Reducing the Effective Entropy of GS Cookies

3/2007

skape
mmiller@hick.org

Contents

Chapter 1

Foreword

Abstract: This paper describes a technique that can be used to reduce the effective entropy in a given GS cookie by roughly 15 bits. This reduction is made possible because GS uses a number of weak entropy sources that can, with varying degrees of accuracy, be calculated by an attacker. It is important to note, however, that the ability to calculate the values of these sources for an arbitrary cookie currently relies on an attacker having local access to the machine, such as through the local console or through terminal services. This effectively limits the use of this technique to stack-based local privilege escalation vulnerabilities. In addition to the general entropy reduction technique, this paper discusses the amount of effective entropy that exists in services that automatically start during system boot. It is hypothesized that these services may have more predictable states of entropy due to the relative consistency of the boot process. While the techniques described in this paper do not illustrate a complete break of GS, any inherent weakness can have disastrous consequences given that GS is a static, compile-time security solution. It is not possible to simply distribute a patch. Instead, applications must be recompiled to take advantage of any security improvements. In that vein, the paper proposes some solutions that could be applied to address the problems that are outlined.

Thanks: Aaron Portnoy for lending some hardware for sample collection. Johnny Cache and Richard Johnson for discussions and suggestions.

Chapter 2

Introduction

Stack-based buffer overflows are generally regarded as one of the most common and easiest to exploit classes of software vulnerabilities. This prevalence has lead to the implementation of many security solutions that attempt to prevent the exploitation of these vulnerabilities. Some of these solutions include StackGuard[1], ProPolice[2], and Microsoft's `/GS` compiler switch[5]. The shared premise of these solutions involves the placement of a *cookie*, or *canary*, between the buffers stored in a stack frame and the stack frame's return address. The cookie that is placed on the stack is used as a marker to detect if a buffer overflow has occurred prior to allowing a function to return. This simple concept can be very effective at making the exploitation of stack-based buffer overflows unreliable.

The cookie-based approach to detecting stack-based buffer overflows involves three general steps. First, a cookie that will be inserted into a function's stack frame must be generated. The approaches taken to generate cookies vary quite substantially, some having more implications than others. Once a cookie has been generated, it must be pushed onto the stack in the context of a function's prologue at execution time. This ensures that the cookie is placed before the return address (and perhaps other values) on the stack. Finally, a check must be added to a function's epilogue to make sure that the cookie that was stored in the stack frame is the value that it was initialized to in the function prologue. If an overflow of a stack-based buffer occurs, then it's likely that it will have overwritten the cookie stored after the buffer. When a mismatch is detected, steps can be taken to securely terminate the process in a way that will prevent exploitation.

The security of a cookie-based solution hinges on the fact that an attacker doesn't know, or is unable to generate, the cookie that is stored in a stack frame. Since it's impossible to guarantee in all situations that an attacker won't

be able to generate the bytes that compose the value of a cookie, it really all boils down to the cookie being kept secret. If the cookie is not kept secret, then the presence of a cookie will provide no protection when it comes to exploiting a stack-based buffer overflow vulnerability. Additionally, if an attacker can trigger an exploitable condition before the cookie is checked, then it stands that the cookie will provide no protection. One example of this might include overwriting a function pointer on the stack that is called prior to returning from the function.

While the StackGuard and ProPolice implementations are interesting and useful, the author feels that no implementation is more critical than the one provided by Microsoft. The reason for this is the simple fact that the vast majority of all desktops, and a non-trivial number of servers, run applications compiled with Microsoft's Visual C compiler. Any one weakness found in the Microsoft's implementation could mean that a large number of applications are no longer protected against stack-based buffer overflows. In fact, there has been previous research that has pointed out flaws or limitations in Microsoft's implementation. For example, David Litchfield pointed out that even though stack cookies are present, it may still be possible to overwrite exception registration records on the stack which may be called before the function actually returns. This discovery was one of the reasons that Microsoft later introduced SafeSEH (which had its own set of issues)[6]. Similarly, Chris Ren et al from Cigital pointed out the potential implications of a function pointer being used in the path of the error handler for the case of a GS cookie mismatch occurring[9]. While not directly related to a particular flaw or limitation in GS, eEye has described some of the problems that come when secrets get leaked[3].

Even though these issues and limitations have existed, Microsoft's GS implementation at the time of this writing is considered by most to be secure. While this paper will not present a complete break of Microsoft's GS implementation, it will describe certain quirks and scenarios that may make it possible to reduce the amount of effective entropy that exists in the cookies that are generated. As with cryptography, any reduction of the entropy that exists in the GS cookie effectively makes it so there are fewer unknown portions of the cookie. This makes the cookie easier to guess by reducing the total number of possibilities. Beyond this, it is expected that additional research may find ways to further reduce the amount of entropy beyond that described in this document. One critical point that must be made is that since the current GS implementation is statically linked when binaries are compiled, any flaw that is found in the implementation will require a recompilation of all binaries affected by it. To help limit the scope, only the 32-bit version of GS will be analyzed, though it is thought that similar attacks may exist on the 64-bit version as well.

The structure of this paper is as follows. In chapter 3, a brief description of the Microsoft's current GS implementation will be given. Chapter 4 will describe some techniques that may be used to attack this implementation. Chapter 5 will provide experimental results from using the attacks that are described in chapter 4. Chapter 6 will discuss steps that could be taken to improve the

current GS implementation. Finally, chapter 7 will discuss some areas where future work could be applied to further improve on the techniques described in this document.

Chapter 3

Implementation

As was mentioned in the introduction, security solutions that are designed to protect against stack-based buffer overflows through the use of cookies tend to involve three distinct steps: cookie generation, prologue modifications, and epilogue modifications. Microsoft's GS implementation is no different. This chapter will describe each of these three steps independent of one another to paint a picture for how GS operates.

3.1 Cookie Generation

Microsoft chose to have the GS implementation generate an image file-specific cookie. This means that each image file (executable or DLL) will have their own unique cookie. When used in conjunction with a stack frame, a function will insert its image file-specific cookie into the stack frame. This will be covered in more detail in the next section. The actual approach taken to generate an image file's cookie lives in a compiler inserted routine called `__security_init_cookie`. This routine is placed prior to the call to the image file's actual entry point routine and therefore is one of the first things executed. By placing it at this point, all of the image file's code will be protected by the GS cookie.

The guts of the `__security_init_cookie` routine are actually the most critical part to understand. At a high-level, this routine will take an XOR'd combination of the current system time, process identifier, thread identifier, tick count, and performance counter. The end result of XOR'ing these values together is what ends up being the image file's security cookie. To understand how this actually works in more detail, consider the following disassembly from an application compiled with version 14.00.50727.42 of Microsoft's compiler. Going straight to the disassembly is the best way to concretely understand the implementation,

especially if one is in search of weaknesses.

Like all functions, the `__security_init_cookie` function starts with a prologue. It allocates storage for some local variables and initializes some of them to zero. It also initializes some registers, specifically `edi` and `ebx` which will be used later on.

```
.text:00403D58     push ebp
.text:00403D59     mov ebp, esp
.text:00403D5B     sub esp, 10h
.text:00403D5E     mov eax, __security_cookie
.text:00403D63     and [ebp+SystemTimeAsFileTime.dwLowDateTime], 0
.text:00403D67     and [ebp+SystemTimeAsFileTime.dwHighDateTime], 0
.text:00403D6B     push ebx
.text:00403D6C     push edi
.text:00403D6D     mov edi, 0BB40E64Eh
.text:00403D72     cmp eax, edi
.text:00403D74     mov ebx, 0FFFF0000h
```

As part of the end of the code above, a comparison between the current security cookie and a constant `0xbb40e64e` is made. Before `__security_init_cookie` is called, the global `__security_cookie` is initialized to `0xbb40e64e`. The constant comparison is used to see if the GS cookie has already been initialized. If the current cookie is equal to the constant, or the high order two bytes of the current cookie are zero, then a new cookie is generated. Otherwise, the complement of the current cookie is calculated and cookie generation is skipped.

```
.text:00403D79     jz  short loc_403D88
.text:00403D7B     test eax, ebx
.text:00403D7D     jz  short loc_403D88
.text:00403D7F     not eax
.text:00403D81     mov __security_cookie_complement, eax
.text:00403D86     jmp short loc_403DE8
```

To generate a new cookie, the function starts by querying the current system time using `GetSystemTimeAsFileTime`. The system time as represented by Windows is a 64-bit integer that measures the system time down to a granularity of 100 nanoseconds. The high order 32-bit integer and the low order 32-bit integer are XOR'd together to produce the first component of the cookie. Following that, the current process identifier is queried using `GetCurrentProcessId` and then XOR'd as the second component of the cookie. The current thread identifier is then queried using `GetCurrentThreadId` and then XOR'd as the third component of the cookie. The current tick count is queried using `GetTickCount` and then XOR'd as the fourth component of the cookie. Finally, the current performance counter value is queried using `QueryPerformanceCounter`. Like system time, this value is also a 64-bit integer, and its high order 32-bit integer and low order 32-bit integer are XOR'd as the fifth component of the cookie. Once these XOR operations have completed, a comparison is made between the

newly generated cookie value and the constant `0xbb40e64e`. If the new cookie is not equal to the constant value, then a second check is made to make sure that the high order two bytes of the cookie are non-zero. If they are zero, then a 10 bit left shift of the cookie is performed in order to seed the high order bytes.

```
.text:00403D89     lea eax, [ebp+SystemTimeAsFileTime]
.text:00403D8C     push eax
.text:00403D8D     call ds:__imp__GetSystemTimeAsFileTime@4
.text:00403D93     mov esi, [ebp+SystemTimeAsFileTime.dwHighDateTime]
.text:00403D96     xor esi, [ebp+SystemTimeAsFileTime.dwLowDateTime]
.text:00403D99     call ds:__imp__GetCurrentProcessId@0
.text:00403D9F     xor esi, eax
.text:00403DA1     call ds:__imp__GetCurrentThreadId@0
.text:00403DA7     xor esi, eax
.text:00403DA9     call ds:__imp__GetTickCount@0
.text:00403DAF     xor esi, eax
.text:00403DB1     lea eax, [ebp+PerformanceCount]
.text:00403DB4     push eax
.text:00403DB5     call ds:__imp__QueryPerformanceCounter@4
.text:00403DBB     mov eax, dword ptr [ebp+PerformanceCount+4]
.text:00403DBE     xor eax, dword ptr [ebp+PerformanceCount]
.text:00403DC1     xor esi, eax
.text:00403DC3     cmp esi, edi
.text:00403DC5     jnz short loc_403DCE
...
.text:00403DCE loc_403DCE:
.text:00403DCE     test esi, ebx
.text:00403DD0     jnz short loc_403DD9
.text:00403DD2     mov eax, esi
.text:00403DD4     shl eax, 10h
.text:00403DD7     or  esi, eax
```

Finally, when a valid cookie is generated, it's stored in the image file's `__security_cookie`. The bit-wise complement of the cookie is also stored in `__security_cookie_complement`. The reason for the existence of the complement will be described later.

```
.text:00403DD9     mov __security_cookie, esi
.text:00403DDF     not esi
.text:00403DE1     mov __security_cookie_complement, esi
.text:00403DE7     pop esi
.text:00403DE8     pop edi
.text:00403DE9     pop ebx
.text:00403DEA     leave
.text:00403DEB     retn
```

In simpler terms, the meat of the cookie generation can basically be summarized through the following pseudo code:

```
Cookie  = SystemTimeHigh
Cookie ^= SystemTimeLow
Cookie ^= ProcessId
```

```
Cookie ^= ThreadId
Cookie ^= TickCount
Cookie ^= PerformanceCounterHigh
Cookie ^= PerformanceCounterLow
```

3.2 Prologue Modifications

In order to make use of the generated cookie, functions must be modified to insert it into the stack frame at the time that they are called. This does add some overhead to the call time associated with a function, but its overall effect is linear with respect to a single invocation. The actual modifications that are made to a function's prologue typically involve just three instructions. The cookie that was generated for the image file is XOR'd with the current value of the frame pointer. This value is then placed in the current stack frame at a precisely chosen location by the compiler.

```
.text:0040214B     mov eax, __security_cookie
.text:00402150     xor eax, ebp
.text:00402152     mov [ebp+2A8h+var_4], eax
```

It should be noted that Microsoft has taken great care to refine the way a stack frame is laid out in the presence of GS. Locally defined pointers, including function pointers, are placed before statically sized buffers in the stack frame. Additionally, dangerous input parameters passed to the function, such as pointers or structures that contain pointers, will have local copies made that are positioned before statically sized local buffers. The local copies of these parameters are used instead of those originally passed to the function. These two changes go a long way toward helping to prevent other scenarios in which stack-based buffer overflows might be exploited.

3.3 Epilogue Modifications

When a function returns, it must check to make sure that the cookie that was stored on the stack has not been tampered with. To accomplish this, the compiler inserts the following instructions into a function's prologue:

```
.text:00402223     mov ecx, [ebp+2A8h+var_4]
.text:00402229     xor ecx, ebp
.text:0040222B     pop esi
.text:0040222C     call __security_check_cookie
```

The value of the cookie that was stored on the stack is moved into `ecx` and then XOR'd with the current frame pointer to get it back to the expected value. Following that, a call is made to `__security_check_cookie` where the stack frame's

cookie value is passed in the `ecx` register. The `__security_check_cookie` routine is very short and sweet. The passed in cookie value is compared with the image file's global cookie. If they don't match, `__report_gsfailure` is called and the process eventually terminates. This is what one would expect in the case of a buffer overflow scenario. However, if they do match, the routine simply returns, allowing the calling function to proceed with execution and cleanup.

```
.text:0040634B     cmp ecx, __security_cookie
.text:00406351     jnz short loc_406355
.text:00406353     rep retn
.text:00406355 loc_406355:
.text:00406355     jmp __report_gsfailure
```

Chapter 4

Attacking GS

At the time of this writing, all publicly disclosed attacks against GS that the author is aware of have relied on getting control of execution before the cookie is checked or by finding some way to leak the value of the cookie back to the attacker. Both of these styles of attack are of great interest and value, but the focus of this paper will be on a different method of attacking GS. Specifically, this chapter will outline techniques that may be used to make it easier to guess the value an image file's GS cookie. Two techniques will be described. The first technique will describe methods for calculating the values that were used as entropy sources when the cookie was generated. These calculations are possible in situations where an attacker has local access to the machine, such as through the console or through terminal services. The second technique describes the general concept of predictable ranges of some values that are used in the context of boot start services, such as `lsass.exe`. This predictability may make the guessing of a GS cookie more feasible in both local and remote scenarios.

4.1 Calculating Entropy Sources

The sources used to generate the GS cookie for a given image file are constant and well-known. They include the current system time, process identifier, thread identifier, tick count, and performance counter. In light of that fact, it only makes sense to investigate the amount of effective entropy each source adds to the cookie. Since it's a requirement that the cookie produced be secret, the ability to guess a value used in the generation of the cookie will allow it to be canceled out of the equation. This is true due to the simple fact that each of the values used to generate the cookie is XOR'd with each other value (XOR is a commutative operation). The ability to guess multiple values can make it possible to seriously impact the overall integrity of the cookie.

While the sources used in the generation of the cookie have long been regarded as satisfactory, the author has found that the majority of the sources actually contribute little to no value toward the overall entropy of the cookie. However, this is currently only true if an attacker has local access to the machine. Being able to know a GS cookie that was used in a privileged process would make it possible to exploit a local privilege escalation vulnerability, for example. There may be some circumstances where the techniques described in this section could be applied remotely, but for the purpose of this document, only the local scenario will be considered. The following subsections will outline methods that can be used to calculate or deterministically find the specific values that were used when a cookie was being generated in a particular process context. As a result of this analysis, it's become clear that the only particular variable source of true entropy for the GS cookie is the low 17 bits of the performance counter. All other sources can be reliably calculated, with some margin of error.

For the following subsections, a modified executable named `vulnapp.exe` was used to extract the information that was used at the time that a process executable's GS cookie was generated. In particular, `__security_init_cookie` was modified to jump into a function that saves the information used to generate the cookie. The implementation of this function is shown below for those who are curious:

```
//
// The FramePointer is the value of EBP in the context of the
// __security_init_cookie routine.  The cookie is the actual,
// resultant cookie value.  GSContext is a global array.
//
VOID DumpInformation(
   PULONG FramePointer,
   ULONG  Cookie)
{
   GSContext[0] = FramePointer[-3];
   GSContext[1] = FramePointer[-4];
   GSContext[2] = FramePointer[-1];
   GSContext[3] = FramePointer[-2];
   GSContext[4] = GetCurrentProcessId();
   GSContext[5] = GetCurrentThreadId();
   GSContext[6] = GetTickCount();
   GSContext[7] = Cookie;
}
```

4.1.1 System Time

System time is a value that one might regard as challenging to recover. After all, it seems impossible to get the 100 nanosecond granularity of the system time that was retrieved when a cookie was being generated. Quite the contrary, actually. There are a few key points that go into being able to recover the system time. First, it's a fact that even though the system time measures

granularity in terms of 100 nanosecond intervals, it's really only updated every 15.625 milliseconds (or 10.1 milliseconds for more modern CPUs). To many, 15.625 may seem like an odd number, but for those familiar with the Windows thread scheduler, it can be recognized as the period of the timer interrupt. For that reason, the current system time is only updated as a result of the timer interrupt firing. This fact means that the alignment of the system time that is used when a cookie is generated is known.

Of more interest, though, is the relationship between the system time value and the creation time value associated with a process or its initial thread. Since the minimum granularity of the system time is 15.6 or 10.1 milliseconds, it follows that the granularity of the thread creation time will be the same. In terms of modern CPUs, 15.6 milliseconds is an eternity and is plenty long for the processor to execute all instructions from the creation of the thread to the generation of the security cookie. This fact means that it's possible to assume that the creation time of a process or thread is the same as the system time that was used when the cookie was generated. This assumption doesn't always work, though, and there are indeed cases where the creation time will not equal the system time that was used. These situations are usually a result of the thread that creates the cookie not being immediately scheduled.

Even if this is the case, it would be necessary to be able to obtain the creation time of an arbitrary process or thread. On the surface, this would seem impossible because task manager prevents a non-privileged user from getting the start time of a privileged process, as figure 4.1 shows.

This is all a deception, though, because there does exist functionality that is exposed to non-privileged users that can be used to get this information. One way of getting it is through the use of the native API routine `NtQuerySystemInformation`. In this case, the `SystemProcessesAndThreadsInformation` system information class is used to query information about all of the running processes on the system. This information includes the process name, process creation time, and the creation time for each thread in each process. While this information class has been removed in Windows Vista, there are still potential ways of obtaining the creation time information. For example, an attacker could simply crash the vulnerable service once (assuming it's not a critical service) and then wait for it to respawn. Once it respawns, the creation time can be inferred based on the restart delay of the service. Granted, service restarts are limited to three times per day in Vista, but crashing it once should cause no major issues.

Using `NtQuerySystemInformation`, it's possible to collect some data that can be used to determine the likelihood that the creation time of a thread will be equal to the system time that was used when a GS cookie was generated. To test this, the author used the modified `vulnapp.exe` executable to extract the system time at the time that the cookie was generated. Following that, a separate program was used to collect the creation time information of the process in question using the native API. The initial thread's creation time was

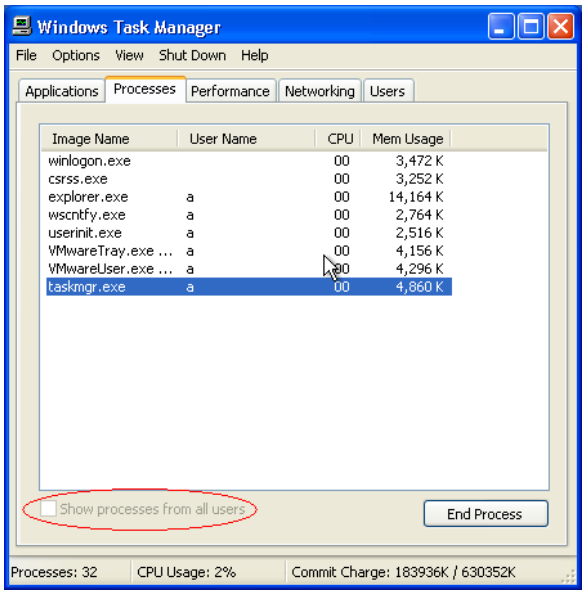

Figure 4.1: Task manager disabling access to privileged processes

then compared with the system time to see if they were equal. Figure 4.2 shows these differences for a sample of 742 cookies taken from a single machine. In most cases, system time and creation time were equal.

Obviously, the data set describing differences is only relevant to a particular system load. If there are many threads waiting to run during the time that a process is executed, then it is unlikely that the system time will equal the process creation time. In a desktop environment, it's probably safe to assume that the thread will run immediately, but more conclusive evidence may be necessary.

Given these facts, it is apparent that the complete 64-bit system time value can be recovered more often than not with a great degree of accuracy just by simply assuming that thread creation time is the same as the system time value.

4.1.2 Process and Thread Identifier

The process and thread identifier are arguably the worst sources of entropy for the GS cookie, at least in the context of a local attack. The two high order bytes of the process and thread identifiers are almost always zero. This means they have absolutely no effect on the high order entropy. Additionally, the process and thread identifier can be determined with 100 percent accuracy in a local context using the same API described in the previous section on getting

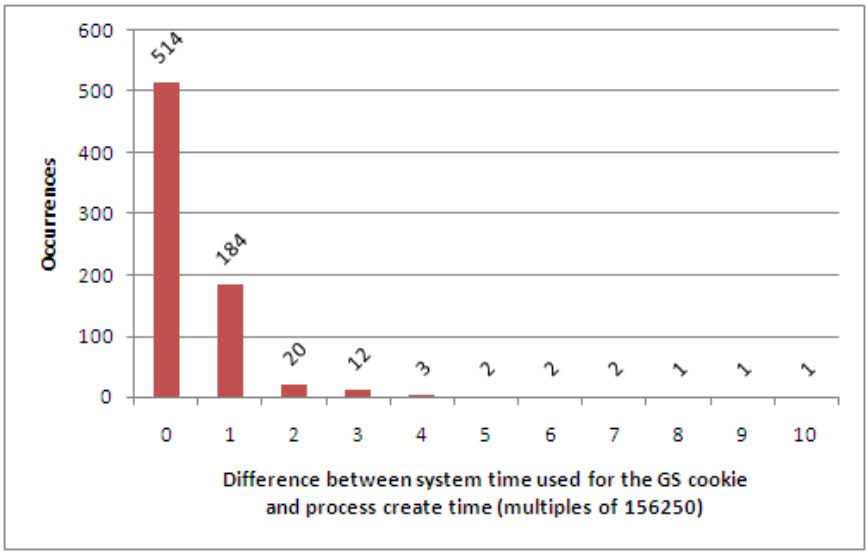

Figure 4.2: Difference between system time and create time

the system time. This involves making use of the `NtQuerySystemInformation` native API with the `SystemProcessesAndThreadsInformation` system information class to get the process identifier and thread identifier associated with a given process executable.

The end result, obviously, is that the process and thread identifier can be determined with great accuracy. The one exception to this rule would be Windows Vista, but, as was mentioned before, alternative methods of obtaining the process and thread identifier may exist.

4.1.3 Tick Count

The tick count is, for all intents and purposes, simply another measure of time. When the `GetTickCount` API routine is called, the number of ticks is multiplied by the tick count multiplier. This multiplication effectively translates the number of ticks to the number of milliseconds that the system has been up. If one can safely assume that the that the system time used to generate the cookie was the same as the thread creation time, then the tick count at the time that the cookie was generated can simply be calculated using the thread creation time. The creation time isn't enough, though. Since the `GetTickCount` value measures the number of milliseconds that have occurred since boot, the actual uptime of the system has to be determined.

To determine the system uptime, a non-privileged user can again make use of the `NtQuerySystemInformation` native API, this time with the `SystemTimeOfDayInformation` system information class. This query returns the time that the system was booted as a 64-bit integer measured in 100 nanosecond intervals, just like the thread creation time. To calculate the system uptime in milliseconds, it's as simple as subtracting the boot time from the creation time and then dividing by 10000 to convert from 100 nanosecond intervals to 1 millisecond intervals:

$$EstTickCount = (CreationTime - BootTime) \div 10000$$

Some experimentation shows that this calculation is pretty accurate, but some quantity is lost in translation. From what the author has observed, a constant scaling factor of `0x4e`, or 78 milliseconds, needs to be added to the result of this calculation. The source of this constant is as of yet unknown, but it appears to be a required constant. This results in the actual equation being:

$$EstTickCount = [(CreationTime - BootTime) \div 10000] + 78$$

The end result is that the tick count can be calculated with a great degree of accuracy. If the system time calculation is off, then that will directly affect the calculation of the tick count.

4.1.4 Performance Counter

Of the four entropy sources discussed so far, the performance counter is the only one that really presents a challenge. The purpose of the performance counter is to describe the total number of cycles that have executed. On the outside, the performance counter would seem impossible to reliably determine. After all, how could one possibly determine the precise number of cycles that had occurred as a cookie was being generated? The answer, of course, comes down to the fact that the performance counter itself is, for all intents and purposes, just another measure of time. Windows provides two interesting user-mode APIs that deal with the performance counter. The first, `QueryPerformanceCounter`, is used to ask the kernel to read the current value of the performance counter[8]. The result of this query is stored in the 64-bit output parameter that the caller provides. The second API is `QueryPerformanceFrequency`. This routine is interesting because it returns a value that describes the amount that the performance counter will change in one second[7]. Documentation indicates that the frequency cannot change while the system is booted.

Using the existing knowledge about the uptime of the system and the calculation that can be performed to convert between the performance counter value and seconds, it is possible to fairly accurately guess what the performance counter was at the time that the cookie was generated. Granted, this method is more fuzzy than the previously described methods, as experimental results have shown a large degree of fluctuation in the lower 17 bits. Those results will be discussed

in more detail in chapter 5. The actual equation that can be used to generate the estimated performance counter is to take the uptime, as measured in 100 nanosecond intervals, and multiply it by the performance frequency divided by 10000000, which converts the frequency from a measure of 1 second to 100 nanosecond:

$$EstPerfCounter = UpTime \times (PerfFreq \div 10000000)$$

In a fashion similar to tick count, a constant scaling factor of `-165000` was determined through experimentation. This seems to produce more accurate results in some of the 24 low bits. Based on this calculation, it's possible to accurately determine the entire 32-bit high order integer and the first 15 bits of the 32-bit low order integer. Of course, if the system time estimate is wrong, then that directly effects this calculation.

4.1.5 Frame Pointer

While the frame pointer does not influence an image file's global cookie, it does influence a stack frame's version of the cookie. For that reason, the frame pointer must be considered as an overall contributor to the effective entropy of the cookie. With the exception of Windows Vista, the frame pointer should be a deterministic value that could be deduced at the time that a vulnerability is triggered. As such, the frame pointer should be considered a known value for the majority of stack-based buffer overflows. Granted, in multi-threaded applications, it may be more challenging to accurately guess the value of the frame pointer.

In the Windows Vista environment, the compile-time GS implementation gets a boost in security due to the introduction of ASLR. This helps to ensure that the frame pointer is actually an unknown quantity. However, it doesn't introduce equal entropy in all bits. In particular, octet 4, and potentially octet 3, may have predictable values due to the way that the randomization is applied to dynamic memory allocations. In order to prevent fragmentation of the address space, Vista's ASLR implementation attempts to ensure that stack regions are still allocated low in the address space. This has the side effect of ensuring that a non-trivial number of bits in the frame pointer will be predictable. Additionally, while Vista's ASLR implementation makes an effort to shift the lower bits of the stack pointer, there may still be some bits that are always predictable in octet 2.

4.2 Predictability of Entropy Sources in Boot Start Services

A second attack that could be used against GS involves attacking services that start early on when the system is booted. These services may experience more predictable states of entropy due to the fact that the amount of time it takes to boot up and the order in which tasks are performed is fairly, though not entirely, consistent. This insight may make it possible to estimate the value of entropy sources remotely.

To better understand this type of attack, the author collected 742 samples that were taken from a custom service that was set to automatically start during boot on a Windows XP SP2 installation. This service was simply designed to log the state used at the time that the GS cookie was being generated. While a sampling of the GS cookie state applied to `lsass.exe` would have been more ideal, it wasn't worth the headache of having to patch a critical system service. Perhaps the reader may find it interesting to collect this data on their own. From the samples that were taken, the following diagrams show the likelihood of each individual bit being set for each of the different entropy sources.

Overall, there are a number of predictable bits in things like the high 32-bits of both the system time and the performance counter, the process identifier, the thread identifier, and the tick count. The sources that are largely unpredictable are the low 32-bits of the system time and the performance counter. However, if it were possible to come up with a way to discover the boot time (or uptime) of the system remotely, it might be possible to infer a good portion of the low 32-bits of the system time. This would then directly impact the ability to estimate things like the tick count and performance counters.

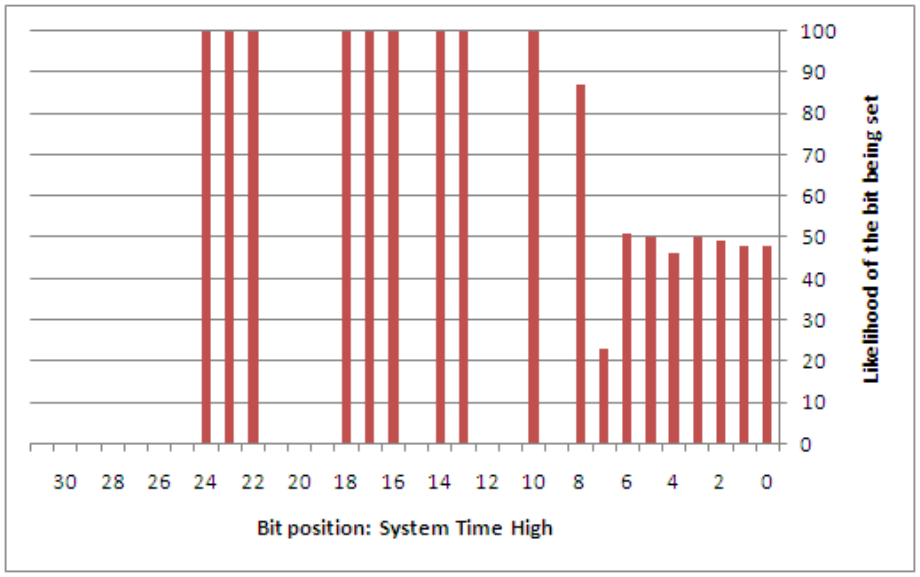

Figure 4.3: Likelihood of a given bit being set in the high 32-bits of System Time for an auto start service

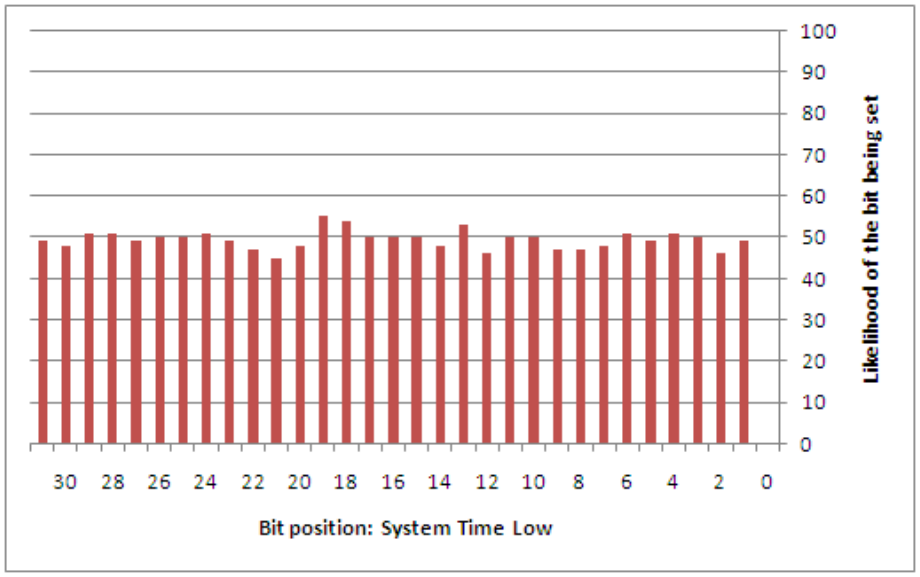

Figure 4.4: Likelihood of a given bit being set in the low 32-bits of System Time for an auto start service

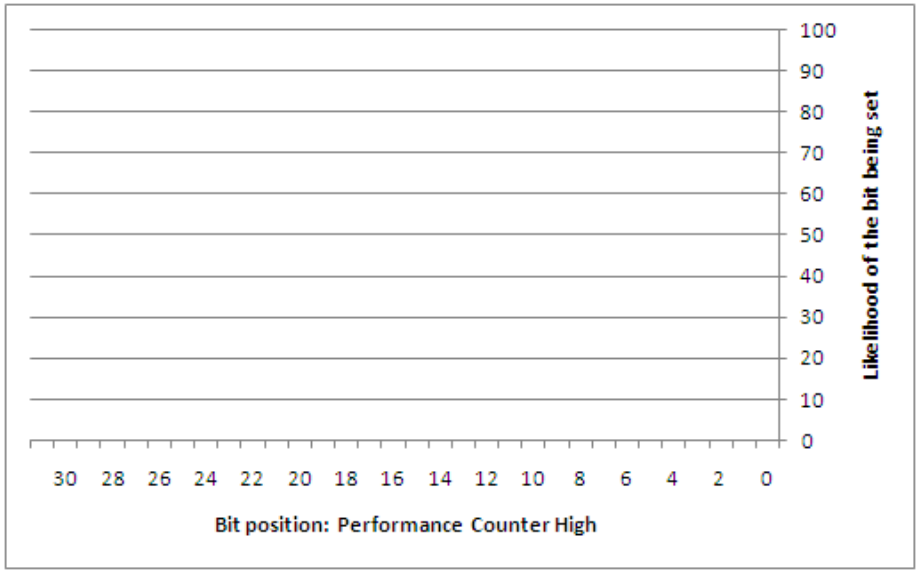

Figure 4.5: Likelihood of a given bit being set in the high 32-bits of the Performance Counter for an auto start service

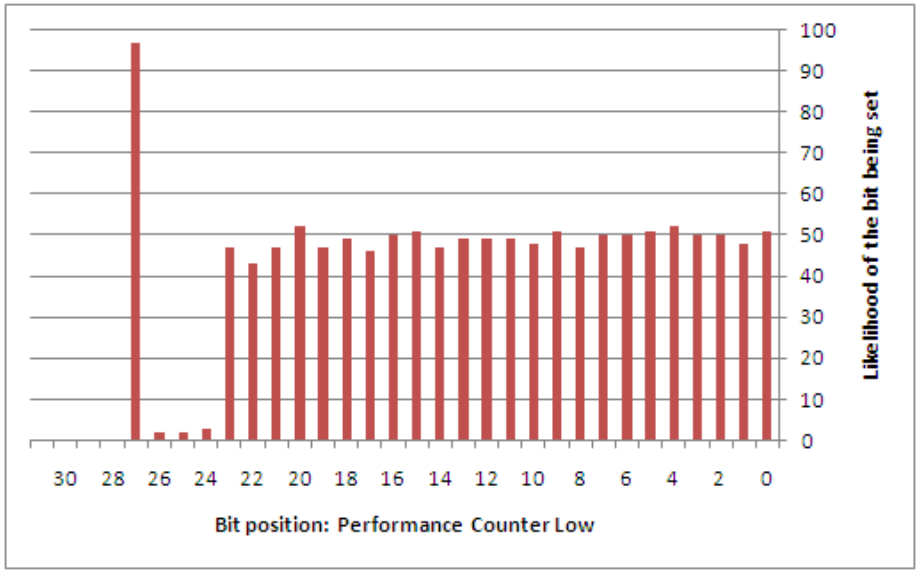

Figure 4.6: Likelihood of a given bit being set in the low 32-bits of the Performance Counter for an auto start service

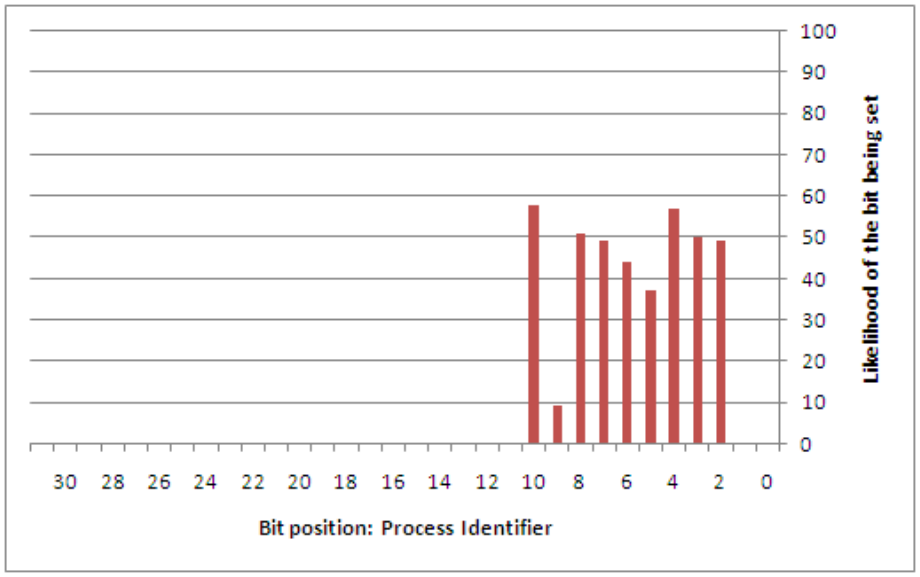

Figure 4.7: Likelihood of a given bit being set in the Process Identifier for an auto start service

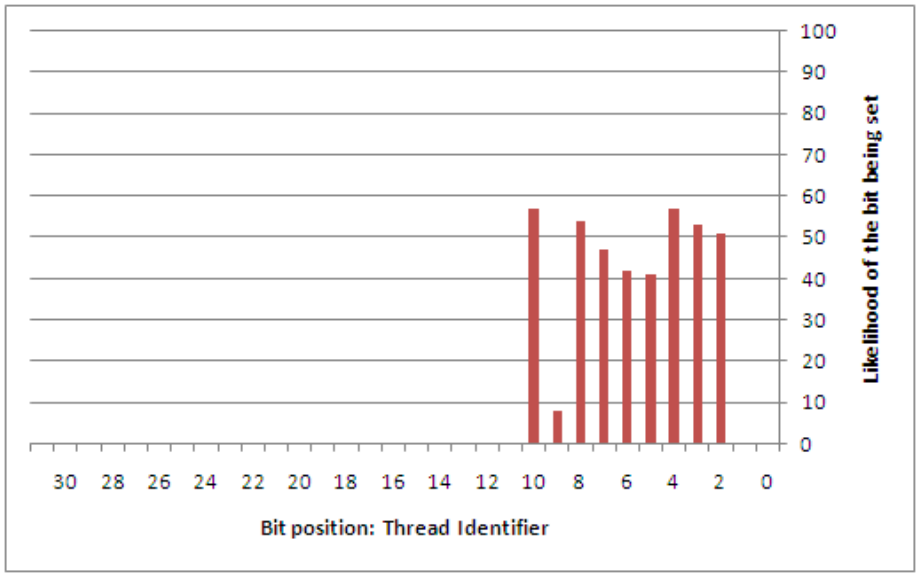

Figure 4.8: Likelihood of a given bit being set in the Thread Identifier for an auto start service

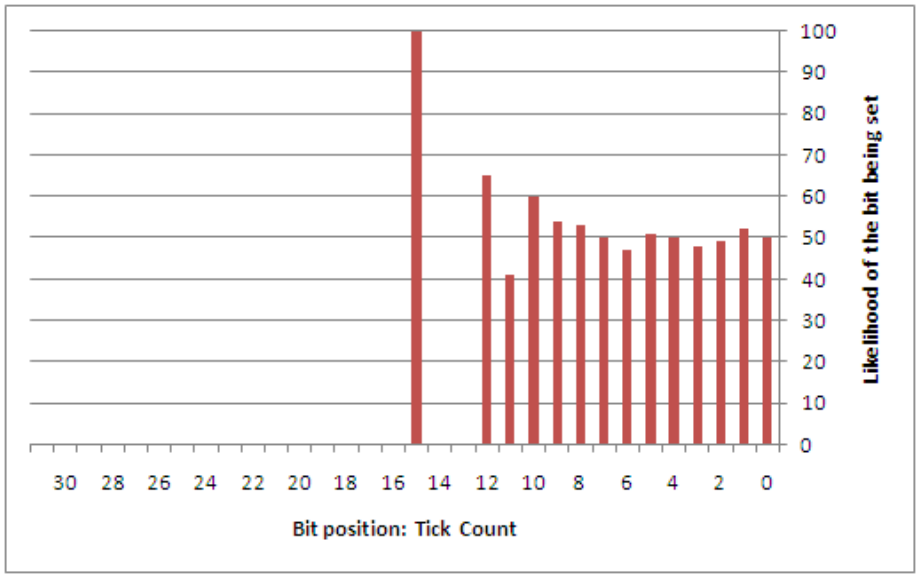

Figure 4.9: Likelihood of a given bit being set in the Tick Count for an auto start service

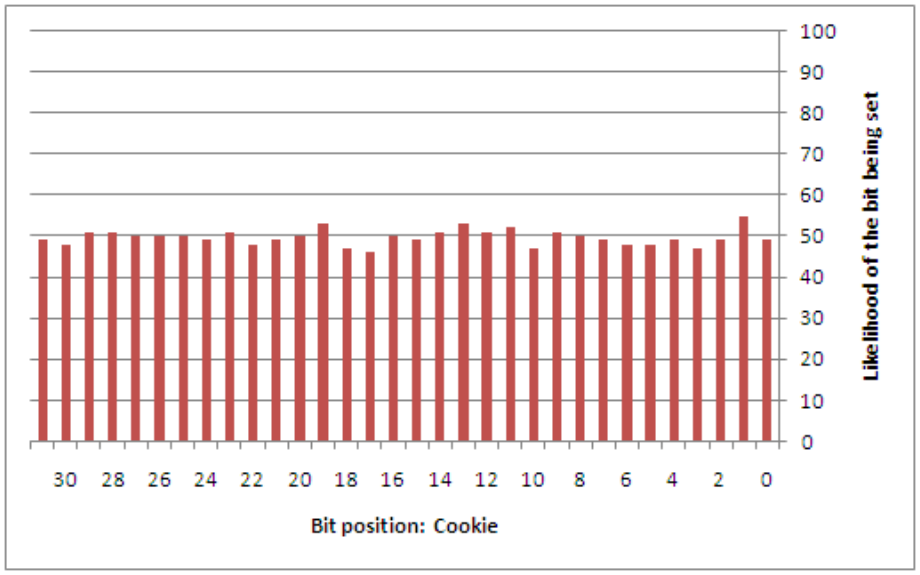

Figure 4.10: Likelihood of a given bit being set in the Cookie for an auto start service

Chapter 5

Experimental Results

This chapter describes some of the initial results that were collected using a utility developed by the author named `gencookie.exe`. This utility attempts to calculate the value of the cookie that was generated for the executable image associated with an arbitrary process, such as `lsass.exe`. While the results of this utility were limited to attempting to calculate the cookie of a process' executable, the techniques described in previous chapters are nonetheless applicable to the cookies generated in the context of dependent DLLs. The results described in this chapter illustrate the tool's ability to accurately obtain specific bits within the different components that compose the cookie, including specific bits of the cookie itself. This helps to paint a picture of the amount of true entropy that is reduced through the techniques described in this document.

The data set that was used to calculate the overall results included 5001 samples which were collected from a single machine. The samples were collected through a few simple steps. First, a program called `vulnapp.exe` that was compiled with `/GS` was modified to have its `__security_init_cookie` routine save information about the cookie that was generated and the values that contributed to its generation. Following that, the `gencookie.exe` utility was launched against the running process in an attempt to calculate `vulnapp.exe`'s GS cookie. A comparison between the expected and actual value of each component was then saved. These steps were repeated 5001 times. The author would be interested in hearing about independent validation of the findings presented in this chapter.

The following sections describe the bit-level predictability of each of the components that are used to generate the GS cookie, including the overall predictability of the bits of the GS cookie itself. The diagrams describe the predictability in terms of the percent of the time that each bit was correctly calculated by `gencookie.exe`. The diagram in figure 5.1 shows with what percentage accuracy each individual component was successfully calculated. For example, the

value used for the low 32-bits of the system time component was successfully determined 77 percent of the time. The low 32-bits of the performance counter and the cookie itself were never calculated exactly. The reason for this discrepancy will be discussed in the following sections.

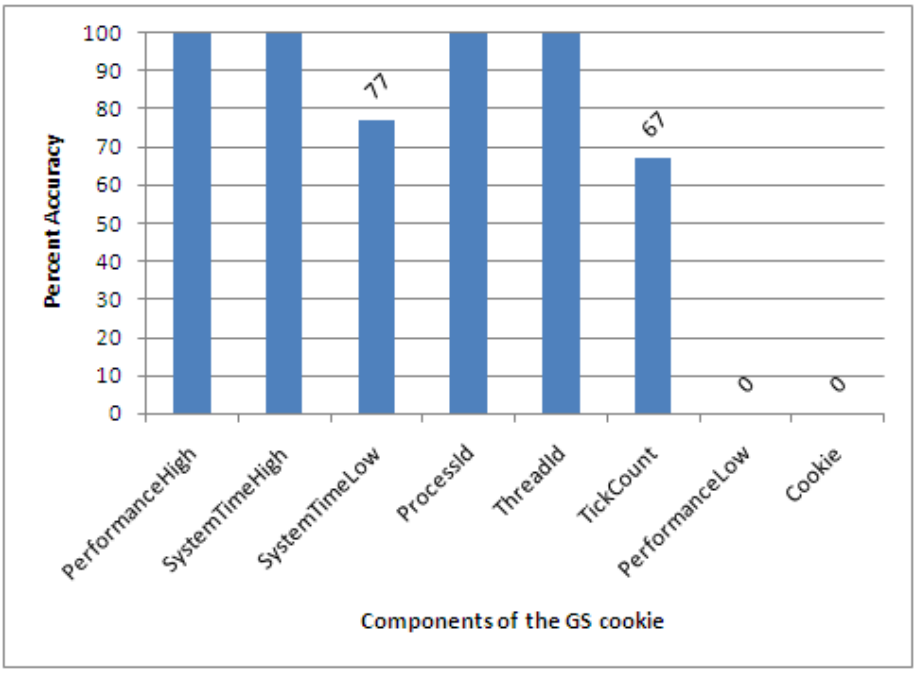

Figure 5.1: Percentage of the time that all bits of individual components were accurately calculated

5.1 System Time

The system time component was highly predictable. The high 32-bit bits of the system time were predicted with 100 percent accuracy. The low 32-bit bits on the other hand were predicted with only 77 percent accuracy (3878 times). The reason for this discrepancy has to do with the thread scheduling scenario described in subsection 4.1.1. Even still, these results indicate that it is likely that the entire system time value can be accurately calculated.

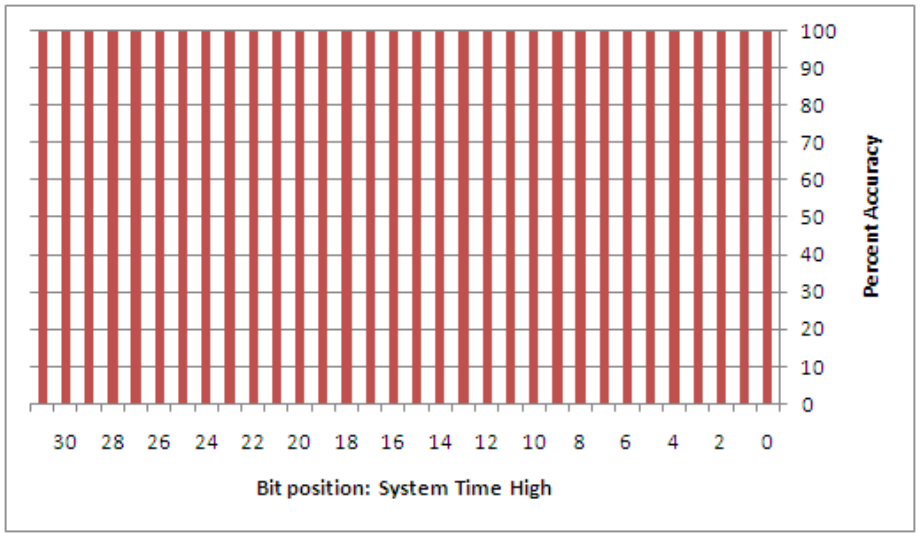

Figure 5.2: Percent accuracy of each bit position for the estimated high 32-bits of the System Time

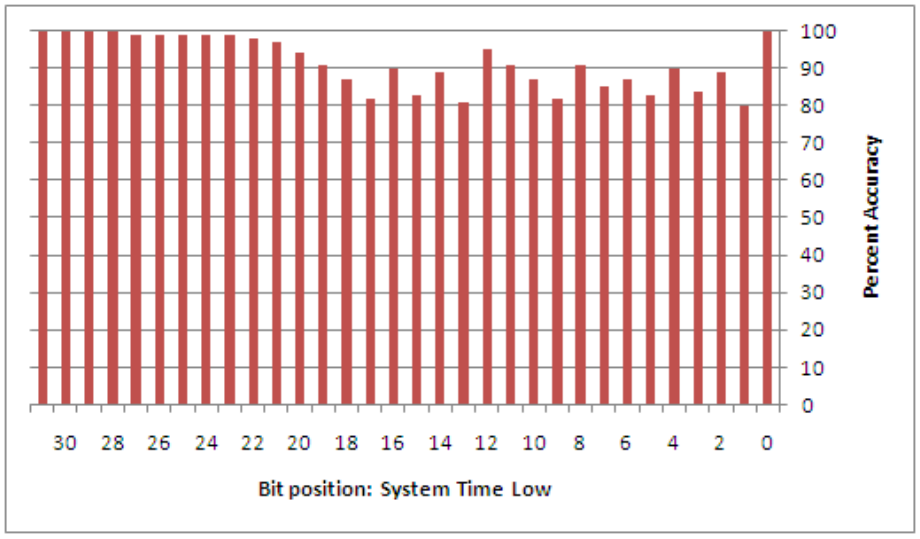

Figure 5.3: Percent accuracy of each bit position for the estimated low 32-bits of the System Time

5.2 Process and Thread Identifier

The process and thread identifier were successfully calculated 100 percent of the time using the approach outlined in section 4.1.2.

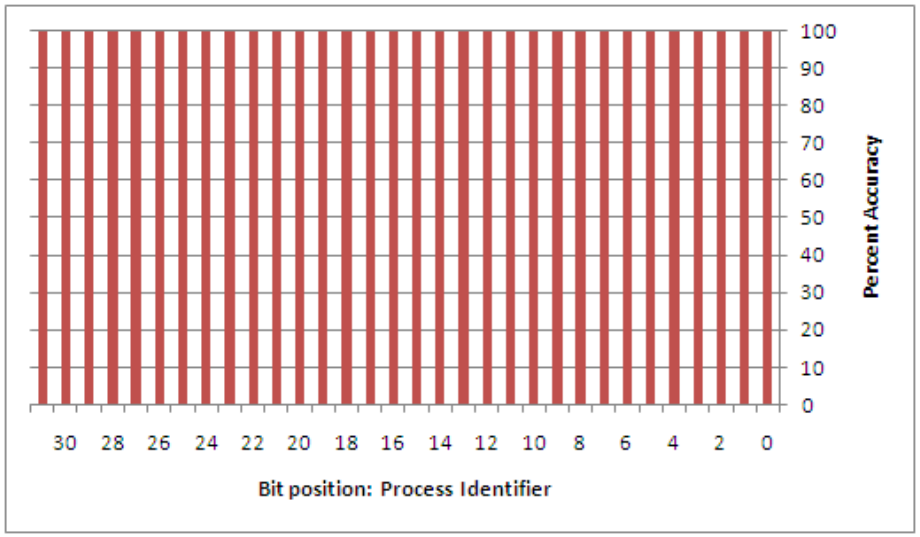

Figure 5.4: Percent accuracy of each bit position for the estimated Process Identifier

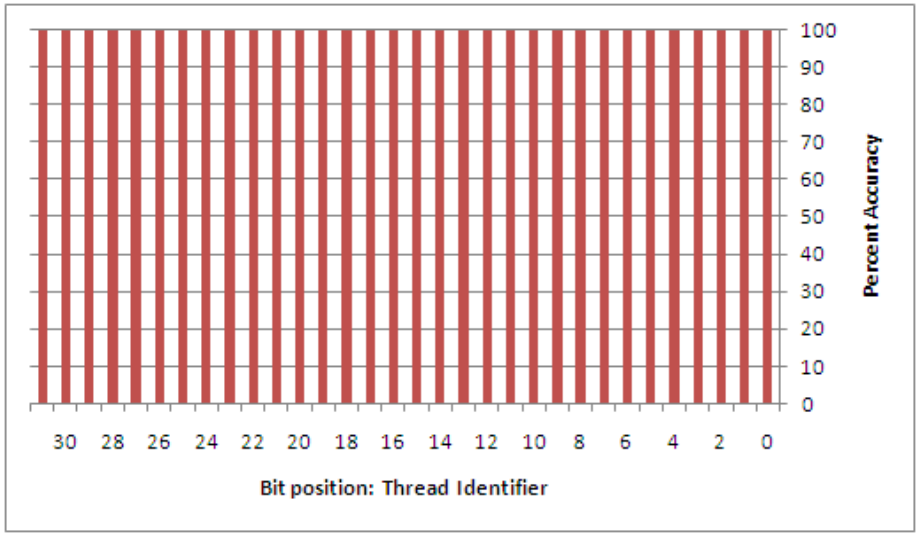

Figure 5.5: Percent accuracy of each bit position for the estimated Thread Identifier

5.3 Tick Count

The tick count was accurately calculated 67 percent of the time (3396 times). The reason for this lower rate of success is due in large part to the fact that the tick count is calculated in relation to the estimated system time value. As such, if an incorrect system time value is determined, the tick count itself will be directly influenced. This should account for at least 23 percent of the inaccuracies judging from how often the system time was inaccurately estimated. The remaining 10 percent of the inaccuracies is as of yet undetermined, but it is most likely related to the an improper interpretation of the constant scaling factor that is applied to the tick count. In any case, it is expected that only a few bits are actually affected in the remaining 10 percent of cases.

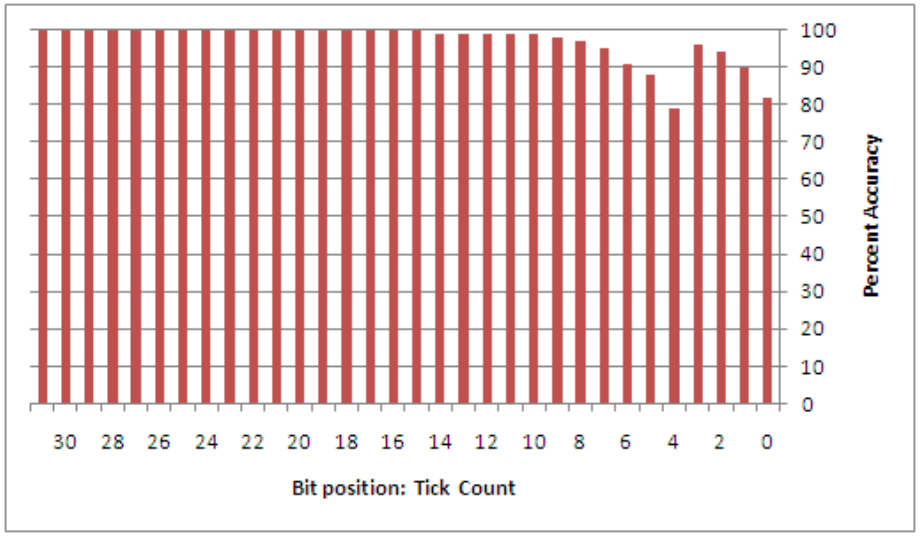

Figure 5.6: Percent accuracy of each bit position for the estimated Tick Count

5.4 Performance Counter

The high 32-bits of the performance counter were successfully estimated 100 percent of the time. The low 32-bits, on the other hand, show the greatest degree of volatility when compared to the other components. The high order 15 bits of the low 32-bits show a bias in terms of accuracy that is not a 50/50 split. The remaining 17 bits were all guessed correctly roughly 50 percent of the time. This makes the low 17 bits the only truly effective source of entropy in the performance counter since there is no bias shown in relation to the estimated versus actual values. Indeed, this is not enough to prove that there aren't observable patterns in the low 17 bits, but it is enough to show that the `gencookie.exe` utility was not effective in estimating them. Figures 5.8 and 5.9 show the percent accuracy for the high and low order 32-bits.

This discrepancy actually requires a more detailed explanation. In reality, the estimates made by the `gencookie.exe` utility are actually not as far off as one might think based on the percent accuracy of each bit as described in the diagrams. Instead, the estimates are, on average, off by only 105,000. This average difference is what leads to the lower 17 bits being so volatile. One thing that's interesting about the difference between the estimated and actual performance counter is that there appears to be a time oriented trend related to how far off the estimates are. The scatter plot diagram in figure 5.7 illustrates the absolute difference between the estimated and actual low 32-bits of the performance counter as taken from the 5001 samples. Due to the way that the samples were taken, it's safe to assume that each sample is roughly equivalent to one second worth of time passing (due to a sleep between sample collection). Further study of this apparent relationship may yield better results in terms of estimating the lower 17 bits of the low 32 bits of the performance counter. This is left for future research.

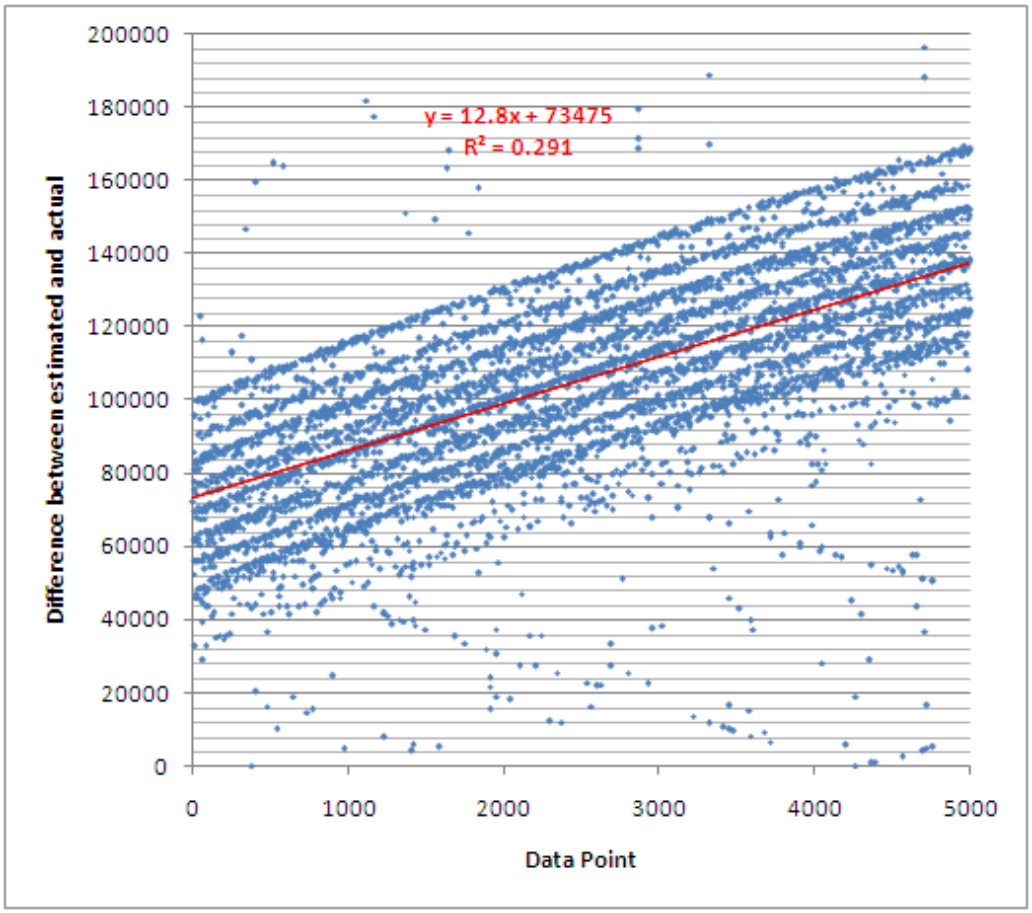

Figure 5.7: Absolute difference between the estimated and actual low 32-bits of the performance counter

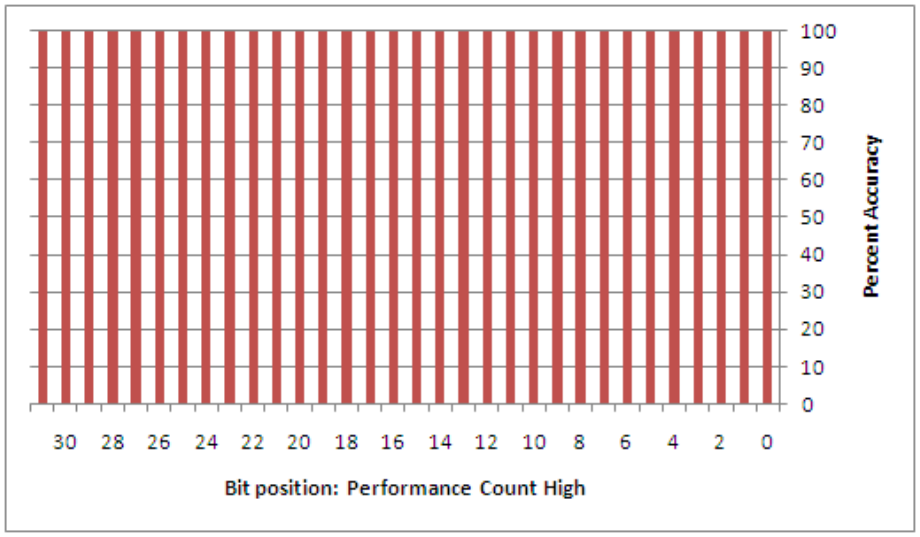

Figure 5.8: Percent accuracy of each bit position for the estimated high 32-bits of the Performance Counter

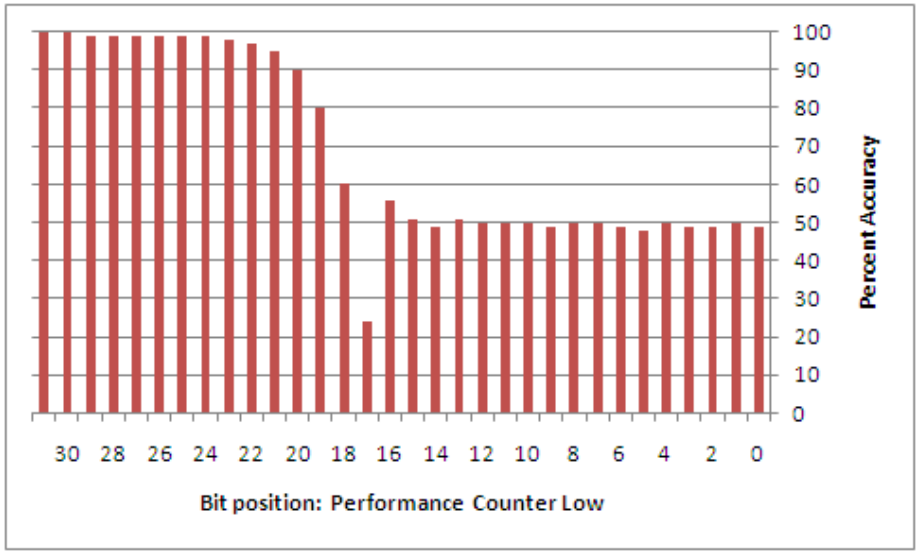

Figure 5.9: Percent accuracy of each bit position for the estimated low 32-bits of the Performance Counter

5.5 Cookie

The cookie itself was never actually guessed during the course of sample collection. The reason for this is tightly linked with the current inability to accurately determine the lower 17 bits of the low 32 bits of the performance counter. Comparing the percent accuracy of the cookie bits with the percent accuracy of the low 32 bits of the performance counter yields a very close match.

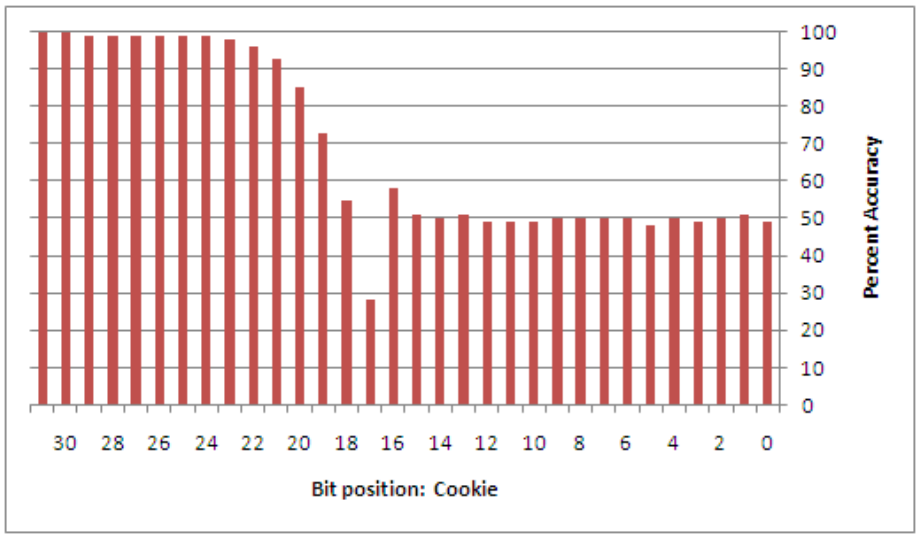

Figure 5.10: Percent accuracy of each bit position for the estimated Cookie

Chapter 6

Improvements

Based on the results described in chapter 5, the author feels that there is plenty of room for improvement in the way that GS cookies are currently generated. It's clear that there is a need to ensure that there are 32 bits of true entropy in the cookie. The following sections outline some potential solutions to the entropy issue described in this document.

6.1 Better Entropy Sources

Perhaps the most obvious solution would be to simply improve the set of entropy sources used to generate the cookie. In particular, the use of sources with greater degrees of entropy, especially in the high order bits, would be of great benefit. The challenge, however, is locating sources that are easy to interact with and require very little overhead. For example, it's not really feasible to have the GS cookie generator rely on the crypto API due to the simple fact that this would introduce a dependency on the crypto API in any application that was compiled with `/GS`. As this document has hopefully shown, it's also a requirement that any additional entropy sources be challenging to estimate externally at a future point in time.

Even though this is a viable solution, the author is not presently aware of any additional entropy sources that would meet all three requirements. For this reason, the author feels that this approach alone is insufficient to solve the problem. If entropy sources are found which meet these requirements, the author would love to hear about them.

6.2 Seeding High Order Bits

A more immediate solution to the problem at hand would involve simply ensuring that the predictable high order bits are seeded with less predictable values. However, additional entropy sources would be required in order to implement this properly. At present, the only major source of entropy found in the GS cookie is the low order bits of the performance counter. It would not be sufficient to simply shift the low order bits of the performance counter into the high order. Doing so would add absolutely no value by itself because it would have no effect on the amount of true entropy in the cookie.

6.3 External Cookie Generation

An alternative solution that could combine the effects of the first two solutions would be to change the GS implementation to generate the cookie external to the binary itself. One of the most dangerous aspects of the GS implementation is that it is statically linked and therefore would require a recompilation of all affected binaries in the event that a weakness is found. This fact alone should be scary. To help address both this problem and the problem of weak entropy sources, it makes sense to consider a more dynamic approach.

One example of a dynamic approach would be to have the GS implementation issue a call into a kernel-mode routine that is responsible for generating GS cookies. One place that this support could be added is in `NtQuerySystemInformation`, though it's likely that a better place may exist. Regardless of the specific routine, this approach would have the benefit of moving the code used to generate the cookie out of the statically linked stub that is inserted by the compiler. If any weakness were to be found in the kernel-mode routine that generates the cookie, Microsoft could issue a patch that would immediately affect all applications compiled to use GS. This would solve some of the concerns relating to the static nature of GS.

Perhaps even better, this approach would grant greater flexibility to the entropy sources that could be used in the generation of the cookie. Since the routine would exist in kernel-mode, it would have the benefit of being able to access additional sources of entropy that may be challenging or clumsy to interact with from user-mode (though the counterpoint could certainly be made as well). The kernel-mode routine could also accumulate entropy over time and feed that back into the cookie, whereas the statically linked implementation has no context with which to accumulate entropy. The accumulation of state can also do more harm than good. It would be disingenuous to not admit that this approach could also have its own set of problems. A poorly implemented version of this solution might make it possible for a user to eliminate all entropy by issuing a non-trivial number of calls to the kernel-mode routine. There may be additional

consequences that have not yet been perceived.

The impact on performance is also a big point of concern for any potential change to the cookie generation path. At a high-level, a transition into kernel-mode would seem concerning in terms of the amount of overhead that might be added. However, it's important to note that the current implementation of GS already transitions into kernel-mode to obtain some of it's information. Specifically, performance counter information is obtained through the system call `NtQueryPerformanceCounter`. Even more, this system call results in an `in` operation on an I/O port that is used to query the current performance counter.

Another important consideration is backward compatibility. If Microsoft were to implement this solution, it would be necessary for applications compiled with the new support to still be able benefit from GS on older platforms that don't support the new kernel interface. To allow for backward compatibility, Microsoft could implement a combination of all three solutions, whereby better entropy sources and seeding of high order bits are used as a fallback in the event that the kernel-mode interface is not present.

As it turns out, Microsoft does indeed have a mechanism that could allow them to create a patch that would affect the majority of the binaries compiled with recent versions of GS. This functionality is provided by exposing the address of an image file's security cookie in its the load config data directory. When the dynamic loader (ntdll) loads an image file, it checks to see if the security cookie address in the load config data directory is non-NULL. If it's not NULL, the loader proceeds to store the process-wide GS cookie in the module-specific's GS cookie location. In this way, the `__security_init_cookie` routine that's called by the image file's entry point effectively becomes a no-operation because the cookie will have already been initialized. This manner of setting the GS cookie for image files provides Microsoft with much more flexibility. Rather than having to update all binaries compiled with GS, Microsoft can simply update a single binary (ntdll.dll) if improvements need to be made to the cookie generation algorithm. The following output shows a sample of `dumpbin /loadconfig` on `kernel32.dll`:

```
Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file c:\windows\system32\kernel32.dll

File Type: DLL

  Section contains the following load config:

            00000048 size
                   0 time date stamp
...
            7C8836CC Security Cookie
```

Chapter 7

Future Work

There is still additional work that can be done to further refine the techniques described in this document. This chapter outlines some of the major items that could be followed up on.

7.1 Improving Performance Counter Estimates

One area in particular that the author feels could benefit from further research has to do with refining the technique used to calculate the performance counter. As was illustrated in figure 5.7, a more thorough analysis of the apparent association between time and the lower 17 bits of the performance counter is necessary. This analysis would directly affect the ability to recover more cookie state information, since the entropy of the lower 17 bits of the performance counter is one of the only things standing in the way of obtaining the entire cookie.

7.2 Remote Attacks

The ability to apply the techniques described in this document in a remote scenario would obviously increase the severity of the problem. In order to do this, an attacker would need the ability to either infer or be able to calculate some of the key elements that are used in the generation of a cookie. This would rely on being able to determine things like the process creation time, the process and thread identifier, and the system uptime. With these values, it should be possible to predict the state of the cookie with similar degrees of accuracy. Of course, methods of obtaining this information remotely are not obvious.

One point of consideration that should be made is that even if it's not possible to directly determine some of this information, it may be possible to infer it. For instance, consider a scenario where a vulnerability in a service is exposed remotely. There's nothing to stop an attacker from causing the service to crash. In most cases, the service will restart at some predefined point (such as 30 seconds after the crash). Using this approach, an attacker could infer the creation time of the process based on the time that the crash was generated. This isn't fool proof, but it should be possible to get fairly close.

Determining process and thread identifier could be tricky, especially if the system has been up for some time. The author is not aware of a general purpose technique that could be used to determine this information remotely. Fortunately, the process and thread identifier have very little effect on high order bits.

The system uptime is an interesting one. In the past, there have been techniques that could be used to estimate the uptime of the system through the use of TCP timestamps and other network protocol anomalies. At the time of this writing, the author is not aware of how prevalent or useful these techniques are against modern operating systems. Should they still be effective, they would represent a particularly useful way of obtaining a system's uptime. If an attacker can obtain both the creation time of the process and the uptime of the system, it's possible to calculate the tick count and performance counter values with varying degrees of accuracy.

The performance counter will still pose a great challenge in the remote scenario. The reliance on the performance frequency shouldn't be seen as an unknown quantity. As far as the author is aware, the performance frequency on modern processors is generally 3579545, though there may be certain power situations that would cause it to be different.

It is also important to note that the current attack assumes that the load time for an image that has a GS cookie is equivalent to the initial thread's creation time. For example, if a DLL were loaded much later in process execution, such as through instantiating a COM object in Internet Explorer, it would not be possible to assume that initial thread creation time is equal to the system time that was obtained when the DLL's GS cookie was generated. This brings about an interesting point for the remote scenario, however. If an attacker can control the time at which a DLL is loaded, it may be possible for them to infer the value of system time that is used without even having to directly query it. One example of this would be in the context of internet explorer, where the client's date and time functionality might be abused to obtain this information.

Chapter 8

Conclusion

The ability to reduce the amount of effective entropy in a GS cookie can improve an attacker's chances of guessing the cookie. This paper has described two techniques that may be used to calculate or infer the values of certain bits in a GS cookie. The first approach involves a local attacker's ability to collect information that makes it possible to calculate, with pretty good accuracy, the values of the entropy sources that were used at the time that a cookie was generated. The second approach describes the potential for abusing the limited entropy associated with boot start services.

While the results shown in this paper do not represent a complete break of GS, they do hint toward a general weakness in the way that GS cookies are generated. This is particularly serious given the fact that GS is a compile time solution. If the techniques described in this document are refined, or new and improved techniques are identified, a complete break of GS would require the recompilation of all affected binaries. The implications of this should be obvious. The ability to reliably predict the value of a GS cookie would effectively nullify any benefits that GS adds. It would mean that all stack-based buffer overflows would immediately become exploitable.

To help contribute to the improvement of GS, a few different solutions were described that could either partially or wholly address some of the weakness that were identified. The most interesting of these solutions involves modifying the GS implementation to make use of a external cookie generator, such as the kernel. Going this route would ensure that any weaknesses found in the cookie generation algorithm could be simply addressed through a patch to the kernel. This is much more reasonable than expecting all existing GS enabled binaries to be recompiled.

It's unclear whether the techniques presented in this paper will have any appreciable effect on future exploits. Only time will tell.

Bibliography

[1] Cowan, Crispin et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan_html/cowan.html; accessed 3/18/2007.

[2] Etoh, Hiroaki. *GCC extension for protecting applications from stack-smashing attacks.* http://www.research.ibm.com/trl/projects/security/ssp/; accessed 3/18/2007.

[3] eEye. *Memory Retrieval Vulnerabilities.* http://research.eeye.com/html/Papers/download/eeyeMRV-Oct2006.pdf; accessed 3/18/2007.

[4] Litchfield, David. *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server* http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf; accessed 3/18/2007.

[5] Microsoft Corporation. */GS (Buffer Security Check).* http://msdn2.microsoft.com/en-us/library/8dbf701c(VS.80).aspx; accessed 3/18/2007.

[6] Microsoft Corporation. */SAFESEH (Image has Safe Exception Handlers).* http://msdn2.microsoft.com/en-us/library/9a89h429(VS.80).aspx; accessed 3/18/2007.

[7] Microsoft Corporation. *QueryPerformanceFrequency Function.* http://msdn2.microsoft.com/en-us/library/ms644905.aspx; accessed 3/18/2007

[8] Microsoft Corporation. *QueryPerformanceCounter Function.* http://msdn2.microsoft.com/en-us/library/ms644904.aspx; accessed 3/18/2007

[9] Ren, Chris et al. *Microsoft Compiler Flaw Technical Note* http://www.cigital.com/news/index.php?pg=art&artid=70; accessed 3/18/2007.

[10] Whitehouse, Ollie. *Analysis of GS protections in Windows Vista* http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf; accessed 3/20/2007.