Exploiting the otherwise non-exploitable

Windows Kernel-mode GS Cookies subverted

Matthew "j00ru" Jurczyk
Hispasec

Gynvael Coldwind

January 11, 2011

Abstract: This paper describes various techniques that can be used to reduce the effective entropy of GS cookies implemented in a certain group of Windows kernel-mode executable images by roughly 99%, or otherwise defeat it completely. This reduction is made possible due to the fact that GS uses a number of extremely weak entropy sources, which can be predicted by the attacker with varying (most often - very high) degree of accuracy. In addition to presenting theoretical considerations related to the problem, the paper also contains a great amount of experimental results, showing the actual success / failure rate of different cookie prediction techniques, as well as pieces of hardware-related information. Furthermore, some of the possible problem solutions are presented, together with a brief description of potential attack vectors against these enhancements. Finally, the authors show how the described material can be practically used to improve kernel exploits' reliability - taking the CVE-2010-4398 [1] kernel vulnerability as an interesting example.

1. Introduction

The *stack-based buffer overflow* class is one of the most commonly known software vulnerability types, since the very first years of the software security industry. As a consequence of Intel processor architecture, design notes and manuals, a majority of system platforms and programming conventions are based on heavy stack utilization, mostly for the purpose storing of *stack frames*, *return addresses* and local variables. Although the design can be considered relatively efficient in terms of native code execution time, it also puts applications or entire systems at serious security and reliability risk. Placing local variables and buffers in the same continuous memory context together with critical pointers to executable code might turn out to be exceptionally dangerous, if the program or a single procedure is prone to a security flaw. Some of the potential scenarios, in which an attacker would be able to alter the *return address*, include string-based buffer overflows (e.g. a typical boundless *strcpy* function call), other buffer overflow classes, *out-of-bounds* array writes and other, less often observed flaws.

Through decades, researchers have invented a number of techniques, improving the general reliability level of stack-based exploitation. At some point in time, whenever a security expert did manage to modify a legit return address, and point it into user-controlled memory areas, he

would be able to execute malicious code in the context of the affected application, with nearly a 100% success rate. In order to address the entire attack surface, compiler vendors began to equip the output binaries with various protection schemes, such as local variable reordering, or stack cookies (otherwise known as *stack canaries*). The latter mechanism is known under several names, depending on the implementation in consideration; the most important of which are: StackGuard and ProPolice for GCC, and /GS for Microsoft Visual C++. The general concept behind each implementation of the technique is to prevent the attacker from making use of a hijacked return address, by validating the stack consistency right before returning from a routine.

Due to the fact that both user- and kernel-mode modules are written in native languages (such as C or C++), which are potentially vulnerable to memory corruption flaws, it is reasonable to apply all available protections mechanisms in both CPU privilege levels. Such an approach can be observed in the context of the Microsoft Windows operating system, since the Microsoft C/C++ compiler (also known as cl.exe) is employed to compile both typical user applications and device drivers. Interestingly enough, thorough research has already been carried out on how safe Microsoft's GS cookie implementation is - the results of skape's work can be found in [1]. However, the author only focused on the entropy level of stack cookie sources applied in ring-3, so a majority of the presented techniques are only applicable in the context of Local Privilege Escalation attacks (i.e. the attacker had to be able to execute code on the victim machine) against user-mode processes with high privileges (such as system *services*). This article aims to present the current protection level provided by GS cookies implemented in the default and custom Windows device drivers, and discuss some of the possible solutions for improving the current implementation.

Note: The presented material concerns all of the kernel modules, running on the Windows XP, Windows Vista and Windows 7 operating systems, except for the core kernel images (*ntoskrnl.exe, ntkrnlpa.exe, ntkrnlmp.exe, ntkrpamp.exe*). This is primarily caused by the fact, that the GS cookie initialization present in the kernel differs from the one found in traditional drivers, and is therefore out of the scope of this paper.

Also, we tend to use Windows version-directed terminology (e.g. *something takes place on Windows Vista*) instead of the DDK/WDK version, due to the fact that most attention was focused on defeating the protection schemes found in the default Windows drivers. Still, all of the presented information is confirmed to be valid for the latest versions of the *Windows Driver Kit*.

One, particularly interesting thing observed by us is that the Windows XP SP3 kernel image makes use of the standard GS protection, but does not initialize the nt!__security_cookie variable, in the first place. Such behavior (i.e. always using the default cookie value) has been confirmed by reverse engineering the executable image, as well as performing empirical tests on two independent machines. All of the factors known to the authors seem to confirm that the stack protection of the Windows XP kernel image is actually a fake, in fact.

2. Generating the cookie

Even though the GS Cookie mechanism is available for both user- and kernel-mode code, and the overall concept is identical, the cookie value generation process itself greatly differs between user and kernel mode. The generation of the user-mode cookie has been described in depth in [2] and is out of scope of this paper.

The kernel mode GS cookie is generated at the driver entry point, either in a procedure called GsDriverEntry, an internal __security_init_cookie procedure, or at the beginning of the DriverEntry function itself. In order to form the global cookie value, only two entropy sources are utilized:

- the virtual address of the global cookie variable,
- the nt!KeTickCount value or its part.

Additionally, the cookie is generated once per system session - if the global variable is detected to be already initialized, it is never filled for the second time. However, three major factors - the size of the cookie variable, the number of effectively used bits and possible prologue / epilogue modifications - differ between separate Windows editions and architectures (be it 32- or 64-bit). We have analyzed the GS cookie generation and verification implementation present in the standard *win32k.sys* driver, across different Windows versions and architectures (i.e. x86 and 86-64, excluding the Itanium IA-64). All of the known differences are thoroughly covered in the following subsections.

2.1. Windows XP / 2003 32-bit

In the implementation used in both Windows XP and Windows 2003, the cookie is a 32-bit variable with only the bottom 16 bits being effectively used, and the rest of the bits cleared. The cookie value is generated by combining (using XOR) the lowest portion of the nt!KeTickCount variable with the virtual address of the cookie variable, right-shifted by 8 bits (see Listing 1). At the very end, the value is truncated to 16 bits, which we suspect to be a string buffer overflow mitigation (for example, we believe that the CVE-2009-1126[3] stack-based *wcscpy* buffer overflow vulnerability is made non-exploitable, even if the GS cookie value can be correctly predicted by an attacker).

```
const DWORD DEFAULT_VALUE = 0xBB40;

if(__security_cookie == 0 || __security_cookie == DEFAULT_VALUE)
{
 __security_cookie = ((&__security_cookie) >> 8) ^ KeTickCount.LowPart) & 0xffff;

 if(__security_cookie == 0)
   __security_cookie = DEFAULT_VALUE;
}

__security_cookie_complement = ~(__security_cookie);
```

Listing 1. Windows XP 32-bit GS cookie generation pseudo-code.

The cookie is stored on the stack using a straightforward assignment in the function's prologue (see Listing 2).

```
mov     eax, ___security_cookie
mov     [ebp+cookie], eax
```

Listing 2. The prologue of an exemplary GS-protected routine, on Windows XP / 2003.

At the end of the function execution path, a check is made using a call to the *__security_check_cookie* function (see Listing 3), which expects the current cookie value in the ECX register and compares it with the global cookie value. Additionally, the top 16 bits of the number are tested against zero.

```
mov     ecx, [ebp+cookie]
call    @__security_check_cookie@4
leave
retn    4

@__security_check_cookie@4 proc near
cmp     ecx, ___security_cookie
jnz     ___report_gsfailure
test    ecx, 0FFFF0000h
jnz     ___report_gsfailure
retn
@__security_check_cookie@4 endp
```

Listing 3. Stack cookie verification, at the end of the GS-protected routine, on Windows XP / 2003.

As one might suppose, the __report_gsfailure function is responsible for halting the system's execution; this is usually achieved by triggering a Blue Screen of Death, with adequate bugcheck code.

2.2. Windows Vista / 7 / 2008 32-bit

In Windows Vista and later, the cookie generation procedure is similar but not identical, with the cookie being a full 32-bit variable (see Listing 4). Again, the cookie is generated by combining the virtual address of the __security_cookie variable with the low part of the nt!KeTickCount variable, but the shifting and truncating has been removed.

```
const DWORD DEFAULT_VALUE = 0BB40E64Eh;

if(__security_cookie == 0 || __security_cookie == DEFAULT_VALUE)
{
 __security_cookie = KeTickCount.LowPart ^ (&__security_cookie);
 if(__security_cookie == 0)
   __security_cookie = DEFAULT_VALUE;
}

__security_cookie_complement = ~__security_cookie;
```

Listing 4. Windows Vista and later 32-bit GS cookie generation pseudo-code.

One relevant modification that should be noted here is the method of storing the cookie on the stack by the function prologue: the cookie is combined (XOR) with the value of the EBP register value and then stored on the stack (see Listing 5).

```
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+cookie], eax
```

Listing 5. The prologue of an exemplary GS-protected routine, on Windows Vista and later.

Accordingly, the cookie check in the epilogue first XORs the stored cookie with EBP, before calling the __security_check_cookie procedure (see Listing 6).

```
mov     ecx, [ebp+cookie]
xor     ecx, ebp
call    @__security_check_cookie@4
leave
retn    4

@__security_check_cookie@4 proc near
cmp     ecx, ___security_cookie
jnz     ___report_gsfailure
retn
@__security_check_cookie@4 endp
```

Listing 6. Stack cookie verification, at the end of the GS-protected routine, on Windows Vista and later.

As one can see, since a total of 32 bits are used by the cookie, the second check (referencing the upper 16 bits) has been removed.

2.3. Windows XP/2003 64-bit and Vista/7/2008 64-bit

The GS Cookie was extended to 64 bits with only 48 bits actually used and the rest cleared. The generation is similar to the previous case (see Listing 7).

```
const QWORD DEFAULT_VALUE = 2B992DDFA232h;

if(__security_cookie == 0 || __security_cookie == DEFAULT_VALUE)
{
 __security_cookie = (KeTickCount ^ (&__security_cookie)) & 0xFFFFFFFFFFFF;
 if(__security_cookie == 0)
   __security_cookie = DEFAULT_VALUE;
}

__security_cookie_complement = ~__security_cookie;
```

Listing 7. The GS cookie generation pseudo-code, on 64-bit Windows platforms.

Similarly to Windows Vista, the cookie is XORed with the stack pointer (RSP) before being placed on the stack (see Listing 8). The __security_check_cookie procedure not only compares the cookie value, but also checks if all of the top 16 bits are clear (see Listing 9).

```
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rsp+cookie], rax
```

Listing 8. The cookie initialization on stack, on 64-bit Windows platforms.

```
mov     rcx, [rsp+cookie]
xor     rcx, rsp
call    __security_check_cookie
retn

__security_check_cookie proc near
cmp     rcx, cs:__security_cookie
jnz     short report_gsfailure
rol     rcx, 10h
test    cx, 0FFFFh
jnz     short report_gsfailure_2
retn    0
```

Listing 9. Stack cookie verification, at the end of the GS-protected routine, on 64-bit Windows platforms.

The GS cookie generation method found in Windows XP 64-bit and Windows 2003 64-bit is generally analogous; the only difference is that the stack address entropy source is not employed, as opposed to newer versions of the system.

2.4. Summary of the cookie generation

Table 1 summarizes the cookie generation mechanism found across different Windows versions.

Windows Version	Cookie Variable Size	Number of bits used	Additional operations while applying	Total number of entropy sources
XP 32-bit and 2003 32-bit	32	16	none	2
Vista 32-bit and later	32	32	XOR with EBP	3
XP 64-bit and 2003 64-bit	64	48	none	2
Vista 64-bit and later	64	48	XOR with RSP	3

Table 1. Summary of the GS cookie protection scheme factors, implemented within various Windows editions.

3. Attacking GS

This section provides detailed information on how each separate source of the kernel GS cookie entropy can be predicted by a potential attacker. Two out of the three entropy sources are considered *trivial* to calculate, and can be dealt with using generic, straight-forward techniques. Consequently, we have put a lot of effort in developing methods which can be used to handle the third, and last factor; the system *tick count*. In this particular case, several attack vectors have been invented; relying on the specific drivers' characteristics and behavior. Accordingly, three separate categories of the Windows device drivers are considered in this document:

1. **Boot / Manual device drivers**
   All executable modules which are loaded into kernel space during the OS boot process are referred to as *boot drivers*. On the other hand, the remaining drivers, manually loaded by either typical user-mode applications or the system itself (upon certain events, such as inserting a flash drive into the port), are called *manual load drivers*, since their load time is not a determined, relative to the system start-up time, value.

2. **Public / Non-public device drivers**
   A great number of images present inside the kernel memory areas provide a convenient communication channel to user-mode client applications (e.g. programs responsible for displaying a graphical user interface, such as antivirus software). In order to do so, these modules are obliged to create a named *device* resource (using the IoCreateDevice kernel API), and make it available to certain users or user groups, by specifying adequate access ACL settings. After performing this operation, the drivers start dispatching signals sent to the public device, and in that sense they can be considered *public* (or *visible*) modules. The remaining group of executable images, including low-level hardware management drivers and mini-filters, is referred to as *non-public*, or *private drivers*.

3. **Default / Custom device drivers**
   The *default* term describes drivers that are present on every installation of the Microsoft Windows operating system, and are loaded into kernel-space by default. One example of such a driver is the graphical management module - win32k.sys. Other drivers, most likely installed by third-party applications, are referred to using the phrase *custom drivers*.

As it turns out, a completely different set of techniques and methods can be applied, depending on the target driver characteristics; *when*, *how*, and *if public* are the key words here. All of the presented considerations can be successfully applied, provided there is local access to the victim system.

3.1 Calculating the *__security_cookie* address

The first entropy source to handle is the virtual address of the global *__security_cookie* value, placed somewhere inside the driver image in consideration. As mentioned before, the global stack canary value is the result of XOR'ing two factors (the tick count and a virtual address within the executable image) in the following manner:

```
mov   edx, ds:__imp__KeTickCount
mov   eax, offset ___security_cookie
shr   eax, 8
xor   eax, [edx]
```

Listing 10. GS cookie initialization disassembly from a Windows XP SP3 (32-bit) executable.

The question is how a potential attacker, logged onto an account with the lowest possible privileges, would be able to know the virtual address of a global variable within one of the kernel modules. As it turns out, it is possible for any user to obtain a list of structures - describing all of the active kernel modules using the native NtQuerySystemInformation API[4], together with the *SystemModuleInformation* parameter. On success, the caller application receives an array of the *SYSTEM_MODULE_INFORMATION_ENTRY* structures (presented on Listing 11), each associated with a separate device driver (including the original kernel image - ntoskrnl.exe - or its equivalent).

```
typedef struct _SYSTEM_MODULE_INFORMATION_ENTRY {
   ULONG     Unknown1;
   ULONG     Unknown2;
   PVOID     Base;
   ULONG     Size;
   ULONG     Flags;
   USHORT    Index;
   USHORT    NameLength;
   USHORT    LoadCount;
   USHORT    PathLength;
   CHAR      ImageName[256];
} SYSTEM_MODULE_INFORMATION_ENTRY, *PSYSTEM_MODULE_INFORMATION_ENTRY;

typedef struct _SYSTEM_MODULE_INFORMATION {
   ULONG  Count;
   SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

Listing 11. The SYSTEM_MODULE_INFORMATION_ENTRY structure definition.

As can be seen, the structure contains every piece of information, that an attacker could possibly need: the full image path and its file name, as well as the image base and image size, respectively called *Base* and *Size.* Given this information, the only missing part of our __security_cookie address puzzle is the global variable offset, relative to the virtual address of the module beginning. This, however, should not pose a serious problem, as this last piece of information can be determined in numerous ways.

First of all the attacker (or a malicious program) is always aware of what driver is to be exploited. Furthermore, the exact driver image version isn't usually kept secret, either. Given the above, one should be able to access a local copy of the .sys file, and manually check the desired offset value. In case of multiple versions of a single module being targeted, the offsets could be simply hard-coded into the exploit, thus avoiding the need to request additional information from the operating system.

Secondly, it is often possible to dynamically download the symbols file for a given executable image, provided that the driver under attack is a default one (such as win32k.sys or afd.sys), and the machine has a working internet connection set up. Given a .pdb symbols file, it is easy to reliably extract the required __security_cookie -related information.

Last, but not least, one can make use of the vulnerable module's assembly code itself. In most cases, the stack canary initialization takes place right inside the driver's entry point (*GsDriverEntry*), and has a very regular form, presented on Listing 12.

```
; __stdcall GsDriverEntry(x, x)
            public _GsDriverEntry@8
_GsDriverEntry@8 proc near

            mov    edi, edi
            push   ebp
            mov    ebp, esp
            mov    eax, ___security_cookie
(...)
```

Listing 12. A simple reference to the __security_cookie address, inside a module entry routine.

Apparently, the first reference to the desired address can be found right after a standard five byte long function prologue. One can take advantage of the observation, by obtaining the imm32 instruction operand, thus automatically receiving the virtual address of the __security_cookie global symbol, relative to the original image base, stored in the PE file headers.

As shown above, the first entropy source can be calculated with a 100% success rate, with no actual rights in the system.

3.2 Calculating the Stack Frame Pointer (EBP register) value

The second source of entropy, used to form the stack canary on Windows Vista and 7, is the EBP / ESP (and RBP/ RSP, accordingly) register content. One might suppose that the value is hidden from any user-mode application, so that the attacker is unable to calculate, or obtain the address of a kernel stack. Fortunately for us, however, it turns out that the Windows kernel does not restrict access to such information to users with any specific rights; the *NtQuerySystemInformation* routine proves to be extremely useful, once again.

This time, our interest is focused around the *SystemExtendedProcessInformation* information class - thanks to this functionality, any process running on the machine can obtain complete information regarding every active process and thread in the system. To be more precise, the native API output is contained inside a list of three structures, presented in Listing 13.

```
typedef struct _SYSTEM_THREAD_INFORMATION
{
       LARGE_INTEGER KernelTime;
       LARGE_INTEGER UserTime;
       LARGE_INTEGER CreateTime;
       ULONG WaitTime;
       PVOID StartAddress;
       CLIENT_ID ClientId;
       LONG Priority;
       LONG BasePriority;
       ULONG ContextSwitches;
       ULONG ThreadState;
       ULONG WaitReason;
} SYSTEM_THREAD_INFORMATION, *PSYSTEM_THREAD_INFORMATION;

typedef struct _SYSTEM_EXTENDED_THREAD_INFORMATION
{
       SYSTEM_THREAD_INFORMATION ThreadInfo;
       PVOID StackBase;
       PVOID StackLimit;
       PVOID Win32StartAddress;
       PVOID TebAddress;
       ULONG Reserved1;
       ULONG Reserved2;
       ULONG Reserved3;
} SYSTEM_EXTENDED_THREAD_INFORMATION, *PSYSTEM_EXTENDED_THREAD_INFORMATION;

typedef struct _SYSTEM_PROCESS_INFORMATION
{
       ULONG NextEntryOffset;
       ULONG NumberOfThreads;
       LARGE_INTEGER SpareLi1;
       LARGE_INTEGER SpareLi2;
       LARGE_INTEGER SpareLi3;
       LARGE_INTEGER CreateTime;
       LARGE_INTEGER UserTime;
       LARGE_INTEGER KernelTime;
       UNICODE_STRING ImageName;
       KPRIORITY BasePriority;
       ULONG UniqueProcessId;
       ULONG InheritedFromUniqueProcessId;
       ULONG HandleCount;
       ULONG SessionId;
       PVOID PageDirectoryBase;
       VM_COUNTERS VirtualMemoryCounters;
       SIZE_T PrivatePageCount;
       IO_COUNTERS IoCounters;
       SYSTEM_EXTENDED_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

Listing 13. The output structures, used to return information upon issuing the *SystemExtendedProcessInformation* information class request.

Two fields, which are by far the most interesting for an attacker, are marked with a red color (and bold). The importance of these values stems from the fact that they describe the top and bottom addresses of the kernel-mode stack, assigned to a given thread. Due to the fact that the API provides information about all of the threads, it is easy to find the stack information of the current execution unit.

Considering the fact that the EBP / ESP address is shifted by a constant offset, relative to the thread's stack base when crafting the stack canary, the attacker can easily calculate the stack-frame factor of the cookie, by just obtaining the current thread's stack base and subtracting a hard-coded value from the top of the stack. One should keep in mind, however, that the vulnerability should be triggered from within the same thread which was used to calculate the stack address value; otherwise, a different stack frame pointer would be used, and the entire attack would end in failure.

3.3. Calculating the CPU Tick Count

As shown, both previous factors used to form the GS cookie in its final form provide hardly any protection against successful exploitation; the correct value can be simply obtained from the system, hence we can be sure that the retrieved information is 100% accurate. The last cookie-generation source is slightly more tricky to calculate, because it is based on a truly unpredictable factor (time), and cannot be directly read from the operating system. Given the above conclusions, the authors have came up with three different approaches, which can be employed in practical attacks, all of which are applicable under a certain set of conditions. Although this explanation doesn't sound like a complete break of GS, it turns out that a great majority of device drivers installed on regular computers falls into at least one of the presented categories.

One question that should be answered before considering possible attack vectors is: *what is the actual time period between a system's clock interrupts?* Since the authors were not aware of any precise answer to this question, a special "*social campaign"* [1] was started, aiming at collecting experimental results from numerous hardware and system platforms, as well as virtualized environments. In one week, the authors managed to collect around 280 submissions, from pretty much every possible Windows edition, and both Intel and AMD processor vendors (interestingly, some of the samples indicated that the test program was also run under *Wine*.). A brief summary of the survey results is presented in Table 2.

[1] The visitors of the http://j00ru.vexillium.org/ticks/ website were requested to post the output of a simple console application returning the GetSystemTimeAdjustment function output.

Tick interval	Occurrence #	Occurrence rate	Comment
100000	4	1,54%	*Wine output*
100144	28	10,84%	Virtual machines' output (most often *VirtualBox*)
149952	1	0,38%	Unknown result
156001	107	41,31%	Regular PC value
156250	119	45,94%	Regular PC value

Table 2. Experimental results of the *GetSystemTimeAdjustment* API function; based on a total of 259 valid samples.

As can be seen, there are only two different values, which can usually be observed on typical, desktop machines - 156001 and 156250. Given the fact that the numbers represent the interval between periodic time adjustments in 100-nanosecond units, it turns out that the magic 156250 number actually stands for exactly $^1/_{64}$ of a second. Apparently, the other frequent duration cannot be translated into such a nice fraction. However, in order to simplify our considerations, we can just approximate these two values as one; a difference of 24900 nanoseconds should not make a relevant difference for an attacker.

All in all, the main observation to be made here is that a duration of $^1/_{64}$s provides a very low entropy level especially if additional factors, such as a deterministic boot process, or the user's ability to query various performance counters, are taken into consideration. When one realizes how weak is that, he will easily grasp the cookie-calculation techniques, presented further in this section. All of the presented methods do nothing but try to estimate the loading time of a device driver, based on all of the available information, that the operating system can provide to a restricted user.

Knowing the basics of the cookie generation weakness, let's figure out what are the possible ways of subverting the flawed implementation.

3.3.1. Boot / manual load drivers

In general, a total of the Windows device drivers can be divided into several groups, based solely on their loading time; a list of the possible service types follows:

- Boot drivers
- System drivers
- Automatic drivers
- Manual-load drivers
- Disabled drivers

Due to the fact that (almost) every legitimate driver must be registered to the *Service Control Manager* mechanism before being allowed to execute, the “load time” setting is explicitly chosen by the registrar, when calling the CreateService API function. This specific characteristic of any device driver can also be investigated manually, by checking the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\NAME\Start registry value.

What sets the modules placed within the first three groups apart is *how early* in the system start-up process the driver is loaded into kernel space. As it turns out, some drivers (i.e. low-level hardware management drivers) are loaded as early as the Windows kernel image itself, while others are given a chance to execute further, during the system boot.

One, particularly interesting observation for a potential attacker is that the entire Windows NT boot process is extremely deterministic - given a list of the installed device drivers (or even without it) and hardware specification (such as the CPU frequency), one should be able to predict the amount of time required to successfully launch the operating system. This is primarily caused by the fact that the only factors which might potentially affect the booting time are *random* hardware delays (e.g. disk drive and random access memory lags) or filesystem-related issues, such as varying fragmentation level. Even though these factors *exist*, it is very unlikely that they can result in a relevant level of tick count unpredictability, given the 1/64s duration.

Having made these observations, one could suppose that the effective entropy level of the kernel modules which belong to one of the first three groups (drivers that are loaded before a user logs in) must be extremely low. In order to confirm (or rebut) the thesis, the authors generated a set of experimental results, presenting the cookie value decomposition of **win32k.sys** (standard Windows graphics driver, otherwise known as the ring-0 part of Windows Subsystem), and **wanarp.sys** (another default device driver, loaded at boot-time). The tests were performed on both a regular computer and a virtual machine; the summarized results are presented on Charts 1 and 2. You can also find the complete study of the experimental results in Chapter 4 (Table 1, Charts 7, 8, 9, 10).

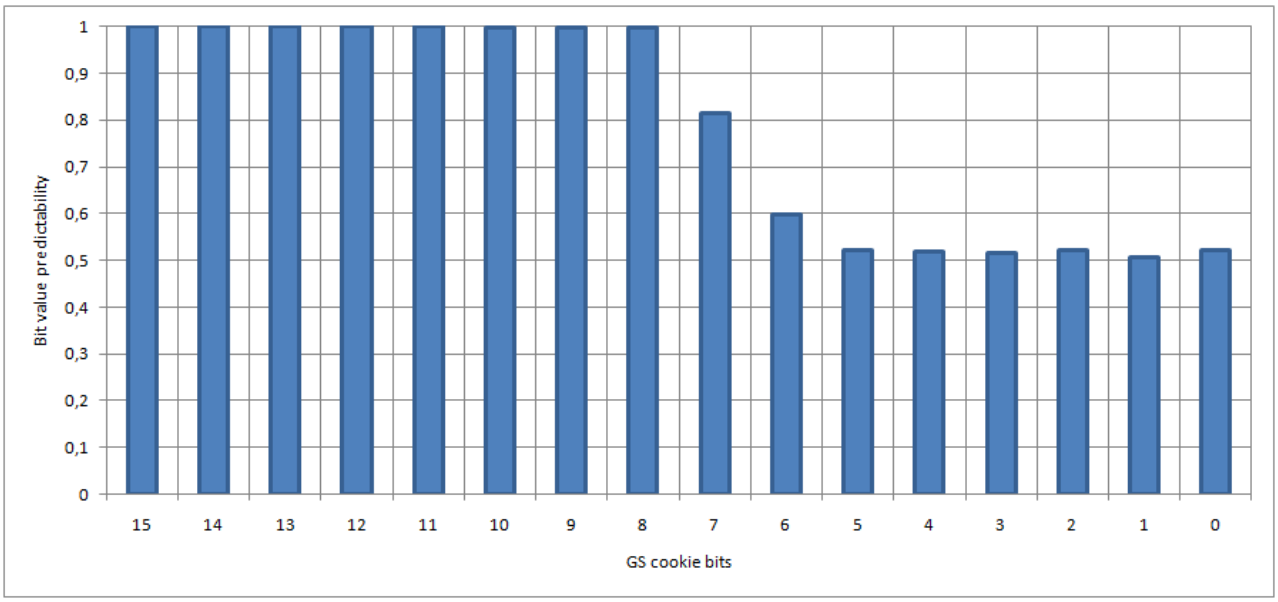

Chart 1. Per-bit entropy level of the GS cookie value for win32k.sys on a regular PC (1.0 - fully predictable, 0.5 - equal bit probability).

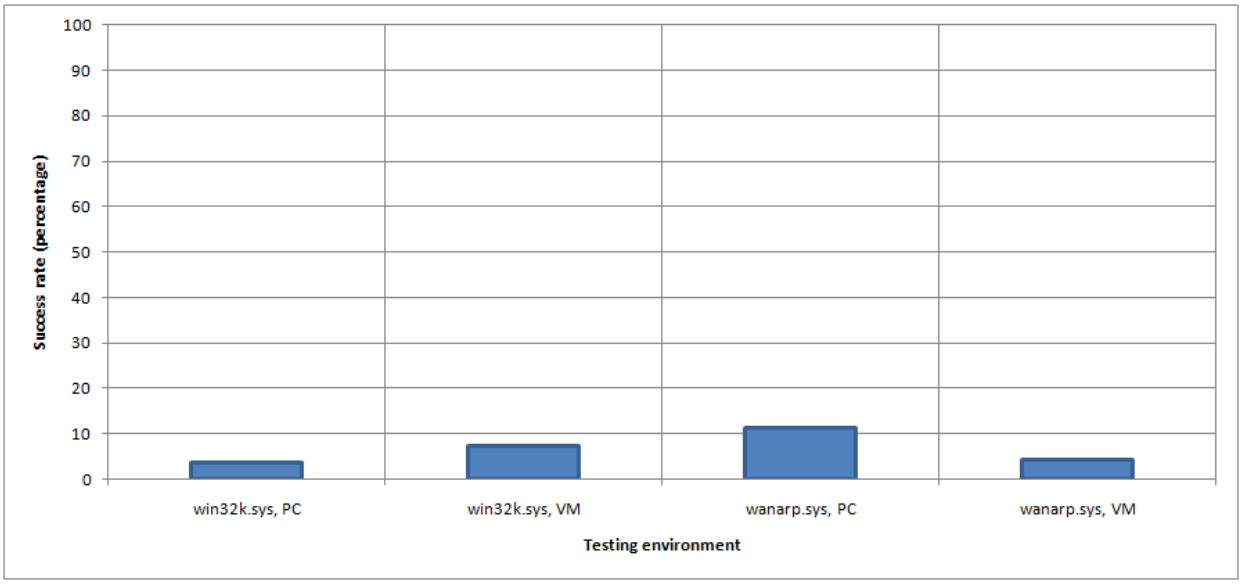

Chart 2. The percentage probability of hitting the correct GS cookie for the win32k.sys and wanarp.sys modules, on both a regular PC and a virtual machine.

As the above charts show, a chance of a successful GS cookie guess is effectively reduced from a potentially feasible probability of $\frac{1}{65536}$ (Windows XP and 2003) or $\frac{1}{4294967296}$ (Windows Vista and later) - provided that all bits of the cookie all truly random - to around $\frac{1}{20}$, due to very weak entropy sources of the canary value generation. What should be noted, is that the device drivers investigated in this subsection are being loaded relatively late in the boot process; e.g.

win32k.sys is launched by the Session Manager (smss.exe) process which, in turn, is created after loading all of the *Boot* and *System* kernel modules. Therefore, the chances of successful exploitation of a driver executed earlier become even greater.

Although the above statistics seem scary, one should keep in mind that calculating the cookie value range for a driver (such as $\langle 857|1037 \rangle$ from Chart 1) is not a trivial task unless the attacker has got some very precise information about the machine and operating system (i.e. CPU and chipset information, a list and description of the registered device drivers, and so on). In order to carry out a successful attack in practice, it is supposed that the best solution for one would be to first get himself a machine with specs equal to the victim's computer, then set up a very similar software environment, and finally perform experimental tests, examining the cookie value spread.

3.3.2. Process-relative / absolute loading-time drivers

In this section, device drivers belonging to the Manual loading group are considered. What should be made clear before going further into the analysis, is the answer to the following question - what does the *manual* term mean, in the first place? Obviously, it is not the user's duty to manually load the essential device drivers. Instead, certain drivers are programatically launched by the associated user-mode applications; one example of such behavior might be anti-rootkit software of some kind - the user decides to perform a full system scan, and runs an AntiRootkit.exe program, which in turn registers AntiRootkit.SYS to SCM, and starts the service. Interestingly, even the well-known *wink32.sys* module can be considered process-relative in terms of its loading time, since it is directly loaded by the SMSS.EXE application, by using the NtSetSystemInformation native call, together with the *SystemLoadAndCallImage* (38) argument.

In this scenario, the attacker is unable to make use of the fact presented in the previous chapter - the absolute (i.e. relative to system boot time) load time cannot be easily calculated, as it is often the user himself, who decides about when a specific service should be launched (directly, or indirectly - through a program functionality, which requires a kernel module to execute). However, one nice thing about the Windows NT kernel is that it is more than willing to share various types of information regarding the current system state, with users with no special privileges (e.g. a typical *Guest* account). And so, if one decides to make the following call:

```
NtQuerySystemInformation(SystemProcessInformation , …);
```

or

```
NtQuerySystemInformation(SystemExtendedProcessInformation, …);
```

he will receive a complete description of each process and thread currently running in the system; the technique has already been used to retrieve the kernel stack base and limit of a given thread. This time, we are going to obtain the creation time of the process, associated with the ring-0 module under attack - the information can be easily extracted from the PROCESS_INFORMATION.CreateTime structure:

```
(…)
     LARGE_INTEGER SpareLi2;
     LARGE_INTEGER SpareLi3;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER UserTime;
     LARGE_INTEGER KernelTime;
(…)
```

The field (currently declared to hold 64-bit integers) represents the absolute date of the process creation, in a standard format (100 nanosecond units, since 1st January of 1601). The question is - how is a potential attacker able to make use of this kind of information? Let's take a look at Image 1:

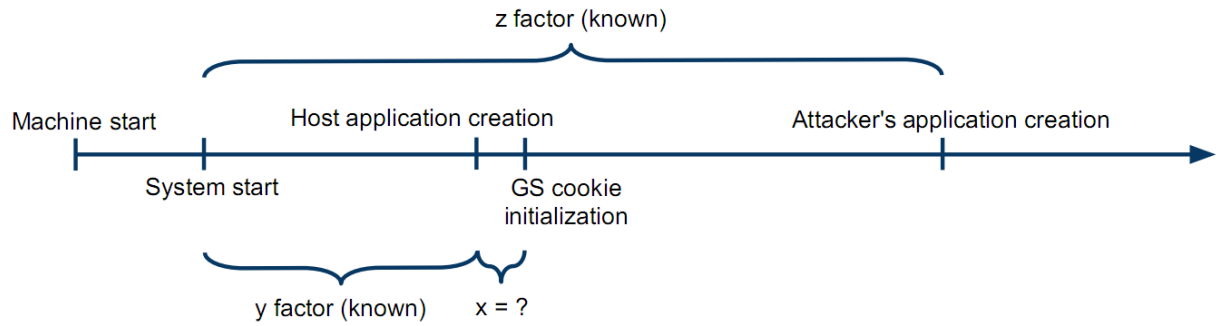

Image 1. The victim machine execution timeline.

As presented, the attacker has a chance to execute his code at some point in time, after the host application was launched and stack canary initialized.

The known factors are:

1. The current tick count, at the time of executing the Attacker's Application,
2. The absolute creation time of the Host Application and Attacker's Application.

Thanks to the above information, we can easily calculate the system tick count, at the time of the Host Process creation:

$$Host\ ticks = Attacker\ ticks - \frac{(Attacker\ CreationTime - Host\ CreationTime)}{GetSystemTimeAdjustement(TimeInterval)}$$

At the time of writing this paper, the authors are unaware of any documented API function, which would make it possible to read the current tick count. Instead, one can obtain the number of milliseconds that have elapsed since the system start, using a function with somewhat misleading name - GetTickCount. One idea of solving the problem might be to calculate the actual tick count, , based on the output of GetTickCount and GetSystemTimeAdjustment:

$$Attacker\ Ticks = \frac{GetTickCount * 10^4}{GetSystemTimeAdjustment(TimeInterval)}$$

Unfortunately, the resolution of the *GetTickCount* output value is insufficiently low (one-milisecond units), thus the result might turn out to be inaccurate. A more precise method of obtaining the tick count would be to directly refer to the original source - in this case, to the KUSER_SHARED_DATA[5] structure, mapped under a constant virtual address of 0x7ffe0000 (or 0xfffe0000 in kernel-mode). This special memory area is common to all processes running in the system, and contains bits of essential information regarding the current system session, such as:

1. Numerous counters and timers,
2. System version,
3. CPUID (Processor Features),
4. Number of physical pages,
5. System-wide pointers, e.g. ntdll!KiFastSystemCall or ntdll!KiFastSystemCallRet.

Fortunately, one of the structure fields contains precisely the information we want to obtain:

```
kd> dt _KUSER_SHARED_DATA
nt!_KUSER_SHARED_DATA
  +0x000 TickCountLow      : Uint4B
  +0x004 TickCountMultiplier : Uint4B
  +0x008 InterruptTime     : _KSYSTEM_TIME
  +0x014 SystemTime        : _KSYSTEM_TIME
  +0x020 TimeZoneBias      : _KSYSTEM_TIME
  (...)
  +0x304 SystemCallReturn : Uint4B
  +0x308 SystemCallPad     : [3] Uint8B
  +0x320 TickCount         : _KSYSTEM_TIME
  +0x320 TickCountQuad     : Uint8B
  +0x330 Cookie            : Uint4B
```

Listing 14. Parts of an internal _KUSER_SHARED_DATA structure, containing the current system tick count.

The first marked DWORD - TickCountLow - is supported on the Windows XP platform, whereas newer systems (Vista, 7) consider the field deprecated, and instead make use of a 64-bit TickCount structure. Listing 15 presents an XP-compatible function, which should be sufficient for a majority of implementations.

```
DWORD GetRealTickCount()
{
  PDWORD lpTicks = (PDWORD)0x7ffe0000;
  return (*lpTicks);
}
```

Listing 15. A simple implementation of a function, responsible for retrieving the current system tick count.

Knowing that the __security_cookie variable of the driver in consideration is initialized, based on the total number of ticks that elapsed since system startup, and having the number of ticks that passed between system start-up and the creation of a user-mode client, the *x variable* (the time between Host App's creation and stack canary initialization) becomes the only unknown part of our equation.

Due to the fact that the unknown period of time is usually very short (especially if loading a kernel module is the first objective achieved by the program) and deterministic, it is claimed that the prediction of the *x variable* should not pose a serious problem for a determined attacker. As mentioned before, win32k.sys can be also considered in terms of process creation time dependency, as it is always loaded in a relatively early stage of the *smss.exe* execution path. Therefore, the authors have performed special tests, aiming to determine the practical *x* variable values on both a regular PC and a virtual machine. Summarized results of the experiments are presented in Charts 3 and 4. Complete results related to process-relative ticks can be found in Chapter 5 (Table 4, Charts 11, 12).

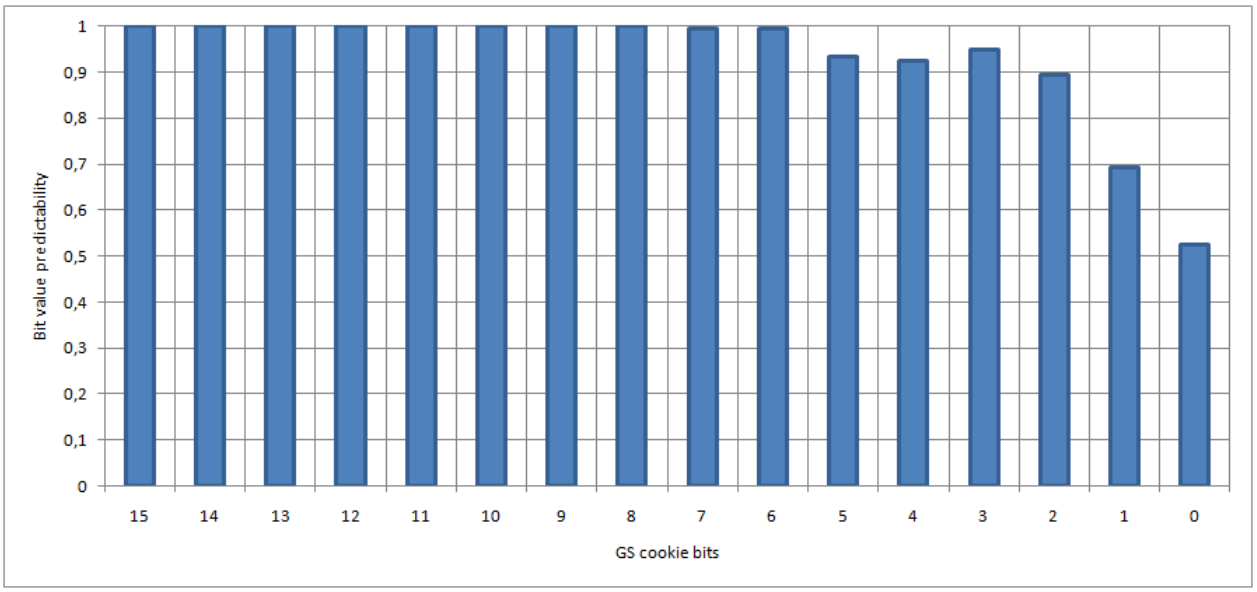

Chart 3. Per-bit entropy level of the unknown *x* variable, using the "process-relative estimation" technique for win32k.sys on a regular PC (1,0 - fully predictable, 0,5 - random).

What should be observed is that the general prediction accuracy has dramatically increased - from 7,19% to 15,28% (over twice) on a virtual machine, and from 3,69% to 41,84% (!) on a typical desktop computer. One of the reasons, why guessing the tick count on a virtual machine becomes less reliable then a real machine, is the higher tick frequency (156250 units on a PC / 100144 units on a VM).

As it turns out, however, there are ways to estimate the tick count even more precisely - by making use of the Windows objects.

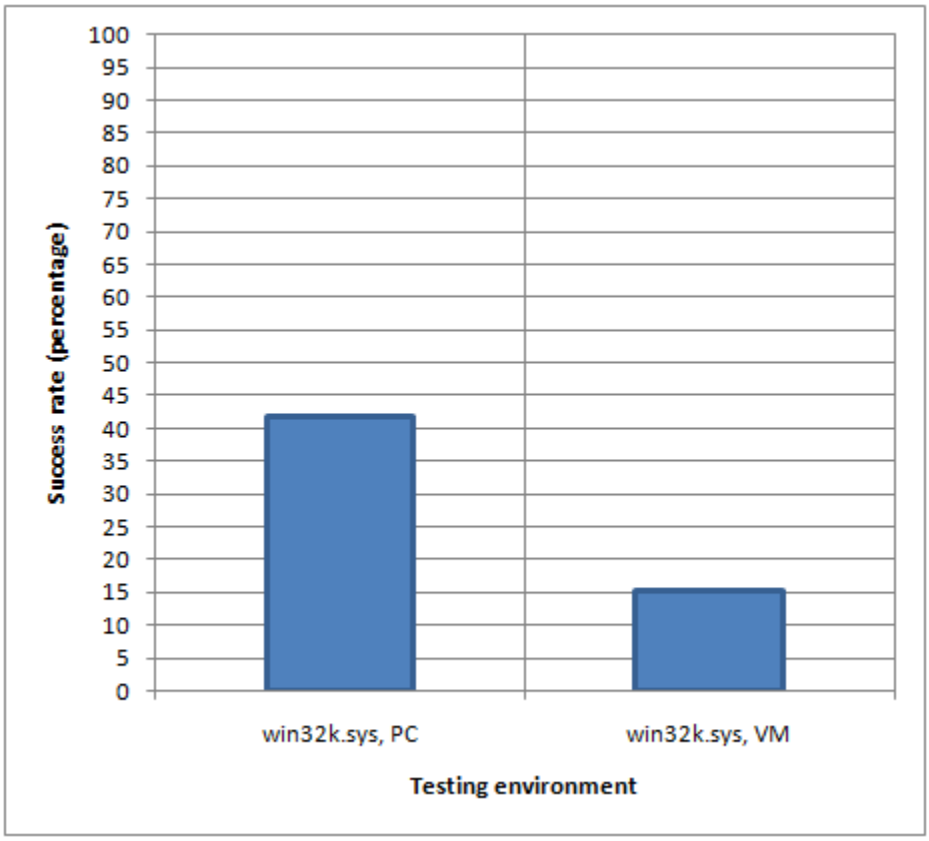

Chart 4. The probability of hitting the correct GS cookie for the win32k.sys module, on both a regular PC and a virtual machine.

3.3.3. Public / non-public drivers

Amongst a variety of other functionalities, Windows makes it possible for any user to receive detailed information about an object (i.e. a resource) previously opened by the requesting thread. This simple goal can be achieved by taking advantage of the NtQueryObject function, together with the ObjectBasicInformation parameter. Upon issuing a correct call to the native routine, an OBJECT_BASIC_INFORMATION structure, presented on Listing 16, is returned to the process.

Concluding from the structure declaration, one might suppose that it is possible to retrieve the creation time of any system resource, provided that the user in consideration (the attacker) has the rights to such a previously opened resource. Unfortunately, this is not entirely true.

In order to figure out how exactly the *CreateTime* field is managed by the kernel itself, one should take a look at the actual implementation of the nt!NtQueryObject routine. Upon disassembling the ntoskrnl.exe image and finding the desired function address, one should observe the assembly code snippet presented on Listing 17.

```
typedef struct _OBJECT_BASIC_INFORMATION {
ULONG                   Attributes;
ACCESS_MASK             DesiredAccess;
ULONG                   HandleCount;
ULONG                   ReferenceCount;
ULONG                   PagedPoolUsage;
ULONG                   NonPagedPoolUsage;
ULONG                   Reserved[3];
ULONG                   NameInformationLength;
ULONG                   TypeInformationLength;
ULONG                   SecurityDescriptorLength;
LARGE_INTEGER           CreationTime;
} OBJECT_BASIC_INFORMATION, *POBJECT_BASIC_INFORMATION;
```

Listing 16. The OBJECT_BASIC_INFORMATION structure defitinion.

```
      cmp     ebx, _ObpSymbolicLinkObjectType
      jnz     short loc_52020A

      mov     eax, [ebp+Object.CreationTime]
      mov     ecx, [eax+LowPart]
      mov     [ebp+ObjectBasicInfo.CreationTime.LowPart], ecx
      mov     eax, [eax+HighPart]
      mov     [ebp+ObjectBasicInfo.CreationTime.HighPart], eax
      jmp     short loc_520214

loc_52020A:
      xor     eax, eax
      lea     edi, [ebp+ObjectBasicinfo.CreationTime.LowPart]
      stosd
      stosd
```

Listing 17. Part of the nt!NtQueryObject system call handler, responsible for filling the output *CreationTime* structure field.

The binary code can be easily translated to a high-level language (Listing 18).

```
if (ObjectType == ObpSymbolicLinkObjectType)
{
  ObjectBasicInfo.CreationTime = Object->CreationTime;
}
else
{
  RtlZeroMemory(&ObjectBasicInfo.CreationTime,sizeof(LARGE_INTEGER);
}
```

Listing 18. High-level language representation of the *CreationTime* structure initialization.

Apparently, the only case, when the *CreationTime* value is correctly managed, is when the request is made for a symbolic link object; otherwise, the output LARGE_INTEGER structure is simply zero-ed out. The above behaviour has been confirmed both empirically and by reverse code engineering not only on Windows XP, but also Windows Vista and 7 kernel images. The question is - how this (seemingly limited) functionality could be of much use to a potential attacker?

Due to the fact that a significant number of device drivers aims to communicate with user-mode applications, they often create named *device resources* in the system - such a device can be subsequently opened by one or more ring-3 programs, by using the typical CreateFile API, and then freely exchanged information with, by using either the *ReadFile* / *WriteFile* pair (MJ_READ_IRP and MJ_WRITE_IRP), or a single *IoControlDevice* (MJ_CONTROL_IRP) function. In order to make the naming scheme more legible, the drivers often decide to create a symbolic link between the "\\.\Driver" symbol, and a previously created "\Devices\Driver" name. Fortunately, this operation is often performed directly from within the *DriverEntry* routine (during the driver initialization), which is just a few instructions away from crafting the __security_cookie value.

Depending on the DriverEntry - or other initialization - function body, the creation time of a symbolic link object might appear to be the most accurate one, as it lies in the closest tick-distance from the moment of filling the global canary variable. Furthermore, it should be noted that the technique is not only restricted to symbolic link objects created by the ring-0 module - any other *symlink* that is created with a constant time offset relative to the cookie initialization, can be made use of.

Adequate tests have been performed during our research, using the "symlink estimation" method; summarized results of the experiments are presented on Chart 5 and Chart 6 (as usual, a more descriptive information about the tests can be found in Table 15 and Charts 13, 14, 15, 16). The authors decided to stick to the win32k.sys module, as it is a usual attack target, and a very representative example of how the discussed technique can work in practise. In order to achieve the best effect, the tick count difference between win32k!__security__cookie initialization and the \BaseNamedObjects\Session symlink creation (performed by the host, smss.exe) was measured. This difference is represented by the unknown *x variable*; i.e. the

time between a known event (calling NtCreateSymbolicLinkObject[6]) and an unknown event - loading a device driver.

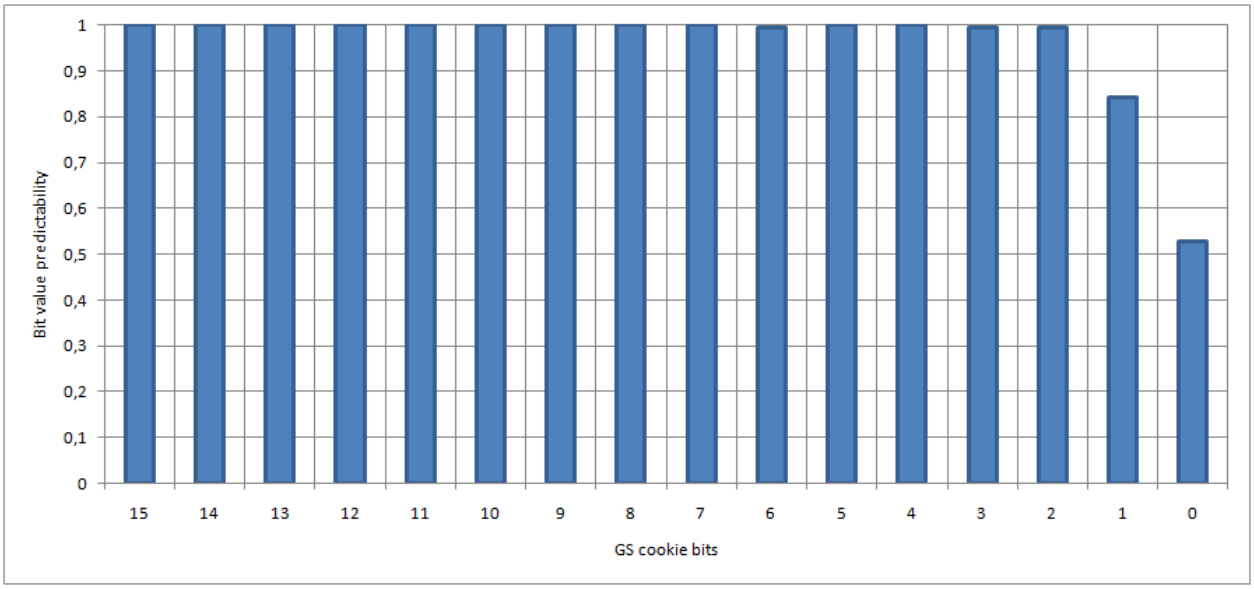

Chart 5. Per-bit entropy level of the unknown *x* variable, using the "symlink estimation" technique for win32k.sys on a regular PC (1,0 - fully predictable, 0,5 - random).

Chart 6. The probability of hitting the correct GS cookie for the win32k.sys and wanarp.sys modules, on both a regular PC and a virtual machine (using the "symlink estimation" technique).

As outlined, the probability of performing a successful attack against the GS cookie has risen to almost $^{1}/_{2}$. Also, the general spread has dramatically decreased, as the *x variable* now ranges

from 5 to 8 (the 70 and 71 values are supposed to be an observational error, as their share does not exceed 1%) . However, one should keep in mind that the results provided by authors are based on a specific behavior of hardware and software platform being tested; hence, other computers are going to produce different outputs, giving a smaller or higher prediction accuracy.

3.3.4. Other techniques

All of the ideas presented in this chapter aim to *guess* the time that elapses between two, certain occurrences taking place on the machine, in order to estimate the number of ticks between the system startup and cookie initialization. Fortunately, significant parts of the unknown period may be reliably calculated, based on the information provided by the operating system, so that the unknown part can be estimated more precisely. We believe that other, potentially more accurate techniques exist, which rely on other events related to the driver loading. However, the techniques discussed in this paper already provide a decent degree of reliability.

Two out of the three techniques described here require the attacker to take advantage of certain Windows system calls. As a consequence, these techniques are only applicable in the context of Local Elevation of Privileges Attacks, i.e. the attacker must have local access to the target system. However, the fact that the tick count resolution is critically low remains, so it is still possible to remotely guess the cookie value with a probability of up to 5% (or more), basing solely on the cpu specs and general system information (which is often known, when a special machine-dedicated attack is being planned and / or performed).

4. Experimental results

In this chapter, the results of practical tests, performed on the authors' machines are discussed in detail.

4.1. Testing platform

The following subsections refer to experimental results, obtained by performing numerous tests on the authors' machines. All of the tests were carried out on one regular PC, with the following specs:

- Intel Core 2 Quad CPU Q8200 @ 2.33 GHz
- Microsoft Windows XP Professional SP3 (32-bit)

and a single virtual machine:

- AMD Phenom II X4 810 @ 2,59 GHz (host)
- Microsoft Windows XP Professional SP3 (32-bit)
- Sun VirtualBox 3.1.8 r61349

4.2. Absolute loading time of a boot-time kernel module

In order to measure the actual tick count that is used to generate the final form of the __security_cookie value, several types of experiments were performed, targeting two default Windows drivers - win32k.sys and wanarp.sys, and two different environments (regular PC and virtual machine).

Module	win32k.sys (PC)	win32k.sys (VM)	wanarp.sys (PC)	wanarp.sys (VM)
Lowest tick value	857	428	938	417
Highest tick value	1028	466	974	570
Spread	171	38	36	153
Most frequent value	960	436	950	453
Top value occurrence #	48	129	225	71
Total samples	1300	1793	2016	1635
Success rate	3,69%	7,19%	11,16%	4,34%

Table 3. A summary of the absolute loading time of two default Windows device drivers, on different hardware platforms.

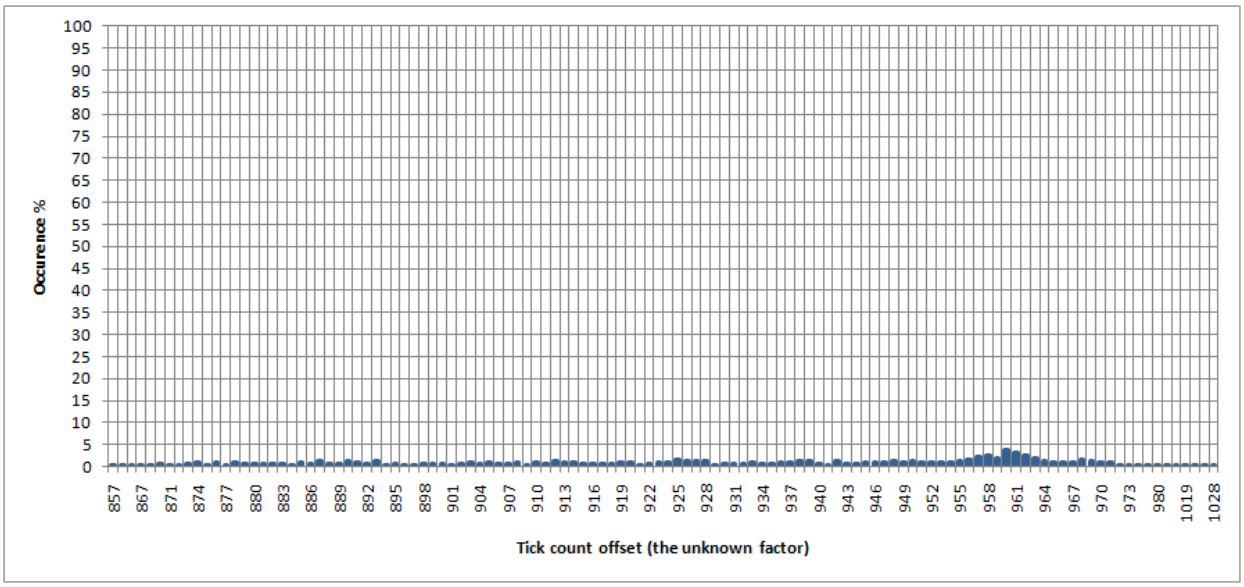

Chart 7. Absolute loading time of win32k.sys, regular PC.

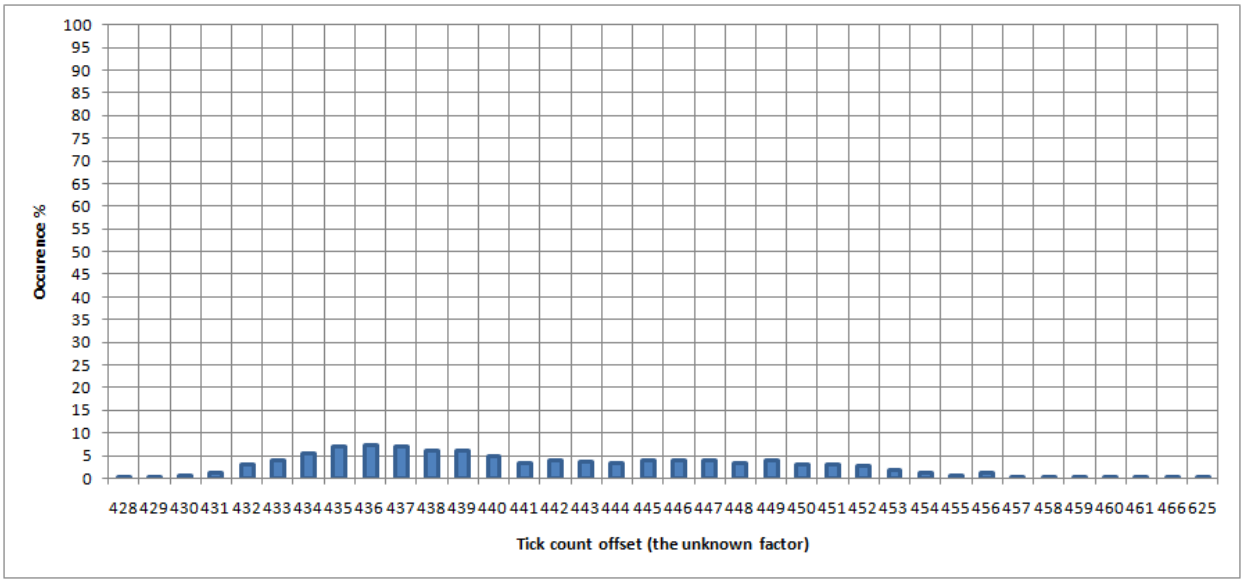

Chart 8. Absolute loading time of win32k.sys, virtual machine.

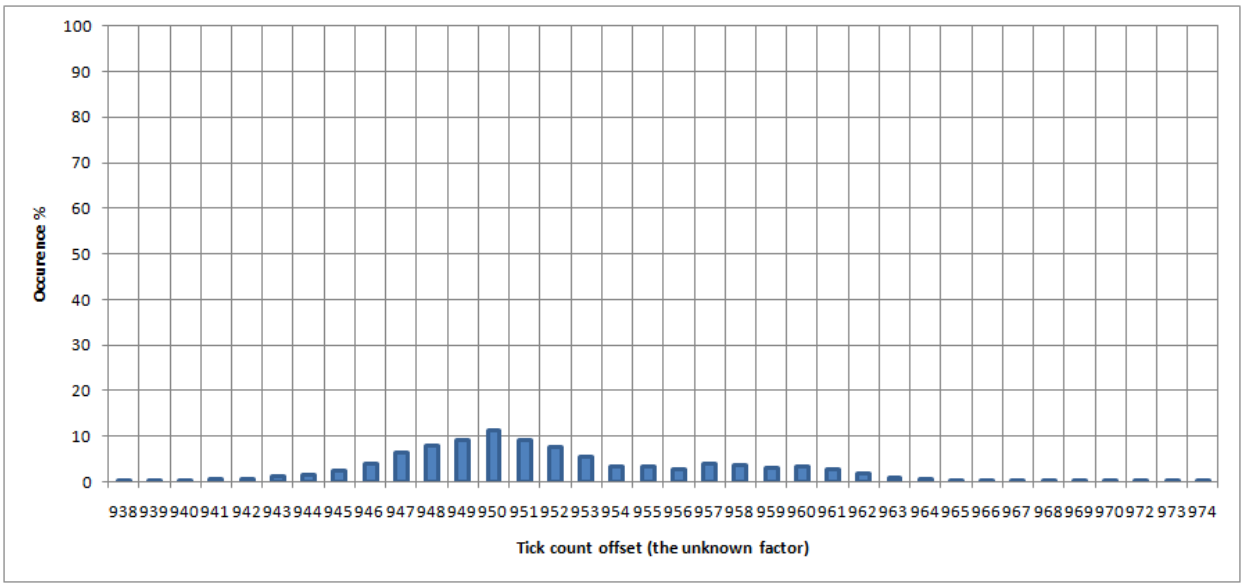

Chart 9. Absolute loading time of wanarp.sys, regular PC.

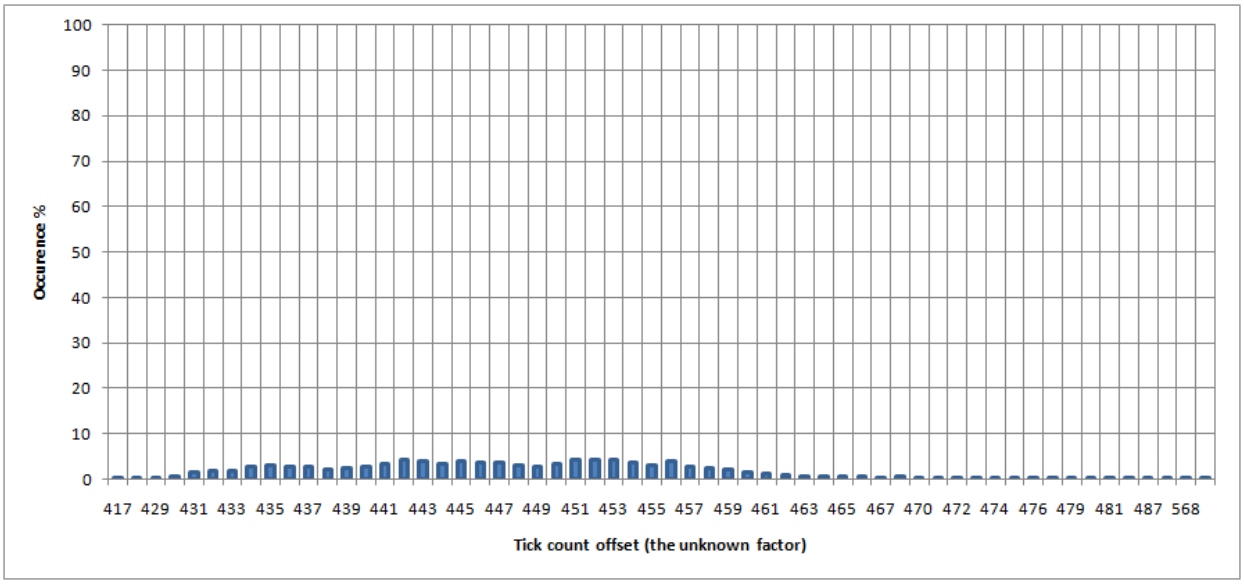

Chart 10. Absolute loading time of wanarp.sys, virtual machine.

4.3. Process-relative loading time of a manual-load driver

Module	win32k.sys (PC)	win32k.sys (VM)
Lowest offset value	91	47
Highest offset value	170	68
Spread	79	21
Most frequent value	105	53
Top value occurrence #	544	274
Total samples	1300	1793
Success rate	41,84%	15,28%

Table 4. A summary of the process-relative offsets, observed in win32k.sys, on a regular PC and a virtual machine.

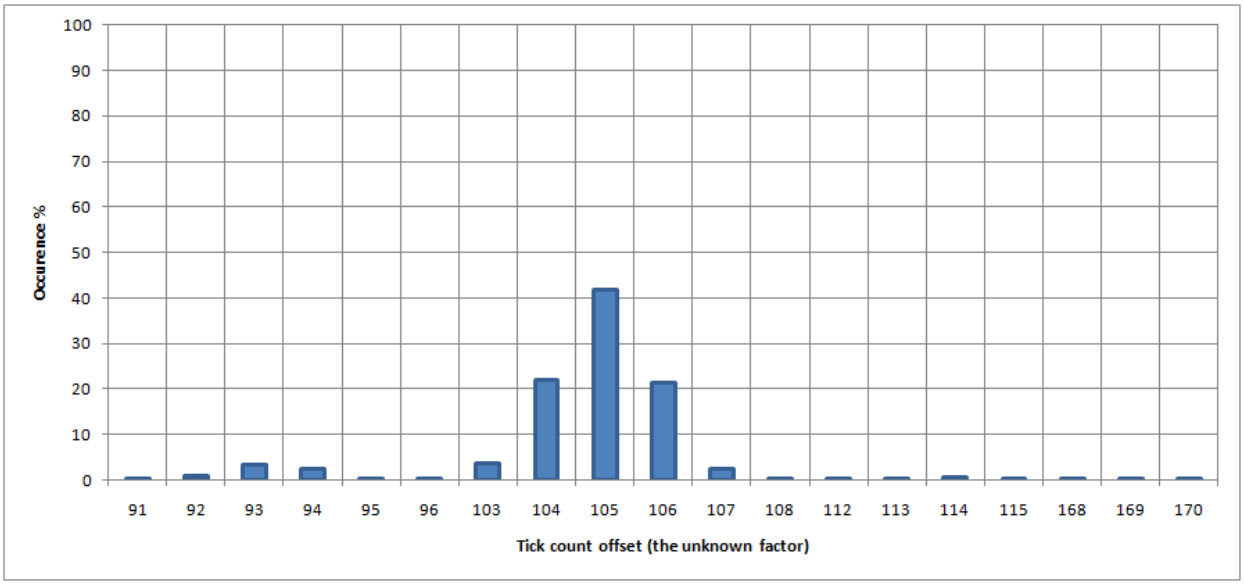

Chart 11. Process-relative loading time of win32k.sys, regular PC.

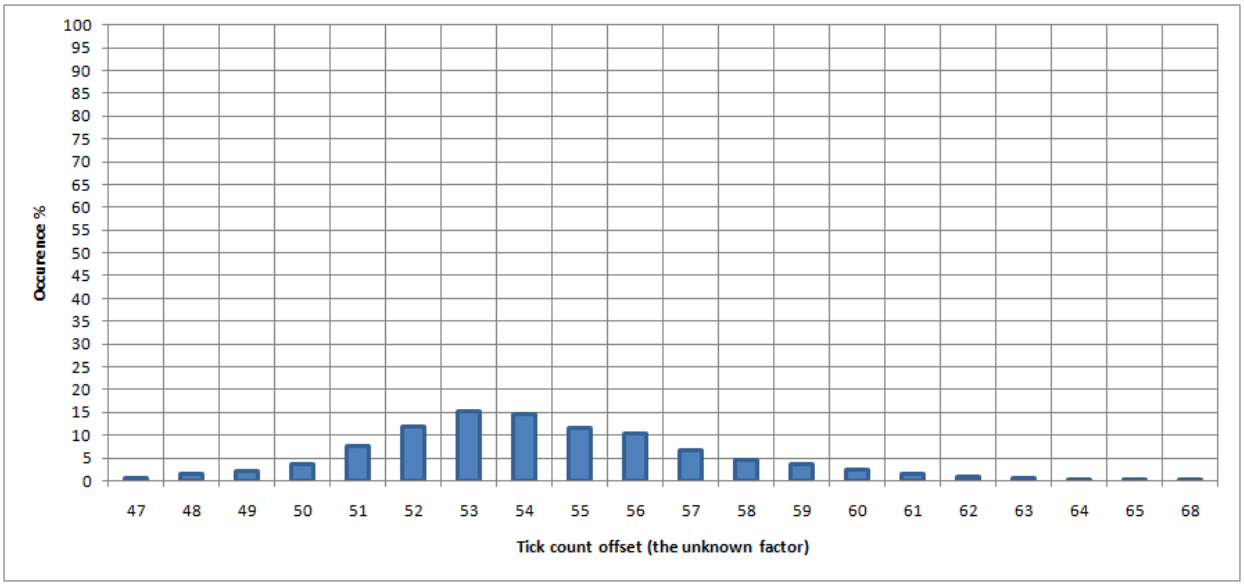

Chart 12. Process-relative loading time of win32k.sys, virtual machine.

4.4. Symlink-relative loading time of a *public* driver

Module	win32k.sys (PC)	win32k.sys (VM)	wanarp.sys (PC)	wanarp.sys (VM)
Lowest ticks value	5	23	6	19
Highest ticks value	71	39	71	37
Spread	66	19	65	18
Most frequent value	6	30	7	29
Top value occurrences	605	394	1162	221
Total samples count	1300	1793	2016	1635
Success rate	46,53%	21,97%	57,63%	13,51%

Table 5. A summary of the symlink offsets, observed in win32k.sys and wanarp.sys, on a regular PC and a virtual machine.

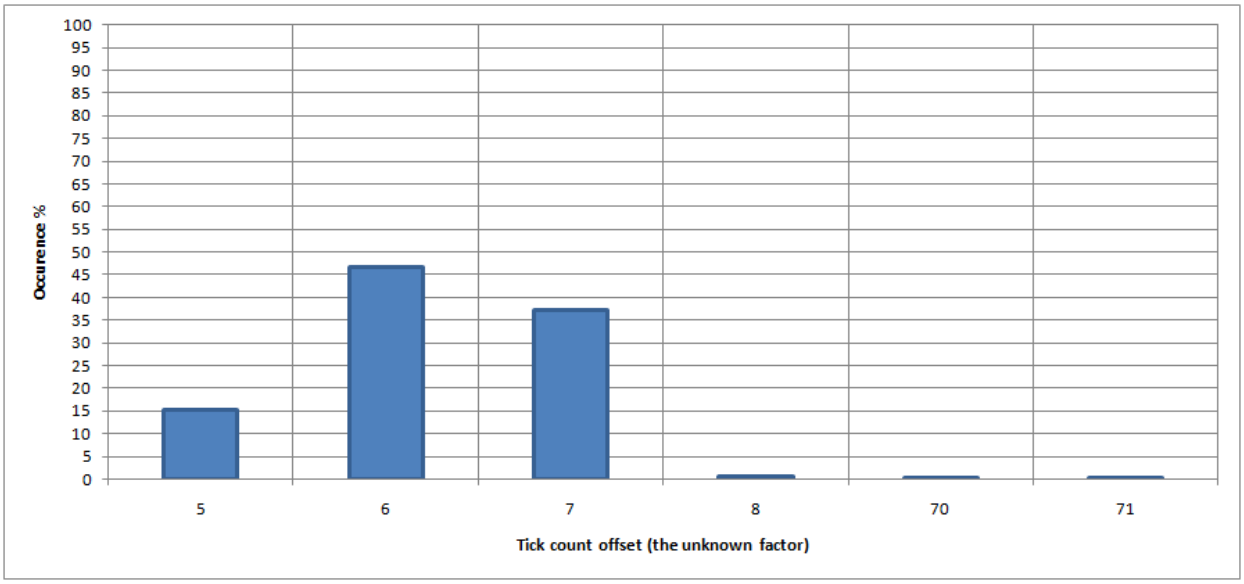

Chart 13. Symlink-relative loading time of win32k.sys, regular PC.

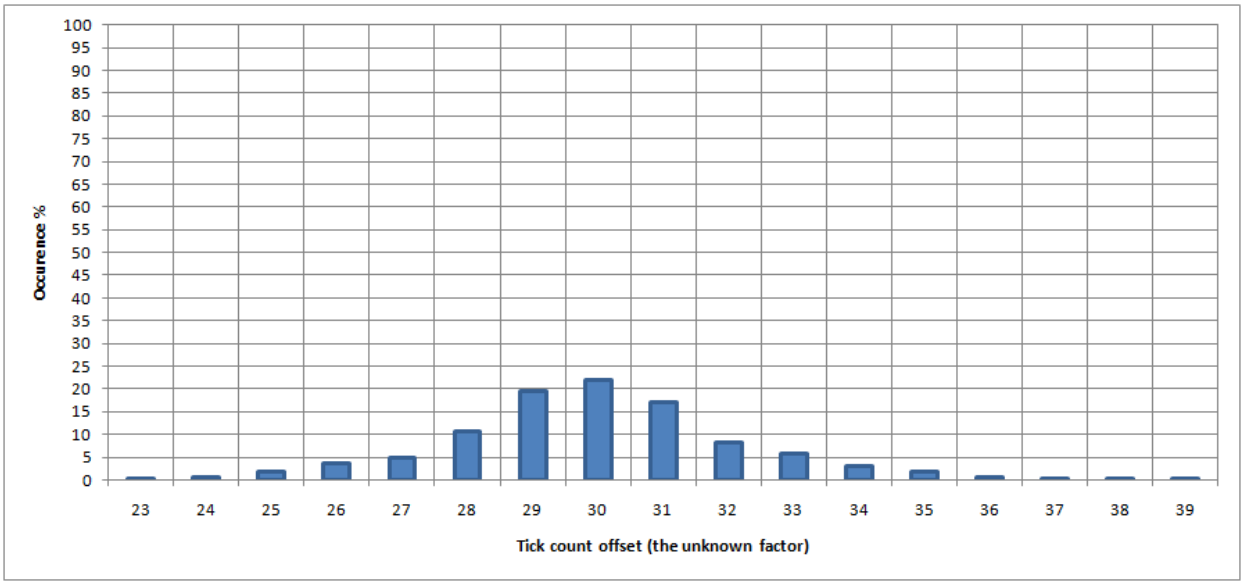

Chart 14. Symlink-relative loading time of win32k.sys, virtual machine.

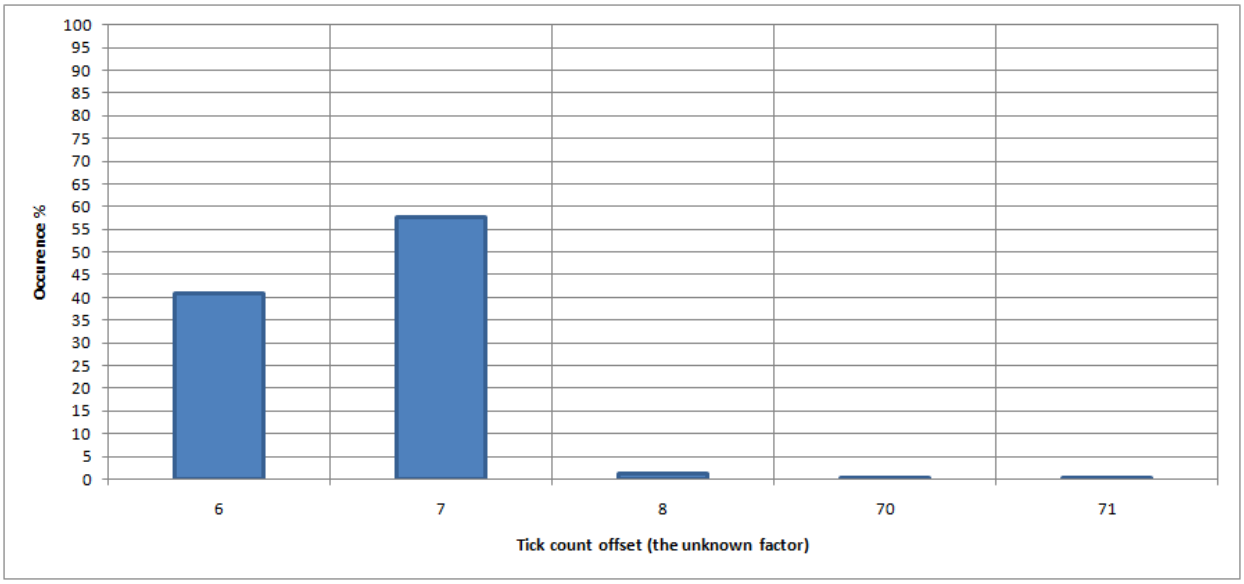

Chart 15. Symlink-relative loading time of wanarp.sys, regular PC.

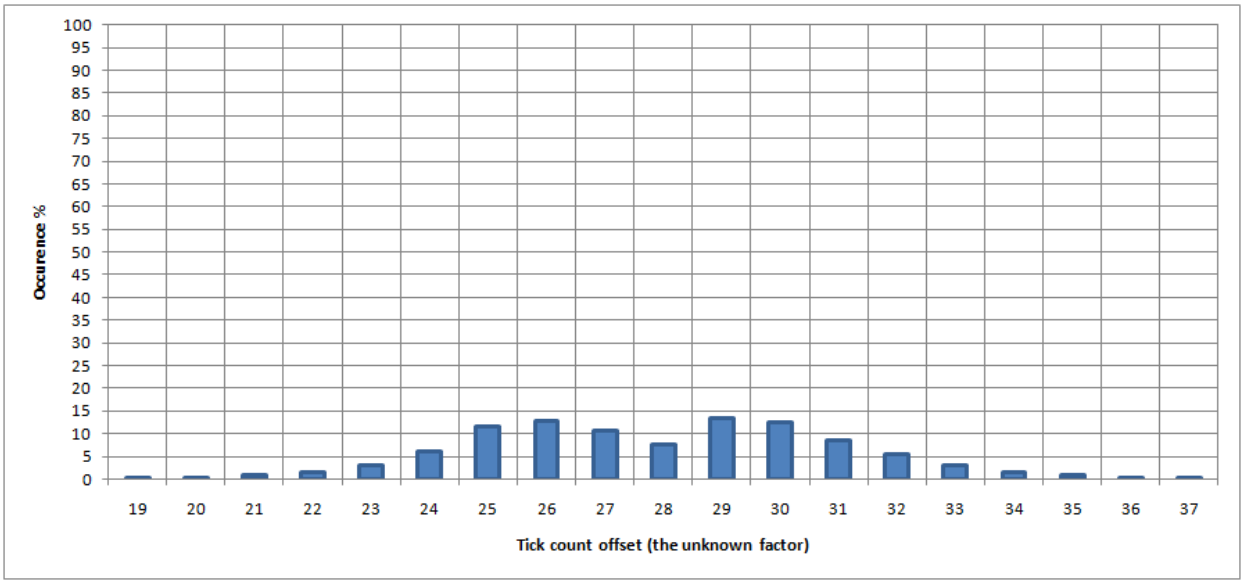

Chart 16. Symlink-relative loading time of wanarp.sys, virtual machine.

5. Improvements

As some of the experimental results presented in Chapter 4 indicate, the current implementation of the kernel GS cookies provides - depending on the particular attack circumstances - from one to four bits of true randomness. The authors believe that the actual protection level bar could be raised by extending the number of high-entropy bits to 16 (for Windows XP and 2003), 32 (for Windows Vista, 2008 and 7) or 48 (x86-64 platforms, in general). The following section outlines some of the potential solutions, addressing the cookie weaknesses presented in the paper.

- When generating the cookie value, make use of a system-wide entropy container. Such a container could be implemented using any named object (resource), such as a named device (e.g. \\.\Random), an emulated registry key (HKLM\System\Random), or any other type of object that the developers find reasonable. The executable image implementing the container would be responsible for collecting truly random information from the surrounding environment (certain users' actions, hardware lags, etc), and would obviously have to be loaded very early during the system boot-up process.

- Make use of higher-resolution timers, which are present in the system. The exact timers within our interest would include the *Performance Counter* (NtQueryPerformanceCounter[7]), or *Time Stamp Counter*. The first timer is being actively used in the user-mode GS protection, and is currently proved to provide about 17 bits of true entropy (according to [2]). However, one should keep in mind not to make use of any system mechanisms, which might indirectly rely on the system ticks' information (such as the GetTickCount Windows API), since no additional entropy would be added to the GS cookie; instead, only the numerical values of the stack canary would change.

- Take advantage of hardware-assisted RNGs or PRNGs.

- Seed the __security_cookie value with a more unpredictable value at the time of placing it on the stack. Instead of using the stack pointer, the system could employ a value, which simply cannot be known by a user-mode application (e.g. a kernel-mode pointer, such as the current *ethread* virtual address). Again, one problem related to the concept is that the positioning of driver-, thread- and process-related information is very deterministic, in the context of images loaded early during the system startup.

- Send random data requests over the local network / internet, previously installing a separate, dedicated machine, running for solely one purpose - supplying high-entropy numbers to other machines in the network. This might, in turn, have a relevant impact on the driver's loading efficiency, and would be prone to reliability problems related to the machine's network.

- Continuously modify the __security_cookie value at system run-time, between pseudo-random time intervals, based on the current machine state. By doing so, an attacker

would have to know the entire system / machine execution path, which (most likely) he doesn't. Such a solution could be implemented in a great variety of ways; e.g. by using the *rdtsc* instruction / performance counter as a pseudo-rand time interval.

In order to avoid potential crashes in certain situations, when the __security_cookie global changes between function's beginning and end, the correct value of the canary would have to be saved locally (e.g. on the stack on a lower address than all the local variables, or better, in a separate thread-specific area), on a per function-call basis, which may introduce further problems.

Although the authors are aware of the fact that some of the solutions might be prone to certain weaknesses under specific conditions, each provides a considerably higher protection level compared to the current number of *random* bits found in the GS cookie (approximated as 1 to 4).

Furthermore, the authors believe that a cookie containing two zero-bytes can render attacks against vulnerabilities involving overflowing a buffer in a zero-terminated string copy operation (e.g. using wcscpy) unexploitable using the standard return-address overwrite methods. Such bytes are present in the Windows XP and 2003 32-bit and all the 64-bit versions of Windows cookies, however, the zero-bytes were removed in 32-bit Vista and newer 32-bit Windows releases. In such case resizing the cookie variable from 32 to 64 bits might be worth doing.

6. Future work

Although a considerable amount of work has been already performed by the authors, accomplishing a decent level of efficacy, there are still numerous fields, which might be potentially improved, or should be further investigated. The most important of these issues are listed below:

- Searching for other (more accurate) points of reference (in the form of events, which have a known time of occurrence). For example, none of the techniques outlined in this paper can be applied to attack a device driver which doesn't provide a public interface to user-mode applications and is loaded dynamically by a ring-3 process upon a certain external event by a boot-time process (such as a user-mode *service*).

- Effectively predicting the security cookie value of the NT kernel image (ntoskrnl.exe), which is based, inter alia, on the RDTSC instruction output.

- Remote attacks. As far as the authors are concerned, a majority of factors considered *trivial* in Local EoP attacks, might be impossible to predict without local access to the victim machine. The only potential attack vector known to the authors, would rely on sniffing packets with certain types of information, which could reveal the absolute boot-

time of the machine under attack (or simply its up time), and the hardware-related specs; then, based on the collected information, trying to blindly guess the GS cookie value, as presented in section 4.1 - by approximating the tick count at the time of cookie initialization, and assuming a particular virtual address of the global __security_cookie variable.

7. Vendor timeline

6 December 2010: Initial e-mail to Microsoft informing that our research indicates that the ring-0 driver cookies are predictable.

6 December 2010: Initial vendor response, confirming reception.

8 December 2010: Second vendor response. Vendor was aware of the low entropy of the cookies and agrees that our approach is reasonable. Vendor statest that there are no plans for updating the mechanism in current versions of Windows, but will be considering it for future versions. Vendor did not request the paper to be released later than the authors originally planned.

8. Conclusion

During the last couple of years, GS cookies implemented by Microsoft compilers, together with other techniques such as stack variable reordering, have been claimed an ultimate weapon against successful attacks, relying on stack-based vulnerabilities. Nowadays, most attention is usually focused around the security of user-mode applications (e.g. internet browsers and related components), due to the fact that any security flaws found within these can lead to a massive amount of computer infections. However, one should keep in mind that the availability and exploitability of Local Escalation of Privilege attacks also affects the entire IT field, especially in the context of server machines, sharing a single operating system amongst multiple user accounts.

The techniques presented in this paper aimed to show, how otherwise non-exploitable vulnerabilities, continuously found within numerous device drivers running on Windows, can be used to compromise a victim's machine with very high success probability. What is more, the paper is expected to completely change the way people look at certain - past and upcoming - kernel *stack-based* buffer overflow flaws - from *Denial of Service conditions*, to *Exploitable with 5-70% success rate*, depending on the executable image under attack. Given the nature of the current GS cookies' implementation, resolving the issue would require all of the vendors to re-compile their device drivers, to take advantage of a new, improved solution. For obvious reasons, such a world-wide task is never going to fully succeed.

The authors strongly believe that other techniques might exist, improving the already accomplished level of accuracy - any researcher is highly encouraged to perform further research in this field; the best situation would be to defeat the protection scheme completely, thus automatically making all of the Windows kernel-mode stack vulnerabilities exploitable into *ring-0* code execution. Moreover, little is still known about how the outlined techniques could be applied remotely - most of the factors taken into account while attacking the machine as a local user, are unable to be obtained without local access to the computer in consideration.

In order to address the weaknesses of each GS cookie entropy source, one or more potential solutions have been described. The authors are aware of the fact that some of the proposed techniques can be attacked one way, or another; although, each and every single concept presents a considerably higher entropy level, reducing the cookie prediction probability from 50%, to numbers that are only satisfactory in a lab environment, or not even as good. Additionally, we especially approve one of the concepts presented by skape in his paper - making the operating system responsible for implementing a common, standardised interface, which would then be used by all of the secure device drivers. By doing so, Microsoft would be able to easily keep track of the protection level provided by the mechanism, and release potential improvements, which would immediately affect all of the kernel modules, no matter whether the original vendor knows about the threat, or not - or whether he still exists, in the first place.

Happy vulnerability hunting!

Appendix A – Microsoft Windows win32k.sys RtlQueryRegistryValues vulnerability exploitation on Windows XP SP3 (32-bit).

Interestingly, the overall concept of analyzing kernel-mode GS cookie implementation was born, while considering possible exploitation vectors of the CVE-2010-4192 vulnerability, publicly disclosed on 2010-11-29 by a Chinese hacker *noobpwnftw*. In a short security advisory placed on the CodeProject[8] website, the discoverer surprisingly presented some of the flawed implementation basics, and possible exploitation paths.

Additionally, the article author presented snippets of a Proof of Concept exploit, claimed to support both Windows Vista and Windows 7. After a quick investigation, it turned out that the vulnerable routine on either of the two supported platforms is not GS-protected. Therefore, the issue becomes a typical stack-based buffer overflow, which can be trivially exploited, by just replacing the original return address with an attacker-controlled value.

However, an older version of the operating system – Windows XP SP3 – is indeed protected by a regular GS cookie, implemented in the manner described in detail in the previous sections. Listings 19 and 20 present the vulnerable routine prologue and epilogue.

```
mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 42Ch
mov     eax, __security_cookie
push    ebx
mov     [ebp-4], eax
```

Listing 19. The flawed function's entry point.

```
mov     ecx, [ebp-4]
pop     edi
pop     esi
pop     ebx
call    __security_check_cookie
leave
retn    8
```

Listing 20. The flawed win32k.sys function's epilogue.

As shown, the win32k!__security_cookie value is placed directly before the stack frame at the routine beginning, and correctly verified before returning to the original caller. Although it wasn't

explicitly explained, the authors suspect that the actual reason of not releasing a reliable exploit for the Windows XP platform was the stack protection itself.

After performing a number of experimental tests, measuring the time and tick deltas between various system events, the authors developed a stable exploit for the vulnerability discussed in this chapter, in order to present the practical usability of the techniques presented in this paper. Although the exploit source code is not going to be published, a video recording – presenting successful exploitation of the Windows XP vulnerability – has been created, and can be downloaded from: http://j00ru.vexillium.org/dump/CVE-2010-4398/exploit.avi.

```
loc_BF87EE99:
       push   dword ptr [ebp+hObject1] ; Handle
       call   edi ; ZwClose
       jmp    loc_BF87F0AB

loc_BF87EEA6:
       push   dword ptr [ebp+hObject2] ; Handle
       call   edi ; ZwClose
       jmp    loc_BF87F0B7
```

Listing 19. Code responsible for dereferencing active object handles.

Interestingly, although making use of the security flaw becomes non-trivial without performing a GS-prediction attack, as opposed to the modern Windows editions, it is believed that it can be still reliably exploited. The goal can be potentially achieved, thanks to the following code, executed right before trying to leave the function - see Listing 19 and 20.

```
if(hObject1 != NULL) ZwClose(hObject1);
if(hObject2 != NULL) ZwClose(hObject2);
```

Listing 20. High-level language representation of assembly code presented in Listing 19.

Thanks to the function stack layout, both handle values are controlled by the attacker, at the time of the ZwClose kernel API calls. In this situation, a potential attacker could make the following decisions:

1. Set both values to NULL's, so that no handle gets dereferenced,
2. Fill one of the variables with an invalid handle value, or a handle with the NOT_CLOSABLE flag set,
3. Fill the variables with valid user-mode handles,
4. Fill the variables with valid kernel-mode handles.

The question is – what particular benefits could be accomplished by an attacker, by performing each action listed below.

1. Filling both *hObject1* and *hObject2* with zeros makes it possible to omit the conditional code blocks. It is a good idea, when one doesn't need to care about the GS cookie, and goes straight into overwriting the return address.

2. Filling either *hObject1* or *hObject2* with an invalid or protected handle results in a KERNEL_INVALID_HANDLE bug check (Blue Screen of Death). By doing so, the attacker loses the ability to run arbitrary code in kernel-mode.

3. Filling the variables with typical handles, created from within user-mode doesn't bring any particular benefits, as the same result could be simply achieved by calling the CloseHandle API from within ring-3.

   The only, truly beneficial choice, is to close a kernel-mode (thus system-wide) handle. By doing so, the attacker could dereference a chosen object one or more times; when the *ReferenceCount* number assigned to the chosen object reached zero, the object pool allocation would be freed, and possibly further reused to store other types of unrelated data. In other words, an attacker is able to trigger typical *use-after-free* conditions using one or more kernel objects. Seemingly, the vulnerability is less trivial, but still possible to be taken advantage of by a determined attacker.

References

[1] National Vulnerability Database. *CVE-2010-4398*.
http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-4398
[2] skape. *Reducing the effective entropy of GS cookies*. Uninformed vol. 7, 2007.
http://uninformed.org/?v=7&a=2&t=sumry
[3] National Vulnerability Database. *CVE-2009-1126*.
http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1126
[4] Sven B. Schreiber, Tomasz Nowak. *Undocumented functions of NTDLL - NtQuerySystemInformation*.
http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/System%20Information/NtQuerySystemInformation.html
[5] Nirsoft, struct KUSER_SHARED_DATA.
http://www.nirsoft.net/kernel_struct/vista/KUSER_SHARED_DATA.html
[6] *Undocumented function of NTDLL* - NtCreateSymbolicLinkObject.
http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Symbolic%20Link/NtCreateSymbolicLinkObject.html
[7] MSDN - NtQueryPerformanceCounter Function
http://msdn.microsoft.com/en-us/library/bb432384%28VS.85%29.aspx
[8] CodeProject, http://www.codeproject.com/

Thanks

We would like to thank the following people - as well as every single anonymous participant - for taking part in the public survey (http://j00ru.vexillium.org/ticks/), and providing empirical information about the GetSystemTimeAdjustment Windows API output (in ascii-alphabetical order):

0vercl0k, 3mpty, 9x19, =BTK=, ADS, Ace Pace, AdiKX, Agares, Ahmed, Arteq, Banana, Bartoo, Claudiu Francu, D0han, D3LLF, Edi, Fanael, Furio v2.0, FurioSan, G. Geshev, GOJU, Gabriel Gonzalez, Galcia, Hoo, Horadrim, Icewall, Idalgo, Ivanlef0u, Jacosz, Jurgi Filodendryta, KORraN, Karton, Kele, Kicaj, Kiro, Krystian Bigaj, KrzaQ, Lewy, Lipa, M4R10, MBu, MDobak, MSM, MagicMac, Makdaam, MarLo, MateuszR, Mawekl, Michal Z., Naidmer, Netrix, Pafi, Paul, Piochu, PiotrB, Qyon, Radom, ReWolf, Redford, Reg, Riddler, Rolek, Rolex-, RomeoKnight, S0obi, Sayane, Sean de Regge, SparK, Samlis Coldwind, TaPiOn, Talv, Thomas, Trol, Tyler Oderkirk, Unknow, Unreal, VGT, Vineel Kumar Reddy Kovvuri, Wawi, Xylitol, YouKnowit, ZaQ32, Zeux, acz, adam_i, ajcek, ajgon, anjw, bartek_sekula, berials, bidek, blejz, bobbobson, bruce dang, cLs, conio, cyriel, d15ea5e, dD3s, dextero, dikamilo, dzeta, en, ergo, eustachy86, faramir, faust1002, ged_ ' "><, globi, grable, h0wl, hazy77, hxv, impulse9, jacekowski, jarekps, jerrythemouse, justhelping :), kaz1007, kernelpool, kij, kosmito, koziolek, krajek, kravietz, kutar, logan, lukasz anwajler, lsalomon, lzsk, m, mINA87, magu, malpka, mariusz, memek, milordi, misiekzap, mjuad, mmm, mrx1, msi, mt3o, muzzy, nek, nezumi, nickname, none, nonek, nuivall, olewales, oshogbo, p____h, pashok, pawelu, pawlos, phil hamer, pkh, pozdrawiamtegouzytkownika, ppkt, rad, renno, ryniek, s4tan, shaql, suN8Hclf, superhero01, tanatos, taviso, tomekby, toxicbeaver, us3r, vashpan, vndctv, vrx, witosuaw, xmoss, xyz69, yabba, yourand, zaak, zakrzak, zarcel.

Furthermore, we would also like to credit the valuable help of the article reviewers: Unavowed, Tavis Ormandy, Ben Hawkes, Marc-Antoine Ruel, Carlos Pizano, Matt Miller, and deus. Thank you!

Disclaimer

The views expressed here are mine alone and not those of my employer.

Gynvael Coldwind