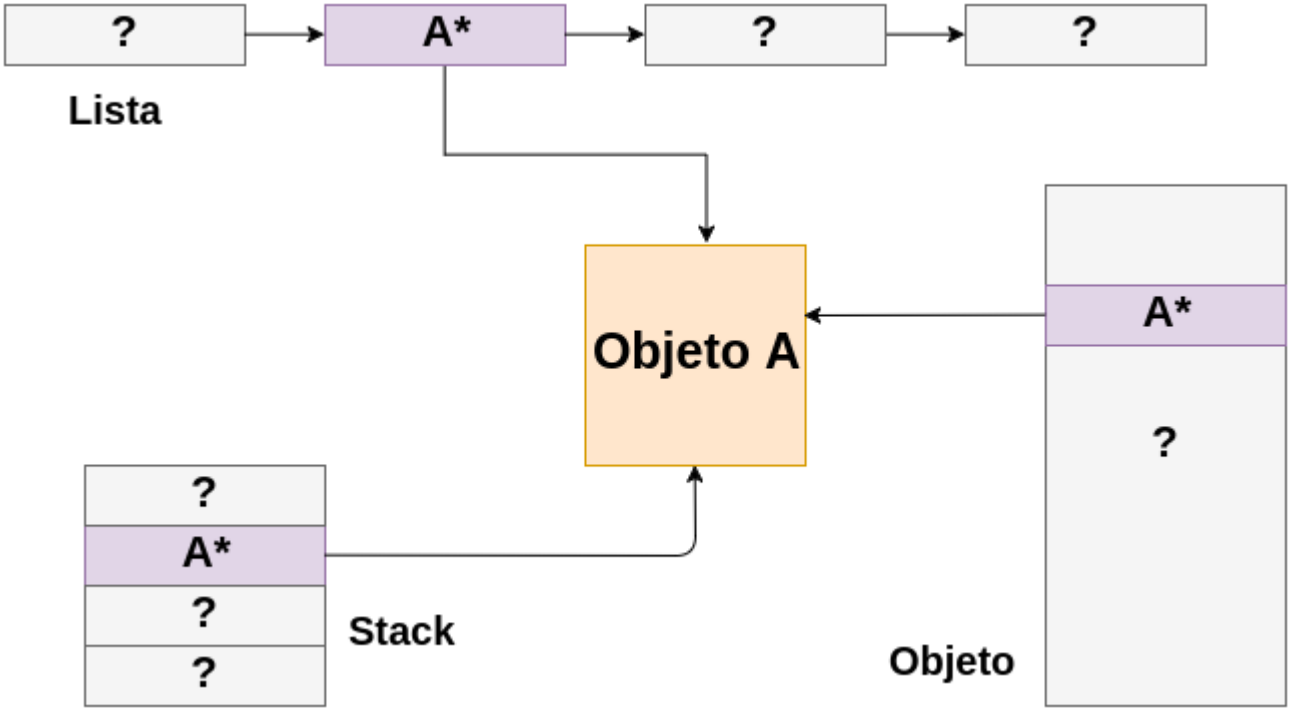
```
A* a_global;

void callback(A* a){
   a_global = a;
   return;
}
```

```
void f(void) {
   if (a_global != NULL)
   {
      a_global->m();
   }
   return;
}
```

¿Cuál es el problema?

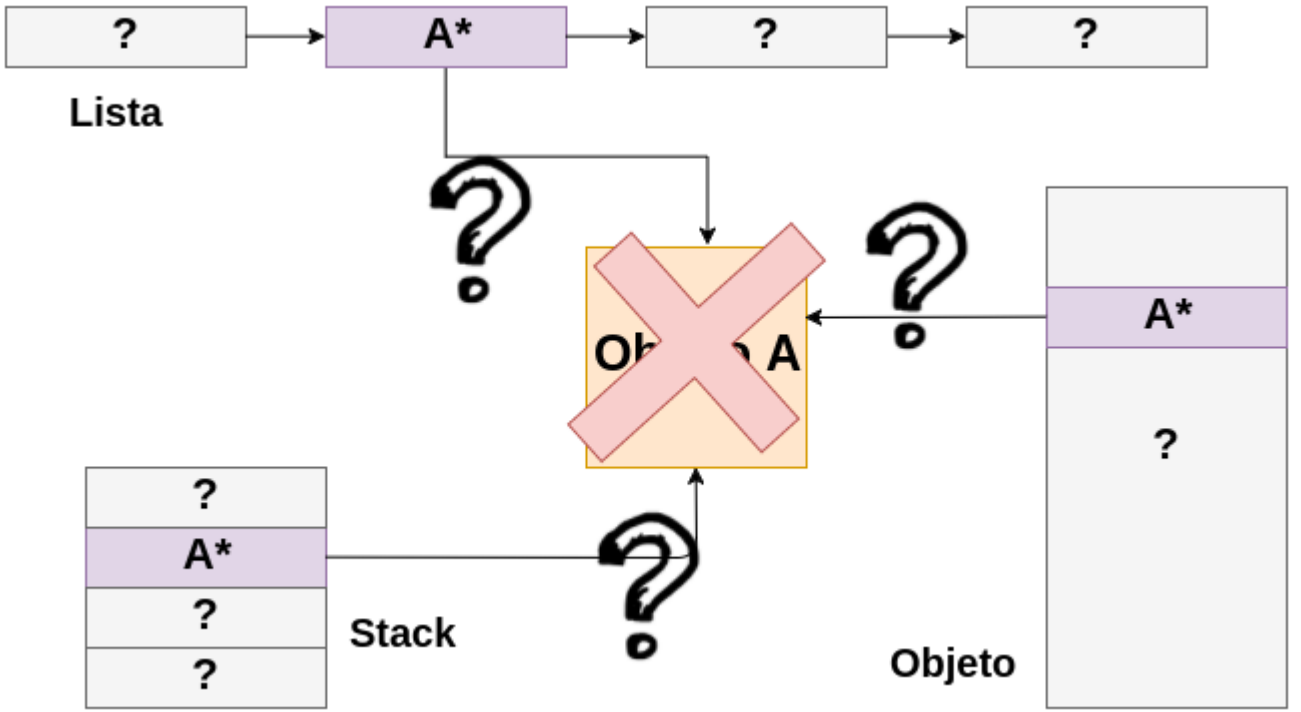
Use After Free

- Parece un problema trivial pero no lo es: en sistemas complejos pueden haber referencias (punteros) a un objeto desde diferentes lugares y ser modificadas concurrentemente inclusive



Use After Free

- Si se borra el objeto, ¿qué hacemos con las referencias? Mientras borramos las referencias, ¿qué sucede si otro thread usa una referencia concurrentemente?



Use After Free

- ¿Cuál es el ciclo de vida de un objeto y cómo gestionarlo?
- Objetos temporales
  - Almacenados en el stack
  - No guardar referencias en variables globales
  - No enviar referencias hacia “arriba” del stack
  - Es seguro enviar referencias hacia “abajo” del stack

Use After Free

- Punteros en C++11 / boost:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr
- Patrón RAII: Resource Acquisition is Initialization
  - La memoria es un recurso más

Use After Free

- std::unique_ptr
  - std::make_unique<A>(...);
  - No hay constructor de copia, solo constructor de move
  - Relación 1 objeto 1 puntero
  - No se paga costo de sincronización
  - Small footprint de memoria: tamaño de un puntero raw
  - Se puede acceder al puntero raw y usar el operador ->

Use After Free

- GNU ISO C++ (unique_ptr.h)

```
/** Takes ownership of a pointer.
 ...
 */
explicit
unique_ptr(pointer __p) noexcept
: _M_t()
{
std::get<0>(_M_t) = __p;
static_assert(!is_pointer<deleter_type>::value,
        "constructed with null function pointer deleter");
}
```

Use After Free

- GNU ISO C++ (unique_ptr.h)

```
      /// Move constructor.
       unique_ptr(unique_ptr&& __u) noexcept
       : _M_t(__u.release(),
  std::forward<deleter_type>(__u.get_deleter())) { }
```

Use After Free

- GNU ISO C++ (unique_ptr.h)

```
/// Dereference the stored pointer.
typename add_lvalue_reference<element_type>::type
operator*() const
{
__glibcxx_assert(get() != pointer());
return *get();
}

/// Return the stored pointer.
pointer
operator->() const noexcept
{
_GLIBCXX_DEBUG_PEDASSERT(get() != pointer());
return get();
}
```

Use After Free

- GNU ISO C++ (unique_ptr.h)

```
/// Destructor, invokes the deleter if the stored pointer is not null.
~unique_ptr() noexcept
{
auto& __ptr = std::get<0>(_M_t);
if (__ptr != nullptr)
  get_deleter()(__ptr);
__ptr = pointer();
}
```

Use After Free

- std::shared_ptr
  - std::make_shared<A>(...);
  - Hay constructor de copia
  - Relación 1 objeto, 1 o más punteros
  - Se paga costo de sincronización. El objeto se destruye al perder la última referencia
  - Memory footprint: (tamaño de un puntero raw)*2
  - Se puede acceder al puntero raw y al operador ->

Use After Free

- GNU ISO C++ (shared_ptr.h)

```
template<typename _Tp, _Lock_policy _Lp>
  class __shared_ptr
{
 ...
 _Tp*          _M_ptr;         // Contained pointer.
 __shared_count<_Lp>  _M_refcount;    // Reference
counter.
};
```

Use After Free

- GNU ISO C++ (shared_ptr.h)

```
template<typename _Tp1>
          __shared_ptr(const __shared_ptr<_Tp1,
_Lp>& __r)
   : _M_ptr(__r._M_ptr),
_M_refcount(__r._M_refcount) // never throws

{ __glibcxx_function_requires(_ConvertibleCon
cept<_Tp1*, _Tp*>) }
```

Constructor de copia

Use After Free

- std::weak_ptr
  - Se crea en relación a un shared_ptr
  - No cuenta para la destrucción del objeto
  - El weak ptr solo se puede utilizar para obtener un shared_ptr (en caso de que el objeto no haya sido eliminado)
    - Mientras el objeto está en uso, existe un shared ptr que impide su eliminación

Use After Free

- Buena práctica: asignar el valor NULL a las variables que contenían un puntero a objeto después de liberarlo
- Lenguajes como Java, .NET, Python, etc. no permiten gestionar la memoria manualmente *
  - * excepto que se usen APIs específicas (ej. Unsafe en Java)
  - El polimorfismo se implementa con vtables también
  - No son vulnerables a Use After Free a menos que haya un bug en la VM
    - Se paga el costo de performance y memory footprint

Demo 9.2

Explotación de Use After Free en stack (espacio de kernel)

Use After Free

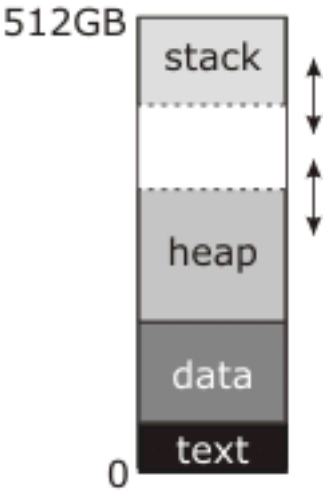
- Explotar Use After Free en el heap tiene dificultades adicionales: ¿cómo logramos “pisar” el lugar liberado? ¿cómo predecir la dirección de memoria donde estaría nuestra “fake vtable”?
- El heap es un área de memoria en la cuál el proceso puede almacenar datos de largo variable y desconocido, en tiempo de ejecución. Ej. arrays, streams, objetos, etc.
- Los procesos utilizan generalmente alocadores dinámicos de memoria en espacio de usuario (ofrecidos por el sistema operativo) para manejar el Heap:
  - Simplificación o abstracción (memoria reservada vs. commiteada)
  - Granularidad (alocar pocos bytes)
  - Memoria contigua a nivel de direcciones virtuales (no necesariamente de direcciones físicas)
  - Anti-fragmentación y gestión de la memoria (cachés)

Use After Free

- Todo proceso tiene al menos 1 heap
- Windows tiene múltiples APIs para manejo de memoria y soporte para múltiples heaps:
  - HeapCreate / HeapDelete
  - HeapAlloc / HeapFree
  - VirtualAlloc / VirtualFree
  - malloc (MSVCRT)
- En Linux
  - mmap es usado para alocar segmentos de memoria
  - brk / sbrk son usados para ampliar o reducir el tamaño del heap
  - malloc (glibc)

Use After Free

- brk syscall (Linux)
  - Definida en mm/mmap.c (kernel)
  - Cambia el tamaño del “segmento de datos” (heap)
    - Esto implica mapear o desmapear memoria física
  - El heap crece hacia direcciones virtuales más altas (el stack hacia direcciones virtuales más bajas)



Use After Free

- brk syscall (Linux)

```
struct mm_struct {
    ...
    unsigned long start_brk, brk, start_stack;
    ...
}
```

include/linux/mm_types.h (Linux kernel)

La estructura mm_struct describe la memoria de un proceso, a nivel de kernel. En particular, start_brk y brk nos permiten visualizar la ubicación del heap.

Use After Free

```
int main(void) {
    char* buff = (char*)malloc(1);
    if (buff != NULL) {
        free(buff);
        buff = NULL;
    }
    return 0;
}
```

Al llamar a “malloc” por primera vez, glibc deberá inicializar y extender el heap del proceso. Utilizará para ello la syscall brk.

Mapa del proceso antes de llamar a malloc:

...
00601000-00602000 … main
7ffff7a24000-7ffff7bce000 … libc.so.6
...

Use After Free

En este caso de ejemplo, glibc llama a la syscall brk con valor 0x623000.

Al entrar a sys_brk (kernel):

```
(gdb) print/x ((struct mm_struct*)(current->mm))->start_brk
$18 = 0x602000
(gdb) print/x ((struct mm_struct*)(current->mm))->brk
$19 = 0x602000
```

El heap tiene tamaño 0 en este momento del proceso.

Al finalizar sys_brk:

```
(gdb) print/x ((struct mm_struct*)(current->mm))->start_brk
$18 = 0x602000
(gdb) print/x ((struct mm_struct*)(current->mm))->brk
$19 = 0x623000
```

Use After Free

Mapa del proceso al retornar de la syscall sys_brk:

...
00601000-00602000 rw-p … main
00602000-00623000 rw-p ... [heap]
7ffff7a24000-7ffff7bce000 r-xp … libc.so.6
…

malloc retornó finalmente la dirección 0x602260, dentro del segmento de heap.

Use After Free

```
int main(void) {
    unsigned int iter_count = 10U;
    char* previous_buff = NULL;

    while (iter_count-- > 0U) {
        char* buff = (char*)malloc(1U);
        printf("Buff: %p - Delta with previous: %lu\n", buff,
(unsigned long)(buff - previous_buff));
        previous_buff = buff;
    }
    return 0;
}
```

Use After Free

Buff: 0x1ca4**260** - Delta with previous: 30032480
Buff: 0x1ca4**690** - Delta with previous: 1072
Buff: 0x1ca4**6b0** - Delta with previous: 32
Buff: 0x1ca4**6d0** - Delta with previous: 32
Buff: 0x1ca4**6f0** - Delta with previous: 32
…

Buff: 0x9b8**260** - Delta with previous: 10191456
Buff: 0x9b8**690** - Delta with previous: 1072
Buff: 0x9b8**6b0** - Delta with previous: 32
Buff: 0x9b8**6d0** - Delta with previous: 32
Buff: 0x9b8**6f0** - Delta with previous: 32
...

Use After Free

Las direcciones donde se alocan los chunks de memoria no son completamente aleatorias. De las trazas anteriores se puede suponer:

- el alocador intenta no fragmentar la memoria (alocaciones contiguas)
- el tamaño mínimo entre 2 chunks (incluyendo la metadata) es de 32 bytes
- Si bien las direcciones son diferentes, las terminaciones son iguales debido a la alineación

Si liberamos un chunk del medio en esta secuencia y generamos una nueva alocación del mismo tamaño, ¿cuál es la dirección más probable para el nuevo chunk?

Use After Free

Buff: 0x1f38260 - Delta with previous: 32735840
Buff: 0x1f38690 - Delta with previous: 1072
Buff: 0x1f386b0 - Delta with previous: 32
Buff: 0x1f386d0 - Delta with previous: 32
Buff: **0x1f386f0** - Delta with previous: 32
Buff: 0x1f38710 - Delta with previous: 32
…

Chunk liberado: **0x1f386f0**
Nuevo buff: **0x1f386f0**

- Parece haber una caché: el siguiente objeto se aloca en el lugar del último liberado. Esto tiene sentido para aprovechar la caché row del CPU

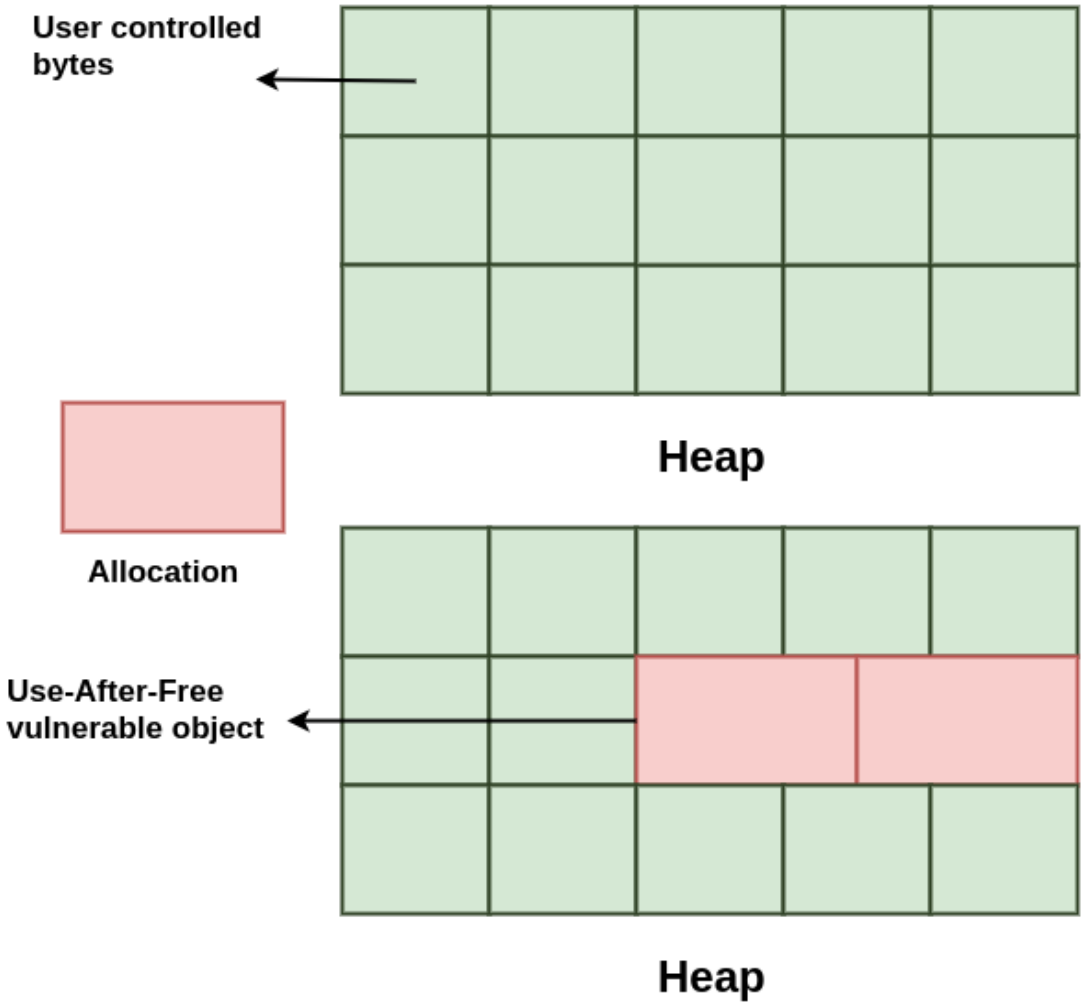
Use After Free

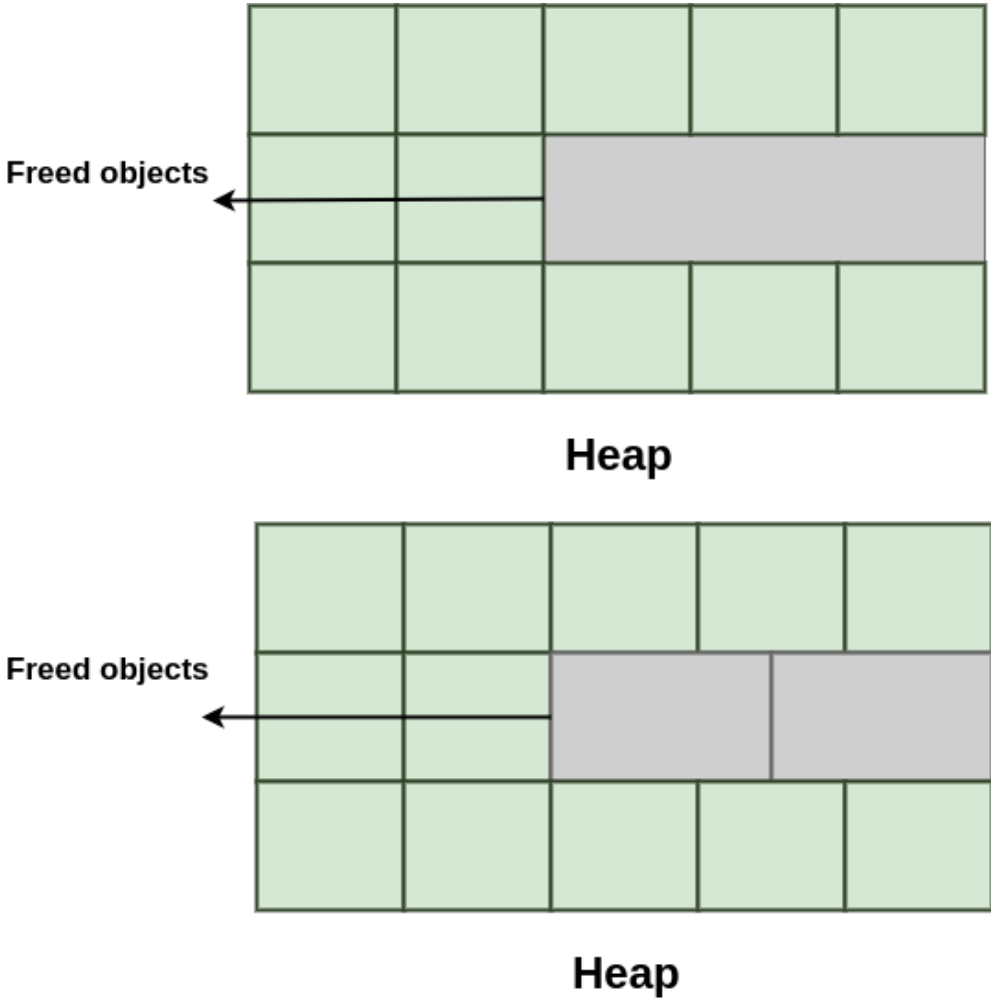
- La técnica de heap spray consiste en generar alocaciones de memoria de tamaños convenientes de forma tal de poder predecir el layout, con una razonable certeza
- No es una vulnerabilidad en sí misma, pero algunos alocadores intentan detectar sprays, randomizar las alocaciones o separar heaps para dificultar la explotación (heap isolation)
- No hay una técnica universal: tiene que ser ajustada en función del alocador dinámico y el heap
- No es una técnica 100% confiable
- En 32 bits se puede generar un agotamiento de la memoria porque el rango virtual de direcciones tiene relación con el rango físico. En 64 bits es imposible

Use After Free

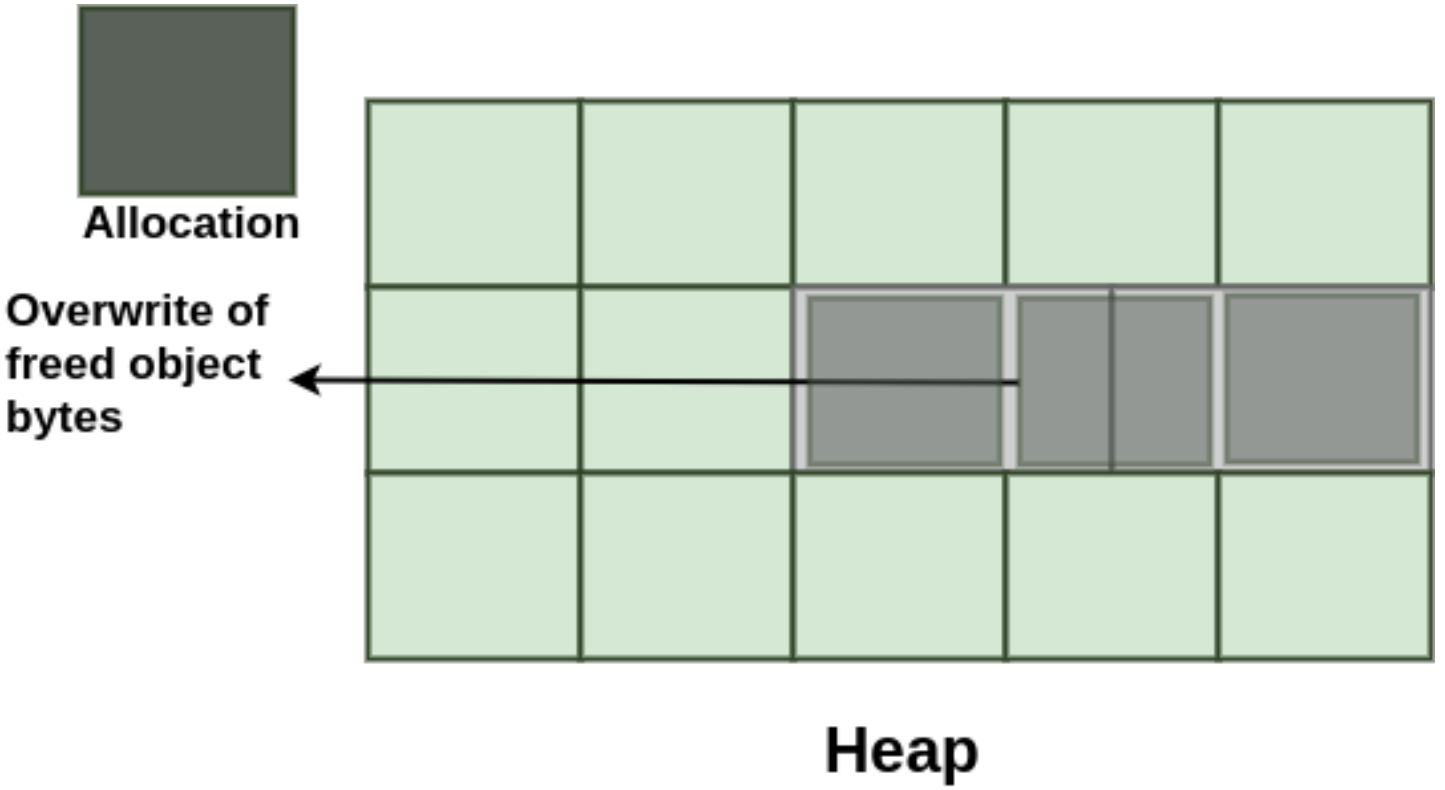
- Ejemplo:
  - alocamos objetos de un cierto tamaño en todo el heap
  - en el medio de ellos, liberamos 1 o varios objetos
  - generamos una alocación del objeto vulnerable a use-after-free. El objeto luego se libera. No conocemos su ubicación exacta
  - Liberamos objetos de forma conveniente
  - Generamos alocaciones de objetos de un tamaño conveniente de forma tal de predecir el overlap con el objeto liberado (vulnerable a use-after-free)
  - Nunca conocemos las direcciones exactas: nos basamos en el overlap relativo (offsets)

Use After Free





Use After Free



Use After Free

- ¿Cómo generar alocaciones?
  - Eso depende de cada proceso atacado. Por ejemplo, en un browser instanciar arrays desde JavaScript va a significar instanciar arrays en el heap nativo
  - Los arrays y strings son interesantes para el atacante por la cantidad de bytes que controlan directamente en el heap. Por esa razón, suelen haber heaps separados para ellos. Otros objetos quizás permitan controlar menos bytes pero pueden también ser interesantes
  - Se puede jugar con el tamaño de las alocaciones para que sean manejadas por diferentes alocadores dinámicos y para minimizar el espacio de garbage bytes entre ellas

Use After Free

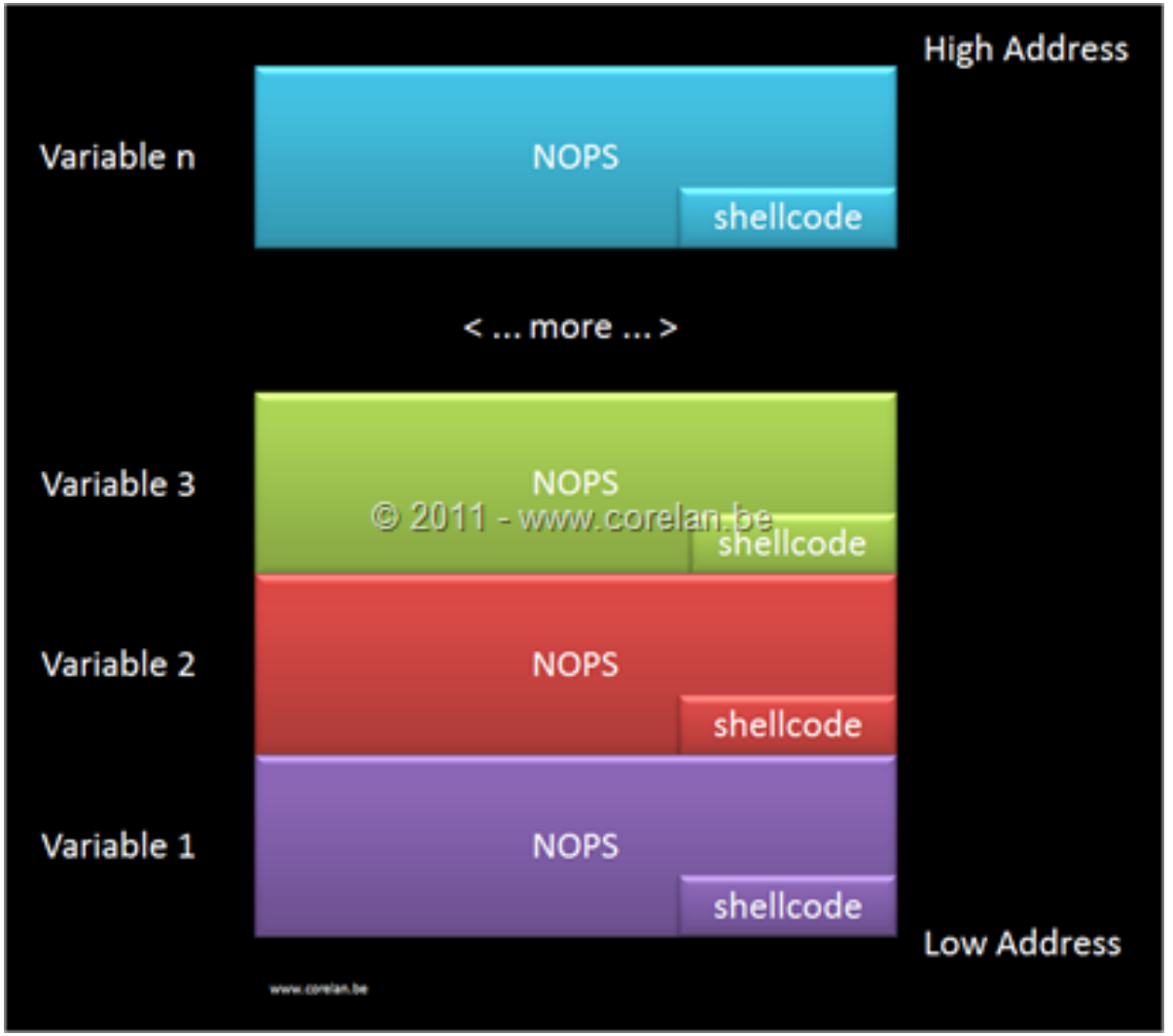


Imagen de corelan.be

Use After Free

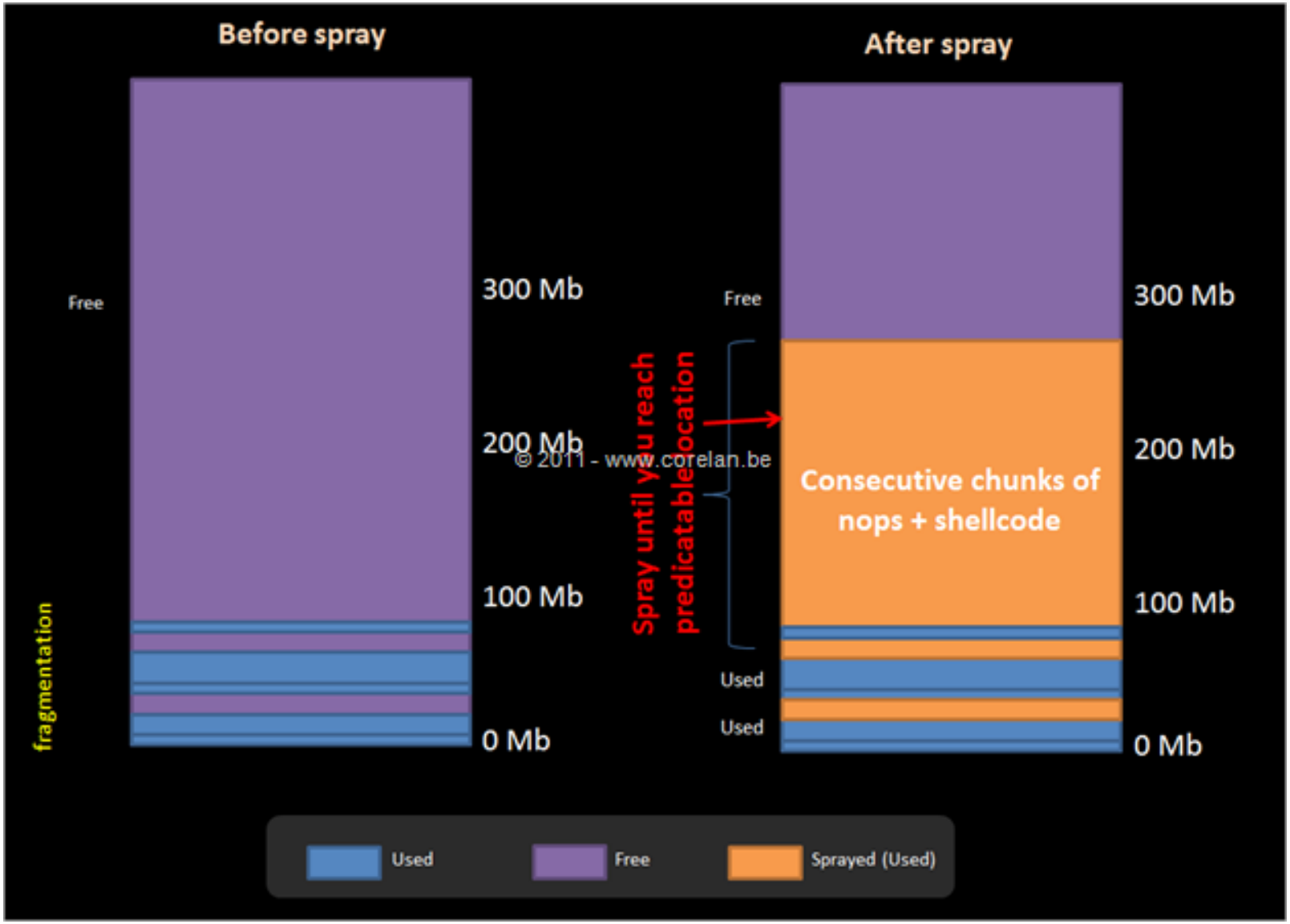


Imagen de corelan.be

Use After Free

- Memory safety & Type confusion
  - Los entornos gestionados (ej. VMs de lenguajes como Java, .NET, etc.) aseguran que las lecturas/escrituras a los campos de un objeto se hagan según el tipo, offset y límites correctos
    - Ej. en el offset 0x0 del objeto hay un int, que ocupa 4 bytes. No está en el offset 0x1 y no ocupa 8 bytes.
  - No pueden haber lecturas/escrituras fuera de los límites del tipo de dato, y no se pueden usar para una operación diferente a la adecuada (ej.: no se puede dereferenciar a un int como si fuera un puntero a objeto)

Use After Free

```
class A {
    public int intVar1;
}

class B {
    public int intVar1;
    public int intVar2;
}
```

```
private static A a = new A();

private static B b = new B();
```

Java

- Ningún código (bytecode interpretado o código JITteado) puede escribir en “a.intVar1” un tipo de dato String, o en “a.intVar2” algún valor
- El bytecode es verificado antes de interpretarse o compilarse

Use After Free

```
private static void jitIt() {
    a.intVar1 = b.intVar1 % 1;
    b.intVar1 = b.intVar2 % 2;
    b.intVar2 = a.intVar1 % 3;
}
```

```
7fffd8d1f49c:   mov   0xc(%rdi),%eax
7fffd8d1f49f:   mov   $0x1,%ebx
7fffd8d1f4a4:   cmp   $0x80000000,%eax
7fffd8d1f4aa:   jne   0x7fffd8d1f4bb
7fffd8d1f4b0:   xor   %edx,%edx
7fffd8d1f4b2:   cmp   $0xffffffff,%ebx
7fffd8d1f4b5:   je    0x7fffd8d1f4be
7fffd8d1f4bb:   cltd
7fffd8d1f4bc:   idiv  %ebx
7fffd8d1f4be:   mov   %edx,0xc(%rsi)
```

RSI = puntero a "a"

RDI = puntero a "b"

Use After Free

```
7fffd8d1f4c1: mov  0x10(%rdi),%eax
7fffd8d1f4c4: mov  $0x2,%ebx
7fffd8d1f4c9: cmp  $0x80000000,%eax
7fffd8d1f4cf: jne  0x7fffd8d1f4e0
7fffd8d1f4d5: xor  %edx,%edx
7fffd8d1f4d7: cmp  $0xffffffff,%ebx
7fffd8d1f4da: je   0x7fffd8d1f4e3
7fffd8d1f4e0: cltd
7fffd8d1f4e1: idiv %ebx
7fffd8d1f4e3: mov  %edx,0xc(%rdi)
```

b.intVar1 = b.intVar2 % 2;

RSI = puntero a “a”

RDI = puntero a “b”

```
7fffd8d1f4e6: mov  0xc(%rsi),%eax
7fffd8d1f4e9: mov  $0x3,%esi
7fffd8d1f4ee: cmp  $0x80000000,%eax
7fffd8d1f4f4: jne  0x7fffd8d1f505
7fffd8d1f4fa: xor  %edx,%edx
7fffd8d1f4fc: cmp  $0xffffffff,%esi
7fffd8d1f4ff: je   0x7fffd8d1f508
7fffd8d1f505: cltd
7fffd8d1f506: idiv %esi
7fffd8d1f508: mov  %edx,0x10(%rdi)
```

b.intVar2 = a.intVar1 % 3;

Use After Free

- Si se lograra engañar a la VM para poner un puntero a “a” en RDI, tendríamos una lectura/escritura fuera de los límites del objeto de tipo A. Podríamos pisar memoria de un objeto contiguo del heap
- Supongamos ahora que el tipo A tiene como primer miembro una referencia a un objeto B, sobre el cual se hacen operaciones de lectura/escritura

Use After Free

```
class A {
    public B refToB;
}

class B {
    public int intVar1;
    public int intVar2;
}

private static A a = new A();

private static B b = new B();

private static void jitIt() {
    a.refToB.intVar1 = a.refToB.intVar1 % 1;
    b.intVar1 = b.intVar2 % 2;
}
```

Java

Use After Free

```
7fffd8d221a7:  mov    0xc(%rax),%edi
7fffd8d221aa:  push   %r10
7fffd8d221ac:  cmp    0x1e420dfd(%rip),%r12
7fffd8d221b3:  je     0x7fffd8d22230
                ...
7fffd8d22236:  mov    0xc(%rdi),%eax
7fffd8d22239:  mov    $0x1,%ebx
7fffd8d2223e:  cmp    $0x80000000,%eax
7fffd8d22244:  jne    0x7fffd8d22255
7fffd8d2224a:  xor    %edx,%edx
7fffd8d2224c:  cmp    $0xffffffff,%ebx
7fffd8d2224f:  je     0x7fffd8d22258
7fffd8d22255:  cltd
7fffd8d22256:  idiv   %ebx
7fffd8d22258:  mov    %edx,0xc(%rdi)
```

RAX = puntero a "a"
EDI = a.refToB (puntero comprimido a "b")

RDI = a.refToB (puntero descomprimido a "b")

Use After Free

- Si logramos poner un objeto b’ de tipo B en memoria liberada donde había uno de tipo A (manteniendo la referencia “a”), podemos controlar la dirección de cada escritura/lectura:
  - b’.intVar1 = setear la dirección de memoria
  - a.refToB.intVar1 = setear el valor

Lab

Lab 9.1: Explotación de Use After Free en stack (espacio de usuario)

Lab

Lab 9.2: Explotación con Heap Spray

Setear EIP a la dirección 0x41414141

Referencias

- https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
- https://msdn.microsoft.com/en-us/library/ms810603.aspx