PREV NEXT

# Chapter 3: Buffer Overrun Defenses

## Overview

Windows Vista adds several mitigations against buffer overruns and improves many of the defenses that were already in place in Windows XP SP2 and Windows Server 2003 SP1. Before we start discussing the new features, we need to make several important points. First, you still have to write solid code. As we said in *Writing Secure Code*, all of these defenses are like the seatbelts in your car–none of them are guaranteed to save you from your mistakes, and if you ever have to use them, you're not having a good day (Howard and LeBlanc 2003)! Like the seat belts (and air bags, and antilock brakes, etc.), no one can guarantee that you'll never have a bad day, but if you do, it's always a good thing if you have something to help keep it from getting worse. Our friend Jon Pincus, a researcher in Microsoft Research, expresses this principle very well (Jon Pincus, personal communication):

> *For any given mitigation, or set of mitigations, a sufficiently complex attack can overcome the mitigations. For any given effective mitigation, there exists some set of attacks which are stopped completely.*

If you come up with some completely contrived piece of attack code with a dozen different flaws that manages to execute arbitrary code despite all the defenses we added, you haven't discovered anything new or exciting. However, if you take a careful look at what we've done and find room for improvement, please let us know–we're always interested in making our platform more secure.

Figure 3-1 shows a graphic way to envision the problem.

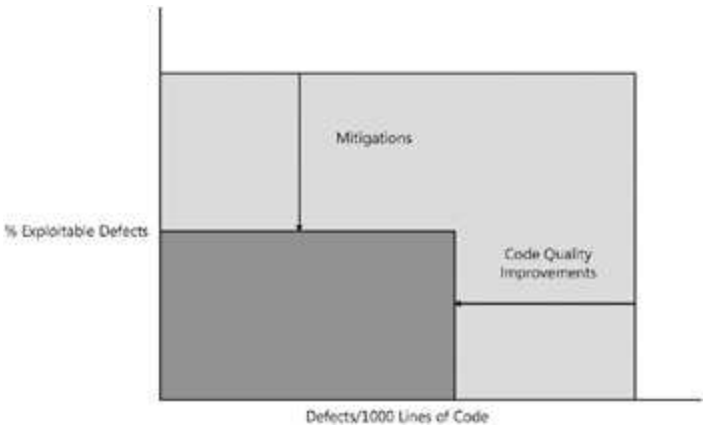


Figure 3-1: Diagram showing effect of mitigations and code improvement.

Any piece of software will have between 5 and 50 defects per 1,000 lines of code (kloc, which is pronounced k-loc). How to properly count software defects has been the subject of much study and debate, and this book can't possibly address the topic thoroughly enough. A good reference on the topic is "Bugs or Defects?" (Humphrey 1999). The numbers quoted here are reasonably close to those reported by various researchers. For example, Humphrey cites figures from three different projects from the Jet Propulsion Laboratory. Despite rigorous development practices, from 6.5 to 9 defects/kloc were found after the programs were compiled, unit tested and integration tested. An average developer with average development practices will make 50 to 100 mistakes/kloc, many of which may be caught by

compiler warnings and testing. Code review, education, experience, and static analysis tools can drive the actual count to the lower portion of the range. Fuzzing, or better yet, targeted test harnesses to validate the code, will also drive down the defect count. Unfortunately, you aren't likely to drive defects to zero for any nontrivial application, and the closer to zero defects you try to get, the more expensive the code becomes to create, so it might be best to just accept that there will be the occasional error.

Out of the defects that do exist in the code, some will be exploitable. Even the most trivial defenses can eliminate some attacks. For example, an off-by-one overwrite of a stack buffer used to be typically exploitable, but with the introduction of stack protection, these are now stopped. The techniques we discuss in this chapter are all efforts to ensure that some of the errors that escape us don't result in an exploitable condition.

The second point we'd like to make is that Microsoft didn't actually pioneer much of this, and this text does not claim that's the case. What we are doing is delivering these features in an extremely widely used, general-purpose operating system while doing our best to maintain backward compatibility and good performance with an extremely wide range of applications.

PREV PREV NEXT NEXT

# ASLR

Address Space Layout Randomization (ASLR) is a technique that makes it more difficult for an exploit to locate system APIs, which in turn makes it harder to leverage system APIs in exploits to run arbitrary code. The approach involves randomizing where system libraries are loaded, all of the starting points of the stack, and the starting points of the heaps. On a Windows XP system, all of these are known to an attacker and may vary somewhat depending on the service pack level of the system. There are even references available on the Web that document this information by version and service pack level.

ASLR includes randomizing all of the following elements of a running program:

- **Image randomization** The addresses where the executable and DLLs are loaded varies
- **Stack randomization** The starting address for each thread's stack varies
- **Heap randomization** The base address for heap allocations will also vary

One common attack is to call *LoadLibrary* to force the application to load a DLL. The assembly needed to do this is very trivial, and amounts to:

```
Push location of path buffer
Call location of LoadLibrary
```

Thus, if the attacker can write a path into a buffer with a known location–which need not involve an overflow–and then redirect execution into a spot where a very small amount of shell code is located, your application will load a library, possibly across the network, and invoke DllMain. As an attacker, this is a handy technique because a very complex attack can be written in a high-level language, while the minimum size of the shell code needed becomes very small.

ASLR can thwart such an attack by eliminating preconditions needed by the attacker. The first problem the attacker encounters is that the address to be jumped into needs to be known, although not exactly. If an attacker has a large buffer to work with, writing a large number of NOP (no-op) instructions into a buffer creates a NOP slide, and the instruction pointer only needs to be redirected to somewhere in the slide. Next, the address of the library path needs to be known exactly, and so does the address to call the LoadLibrary function. If the starting point of the stack isn't exactly the same at all times, then finding the buffer with the path is more difficult and so is figuring out where to find the shell code. Finally, if LoadLibrary can't be counted on to be in the same place at all times, this will pose additional difficulties.

Here's an application you can use to take a look at how ASLR and stack randomization work on your system:

```
#include <windows.h>
#include <stdio.h>
unsigned long g_GlobalVar = 0;

void foo( void )
{
   printf( "Address of function foo = %p\n", foo );
   g_GlobalVar++;
}

int main( int argc, char* argv[] )
{
   HMODULE hMod = LoadLibrary( L"Kernel32.dll" );
   // Note – this is for release builds
   HMODULE hModMsVc = LoadLibrary( L"MSVCR80.dll" );
   char StackBuffer[256];

   void* pvAddress = GetProcAddress(hMod, "LoadLibraryW");
   printf( "Kernel32 loaded at %p\n", hMod );
   printf( "Address of LoadLibrary = %p\n", pvAddress );
   printf( "Address of main = %p\n", main );

   pvAddress = GetProcAddress( hModMsVc, "system" );
   printf( "MSVCR80.dll loaded at %p\n", hModMsVc );
   printf( "Address of system function = %p\n", pvAddress );

   foo();
   printf( "Address of g_GlobalVar = %p\n", &g_GlobalVar );
   printf( "Address of StackBuffer = %p\n", StackBuffer );

   if( hMod ) FreeLibrary( hMod );
   if( hModMsVc ) FreeLibrary( hModMsVc );

   return 0;
}
```

The test app was written using the retail shipping version of Visual Studio 2005, Service Pack 1, and the */dynamicbase* linker option, released in Visual Studio 2005, Service Pack 1. Here we find that LoadLibrary shows up in different places across reboots of the system:

```
First test:
Kernel32 loaded at 77AC0000
Address of LoadLibrary = 77B01E7D
```

```
Second test:
Kernel32 loaded at 77160000
Address of LoadLibrary = 771A1E7D
```

What is also of interest is that the location of the C runtime library doesn't change–it loads at address 0x78130000 across both reboots. The C runtime library this application is dynamically linked with was linked with an older linker that did not have the */dynamicbase* capability, and it won't utilize image randomization. If I'd gone to the trouble to load the entire Visual Studio update with the new C run-time DLL on my system, we wouldn't see this problem, but it does show us something to be careful of–many applications are built using third-party components, and if all of the libraries your vendor supplies aren't also rebuilt using the new flag, then they won't be relocated and will counteract many of the benefits of ASLR. If it's a library that comes with source, be sure to rebuild it yourself to get the benefits of ASLR, although you must also be careful to test and ensure that the change doesn't break something. Most DLLs won't be disturbed by ASLR, but this opt-in behavior is designed to keep applications from randomly breaking when upgrading to Vista.

## Limitations of ASLR

ASLR is performed systemwide, once per reboot. One note is that if all of the processes using a particular DLL unload ASLR then it would be loaded in a random place on the next load, but many of the system DLLs are always loaded by several processes and only get randomized at the next reboot. If you're dealing with a local attack, an attacker could easily determine addresses, and then launch the attack. This feature helps inhibit network-based attacks, especially worms. If an application has an information disclosure vulnerability (where it reveals exception details to an attacker) or has a format string vulnerability (which would allow the values on the stack to be read), it may be possible for an attacker to learn the memory locations needed to overcome this mitigation. An attacker-controller read might also be used to place information needed for an attack into either a known location or a location with a known offset. Another implication is that if the network service restarts itself on failure, the attacker now has more chances to find where to call the system API, which is why we recommend that services be configured to restart automatically a small number of times. To reduce virtual address space fragmentation, the library is only relocated across 256 different possible load addresses; note that the randomization is in the second-most significant byte of the address.

## Performance and Compatibility Implications

One item that comes to mind immediately when thinking about relocating DLLs is that software developers sometimes spend considerable time figuring out just where to set the base address of a DLL, because when two DLLs try to load into the same spot (or have overlapping ranges), the last one to load is relocated to a different address. The relocation process can be very expensive. Every fixed address in the entire DLL needs to be changed to reflect the new starting point, so anyone concerned with load performance should try very hard to prevent relocation. If relocation is such a bad thing, then, what's the impact of randomly relocating everything?

The way ASLR is implemented deals with the performance concerns by delaying fixups until that page of the DLL is loaded into memory. For example, Kernel32.dll exports 1,202 functions, occupying about 160 pages of memory. A typical console application might only use a dozen or so functions exported by Kernel32.dll and would only need to fix up the pages required to load those functions.

A second performance enhancement comes because the DLLs don't set their own addresses any longer; Vista now packs them in with as little slack space between loaded DLLs as possible. While 2 gigabytes of address space used to seem like a lot, a modern application can often use as much memory as

possible. With all the DLLs being loaded into a relatively contiguous space, there's effectively more memory available for other purposes. Having the memory addresses relatively close to one another can sometimes help the processor cache perform more effectively.

As implemented in Windows Vista, the compatibility implications of ASLR aren't much of an issue. Microsoft Office 2007 is a very large code base, and has a wide variety of programming practices and techniques. When Office enabled ASLR and ran a test pass, we didn't find any substantial performance degradation, and no regressions. Although there may be things a developer could do to break themselves if ASLR were enabled, like hard-code function addresses, if all of Office can turn this on with remarkably few problems, most apps should be in great shape.

PREV PREV NEXT NEXT

# Stack Randomization

Stack randomization changes the base address for every thread, which makes it more difficult for an attacker to find a place to jump to within an application. For example, assume the attacker wants to make a call to *system*, but the C runtime DLL is loaded in a random location that your code also calls *system*. If the application code is loaded at predictable addresses, the attacker can then just jump to where you called *system* yourself. Obviously, a real attack would also include setting up arguments and buffers, and some serious mistakes would need to be made before an attacker could have this much control.

Using the sample program listed in the ASLR section, you get the results shown in Table 3-1.

Table 3-1: Effect of Stack Randomization on Application Addresses

Open table as spreadsheet

| | Original Value | After Restarting Application |
|---|---|---|
| Address of main | 00281020 | 00F41020 |
| Address of function foo | 00281000 | 00F41000 |
| Address of g_GlobalVar | 0028336C | 00F4336C |
| Address of StackBuffer | 001EFBC4 | 002EF8BC |

To get this protection, you have to use */dynamicbase* in your linker options. Note that even though the starting address of the executable command is randomized, the offset between the various code elements remains constant. In this example, the offset between the entry point of *main()* and *foo()* is 32 bytes (0x20). If you have any important information stored in global variables, these addresses will be randomized as well. While it is generally a bad practice to store things such as function pointers in global memory, if you do, this mitigation will make your code more difficult to attack. Encoded pointers, which are discussed in Chapter 9, Miscellaneous Defenses and Security Technologies should also be used.

Even though the address of the stack buffers has always been somewhat unpredictable in multithreaded applications, when using this mitigation we find that the location of the stack for the main thread is randomized; an interesting aspect to take note of is that the offset between the stack and the main module's code isn't fixed from one instance of the application to the next. If we've depriving the

attackers of fixed addresses, knowledge of offsets becomes even more important, and there's no longer a relationship between the stack addresses and the code addresses.

### Performance and Compatibility Implications

Unless you are doing something really unusual (and inadvisable), there shouldn't be any performance or compatibility issues because of randomized stack locations.

PREV PREV NEXT NEXT

# Heap Defenses

In the last several years, heap exploits have gone from exotic to typical. Countermeasures such as */GS* (explained later in this chapter) and better programming practices have made stack overruns less frequent, but the heap is actually easier in some ways to exploit. First, the size of the allocation is typically determined using an arithmetic calculation, and both computers and programmers can be very bad at math–although computers are faster and more predictable. For a more comprehensive look at how computers can mangle integer manipulation, take a look at "Integer Handling with the C++ SafeInt Class" (LeBlanc 2004), and "Another Look at the SafeInt Class" (LeBlanc 2005).

To make matters worse, common behaviors for a heap make unreliable exploits work more often. Let's say that you can get execution flow to jump into some spot in the heap, but you can't control exactly where it will go. One obvious attack is to put an enormous amount of data into the heap and use a very large NOP slide that causes execution of anything in the NOP slide to run down into your shell code. It isn't always possible to put very large amounts of data into the heap, so an alternate approach is a technique known as "heap spraying," which is a technique first noted in eEye's advisory about the IDA overflow (eEye 2001), which was well known because it was developed into the Code Red worm in 2001. Heap spraying involves writing a large number of copies of your shell code into the heap; depending on the details of the exploit, you would be much more likely to have the execution jump into the shell code because large allocations would typically end up in another location. If you're faced with executing legitimate allocations, none of the defenses discussed in the remainder of this section apply, but NX (No eXecute) will stop these attacks cold. NX will be covered in the next section.

Another problem exploits a very common programming flaw–dangling pointers and double-free conditions. Mike Marcelais of the Office Trustworthy Computing team came up with the following three common double-free conditions that can be dangerous (personal communication):

The first common double-free problem is when the heap structures in a way that an attacker can exploit. If the heap manager does not maintain the control data adjacent to the user data, this attack may not be possible. The base Windows heap in Windows Vista has also been hardened against this attack–more on this in a moment.

There is a pattern of *alloc(a), free(a), alloc(b), free(a), alloc(c)* all on the same address, as illustrated by the following code:

```
char* ptrA = new char[64];
// some code here
delete[] ptrA;
```

```
// now we need some more memory
char* ptrB = new char[64];
// Note that ptrB has the same value as ptrA
// Some more code, just to confuse things
// And now we make a mistake
delete[] ptrA;
// Oops! We just freed ptrB!
// Now we need more memory
char* ptrC = new char[64];
// ptrC will now be used to write memory that
// the code dealing with ptrB thinks is validated
```

The second condition is that efficient heap behavior reallocates recently freed memory that is the same size. In this case, the function that requested *alloc(b)* has a pointer to memory controlled by the function that called *alloc(c).* If the attacker can control the memory written into *alloc(c),* this situation is very exploitable, and there isn't anything an allocator can do to prevent this from causing problems. An obvious attack here is that function *b* validates user input and copies it to allocated memory. Function *c* then copies something else over the validated input, and then a subsequent operation is performed on the data in allocation *b.*

A third attack is a pattern of *alloc(a), free(a), alloc(b), use(a), free(a),* as shown here:

```
char* ptrA = new CFoo;
// Some code, and we then delete allocation A
delete[] ptrA;
// Now we need another allocation the same size
// Note that ptrA and ptrB point to the same memory
char* ptrB = new CFoo;
// Copy some data into ptrB
// Do something with ptrA, not knowing that ptrB has changed things
// If ptrA is a class, this includes calling the destructor
delete[] ptrA;
```

Note that if allocation *a* contains an object with a destructor, that's equivalent to using the memory. In this case, the code using *ptrB* is changing the contents of the buffer pointed to by *ptrA,* while *ptrA* is believed to have validated data. There is some potential for the usage of *ptrB* to attack *ptrA,* and in this case, the converse is true as well–the usage of *ptrA* could very easily cause the data kept in *ptrB* to become invalid.

One good programming practice prevents this type of error from being exploitable: always set pointers to null when freeing them, although this still won't help if there are multiple copies of the same pointer. Then the *use(a)* step will cause a null dereference crash; in the previous example, freeing or deleting a null pointer is benign–not only does it not crash, but the pattern of *alloc(a), free(a), alloc(b), free(a), alloc(c)* becomes non-exploitable as well, since calling delete on a null pointer doesn't do anything, and functions *b* and *c* would then have allocations in different places. The following C++ inline functions would help:

```
template < typename T >
void DeleteT( T*& tPtrRef )
{
   assert( tPtrRef != NULL );
   delete tPtrRef;
   tPtrRef = NULL;
}
```

```
// Use when allocation is new T[count]
template < typename T >
void DeleteTArray( T*& tPtrRef )
{
   assert( tPtrRef != NULL );
   delete[] tPtrRef;
   tPtrRef = NULL;
}
```

The reason these functions must be templatized is that for delete or delete[] to properly call object destructors, the object type must be known. A debugging assert will help catch and fix these conditions because, at run time, the second delete will be benign and without the assert, you wouldn't catch the double-free problem, which could be a symptom of other serious errors. While the heap manager can't protect you against the last two problems listed here, some of the other countermeasures might help. Our advice is that all double-free bugs should be fixed. Another good approach is to use smart pointer classes, although the behavior of the class must be understood before it is used.

In the case of a heap overrun, the effects depend on the heap manager being used. The default Windows heap places control data immediately before and after every allocation, and attackers can target both the control data and data kept on the heap in adjacent allocations. A number of researchers have found ways to attack the default Windows heap, and improvements in the Windows Vista heap have been numerous. Here's a partial list of recent improvements:

- **Checking validity of forward and back links** A free block has the address of the previous and next free blocks stored immediately after the block header. Basically, the value of the forward link turns into the value to write, and the value of the backward link is where to write the forward link value. This leads to an arbitrary 4 bytes (on a 32-bit system) being written anywhere in memory. The change is to check that the structures at those locations properly point back to where it started. This improvement was delivered in Windows XP SP2.

- **Block metadata randomization** Part of the block header is XOR'd with a random number, which makes determining the value to overwrite very difficult. The performance impact is small, but the benefits are large.

- **Entry integrity check** The previous 8-bit cookie has been repurposed to validate a larger part of the header. Another change with low performance impact, but it's difficult to attack.

- **Heap base randomization** This was mentioned earlier in the ASLR section.

- **Heap function pointer randomization** Function pointers used by the heap are encoded. This technique will be discussed at greater length in Chapter 10.

Basically, we must expect the abilities of the attackers to continue to improve, but we must also expect the defenders to continue to improve. Any list of attacks and countermeasures we can give you is probably going to be out of date by the time you read this book. What is important is knowing how to protect yourself and how to leverage the capabilities of the operating system to help protect your customers.

The first heap countermeasure that's new to Windows Vista is enabling application termination on heap corruption in your application, although it is possible for the exploit to happen before the heap manager notices corruption. In older versions of the heap, the default behavior when the application's heap

became corrupt was to just to leak the corrupted memory and keep running, even in the face of poorly behaved code. For example, here's something guaranteed to cause problems:

```
char* pBuf = (char*)malloc(128);
char* pBuf2 = (char*)malloc(128);
char* pBuf3 = (char*)malloc(128);

memset(pBuf, 'A', 128*3);
printf("Freeing pBuf3\n");
free(pBuf3);
printf("Freeing pBuf2\n");
free(pBuf2);
printf("Freeing pBuf\n");
free(pBuf);
```

On Windows Vista, even without the heap set to *terminate on corruption,* this code won't get any further than the first call to free before it causes the application to abort. On earlier versions of the operating system, including Windows Server 2003 and Windows XP, it executes all the printf statements and exits normally. Note that this has to be tested with release builds, because the debug heap does extra checking.

It's always better to crash than to run someone else's shell code, but your customers won't appreciate crashing either. Enabling *terminate on corruption* for your process's heap should be done early in your development cycle to give you time to shake out any previously benign bugs. Additionally, if your application can host third-party code in some form, such as plug-ins, you may want to think about getting the third-party code out of your process. A surprisingly large number of the crashes in Microsoft Office and Internet Explorer are due to code that isn't shipped by Microsoft. To enable the heap to terminate the application on corruption, simply add this code snippet to your application's main or WinMain function:

```
bool EnableTerminateOnCorrupt()
{
      if( HeapSetInformation( GetProcessHeap(),
                              HeapEnableTerminationOnCorruption,
                              NULL,
                              0 ) )
     {
          printf( "Terminate on corruption enabled\n" );
          return true;
     }
     printf( "Terminate on corruption not enabled - err = %d\n",
               GetLastError() );
     return false;
}
```

Obviously, you wouldn't leave diagnostic printf statements in your shipping code, so handle errors however you like. We'd suggest making an unusual failure such as this an exception–or this could be a protection that is only enabled on Vista and just ignore errors when running an earlier version of Windows.

An additional countermeasure is that the low fragmentation heap (LFH) is historically more resistant to attack than the standard Windows heap. Why not use the LFH all of the time? The answer is that heap performance is very dependent on how an application uses the heap, and in fact, Vista may decide to use the LFH at run time if usage patterns make it beneficial. Before shipping code with the LFH, use

performance benchmarking to see if there's an improvement or a decrease in performance. If the LFH works well with the application, here's how to enable it:

```
bool EnableLowFragHeap()
{
     ULONG ulHeapInfo = 2;
     if( HeapSetInformation( GetProcessHeap(),
                             HeapCompatibilityInformation,
                             &ulHeapInfo,
                             sizeof( ULONG ) ) )
     {
       printf( "Low fragmentation heap enabled\n" );
       return true;
     }
     printf( "Low fragmentation heap not enabled - err = %d\n",
                    GetLastError() );
     return false;
}
```

PREV PREV NEXT NEXT

# NX

NX, short for "No eXecute" is known by several names. Intel calls it the "Execute Disable" or XD-bit, AMD calls it "Enhanced Virus Protection," and it's also been written as W^X, which translates to write or execute, but not both. Windows refers to NX as "Data Execution Prevention," and the settings can be found using the System Control Panel applet, under Advanced, Performance Options. Non-executable stacks and heaps aren't especially new–Sun's Solaris operating system had a setting to enable a non-executable stack several years ago, and Open-BSD moved to NX in OpenBSD v3.3 and later. In the Windows line of operating systems, NX has been available since Windows Server 2003 shipped and then was back ported to Windows XP in Service Pack 2. An indication of just how far back NX was anticipated is that one of the flags to the *VirtualProtect* function, as it originally was documented back in 1993 when Windows NT 3.1 shipped, was a flag to set whether a page was executable. Since the x86 family of processors didn't support NX on a per-page basis until recently, the flag had no effect on x86 processors until some additional work was done to detect NX support at the processor level (David Cutler, personal communication). It didn't have an effect on some of the other processors originally supported by Windows NT, but it is interesting to note that the architects of the operating system anticipated this need.

By default the majority of core operating system components support NX. You can see if a component is protected by NX by looking at the process under Vista's Task Manager; it's an optional column named Data Execution Prevention. Although the dialog box that allows you to configure NX refers to software NX as a fallback, "software NX" doesn't protect much more than exception handlers and only handles a small subset of the attacks that real hardware NX will stop.

The concept here is simple: If a page of memory, whether it is on the stack or the heap, is writable, we ought not be executing code from that page. The vast majority of the exploits seen today will execute code from either the heap or the stack. You might wonder why NX isn't a sufficient protection against malware–take a look at "Bypassing Windows Hardware-enforced Data Execution Prevention" (Skape and Skywing 2005).

## Bypassing NX

Basically, there have to be ways to enable executable memory. One example would be when we load a DLL after process initialization. The operating system has to allocate pages and write the instructions into process memory, and the system expects to be able to execute these instructions once done. If a piece of shell code could first cause *VirtualProtect* to be called with the correct parameters, NX is then defeated. As it turns out, calling *VirtualProtect*, with all the correct parameters in place, is a little difficult due to needing to write very arbitrary values–even on systems without ASLR protection. Matt Miller (aka Skape) of the Metasploit project, and Ken Johnson (aka Skywing) point out that it is also possible to call NtSetInformationProcess to disable NX for an entire process, unless the application has been compiled with */NXCOMPAT*. This functionality allows for backward compatibility and allows an application to continue to work if it happens to load a DLL that isn't compatible with NX protection.

As of this writing, the combination of NX and ASLR appears to stop all but the most contrived attacks, but we're not foolish enough to claim that this set of protections is invincible (and most certainly not unbreakable). Let's take a look at how this feature works in a small snippet of sample code:

```
#include <stdio.h>
#include <string.h>

// win32_exec - EXITFUNC=seh CMD=calc.exe Size=164
// Encoder=PexFnstenvSub http://metasploit.com
const unsigned char scode[] =
"\x33\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xc9"
"\xfa\x9b\x8c\x83\xeb\xfc\xe2\xf4\x35\x12\xdf\x8c\xc9\xfa\x10\xc9"
"\xf5\x71\xe7\x89\xb1\xfb\x74\x07\x86\xe2\x10\xd3\xe9\xfb\x70\xc5"
"\x42\xce\x10\x8d\x27\xcb\x5b\x15\x65\x7e\x5b\xf8\xce\x3b\x51\x81"
"\xc8\x38\x70\x78\xf2\xae\xbf\x88\xbc\x1f\x10\xd3\xed\xfb\x70\xea"
"\x42\xf6\xd0\x07\x96\xe6\x9a\x67\x42\xe6\x10\x8d\x22\x73\xc7\xa8"
"\xcd\x39\xaa\x4c\xad\x71\xdb\xbc\x4c\x3a\xe3\x80\x42\xba\x97\x07"
"\xb9\xe6\x36\x07\xa1\xf2\x70\x85\x42\x7a\x2b\x8c\xc9\xfa\x10\xe4"
"\xf5\xa5\xaa\x7a\xa9\xac\x12\x74\x4a\x3a\xe0\xdc\xa1\x0a\x11\x88"
"\x96\x92\x03\x72\x43\xf4\xcc\x73\x2e\x99\xfa\xe0\xaa\xd4\xfe\xf4"
"\xac\xfa\x9b\x8c";

typedef void (*RunShell)( void );
int main( int argc, char* argv[] )
{
   char StackBuf[256];
   RunShell shell = (RunShell)(void*)StackBuf;
   strcpy_s( StackBuf, sizeof( StackBuf ), (const char*)scode );
   (*shell)();
   return 0;
}
```

This code sample is an example of just how ridiculously easy it has become for anyone to obtain completely arbitrary shell code to do whatever they like. All it took to get this string of shell code was to go to *www.metasploit.com*, download and install their framework, go to their Web interface, and then choose from the menu items. Once compiled and run, a running instance of the Windows calculator appears, and the application will then crash because the shell code is written to cause an exception once the command line is executed. Another option is to get the shell code to call *ExitProcess*, which will cause the application to exit cleanly.

If */NXCOMPAT* is added to the linker options, and if the system supports NX (that is, DEP or XD), and if the demo code is run again from the development environment, a message is generated that looks like this:

*Unhandled exception at 0x0012fe40 in NxTest.exe: 0xC0000005: Access violation.*

A quick look at the exception address shows that 0x0012fe40 is the address of *StackBuf*; note that there are no extra instances of calc.exe on the desktop. The system may indicate that the processor doesn't support NX when there is actually a new processor that should support NX. If that's the case, check the BIOS options. It seems that many BIOS manufacturers are defaulting NX to off. If the computer does have NX disabled in the BIOS, talk to the hardware vendor and ask that it be enabled by default for all their computers. Even with hardware NX disabled there may still be the weaker software-based protection, which ironically enough, won't stop the preceding example from executing because there is no buffer overrun corrupting an exception handler.

It's also interesting to see what Windows Vista does with this application when it is simply invoked from the command line. It first tells us that NxTest.exe has stopped working. If we then tell it to close, a help balloon pops up telling us that Data Execution Prevention has stopped the program, which then explains that we've just stopped a security-related flaw.

You must test your application with NX enabled because your customers are going to do it for you, sometimes without knowing what the problem could be, which is that NX is enabled differently on the server line of operating systems and the desktop versions. On the client versions (Windows XP and Vista), the default is that only operating system components are opted in; no other applications have NX enabled. On the server, all applications, except those in the opt-out list, have NX enabled. While that's the default, nervous security administrators could change the default for clients to the setting for the servers. If this isn't tested, there could be seemingly random failures as people upgrade to hardware that supports NX and has it enabled in the BIOS.

Currently, the NXCOMPAT flag in the header is ignored by any Windows operating system prior to Windows Vista. On Windows Vista, if NXCOMPAT is set in the linker options, then the application will be running with NX regardless of the settings at OS level (once hardware NX is available). If NXCOMPAT:NO is set, then NX will not apply to the application. Just like ASLR and the heap settings, NXCOMPAT is set processwide; if you're hosting third-party code, this may cause some problems and is another reason to find a way to get third-party code into its own process.

## Performance and Compatibility Implications

There is no performance impact of using NX except when an exception is raised. There could, however, be compatibility problems, depending on how complex an application is. Note that applications using older versions of the ATL library may have problems with NX. It is quite typical for some applications that perform high-performance image manipulation and rendering to create assembler for the pixel pipeline on the fly. Some interpreted languages will also compile a script into assembler and execute it. If the application allows third-party plug-ins, but you can't be sure that all of the plug-ins are compiled with NX (or are compatible), then you may not be able to immediately use NX in your application. If it is the application that is creating the assembler, the work-around is to obtain a memory page for this use and call *VirtualProtect* to enable execution on that page. If you can predict what assembler is needed, then it is best to write it into memory and disable *write* at the same time you enable *execute* on that page. If the application allows plug-ins, then the best approach is to get the plug-ins out of the process. Getting other people's code out of your process may give you some stability benefits, assuming you write better

code than they do!

Here's a handy class that you can use if you do need to execute code on the fly and would like to do it safely:

```
class WriteThenExecute
{
public:
   WriteThenExecute() : pMemory( NULL ),
                        fIsExecutable( false ),
                        cbMemory( 0 )
   {}

   ~WriteThenExecute()
   {
         if( pMemory != NULL )
            VirtualFree( pMemory, 0, MEM_RELEASE );
   }

   void Allocate(SIZE_T dwSize = 4096 )
   {
      pMemory = VirtualAlloc( NULL, dwSize,
                              MEM_COMMIT, PAGE_READWRITE );
      if( pMemory == NULL )
          throw GetLastError();
      cbMemory = dwSize;
   }

   void CopyBytes( BYTE* pBytes, SIZE_T cbBytes )
   {
      if( cbBytes <= cbMemory )
      {
           DWORD dwPrevProtect;
           CopyMemory( pMemory, pBytes, cbBytes );
           if( VirtualProtect( pMemory, cbMemory,
                               PAGE_EXECUTE_READ, &dwPrevProtect ) )
           {
              fIsExecutable = true;
              return;
           }
           throw GetLastError();
       }
       throw ERROR_INSUFFICIENT_BUFFER;
   }
   template< typename T >
   T GetFunctionPtr()
   {
      if( fIsExecutable )
           return reinterpret_cast< T >( pMemory );
      return reinterpret_cast< T >( NULL );
     }
     private:
     void* pMemory;
     SIZE_T cbMemory;
     bool fIsExecutable;
  };

int main(void)
{
```

```
     WriteThenExecute foo;
     BYTE DemoBuffer[64];
     foo.Allocate();
     foo.CopyBytes( DemoBuffer, sizeof( DemoBuffer ) );
     BYTE* pDemo = foo.GetFunctionPtr< BYTE* >();
}
```

The small main function demonstrates how you might use this class. It's fairly simple–just allocate as much memory as you'll need, copy in the assembly you'd like to run, and then get a function pointer. There are a couple of issues we'd like to bring up before you put this code into production. First, some asserts ought to be sprinkled liberally through the code. Calling *Allocate* more than once will leak memory and corrupt internal state; calling *CopyBytes* more than once will cause you to throw an exception; and the default copy constructor and assignment operator ought to be declared private with no implementation, because copying this class to another instance would cause memory leaks or double-free conditions. Second, it's also a good idea to make a *Release*, and possibly a *MakeWritable* method that would reset the class to its original state. Although we throw DWORDs as exceptions in many places in the examples, you shouldn't do this in production code–use std::exception, or another dedicated exception class instead. Finally, the class isn't thread safe.

PREV PREV NEXT NEXT

# /GS

The */GS* flag has been around since the release of Visual Studio 7.0. In the original approach, it was simply a randomly generated cookie placed between the return address and the local variables on the stack. If the EBP register was pushed onto the stack, the cookie would guard this as well. It would stop some of the simplest attacks against simple functions, and, although it was better than nothing, no developer should ever believe that they're protected against attacks merely because of */GS*. Let's take a look at a sample application:

```
#define _CRT_SECURE_NO_DEPRECATE
#include <stdio.h>
#include <string.h>

void VulnerableFunc( const char* input, char* out )
{
   char* pTmp;
   char buf[256];
   strcpy( buf, "Prefix:" );
   strcat( buf, input );
   // Transform the input, and write it to the output buffer
   for( pTmp = buf; *pTmp != '\0'; pTmp++ )
   {
        *out++ = *pTmp + 0x20;
        }
   }
    int main( int argc, char* argv[] )
   {
       char buf2[256];
       VulnerableFunc( argv[1], buf2 );
       printf( "%s\n", buf2 );
       return 0;
```

```
   }
```

Starting just after we enter *main*, here's the commented disassembly (my comments above the instructions):

```
int main( int argc, char* argv[] )
{
// Save the previous value of the frame pointer on the stack
00411300 push        ebp
// Put the current stack pointer into the ebp register
00411301 mov         ebp,esp
// Create room for buf2
00411303 sub         esp,140h
// Save three more registers on the stack
00411309 push        ebx
0041130A push        esi
0041130B push        edi
   char buf2[256];

   VulnerableFunc( argv[1], buf2 );
// Get the address of buf2, and put it on the stack
0041130C lea         eax,[buf2]
00411312 push        eax
// Find the value of argv[1], also place it on the stack
00411313 mov         ecx,dword ptr [argv]
00411316 mov         edx,dword ptr [ecx+4]
00411319 push        edx
// Call VulnerableFunc, and on return, adjust the stack pointer
0041131A call        VulnerableFunc (41102Dh)
0041131F add          esp,8
```

Once inside *VulnerableFunc*, we have this:

```
void VulnerableFunc( const char* input, char* out )
{
// Save the previous   value of the frame pointer on the stack
00411260 push          ebp
// Put the current stack pointer into the ebp register
00411261 mov           ebp,esp
// Create room for buf and pTmp
00411263 sub           esp,144h
00411269 push          ebx
0041126A push          esi
0041126B push          edi
    char* pTmp;
    char buf[256];
```

In this example, all optimizations and stack checking are disabled so that you can see what typical applications used to do. In this application, the stack is shown by the diagram in Figure 3-2.

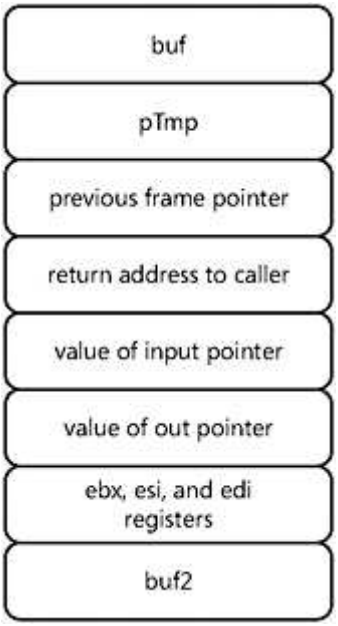

Figure 3-2: Stack diagram for a typical application.

Now let's look at what happens, step by step, as *buf* overflows. The first value to get mangled is *pTmp*, but that won't get us anywhere–it will just be initialized to zero and incremented as we process the *buf* character array. The next value to get overwritten is the previous frame pointer. This is interesting because on return from the second function (in this case, *main*), code execution will jump into the address located here. If we had the classic off-by-one attack, this is where the fun starts.

Next, things start to get really fun: we can now overwrite the return address and set the *input* and *out* values to anything we like. Setting *input* isn't especially interesting–by the time the overrun has happened, we've already used that variable and won't write it again. The *out* parameter is a real problem, because once that is overwritten the attacker can write nearly anywhere in application memory that they'd like once we fall into the *for* loop toward the end of *VulnerableFunc.*

In the original */GS* implementation, a cookie was just placed between the local variables and the stored registers (if any) and the return address. This would leave *pTmp* open to attack, and although in this case, it is uninteresting, that may not always be true, especially if this were a function pointer. In the case of a limited overwrite, it isn't possible to tamper with the previous frame pointer without detection, so */GS* in its simplest incarnation would stop a simple off-by-one (or even off-by-a-few) overflow from being exploitable. In the case of an unchecked overrun, the attacker has the problem of having to fix up the cookie, but recall that overwriting the *out* pointer allows the attacker to put nearly any value he or she would like into any memory location (although in this example, a location with any 0 bytes would be a problem). Let's take a look at what the most recent version of */GS* will do with the same function:

```
void VulnerableFunc( const char* input, char* out )
{
// Make room for the local variables
00401000 sub        esp,104h
// Copy the security cookie into eax
00401006 mov        eax,dword ptr [___security_cookie (403000h)]
// XOR the cookie with the stack pointer
0040100B xor        eax,esp
// Put the resulting cookie at the end of the buffer
0040100D mov        dword ptr [esp+100h],eax
     char* pTmp;
```

```
     char buf[256];

     strcpy( buf, "Prefix:" );
00401014 mov        ecx,dword ptr [string "Prefix:" (4020DCh)]
// Now copy the function arguments into registers before anything can
// tamper with them.
0040101A mov        eax,dword ptr [esp+108h]
00401021 mov        edx,dword ptr [esp+10Ch]
```

As you can see, this implementation is much safer. The arguments to the function are either copied into registers or are put on the stack above the buffer. Additionally, as we discuss in the next section, if an exception were thrown inside this function, the cookie would be checked prior to executing the exception handler.

Note While you should always use */GS*, you still need to write solid code!

PREV PREV NEXT NEXT

# SafeSEH

One of the most significant flaws in the original implementation of the */GS* flag was found by David Litchfield. In any Windows application, there will be at least one, and possibly several, structured exception handlers (SEH). While we don't have room for a thorough review of how exception handlers work and when you should be using them, SEH understand three keywords: *__try*, *__except*, and *__finally*. If you'd like to learn more about exception handlers, look up the "Structured Exception Handling" topic in MSDN, and one of the best explanations is in Jeffrey Richter's *Advanced Windows* books. Any edition will do, although unfortunately, the books are currently out of print.

The *__try* keyword is almost exactly analogous to the C++ *try* keyword and declares that a block of code has an exception handler. The *__except* keyword declares a block that behaves similarly to a block declared with *catch* in C++. Unlike C++, you don't catch thrown exceptions by type, but you can analyze the exception record to determine whether you'd like to handle the exception. You can choose to continue execution after the handler, do something to fix the problem and resume execution, or tell it to continue to search for an applicable exception handler. A *__finally* block is a way for a C program to behave very similarly to how a C++ application would use destructors. No matter how you exit the *__try* block, the *__finally* block is guaranteed to be executed.

We can't say strongly enough that you need to be exceptionally careful when using SEH or C++ exception handling. We've both seen bad code along these lines:

```
__try
{
    // We're not really sure if this will work
    memcpy( dst, src, size );
}
__except( EXCEPTION_EXECUTE_HANDLER )
{
  // I guess something went wrong. Oh well, exceptions are kewl
}
```

Do NOT do this in your code! This sort of code is a great way to get hacked, even if you are using

the */SafeSEH* flag. So let's take a look at what can go wrong with SEH, and why it became a problem for the */GS* flag. When an exception handler is registered for a block of code, an *EXCEPTION_REGISTRATION* structure is pushed onto the stack. The *EXCEPTION_ REGISTRATION* structure contains a pointer to the next *EXCEPTION_REGISTRATION* structure, and the address of the current exception handler. Although the amazingly clever Mr. Litchfield has written a 16-page document detailing it (Litchfield 2003), the problem is essentially that you have a function pointer sitting on the stack just waiting to get overwritten. If the attacker can overwrite the buffer far enough, the function pointer to the current exception handler gets overwritten with the address of the attacker's choice. The only thing remaining is to cause an exception before the function exits normally. This problem is why attacks that write a very large amount of information outside of the buffer are frequently exploitable. Once the attacker provokes an exception, the exception handler is called, and because the function hasn't returned, the security cookie isn't checked, and we're now running arbitrary code. Where some of the previously reported flaws in */GS* depended on contrived code with multiple problems to bypass stack protection, this attack works regardless of the code internal to the function, as long as the overwrite extends far enough to hit the exception handler, or is an arbitrary DWORD overwrite, and if an exception can be caused prior to the function exiting normally. As mentioned in the previous section, the Visual Studio 2005 compiler treats calling an exception handler as if the function had exited (which is often the result of calling an exception handler) and checks the security cookie prior to executing the handler. While Litchfield's attack is thwarted, an arbitrary DWORD written on top of an exception handler will still cause problems unless we use SafeSEH.

Here's an application we can use to demonstrate both the problem and the various solutions:

```
#define _CRT_SECURE_NO_DEPRECATE
#include <windows.h>
#include <stdio.h>
win32_exec - EXITFUNC=process CMD=calc.exe Size=164
Encoder=PexFnstenvSub http://metasploit.com
unsigned char scode[] =
"\x33\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x46"
"\x12\x29\x79\x83\xeb\xfc\xe2\xf4\xba\xfa\x6d\x79\x46\x12\xa2\x3c"
"\x7a\x99\x55\x7c\x3e\x13\xc6\xf2\x09\x0a\xa2\x26\x66\x13\xc2\x30"
"\xcd\x26\xa2\x78\xa8\x23\xe9\xe0\xea\x96\xe9\x0d\x41\xd3\xe3\x74"
"\x47\xd0\xc2\x8d\x7d\x46\x0d\x7d\x33\xf7\xa2\x26\x62\x13\xc2\x1f"
"\xcd\x1e\x62\xf2\x19\x0e\x28\x92\xcd\x0e\xa2\x78\xad\x9b\x75\x5d"
"\x42\xd1\x18\xb9\x22\x99\x69\x49\xc3\xd2\x51\x75\xcd\x52\x25\xf2"
"\x36\x0e\x84\xf2\x2e\x1a\xc2\x70\xcd\x92\x99\x79\x46\x12\xa2\x11"
"\x7a\x4d\x18\x8f\x26\x44\xa0\x81\xc5\xd2\x52\x29\x2e\x6c\xf1\x9b"
"\x35\x7a\xb1\x87\xcc\x1c\x7e\x86\xa1\x71\x48\x15\x25\x3c\x4c\x01"
"\x23\x12\x29\x79";
char* pHeapShell;
// Function to demonstrate overwriting a structured exception
// handler.
void foo( int elements )
{
   char* ptr = NULL;
   char buf[ 32 ];
   int i;

   // For loop to cause the overwrite.
   // The value written into each byte is
   // incremented to make it easier to determine where
   // the exception handler address is located
   for( i = 0; i < elements; i++ )
   {
      buf[i] = 0x20 + i;
```

```
      // In this particular piece of code, the
      // exception pointer starts at an offset of 0x30
      // bytes from the start of the buffer
      // Note – if this is compiled with a newer
      // compiler, the offset might be different
      // on my pre-release version, the test should be
      // for buf[i] == 0x58
      if( buf[i] == 0x50 )
      {
         DWORD* pdw = (DWORD*)( buf + i );
         // To try and execute an exception handler
         // that isn't on the heap, change this to:
         // *pdw = (DWORD)scode;
         *pdw = (DWORD)pHeapShell;
         i += 3;
      }
   }
       buf[elements] = 0;

       printf( "%s\n", buf );

       // The try-except block that we need to have an
       // exception handler nearby. Note that if we didn't have
       // a try-except or try-catch block of our own, there would
       // an exception handler for the application
       __try
       {
          // All of this is fairly silly code that prevents
          // the compiler from optimizing the whole block away
          if( buf[31] == '\0' )
       {
                ptr = buf;
       }
       *ptr = 'A';
    }
          // This __except statement swallows all exceptions
          // Don't do this in your code
          __except( EXCEPTION_EXECUTE_HANDLER )
    {
             printf( "Caught an exception!\n" );
             return;
    }
 // This statement is needed for the ptr variable
 // not to get optimized out
    printf( "%s\n", ptr );
}
int main(int argc, char* argv[])
{
    // See just how far the user would like to
    // overwrite the buffer...
    int elements = atoi( argv[1] );
    // Create a heap buffer to put the shell code into
    pHeapShell = (char*)malloc( sizeof( scode ) );
        if( pHeapShell != NULL )
     {
         // Copy the shell code into our buffer
         memcpy( pHeapShell, scode, sizeof( scode ) );
     }
foo( elements );
```

```
printf( "Address of shell code is %p\n", scode );
if( pHeapShell != NULL )
   free( pHeapShell );
return 0;
}
```

This program makes no attempt to pretend to be an actual application someone might write intentionally, but which happens to make a few mistakes. Some of the errors shown here are unfortunately all too common–the most egregious example is that we're taking user-supplied data and using it as a count. If we compile this application using Visual Studio .NET, using default settings and */GS* enabled, we get several different behaviors. With a small count, the output will look like this:

```
c:\ projects> SafeSehTest.exe 1
A
Address of shell code is 00408040
```

Now let's say we raise the count to be large enough to trip the exception handler within the *try-except* block:

```
c:\ projects>SafeSehTest.exe 32
!"#$%&'()*+,-./0123456789:;<=>?
Caught an exception!
```

This will be followed shortly by Windows Vista telling us that the application has a buffer overrun and cannot safely continue. So far, we haven't given the application enough elements to overwrite the exception handler; let's try a larger number:

```
c:\ projects>SafeSehTest.exe 92
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN¤î
```

We're now looking at an extra instance of calc.exe running on the desktop. The shell code nicely called *exit* on the process once it ran. Note that we used EXITFUNC = process, not EXITFUNC = seh as in the previous example because if we continue to throw exceptions into a process with a corrupted exception handler, we'll just keep recursively calling the shell code until the process overflows the stack–it runs out of room for the stack, not an overwrite condition. This will result in about 100 instances of calc.exe running–or attempting to run and failing–on the desktop. If we keep increasing the size of the overwrite, the application will cause an exception before it hits our contrived exception; it doesn't print anything, but we do have calc.exe running.

To test how the changes in Visual Studio 2005 help protect things in this case, let's modify the code a bit. First, get rid of the straightforward stack overwrite by commenting out the first line inside the *for* loop, and change the problem to a far more serious case where an arbitrary DWORD is written to the address of our choice. With this sort of attack, we'll find that */GS* won't help us because the cookie isn't tampered with. With the compiler used for this example, we find that the address of the exception handler pointer is located at *buf* + 0x38, not *buf* + 0x30, like it was in the version compiled with Visual Studio .NET. Note that we have two mitigations on by default: */GS* and SafeSEH. We can also enable NX. If we enable only SafeSEH, we're told that SafeSehTest.exe has stopped working. This is to be expected because the shell code isn't a registered exception handler.

Now, restore the overwrite by enabling the first statement in the *for* loop, re-enable */GS*, and disable SafeSEH. Now the app crashes, just as it did with only SafeSEH enabled. This is interesting, because */GS* on the older compiler was unable to save us from this attack. The change we're seeing is

that if */GS* is enabled, and we try to jump to an exception handler, the stack cookie is checked, and the clever attack noted by Mr. Litchfield is foiled.

For a third approach, disable SafeSEH and */GS*, then enable NX. If we try the code in this case, we find that the attack is foiled again. It's interesting how three different sets of protections will each individually stop the original attack. Even with the far more dangerous arbitrary 4-byte overwrite, two of the three protections will take effect to prevent an attack against the exception handler. Please note that we're not claiming arbitrary 4-byte overwrites are no longer dangerous–just those that are only able to overwrite one exception handler pointer. Even with all these countermeasures in place, an arbitrary 4-byte overrun is considered very dangerous and presumed exploitable.

If compiling 64-bit code, the exception records are compiled into the binary and aren't kept on the stack, which makes 64-bit executables much safer–at least from SEH attacks.

PREV PREV NEXT NEXT

# Summary

We've covered ASLR, stack and heap randomization, heap defenses, NX, */GS*, and SafeSEH. Unless there are compelling reasons, you need to be using all of these in your application. Here's what you should be doing:

- **ASLR** Link with the */dynamicbase* option
- **Heap defenses** Enable *HeapTerminateOnCorrupt*
- **NX** Link with */NXCOMPAT*
- ***/GS*** Don't do anything! Just take the defaults!
- **SafeSEH** Link with */SAFESEH*
- **Use the latest compiler** At the very least, Visual Studio 2005 with Service Pack 1–and ensure all of the DLLs the process links with are also compiled with the same compiler and C runtime library.

So, what's still exploitable with all these overlapping countermeasures in place? Note that there are ways to overcome every one of these mitigations. Overcoming them all at once is a lot harder, but it is still possible. Data-driven attacks should also be considered. As a quick example, say an application would call *LoadLibrary* on a DLL, and an attacker was able to overwrite the DLL path or name–you'd load the wrong DLL, and call the attacker's *DllMain* function. Some of the most prevalent attacks against Web servers don't involve arbitrary code but attack the logic of the underlying application, and these attacks won't be thwarted by compiler or linker options. At the end of the day, the only way to get secure code is to write solid code.

PREV PREV NEXT NEXT

## Call to Action

- You should link your code with */dynamicbase*, run a full test run, and ship with this option enabled
- You should link your applications with the */NXCOMPAT* switch

PREV PREV NEXT NEXT

## References

**(Howard and LeBlanc 2003)** Howard, Michael, and David LeBlanc. *Writing Secure Code*, 2nd Edition. Redmond, WA: Microsoft Press, 2003.

**(Humphrey 1999)** "Bugs or Defects?" *http://www.sei.cmu.edu/news-at-sei/columns/watts_new/1999/March/watts-mar99.htm*.

**(LeBlanc 2004)** LeBlanc, David. "Integer Handling with the C++ SafeInt Class," *http://msdn.microsoft.com/library/en-us/dncode/html/secure01142004.asp*. January 2004.

**(LeBlanc 2005)** LeBlanc, David. "Another Look at the SafeInt Class," *http://msdn2.microsoft.com/en-us/library/ms972819.aspx.* May 2005.

**(eEye 2001)** eEye Digital Security. "Microsoft Internet Information Services Remote Buffer Overflow," *http://research.eeye.com/html/advisories/published/AD20010618.html*. June 2001.

**(Skape and Skywing 2005)** "Bypassing Windows Hardware-enforced Data Execution Pre-vention," *http://www.uninformed.org/?v=2&a=4&t=txt.* October 2005.

**(Litchfield 2003)** Litchfield, David. "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," *http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf*. September 2003.

PREV NEXT