

# *Using Immunity Debugger to Write Exploits*

Dave Aitel, Nicolas Waisman

[dave@immunityinc.com](mailto:dave@immunityinc.com)

nicolas.waisman@immunityinc  
.com



Security Research

# Who am I?

- CTO, Immunity Inc.
- Responsible for new product development
  - Immunity Debugger
  - SILICA
  - Immunity CANVAS

# Software companies now understand the value of security

- Over the past few years regular users have become more aware of security problems
- As a result 'security' has become a valuable and marketable asset
- Recognizing this, the computer industry has invested in both hardware and software security improvements

# Immunity Debugger is a strategic answer to defensive advances

- ASLR, NX, /gS and high levels of automated and manual code auditing have raised the bar significantly
- Attackers operate at a distinct disadvantage
  - No source code or internal documentation on structures and protocols
  - Vulnerabilities must be created into reliable exploits

# But attackers have their own resources

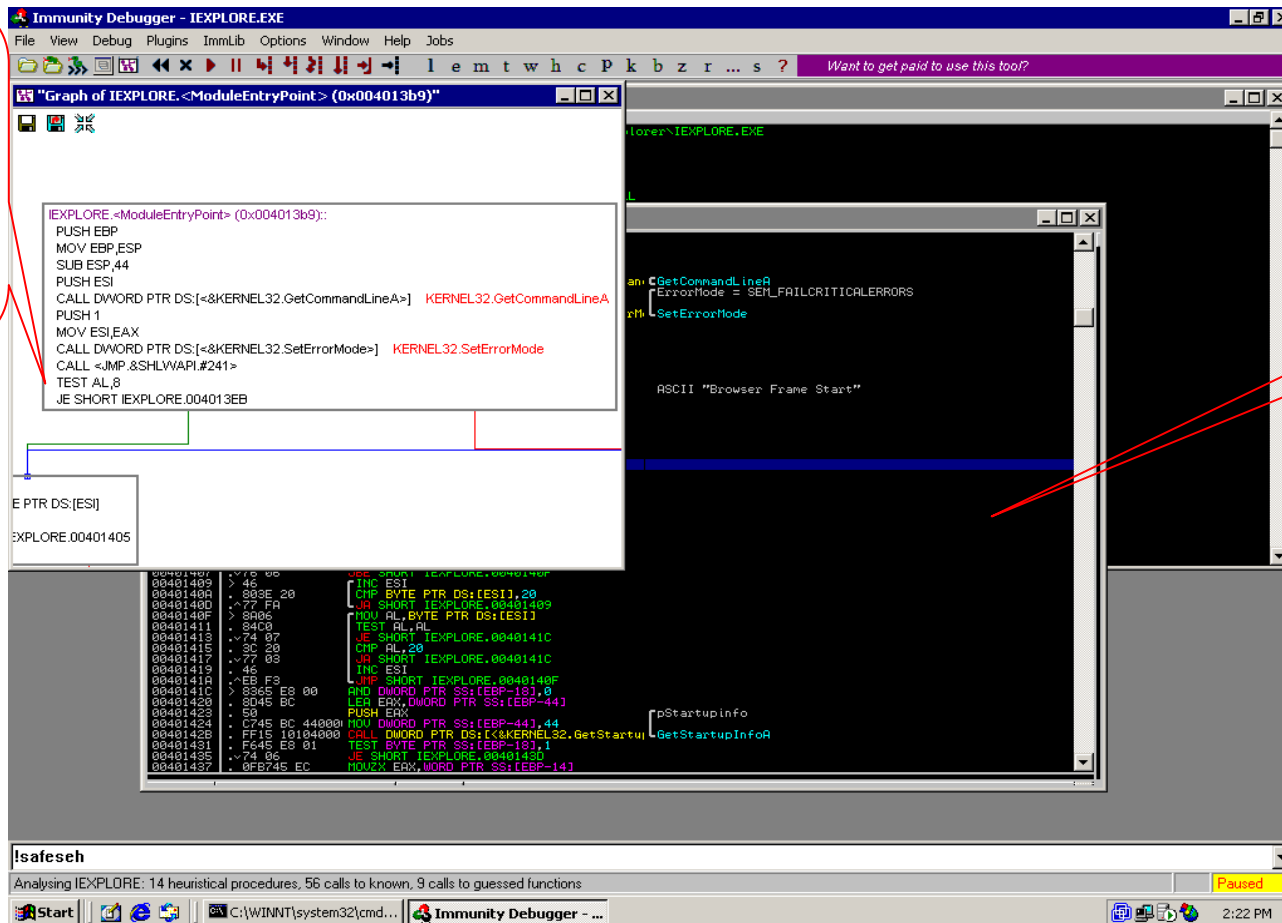
- Used to working in small teams
- Broad range of knowledge (Unix hackers that know Win32, etc)
- Exploit development knowledge is often not fed back to defensive teams, allowing for knowledge leadership over a long time period
  - i.e. new bug classes and attack surfaces

# Attackers will defeat the current generation through profound and rapid tool innovation

- Interfaces
- Analysis engines
- Integration into existing tool-sets
- Teamwork and coordination

# Better interfaces save valuable time

Pure-Python  
Graphing



Usable  
GUI

WinDBG-  
like  
command  
line

# Python integration offers useful analysis

- safeseh discovery
- stack/heap variable sizing
- most importantly – custom automated binary analysis can be written cheaply and easily!
- Static and runtime analysis

# Existing toolsets are also in Python

- Python x86 emulators
- Python exploit frameworks
- Python web application analysis
- PEID
- Non-python toolkits can be accessed easily via Sockets or XML-RPC

# Hackers already work in teams...

- But their tools don't – yet
- Ongoing efforts include
  - SVN + Debugger
  - Portable function fingerprints
  - Global RE database
- While previous efforts have broken ground in team binary analysis, in a year, this will be the default mode of operation

# Two examples of how Immunity Debugger changes assessment and exploitation

- File Include/SQL Injection bugs
- Heap Overflows

# SQL Injection/File Include

- Traditionally web applications are looked at via code review or remote blind assessment
  - But complexity is rising and closed source modules are common
- With ID's `sql_hooker.py` and `sqllistener.py`
  - All SQL Queries get sent to the attacker via XML-RPC
  - Python lets you filter on only interesting results at server side

# Heap overflows are dead, long live heap overflows

- Common technique for reliable exploitation of heap overflows is the write4 primitive
- OS Vendors are well aware of this

And so... heap protection has been introduced

- Windows XP SP2, Windows 2003 SP1 and Vista introduced different heap validity checks to prevent `unlink()` write4 primitives
- Similar technologies are in place in glibc in Linux
- There are no generic ways to bypass the new heap protection mechanisms
  - The current approaches have a lot of requirements: **How do we meet these requirements?**

# XP SP2 makes our work hard

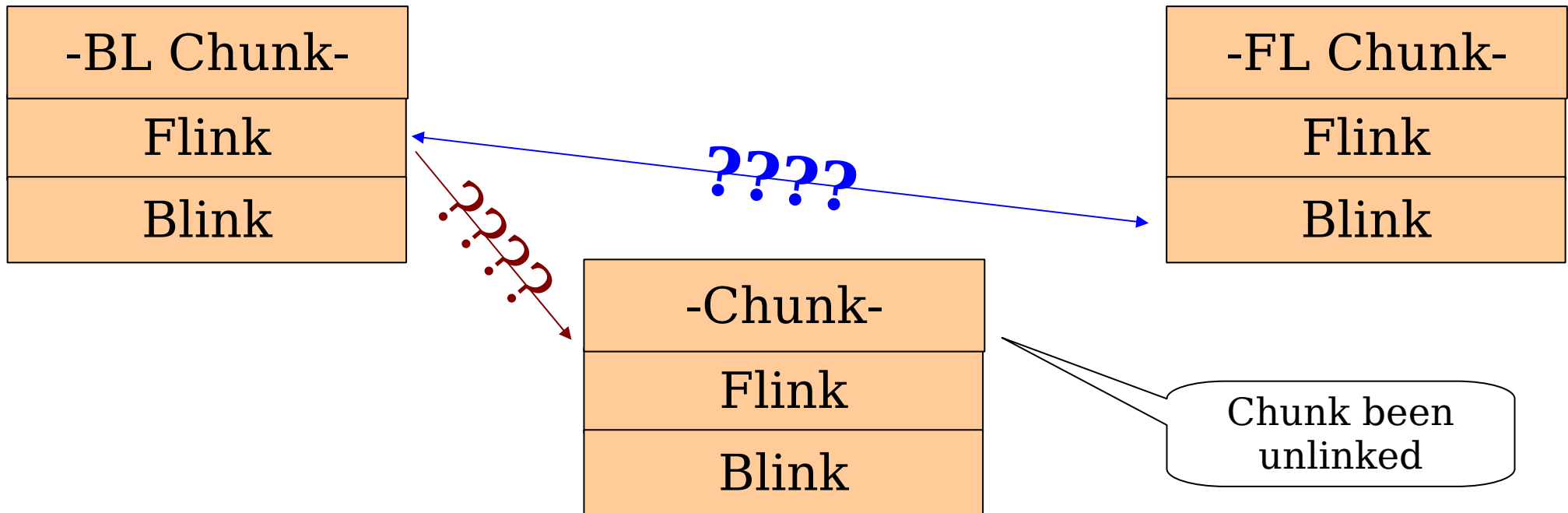
- Windows XP SP2 introduced the first obvious protection mechanism
  - unlinking checks:

```
blink = chunk->blink  
fblink = chunk->fblink
```

```
if blink->fblink == fblink->blink  
and blink->fblink == chunk
```

# and harder...

- Windows XP SP2 introduced the first obvious protection mechanism
  - unlinking checks:



# XP SP2 ( and Vista) introduced more heap protections

- Low Fragmentation Heap Chunks:  
metadata semi-encryption

```
subsegment = chunk->subsegmentcode  
subsegment ^= RtlpLFHKey  
subsegment ^= Heap  
subsegment ^= chunk >> 3
```

# Vista heap algorithm changes make unlink() unlikely

## - Vista Heap Chunks:

metadata semi-encryption and integrity check

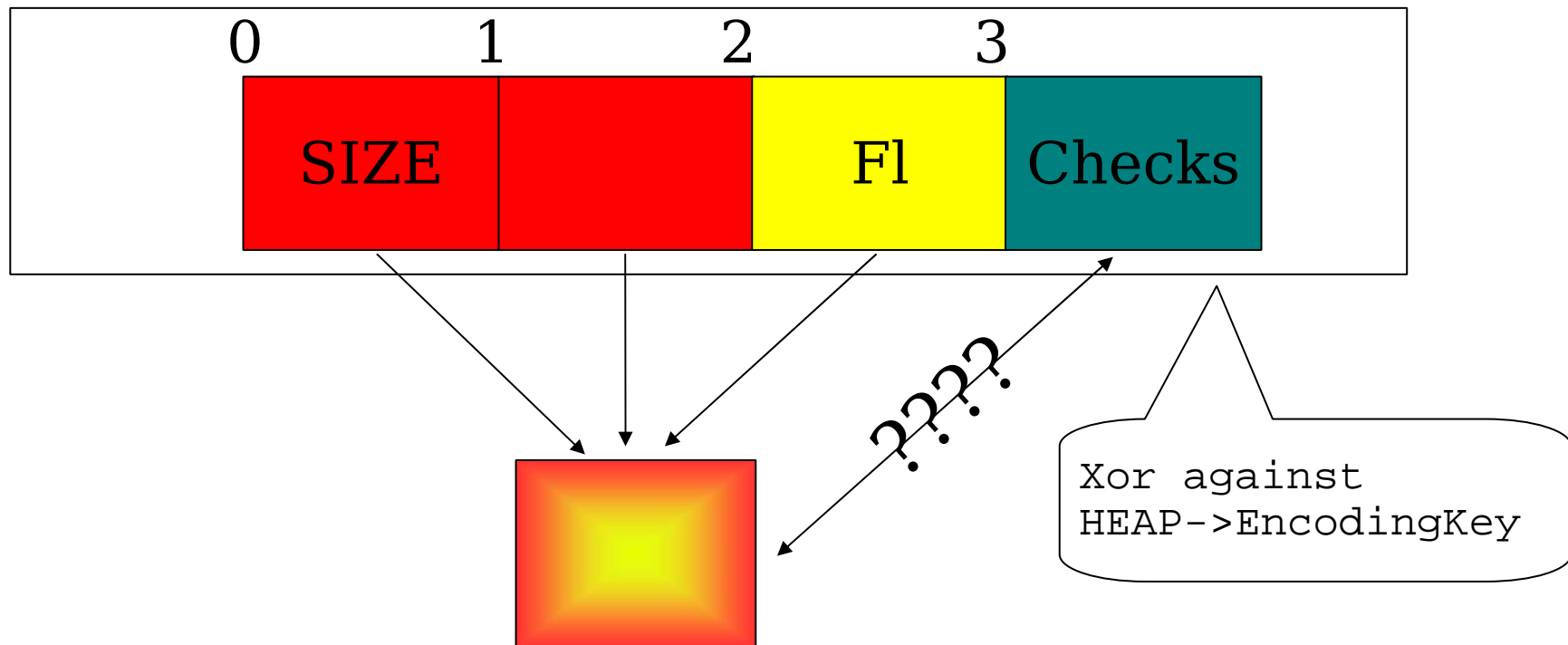
```
* ( chunk )      ^=  HEAP->EncodingKey
checksum         =  (char) * ( chunk + 1 )
checksum        ^=  (char) * ( chunk )
checksum        ^=  (char) * ( chunk + 2 )

if checksum == chunk->Checksum
```

# Checksum makes it hard to predict and control the header

## - Vista Heap Chunks:

metadata semi-encryption and integrity check



# Other protections in Vista are not heap specific

- Other protection mechanisms:
  - ASLR of pages
  - DEP (Hardware NX)
  - Safe Pointers
  - SafeSEH (stack)
  - etc.

# A lot of excellent work has been done to bypass heap protections

- Taking advantage of Freelist[0] split mechanism (*"Exploiting Freelist[0] on XP SP2"* by Brett Moore)
- Taking advantage of Single Linked List unlink on the Lookaside (Oded Horovitz and Matt Connover)
- Heap Feng Shui in Javascript (Alexander Sotirov)

# We no longer use heap algorithms to get write4 primitives

- Generic heap exploitation approaches are obsolete. There is no more easy write4.
  - Sinan: *“I can make a strawberry pudding with so many prerequisites”*
- Application specific techniques are needed
  - We use a methodology based on understanding and controlling the algorithm to position data carefully on the heap

# We have been working on this methodology for years

- All good heap overflow exploits have been in careful control of the heap for years to reach the maximum amount of reliability
- We now also attack not the heap metadata, but the heap data itself
  - Because our technique is specific to each program, generic heap protections can not prevent it
- Immunity Debugger contains powerful new tools to aid this process

# Previous exploits already carefully crafted the heap

- Spooler Exploit:
  - Multiple Write4 with a combination of the Lookaside and the FreeList
- MS05\_025:
  - Softmemleaks to craft the proper layout for two Write4 in a row
- Any other reliable heap overflow
- These still used write4s from the heap algorithms themselves!

# To establish deterministic control over the Heap you need

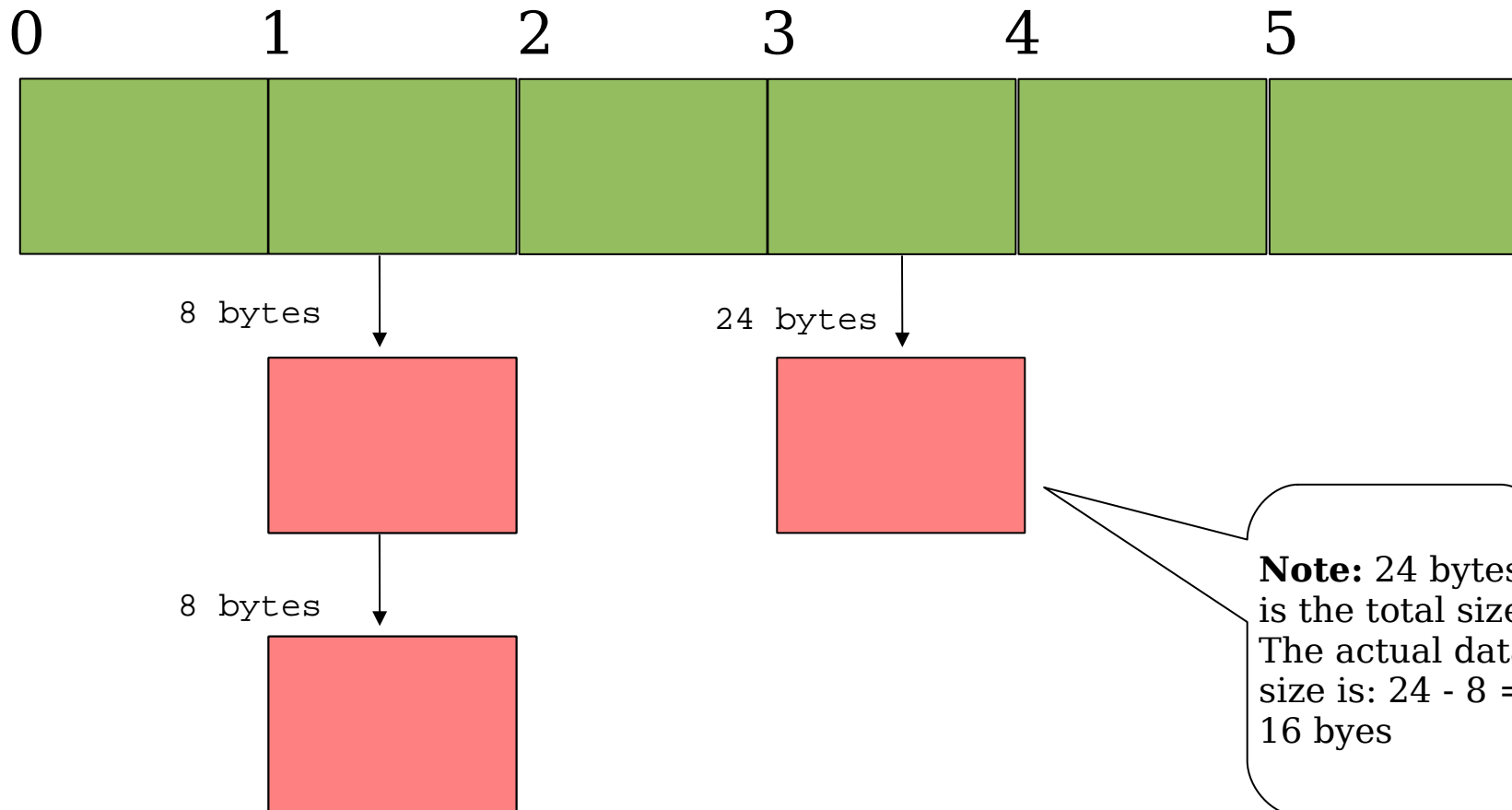
- Understanding of the allocation algorithm
- Understanding of the layout you are exploiting
- A methodology to control the layout
- The proper tools to understand and control the allocation pattern of a process

# The heap, piece by piece

- Understanding the algorithm
  - Structures where chunks are held:
    - Lookaside
    - FreeList
- Understanding Chunk Behaviour
  - Coalescing of Chunks
  - Splitting of Chunks

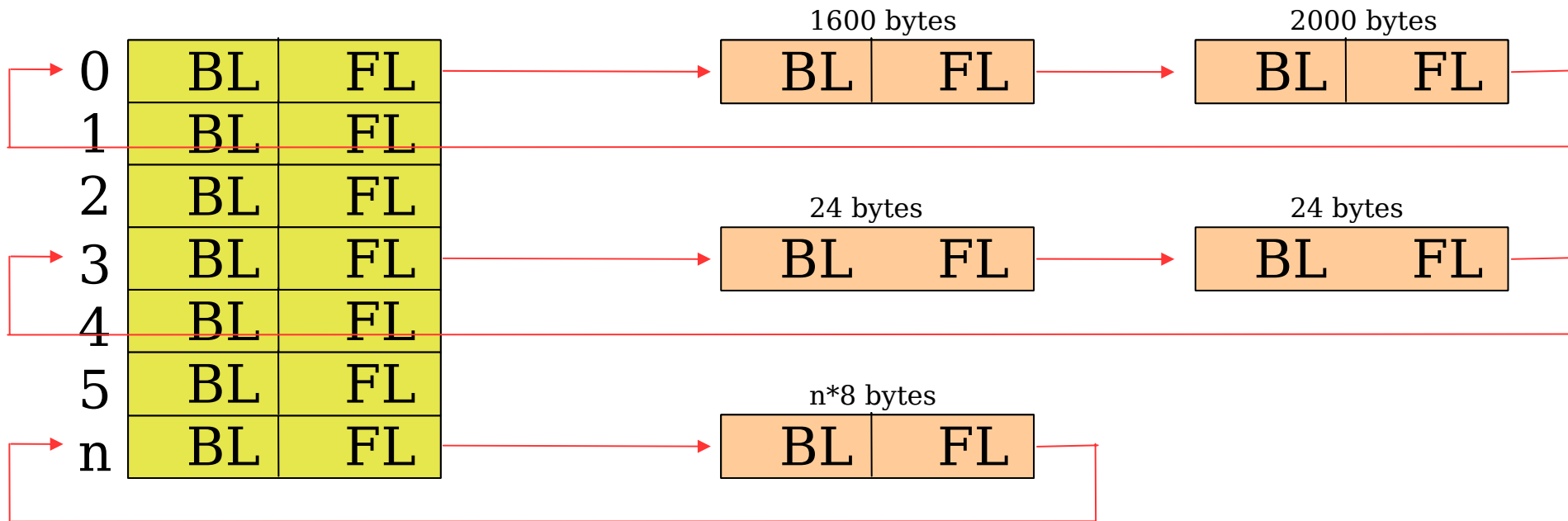
# A quick look at the lookaside

- Lookaside



# A quick look at the FreeList data structure

- FreeList

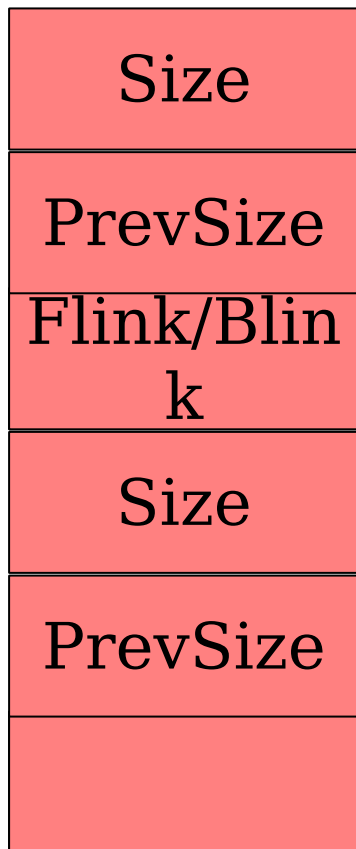


Where  $n < 128$

# Chunk coalescing: contiguous free chunks are joined to minimize fragmentation

**Back\_chunk** →

**ptr** →



**PSize =  $*(ptr+2)$**

**Back\_chunk =  $ptr - (PSize * 8)$**

**if Back\_chunk is not BUSY:  
    unlink(Back\_chunk)**

# Chunks are split into two chunks when necessary

- Chunk splitting happens when a chunk of a specific size is requested and only larger chunks are available
- After a chunk is split, part of the chunk is returned to the process and part is inserted back into the FreeList

# The life-cycle of a heap overflow

- There are four distinct segments in a heap exploit's life that you need to understand and control:
  - Before the overflow
  - Between the overflow and a “Write4”
  - Between the “Write4” and the function pointer trigger
  - Hitting payload and onward (surviving)

} Might be the same

# Heaps do not all start in the same layout

- With heap overflows it is not always easy to control how an overwritten chunk will affect the operation of the heap algorithm
- Understanding how the allocation algorithm works, it becomes apparent that doing three allocations in a row does not mean it will return three bordering chunks
- Typically this problem is because of “Heap Holes”

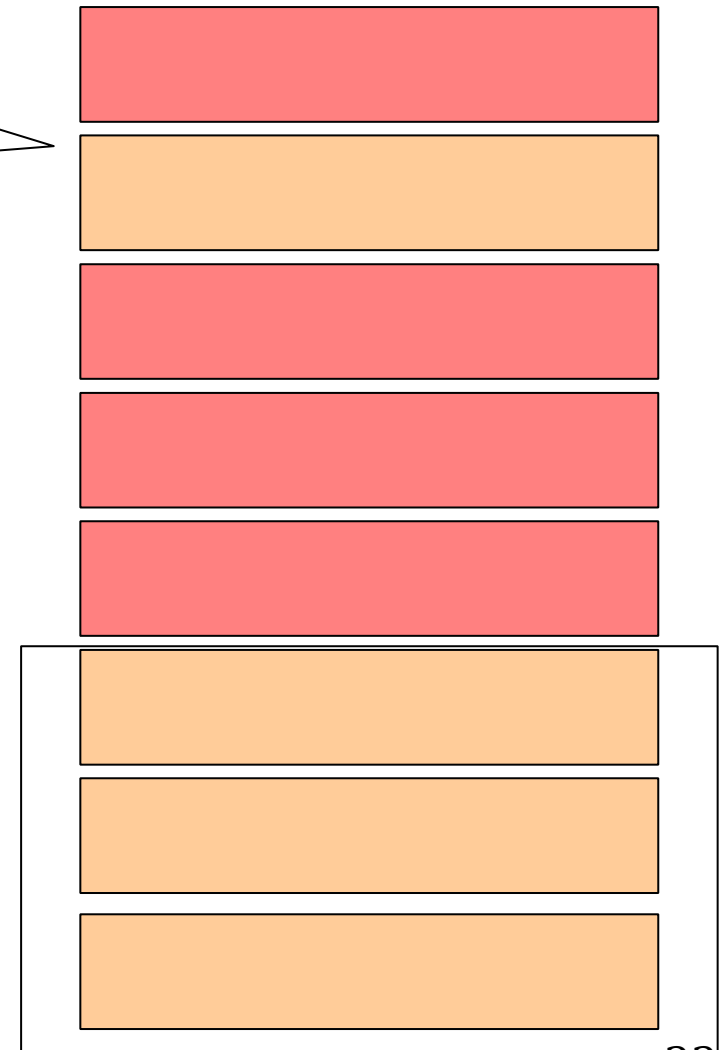
# Heap Holes

- Assume

Chunk is part of the FreeList[97]

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

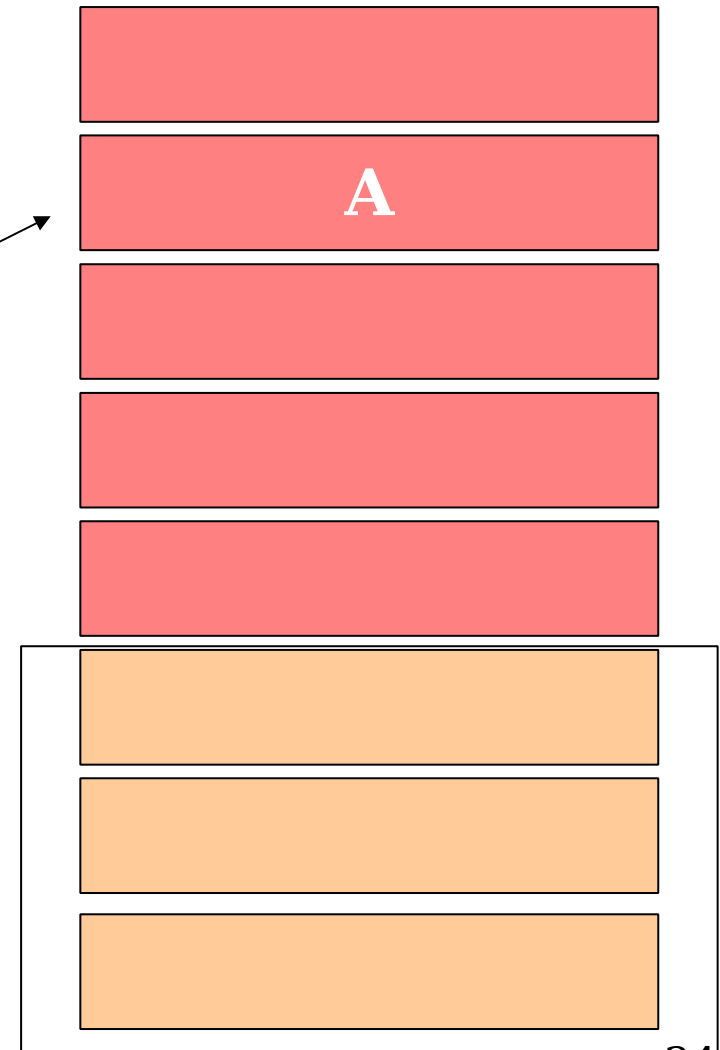


# Heap Holes

- Assuming

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

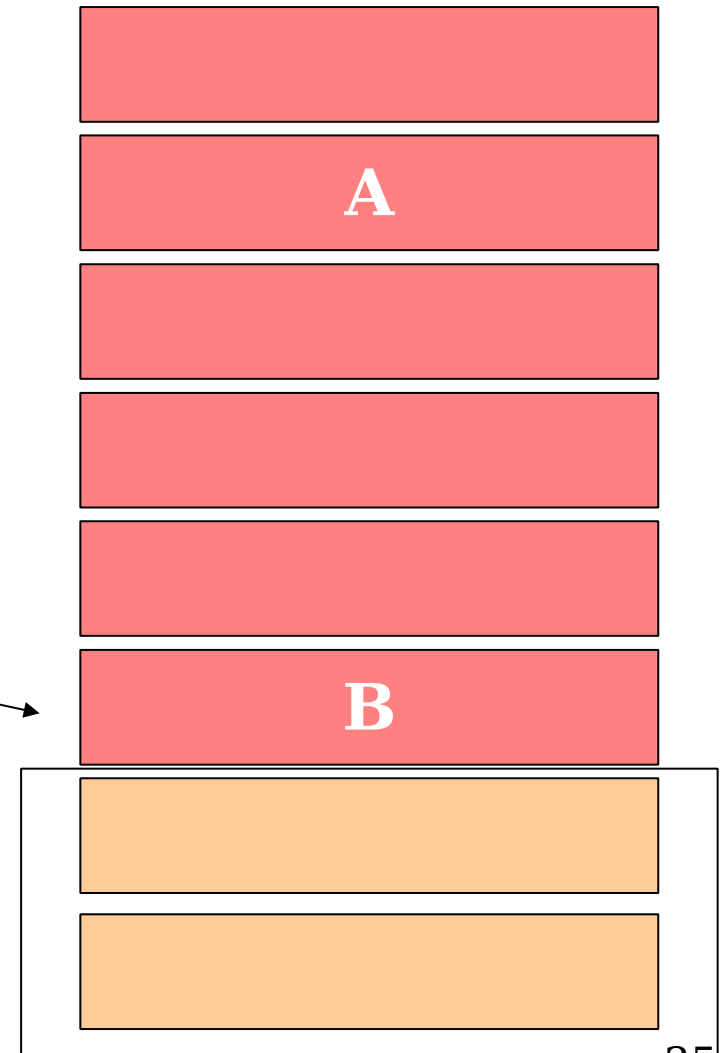


# Heap Holes

- Suppose

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

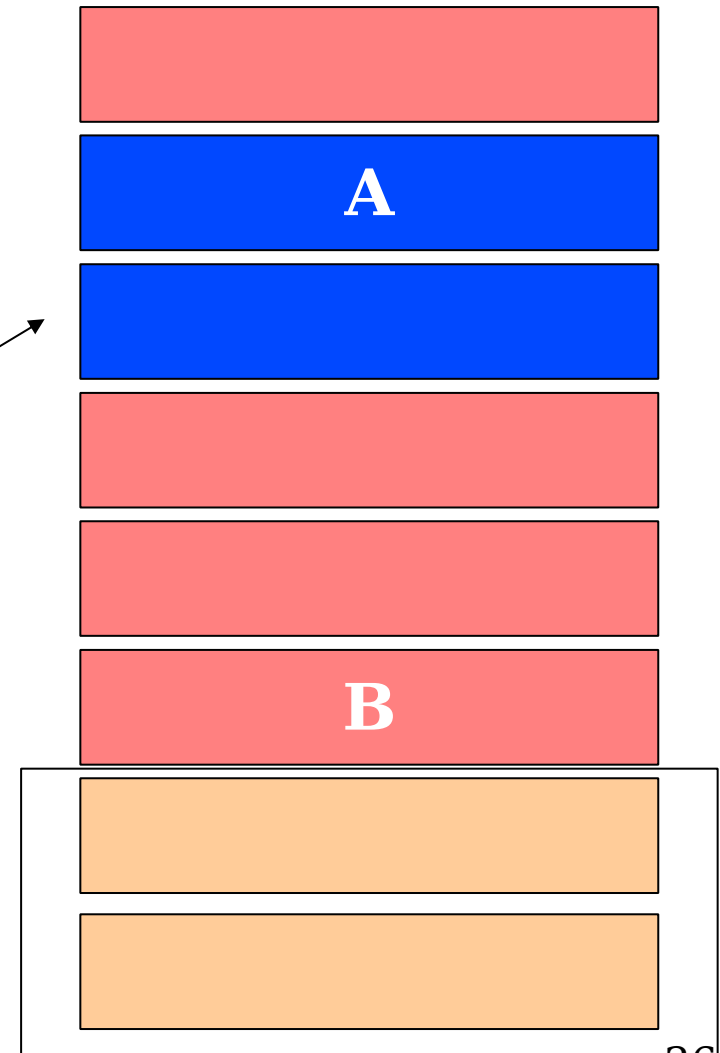


# Heap Holes

- Suppose

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

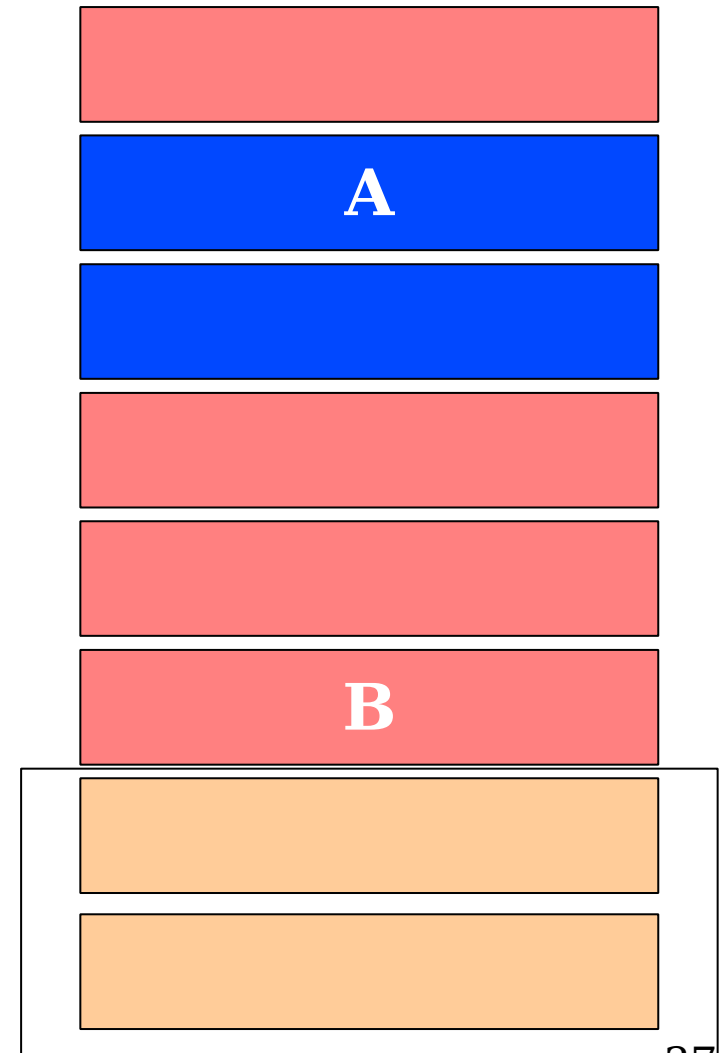


# Heap Holes

- Suppose

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```



# Two types of memory leaks are used in heap exploitation

- A memleak is a portion of memory that is **allocated** but **not deallocated** throughout the life of the target
- There are two types of memleaks:
  - Hard: Memleaks that remain allocated throughout the entire life of the target
  - Soft: Memleaks that remain allocated only for a set period of time (e.g. a memleak based on one connection)

# Several bad coding practises lead to hard memleaks

- Allocations within a try-except block that forget to free in the except block
- Use of RaiseException() within a function before freeing locally bound allocations (RPC services do this a lot)
- Losing track of a pointer to the allocated chunk or overwriting the pointer. No sane reference is left behind for a free
- A certain code flow might return without freeing the locally bound allocation

# Soft memory leaks are almost as useful to exploit writers

- Soft Memleaks are much easier to find:

- Every connection to a server that is not disconnected, allocates memory
- Variables that are set by a command and remain so until they are unset
- Ex:

**X-LINK2STATE CHUNK=A** allocates 0x400 bytes.

**X-LINK2STATE LAST** **CHUNK=A** free that chunk.

# We correct our heap layout with memory leaks

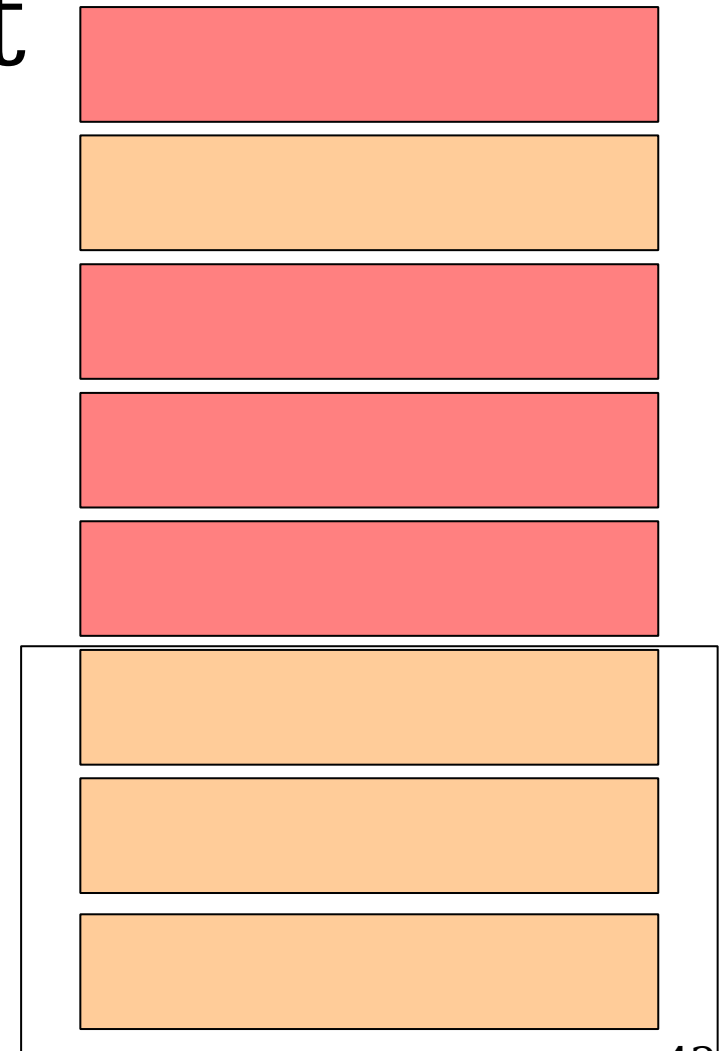
- In summary, memleaks will help us do different things:
    - Empty the Lookaside
    - Empty the FreeList
    - Leaving Holes for a specific purpose
- } Both have the same objective: to allow us to have consecutive chunks

# Heap Rule #1: Force and control the layout

- Assume again

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```



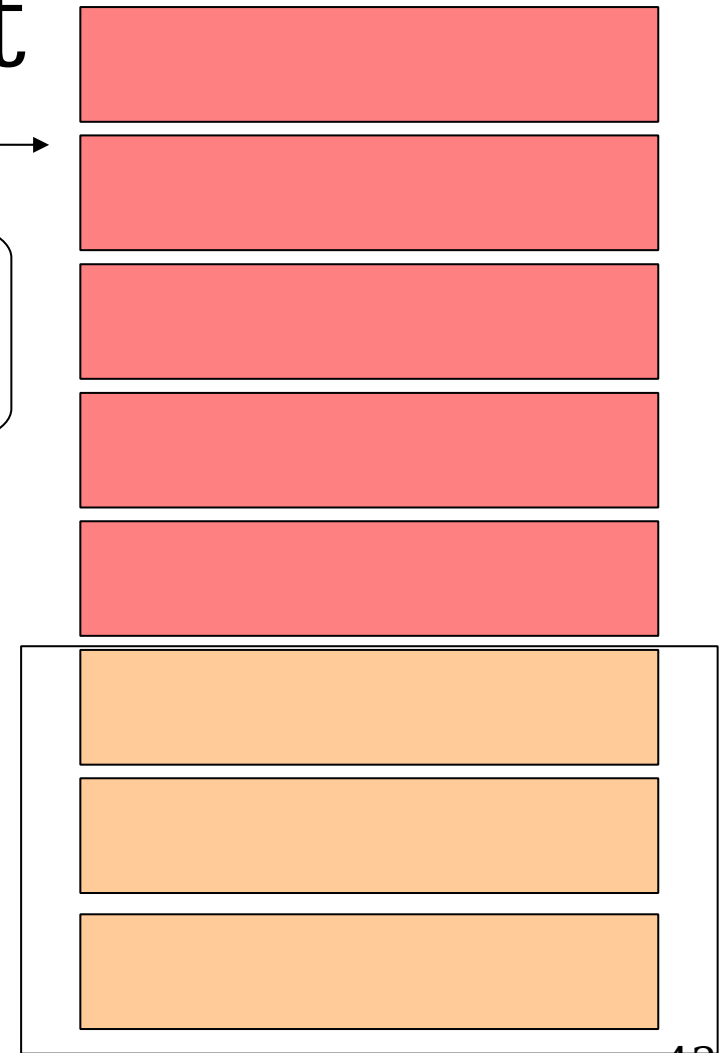
# Heap Rule #1: Force and control the layout

- `memleak(768)` →

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

Calculating size:  
 $768 + 8 = 776$   
 $776/8 = \mathbf{entry\ 97}$

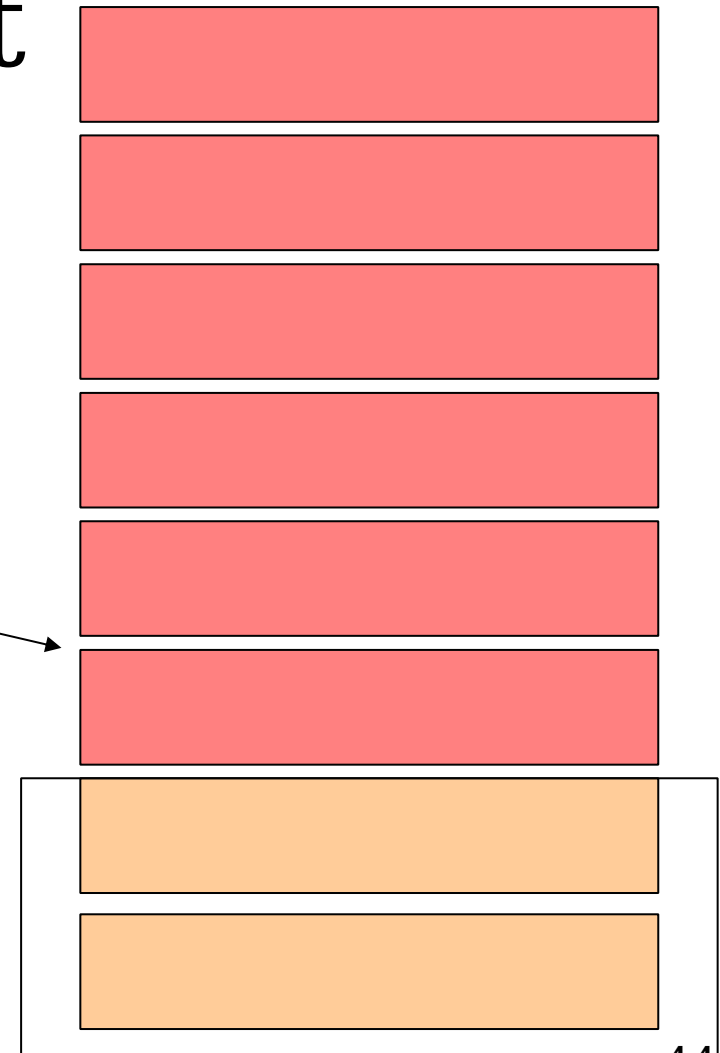


# Heap Rule #1: Force and control the layout

- memleak(768)

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

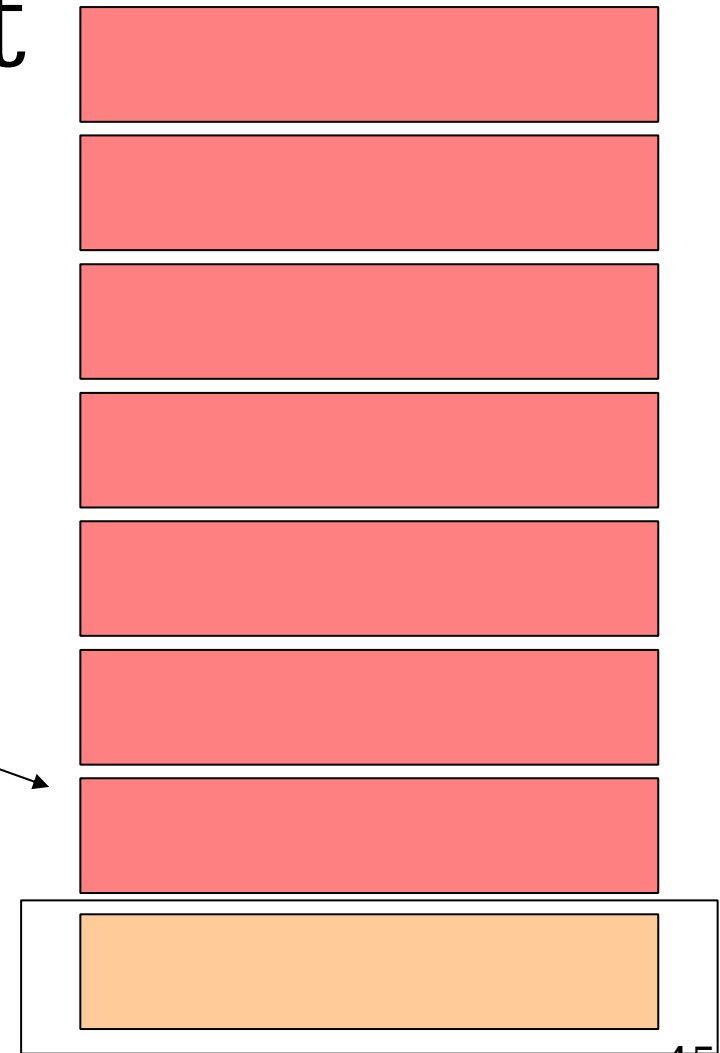


# Heap Rule #1: Force and control the layout

- memleak(768)

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

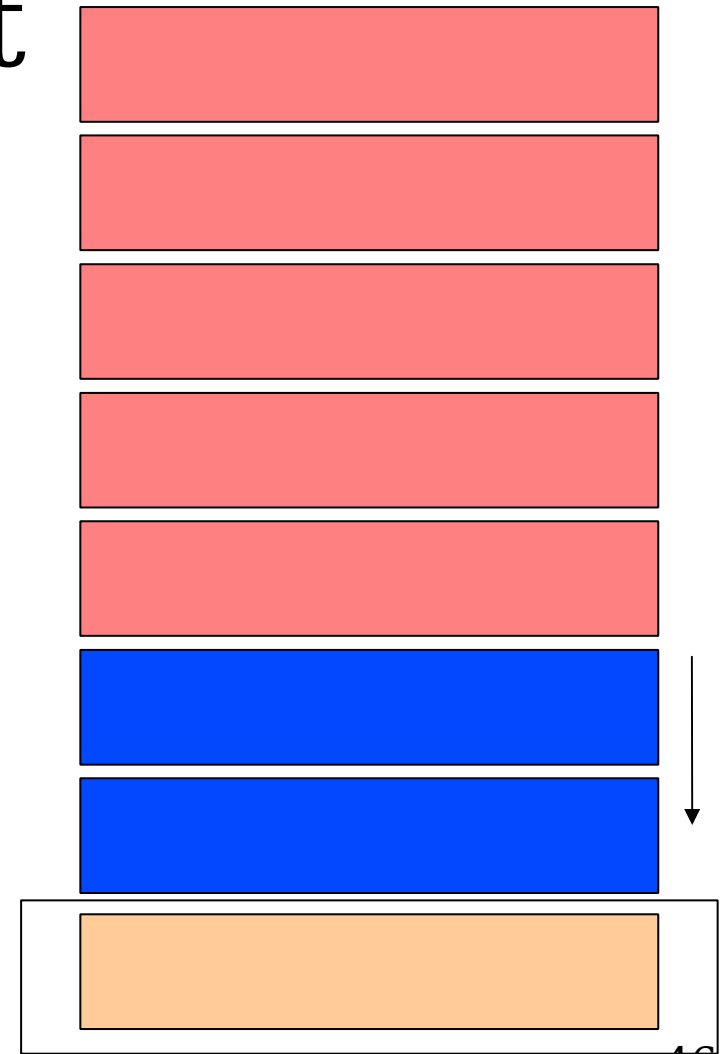


# Heap Rule #1: Force and control the layout

- memleak(768)

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```

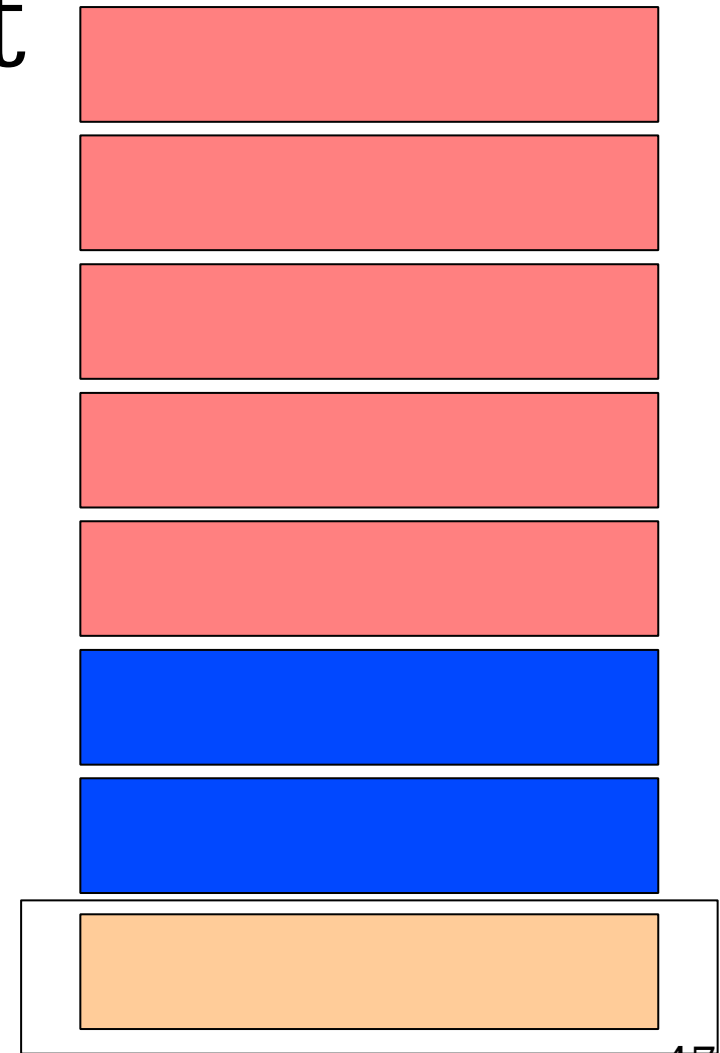


# Heap Rule #1: Force and control the layout

- memleak(768)

Vulnerable(function)

```
A = Allocate(0x300);  
B = Allocate(0x300);  
[...]  
Overwrite(A);  
fn_ptr = B[4];  
fn_ptr("hello world");
```



# Good exploits are the result of Intelligent Debugging

- With the new requirements for maximum deterministic control over the algorithm, exploiting the Win32 heap relies on intelligent debugging
- The need for a debugger that will fill these requirements arises

# Immunity Debugger is the first debugger specifically for vulnerability development

- Powerful GUI
- WinDBG compatible commandline
- Powerful Python based scripting engine

# Immunity Debugger's specialized heap analysis tools

- A series of scripts offering everything needed for modern Win32 Heap exploitation

`!heap`

`!searchheap`

`!funsniff`

`!heap_analyze_chunk`

`!hippie`

`!modptr`

# Immunity Debugger

- Dumping the Heap:
  - !heap -h ADDRESS

- Scripting example:

```
pheap = imm.getHeap( heap )  
for chunk in pheap.chunks:  
    chunk.printchunk()
```

CPU - thread		Log data	
Address	Message	Address	Message
77FCC3E9	8945		
77FCC3EC	8306		

### Heap dump 0x00820000

Address	Chunks
0x00820468	[094] 0x00820468 -> [ 0x00820468   0x00820468 ]
0x00820470	[095] 0x00820470 -> [ 0x00820470   0x00820470 ]
0x00820478	[096] 0x00820478 -> [ 0x00820478   0x00820478 ]
0x00820480	[097] 0x00820480 -> [ 0x00820480   0x00820480 ]
0x00820488	[098] 0x00820488 -> [ 0x00820488   0x00820488 ]
0x00820490	[099] 0x00820490 -> [ 0x00820490   0x00820490 ]
0x00820498	[100] 0x00820498 -> [ 0x00820498   0x00820498 ]
0x008204a0	[101] 0x008204a0 -> [ 0x008204a0   0x008204a0 ]
0x008204a8	[102] 0x008204a8 -> [ 0x008204a8   0x008204a8 ]
0x008204b0	[103] 0x008204b0 -> [ 0x008204b0   0x008204b0 ]
0x008204b8	[104] 0x008204b8 -> [ 0x008204b8   0x008204b8 ]
0x008204c0	[105] 0x008204c0 -> [ 0x008204c0   0x008204c0 ]
0x008204c8	[106] 0x008204c8 -> [ 0x008204c8   0x008204c8 ]
0x008204d0	[107] 0x008204d0 -> [ 0x008204d0   0x008204d0 ]
0x008204d8	[108] 0x008204d8 -> [ 0x008204d8   0x008204d8 ]
0x008204e0	[109] 0x008204e0 -> [ 0x008204e0   0x008204e0 ]
0x008204e8	[110] 0x008204e8 -> [ 0x008204e8   0x008204e8 ]
0x008204f0	[111] 0x008204f0 -> [ 0x008204f0   0x008204f0 ]
0x008204f8	[112] 0x008204f8 -> [ 0x008204f8   0x008204f8 ]
0x00820500	[113] 0x00820500 -> [ 0x00820500   0x00820500 ]
0x00820508	[114] 0x00820508 -> [ 0x00820508   0x00820508 ]
0x00820510	[115] 0x00820510 -> [ 0x00820510   0x00820510 ]
0x00820518	[116] 0x00820518 -> [ 0x00820518   0x00820518 ]
0x00820520	[117] 0x00820520 -> [ 0x00820520   0x00820520 ]
0x00820528	[118] 0x00820528 -> [ 0x00820528   0x00820528 ]
0x00820530	[119] 0x00820530 -> [ 0x00820530   0x00820530 ]
0x00820538	[120] 0x00820538 -> [ 0x00820538   0x00820538 ]
0x00820540	[121] 0x00820540 -> [ 0x00820540   0x00820540 ]
0x00820548	[122] 0x00820548 -> [ 0x00820548   0x00820548 ]
0x00820550	[123] 0x00820550 -> [ 0x00820550   0x00820550 ]
0x00820558	[124] 0x00820558 -> [ 0x00820558   0x00820558 ]
0x00820560	[125] 0x00820560 -> [ 0x00820560   0x00820560 ]
0x00820568	[126] 0x00820568 -> [ 0x00820568   0x00820568 ]
0x00820570	[127] 0x00820570 -> [ 0x00820570   0x00820570 ]
0x00820640	0x00820640> size: 0x00000040 (0008) prevsize: 0x000000640 (00c8)
0x00820640	heap: *0x00820000* flags: 0x00000001 (B)
0x00820680	0x00820680> size: 0x00001808 (0301) prevsize: 0x00000040 (0008)
0x00820680	heap: *0x00820000* flags: 0x00000001 (B)
0x00821e88	0x00821e88> size: 0x00002178 (042f) prevsize: 0x00001808 (0301)
0x00821e88	heap: *0x00820000* flags: 0x00000010 (F!T)
0x00821e88	next: 0x00820178 prev: 0x00820178
0x00000000	-----

007)  
)  
042)  
!T)

# Searching the heap using Immlib

- Search the heap

```
- !searchheap
```

```
  what    (size,usize,psize,upsize,flags,address,next,prev)
```

```
  action (=,>,<,>=,<=,&,not,!=)
```

```
  value  (value to search for)
```

```
  heap   (optional: filter the search by heap)
```

- Scripting example:

```
SearchHeap(imm, what, action, value, heap = heap)
```

# Comparing a heap before and after you break it

- Dumping a Broken Heap:
  - Save state:
    - !heap -h ADDRESS -s
  - Restore State:
    - !heap -h ADDRESS -r

# Heap Fingerprinting

- To craft a correct Heap layout we need a proper understanding of the allocation pattern of different functions in the target process
- This means there is a need for fingerprinting the heap flow of a specific function

# Heap Fingerprinting

- `!funsniff <address>`
  - fingerprint the allocation pattern of the given function
  - find memleaks
  - double free
  - memory freed of a chunk not belonging to our current heap flow (Important for soft memleaks)



# Automated data type discovery using Immlib

- As we now know overwriting the metadata of chunks to get a unlink primitive is mostly no longer viable
- The next step of heap exploitation is taking advantage of the **content of chunks**
- We need straightforward runtime recognition of chunk content

# Immunity Debugger offers simple runtime analysis of heap data to find data types

- String/Unicode
- Pointers ( Function Pointer, Data pointer, Stack Pointer)
- Double Linked lists
  - Important because they have their own unlink() write4 primitives!

# Data Discovery

- `!heap -h HEAP_ADDRESS -d`
  - See next slide for awesome screenshot of this in action!

KNOWING YOU'RE SECURE

### Heap dump 0x00c50000

Address	Chunks
0x00c56fb8	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c56fe4	> Pointer: 0x00c550a8 in 0x00c50000!
0x00c56ff0	> Pointer: 0x00070044 in 0x00070000!
0x00c56ff8	> Pointer: 0x00c57218 in 0x00c50000!
0x00c56ffc	> Unicode: ',NoCacheCC'
0x00c56ff0	0x00c56ff0> size: 0x00000220 (0044) prevsize: 0x00000038 (0007)
0x00c56ff0	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c56ff8	> Pointer: 0x00c57218 in 0x00c50000!
0x00c56ffc	> Unicode: ',NoCacheCCCCCCCCCCCCCCCCCCCC'
0x00c57210	0x00c57210> size: 0x00000220 (0044) prevsize: 0x00000220 (0044)
0x00c57210	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c5721c	> Unicode: ',NoCacheCC'
0x00c57430	> Pointer: 0x0044000c in 0x00420000!
0x00c57430	0x00c57430> size: 0x00000060 (000c) prevsize: 0x00000220 (0044)
0x00c57430	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c57444	> Pointer: 0x00c57218 in 0x00c50000!
0x00c57498	> Double Linked List: ( 0x00c50178, 0x00c59358 )
0x00c57490	0x00c57490> size: 0x000018d8 (031b) prevsize: 0x00000060 (000c)
0x00c57490	heap: *0x00c50000* flags: 0x00000000 (F)
0x00c57490	next: 0x00c50178 prev: 0x00c59358
0x00c58d68	0x00c58d68> size: 0x000000f0 (001e) prevsize: 0x000018d8 (031b)
0x00c58d68	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c58e58	0x00c58e58> size: 0x000003f0 (007e) prevsize: 0x000000f0 (001e)
0x00c58e58	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c59248	0x00c59248> size: 0x00000018 (0003) prevsize: 0x000003f0 (007e)
0x00c59248	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c5926c	> Pointer: 0x000ab8f0 in 0x00070000!
0x00c59270	> String: 'LMEH'
0x00c59294	> Pointer: 0x00c59338 in 0x00c50000!
0x00c592c4	> Pointer: 0x00020002 in 0x00020000!
0x00c592e4	> Pointer: 0x00c520c8 in 0x00c50000!
0x00c59338	> Pointer: 0x00c59268 in 0x00c50000!
0x00c5933c	> Unicode: 'IMM2311'
0x00c59358	> Double Linked List: ( 0x00c57498, 0x00c50178 )
0x00c59260	0x00c59260> size: 0x00000020 (0004) prevsize: 0x00000018 (0003)
0x00c59260	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c5926c	> Pointer: 0x000ab8f0 in 0x00070000!
0x00c59270	> String: 'LMEH'
0x00c59280	0x00c59280> size: 0x000000b0 (0016) prevsize: 0x00000020 (0004)
0x00c59280	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c59294	> Pointer: 0x00c59338 in 0x00c50000!
0x00c59330	0x00c59330> size: 0x00000020 (0004) prevsize: 0x000000b0 (0016)
0x00c59330	heap: *0x00c50000* flags: 0x00000001 (B)
0x00c59338	> Pointer: 0x00c59268 in 0x00c50000!
0x00c5933c	> Unicode: 'IMM2311'
0x00c59358	> Double Linked List: ( 0x00c57498, 0x00c50178 )
0x00c59350	0x00c59350> size: 0x000000b0 (0196) prevsize: 0x00000020 (0004)

# Data Discovery can be scripted easily

```
import libdatatype

dt = libdatatype.DataTypes( imm )

ret = dt.Discover( memory, address, what)

memory      memory to inspect

address     address of the inspected memory

what        (all, pointers, strings,
             asciistrings, unicodestrings,
             doublelinkedlists, exploitable)

for obj in ret:
    print ret.Print()
```

# Heap Fuzzing helps you discover a way to obtain the correct layout

- Sometimes controlling the layout is not as easy as you think, even though it sounds straightforward in theory
- From this the concept of Fuzzing the Heap arises, to help in discovering the correct layout for your process (manually or automatically)

# Heap Fuzzing

- `!chunkanalyzehook`
- Get the status of a given chunk at a specific moment. Answers the common questions:
  - What chunks are bordering your chunk?
  - What is the data in those chunks?

# Heap Fuzzing

- *Run the script, Fuzz and get result...*

- usage:

`!chunkanalyzehook (-d) -a ADDRESS <exp>`

`-a ADDRESS`      address of the hook

`-d`                find datatypes

`<exp>`            how to find the chunk

ex: *`!chunkanalyzehook -d -a 0x77fcb703 EBX - 8`*

CPU - thread 0000025C, module win32spl

76A57C06	E8 DFA5FFFF	CALL <JMP.&SPOOLSS.DllAllocSplMem>
76A57C0B	8BF8	MOV EDI,EAX
76A57C0D	3BF8	CMP EDI,EBX
76A57C0F	<74 27	JE SHORT win32spl.76A57C38
76A57C11	FF76 04	PUSH DWORD PTR DS:[ESI+4]
76A57C14	FF75 08	PUSH DWORD PTR SS:[EBP+8]
76A57C17	68 EC59A576	PUSH win32spl.76A559EC UNICODE ""Zws\Zws"
76A57C1C	57	PUSH EDI
76A57C1D	FF15 BC11A576	CALL DWORD PTR DS:[<&USER32.wsprintfW>] USER32.wsprintfW
76A57C23	83C4 10	ADD ESP,10
76A57C26	57	PUSH EDI
76A57C27	E8 68A4FFFF	CALL <JMP.&SPOOLSS.AllocSplStr>
76A57C2C	8985 88FDFFFF	MOV DWORD PTR SS:[EBP-278],EAX
76A57C32	57	PUSH EDI
76A57C33	E8 B8A5FFFF	CALL <JMP.&SPOOLSS.DllFreeSplMem>
76A57C38	E8 1DA3FFFF	CALL win32spl.76A51F5A
76A57C3D	E8 7FA9FFFF	CALL win32spl.76A525C1
76A57C42	8BF0	MOV ESI,EAX
76A57C44	8975	
76A57C47	E8 1	
76A57C4C	3BF3	
76A57C4E	<0F84	
76A57C54	8B85	
76A57C5A	8946	
76A57C5D	895D	
76A57C68	58 4	
76A521EA	=<JMP.	

Registers (FPU)

EAX	42424242
ECX	42424242
EDX	00C59810
EBX	00000083
ESP	0081F034
EBP	0081F1CC
ESI	00C59810
EDI	00C50000
EIP	77FCC663 ntdll.77FC
C 0	ES 0023 32bit 0(F
P 1	CS 001B 32bit 0(F
A 0	SS 0023 32bit 0(F
Z 1	DS 0023 32bit 0(F
S 0	FS 0038 32bit 7FF
T 0	GS 0000 NULL

Log data

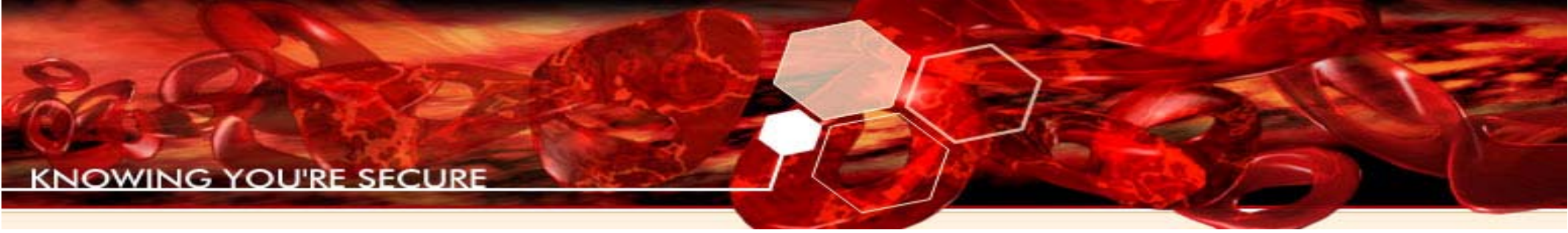
Address	Message
7C2D0000	Module C:\WINNT\system32\ADVAPI32.DLL
7C340000	Module C:\WINNT\system32\SECUR32.DLL
7C4E0000	Module C:\WINNT\system32\KERNEL32.dll
77FA144B	Attached process paused at ntdll.DbgBreakPoint
	Expression: ['EDI', '-', '8']
	Hooking on expression: '['EDI', '-', 8']
	Thread 00000434 terminated, exit code 0
	Thread 0000046C terminated, exit code 0
00C59600	> Hit Hook 0x76a57c1c, checking chunk: 0x00c59600
	=====
00C59600	0x00c59600> size: 0x00000210 (0042) preusize: 0x00000038 (0007)
00C59600	heap: *0x00000000* flags: 0x00000001 (B)
00C59810	0x00c59810> size: 0x000007f0 (00fe) preusize: 0x00000210 (0042)
00C59810	heap: *0x00000000* flags: 0x00000010 (F T)
00C59810	next: 0x00c57498 prev: 0x00c50178
77FCC663	Access violation when writing to [42424242]

# Inject Hook

- One of the biggest problems when hooking an allocation function is speed
- Allocations are so frequent in some processes that a hook ends up slowing down the process and as a result changing the natural heap behaviour (thus changing the layout)
  - lsass
  - iexplorer

# Inject Hooks into the target process speeds things up

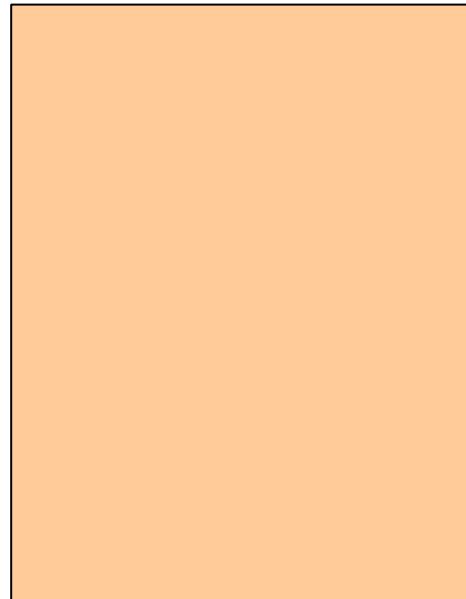
- This means doing function redirection and logging the result in the debugger itself (Avoiding breakpoints, event handling, etc)
- Can be done automatically via Immlib



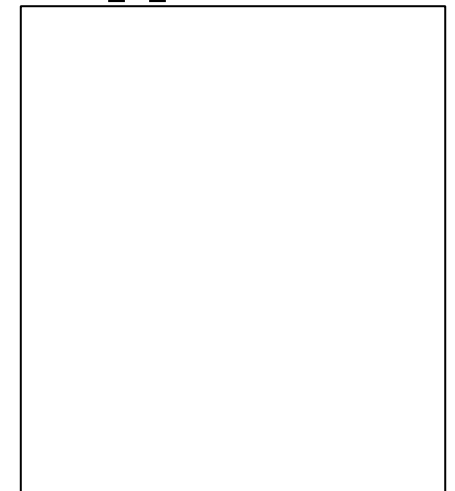
# Inject Hook

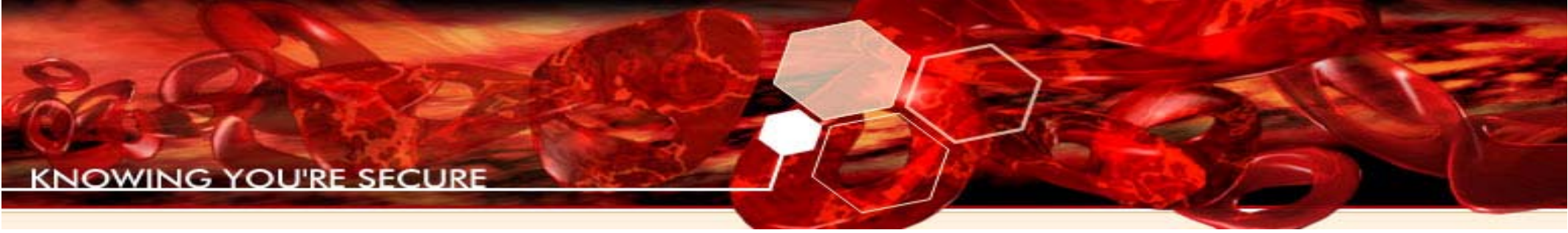
**VirtualAllocEx**

process



mapped mem

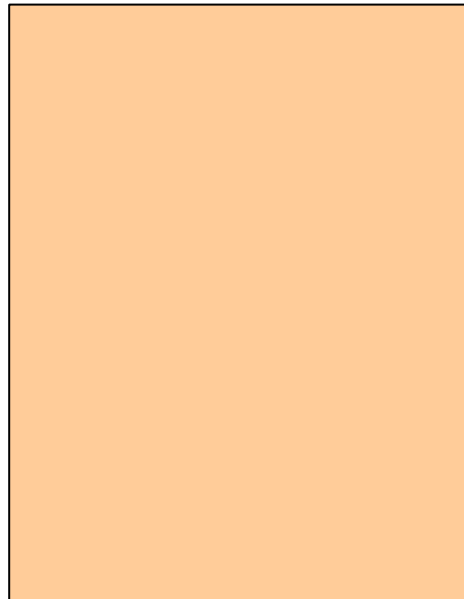




# Inject Hook

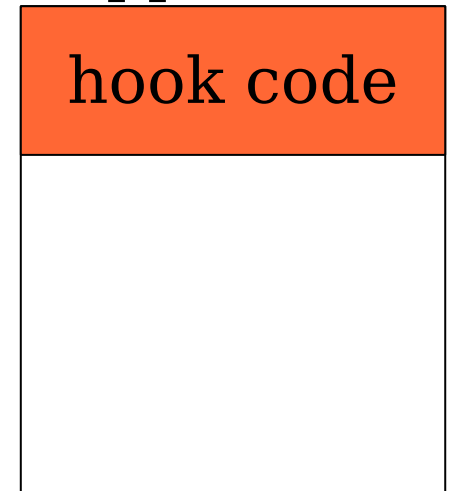
**InjectHooks**

process



mapped mem

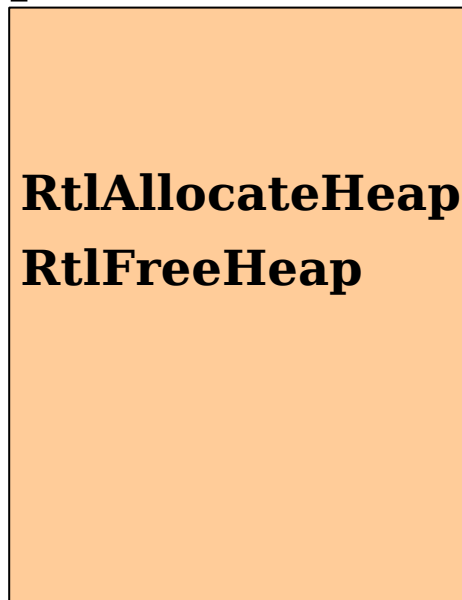
hook code



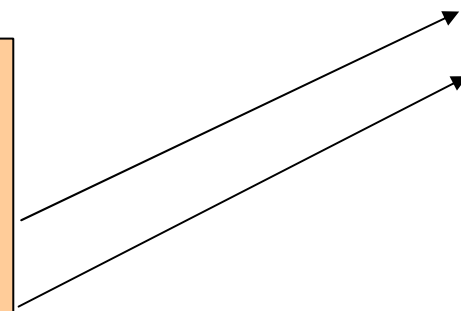
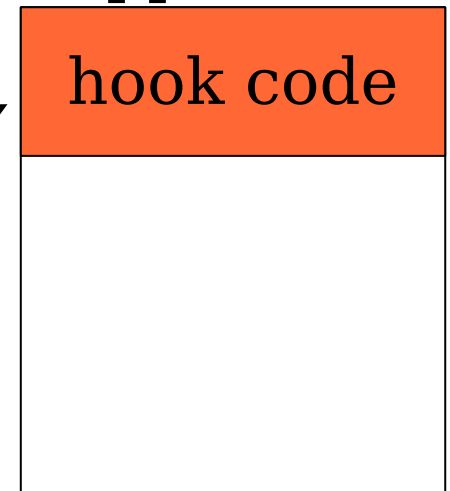
# Inject Hook

**Redirect  
Function**

process

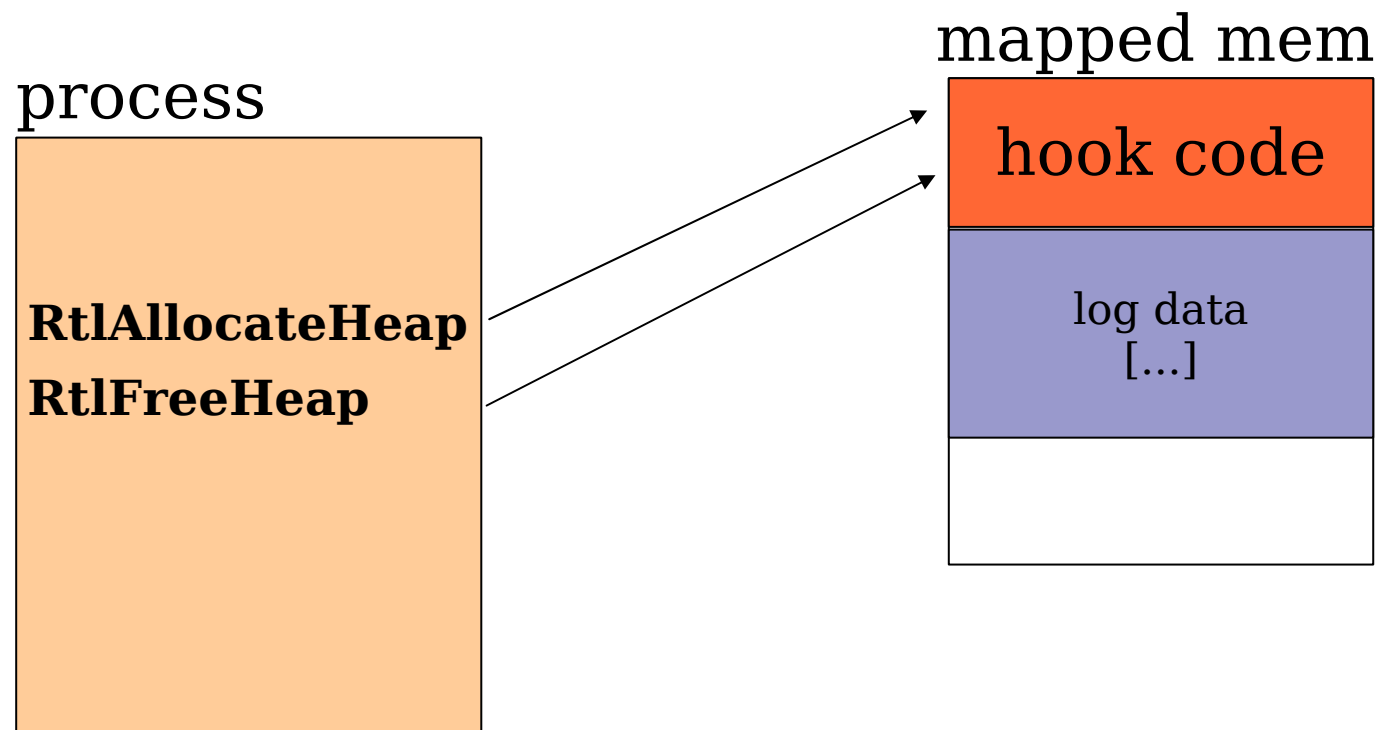


mapped mem

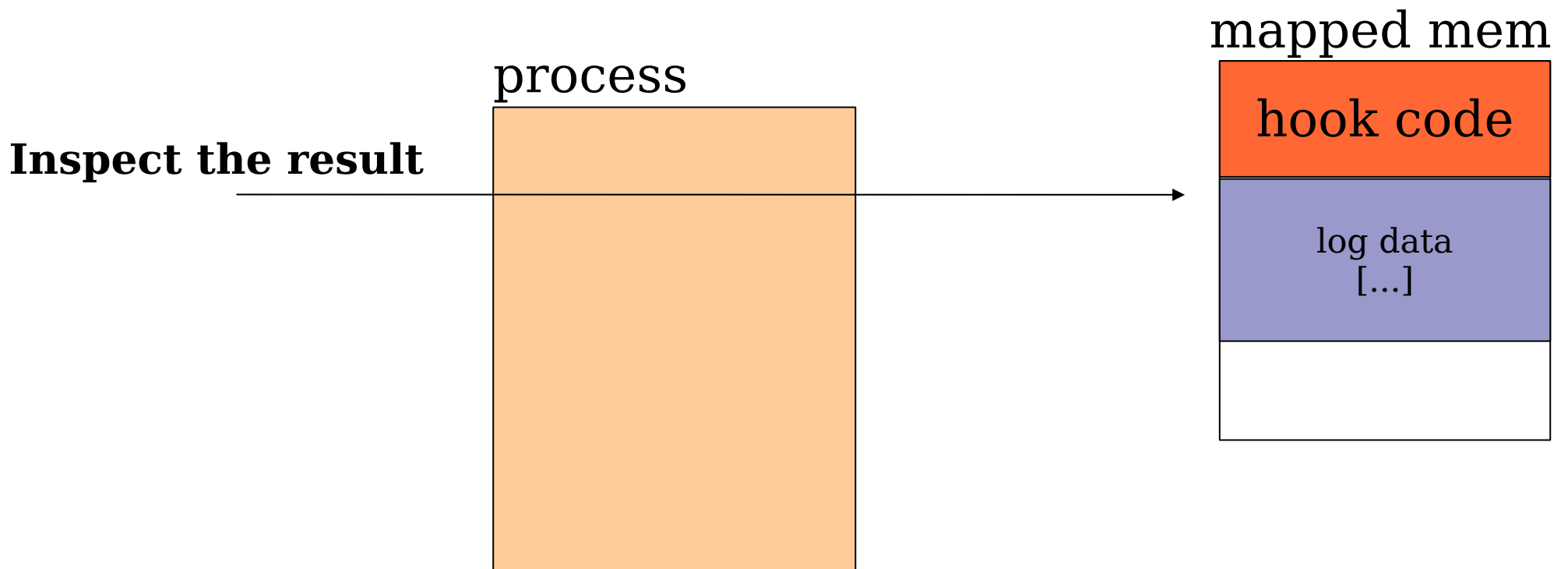


# Inject Hook

**Run the program**



# Inject Hook



# Inject Hook

- Hooking redirection:
  - !hippie -af -n tag\_name
- Hooking redirection as script:

```
fast = immllib.STDCALLFastLogHook( imm )  
fast.logFunction( rtlallocate, 3 )  
fast.logRegister( "EAX" )  
fast.logFunction( rtlfree, 3 )  
fast.Hook( )
```

# The future

- In the near future ID will have a heap simulator that, when fed with heap flow fingerprints, will tell you which function calls are needed to get the correct heap layout for your target process
- Simple modifications to existing scripts can put memory access breakpoints at the end of every chunk to find out exactly when a heap overflow happens
  - This is great for fuzzers

# Automating exploitation

- Stack overflows
  - Automation of simple exploitation techniques (bad bytes, etc) will be built into VisualSploit+ID
- Anti-DEP scripts already working!
- Deep protocol analysis and fuzzer integration on its way

# Conclusions

- Exploiting heap vulnerabilities has become much more costly
- Immunity Debugger offers tools to drastically reduce the effort needed to write reliable heap overflows
  - On older Windows platforms getting a reliable write4 the traditional way
  - On newer Windows platforms by abusing program-specific data structures

Thank you for your time

Contact us at:

[dave@immunityinc.com](mailto:dave@immunityinc.com)

**[nicolas.waisman@immunityinc.com](mailto:nicolas.waisman@immunityinc.com)**



Security Research Team