Reverse Engineering
Class 9

Exploit Writing II
Use After Free

martin.uy
Open by default.

Use After Free

Polymorphism and virtual methods

```
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;
```

Which method is executed?

```
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;
```

Which method is executed?

```
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

- Virtual methods
  - What to execute is decided in run time

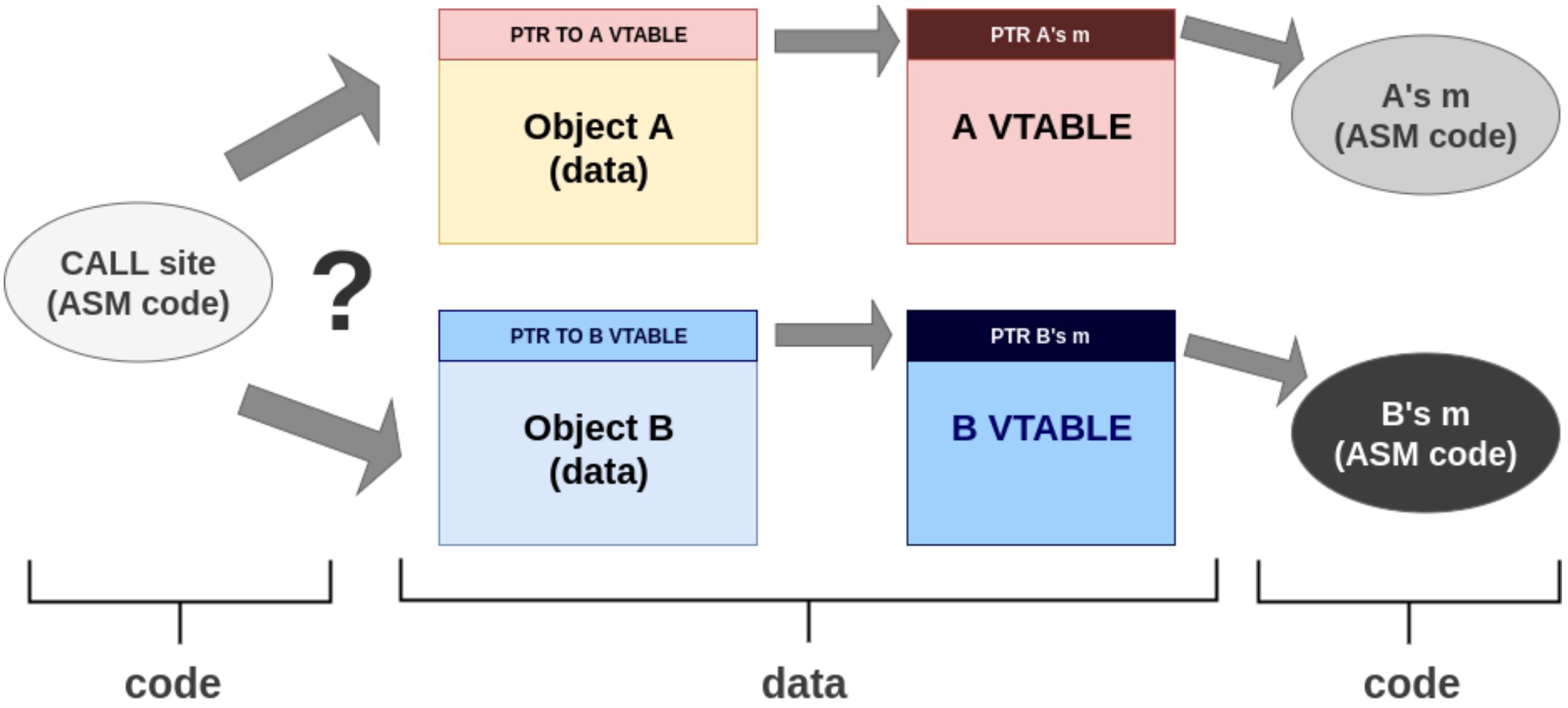
```
class A {
public:
   virtual void m();
};

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

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

Use After Free

- **Virtual methods**
  - There isn't a single possible target to generate a direct CALL in compile time
    - Indirect CALL: depends on run time data
  - Have a performance cost
    - In C++ a method is not virtual unless declared as such
    - In Java methods are virtual by default. However, optimizations are made to avoid performance penalty when not needed
- **Non-virtual methods**
  - Target is known in compile time and is unique
  - Better performance

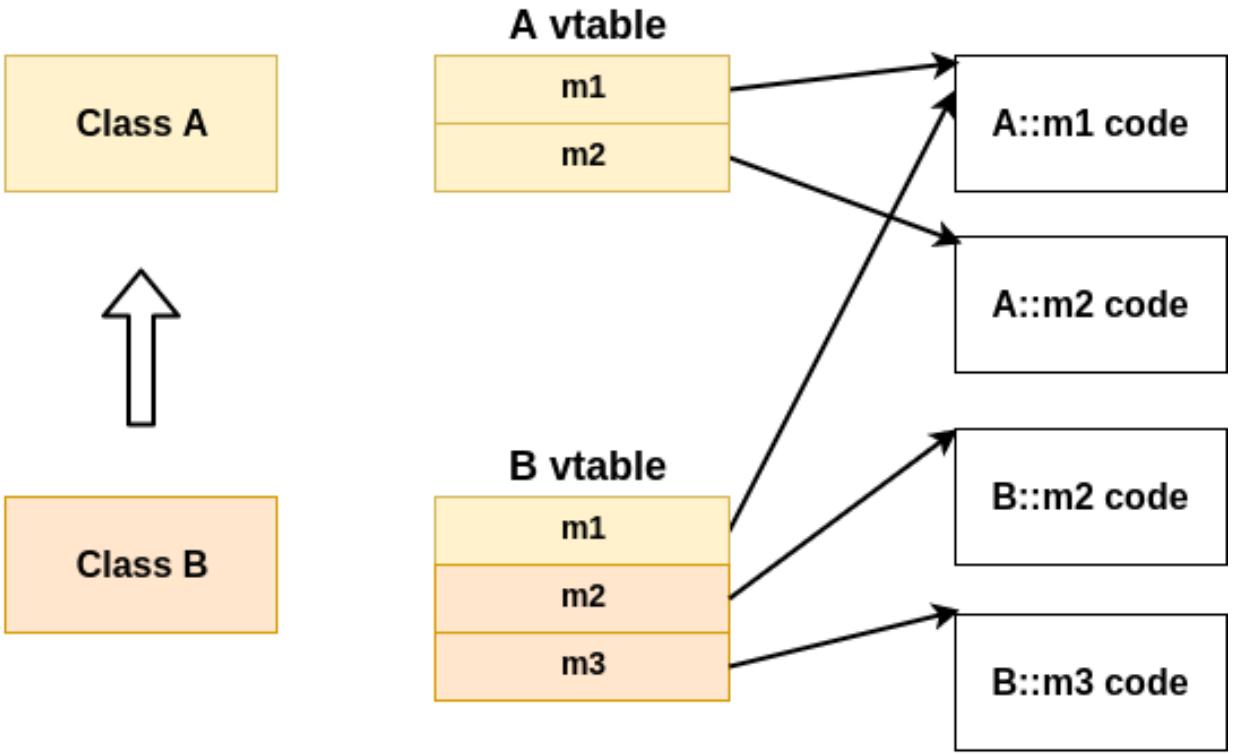
Use After Free



Use After Free

- Virtual methods table (vtable)
  - If a class has virtual methods, there is a pointer in the object to a table with pointers to virtual methods
    - If there are no virtual methods, this pointer does not exist and memory is saved (the object looks like a C struct)
  - When a class inherits from other classes, vtable includes vtables from parent classes

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 for a virtual method

Use After Free

- %rax = *(%rbp – 0x28)
  - Read the pointer to the object from a local variable and store the value in %rax register
    - I.e: variable “a”
  - Object can be of A or B type, depending o what has been assigned to “a” variable in run time
- %rax = *(%rax)
  - %rax now points to A or B class vtable

Use After Free

- %rax = *(%rax)
  - %rax now points to “m” method (located in position 0 of the vtable)
  - “m” method is in the same position 0 of A and B class vtables
    - Code dereferences “m” method without knowing from which vtable will be obtained in run time. All it’s known is that the method is in vtable’s first entry

Use After Free

- %rdx = *(%rbp – 0x28)
- %rdi = %rdx
  - In %rdi goes the first parameter for the called function (x86_64 SystemV ABI)
  - This first parameter is a pointer to the object (“this” in C++)
- CALL *%rax
  - Indirect call to “m” method. “m” address was previously loaded in %rax register

Use After Free

- The interesting thing, from the exploitation point of view, is the mix between data and code: there are pointers to code in data areas
  - The object (and, thus, the pointer to the vtable) may be located in the stack, heap or .data sections
  - vtables are in .rodata section
  - Vtable entries point to methods located in .text section

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
```

Class A vtable

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

Class B vtable

Use After Free

Is there polymorphism in 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

Example of polymorphism in C

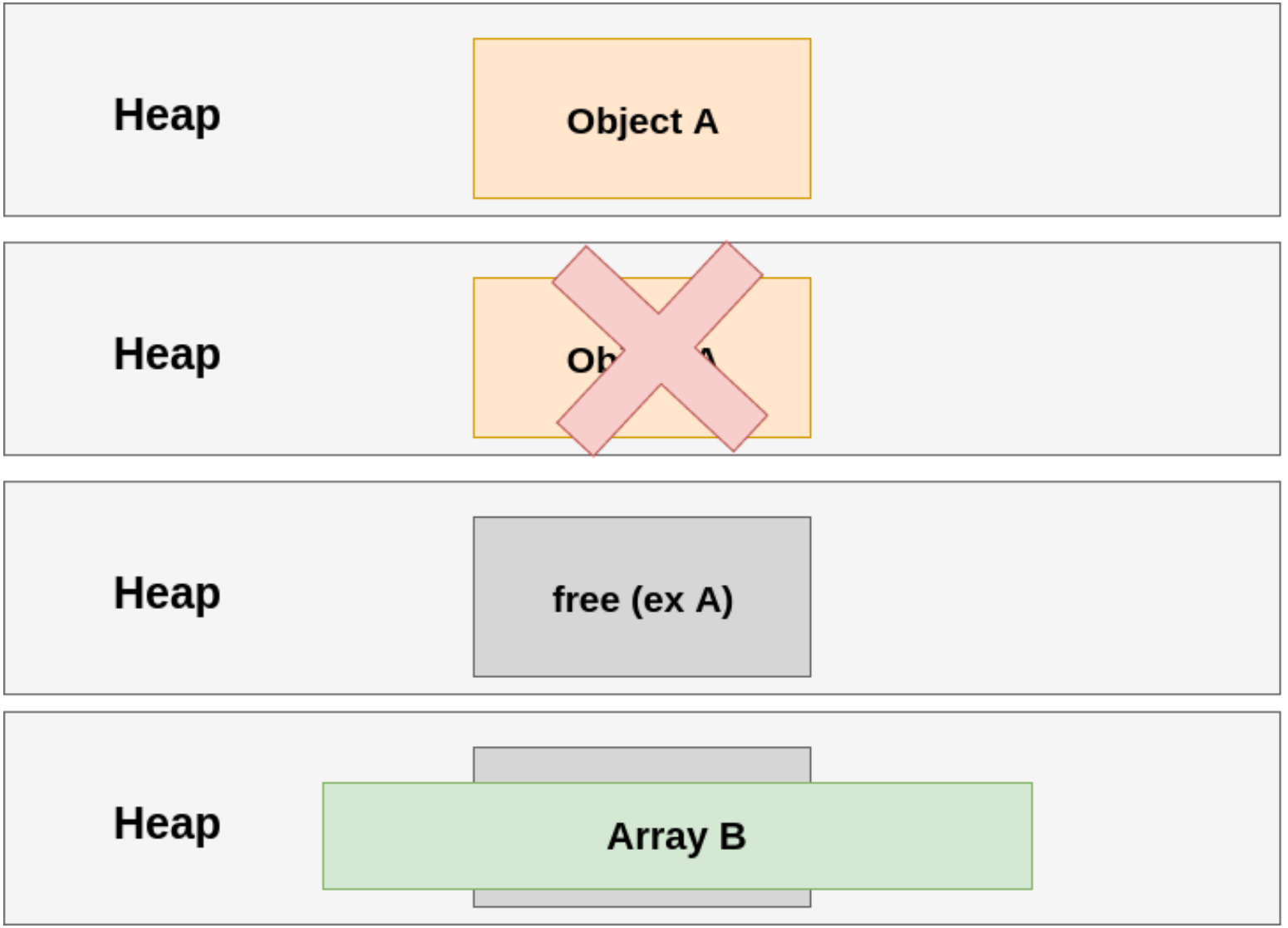
Use After Free

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

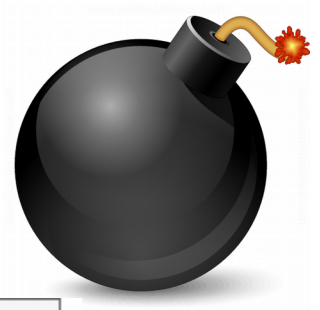
```
int main() {
   A* a = new A();
   ...
   delete a;
   ...
   a->m();
   return 0;
}
```

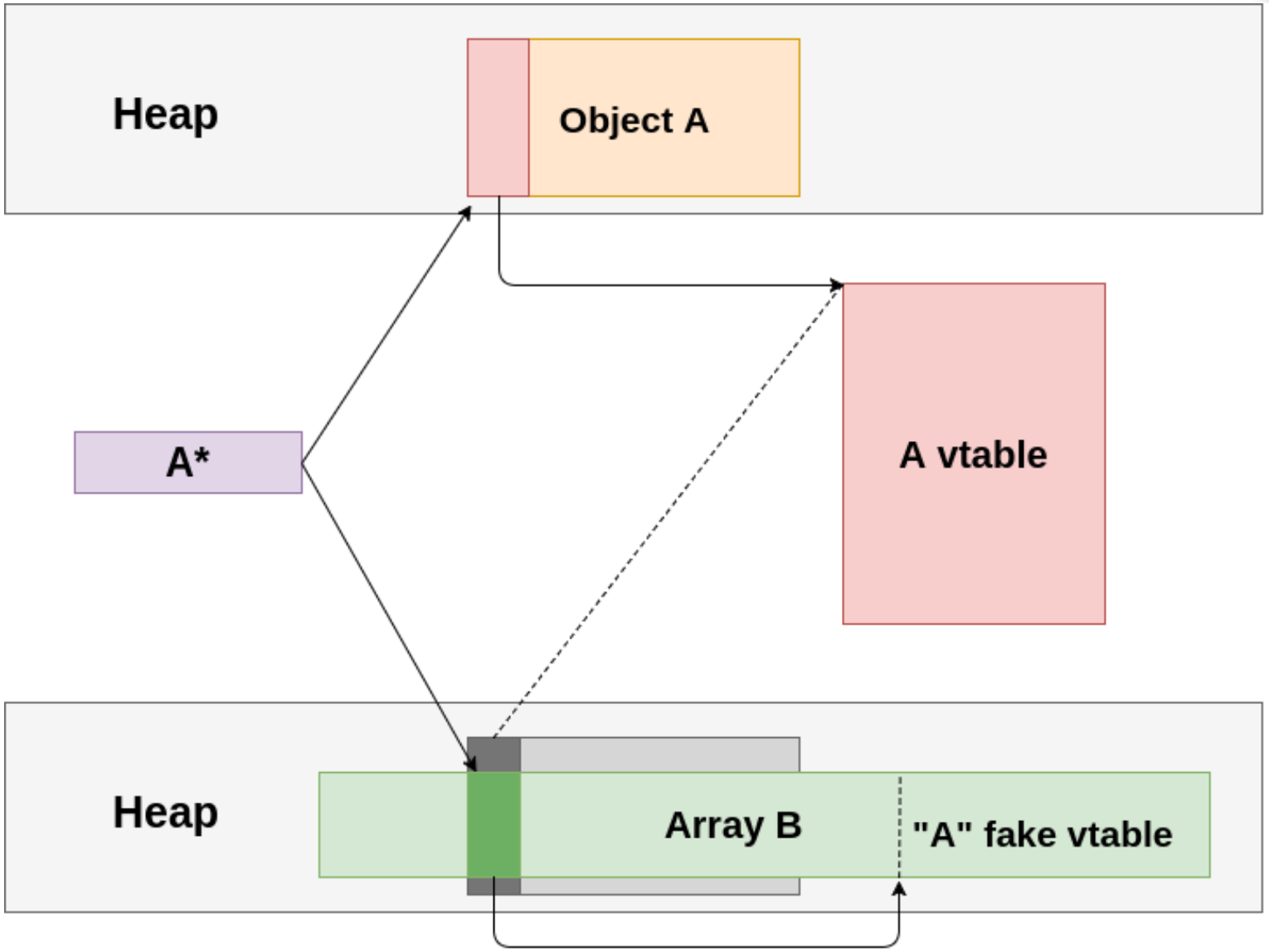
What is the problem?

Use After Free



Use After Free





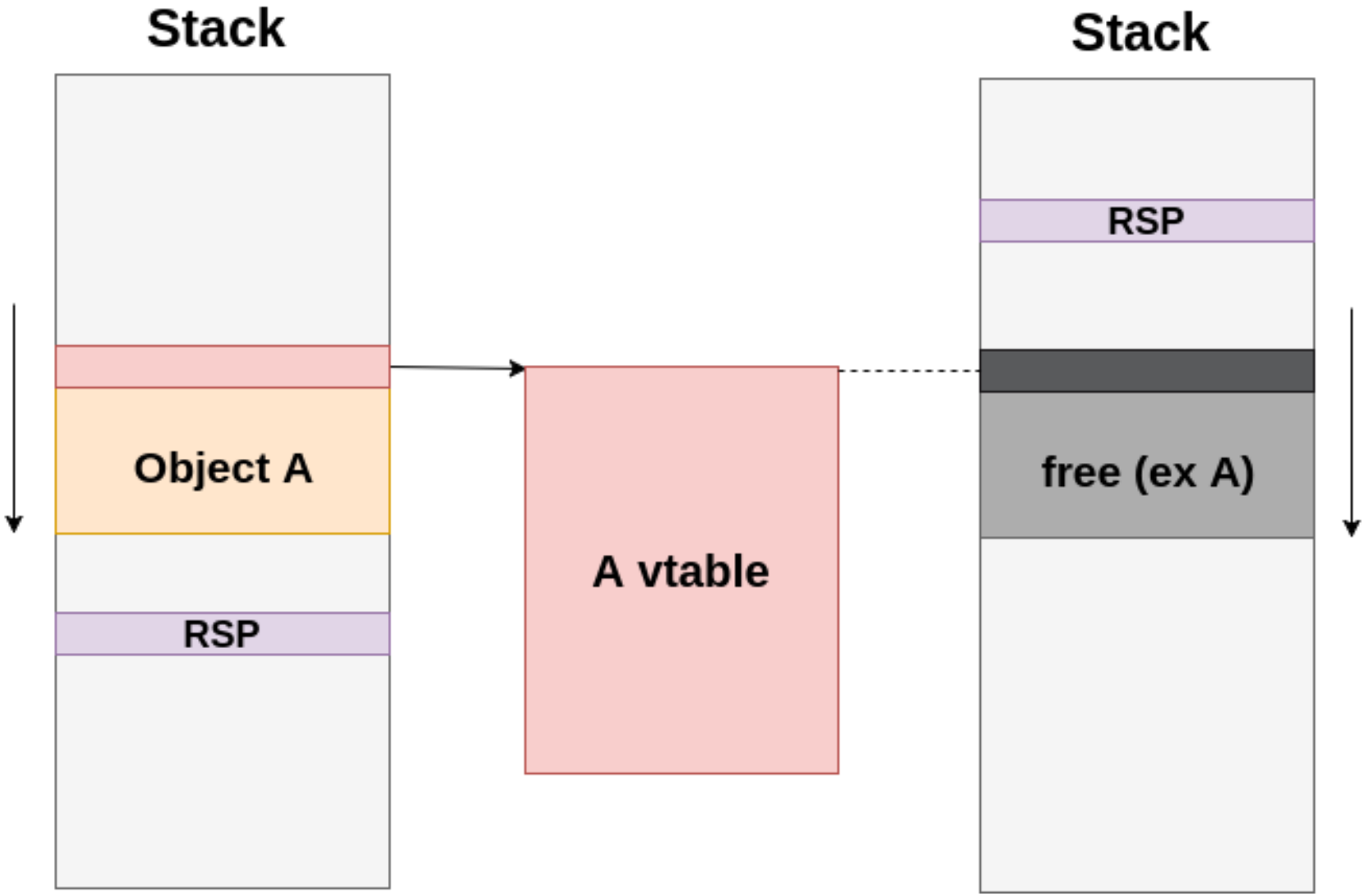
Use After Free

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

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

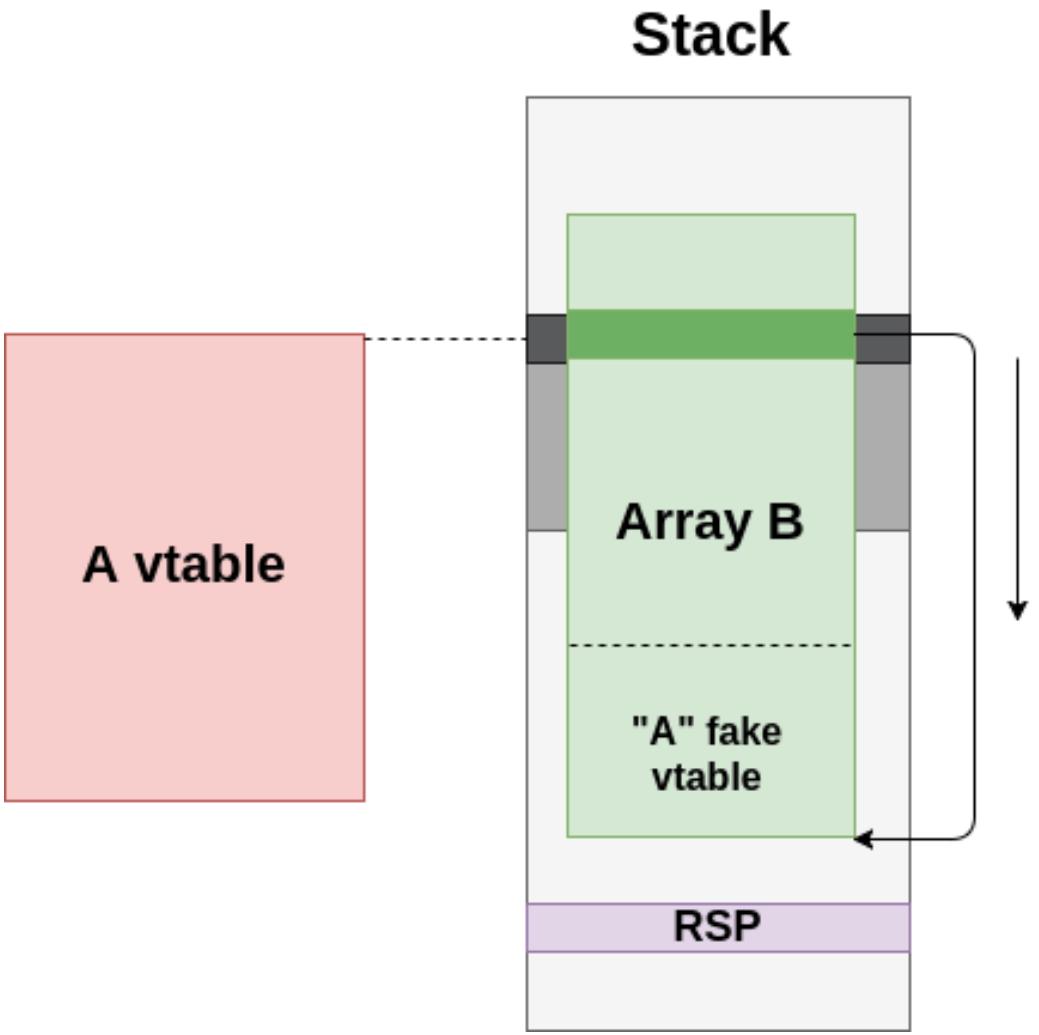
```
int main() {
   A* a = f();
   ...
   a->m();
   return 0;
}
```

What is the problem?

Use After Free



Use After Free



Use After Free

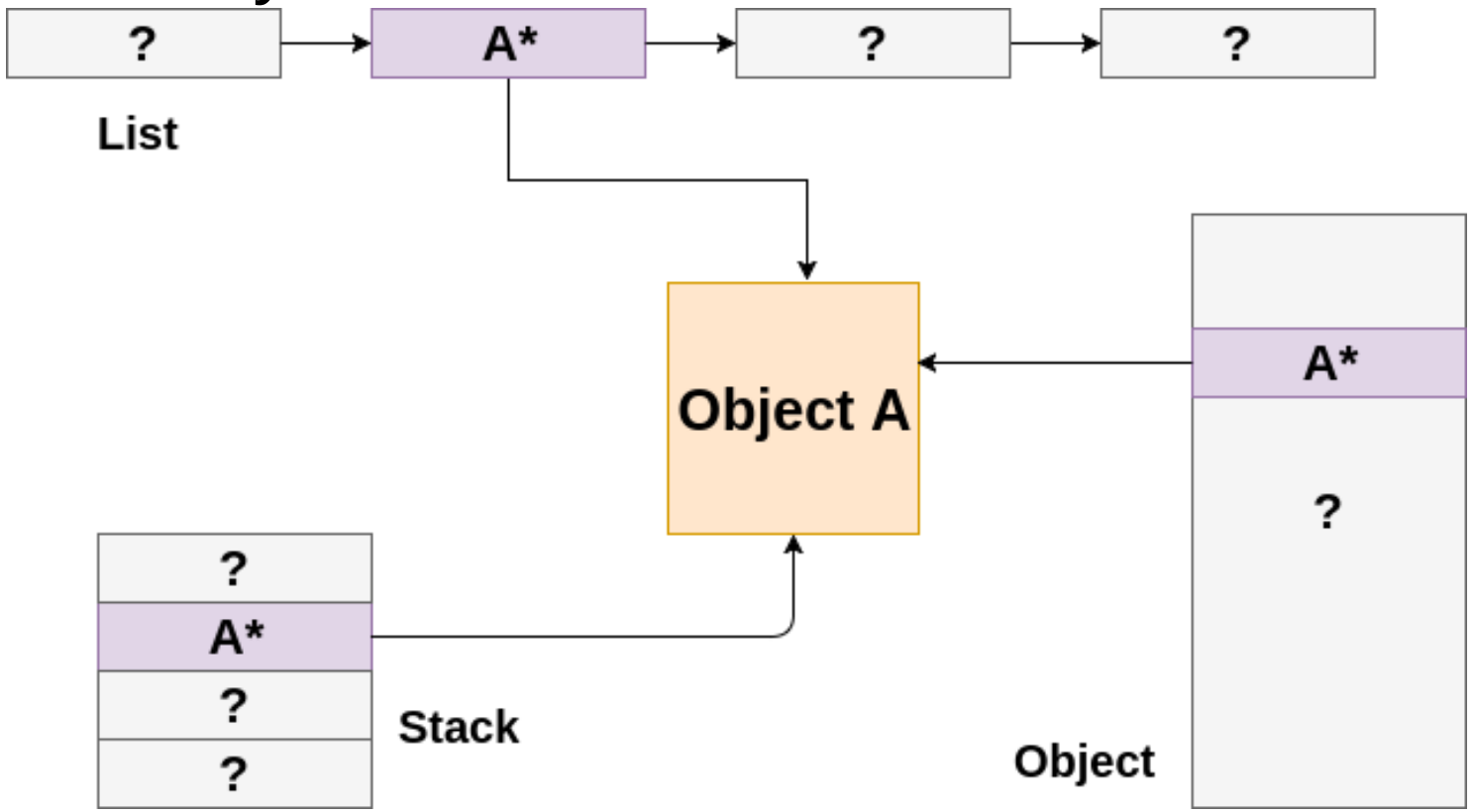
```
A* a_global;

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

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

What is the problem?

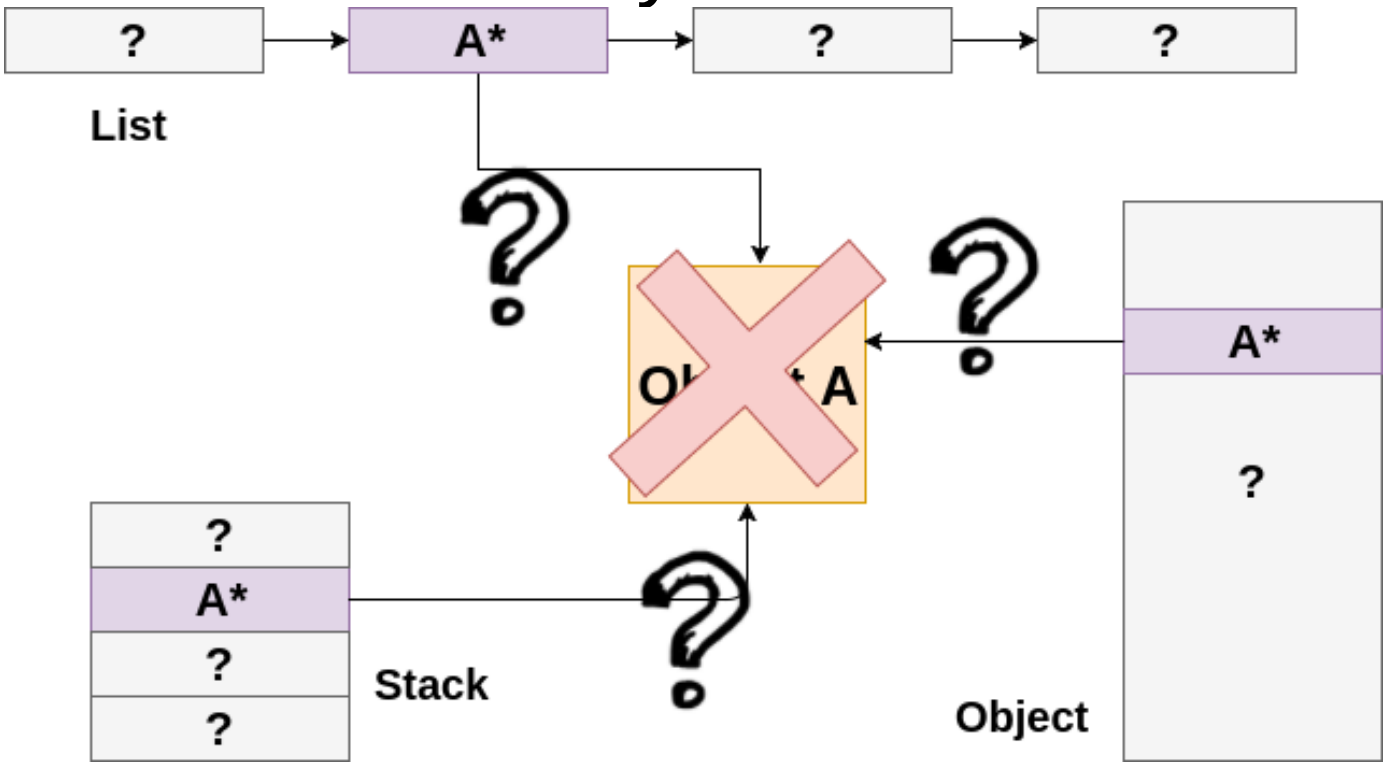
Use After Free

- Looks like a trivial problem but it’s not: in complex systems there may be references (pointers) to an object from different places and can be even modified concurrently



Use After Free

- If an object is deleted, what should be done with references? While deleting references, what would happen if a different thread uses a reference concurrently?



Use After Free

- What is the object life cycle and how to manage it?
- Temporary objects
  - Stored in the stack
  - Do not save references in global variables
  - Do not send references up in the stack
  - It’s safe to send references down in the stack

Use After Free

- Pointers in C++11 / boost:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr
- RAII pattern: Resource Acquisition is Initialization
  - Memory is just another resource

Use After Free

- std::unique_ptr
  - std::make_unique<A>(...);
  - There is no copy constructor, only move constructor
  - 1 object 1 pointer relationship
  - No synchronization cost
  - Small memory footprint: size of a raw pointer
  - Raw pointer can be accessed and “→” operator used

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>(...);
  - There is copy constructor
  - 1 object, 1 or more pointers relationship
  - Synchronization cost is paid. Object is destroyed when the last reference is lost
  - Memory footprint: (raw pointer size)*2
  - Raw pointer can be accessed and “→” operator used

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*>) }
```

Copy constructor

Use After Free

- std::weak_ptr
  - Created in relationship to a shared_ptr
  - Does not count toward object's destruction
  - weak ptr can only be used to obtain a shared_ptr (if the object was not destroyed)
    - While object is in use, a shared ptr that prevents deletion exists

Use After Free

- Good practice: assign NULL value to variables that held an object pointer after freeing it
- Languages like Java, .NET, Python, etc. do not allow to manage memory manually *
  - * with the exception of specific APIs (I.e. Unsafe in Java)
  - Polymorphism is implemented with vtables too
  - Not vulnerable to Use After Free unless there is a bug in the VM
    - Performance and memory footprint costs are paid

Demo 9.2

Stack Use After Free exploitation (kernel space)

Use After Free

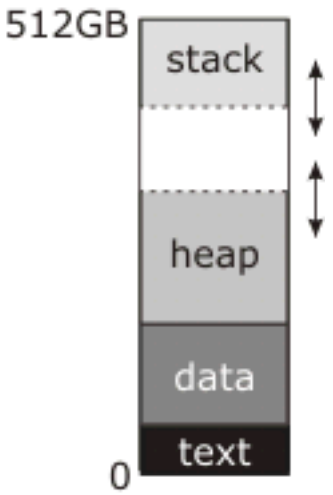
- Exploiting Use After Free in heap has additional challenges: how to overwrite freed space? How to predict memory location where “fake vtable” would be located?
- Heap is a memory area where process can store variable (and unknown) length data, in run time. I.e. arrays, streams, objects, etc.
- Processes generally use dynamic memory allocators in user space (provided by the operating system) to manage the Heap:
  - Simplification or abstraction (reserved vs. committed memory)
  - Granularity (allocate a few bytes only)
  - Contiguous memory at virtual addressing level (not necessarily at physical addressing level)
  - Anti-fragmentation and memory management (caches)

Use After Free

- Every process has at least 1 heap
- Windows has multiple APIs to manage memory and supports multiple heaps:
  - HeapCreate / HeapDelete
  - HeapAlloc / HeapFree
  - VirtualAlloc / VirtualFree
  - malloc (MSVCRT)
- In Linux
  - mmap is used to allocate memory segments
  - brk / sbrk are used increase or reduce the heap size
  - malloc (glibc)

Use After Free

- brk syscall (Linux)
  - Defined in mm/mmap.c (kernel)
  - Changes “data segment” size (heap)
    - This means mapping or unmapping physical memory
  - Heap grows towards higher virtual memory addresses (stack towards lower virtual memory addresses)



Use After Free

- brk syscall (Linux)

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

include/linux/mm_types.h (Linux kernel)

mm_struct structure describes a process memory, at kernel level. In particular, start_brk and brk show the heap location.

Use After Free

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

When calling “malloc” for the first time, glibc has to initialize and extend the process heap. Brk syscall will be used for that.

Process map before calling malloc:

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

Use After Free

In this example case, glibc executes brk syscall with value 0x623000.

When entering 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

Process heap has size 0 at this moment.

When finalizing 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

Process map when returning from sys_brk syscall:

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

malloc finally returned address 0x602260, within heap segment.

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

Memory addresses when memory chunks are allocated are not completely random. From the previous traces it's possible to assume:

- Allocator tries not to fragment the memory (contiguous allocations)
- Minimum size between 2 chunks (including metadata) is 32 bytes
- Even though addresses are different, endings are equal due to alignment

If a chunk in the middle of this sequence is freed and a new allocation of the same size happens, what's the most likely address for the new 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
…

Freed chunk: **0x1f386f0**
New buff: **0x1f386f0**

- Looks like there is a cache: next object is allocated in the location of the last freed one. This makes sense to take advantage of the CPU row cache.

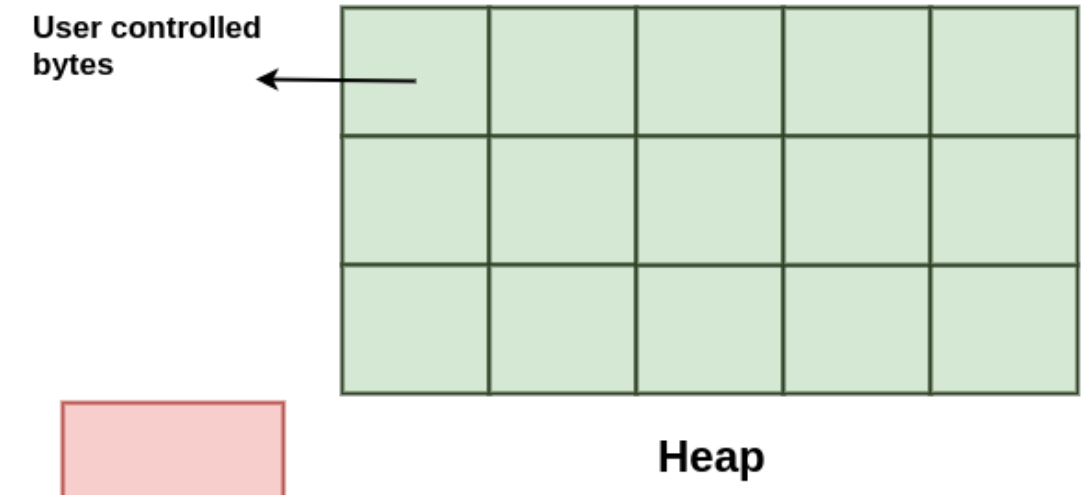
Use After Free

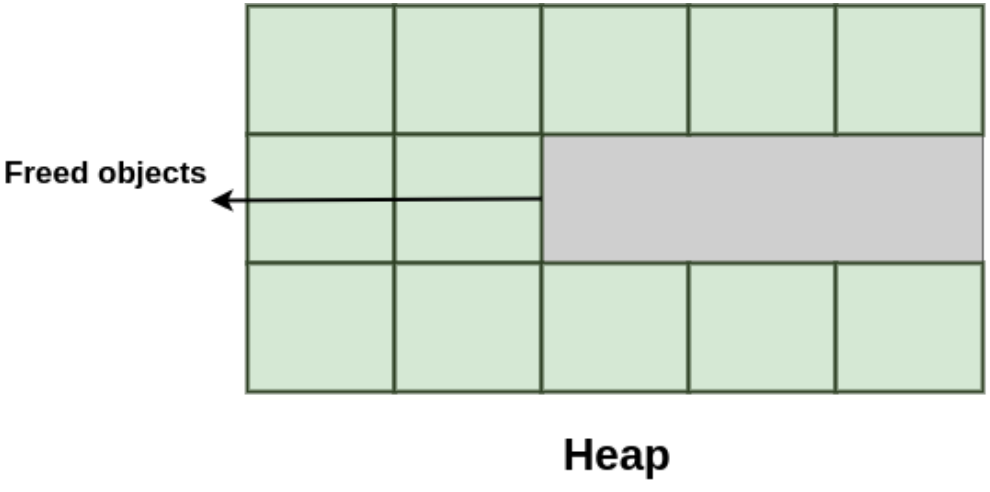
- Heap spray technique consists of generating memory allocations of convenient sizes to predict the layout, with a reasonable certainty
- It’s not a vulnerability itself, but some allocators try to detect sprays, randomize allocations or separate them in different heaps to make exploitation more difficult (heap isolation)
- There isn’t a universal technique: has to be adjusted to each dynamic allocator and heap
- It’s not a 100% reliable technique
- In 32 bit platforms a memory exhaustion can be generated because virtual addresses range is comparable to physical addresses range. In 64 bits is not possible

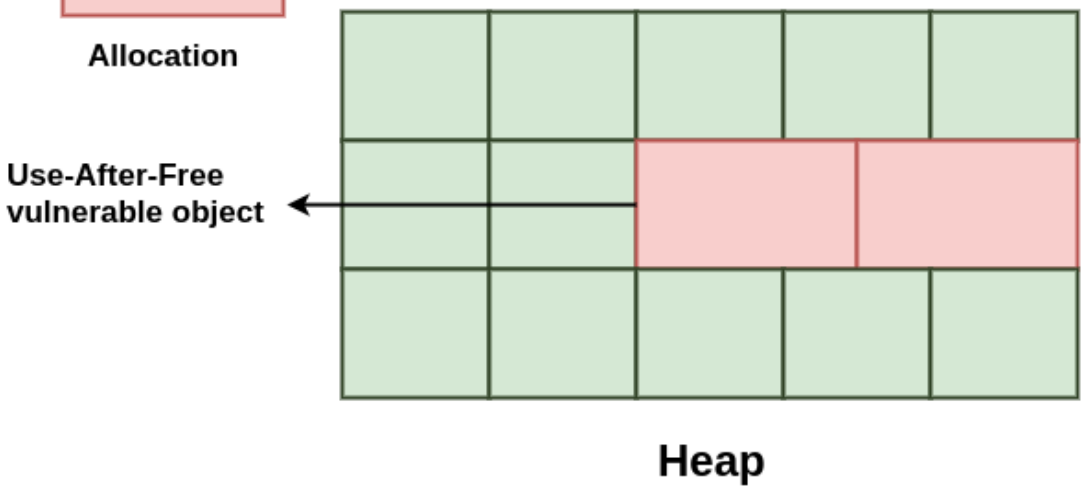
Use After Free

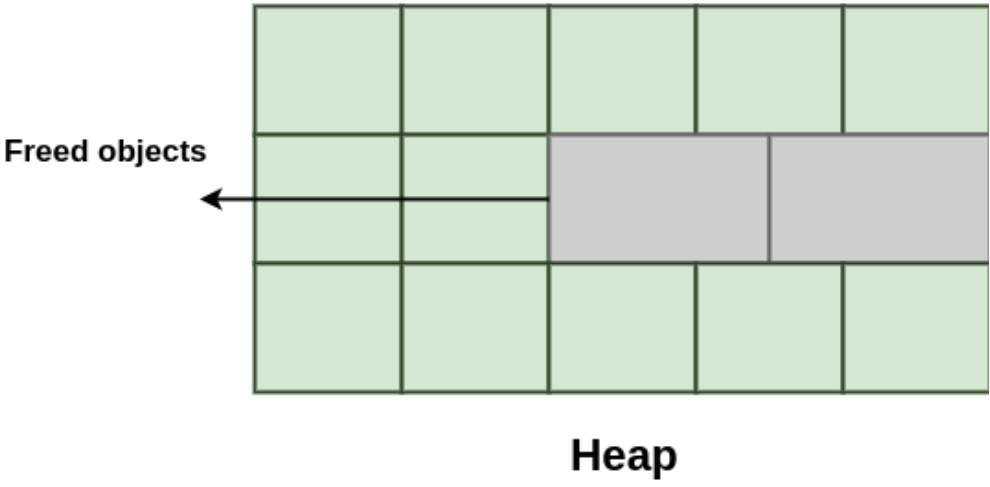
- Example:
  - Objects of a fixed size are allocated all over the heap
  - 1 or more objects in the middle of the heap are freed
  - The allocation of the object vulnerable to use-after-free is generated. This object is then freed. The exact location is not known
  - Objects are conveniently freed
  - New objects of a convenient size are allocated such that there is an overlap with the freed object (vulnerable to use-after-free)
  - We never know exact locations: everything is based on a relative overlap (offsets)

Use After Free

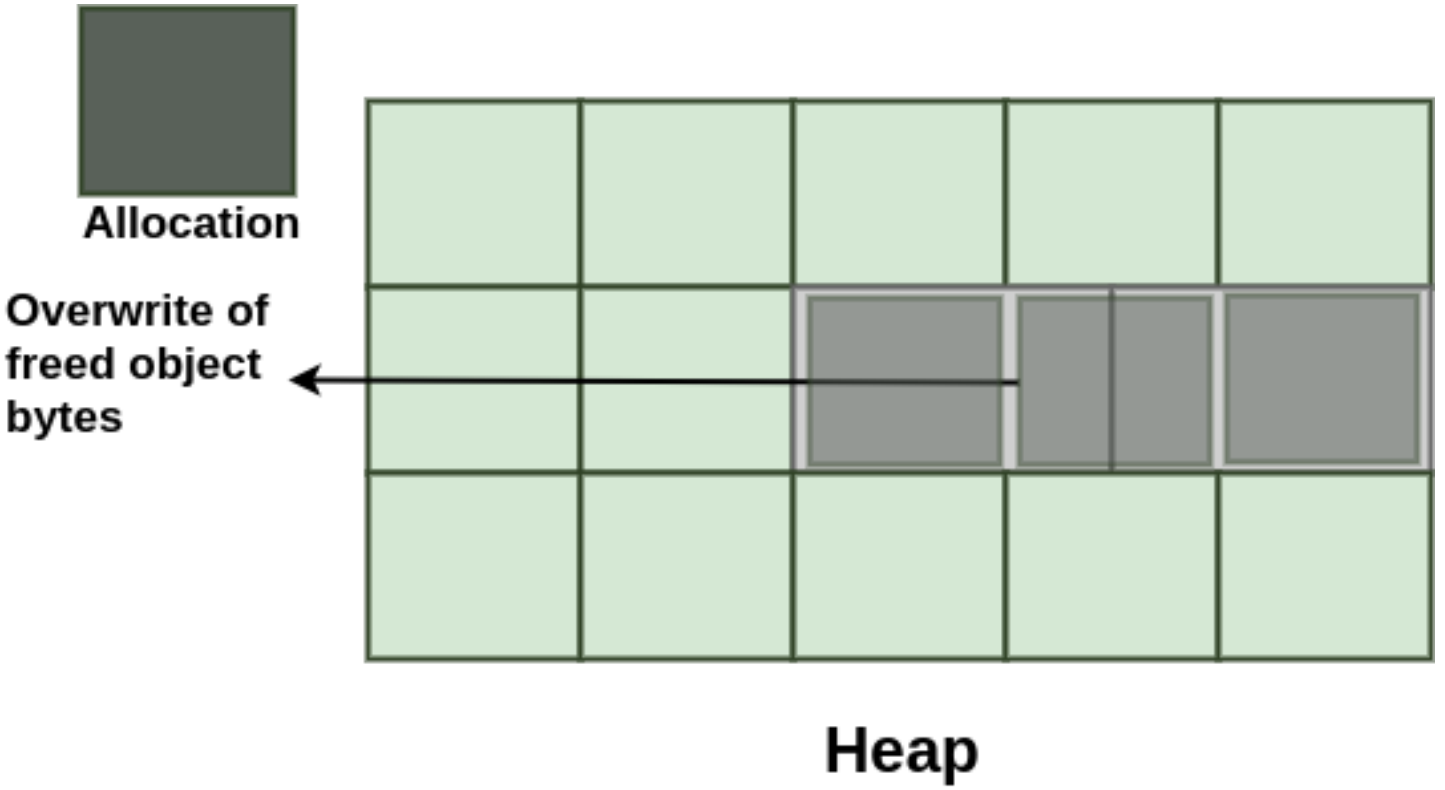








Use After Free



Use After Free

- How to generate allocations?
  - That depends on the target process. In example, in a browser creating arrays from JavaScript will cause the creation of arrays in the native heap
  - Arrays and strings are interesting for the attacker because of the number of bytes directly controlled in the heap. For this reason, separate heaps are commonly used for them. Other objects may allow to control a smaller number of bytes but may be interesting too
  - It’s possible to play with different sizes for the allocations to be managed by different dynamic allocators, and minimize garbage space between them

Use After Free

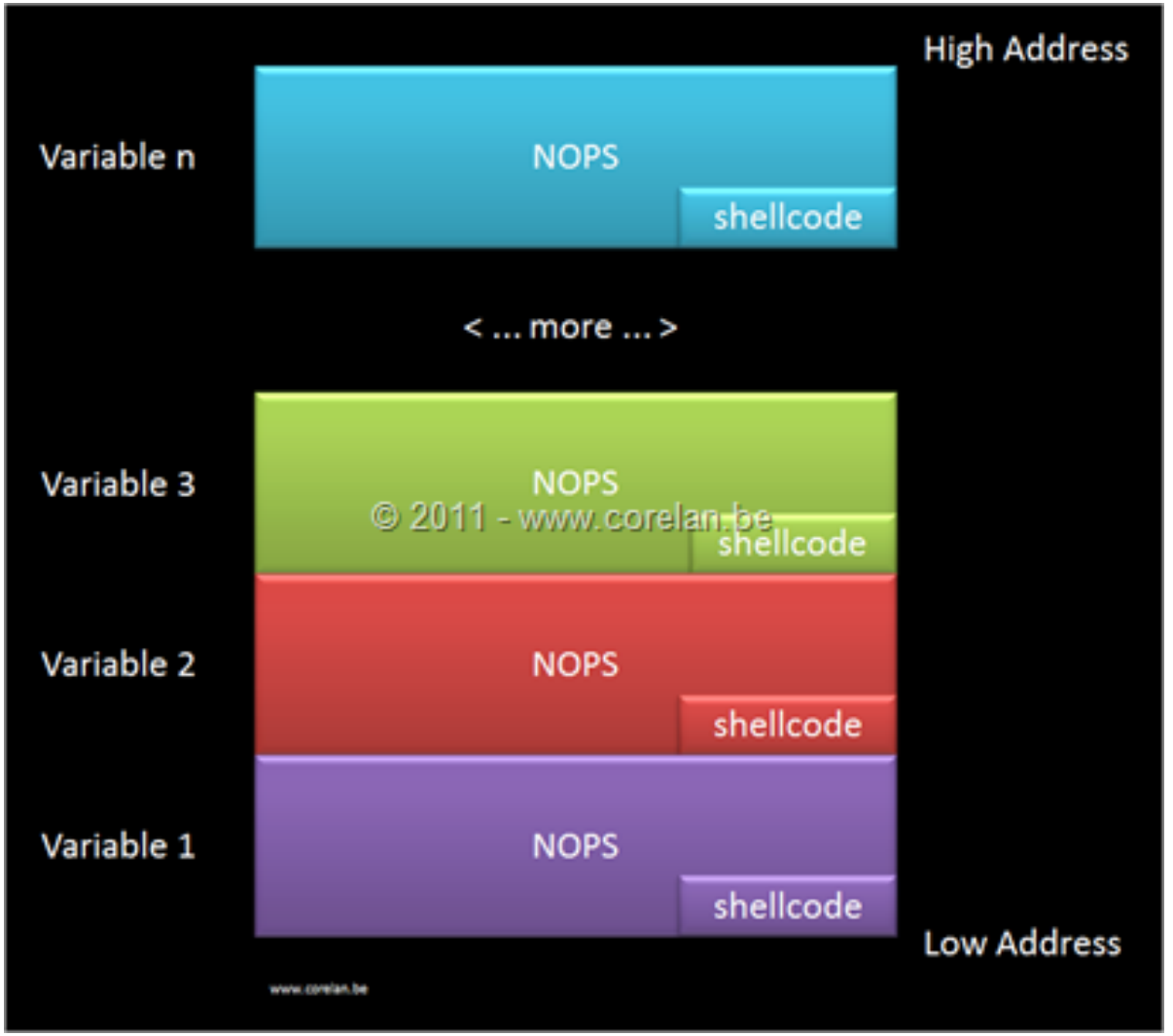


Image from corelan.be

Use After Free

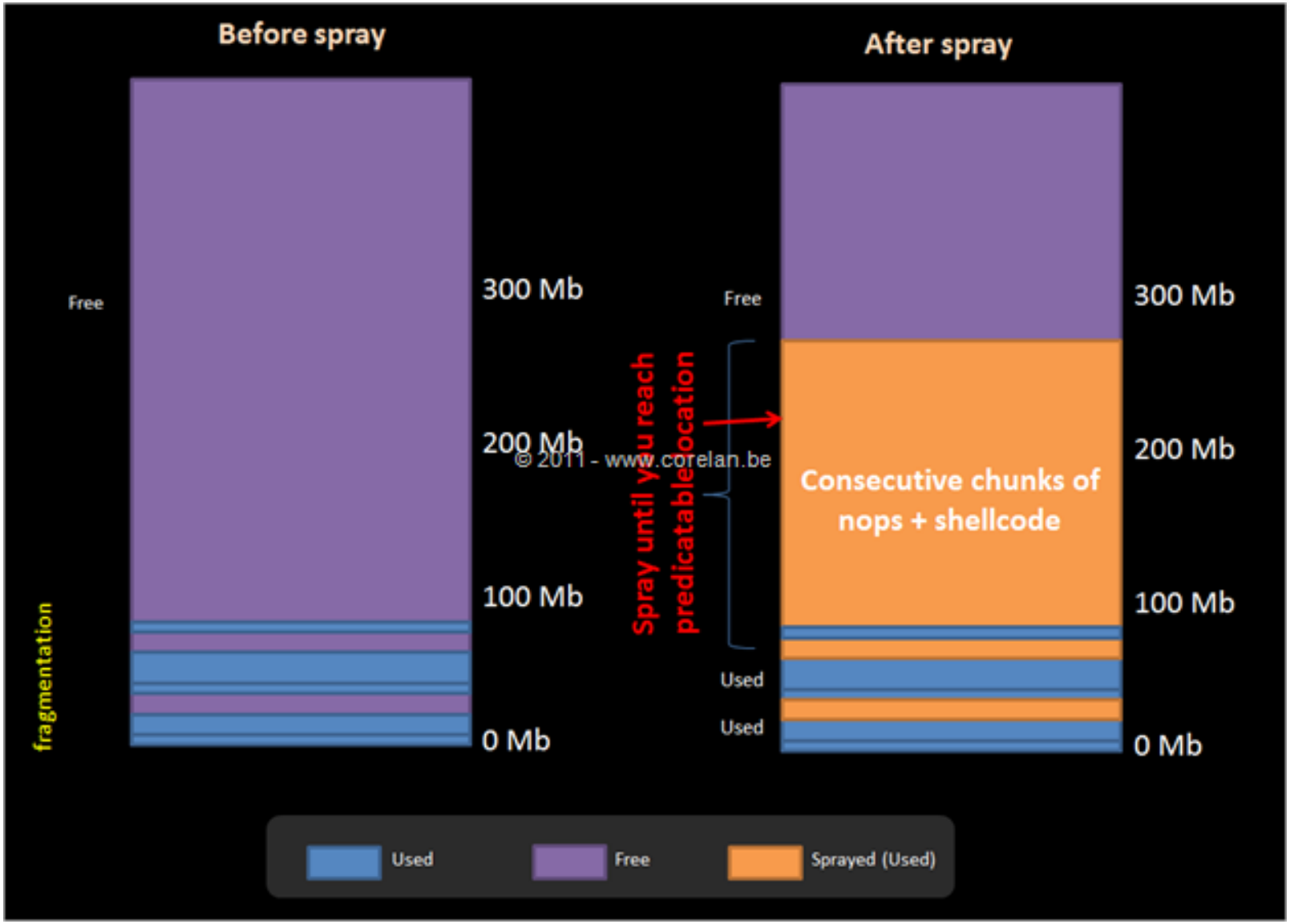


Image from corelan.be

Use After Free

- Memory safety & Type confusion
  - Managed environments (I.e. VMs such as Java, .NET, etc.) ensure that object fields read/write operations obey the correct type, offset and limits
    - I.e. in the object 0x0 offset there is an int, 4 bytes long. It's not in the offset 0x1 and it's not 8 bytes long.
  - Read/write operations out of data type boundaries are not allowed, and usage has to be according to the data type (i.e. an int cannot be dereferenced as if it were a pointer to an object)

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

- No code (interpreted bytecode or JITted code) can write a String in “a.intVar1”, or any value in “a.intVar2”
- Bytecode is verified before being interpreted or compiled

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 = pointer to "a"

RDI = pointer to "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 = pointer to “a”

RDI = pointer to “b”

b.intVar2 = a.intVar1 % 3;

```
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)
```

Use After Free

- If the VM is tricked to set a pointer to “a” in RDI, there would be a read/write operation out of the A data type boundaries. Memory from a contiguous heap object could be overwritten
- Let’s assume that A data type has a reference to B in its first field, over which read/write operations are performed

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 = pointer to "a"
EDI = a.refToB (compressed pointer to "b")

RDI = a.refToB (uncompressed pointer to "b")

Use After Free

- If we manage to set a b’ object of B data type in freed memory where an A object was located (keeping the “a” reference), we have control over each read/write operation:
  - b’.intVar1 = set the memory address
  - a.refToB.intVar1 = set the value

Lab

Lab 9.1: Use After Free exploitation in stack (user space)

Lab

Lab 9.2: Exploitation with Heap Spray

Set EIP to address 0x41414141

References

- 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