The Evolution of Microsoft’s Exploit Mitigations

Past, Present, and Future

Tim Burrell
tim.burrell@microsoft.com
Microsoft Security Engineering Center (MSEC)
Security Science
Credits

- Peter Beck, Matt Miller (MSEC)
- Louis Lafreniere (Compiler team)

- Many others in these teams who helped along the way
Agenda

• Defining the purpose of exploit mitigations

• Microsoft’s exploit mitigation evolution
  – The past
  – The present/future
    • Windows7
    • Visual Studio 2010
The purpose of exploit mitigations

- Goal: decrease the probability of successful exploitation
  - Prevent the use of specific exploitation techniques
  - Reduce the reliability of exploitation techniques
- Generic protection for known & unknown vulnerabilities in all products, not just Microsoft products!
ACT I

THE PAST
Pre-XP SP2: The era of uninhibited worms

- Reliable exploitation techniques already existed
  - And they affected Windows, too!

- Exploits were developed, worms raged
  - Jul, 2000: IIS Code Red (MS01-033)
  - Jan, 2003: SQL Slammer (MS02-039)
  - Aug, 2003: Blaster (MS03-026)
  - May, 2004: Sasser (MS04-011)

- No platform exploit mitigations existed
  - Attack surface was very big
  - Exploitation techniques were uninhibited
Same techniques, different OS

- **Stack**: return address overwrite \([\text{Aleph96}]\)

- **Heap**: free chunk unlink \([\text{Solar00, Maxx01, Anon01}]\)
Visual Studio 2002

- GS v1 released

**Mitigation**

- **Behavior**
  - Compiler heuristics identify at-risk functions
  - Prologue inserts cookie into stack frame
  - Epilogue checks cookie & terminates on mismatch
GS v1 weaknesses

- **Adjacent local/parameter overwrite** [Ren02]
  ```c
  void vulnerable(char *in) {
    int unsafe = 0; char buf[256];
    strcpy(buf, in); ← overflow!
    if (unsafe != 0) DoSomethingUnsafe(); ← unsafe is corrupt
    return; ← GS cookie checked
  }
  ```

- **SEH overwrite bypass** [Litchfield03]

  ![Diagram of SEH chains]
  
  Normal SEH Chain:
  - N
  - H
  - app!_except_handler4
  - k32!_except_handler4
  - 0xffffffff

  Corrupt SEH Chain:
  - N
  - H
  - 0x7c1408ac
  - 0x414106eb [jmp +6]
  - pop eax
  - pop eax
  - ret

Visual Studio 2003

• GS v1.1 released with VS2003

• SafeSEH added, reliant on XP+ & recompile

Buffer overrun

Lower addresses

Higher addresses

Other local variables | Buffer local variables | GS Cookie | Saved EBP | Return address | Parameters

Safe SEH Handler

app!_except_handler4

Valid

SafeSEH Table

app!eh1
app!eh2
app!_except_handler4
...

Invalid SEH Handler

app!_main+0x1c

Not found in table

Mitigation

Visual Studio 2003

• GS v1.1 released with VS2003

• SafeSEH added, reliant on XP+ & recompile

Buffer overrun

Lower addresses

Higher addresses

Other local variables | Buffer local variables | GS Cookie | Saved EBP | Return address | Parameters

Safe SEH Handler

app!_except_handler4

Valid

SafeSEH Table

app!eh1
app!eh2
app!_except_handler4
...

Invalid SEH Handler

app!_main+0x1c

Not found in table

Mitigation
Limitations of SafeSEH
- Handler can be in an executable non-image region
- Handler can be inside a binary lacking SafeSEH
Windows XP SP2 arrives

- System binaries built with GS v1.1 & SafeSEH

- Data Execution Prevention (DEP)
  - Hardware-enforced non-executable pages
  - Software-enforced SEH handler validation
First round of heap mitigations

- Safe unlinking \((E \rightarrow B \rightarrow F == E \rightarrow F \rightarrow B == E)\)
- Heap header cookie validation

Limited randomization of PEB/TEB

- Reduces the reliability of certain techniques

Pointer encoding

- Protect UEF, VEH, and others via EncodeSystemPointer
**Same NX bypass, new OS**

- **Return to libc** \([\text{Solar97, Nergal01}]\)

<table>
<thead>
<tr>
<th>Stack Layout</th>
<th>Local Variables</th>
<th>Saved EBP</th>
<th>Return address</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploit Buffer</td>
<td>Padding</td>
<td>Address of system</td>
<td>Fake Return Address</td>
<td>Address of “cmd”</td>
</tr>
</tbody>
</table>

- **Many variations**
  - Return into VirtualProtect/VirtualAlloc
  - Disable DEP via ProcessExecuteFlags \([\text{Skape05}]\)
  - Create executable heap & migrate to it
  - Return-oriented programming \([\text{Shacham08}]\)
• Unsafe lookaside list allocations \cite{Anisimov04,Conover04-2}  
  – Overwrite free chunk on lookaside list & then cause allocation

• Unsafe unlinking of free chunks \cite{Conover04-2}  
  – Overwrite free chunk with specific Flink and Blink values

• Unsafe unlink via \texttt{RtlDeleteCriticalSection} \cite{Falliere05}  
  – Overwrite critical section structure on heap & delete it

• Exploiting \texttt{FreeList[0]} \cite{Moore05}  
  – Overwrite free chunk stored at \texttt{FreeList[0]} with specific data
• GS v2 released with VS2005
  – Shadow copy of parameters is made
  – Strict GS pragma

• C++ `operator::new` integer overflow detection [Howard07]
Windows Vista arrives

- **Address Space Layout Randomization (ASLR)** \[PaX02\]
  - Make the address space unpredictable

<table>
<thead>
<tr>
<th>Region</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>8 bits</td>
</tr>
<tr>
<td>Heap</td>
<td>5 bits</td>
</tr>
<tr>
<td>Stack</td>
<td>14 bits</td>
</tr>
</tbody>
</table>
Windows Vista arrives

• **Second round of heap mitigations** [Marinescu06]
  
  – Removal of lookaside lists and array lists
  
  – Block metadata encryption
  
  – Header cookie scope extended, validated in more places
  
  – Dynamic change of heap allocation algorithms (LFH)
  
  – Terminate on heap corruption (default for system apps)
  
  – \texttt{RtlDeleteCriticalSection} technique mitigated by \texttt{RtlSafeRemoveEntryList}
  
  – \texttt{FreeList[0]} technique mitigated by \texttt{RtlpFastRemoveFreeBlock}
Exploitation

Same ASLR evasions, new OS

- Partial address overwrite [Durden02]
- Address information disclosure [Soeder06]
- Reduced entropy on some platforms [Whitehouse07]
- Brute forcing [Nergal01, Durden02, Shacham04]
- Non-relocateable/predictable addresses [Sotirov08]
Newer heap techniques, partial & still less universal

- **HEAP structure overwrite** [Hawkes08]
  - Overwrite pointer in alloc’d chunk with heap base
  - Cause pointer to be freed & then re-allocated
  - Overwrite with specially crafted **HEAP** structure

- **LFH bucket/header overflow** [Hawkes08]

- Still need to evade DEP and ASLR if enabled
Windows Vista SP1 and Windows Server 2008 RTM

- **SEH Overwrite Protection (SEHOP)**
  - Dynamic SEH chain validation
  - GS+SEHOP = robust mitigation for most stack buffer overruns!

- **Kernel mode ASLR**
  - NT/HAL (5 bits of entropy)
  - Drivers (4 bits of entropy)

---

**Valid SEH Chain**

- N H → app\_except\_handler4
- N H → k32\_except\_handler4
- N H → ntdll\_FinalExceptionHandler

**Invalid SEH Chain**

- N H → app\_main+0x1c
- 0x41414141

Can’t reach validation frame!
Exploit Mitigations Timeline

- **Stack**
  - /GS 1.0
  - /GS 1.1
  - /GS 2.0
  - EH4
  - SEHOP

- **Heap**
  - Heap 1.0
  - Heap 2.0
  - HeapTerm

- **Code Execution**
  - DEP
  - /NXCOMPAT
  - ASLR
  - DEP + ATL

Timeline:
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
ACT II

THE PRESENT
Evolution of OS mitigations

• XPSP2
  – GS applies to both kernel and user mode
  – Heap mitigations are user mode only

• Vista: DEP + ASLR
  – Significantly increase difficulty of user mode exploitation

• Windows 7
  – Further improve kernel mode mitigations
Pool Overruns

• Very similar to Heap Overruns
• Allow arbitrary write what/where via unlink
• Occurs when
  – Merging adjacent free chunks
  – Removing chunk from *ListHead*

![Diagram of Pool Overrun](image-url)
Safe unlinking

• Checks integrity of \texttt{LIST\_ENTRY} structure
  \[(E\rightarrow B\rightarrow F == E\rightarrow F\rightarrow B == E)\]

• XPSP2 added this check in Heap

• Windows 7 RC has check in Kernel Pool
  – \textbf{Free} as well as Checked builds
Safe Unlinking - benefits

• Security
  – Mitigates arbitrary writes via unlink
  – Other exploit vectors far less generic

• Reliability
  – Detects corruption as early as possible
  – Bugchecks with unique code (0x19, 3, ...)

Mitigation
Safe Unlinking - costs

• Performance
  – Doesn’t hit performance measurably
  – A few extra instructions
  – No additional paging

• Compatibility
  – Pool corruption always bad, no exceptions
MS08-001 IGMP Pool Overrun

• Pool overrun in `tcpip.sys` [Kortchinsky08]

• Root cause is arithmetic overflow in buffer size calculation
  – One loop counts entries using a 16-bit counter
  – Counter wraps around past 65535
  – Memory is allocated based on counter
  – A different loop copies entries into buffer
UINT16 SourceCount = 0;
for (...) {
    if (...) SourceCount++;
}

RecordEntry = ExAllocatePoolWithTag(
    NonPagedPool,
    HeaderSize + (AddressBytes * SourceCount),
    IpGenericPoolTag);
SourceList = RecordEntry + HeaderSize;
for ( ... )
{
    if ( ... ){
        RtlCopyMemory(
            SourceList,
            ...,
            AddressBytes);
        SourceList += AddressBytes;
    }
}
Pool Mitigations

• Safe unlinking prevents all current variants of documented pool overrun exploits
• “Makes it immeasurably harder to exploit”
  – We’re not saying impossible
  – Also mitigates MS07-017, MS08-001, MS08-007
• Only safe unlinking right now
  – No pointer encoding, cookies etc
  – No protection of LookAside lists
Other enhancements

- Increased entropy for kernel mode ASLR
  - Drivers: 6 bits on x86, 8 bits on x64
ACT III

THE FUTURE
GS – effective or not?

• Vista
  – GS fundamentally the same
  – Many bypasses closed off via OS improvements
    • EH abuse
    • NX/DEP
    • ASLR

• Vista released worldwide 30\textsuperscript{th} January 2007

• MS07-017 security bulletin 10\textsuperscript{th} April 2007
  – Trivially exploitable stack overflow in ANI file parsing
The GS heuristic

- Not all functions GS-protected
  - Obvious and less obvious performance cost

- Insert cookie for
  - arrays of size > 4 with element size <= 2 (char/wchar)
  - Structures containing arrays with element size <= 2

- Originally designed to mitigate overflows arising from untrusted **string** data
The target of the overflow was a ANIHEADER structure on the stack:

typedef struct __ANIHEADER {
    DWORD cbSizeof;
    DWORD cFrames;
    DWORD cSteps;
    DWORD cx, cy;
    DWORD cBitCount, cPlanes;
    DWORD jifRate;
    DWORD fl; } ANIHEADER, *PANIHEADER;
MS07-017 – ANI stack overflow

• The ANIHEADER overflow equivalent to:

```c
ANIHEADER  myANIheader;
memcpy(&myANIheader,
       untrustedFileData->headerdata,
       untrustedFileData->headerlength);
```

• No character buffers on the stack
  ⇒ No GS protection
  ⇒ myANIheader is being treated like a character buffer
### Target buffer mitigated by GS?

<table>
<thead>
<tr>
<th>Security bulletin</th>
<th>GS?</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS03-026 (Blaster)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MS06-040</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MS07-029</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MS04-035 (Exchange)</td>
<td>No</td>
<td>DWORD array</td>
</tr>
<tr>
<td>MS06-054 (.PUB)</td>
<td>No</td>
<td>structure populated from file</td>
</tr>
<tr>
<td>MS07-017 (.ANI)</td>
<td>No</td>
<td>structure populated from file</td>
</tr>
</tbody>
</table>
Vista SP1

- In development at time of ANI vulnerability
- `#pragma strict_gs_check`?
- More aggressive GS heuristic
- Much more aggressive GS heuristic
- Any address-taken local variable is considered a potential target!
<table>
<thead>
<tr>
<th>Security bulletin</th>
<th>Legacy GS</th>
<th>Strict GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS03-026 (Blaster)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS06-040</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS07-029</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS04-035 (Exchange)</td>
<td>No</td>
<td>DWORD array</td>
</tr>
<tr>
<td>MS06-054 (.PUB)</td>
<td>No</td>
<td>Data structure</td>
</tr>
<tr>
<td>MS07-017 (.ANI)</td>
<td>No</td>
<td>Data structure</td>
</tr>
</tbody>
</table>
#pragma strict_gs_check(on)

```c
void main()
{
    int i;

    printf("%d", (int) &i); // address-taken
}
```
strict GS

• Applied in a very targeted way for Vista SP1

<table>
<thead>
<tr>
<th>Binary</th>
<th>Functions in DLL</th>
<th>OS</th>
<th>Number of cookies</th>
<th>% protected functions</th>
<th>Factor increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>qasf.dll</td>
<td>1526</td>
<td>Vista RTM (GS)</td>
<td>58</td>
<td>3.80%</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vista SP1 (strict GS)</td>
<td>202</td>
<td>13%</td>
<td></td>
</tr>
<tr>
<td>avifil32.dll</td>
<td>494</td>
<td>Vista RTM (GS)</td>
<td>40</td>
<td>8.10%</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vista SP1 (strict GS)</td>
<td>134</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>WMASF.dll</td>
<td>1484</td>
<td>Vista RTM (GS)</td>
<td>40</td>
<td>2.70%</td>
<td>13.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vista SP1 (strict GS)</td>
<td>524</td>
<td>35%</td>
<td></td>
</tr>
</tbody>
</table>

• But not suitable for system-wide deployment
  \[\Rightarrow GS++\]
Issues of scale

Vista SP1 approach was targeted

Can we make the default /GS better?
Enhancing GS

• Increased coverage
  – Protect more stuff

• Smarter coverage
  – Don’t protect where it’s unnecessary

• Different models for how this might work
GS++ heuristic?

- All arrays?
- All structures?

What subset is most likely to contain untrusted data?

Performance concerns!
GS++ heuristic

Arrays where element type not of pointer type:
- char myBuf[]
- DWORD myBuf[]
- HANDLE myBuf[]

and size of array is >2 elements
GS++ heuristic

- Structures:
  - Containing an array where element type is not of pointer type.

- Made up of pure data:
  - No members of pointer type
  - >8 bytes in size
  - Default constructor/destructor

```c
struct _ANIHEADER{
    DWORD cbSizeof;
    DWORD cFrames;
    DWORD cSteps;
    DWORD cx, cy;
    DWORD cBitCount
    DWORD cPlanes;
    DWORD jifRate;
    DWORD fl; }
```
Impact on cookie count

GS-protected functions in sample code

<table>
<thead>
<tr>
<th></th>
<th>Original GS</th>
<th>VS2010 GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>User/client</td>
<td>9608</td>
<td>12846</td>
</tr>
<tr>
<td>Kernel</td>
<td>2361</td>
<td>4686</td>
</tr>
<tr>
<td>User/client (% total fns)</td>
<td>6.0%</td>
<td>8.0%</td>
</tr>
<tr>
<td>Kernel mode (% total fns)</td>
<td>5.2%</td>
<td>10.4%</td>
</tr>
</tbody>
</table>

⇒Cookie increase between 2% and 5%
GS optimization

- No GS cookies when usage is provably safe

```c
STDAPIL ConsumeData(BYTE *pbData)
{
    BYTE Temp[MAX];

    if (pbData)
    {
        ...
        memcpy(Temp, pbData, ARRAYSIZE(Temp));
        ...
    }
}```
GS optimization

- No GS cookies when usage is provably safe

```c
STDAPI FillBuffer(wchar_t *pBuf, int count)
{
    ...
    memcpy(pBuf, GetData(), count*sizeof(wchar_t));
    ...
}

STDAPI ParseData()
{
    wchar_t buffer[BUF_SIZE];
    FillBuffer(buffer, _countof(buffer));
    ...
}
```
GS enhancements [VS2010]

- GS heuristic
  - Identify more potential hazards

- GS optimization
  - Some potential hazards turn out to be safe

Increased scope of heuristic:

- MS07-017 LoadAniIcon function
  - ANI data structure

- Integer array
  - String buffer

- Integer
  - GS-protected
## Impact on cookie count

<table>
<thead>
<tr>
<th></th>
<th>Original GS</th>
<th>VS2010 GS</th>
<th>VS2010 GS [with GS opt]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User/client</strong></td>
<td>9608</td>
<td>12846</td>
<td>11654</td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td>2361</td>
<td>4686</td>
<td>3909</td>
</tr>
<tr>
<td><strong>User/client (% total fns)</strong></td>
<td>6.0%</td>
<td>8.0%</td>
<td>7.3%</td>
</tr>
<tr>
<td><strong>Kernel mode (% total fns)</strong></td>
<td>5.2%</td>
<td>10.4%</td>
<td>8.7%</td>
</tr>
</tbody>
</table>
## Impact on stack overflow security bulletins

<table>
<thead>
<tr>
<th>Security bulletin</th>
<th>Original GS</th>
<th>VS2010 GS</th>
<th>Strict GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS03-026 (Blaster)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS06-040</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS07-029</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS04-035 (Exchange)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS06-054 (.PUB)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS07-017 (.ANI)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
... but GS not a panacea

<table>
<thead>
<tr>
<th>Security bulletin</th>
<th>Original GS</th>
<th>VS2010 GS</th>
<th>Strict GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS03-026 (Blaster)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS06-040</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS07-029</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS04-035 (Exchange)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS06-054 (.PUB)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS07-017 (.ANI)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MS08-072</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MS08-067</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Still need to write secure code!

- Even the new heuristic will not cover all cases
- GS does not apply to some types of stack-based attacks (for example underflow).
Enhanced GS

• In Visual Studio 2010
  – Same /GS switch
  – Enhanced GS++ heuristic
  – GS optimization
Conclusion

• Modern exploitation is difficult & not universal
  – Techniques are tied to specific vulnerability scenarios

• Gaps do exist that can make exploitation easier
  – But these are the exception, not the rule

• We are committed to protecting our customers
  – Continued improvement of our mitigation technology
  – Providing actionable exploitability data with bulletins
Questions?

Thank you!

• Security Science at Microsoft

• Security Research & Defense blog
References

References

[Korthcinsky08] Kostya Kortchinsky. Real World Kernel Pool Exploitation, SyScan 08 Hong Kong