Michael Howard

Contents



A lot of code is written in C and C++ and unfortunately a lot of this code has security vulnerabilities that many developers do not know about. Programs written in any language can have vulnerabilities that leave their users open to attack, but it's the C and C++ languages that have a special place in Internet history because so many security vulnerabilities are due to the very thing that makes these two programming languages so popular: the unbridled access to computer hardware and the performance that comes with it. When you read about security and C or C++ crops up, the words "buffer" and "overrun" are usually pretty close by because buffers are typically an example of direct access to memory. This kind of direct access is very powerful--and very, very dangerous.

There are a number of reasons for the many buffer overruns in production C and C++ code. The first I have already mentioned: the languages provide direct

access to vulnerable memory. Second, developers make mistakes. And third, there are normally no defenses offered by compilers. It is feasible to provide remedies for the first issue, but then C and C++ start to become different languages.

The reason for developers making mistakes could be partially addressed through education, but I really don't see the educational institutions stepping up. Sure there is a place in industry for security education, too, but we are all part of the solution or part of the problem, and I would love to see colleges doing more to educate students about software security. You're probably asking, "Why are educational institutions not attempting to teach this critically important subject?" To be honest, I have absolutely no idea. It's kind of depressing, actually.

Finally, even with an excellent education, some security issues are complex enough that even well-educated engineers won't catch everything. We humans are not perfect.

The issue about needing to build more defenses into compilers is something the Microsoft Visual C++ team has been working on and slowly improving over the years, with help from our security team. This column outlines some of the buffer overrun defenses available in Visual C++® 2005 and beyond. Note that some other compilers do offer defenses, but Visual C++ has two major advantages over the likes of gcc. First, all these defenses are in the toolset by default; there is no need to download some funky add-in. Second, the options are easy to use.

In no particular order, the defenses offered by the Visual C++ toolset are:

- Stack-based Buffer Overrun Detection (/GS)
- Safe Exception Handling (/SafeSEH)
- Data Execution Prevention (DEP) Compatibility (/NXCompat)
- Image Randomization (/DynamicBase)
- Automatic use of safer function calls
- C++ operator::new

Before I discuss each of these in detail I want to point out that these defenses do not compensate for insecure code. You should always strive to create the most secure code possible, and if you don't know how to do that, then run right out and read some of the very good books available on the subject.

Typical Stack Compared to One Compiled with /GS (Click the image for a larger view)

I also want to point out that these are all Security Development Lifecycle (SDL) requirements at Microsoft, which means that C and C++ code must use these options in order to ship. There are occasional exceptions, but they are relatively rare, so I won't be going into detail about them here.

Finally, you must keep in mind this important point: it is possible, depending on the code in question, to circumvent these elaborate defenses. The more defenses used by the code, the harder they are to work around, but no defense is perfect. They are all speed bumps to reduce the chance an exploit will succeed. You have been warned! The only variant is the use of safer function calls, which are real defenses and can remove vulnerabilities. Let's look at each defense in detail.

Stack-Based Buffer Overrun Detection (/GS)

Stack-based buffer overrun detection is the oldest and most well-known defense available in Visual C++. The goal of the /GS compiler flag is simple: reduce the chance that malicious code will execute correctly. The /GS option is on by default in Visual C++ 2003 and later, and it detects certain kinds of stack smash at run time. It goes about doing this by including a random number in a function's stack just before the return address on the stack, and when the function returns, function epilogue code checks this value to make sure it has not changed. If the cookie, as it's called, has changed, execution is halted.

The function prologue code that sets the cookie looks like this:

Copy Code

```
sub    esp, 8

mov    eax, DWORD PTR ___security_cookie

xor    eax, esp
```

```
mov    DWORD PTR __$ArrayPad$[esp+8], eax

mov    eax, DWORD PTR _input$[esp+4]
```

And shown here is the function epilogue code that checks the cookie:

Copy Code

```
mov    ecx, DWORD PTR __$ArrayPad$[esp+12]

add    esp, 4

xor    ecx, esp

call    @__security_check_cookie@4

add    esp, 8
```

Visual C++ 2005 also moves data around on the stack to make it harder to corrupt that data. Examples include moving buffers to higher memory than non-buffers. This step can help protect function pointers that reside on the stack, for example. Alternately, moving pointer and buffer arguments to lower memory at run time mitigates various buffer overrun attacks. See the diagram to compare a typical stack with a /GS stack.

The /GS compiler option is not applied in any of the following situations:

- Functions do not contain a buffer.
- Optimizations are not enabled.

- Functions are defined to have a variable argument list.
- Functions are marked with the naked keyword (C++).
- Functions contain inline assembly code in the first statement.
- The buffer isn't an 8-byte type and is less than 4 bytes in size.

An option was added to Visual C++ 2005 SP1 to make the /GS heuristics much more aggressive so that more functions would be protected. Microsoft added this because a small number of security bulletins were issued in code that had stack-based buffer overruns and the code, even though it was compiled with /GS, was not protected by a cookie. This new option increases the number of protected by functions substantially.

Use this option by placing the following line in modules where you want added protection, such as code that handles data from the Internet:

Copy Code

```
#pragma strict_gs_check(on)
```

This pragma is just one example of how Microsoft is constantly evolving the /GS capability. The original version in Visual C++ 2003 was pretty simple, and then it was updated in Visual C++ 2005 SP1 and again in Visual C++ 2008 as we learned about new attacks as well as new ways to circumvent existing attacks. In our analysis, we have found that /GS causes no real compatibility or performance problems.

### Safe Exception Handling (/SafeSEH)

The CodeRed worm that affected Internet Information Server (IIS) 4.0 was caused by a stack-based buffer overrun. Interestingly, /GS would not have caught the problem exploited by the worm because the code did not overrun the affected function's return address; rather, it corrupted an exception handler on the stack. This is a great example of why you must constantly focus on writing secure code and not rely solely on these types of compiler-based defenses.

An exception handler is code executed when an exceptional condition occurs, such as division by zero. The address of the handler is held on the stack frame of the function and is therefore subject to corruption. The linker included with Visual Studio® 2003 and later includes an option to store the list of valid exception handlers in the image's PE header at compile time. When an exception is raised at run time, the operating system checks the image header to determine whether the exception handler address is correct; if it is not, the application is terminated. This would have prevented CodeRed if the technology existed at the time the code was linked. There is no performance hit whatsoever from the /SafeSEH option, other than when an exception is raised, so you should always link with this option.

### DEP Compatibility (/NXCompat)

Just about every CPU produced today supports a no execute (NX) capability, which means the CPU will not execute non-code pages. Think about the implications of this for a moment: just about every buffer overrun vulnerability is a

data bug; the attacker then injects data into the process by way of the buffer overrun and then continues execution within the malicious data buffer. Why on earth is the CPU running data?

Linking with the /NXCompat option means your executable will be protected by the CPU's no execute capability. In our experience, the security team at Microsoft has seen very few compatibility issues because of this option, and there is no performance degradation whatsoever.

Windows Vista® SP1 also adds a new API that will enabled DEP for your running process and once it is set, it cannot be unset:

Copy Code

```
SetProcessDEPPolicy(PROCESS_DEP_ENABLE);
```

Image Randomization (/DynamicBase)

Windows Vista and Windows Server® 2008 support image randomization, which means, when the system boots, it shuffles operating system images around in memory. The purpose of this feature is to simply remove some of the predictability from attackers. This is also known as Address Space Layout Randomization (ASLR). Note that for ASLR to be of any use whatsoever, you must also have DEP enabled.

By default, Windows® will only juggle system components around. If you want your image to be moved around by the operating system (highly recommended), then you should link with the /DynamicBase option. (This option is available in the Visual Studio 2005 SP1 and later toolset.) There is an interesting side effect of linking with /DynamicBase--the operating system will also randomize your stack, which helps reduce predictability and makes it much harder for attackers to successfully compromise a system. Note also that the heap is also randomized in Windows Vista and Windows Server 2008, but this is there by default, you do not need to compile or link with any special options.

Safer Function Calls

Take a look at the following lines of code:

Copy Code

```
void func(char *p) {
    char d[20];
    strcpy(d,p);
    // etc
}
```

Assuming *p contains untrusted data, this code represents a security vulnerability. The perverse thing about this code is that the compiler could

potentially have promoted the call to strcpy to a safer function call that bounded the copy operation to the size of the destination buffer. Why? Because the buffer size is static and known at compile time!

With Visual C++ you can add the following line to your stdafx.h header file:

Copy Code

```
#define _CRT_SECURE_COPP_OVERLOAD_STANDARD_NAMES 1
```

The compiler will then go ahead and emit the following code from the initial, unsafe function:

Copy Code

```
void func(char *p) {

    char d[20];

    strcpy_s(d,__countof(d), p);

    // etc

}
```

As you can see, the code is now secure, and yet the developer did nothing other than add a #define. This is one of my favorite additions to Visual C++ because about 50 percent of unsafe function calls can be promoted to safer calls automatically.

C++ Operator::new

Finally, Visual C++ 2005 and later adds a defense that detects integer overflow when calling operator::new. You can start with code like this:

Copy Code

```
CFoo *p = new CFoo[count];
```

Visual C++ compiles it to this:

Copy Code

```
00004    33 c9           xor    ecx, ecx

00006    8b c6           mov    eax, esi

00008    ba 04 00 00 00  mov    edx, 4

0000d    f7 e2           mul    edx

0000f    0f 90 c1       seto    cl

00012    f7 d9           neg    ecx

00014    0b c8            or    ecx, eax

00016    51             push    ecx

00017    e8 00 00 00 00 call    ??2@YAPAXI@Z    ; operator new
```

After the amount of memory to allocate is calculated (mul edx), the CL register is set or not set depending on the value of the overflow flag after the multiplication, so, ECX will be 0x00000000 or 0xFFFFFFFF. Because of the next operation (or ecx) the ECX register will either be 0xFFFFFFFF or the value held in EAX, which is result of the initial multiply. This is then passed to operator::new, which will fail in the face of the 2^N-1 allocation.

This defense is free. There is no compiler switch; this is simply what the compiler does.

What if It Fails?

Ah! This thorny question! If any of the defenses listed above are triggered, there is a pretty nasty result: the application quits. That's hardly optimal, but it is a heck of a lot better than running the attacker's malicious code.

The SDL mandates that new code use all these defensive options simply because there are so many attacks out there and you can never verify that your code is 100 percent free of vulnerabilities. One of the catch phrases of the SDL is "assume your code will fail; now what?" In the real world "now what" means put up a fight! Don't let the attacker's code have an easy time; make it hard for the attacker to get exploits to work. Don't give up!

So compile with the latest version of the C++ compiler to get a better /GS, and link with the latest linker to get the benefit of CPU and operating systems defenses.

Send your questions and comments to briefs@microsoft.com.

**Michael Howard** is a Principal Security Program Manager at Microsoft focusing on secure process improvement and best practices. He is the coauthor of five security books including *Writing Secure Code for Windows Vista*, *The Security Development Lifecycle*, *Writing Secure Code*, and *19 Deadly Sins of Software Security*. View his blog at blogs.msdn.com/michael_howard.