

# Web Application Security - Buffer Overflows: Are you really at risk?

Tom Olzak  
July 2006

OK, I know what you're thinking, "I can't read another paper on buffer overflows." But just stay with me for a moment. There's been a lot of information floating around the Internet about the dangers of buffer overflow vulnerabilities in web applications, but just how vulnerable are your web apps? In this paper I'll explore the answer to that question, and the answer may pleasantly surprise you.

## A Buffer Overflow Refresher

Buffer overflows occur when more data is written to memory than was allocated by the program. In other words, the programmer was responsible for properly managing memory and made a common error. It doesn't take much of an overflow to impact a system. As little as one byte of data can cause a security incident. The scary thing is, buffer overflow vulnerabilities are one of the most popular targets of criminal hackers.

I've listed three common overflow types. There are more, but the fundamental causes and effects are the same.

### Stack Overflow

The [stack](#) is an area of memory where your computer stores information that won't fit in its [registers](#). There are different types of stacks created depending on the applications and the operating system (OS) executing on your system. For our purposes, I'll demonstrate the use of a stack allocated to an application [thread](#).

Example 1 in Figure 1 depicts a stack with two entries. The entries in a stack are typically located by using an offset from the stack's starting address, and the stack returns data using the principle of First In Last out.

The first entry in our stack is located at offset 0000. The offset for each new entry is stored in the [stack pointer](#) register. In this case, the value 10 is the newest entry. Since there were already four bytes of data in stack, this data value was located at the fifth byte (counting from 0) in the stack space, and the stack pointer was set to 0004. When the application requests the last value added to the stack, the value located at 0004 will be

returned and the stack pointer set to 0000. If no other new entries are made before the next request for stack data, then the value located at 0000 will be returned.

The values in Example 1 represent the location in the program where execution will resume when the current function completes its tasks and a value passed to a called function. In other words, a function was called during program execution. In order for the program to know where to resume once the called function returns control back to the calling function, the address of the next line of code to be executed is stored in the stack. In this case, the value of that address is 0650. The value of 10 in offset 0004 represents a value passed to the called function. In this example, there's no problem. The programmer assumed that the value passed to the called function would not exceed 4 bytes. The called function executes, and control is returned to the appropriate line of code in the calling function.

Example 1	
Stack Address	Value
0000	0650
0004	"10"

  

Example 2	
Stack Address	Value
0000	0850
0004	"1000"

  

Example 3	
Stack Address	Value
0000	9999
0004	"9999"

Figure 1: Stack Examples

In Example 2, an attacker is taking advantage of a buffer overflow vulnerability in the application. The attacker found that the programmer didn't add code to verify the size or data type of the data passed to the called function. By entering the value 10000850 (which exceeds the expected 4-byte limit), the attacker has succeeded in overwriting the return address stored in offset 0000 with the address of a malicious executable. When the called function completes its tasks, control will be handed over to the malicious program at address 0850 instead of the next line in the calling function at address 0650.

Not all buffer overflow attacks are designed to cause the execution of malicious code. In Example 3, the attacker simply entered a series of 9's. In this case, the program will probably [crash](#) when it attempts to return control to the calling function.

The data provided to the called function might come from a variety of sources. The key point to take from this example is that the input was not properly validated.

(Note: For you purists out there, I know that certain values in the stack might not be stored most significant digit first. This is just easier for demonstration purposes.)

## Heap Overflow

When a program retrieves a large amount of data for processing, a portion of memory known as the [heap](#) is allocated to handle the loaded data. In low-level languages like C and C++, the programmer is responsible for ensuring the proper amount of memory is set aside. If the loaded data is larger than the allocated heap memory, the system could crash.

## Integer Overflow

When adding two integers, the result occasionally exceeds the memory allocated for the result. The following example is from the [OWASP Buffer Overflows](#) document.

When added together, the following two eight bit integers (10 + 5) fit nicely into an eight bit result space:

$$\begin{array}{r} 0000\ 1010\ (10) \\ +0000\ 0101\ (5) \\ \hline 0000\ 1111\ (15) \end{array}$$

But what about the following:

$$\begin{array}{r} 1100\ 0000\ (208) \\ 1101\ 0000\ (192) \\ \hline 0001\ 1001\ 0000\ (400) \end{array}$$

The sum of 400 won't fit in an 8 bit memory space.

The integer overflow is not necessarily a good vehicle for outside attacks. But if your application doesn't return an exception error, your data integrity might be a little off. In this case, you might end up with a value of 144 (1001 0000) instead of 400 in your database or in your next processing step.

## The Danger of Buffer Overflows to Web Apps

So now that I've covered the dangers of buffer overflows, let's get down to the business of assessing the real-world risk of buffer overflow vulnerabilities in custom developed web applications. There are three principle factors that determine the vulnerability of a web application environment to buffer overflow exploits:

1. Programming languages used
2. Vulnerabilities associated with the underlying infrastructure
3. Programming processes and techniques

## Programming Languages Used

In order for programmers to inadvertently introduce buffer overflow vulnerabilities into your custom web applications, they must use languages that rely on them to allocate memory and define data types. Development environments like Java and .NET don't allow programmers that much latitude. This makes applications written in Java or .NET all but immune to buffer overflow issues. The following table lists common programming languages/environments and whether they're safe or unsafe relative to introducing overflow risks:

* Language/Environment	* Compiled or Interpreted	* Strongly Typed	* Direct Memory Access	* Safe or Unsafe
* Java, Java Virtual Machine (JVM)	* Both	* Yes	* No	* Safe
* .NET	* Both	* Yes	* No	* Safe
* Perl	* Both	* Yes	* No	* Safe
* Python - interpreted	* Interpreted	* Yes	* No	* Safe
* Ruby	* Interpreted	* Yes	* No	* Safe
* C/C++	* Compiled	* No	* Yes	* Unsafe
* Assembly	* Compiled	* No	* Yes	* Unsafe
* COBOL	* Compiled	* Yes	* No	* Safe

**Table 1: Buffer Overflow Risks by Language/Environment**  
(OWASP, 2006)

Looking at this table, it's apparent that the languages and environments most commonly used for web application development today (e.g. Java, .NET, Perl) are safe. This doesn't mean that using .NET technology, for example, makes you completely immune. Errors in the programming language or development environment itself might introduce one or two overflow problems. For this reason, make sure you're always using the most current release.

## Vulnerabilities Associated with the Underlying Infrastructure

Now you've begun to feel a little safer—a little less paranoid—because your web applications are all written in Java... not so fast. You're not on high ground just yet.

Those reasonably safe web applications have to run in OS's which in turn run in a network environment full of services and utilities that are probably still written completely or partially in C, C++, or another low level language. Some of the weak points in your buffer overflow defenses might include:

- Microsoft Windows
- Linux
- Apache
- IIS
- MySQL
- Oracle

- And one of a thousand other tools and applications that help deliver web services to your users

Consider not using any product provided by a vendor who doesn't react quickly to announcements about buffer overflow vulnerabilities. And of course, keep your applications and operating systems patched.

## **Programming Processes and Techniques**

In the final analysis, it still comes down to how aware you and your development team are about the proper management of input into your web application environment. One common mistake organization's make is relying on the safety of an environment like .NET while developing applications that call external tools and applications that are written in unsafe low level languages like C.

The primary reason many utilities and resource intensive applications are still written in low level languages is to take advantage of faster execution times. Yes, you might have to give up a little response time when you move to a higher level language. You'll have to decide whether the reduction in risk is worth the tradeoff in performance. If you must maintain higher performance levels, consider selecting products from vendors who demonstrate their processes--both manual and automated--for identifying and eliminating buffer overflow vulnerabilities.

## **So What Does This All Mean?**

Although the risk still exists that your custom web application might fall victim to a buffer overflow attack, it isn't very likely. This is supported by the fact that the Web Application Security Consortium (WASC) has been unable to list any media reports that describe a successful buffer overflow attack against a web application (<http://webappsec.org/projects/whid/>).

According to Jeremiah Grossman, founder of WASC, organizations are better off focusing on the vulnerabilities that are known successful targets or weaknesses in web applications, including (2006):

- Cross-site Scripting
- SQL Injection
- Authentication/Authorization Loopholes
- Business Logic Flaws

Before I close out this topic, I'd like to leave you with a list of things you can do to protect yourself from buffer overflow vulnerabilities no matter where they might be found or potentially introduced into your environment. (And most of them are a good idea even if you're not worried about buffer overflows...)

- Check your buffer accesses by using safe string and buffer handling functions. For example, use [strncat](#) instead of [strcat](#). Use [strncpy](#) instead of [strcpy](#).
- Use operating system level buffer overflow defenses like the compiler switch [/GS](#).
- Keep your development environments up to date. More defenses are added with every release.
- Keep your OS and other underlying software infrastructure patched.
- Use high level programming languages and development environments whenever possible.
- When using low-level languages, make sure to audit the code for potential data type or memory allocation errors.
- Consider non-executable stack technology. This is potentially helpful in preventing certain types of stack overflow attacks. However, it can introduce compatibility issues. For example, programs written in Java that compile code on the fly will most likely fail to run properly.
- Train your developers. Make sure they ask the right questions when writing applications or selecting tools.

---

Copyright 2006 Thomas W. Olzak. Tom Olzak, MBA, CISSP, MCSE, is President and CEO of Erudio Security, LLC. He can be reached at [tom.olzak@erudiosecurity.com](mailto:tom.olzak@erudiosecurity.com) or by visiting <http://adventuresinsecurity.com>

Check out Tom's book, [Just Enough Security](#)

Additional security management resources are available at <http://adventuresinsecurity.com>

Listen to Tom's podcasts at <http://blastpodcast.com/viewpodcast.html?id=441>

Free security training available at <http://adventuresinsecurity.com/SCourses>

---

## **Works Cited**

Grossman, J. (2006). *Myth-busting web application buffer overflows*. Retrieved July 24, 2006 from [http://www.whitehatsec.com/articles/mythbusting\\_buffer\\_overflow.shtml](http://www.whitehatsec.com/articles/mythbusting_buffer_overflow.shtml)

OWASP (2006). *Buffer overflows*. Retrieved July 24, 2006 from [http://www.owasp.org/index.php/Buffer\\_Overflows](http://www.owasp.org/index.php/Buffer_Overflows)

## **Other Resources**

[Understanding Buffer Overflows](#)