DISSECTING AN OPERATING SYSTEM VENDOR’S COMMITMENT TO HOST SECURITY

Windows Vista: Exploitation Countermeasures

Richard Johnson
richardj@microsoft.com

Toorcon 8
Sept 29, 2006
Memory corruption vulnerability exposure can be mitigated through memory hardening practices.

OS vendors have a unique opportunity to fight memory corruption vulnerabilities through hardening the memory manager.

Microsoft is raising the technology bar to combat external threats.
Microsoft is raising the technology bar to combat external threats

New features you’ve probably heard about
- Privilege Separation
- IE Protected Mode
- Kernel Patch Protection
- Code Integrity

New features we are covering today
- Address Space Layout Randomization
- Windows Vista Dynamic Memory Allocator
Red Hat Enterprise Linux

- Images
  - Section reordering
  - DLL randomization
  - EXE randomization*

- Stack
  - Protected control flow data*
  - Local variable protection*
  - Segment randomization
  - Non-executable

- Heap
  - Segment randomization
  - Non-executable
Comparing Exploitation Countermeasures

- **OpenBSD**
  - Images
    - DLL randomization
    - Section reordering
  - Stack
    - Protected control flow data*
    - Local variable protection
    - Segment randomization
    - Non-executable
  - Heap
    - Non-executable
    - Segment randomization

- **Apple OS X**
  - Images
    - No protection
  - Stack
    - No protection
  - Heap
    - No protection
Windows Vista

- Images
  - EXE randomization
  - DLL randomization

- Stack
  - Protected exception handlers
  - Protected control flow data
  - Local variable protection
  - Segment randomization
  - Non-executable

- Heap
  - Protected heap management data
  - Segment randomization
  - Non-executable
A quick look at what you’ve already been exposed to:

- Stack Cookies (/GS)
- Heap Mitigations (XP SP2)
- Structured Exception Handling (SafeSEH)
- Unhandled Exception Filter (MS06-051)
- Hardware DEP/NX
Windows Vista Exploitation Countermeasures

- New in Windows Vista
  - Address Space Layout Randomization
    - The History of ASLR
    - Architectural Considerations
    - Vista ASLR Technical Details
    - Testing Methodology
  - Dynamic Memory Allocator
    - A Short Lesson in Heap Exploitation
    - Improvements in Vista Heap Management
    - Vista Dynamic Memory Allocator Internals
    - Testing Methodology
Windows Vista ASLR is a technology that makes exploitation of a vulnerability a statistical problem.

Address Space Layout Randomization allows for the relocation of memory mappings, making the a process’ address space layout unpredictable.
ASLR Theory

- Exploitation relies on prior knowledge of the memory layout of the targeted process

Published Research

- PaX Documentation
  - PaX Team (http://pax.grsecurity.net/docs/aslr.txt)
- “On the Effectiveness of Address Space Layout Randomization”
  - Shacham, et al Stanford University
Windows Vista Process Model
- Most applications are threaded

Windows Vista Memory Management
- File mappings must align at 64k boundaries
- Shared mappings must be used to keep memory overhead low and preserve physical pages
- Fragmentation of the address space must be avoided to allow for large allocations
- Supports hardware NX
Image Mapping Randomization

- Random base address chosen for each image loaded once per boot
- 8 bits of entropy
- Fix-ups applied on page-in
- Images are mapped at the same location across processes
- 99.6% Effective
Heap Randomization
- Random offset chosen for segment allocation using 64k alignment (5-bit entropy)

Stack Randomization
- Random offset chosen for segment allocation using 64k or 256k alignment.
- Random offset within first half of the first page
Three pieces to the strategy

- Address Space Randomization
- Non-Executable Pages
- Service Restart Policy
Assumptions

- ASLR will only protect against remote exploitation
- ASLR requires NX to remain effective
- ASLR requires a limit on the number of exploitation attempts to remain effective
Prior to Windows Vista, NX could be disabled in a process if PERMANENT flag was not set

- Loading a DLL that is not NX compatible
  - No relocation information
  - Loaded off removable media
  - Open handle to a data mapping of the file

- Call NtSetInformationProcess with the MEM_EXECUTE_OPTION_ENABLE flag
In Windows Vista, NX cannot be disabled once turned on for a process.

Most processes enable NX by default.
Reducing the brute force space

- Code symmetry
  - Each location shifts stack pointer 20 bytes

```
kernel32+0xa1234:
  retn 16
  pop ebx
  pop ebp
  retn 8

kernel32+0xb1234:
user32+0x01234:
  jz 0x12345678
  sub esp, 16
  xor eax, eax
  ret

advapi32+0x51234:
  lea esp, [esp+20]
  pop eax
  call eax
```

- Advanced return address location
  - Emulation - EEREAP
Partial overwrites

- Given known addresses at known offsets, partial overwrites yield predictable results without full knowledge of the address space layout.

- With randomization in play, only bounded overflows can be used reliably for a single partial overwrite.
Partial overwrites

- A single partial overwrite can be used to execute a payload or gain additional overwrites.

D:\>partial
banner1: 0040100a banner2: 0040100f
hello world!

D:\>partial own
banner1: 0040100a banner2: 0040100f
owned!
Partial overwrites

A single partial overwrite can be used to execute a payload or gain additional partial overwrites.

```c
int main(int argc, char **argv)
{
    struct Object obj1;
    char buf[32];
    struct Object obj2;

    ... printf("banner1: %08x banner2: %08x\n", banner1, banner2);
    if(argv[1] != 0)
        strncpy(buf, overflow, sizeof(overflow));
    obj1.func();

    return 0;
}
```

```
004011ea 6a30            push    30h
004011ec 68b8114200      push    offset partial!overflow
004011f1 8d4dc4          lea     ecx,[ebp-3Ch]
004011f4 51              push    ecx
004011f5 e816060000      call    partial!strncpy (00401810)
004011fa 83c40c          add     esp,0Ch
```
Partial overwrites

- A single partial overwrite can be used to execute a payload or gain additional partial overwrites.

```
0:000> bp 004011f5
0:000> g
banner1: 0040100a banner2: 0040100f
Breakpoint 0 hit
partial!main+0x65:
004011f5 e816060000      call    partial!strncpy (00401810)
0:000> dt obj1
Local var @ 0x12ff38 Type Object
  +0x000 next             : (null)
  +0x004 val              : 17895697
  +0x008 func             : 0x0040100a     partial!ILT+5(_banner1)+0
0:000> p
partial!main+0x6a:
004011fa 83c40c          add     esp,0Ch
0:000> dt obj1
Local var @ 0x12ff38 Type Object
  +0x000 next             : 0x41414141 Object
  +0x004 val              : 1094795585
  +0x008 func             : 0x004040100f     partial!ILT+10(_banner2)+0
0:000> g
owned!
```
Information Leaking

- Uninitialized memory
- Use multiple attempts to gain address layout information that will get you code execution
- Additional image map locations can usually be inferred from one DLL address

Heap spraying reduces the need for accuracy
Non-randomized data as arguments to functions
  - SharedUserData / ReadOnlySharedMemoryBase
  - Non-relocatable resource dlls

3rd party binaries
Software Development Process
- Create NX and ASLR compatible binaries
- Keep service restart policies in mind
- Ensure information leak bugs are addressed

Technology
- Use hardware that supports NX
The majority of currently exploited vulnerabilities in Microsoft products are overflows into heap memory.

Heap exploitation relies on corrupting heap management data or attacking application data within the heap.
VTable Overwrites

Class objects contain a list of function pointers for each virtual function in the class called a vtable.

class MyClass {
public:
    MyClass();
    virtual ~MyClass();
    virtual MemberFunction();
    int MemberVariable;
};

Overwriting virtual function pointers is the simplest method of heap exploitation.
A Short Lesson in Heap Exploitation

HEAP_ENTRY Overflow

- Scenario: Heap-based buffer overflow allows for writing into adjacent free heap block

- Attack: Overwrite FLINK and BLINK values and wait for HeapAlloc()

```plaintext
mov dword ptr [ecx],eax
mov dword ptr [eax+4],ecx
EAX = Flink, EBX = Blink
```

- Allows one or two 4-byte writes to controlled locations
HEAP_ENTRY Overflow Mitigations in XP SP2

- List integrity checked during heap allocation
  - 8-bit Cookie
    - Verified on allocation after removal from free list

```
LIST_ENTRY->Flink->Blink == LIST_ENTRY->Blink->Flink == LIST_ENTRY
```
HEAP_ENTRY Overflow Mitigations in XP SP2

- Defeated by attacking the lookaside list
  - First heap overwrite takes control of Flink value in a free chunk with a lookaside list entry
  - Allocation of the corrupted chunk puts the corrupt Flink value into the lookaside list
  - Next HeapAlloc() of the same sized chunk will return the corrupted pointer
- Heap segment randomization
- HEAP_ENTRY integrity checks
- Block entry randomization
- Linked-list validation and substitution
- Function pointer hardening
- Terminate on Error
- **Heap Entry**
  - Checksum for Size and Flags
  - Size, Flags, Checksum, and Previous Size are XOR'd against random value

- Adds extra resilience against overflows into Flink and Blink values
Linked-lists

- Forward and backward pointer validation on unlink from any list

Lookaside lists

- Replaced by Low-Fragmentation Heap
Function pointer hardening
  ▪ CommitRoutine and InterceptRoutine function pointers encoded
  ▪ CRT atexit() destructors encoded

Terminate on Error
  ▪ Opt-in API that cannot be disabled
  ▪ Ensures program cleanup does not utilize tainted heap structures
The Low-Fragmentation Heap is enabled by default in Windows Vista.

The LFH replaces lookaside lists and is similar in nature:
- 128 buckets of static sized buffers
- Utilized for reoccurring allocations of the same size
HEAP_ENTRY

- Doubly-linked list pointers are only validated when unlinking a node

- **Attack**
  - If list head pointers can be corrupted prior to an insert, the destination of a 4-byte write can be controlled
  - The address of the free chunk being inserted into the list will be written to the corrupted linked list pointer

- **Assessment**
  - Writing the address of the chunk may be only be helpful in limited circumstances
  - It is difficult to find a list head to overwrite

```
InsertHeadList(ListHead, Entry)
Flink = ListHead->Flink;
Entry->Flink = Flink;
Entry->Blink = ListHead;
Flink->Blink = Entry;
ListHead->Flink = Entry;
```

```
InsertTailList(ListHead, Entry)
Blink = ListHead->Blink;
Entry->Flink = ListHead;
Entry->Blink = Blink;
Blink->Flink = Entry;
ListHead->Blink = Entry;
```
HEAP_UCR_DESCRIPTOR

- **Attack**
  - Repeated large allocations will result in the allocation of a new segment
  - HEAP_UCR_DESCRIPTOR is at a static offset from first allocation in a segment
  - If fake descriptor points at allocated memory, the next heap allocation will write a HEAP_UCR_DESCRIPTOR to a controlled address

- **Assessment**
  - Limited control of the data written should effectively reduce this to a partial DWORD overwrite
  - Increased complexity with multi-stage attack requires a high degree of control such as active scripting
**HEAP_UCR_DESCRIPTOR**

### Attack
- Repeated large allocations will result in the allocation of a new segment
- HEAP_UCR_DESCRIPTOR is at a static offset from first allocation in a segment
- If fake descriptor points at allocated memory, the next heap allocation will write a HEAP_UCR_DESCRIPTOR to a controlled address

### Assessment
- Limited control of the data written should effectively reduce this to a partial DWORD overwrite
- Increased complexity with multi-stage attack requires a high degree of control such as active scripting
**HEAP_UCR_DESCRIPTOR**

**Attack**
- Repeated large allocations will result in the allocation of a new segment
- HEAP_UCR_DESCRIPTOR is at a static offset from first allocation in a segment
- If fake descriptor points at allocated memory, the next heap allocation will write a HEAP_UCR_DESCRIPTOR to a controlled address

**Assessment**
- Limited control of the data written should effectively reduce this to a partial DWORD overwrite
- Increased complexity with multi-stage attack requires a high degree of control such as active scripting
**_LFH_BLOCK_ZONE_**

**Attack**
- New SubSegments are created at the location specified by the FreePointer in the LFH_BLOCK_ZONE structure.
- Control of the FreePointer allows writing a HEAP_SUBSEGMENT to an arbitrary location.
- Allocation size and number of allocations affect fields in the HEAP_SUBSEGMENT structure.

**Assessment**
- Limited control of the data written should effectively reduce this to a partial DWORD overwrite.
- Increased complexity attack requires a high degree of control such as active scripting.

---

<table>
<thead>
<tr>
<th><em>LFH_BLOCK_ZONE</em></th>
<th><em>HEAP_SUBSEGMENT</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000 ListEntry</td>
<td>+0x000 LocalInfo</td>
</tr>
<tr>
<td>+0x008 FreePointer</td>
<td>+0x004 UserBlocks</td>
</tr>
<tr>
<td>+0x00c Limit</td>
<td>+0x008 AggregateExchg</td>
</tr>
<tr>
<td>+0x010 BlockSize</td>
<td>+0x010 BlockCount</td>
</tr>
<tr>
<td>+0x012 Flags</td>
<td>+0x016 SizeIndex</td>
</tr>
<tr>
<td>+0x014 BlockCount</td>
<td>+0x017 AffinityIndex</td>
</tr>
<tr>
<td>+0x016 SizeIndex</td>
<td>+0x010 Alignment</td>
</tr>
<tr>
<td>+0x017 AffinityIndex</td>
<td>+0x018 SFreeListEntry</td>
</tr>
<tr>
<td>+0x01c Lock</td>
<td>+0x01c Lock</td>
</tr>
</tbody>
</table>
Default exploit mitigations on popular client operating systems

<table>
<thead>
<tr>
<th></th>
<th>Full Coverage</th>
<th>Partial Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Images</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section Reordering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE Randomization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLL Randomization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame Protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Var Protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Randomization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Executable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Metadata Protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Randomization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Executable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OS vendors have a unique opportunity to fight memory corruption vulnerabilities through hardening the memory manager.

Microsoft is committed to closing the gap as much as possible and Windows Vista will have the strongest pro-active vulnerability defense of any Windows release.

These protections will continue to evolve to prevent wide-spread exploitation of software vulnerabilities.

Exploitation mitigations do not solve the problem of software vulnerabilities, but do provide a stop-gap during times of exposure.
Thank you for attending

Please contact us at switech@microsoft.com for feedback on Microsoft’s mitigation technologies