GS and ASLR in Windows Vista™

Ollie Whitehouse

Agenda

symantec.

1 Introduction to GS / Detecting GS

2 GS Analysis Findings

3 Introduction to ASLR

4 ASLR Analysis Findings

5 Conclusions

GS and ASLR in Windows Vista™

- Research conducted by Symantec in 2006
  - Part of our larger research project into Windows Vista™
- GS research goals:
  - Understand the implementation of GS
  - Develop means to be able to identify GS and non-GS binaries
  - Understand which binaries in Windows Vista™ are not GS protected
  - Understand any impact ASLR has on GS cookies
- ASLR research goals:
  - Assess the implementation

Confidence in a connected world.

Introduction to GS

Introduction to GS

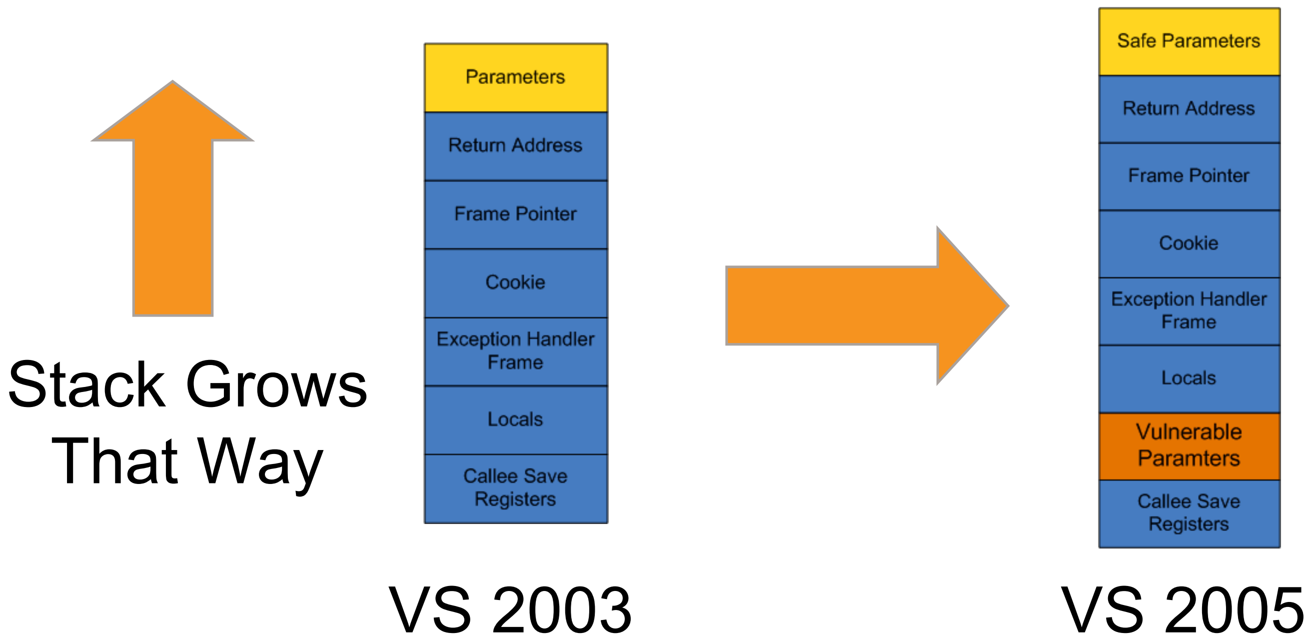
- Stack overflow mitigation
  - Uses cookies placed on the stack
  - These are verified on function return
  - If the cookie is incorrect a stack overflow is assumed
  - The program is shut down
- About the GS Cookie
  - The unique is a random 32bit value
  - A master copy is located in memory
  - With ASLR this becomes random

Introduction to GS

- Implemented via function prologs and epilogs
  - Added at compile time to appropriate functions
  - Prolog pushes the cookie on to the stack on function entry
  - Epilog checks the cookie before function return
- 3rd generation GS in Visual Studio 2005
  - First introduced in Visual Studio 2002
  - We will only be covering Visual Studio 2003’s and 2005’s implementations

Introduction to GS

- GS has improved with Visual Studio 2005
  - 2003 didn't protect vulnerable parameters
- Result of these improvements – new stack layout



Introduction to GS

- GS won’t always be applied however!
  - I refer to these as ‘The GS Rules’
- The Rules Are:
  - Functions that do not contain a stack buffer.
  - If optimizations (/O Options (Optimize Code)) are not enabled.
  - Functions with a variable argument list (...).
  - Functions marked with naked (C++).
  - Functions containing inline assembly code in the first statement.
  - If a parameter is used only in ways that are less likely to be exploitable in the event of a buffer overrun.

Confidence in a connected world.

Detecting GS

Detecting GS Binaries

- My original goals
  - To be able to say if a binary is or is not GS compiled
  - To be able to do this without symbols
- What I found
  - Depending on the version of Visual Studio (2003 versus 2005) slightly different approaches were needed
  - Technique similar to FLIRT signatures used (conceived by Ilfak of Data Rescue)
  - This resulted in accurate results on if a binary contained GS code
  - But also presented problems when dealing with statically linked code or ‘The GS Rules’
  - …. But we’ll get to that in a bit

Quick Introduction to FLIRT

- Originally conceived by Ilfak Guilfanov of Data Rescue
  - http://www.datarescue.com/idabase/flirt.htm
- Simple idea – great results
  - Take a disassembly (bigger the better)
  - Understand how this can be optimized
  - Now for each potential implementation of the disassembly remove the variable portions
  - For optimal speed create if/else branches so your code becomes unreadable
  - Scan binaries for these signatures without the need to disassemble

Introduction to FLIRT

- The Original Disassembly

```
3B0DCC012309        cmp     ecx,[L092301CC]
7509                jnz     L09204E27
F7C10000FFFF        test    ecx,FFFF0000h
7501                jnz     L09204E27
C3                  retn
```

- Now Remove the Variable Portions

```
3B0DCC012309        cmp     ecx,[L092301CC]
7509                jnz     L09204E27
F7C10000FFFF        test    ecx,FFFF0000h
7501                jnz     L09204E27
C3                  retn
```

- Leaves Us With A Signature of

```
3B 0D [skip 4] 75 [skip 1] F7 C1 [skip 4] 75 [skip 1] C3
```

Detecting GS Binaries (VS2003)

- How do we detect GS compiled VS 2003 binaries?
- Check for __security_error_handler wrapper function

```
6A08            push    00000008h
68C8243021      push    L213024C8
E882020000      call    SUB_L21316B44
8365FC00        and     dword ptr [ebp-04h],00000000h
6A00            push    00000000h
6A01            push    00000001h
E86D020000      call    jmp_MSVCR71.dll!....
59              pop     ecx
59              pop     ecx
EB07            jmp     L213168DA
L213168D3:
33C0            xor     eax,eax
40              inc     eax
C3              retn
```

Detecting GS Binaries (VS 2003)

- How does the wrapper function get called?
- Back one step (indirect jump)
  - `L213168F0:`
  - `E9C1FFFFFF          jmp      L213168B6`
- Back two steps (cookie compare)
  - `SUB_L213168E7:`
  - `3B0DA8943121        cmp      ecx,[L213194A8]`
  - `7501                jnz      L213168F0`
  - `C3                  retn`
- So
  - Epilog -> Compare cookie -> Indirect jump -> Calling wrapper

Detecting GS Binaries (VS2003)

- Signature used

```
6A08            push    00000008h
68C8243021      push    L213024C8
E882020000      call    SUB_L21316B44
8365FC00        and     dword ptr [ebp-04h],00000000h
6A00            push    00000000h
6A01            push    00000001h
E86D020000      call    jmp_MSVCR71.dll!....
59              pop     ecx
59              pop     ecx
EB07            jmp     L213168DA
L213168D3:
33C0            xor     eax,eax
40              inc     eax
C3              retn
```

Detecting GS Binaries (VS 2003)

- Results
  - Able to identify VS 2003 GS compiled binaries
  - BUT not able to identify at function level
  - This will potentially miss binaries which are statically linked with GS code
  - However I never found any examples

Example Detecting VS2003

- Example

```
D:\Code\C\GSAudit\Debug>GSAudit.exe | findstr 2003
[*] C:\Windows\System32\\AAAAAA.exe is /GS compiled (2003)
[*] C:\Windows\System32\\atl71.dll is /GS compiled (2003)
[*] C:\Windows\System32\\ceutil.dll is /GS compiled (2003)
[*] C:\Windows\System32\\cttune.cpl is /GS compiled (2003)
[*] C:\Windows\System32\\DEVMAN.DLL is /GS compiled (2003)
[*] C:\Windows\System32\\dllcache\netfxocm.dll is /GS compiled (2003)
```

Detecting GS Binaries (VS 2005)

- VS 2005 - harder to detect (if done properly)
  - As statically linked libraries may be GS compiled
  - BUT the main application may not be
  - Same is true for VS 2003 but less common
  - So simply checking for a 'signature' can yield false positives
- VS 2005 is the primary compiler for Windows Vista™
  - So had to solve this problem
  - Couple of approaches taken
- I also wanted to understand
  - Functions which fell under 'The GS Rules'

Detecting GS Binaries (VS 2005)

- We FLIRT signature __security_check_cookie
- We find the compare in __security_check_cookie

```
3B0DCC012309      cmp         ecx,[L092301CC]
```

- This allows us to locate __security_cookie
  - We then scan for every function which does
    - `MOV EAX,__security_cookie`
  - This is used to locate every GS protected function
- This then allows us to say
  - foo.exe has (x) functions which call __security_check_cookie

Example Detecting (VS2005)

- Example using VS2005 analyze option

```
D:\Code\C\GSAudit\Debug>GSAudit.exe -a
[i] /GS Audit - Ollie Whitehouse
[i] use '-h' for help!

[i] Analyze Mode: On
[*] C:\Windows\System32\\Audiodev.dll has /GS __security_check_cookie present (2
005) - type 2
[i] Number of MOV EAX,__security_cookie 101 - File size 480768 (bytes)
[*] C:\Windows\System32\\blackbox.dll has /GS __security_check_cookie present (2
005) - type 3
[i] Number of MOV EAX,__security_cookie 69 - File size 233472 (bytes)
[*] C:\Windows\System32\\cdm.dll has /GS __security_check_cookie present (2005)
- type 2
[i] Number of MOV EAX,__security_cookie 24 - File size 75544 (bytes)
[*] C:\Windows\System32\\CEWMDM.dll has /GS __security_check_cookie present (200
5) - type 2
[i] Number of MOV EAX,__security_cookie 54 - File size 226816 (bytes)
```

Detecting GS Binaries (VS 2005)

- BUT we wanted to be able to say
  - foo.exe has (n) functions of which (x) are GS protected which is (y)%
- Solution
  - IDAPython (caveat++) to export the total number of functions for each binary!
  - Allowed me to correlate total number of functions versus total GS protected functions

Detecting GS Binaries (VS 2005)

- Why this approach?
  - It was the quickest to develop initially
  - Shows me binaries with lots of functions and low number of GS checks
  - This allows me to prioritise manual analysis

Detecting GS Binaries (VS 2005)

- Is there a better approach?
  - Yes (in some respects)
- Did this achieved my original goals?
  - I can tell if NO GS code is present
  - But I can't tell if 'The GS Rules' are in play
  - I also can't tell if there are other unprotected stack buffers
  - So… Sort Of…

GS Analysis Findings – Next Problem

- So a new problem
  - Need to be able to see for every function if
    - A) It has local stack variables over four bytes
    - B) Is or is NOT GS protected
  - This will allow us to categorically say
    - Is the application GS compiled
    - OR is it linked with GS code
    - If it is GS compiled
    - ARE there any functions which fall under the GS rules

GS Analysis Findings – Next Problem

- Solution
  - IDA based (.idc)
    - Could use Phoneix from Microsoft (only non commercial though)
  - Current implementation only works with Symbols
  - Can be combined with FLIRT signatures from GSAudit
  - Scans every function
  - Works out size of local stack buffers (using Halvars BugScam code) – i.e. is it > 4 bytes
  - Checks to see if function is GS protected
  - Flags if local stack variable size > 4 and NOT GS protected
- Perfect?
  - Alas not, but proof of concept does work…

Confidence in a connected world.

GS Analysis Findings

GS Analysis Findings

- Windows Vista™ RTM 32 bit – `C:\Windows`
  - ~150 binaries had NO GS code present
  - That is to say they where either not GS compiled
  - OR did not have local stack buffers which required GS protection
- Caveats
  - I explicitly added checks for drivers (`GSDriverEntry()`)
  - Not all these binaries will be authored by Microsoft – i.e. 3rd parties
  - Others will be legacy binaries (Microsoft indicated some were from NT4)

GS Analysis Findings

- Using the statistical approach
  - Binaries with a large number of total functions BUT low number of GS checks were flagged
    - 1000 functions / 30 checks
    - 38,871 functions / 1,568 checks
    - 8,250 functions / 2 checks
    - 294 functions / 4 checks
    - 166 functions / 3 checks
  - These five were manually investigated
  - Showed there was no statistical link between total functions and GS checks
  - This was expected - all were GS compiled

GS – Other Observations

- There is a bug in Image randomization (we'll discuss this in more detail later)
  - Which impacts where the GS master cookie is stored
  - David Litchfield of NGS talked about attacking the master cookie in previous versions of Visual Studio with an arbitrary 4 byte overwrite
  - BUT although we know where the GS master cookie will be 25% of the time
  - It doesn't currently yield us anything
  - As Microsoft now XOR the GS master cookie with EBP when placing it on the stack
  - EBP is subject to ASLR ;-((potentially – if not overwrite SEH)

Oh! A Quick Note

- Compile this code and GS protect it

```
#include "stdafx.h"
void vulnerable(char *input){
   char foo[4];
   strcpy(foo,input);
}
int _tmain(int argc, _TCHAR* argv[])
{
   vulnerable(argv[1]);
   return 0;
}
```

- Result – not GS protected (due to stack buffer <= 4)

Confidence in a connected world.

Introduction to ASLR

Introduction to ASLR

- Conceived as part of the PaX project
- Entropy to where the stack, heap and code sections exist
- Makes exploitation of vulnerabilities using fixed offsets harder
- Previously only available via third party solutions on Windows, with Windows Vista™ now native support
- Applications need to be linked with Visual Studio 2005 SP1 and the `/dynamicbase` flag
- Affects not only the main program binary but DLL's as well (if they are ASLR enabled)
- Legacy applications will require recompilation

Introduction to ASLR

Section	Bits of Entropy	Expected Locations	Observed Locations
Heap – HeapAlloc	5+	32+	
Heap – Malloc	5+	32+	
Heap – CreateHeap / HeapAlloc	5+	32+	
Stack	14	16,384	
Image (Code)	8	256	
PEB	4	16	

Introduction to ASLR

- Microsoft kind enough to provide basic heuristics
- Heap
  - Request an allocation of size (rand(0..31) * 64kb) then free the extra memory.
- Stack:
  - 1. Skip rand(0..31) STACK_SIZE (typically 64kb or 256kb) spaces, then allocate stack
  - 2. Skip rand(0..PAGE_SIZE/2) (rounded to PTR alignment: 4b (x86), 8b (x64) or 16b (IA64)) bytes from top of stack
- Image:
  - Heuristic: Offset the starting address for the first image (NTDLL.DLL) by (rand(0..255) * 64kb) and then pack all images after that

ASLR Analysis Findings

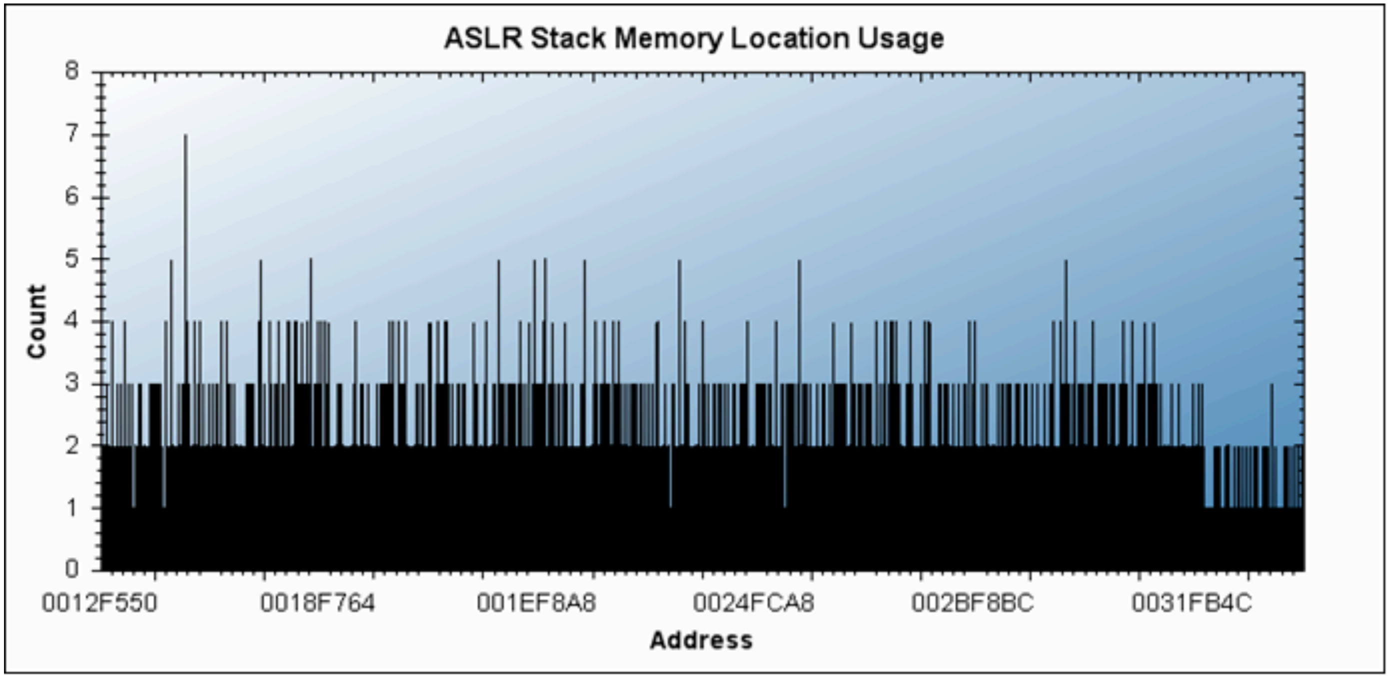
ASLR Findings

- Based on a run of 11,500 executions
- The 32bit RTM release was used on an AMD3200 CPU
- Rebooted between each run
- This was to ensure:
  - A) The entropy was reset
  - B) So I could measure image randomization
- Results have been confirmed by Microsoft

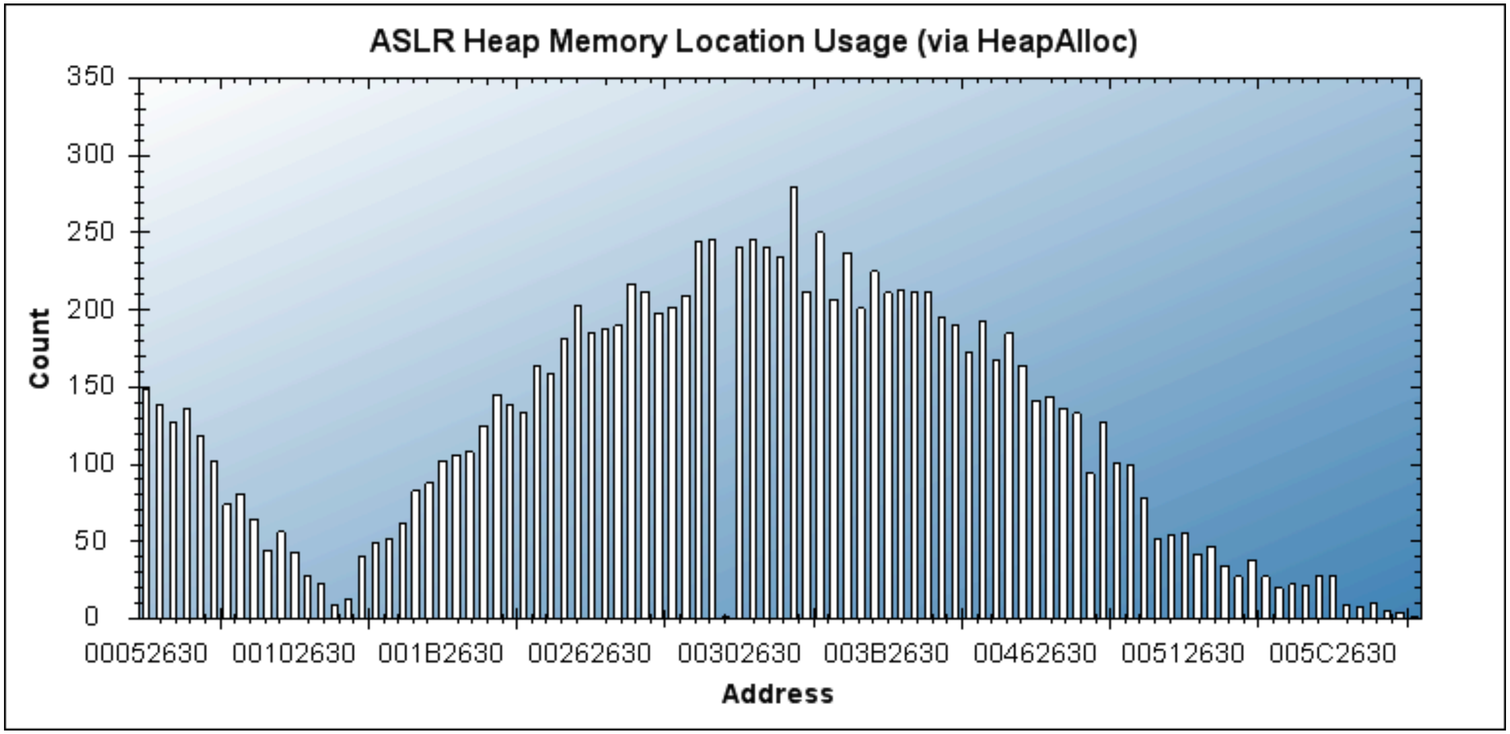
Introduction to ASLR

Section	Bits of Entropy	Expected Locations	Observed Locations
Heap – HeapAlloc	5+	32+	95
Heap – Malloc	5+	32+	192
Heap – CreateHeap / HeapAlloc	5+	32+	209
Stack	14	16,384	8,568
Image (Code)	8	256	255
PEB	4	16	16

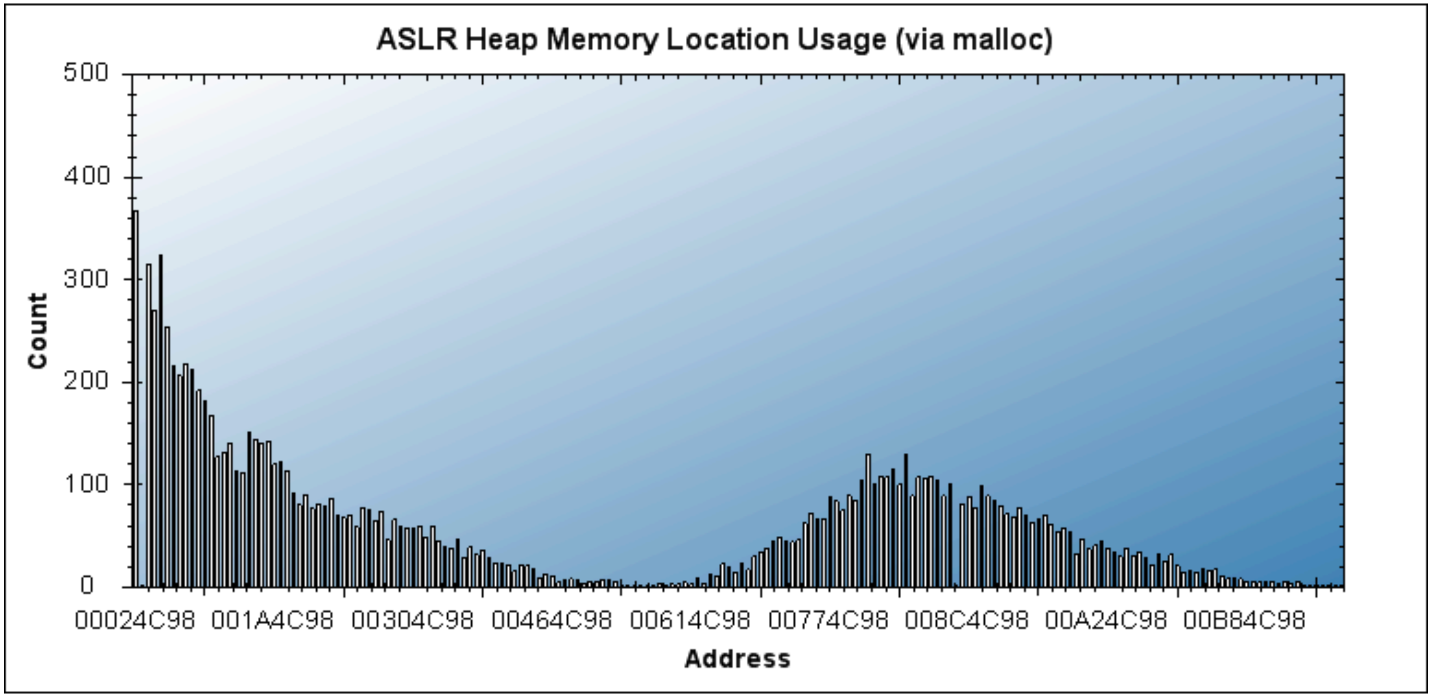
Stack – Near Uniform Distribution



Heap – via HeapAlloc()



Heap – via malloc()



Heap – via CreateHeap() / HeapAlloc()

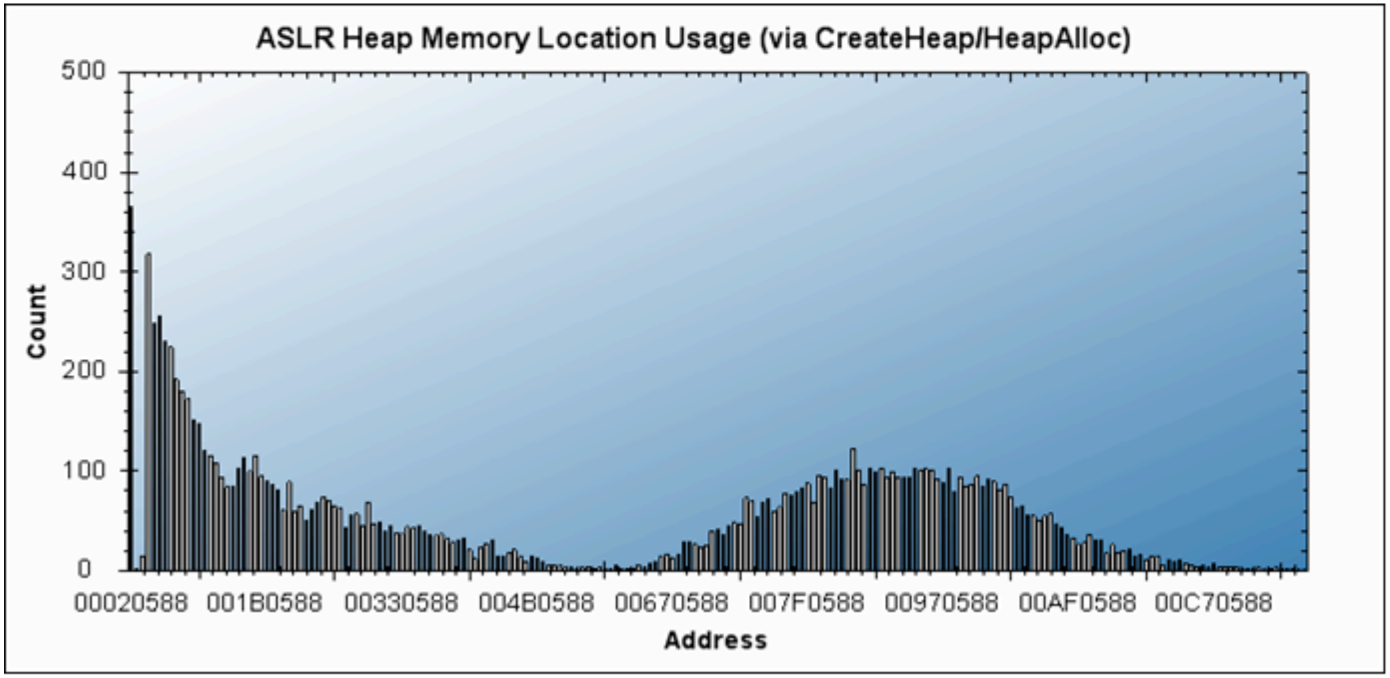
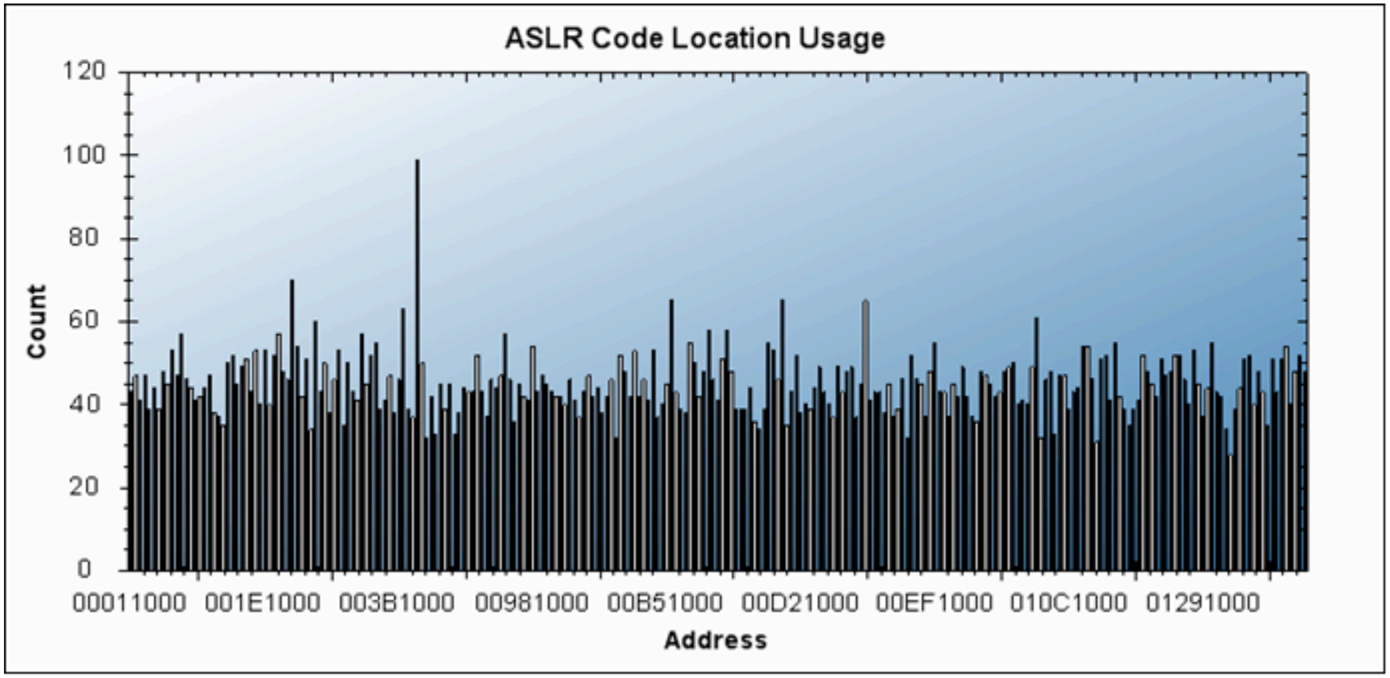


Image – I Spy a Spike!



PEB – I Spy Two Spikes!

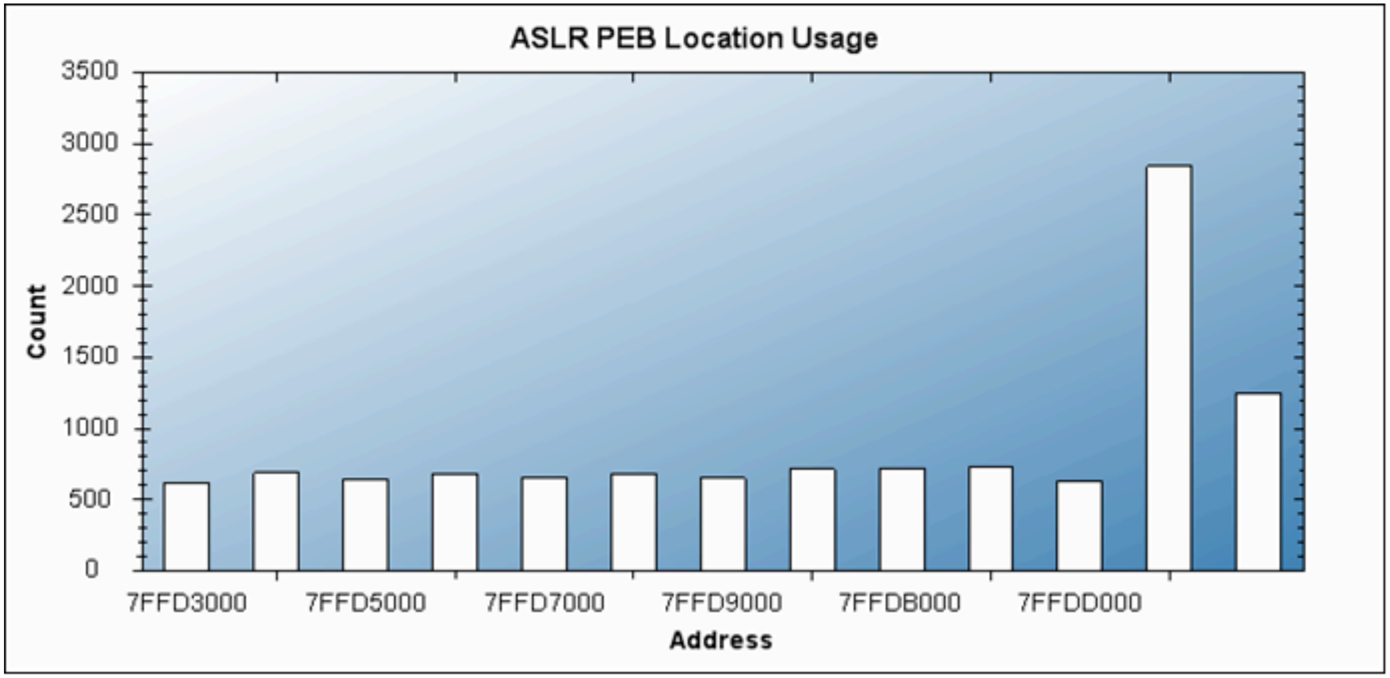


Image Randomization Bug

- Microsoft nice enough to provide offending code

```
if ((ImageInfo->ExportedImageInformation.ImageCharacteristics & IMAGE_FILE_DLL) == 0) {
//
// This is an executable not a DLL so don't consume the valuable DLL
// space for this (ie, it's better if we use the same VA space for
// all executables).
//

RelocateExe:

TSCStart = ReadTimeStampCounter ();

Delta = (ULONG) ((TSCStart & ((16 * _1mb) / X64K - 1)) * X64K);

if (Delta == 0) {
   Delta = X64K;
}
```

PEB Randomization Bug

- Microsoft nice enough to provide offending code again

```
KeQueryTickCount (&CurrentTime);

CurrentTime.LowPart &= ((X64K >> PAGE_SHIFT) - 1);
if (CurrentTime.LowPart <= 1) {
   CurrentTime.LowPart = 2;
}

//
// Select a varying PEB address without fragmenting the address space.
//

HighestVadAddress = (PVOID) ((PCHAR)HighestVadAddress - (CurrentTime.LowPart << PAGE_SHIFT));

if (MiCheckForConflictingVadExistence (TargetProcess, HighestVadAddress, (PVOID) ((PCHAR)
   HighestVadAddress + NumberOfBytes - 1)) == FALSE) {

//
// Got an address ...
//
   *Base = HighestVadAddress;
    goto AllocatedAddress;
}
```

ASLR – Other Observations

- Microsoft used RtlRandom instead of RtlRandomEx
  - “*The RtlRandomEx function is an improved version of the RtlRandom function*.”
  - “*Compared with the RtlRandom function, RtlRandomEx is twice as fast and produces better random numbers…*”
  - Microsoft have confirmed this will be resolved
- A Reseeding Method Was Also Discovered
  - This removed the requirement to reboot to get the image rebased
  - Simply update the last file write time
  - But produced some crazy results – paper contains more details

ASLR – Findings Summary

- Stack has pretty much uniform distribution
- Heap distribution is no where near uniform
- Using `HeapAlloc()` verus `malloc()` results in lower entropy in terms of locations used
- Both PEB and Image randomization have bugs in their implementation (the PEB bug has been present since XP SP2)
- End of the world?
  - Not really, just an increased likelihood of successful exploitation
  - But still better than no having anything at all
- When will these be fixed?
  - ETA is Windows Vista™ SP1 / Longhorn

symantec.

Confidence in a connected world.

Conclusions

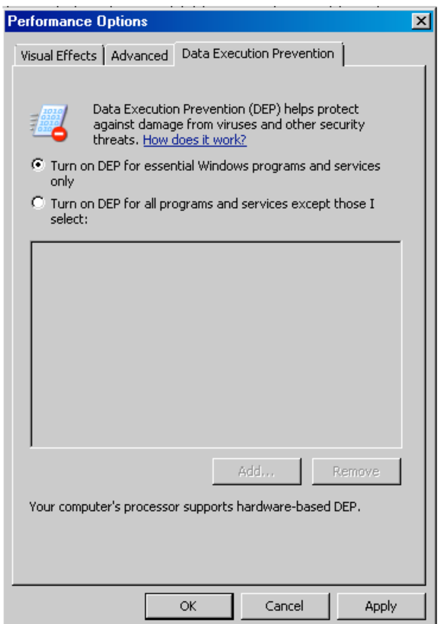
Conclusions

- We can now detect non GS protected binaries
  - This allows us to understand where lower hanging fruit is
- We can now detect non GS protected functions in GS binaries
  - Which have local stack variables
  - This again allows us to locate lower hanging fruit
- We know that binaries that use `HeapAlloc` are afforded less protection than those that use `malloc`
- We know that there are biases for the heap
- We know that image and PEB randomization have bugs
  - Which improve slightly the chance of successful exploitation

Finally

- GS White Paper
  - http://www.symantec/ - URL TBC
- ASLR White Paper
  - http://www.symantec/ - URL TBC
- Both papers contain supporting code
- Raw ASLR data available on request!
- Thanks to
  - Nitin Kumar Goel of Microsoft for his candidness
  - Zulfikar Ramzan and Matt Conover of Symantec for their help
  - Tim Newsham of iSEC Partners for his peer review and help
  - John Cartwright / Halvar Flake for their IDC code

For ASLR to be effective, DEP/NX must be enabled by default too.



Michael Howard, Microsoft

symantec.

Confidence in a connected world.

Thank You!

Ollie Whitehouse

ollie_whitehouse@symantec.com

http://www.symantec.com/