Understanding and bypassing
Windows Heap Protection

Nicolas Waisman
nicolas@immunityinc.com

Security Research
Who am I?

- Senior Security Researcher and Regional Manager at Immunity, Inc.
- Research and Development of reliable Heap Overflow exploitation for CANVAS attack framework
- Leading Immunity's latest project: the VulnDev oriented Immunity Debugger
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
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:

\[
\begin{align*}
\text{blink} &= \text{chunk}\rightarrow\text{blink} \\
\text{flink} &= \text{chunk}\rightarrow\text{flink}
\end{align*}
\]

\[
\text{if } \text{blink}\rightarrow\text{flink} == \text{flink}\rightarrow\text{blink} \\
\text{and } \text{blink}\rightarrow\text{flink} == \text{chunk}
\]
and harder...

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

```
- BL Chunk-  
  Flink 
  Blink

- FL Chunk-  
  Flink 
  Blink

- Chunk-  
  Flink 
  Blink
```

Chunk been unlinked
XP SP2 (and Vista) introduced more heap protections

- Low Fragmentation Heap Chunks:
  metadata semi-encryption

\[
\text{subsegment} = \text{chunk} \rightarrow \text{subsegment code} \\
\text{subsegment} \otimes \text{RtlpLFHKey} \\
\text{subsegment} \otimes \text{Heap} \\
\text{subsegment} \otimes \text{chunk} >> 3
\]
Vista heap algorithm changes make unlink() unlikely

- Vista Heap Chunks:
  metadata semi-encryption and integrity check

\[
\begin{align*}
*(\text{chunk}) & \equiv \text{HEAP}\rightarrow\text{EncodingKey} \\
\text{checksum} & = (\text{char}) * (\text{chunk} + 1) \\
\text{checksum} & \equiv (\text{char}) * (\text{chunk}) \\
\text{checksum} & \equiv (\text{char}) * (\text{chunk} + 2)
\end{align*}
\]

if \text{checksum} == \text{chunk} \rightarrow \text{Checksum}
Checksum makes it hard to predict and control the header

- Vista Heap Chunks:
  metadata semi-encryption and integrity check

```
0 1 2 3
 SIZE Fl Checks
```

Xor against
HEAP->EncodingKey
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

```
0 1 2 3 4 5
```

8 bytes

8 bytes

24 bytes

Note: 24 bytes is the total size. The actual data size is: 24 - 8 = 16 bytes
A quick look at the FreeList data structure

- FreeList

<table>
<thead>
<tr>
<th>0</th>
<th>BL</th>
<th>FL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BL</td>
<td>FL</td>
</tr>
<tr>
<td>2</td>
<td>BL</td>
<td>FL</td>
</tr>
<tr>
<td>3</td>
<td>BL</td>
<td>FL</td>
</tr>
<tr>
<td>4</td>
<td>BL</td>
<td>FL</td>
</tr>
<tr>
<td>5</td>
<td>BL</td>
<td>FL</td>
</tr>
<tr>
<td>n</td>
<td>BL</td>
<td>FL</td>
</tr>
</tbody>
</table>

Where \( n < 128 \)
Chunk coalescing: contiguous free chunks are joined to minimize fragmentation

\[
\text{PSize} = \ast (\text{ptr} + 2) \\
\text{Back_chunk} = \text{ptr} - (\text{PSize} \ast 8) \\
\text{if Back_chunk is not BUSY:} \\
\text{unlink(Back_chunk)}
\]
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 configuration

- 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

Vulnerable(function)

```c
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)
Memleaks leak memory that is never freed back to the allocator

- Memory stays allocated and busy until the process/service is restarted
  - Obviously this is the kind of memory leak most programmers are trained to find and remove from their programs

- Several bad coding practices lead to hard memleaks
  - Sometimes can be found via static analysis
Hard Memleaks come from many places

- 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: \texttt{X-LINK2STATE CHUNK=A} allocates \texttt{0x400} bytes.
    \texttt{X-LINK2STATE LAST} \texttt{CHUNK=A} free that chunk.
We correct our heap layout with memory leaks

• In summary, memleaks will help us do different things:
  – Filling the Lookaside
  – Filling 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 = entry 97
Heap Rule #1: Force and control the layout

- memleak(768)

Vulnerable(function)

```c
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");
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_analize_chunk
  !hippie                  !modptr
Immunity Debugger

- Dumping the Heap:
  - `!heap -h ADDRESS`

- Scripting example:
  ```
pheap = imm.getHeap( heap )
for chunk in pheap.chunks:
    chunk.printchunk()
  ```
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)
## Function Sniffing

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x77d4178c</td>
<td>Free (0x00070000, 0x00000000, 0x000a2808)</td>
</tr>
<tr>
<td>0x77d4178c</td>
<td>Free (0x00070000, 0x00000000, 0x000a6a30)</td>
</tr>
<tr>
<td>0x77d4178c</td>
<td>Free (0x00070000, 0x00000000, 0x000b7f58)</td>
</tr>
<tr>
<td>0x77f8f134</td>
<td>Free (0x00070000, 0x00000000, 0x000a6950)</td>
</tr>
<tr>
<td>0x77d3c2f7</td>
<td>Free (0x00070000, 0x00000000, 0x000d01f0)</td>
</tr>
<tr>
<td>0x77d3c2f7</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x77f8f134</td>
<td>Free (0x00070000, 0x00000000, 0x000a6978)</td>
</tr>
<tr>
<td>0x77d3c2f7</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x76a19c54</td>
<td>Free (0x005000, 0x00000000, 0x00c56f08)</td>
</tr>
<tr>
<td>0x78001532</td>
<td>Alloc (0x00230000, 0x00000000, 0x00000000) -&gt; 0x002373b8</td>
</tr>
<tr>
<td>0x77f8e6b9</td>
<td>Alloc (0x00070000, 0x00000000, 0x00000020) -&gt; 0x000a6978</td>
</tr>
<tr>
<td>0x77f8e6b9</td>
<td>Alloc (0x00070000, 0x00000000, 0x00000020) -&gt; 0x000a6950</td>
</tr>
<tr>
<td>0x77c58dc67</td>
<td>Alloc (0x00070000, 0x00100000, 0x00000000) -&gt; 0x000a6a30</td>
</tr>
<tr>
<td>0x76b0109</td>
<td>Free (0x00070000, 0x00000000, 0x000a6a30)</td>
</tr>
<tr>
<td>0x76b01c06</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x76b01c0b</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x76b01c10</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x76b01c15</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x76b01c1a</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x77f8f134</td>
<td>Free (0x00070000, 0x00000000, 0x000a6950)</td>
</tr>
<tr>
<td>0x77f8f134</td>
<td>Free (0x00070000, 0x00000000, 0x000a6978)</td>
</tr>
<tr>
<td>0x76b01bea</td>
<td>Free (0x00230000, 0x00000000, 0x002373b8)</td>
</tr>
<tr>
<td>0x76a94620</td>
<td>Free (0x00c50000, 0x00000000, 0x00c55f80)</td>
</tr>
<tr>
<td>0x76a94620</td>
<td>Free (0x00c50000, 0x00000000, 0x00c56fb0)</td>
</tr>
<tr>
<td>0x76a94620</td>
<td>Free (0x00c50000, 0x00000000, 0x00c56d90)</td>
</tr>
<tr>
<td>0x00000000</td>
<td>Chunk freed but not allocated on this heap flow</td>
</tr>
<tr>
<td>0x76b01c1a</td>
<td>Free (0x00070000, 0x00000000, 0x00000000)</td>
</tr>
<tr>
<td>0x00000000</td>
<td>Namelk detected</td>
</tr>
<tr>
<td>0x78001532</td>
<td>Alloc (0x00230000, 0x00000000, 0x00000110) -&gt; 0x00237440</td>
</tr>
<tr>
<td>0x00237438</td>
<td>Size: 0x00000118 (0023) prevsize: 0x00000088 (0011)</td>
</tr>
<tr>
<td></td>
<td>heap: <em>0x00000000</em> flags: 0x00000001 (0)</td>
</tr>
</tbody>
</table>
| 0x00237440  | String: ',NoCacheCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Automated data type discovery using Immlib

- As we now know overwriting the metadata of chunks to get a Write4 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!
<table>
<thead>
<tr>
<th>Address</th>
<th>Chunks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00c56fb8</td>
<td>heap: #0x00c50000* flags: 0x00000001 (B)</td>
</tr>
<tr>
<td>0x00c56fe4</td>
<td>Pointer: 0x00c550a8 in 0x00c50000!</td>
</tr>
<tr>
<td>0x00c56f44</td>
<td>Pointer: 0x00c57044 in 0x00c50000!</td>
</tr>
<tr>
<td>0x00c56f80</td>
<td>Pointer: 0x00c5718 in 0x00c50000!</td>
</tr>
</tbody>
</table>
| 0x00c56ff0 | Unicode: ',NoCacheCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Data Discovery can be scripted easily

```python
import libdatatypetype

dt = libdatatypetype.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 heaps 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

- !chunkanalizehook
- 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:
  
  !chunkanalizehook (-d) -a ADDRRES <exp>
  
  -a ADDRESS address of the hook
  
  -d find datatypes
  
  <exp> how to find the chunk
  
  ex: !chunkanalizehook -d -a 0x77fcb703 EBX - 8
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

RtlAllocateHeap
RtlFreeHeap

process

mapped mem

hook code
Inject Hook

Run the program

- RtlAllocateHeap
- RtlFreeHeap

mapped mem

hook code

log data

[...]
Inject Hook

Inspect the result

mapped mem

process

hook code

log data

[...]

KNOWING YOU'RE SECURE
Inject Hook

- Hooking redirection:
  - `!hippie -af -n tag_name`

- Hooking redirection as script:

  ```python
  fast = immlib.STDCALLFastLogHook( imm )
  fast.logFunction( rtlallocate, 3 )
  fast.logRegister( "EAX" )
  fast.logFunction( rtlfree, 3 )
  fast.Hook()
  ```
Finding Function Pointers

- If we achieve our write primitive by overwriting some structure in the data of the chunk (e.g. a doubly linked list, data pointers, etc.) we need to figure out what function pointers are triggered after our write primitive so we can target those function pointers.
Finding Function Pointer

- `!modptr <address>`
  - this tool will do data type recognition looking for all function pointers on a .data section, overwriting them and hooking on Access Violation waiting for one of them to trigger and logging it
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.
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 me at:

nicolas.waisman@immunityinc.com

Security Research Team