SYMANTEC ADVANCED THREAT RESEARCH

Analysis of GS protections in Microsoft® Windows Vista™

Ollie Whitehouse, Architect,
Symantec Advanced Threat Research

Symantec Advanced Threat Research

Analysis of GS protections in Microsoft Windows Vista

Contents

Abstract: The Microsoft Visual Studio® compiler supports a Buffer Security Check (GS) option to protect stack variables from overflows that result in arbitrary code execution. We developed techniques to identify the presence of GS protection in binaries and used them to identify which programs are and which programs are not protected by the GS option in the 32-bit RTM release of Windows Vista. We also measured the randomness of the GS cookies and the effect of Address Space Layout Randomization (ASLR) on the placement of the master cookie.

Introduction

The Visual Studio C++ compiler supports a Buffer Security Check option, known by its flag name, "GS." This option causes the compiler to add checks that protect the integrity of the return address and other important stack metadata associated with procedure invocation. The "GS" protections do not eliminate vulnerabilities, but rather make it more difficult for an attacker to exploit vulnerabilities.

We developed techniques to detect the presence of GS protections in binaries compiled with Visual Studio 2003 (VS2003) and Visual Studio 2005 (VS2005). We encountered several challenges when implementing GS detections, and this paper outlines our solutions to these challenges. We then used these techniques to analyze the binaries of a stock 32-bit RTM release of Microsoft Windows Vista. We found that most binaries were compiled with GS protections and were able to point out some binaries in the default installation that are not GS-protected. Finally, we measured the effects of Address Space Layout Randomization (ASLR) on the placement of the GS master cookie and measured the randomness of cookie values.

Prior research

There have been several papers describing buffer security checks and outlining attacks against them. One significant paper of note is Litchfield's paper on attacks against the Buffer Security Check implemented by Visual Studio 2003 [9]. Techniques for identifying segments of code within a binary have also been discussed previously, notably the FLIRT techniques used by DataRescue in their IDA disassembler [7]. To our knowledge, our work is the first to identify Buffer Security Checks in compiled binaries or to build a list of unprotected binaries in the Windows Vista RTM release.

Organization

The remainder of this paper is organized as follows: The next section provides an overview of the Buffer Security Check option as implemented in Visual Studio 2003 and 2005. The section titled "Measuring GS protections in Windows Vista" describes our techniques for identifying GS protections. The "Analysis" section presents our analysis of Windows Vista binaries using techniques presented in the previous section. The closing sections present our future goals and conclusions.

The Buffer Security Check options

GS overview

The Buffer Security Check option, known by its flag name "GS," is used to mitigate buffer overflow vulnerabilities in C and C++ code that allow an attacker to overwrite important stack data and seize control of the program. The primary goal of GS protection is to detect corruption of a function's return address that is stored on the stack and abort execution if corruption is detected. The GS feature also provides some other protections by careful layout of stack data.

The GS option was introduced in Visual Studio 2002 [14] and has undergone several revisions since then. The Windows Vista system was built primarily with Visual Studio 2003 (VS2003) and Visual Studio 2005 (VS2005), and we restrict our discussion to these versions.

The Visual Studio GS option works by placing a distinguished value, known as a cookie, onto the stack during the start of each function. A cookie value is copied from a program-wide master cookie and placed on the stack in between the function's return address and any space allocated for local variables. Because buffer overruns overwrite a contiguous range of memory, and because the cookie value is chosen to be unpredictable, it is assumed that if the cookie value has not been modified, a buffer overflow has not corrupted any data past the cookie, such as the return address. The cookie value on the stack is checked against the original master cookie at the end of the function before the function returns to ensure that it has not been overwritten either in a malicious manner or by accident. If the cookie is found to have been modified, the program is terminated. The code to place and check the cookie is integrated into the prologue and epilogue of each protected function during compilation.

The master cookie value that is copied onto the stack in the prologue and compared against in the epilogue is a global value initialized by the C runtime (CRT). While the program is starting up, the `__security_init_cookie` function is called to initialize the master cookie value and store its value in the `__security_cookie` variable [12].

There is a cost involved in implementing the GS feature. Additional code is added to every protected function, and additional stack storage is used to store cookie values. For this and other reasons, GS protection is not necessarily applied to all functions even if the GS option has been selected. A function will not be protected with GS protections if any of the following hold true:

- The optimization (O) option is not enabled
- The function does not contain a stack buffer
- The function is marked with *naked* in C++.
- The function has a variable argument list ("…")
- The function begins with inline assembly code
- The compiler determines that the function's variables are used only in ways that are less likely to be exploitable

Because of these restrictions, the GS option does not always protect vulnerable code. We identified one additional restriction not mentioned in the Microsoft documentation: Functions are only protected if they have a buffer of 5 bytes or more. Figure 1 shows an example of a vulnerable C program that will not be protected by the GS option. The function *vulnerable* has a vulnerability allowing writes past the end of the *foo* buffer. Because the buffer is only 4 bytes long, GS protections are not applied.

```
#include "stdafx.h"

void vulnerable(char *input){
        char foo[4];

        strcpy(foo, input);
}

int _tmain(int argc, _TCHAR* argv[])
{
        vulnerable(argv[1]);
        return 0;
}
```

Figure 1. Example code with a buffer that will not be protected by the GS option

The stack frame layout used by VS2003 is shown in the left-hand side of Figure 2. In this figure, smaller addresses are lower, larger addresses are higher, and the stack grows downwards while buffers extend upwards. The placement of the cookie protects the frame pointer and return address from buffer overflows in any of the locals. This design has several weaknesses as identified in Litchfield's paper [9]. The cookie does not protect the exception handler frame, and unless mitigated with the /SafeSEH option, an attacker can use the exception mechanism to execute arbitrary code. The cookie mechanism itself can also be defeated in some situations by using an *out* parameter to overwrite the original __security_cookie value with a known value.

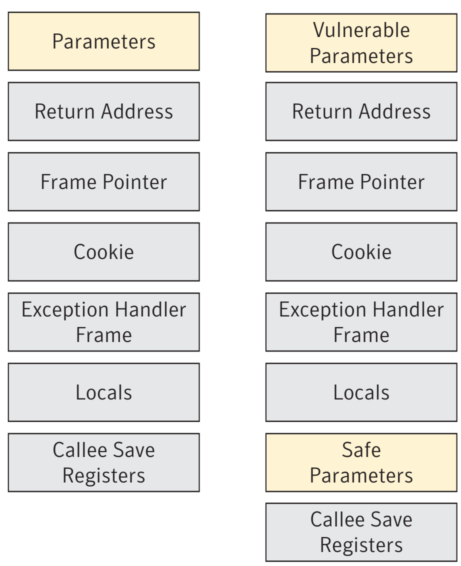

Figure 2. Stack frame layouts used in Visual Studio 2003 (left) and 2005 (right)

Microsoft addressed these issues in VS2005 (code-named Whidbey), with the stack frame layout shown in the right-hand side of Figure 2. These changes are described by Microsoft's Bray [2]:

"The Whidbey compiler will do something to address this by identifying vulnerable arguments and copying those arguments to memory addresses lower than the local buffers… The code of the function then makes use of the copy of the function argument rather than the original argument. We often refer to this as parameter shadowing… This improvement makes it more difficult to use out parameters and pass by reference variables to circumvent the security checks architecture. For example, in VC 2002 an out parameter that was changed by a buffer overrun to point to the __security_cookie variable would make it possible for an attacker to get a predictable cookie value thus preventing the security check in the function epilog from triggering."

Measuring GS protections in Windows Vista

We developed a tool for identifying GS protections in binaries. We call our tool GSAudit, which is available from the author on request. GSAudit identifies if a binary was compiled with GS protections or not. For programs that are GS-protected, GSAudit identifies which version of Visual Studio was used to compile the code. For reasons that will be discussed shortly, GSAudit also identifies how many GS checks are performed and where in the binary these checks occur for all VS2005-compiled binaries.

Debugger symbols are not available for all binaries, so our techniques do not rely on the presence of debugging symbols. To identify the presence of GS protections in a binary, we search for patterns present in the function prologue and epilogues generated by the GS option as well as some auxiliary functions and data. We use a technique similar to Data Rescue's FLIRT technology [7]. This method identifies functions by searching for matching machine code sequences while ignoring addresses that may have changed due to relocation during linking.

Identifying candidate binaries

Before we began to identify which binaries contained GS protection, it was necessary to find a list of candidate binaries. The GS option is only relevant to unmanaged C and C++ code, and we were interested in measuring how many of these binaries were protected.

To identify binaries compiled from C and C++, we parsed the PE headers of all candidate files to determine if the files had valid PE headers and if the program used native unmanaged code or managed code. The method we used to identify managed code is rudimentary: Any binary with a PE header that contained the string "MSCOREE.DLL" was considered to contain managed code. While this method is prone to false positives, we did not notice any false positives in a manual review of our results.

One additional refinement was necessary to deal with DLLs that did not contain any code and therefore were not eligible for GS protections: We only considered DLLs if they contained at least one executable section. While on the whole this approach worked well, it did miss a class of binary. We discovered several binaries that contain an executable .text section that is referenced by the COM+ runtime header, which did not contain any executable code. Figure 3 shows the PE header for such a DLL. Analyzing the executable section, we can see that it is small in size and does not contain executable code, as shown in Figure 4. We manually reviewed, identified, and removed all of these binaries from our results. All of the removed binaries had filenames ending with `.ni.dll`.

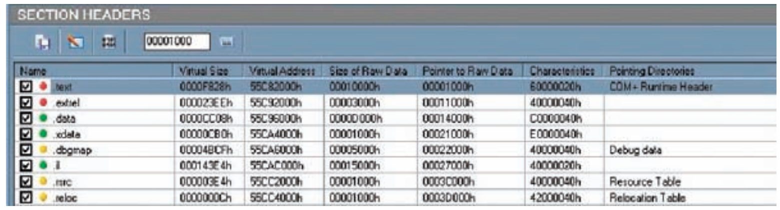

Figure 3. Section list in the PE header of an anomalous binary

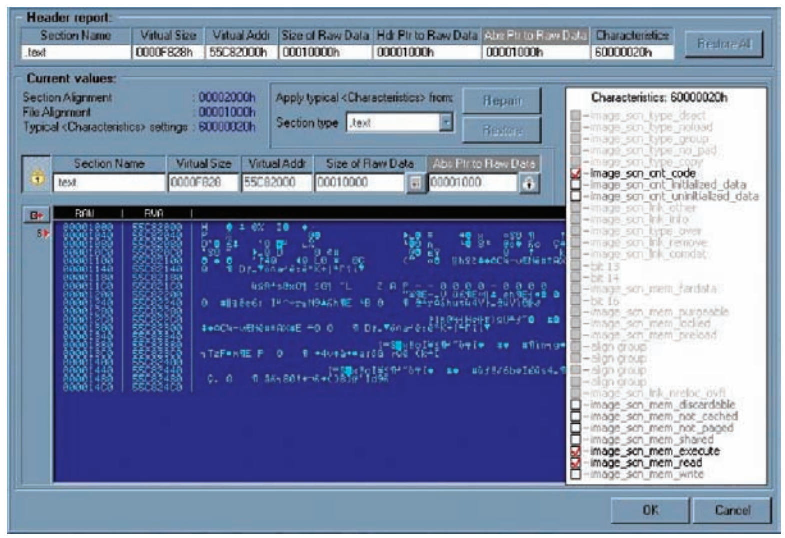

Figure 4. Section details for the anomalous binary

Detecting GS protection in Visual Studio 2003

Identifying the presence of GS protections in a binary compiled with Visual Studio 2003 is straightforward. The epilogue of a GS-protected function generated by Visual Studio 2003 is shown in Figure 5. It checks the validity of the stack cookie (in the ECX register) against the master cookie value (stored at L213194A8) and, if it has been tampered, jumps to an error-handling function. The error-handling function, shown in Figure 6, calls the __security_error_handler function in an external DLL.

```
213168E7   SUB_L213168E7:
213168E7     cmp   ecx,[L213194A8]
213168ED     jnz   L213168F0
213168EF     retn
213168F0   L213168F0:
213168F0     jmp   L213168B6
```

Figure 5. VS2003 function epilogue with cookie comparison code

We created a fingerprint based on the code in Figure 6 using a technique similar to FLIRT [7]. The fingerprint matches instructions while ignoring arguments that may be altered during relocation, such as the address in the CMP instruction. Searching binaries for code sequences that match this signature identifies the presence of GS protections.

```
213168B6   L213168B6:
213168B6      push 00000008h
213168B8      push L213024C8
213168BD      call SUB_L21316B44
213168C2      and  dword ptr [ebp-04h], 00000000h
213168C6      push 00000000h
213168C8      push 00000001h
213168CA      call jmp_MSVCR71.dll!
                            __security_error_handler
213168CF      pop  ecx
213168D0      pop  ecx
213168D1      jmp      L213168DA
213168D3   L213168D3:
213168D3      xor  eax,eax
213168D5      inc  eax
213168D6      retn
```

Figure 6. VS2003 error-handling code, which calls __security_error_handler

Detecting GS protection in Visual Studio 2005

We used similar techniques to identify GS protection in programs compiled with VS2005. Figure 7 shows a function epilogue in a program compiled with VS2005. The epilogue code is sometimes altered during optimization—for example, the RETN is sometimes implemented with REPL RETN—but the general structure remains unchanged. We identified seven variations and created a fingerprint that covers each of these cases, although one of the alternatives occasionally causes false matches. We will describe shortly how we dealt with false matches.

```
09204E16   SUB_L09204E16:
09204E16      cmp  ecx, [L092301CC]
09204E1C      jnz  L09204E27
09204E1E      test ecx, FFFF0000h
09204E24      jnz  L09204E27
09204E26      retn
```

Figure 7. VS2005 function epilogue with cookie comparison code

We encountered two problems when analyzing VS2005 binaries using this technique. First we noticed that some binaries that were compiled with VS2005 without using the GS option contained some functions with GS protections. We tracked down the source of these functions to standard libraries included with VS2005 that were compiled with GS protections. As a result, we cannot assume that a binary with GS-protected functions was compiled with the GS option. The second problem was that many device drivers were found to have GS code but never use GS protections. As a result, we cannot assume that a program with no GS-protected functions was compiled without the GS option.

Counting protected functions

To properly account for unprotected binaries that contain some statically linked protected functions, we decided to measure how many functions in a binary were GS-protected as a fraction of the total number of functions. Since the VS2003 libraries are not compiled with GS, this extra analysis was only performed for VS2005-compiled binaries.

First we count the number of functions that use GS protection. Identification is achieved by locating an epilogue with cookie-checking code. Successfully locating the epilogue allows us to obtain the address of the master cookie value, `__security_cookie`. Once we have this address, we can find all functions that access the security cookie searching the binary for code that loads this value, as shown in Figure 8. Since our epilogue fingerprint can result in false matches, an additional check is done to ensure that there is at least one such reference found. This extra check rejects the false epilogue matches mentioned previously.

```
mov   eax, __security_cookie
```

Figure 8. Code to access the security cookie

Next we compute the number of functions in the target binary. This problem is more difficult, and our solution is inexact, but we believe the measurements will still prove useful. To obtain the total number of functions in a binary, we used IDAPython [5] to count the functions identified by the IDA Pro 5.0 disassembler [4]. We wrote a separate tool, FuncDump.py, which produces a CSV file that contains the filename and the total number of functions. The source code is available from the author. The resulting CSV file is parsed by our GSAudit program and integrated into its analysis.

This technique does not always definitively determine whether a binary is GS-compiled. However, the results can still help security researchers identify binaries that may warrant further analysis. Binaries with a large number of functions and a low number of GS checks would be the most likely candidates for manual analysis.

Kernel drivers

We observed that some kernel drivers initialize a GS cookie (as demonstrated in Figure 9) but never use it in their execution. Our fingerprints won’t match these binaries since they do not have any protected functions, but the binaries are clearly compiled with the GS option. One could argue that these binaries do not leverage GS protection in their execution. However, it is important to understand why: These drivers use pointers and the heap during execution; they do not use local stack variables and hence do not need GS protection.

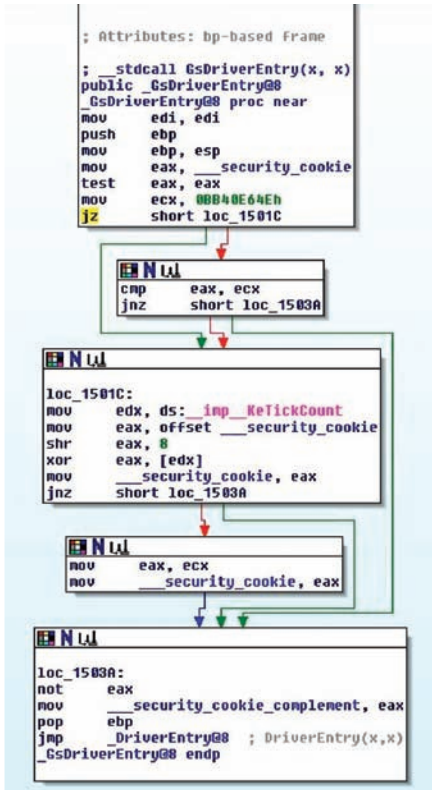

Figure 9. GS cookie initialization by kernel driver

To accommodate these drivers we created a fingerprint to match the cookie initialization code in the _GsDriverEntry function. The fingerprint is based on the code in Figure 10. We used this fingerprint to determine if a driver was compiled with GS protections even if no protected functions were found.

```
00012305   _GsDriverEntry@8 proc near
00012305     mov   edi, edi
00012307     push  ebp
00012308     mov   ebp, esp
0001230A     mov   eax, ___security_cookie
0001230F     test  eax, eax
00012311     mov   ecx, 0BB40h
00012316     jz    short loc_1231C
00012318     cmp   eax, ecx
0001231A     jnz   short loc_1233F
```

Figure 10. Sample GSDriverEntry code

Analysis

We used our GSAudit tool to measure the use of GS in Windows Vista. We identified and measured the DLL, SYS, and EXE files in the C:\Windows directory of a fresh installation of the 32-bit Windows Vista RTM release. We analyzed the results to determine how many of the binaries were compiled with GS protections and how many of the functions were protected. We then performed a more detailed manual analysis on a small, random sample of the unprotected binaries. We also measured the randomness of the GS cookies themselves, and the placement of the master cookie in memory.

GS master cookie values

To measure the randomness of the GS master cookie, we wrote a small program that prints out the master cookie value. We ran this program 2,340,878 times and did not notice any unexpected predictability in the measured cookie values. An analysis of the values showed that of the 2,340,878 values collected, no value was used more than once.

GS master cookie locations

To understand the impact of Address Space Layout Randomization (ASLR) on the location of the GS master cookie, we created a test program, GSCan.exe, which records the address of the __security_cookie master cookie to a CSV file. The source to this program is available from the author. We compiled the test program with the ASLR and GS options (/dynamicbase /GS) and measured 15,000 executions on several platforms: an AMD3200, 32-bit VMware Server 1.0.3, and Microsoft Virtual Server 2005 R2 x64. All three platforms were running a fresh install of the 32-bit Windows Vista RTM. We observed that the GS master cookie could be located at 1 of 255 locations in memory and the frequency distribution of locations was not uniform. Plots of these distributions can be found in Appendix 0.

The environment with the most uniform distribution was VMware Server 1.0.1. However, the VMware Server data exhibited a bias toward using one memory address more than any other. The address occurring most frequently was used 114 times, while all other addresses were only used between 35 and 85 times each. As a result, an attack using an arbitrary pointer-overwrite to target the master cookie would have a much greater chance of success if they chose the most frequently occurring address (0.8 percent) than if they chose an address at random (0.4 percent).[1]

The distributions observed when testing the AMD3200 platform and the Microsoft Virtual Server platform were even less uniform. In both cases half of the addresses were used significantly more often than the other half. On native hardware, 127 addresses were used between 2 and 19 times, while the other 128 addresses were used between 80 and 137 times. In the case of virtualized hardware, this resulted in 127 addresses being used between 9 and 36 times, while the other 128 addresses were used between 74 and 116 times.

[1] This sort of attack can be mitigated by a technique described by Litchfield [9]. The technique, which uses `VirtualProtect` to protect the master cookie from being overwritten, is not currently implemented in Windows Vista.

Our measurements indicate that the amount of entropy in the address randomization differs between environments. However, we should point out that our ASLR measurements were made without rebooting the system between measurements. Microsoft has said that, on each reboot, the addresses of binaries protected with ASLR will be changed. To validate our measurements, we configured the VMware and AMD3200 platforms to boot up, run our test case, and reboot. We used this setup to measure an additional 10,000 iterations and again observed biases towards a single (albeit different) memory address. The data can be found in Appendix 0.

Binaries without GS code

We used our GSAudit tool to construct a list of all binaries in the Windows Vista RTM in the C:\Windows directory that did not contain GS protections. The complete list can be found in Appendix IV.

It should be noted that Microsoft did not write all the binaries that are installed with Windows Vista. We identified many binaries that were written by third parties or were legacy libraries from previous versions of Windows.

Since we do not have a foolproof method for detecting whether VS2005-compiled binaries were compiled with GS protections, we decided to perform a manual analysis of a random selection of binaries. Figure 11 summarizes the highlights of our manual analysis. We also plotted the number of GS-protected functions against the total number of functions to see if there was a correlation. This plot is shown in Appendix 0 and shows that there is not a strong correlation.

<table>
<tr><th>File Analyzed</th><th>Finding</th></tr>
<tr><td><code>rdpdr.sys</code>
1000 functions,
30 checks</td><td>Located function <code>PiRegStateOpenClassKey</code>, which contains a stack-based variable of 46 bytes used in a <code>_snwprintf</code> operation.[2] The function is GS-protected, which indicates that the code is GS-compiled. However, it should be noted that Microsoft has deprecated the <code>_snwprintf</code>[3] function for the <code>_snwprintf_s</code> function instead. This indicates that Microsoft is not adhering to secure development practices in all cases.

Microsoft was kind enough to supply the author with details as to why this is used. The following is taken from the source code.

<code>//</code>
<code>// Convert the binary GUID into // its corresponding unicode</code>
<code>// string.</code>

<code>// Note: _snwprintf is used in</code>
<code>// place of RtlStringCchPrintfW, // so as not to drag in</code>
<code>// ntstrsafe.lib, which would be // required for w2k</code>
<code>// compatibility as _vsnwprintf // is not exported by that OS.</code>
<code>//</code></td></tr>
<tr><td><code>wmp.dll</code>
38,871 functions,
1,568 checks</td><td>Located function Session:_SetLockTimeoutError, which contains a stack-based variable of 198 bytes used in a _snwprintf_s operation. The function is GS-protected, which indicates that the code is GS-compiled.</td></tr>
<tr><td><code>nvlddmkm.sys</code>
8,250 functions,
2 checks</td><td>Numerous functions that perform unsafe string operations. This indicates that the code is not GS-compiled. However, it should be noted that this driver is described as “NVIDA Compatible Windows 2000 Miniport Drivers” and is copyrighted by NVIDA Corporation.</td></tr>
<tr><td><code>Wlanapi.dll</code>
166 functions,
3 checks</td><td>No local variables discovered over 4 bytes in length. However, of note was the <code>AcmReasonCodeToString()</code> function. This function calls <code>LoadStringW()</code>,[4] which is documented by Microsoft as being dangerous. This function is exposed to other applications through the <code>WLanReasonCodeToString</code> function.[5] The buffer and size are user-supplied. The only input validation ensures that they are not NULL. If the calling application was not GS-compiled, then this code could introduce a stack-based overflow to the application.</td></tr>
<tr><td><code>MrxSmb.sys</code>
294 functions,
4 checks</td><td>Located function <code>MrxDaveSkipLrps</code>, which contains a stack-based variable of 26 bytes. The function is GS-protected; this indicates that the code is GS-compiled.</td></tr>
</table>

Figure 11. Manual analysis results

[2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_crt__snprintf.2c_._snwprintf.asp
[3] http://msdn2.microsoft.com/en-us/library/ms235384(VS.80).aspx
[4] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/resources/strings/stringreference/stringfunctions/loadstring.asp
[5] http://msdn2.microsoft.com/en-us/library/ms706768.aspx

Future research

Our GSAudit tool can identify binaries that are not or may not be protected by compiler GS checks. However, this identification does not yield the most useful results. A more interesting measurement would be a list of functions that both lack GS protection and contain local buffers. It should be possible to identify functions that contain local buffers but do not perform GS checks by analyzing the assembly code of a function. Automating this process would allow us to identify the binaries that have potentially exploitable stack buffers, without the need for manual investigation. We've begun prototyping a tool that makes use of debugging symbols, the IDA disassembler, and code from Bugscam [6] to identify the size of local stack variables in functions that do not have GS protection. Since IDA can incorrectly identify local stack buffers, our prototype currently suffers from a high number of false positives. However, the approach appears promising, and future work will be aimed at refining the process.

Finally, it is not clear under what situations the compiler will omit GS checks. Microsoft's documentation states that GS is not used "if a parameter is used only in ways that are less likely to be exploitable in the event of a buffer overrun." [11] This statement is vague and imprecise. To better understand this, we intend to investigate which coding styles and constructs result in code that would not be protected.

Conclusions

We've described the implementation of GSAudit, which can programmatically identify GS-compiled binaries from VS2003. Although it can't definitively detect whether a binary that was compiled with VS2005 was built with the GS option, it can identify the number of functions that utilize GS functionality compared with the total number of functions in the binary. This approach provides a good indication of binaries that might not have been compiled with the GS option, significantly reducing the number of binaries that must be manually investigated. We've also described ongoing efforts to improve the usefulness of our tool and further reduce the need for manual analysis.

We analyzed the binaries provided with Windows Vista, and while most binaries were compiled with GS protections, we were able to identify binaries that were not compiled with GS protections. Our techniques pointed out several binaries that had few protected functions, and through manual analysis, we were able to identify a binary that, while containing GS-protected functions, was not itself compiled with GS protections. This provides confirmation that our technique of counting GS-protected functions is useful. However, the amount of manual investigation required is still significant. We hope ongoing research can address this issue.

During our analysis we observed that Address Space Layout Randomization (ASLR) does not place the GS master cookie as randomly as it could. We observed that the amount of entropy in the address randomization differs across platforms and is not uniform on any of the platforms we tested.

It is also clear from this research that there is no statistical link between the total number of functions in a binary and the number of functions that use local stack variables.

Acknowledgments

The author would like to acknowledge the help and support of Oliver Friedrichs, Matt Conover, and Orlando Padillia of Symantec; and Tim Newsham. The author would also like to acknowledge Nitin Kumar Goel of Microsoft, who reviewed this research and provided candid feedback.

References

1. B. Bray, "Compiler Security Checks In Depth," Feb. 2002, www.codeproject.com/tips/seccheck.asp
2. B. Bray, "Security Improvements to the Whidbey C Compiler," Nov. 2003, http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx
3. E. Carrera, "Introduction to IDAPython," June 2005, http://dkbza.org/idapython_intro.html
4. DataRescue, "IDA Pro," www.datarescue.com/idabase/index.htm
5. G. Erdelyi, "IDAPython," http://d-dome.net/idapython
6. H. Flake, "BugScam," http://sourceforge.net/projects/bugscam
7. I. Guilfanov, "Fast Library Identification and Recognition Technology," 1997, www.datarescue.com/idabase/flirt.htm
8. M. Howard, "Address Space Layout Randomization in Windows Vista," May 2006, http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx
9. D. Litchfield, "Defeating the Stack Based Overflow Prevention Mechanism of Microsoft Windows 2003 Server," Sept. 2003, www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
10. D. Litchfield, "Buffer Underruns, DEP, ASLR and Improving the Exploitation Prevention Mechanisms (XPMs) on the Windows Platform," Sept. 2005, www.ngssoftware.com/research/papers/xpms.pdf
11. Microsoft, "GS (Buffer Security Check)," http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx
12. Microsoft, "Run-Time Library Reference—__security_init_cookie," http://msdn2.microsoft.com/en-us/library/ms235362(VS.80).aspx
13. Microsoft, "Microsoft Portable Execution and Common Object File Format Specification," May 2006, www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
14. S. Toub, "Write Faster Code with Modern Language Features of Visual C++ 2005," MSDN Magazine, May 2004, http://msdn.microsoft.com/msdnmag/issues/04/05/VisualC2005/#S5

Appendix I. GSAudit Results

The following scatter graph (Figure 12) shows that, in general, there is an upward trend in the number of GS checks made in a binary when compared with the total number of function calls available. The data included in Figure 12 only includes binaries that have less than 10,000 function calls. The result is that 14 binaries were omitted.

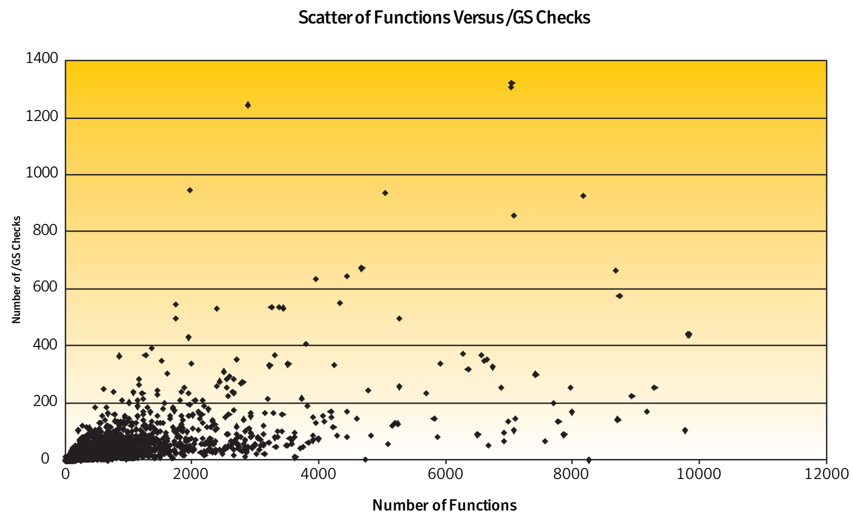

Figure 12. Comparison between number of GS checks and total number of functions for binaries with less than 10,000 functions

When sorted and plotted, we can see a general upward trend, with peaks at the end of each GS check range where a binary has a large number of functions.

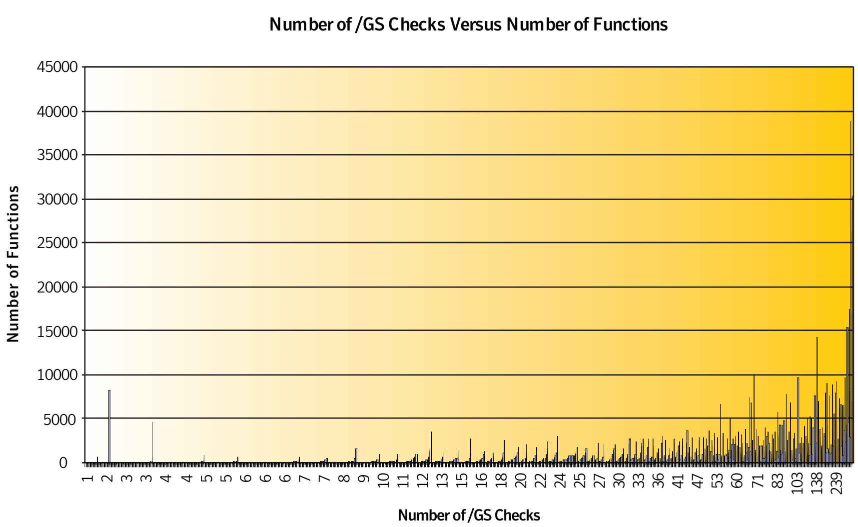

Figure 13. A sorted comparison between the number of GS checks and the total number of functions

Figure 13 indicates an upward trend, but no statistical relationship between the number of functions that contain GS cookies and the total number of functions. This lack of relationship is due to the fact that there is no statistical link between the total number of functions that leverage local stack variables and the total number of functions in a binary.

Appendix II. Location of __security_cookie (Without Reboots)

The first graph (Figure 14) shows the location of `__security_cookie` on Windows Vista RTM 32bit under Microsoft Virtual Server 2005 R2 running on Windows XP x64. This is from a run of 15,000.

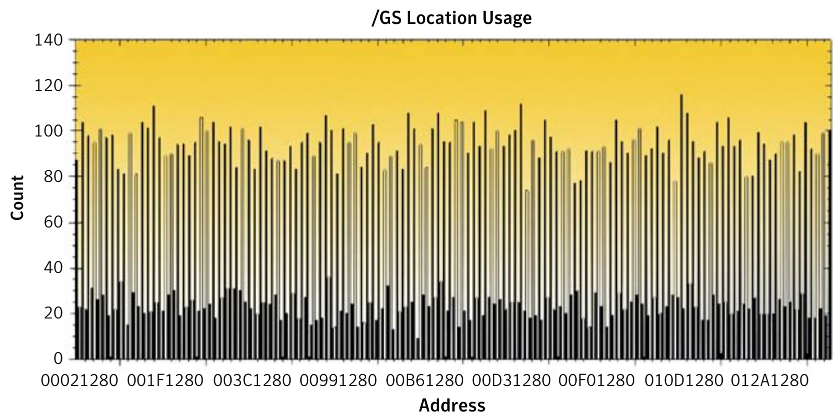

Figure 14

Figure 15 shows the location of `__security_cookie` on Windows Vista RTM 32-bit under VMware Server 1.0.1 32-bit running on Windows XP x64. This is from a run of 15,000.

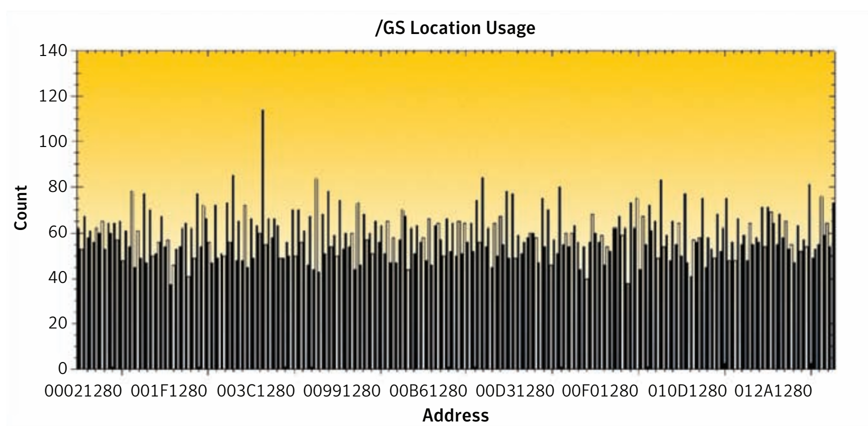

Figure 15

Figure 16 shows the location of __security_cookie on Windows Vista RTM 32-bit under native hardware. This is from a run of 15,000.

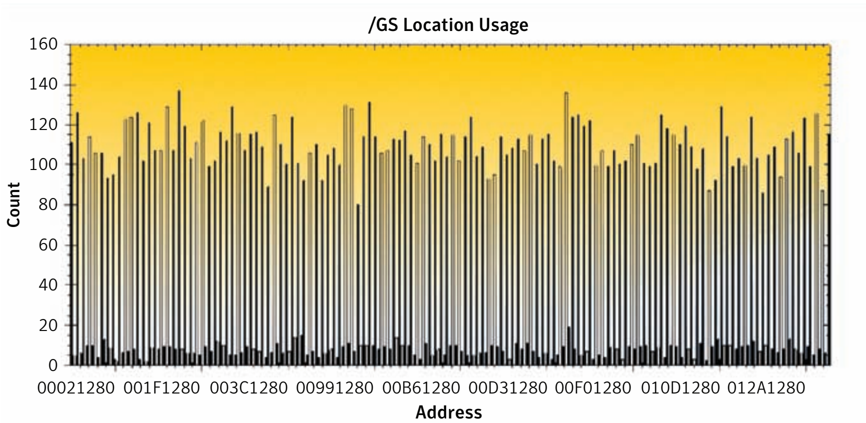

Figure 16

What can be seen from the above is all three have different randomness profiles. Figure 17 shows the location of __security_cookie on Windows Vista RTM 32-bit under native hardware with a reboot between each plot. This is from a run of 900.

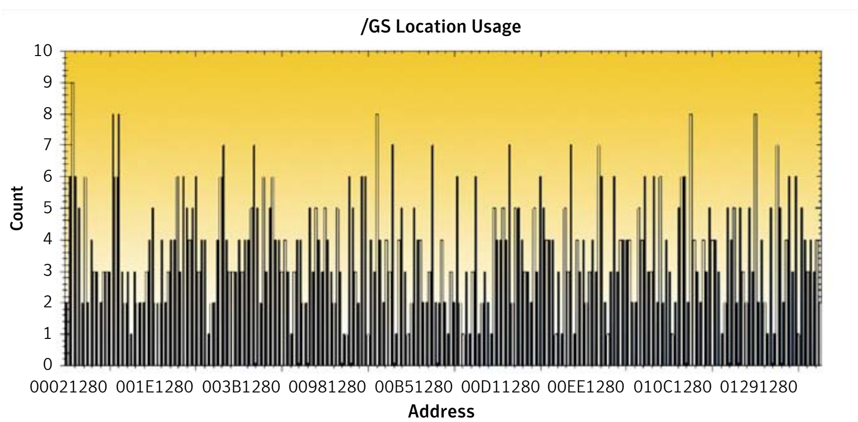

Figure 17

Appendix III. Location of __security_cookie (With Reboots)

Figure 18 shows the location of `__security_cookie` on Windows Vista RTM 32bit under VMWare Server 1.0.1 32bit running on Windows XP x64 with a reboot between each plot. This is from a run of 10,000.

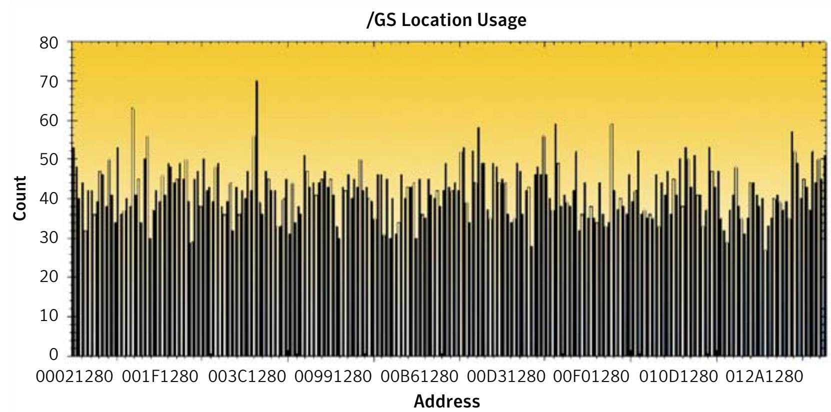

Figure 18

Figure 19 shows the location of `__security_cookie` on Windows Vista RTM 32-bit under native hardware with a reboot between each plot. This is from a run of 10,000.

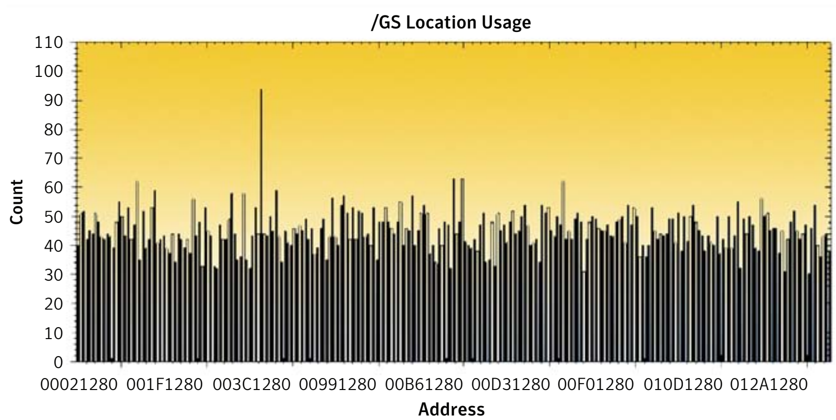

Figure 19

Appendix IV. Table of Binaries Without GS Code from Windows Vista 32-bit RTM

The following table shows the binaries from C:\Windows that were found not to contain GS code.

c:\windows\assembly\NativeImages_v2.0.50727_32\System.EnterpriseSe#\59192aecec284fba3e9b4b6ec41a755d\System.EnterpriseServices.Wrapper.dll	c:\windows\Boot\PCAT\memtest.exe
c:\windows\Microsoft.NET\Framework\sbs_diasymreader.dll	c:\windows\Microsoft.NET\Framework\sbs_iehost.dll
c:\windows\Microsoft.NET\Framework\sbs_microsoft.jscript.dll	c:\windows\Microsoft.NET\Framework\sbs_microsoft.vsa.vb.codedomprocessor.dll
c:\windows\Microsoft.NET\Framework\sbs_mscordbi.dll	c:\windows\Microsoft.NET\Framework\sbs_mscorrc.dll
c:\windows\Microsoft.NET\Framework\sbs_mscorsec.dll	c:\windows\Microsoft.NET\Framework\sbs_system.configuration.install.dll
c:\windows\Microsoft.NET\Framework\sbs_system.data.dll	c:\windows\Microsoft.NET\Framework\sbs_system.enterpriseservices.dll
c:\windows\Microsoft.NET\Framework\sbs_VsaVb7rt.dll	c:\windows\Microsoft.NET\Framework\sbs_wminet_utils.dll
c:\windows\SoftwareDistribution\Download\Install\mpas-d.exe	c:\windows\System32\Boot\winload.exe
c:\windows\System32\Boot\winresume.exe	c:\windows\System32\BOOTVID.DLL
c:\windows\System32\crtdll.dll	c:\windows\System32\ctl3d32.dll
c:\windows\System32\C_ISCII.DLL	c:\windows\System32\drivers\hgfs.sys
c:\windows\System32\DriverStore\FileRepository\atiixpad.inf_e8d83e66\ati2drad.dll	c:\windows\System32\DriverStore\FileRepository\atiixpad.inf_e8d83e66\ati2mpad.sys
c:\windows\System32\DriverStore\FileRepository\atiixpag.inf_6b9aff66\ati2cqag.dll	c:\windows\System32\DriverStore\FileRepository\atiixpag.inf_6b9aff66\ati2dvag.dll
c:\windows\System32\DriverStore\FileRepository\atiixpag.inf_6b9aff66\ati3duag.dll	c:\windows\System32\DriverStore\FileRepository\atiixpag.inf_6b9aff66\atikvmag.dll
c:\windows\System32\DriverStore\FileRepository\atiixpag.inf_6b9aff66\ativvaxx.dll	c:\windows\System32\DriverStore\FileRepository\hal.inf_59c500ab\halacpi.dll
c:\windows\System32\DriverStore\FileRepository\hal.inf_59c500ab\halmacpi.dll	c:\windows\System32\DriverStore\FileRepository\hpojscan.inf_c876c5d8\hpojwia.dll
c:\windows\System32\DriverStore\FileRepository\ialmnt5.inf_c1262cce\ialmdd5.dll	c:\windows\System32\DriverStore\FileRepository\ialmnt5.inf_c1262cce\ialmdev5.dll
c:\windows\System32\DriverStore\FileRepository\ialmnt5.inf_c1262cce\ialmdnt5.dll	c:\windows\System32\DriverStore\FileRepository\ialmnt5.inf_c1262cce\ialmnt5.sys
c:\windows\System32\DriverStore\FileRepository\ialmnt5.inf_c1262cce\ialmrnt5.dll	c:\windows\System32\DriverStore\FileRepository\nv4_disp.inf_73ea8d0d\nv4_disp.dll
c:\windows\System32\DriverStore\FileRepository\nv4_disp.inf_73ea8d0d\nv4_mini.sys	c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXAAFCIC.DLL
c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXACFCIC.DLL	c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXADFCIC.DLL
c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXAEFCIC.DLL	c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXCAFCIC.DLL
c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXMAFCIC.DLL	c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXMDFCIC.DLL

c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXROFCIC.DLL	c:\windows\System32\DriverStore\FileRepository\prnlx001.inf_f13f0471\I386\LXSYFCIC.DLL
c:\windows\System32\DriverStore\FileRepository\ps5333nu.inf_e0d01920\s3gNB.dll	c:\windows\System32\DriverStore\FileRepository\ps5333nu.inf_e0d01920\s3gNBm.sys
c:\windows\System32\DriverStore\FileRepository\sisgr.inf_e9f71680\sisgrp.sys	c:\windows\System32\DriverStore\FileRepository\sisgr.inf_e9f71680\sisgrv.dll
c:\windows\winsxs\x86_subsystem-for-unix-based-applications_31bf3856ad364e35_6.0.6000.16386_none_71b195c9f3048b05\posixsscom.dll	c:\windows\System32\DriverStore\FileRepository\wdma_via.inf_42fdb9e8\ac97via.sys
c:\windows\System32\dxmasf.dll	c:\windows\System32\expsrv.dll
c:\windows\System32\hal.dll	c:\windows\System32\halacpi.dll
c:\windows\System32\halmacpi.dll	c:\windows\System32\iac25_32.ax
c:\windows\System32\ir32_32.dll	c:\windows\System32\ir41_32.ax
c:\windows\System32\ir41_qc.dll	c:\windows\System32\ir41_qcx.dll
c:\windows\System32\ir50_32.dll	c:\windows\System32\ir50_qc.dll
c:\windows\System32\ir50_qcx.dll	c:\windows\System32\ivfsrc.ax
c:\windows\System32\KBDJPN.DLL	c:\windows\System32\KBDKOR.DLL
c:\windows\System32\kd1394.dll	c:\windows\System32\kdcom.dll
c:\windows\System32\kdusb.dll	c:\windows\System32\ksuser.dll
c:\windows\System32\mfc40.dll	c:\windows\System32\mfc40u.dll
c:\windows\System32\msdxm.ocx	c:\windows\System32\msimg32.dll
c:\windows\System32\msvbvm60.dll	c:\windows\System32\msvcrt20.dll
c:\windows\System32\panmap.dll	c:\windows\System32\PSHED.DLL
c:\windows\System32\setupcl.exe	c:\windows\System32\sfc.dll
c:\windows\System32\shunimpl.dll	c:\windows\System32\sqlunirl.dll
c:\windows\System32\sqlwid.dll	c:\windows\System32\sqlwoa.dll
c:\windows\System32\vbajet32.dll	c:\windows\System32\vfpodbc.dll
c:\windows\System32\drivers\lgtosync.sys	c:\windows\System32\winload.exe
c:\windows\System32\winresume.exe	c:\windows\winsxs\Backup\x86_microsoft-windows-b..environment-windows_31bf3856ad364e35_6.0.6000.16386_none_6701d52e8fdf8d45_winload.exe_75835076
c:\windows\winsxs\Backup\x86_microsoft-windows-b..environment-windows_31bf3856ad364e35_6.0.6000.16386_none_6701d52e8fdf8d45_winresume.exe_85cd1215	c:\windows\winsxs\Backup\x86_microsoft-windows-b..gertransport-serial_31bf3856ad364e35_6.0.6000.16386_none_0f7ecb22afbfde41_kdcom.dll_db5e7744
c:\windows\winsxs\Backup\x86_microsoft-windows-b..re-memorydiagnostic_31bf3856ad364e35_6.0.6000.16386_none_d5fe8c6e07b249ea_memtest.exe_01d80391	c:\windows\winsxs\Backup\x86_microsoft-windows-bootvid_31bf3856ad364e35_6.0.6000.16386_none_3642b97d89494bc7_bootvid.dll_c188118d
c:\windows\winsxs\Backup\x86_microsoft-windows-gdi-painting_31bf3856ad364e35_6.0.6000.16386_none_7535161f1f2100ed_msimg32.dll_2a4e0bd8	c:\windows\winsxs\Backup\x86_microsoft-windows-pshed_31bf3856ad364e35_6.0.6000.16386_none_59bc215430297e40_pshed.dll_f6ac239e

c:\windows\winsxs\x86_microsoft-network-internet-access_31bf3856ad364e35_6.0.6000.16386_none_b85711c14117830d\cclitesetupui.exe	c:\windows\winsxs\x86_microsoft-windows-b..buggertransport-usb_31bf3856ad364e35_6.0.6000.16386_none_9b46e79f0d9c56ff\kdusb.dll
c:\windows\winsxs\x86_microsoft-windows-b..environment-windows_31bf3856ad364e35_6.0.6000.16386_none_6701d52e8fdf8d45\winload.exe	c:\windows\winsxs\x86_microsoft-windows-b..environment-windows_31bf3856ad364e35_6.0.6000.16386_none_6701d52e8fdf8d45\winresume.exe
c:\windows\winsxs\x86_microsoft-windows-b..gertransport-serial_31bf3856ad364e35_6.0.6000.16386_none_0f7ecb22afbfde41\kdcom.dll	c:\windows\winsxs\x86_microsoft-windows-b..re-memorydiagnostic_31bf3856ad364e35_6.0.6000.16386_none_d5fe8c6e07b249ea\memtest.exe
c:\windows\winsxs\x86_microsoft-windows-b..uggertransport-1394_31bf3856ad364e35_6.0.6000.16386_none_61949536f6f76e24\kd1394.dll	c:\windows\winsxs\x86_microsoft-windows-bootvid_31bf3856ad364e35_6.0.6000.16386_none_3642b97d89494bc7\BOOTVID.DLL
c:\windows\winsxs\x86_microsoft-windows-crtdll_31bf3856ad364e35_6.0.6000.16386_none_df9e2f858dc40ff1\crtdll.dll	c:\windows\winsxs\x86_microsoft-windows-ctl3d32_31bf3856ad364e35_6.0.6000.16386_none_c7f2246c57358efd\ctl3d32.dll
c:\windows\winsxs\x86_microsoft-windows-d..tshow-kernelsupport_31bf3856ad364e35_6.0.6000.16386_none_e5cada609a6133bd\ksuser.dll	c:\windows\winsxs\x86_microsoft-windows-gdi-painting_31bf3856ad364e35_6.0.6000.16386_none_7535161f1f2100ed\msimg32.dll
c:\windows\winsxs\x86_microsoft-windows-i..l-keyboard-00000411_31bf3856ad364e35_6.0.6000.16386_none_e50b4b87674cc257\KBDJPN.DLL	c:\windows\winsxs\x86_microsoft-windows-i..l-keyboard-00000412_31bf3856ad364e35_6.0.6000.16386_none_e57cd2b56703c6de\KBDKOR.DLL
c:\windows\winsxs\x86_microsoft-windows-i..odepage-57002-57011_31bf3856ad364e35_6.0.6000.16386_none_3734d6eadb683c21\C_ISCII.DLL	c:\windows\winsxs\x86_microsoft-windows-indeo4-codecs_31bf3856ad364e35_6.0.6000.16386_none_39975c8d5a6988b1\ir41_32.ax
c:\windows\winsxs\x86_microsoft-windows-indeo4-codecs_31bf3856ad364e35_6.0.6000.16386_none_39975c8d5a6988b1\ir41_qc.dll	c:\windows\winsxs\x86_microsoft-windows-indeo4-codecs_31bf3856ad364e35_6.0.6000.16386_none_39975c8d5a6988b1\ir41_qcx.dll
c:\windows\winsxs\x86_microsoft-windows-indeo5-codecs_31bf3856ad364e35_6.0.6000.16386_none_22c9c1557410d750\iac25_32.ax	c:\windows\winsxs\x86_microsoft-windows-indeo5-codecs_31bf3856ad364e35_6.0.6000.16386_none_22c9c1557410d750\ir50_32.dll
c:\windows\winsxs\x86_microsoft-windows-indeo5-codecs_31bf3856ad364e35_6.0.6000.16386_none_22c9c1557410d750\ir50_qc.dll	c:\windows\winsxs\x86_microsoft-windows-indeo5-codecs_31bf3856ad364e35_6.0.6000.16386_none_22c9c1557410d750\ir50_qcx.dll
c:\windows\winsxs\x86_microsoft-windows-indeo5-codecs_31bf3856ad364e35_6.0.6000.16386_none_22c9c1557410d750\ivfsrc.ax	c:\windows\winsxs\x86_microsoft-windows-m..nents-mdac-odbc-jet_31bf3856ad364e35_6.0.6000.16386_none_c91f67973cf2633d\vfpodbc.dll
c:\windows\winsxs\x86_microsoft-windows-m..nents-mdac-sqlunirl_31bf3856ad364e35_6.0.6000.16386_none_39dff6607f42ed85\sqlunirl.dll	c:\windows\winsxs\x86_microsoft-windows-m..ponents-mdac-sqlwid_31bf3856ad364e35_6.0.6000.16386_none_17440058708f9849\sqlwid.dll
c:\windows\winsxs\x86_microsoft-windows-m..ponents-mdac-sqlwoa_31bf3856ad364e35_6.0.6000.16386_none_174a466c7089e370\sqlwoa.dll	c:\windows\winsxs\x86_microsoft-windows-m..s-components-jetvba_31bf3856ad364e35_6.0.6000.16386_none_735b8f8d953639a8\expsrv.dll
c:\windows\winsxs\x86_microsoft-windows-m..s-components-jetvba_31bf3856ad364e35_6.0.6000.16386_none_735b8f8d953639a8\vbajet32.dll	c:\windows\winsxs\x86_microsoft-windows-mediaplayer-core_31bf3856ad364e35_6.0.6000.16386_none_09330123522ea8c1\dxmasf.dll
c:\windows\winsxs\x86_microsoft-windows-mediaplayer-core_31bf3856ad364e35_6.0.6000.16386_none_09330123522ea8c1\msdxm.ocx	c:\windows\winsxs\x86_microsoft-windows-mfc40u_31bf3856ad364e35_6.0.6000.16386_none_f0dc500958a528b5\mfc40u.dll

c:\windows\winsxs\x86_microsoft-windows-mfc40_31bf3856ad364e35_6.0.6000.16386_none_57c82c1ae4dbe668\mfc40.dll	c:\windows\winsxs\x86_microsoft-windows-msvbvm60_31bf3856ad364e35_6.0.6000.16386_none_c04d02d754cecca9\msvbvm60.dll
c:\windows\winsxs\x86_microsoft-windows-msvcrt20_31bf3856ad364e35_6.0.6000.16386_none_ebed1a7373e6e8e7\msvcrt20.dll	c:\windows\winsxs\x86_microsoft-windows-panmap_31bf3856ad364e35_6.0.6000.16386_none_67259240223a18cd\panmap.dll
c:\windows\winsxs\x86_microsoft-windows-pshed_31bf3856ad364e35_6.0.6000.16386_none_59bc215430297e40\PSHED.DLL	c:\windows\winsxs\x86_microsoft-windows-setupcl_31bf3856ad364e35_6.0.6000.16386_none_567843d7ee5cdd00\setupcl.exe
c:\windows\winsxs\x86_microsoft-windows-sfc_31bf3856ad364e35_6.0.6000.16386_none_a4ff01505f4694a4\sfc.dll	c:\windows\winsxs\x86_microsoft-windows-shunimpl_31bf3856ad364e35_6.0.6000.16386_none_535fb43f376a866c\shunimpl.dll
c:\windows\winsxs\x86_microsoft-windows-vcm-core-codecs_31bf3856ad364e35_6.0.6000.16386_none_6a6bff15db84b924\ir32_32.dll	c:\windows\winsxs\x86_microsoft.windows.i..utomation.proxystub_6595b64144ccf1df_1.0.6000.16386_none_b80a29519535473c\sxsoaps.dll
c:\windows\winsxs\x86_netfx-sbs_diasymreader_dll_31bf3856ad364e35_6.0.6000.16386_none_a4786bd9e234ccdf\sbs_diasymreader.dll	c:\windows\winsxs\x86_netfx-sbs_iehost_dll_31bf3856ad364e35_6.0.6000.16386_none_158168a6457f1679\sbs_iehost.dll
c:\windows\winsxs\x86_netfx-sbs_microsoft_jscript_dll_31bf3856ad364e35_6.0.6000.16386_none_faa03a14948da139\sbs_microsoft.jscript.dll	c:\windows\winsxs\x86_netfx-sbs_mscordbi_dll_31bf3856ad364e35_6.0.6000.16386_none_60f937e93a64acb2\sbs_mscordbi.dll
c:\windows\winsxs\x86_netfx-sbs_mscorrc_dll_31bf3856ad364e35_6.0.6000.16386_none_9f231a637063fa04\sbs_mscorrc.dll	c:\windows\winsxs\x86_netfx-sbs_mscorsec_dll_31bf3856ad364e35_6.0.6000.16386_none_e3c6ee04df465dd4\sbs_mscorsec.dll
c:\windows\winsxs\x86_netfx-sbs_ms_vsa_vb_codedomproc_31bf3856ad364e35_6.0.6000.16386_none_f5727b5699105db2\sbs_microsoft.vsa.vb.codedomprocessor.dll	c:\windows\winsxs\x86_netfx-sbs_sys_config_install_dll_31bf3856ad364e35_6.0.6000.16386_none_bd13919a387c95c9\sbs_system.configuration.install.dll
c:\windows\winsxs\x86_netfx-sbs_sys_data_dll_31bf3856ad364e35_6.0.6000.16386_none_fc52ff10efdba1d1\sbs_system.data.dll	c:\windows\winsxs\x86_netfx-sbs_sys_enterprisesvc_dll_31bf3856ad364e35_6.0.6000.16386_none_5ef2978f28e693be\sbs_system.enterpriseservices.dll
c:\windows\winsxs\x86_netfx-sbs_vsavb7rt_dll_31bf3856ad364e35_6.0.6000.16386_none_9178a41770a55dbc\sbs_VsaVb7rt.dll	c:\windows\winsxs\x86_netfx-sbs_wminet_utils_dll_31bf3856ad364e35_6.0.6000.16386_none_fe642fbd88d269cd\sbs_wminet_utils.dll

About Symantec

Symantec is a global leader in infrastructure software, enabling businesses and consumers to have confidence in a connected world. The company helps customers protect their infrastructure, information, and interactions by delivering software and services that address risks to security, availability, compliance, and performance. Headquartered in Cupertino, Calif., Symantec has operations in 40 countries. More information is available at www.symantec.com.

For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 (800) 745 6054.

Symantec Corporation
World Headquarters
20330 Stevens Creek Boulevard
Cupertino, CA 95014 USA
+1 (408) 517 8000
1 (800) 721 3934
www.symantec.com

03/07 12066477