Preventing Buffer Overflows (for C programmers)

Author: Jedidiah R. Crandall, crandaj@erau.edu

This Document was Funded by the National Science Foundation
Federal Cyber Service Scholarship For Service Program:
Grant No. 0113627

Distributed July 2002

Embry-Riddle Aeronautical University • Prescott, Arizona • USA

Purpose of this Section

This section is intended for the experienced C programmer who would like to learn more about the causes and consequences of various kinds of buffer overflows.

- It is not intended to be a complete list of every known type of buffer overflow.
- This section should give the reader a broad enough view of buffer overflows that they appreciate the complexity of the problem and don't assume that their code is safe just because they do bounds checking.

What can cause buffer overflows?

- Careless use of buffers without bounds checking.
- Formatting and logical errors.
- Unsafe library function calls.
- Off-by-one errors.
- Old code used for new purposes (like UNICODE international characters).
- All sorts of other far-fetched but deadly-serious things you should think about.

Careless use of buffers without bounds checking - Problem.

This is the classic case and the easiest to prevent. Remember that C/C++ doesn't do automatic bounds checking for you. If you declare an array as **int A[100]** there is nothing in the C language to stop you from executing a statement like **A[555] = 1234;**

You don't need to access an array with an invalid index to have a buffer overflow. Pointer arithmetic is an equally likely culprit.

Careless use of buffers without bounds checking - Consequences.

- If the buffer overflow is big enough the attacker can hijack the machine. For example, in UNIX a buffer overflow of less than 50 bytes in a process that has root privileges can be used to "spin a shell." This means that the attacker obtains a command shell with root privileges. Hijacking the machine can also be done by a worm as it spreads. **But never assume that small buffers, even if it's a two byte buffer, are safe because attack code can be placed in another buffer, beyond the return pointer, or on the heap.**
- Any security sensitive data that follows the buffer can be overwritten, such as passwords or variables that designate privileges.
- The software might crash. This can cause a core dump giving the attacker access to any security-sensitive data that was in the program's memory at the time of the core dump.

Careless use of buffers without bounds checking - Recommendations.

- Before you copy to, format, or send input to a buffer make sure it is big enough to hold whatever might be thrown at it.
- Testing should catch most of this kind of buffer overflows. If there is a buffer overflow, the software should crash or data should get corrupted if a very long string is given for input.

Formatting and logical errors – Problem.

The size in bytes of the input might not be what causes the buffer overflow, it might be the input itself.

- For example, if you're converting a large integer to a string (maybe in ternary) make sure the buffer is long enough to hold all possible outputs.

Formatting and logical errors – Consequences.

- Even if the attacker has very little control of the data that overwrites a return pointer, they can always crash the program by sending the program control to random places in memory.
- Crashing the program is a security risk for many reasons, including denial-of-service attacks and core dumps of security-sensitive data.
- It's never safe to assume that a clever attacker can't find a way to give input that causes the output he wants.

Formatting and logical errors – Recommendations.

- Always test a variety of inputs to make sure the program behavior is what you expect.
- Code inspection is likely to catch buffer overflow errors that testing doesn't.
- Assume that ALL buffer overflows are security problems.
- Don't assume that all buffer overflows are caused by long strings.

Unsafe library function calls - Problem.

Unsafe library functions are one of the main constituents of the buffer overflow problem. Even simple ones like **printf()** have caused buffer overflow security problems.

The problem is that many library functions don't do bounds checking unless explicitly told to, and also many **stdio.h** functions use format strings which opens the door to all sorts of weird exploits.

Unsafe library function calls - Consequences.

- Most library function call buffer overflows allow the attacker to hijack the machine using stack smashing.
- They also can corrupt security-sensitive data or crash the program.

Unsafe library function calls - Recommendations.

- Never, ever, ever use **gets()**. Only under freak conditions will it NOT cause a buffer overflow.
- Also avoid functions like **strcpy()** and **strcat()**. Use **strncpy()** and **strncat()** instead.
- Use precision specifiers with the **scanf()** family of functions (**scanf()**, **fscanf()**, **sscanf()**, etc.). Otherwise they will not do any bounds checking for you.
- Be careful with **sprintf()**. Use precision specifiers or use **snprintf()** instead.
- Never use variable format strings with the **printf()** family of functions.
- Every file or path handling library function has its own quirks, so be careful.
- Functions like **fgets()**, **strncpy()**, and **memcpy()** are okay, but make sure your buffer is the size you say it is. Be careful of off-by-one errors.
- When using **streadd()** or **strecpy()**, make sure the destination buffer is four times the size of the source buffer.
- A very useful tool to aid with finding unsafe library function calls during code inspection are static analyzers such as ITS4.
- Testing will catch many, but not all, buffer overflows. Code inspection in combination with testing will produce the best results.

Off-by-one errors - Problem.

- Off-by-one errors occur when a programmer takes the proper precautions in terms of bounds checking, but maybe puts a 512 where she should have put a 511.
- Can happen to the best programmers no matter how well-informed they are about buffer overflows.

Off-by-one errors - Consequences.

Usually off-by-one errors can do no more than crash the program. They can be made to compromise security-sensitive data. But any buffer overflow is a security risk.

Off-by-one errors - Recommendations.

- If you have a 512 byte buffer you can only store 511 characters in the string (the last character is a **NULL**).
- If you use **scanf()** to read into a buffer you also have to account for the NULL: use **scanf("%511s", &My512ByteBuffer)** instead of **scanf("%512s", &My512ByteBuffer)** which is unsafe.
- If you declare an array as **int A[100]**, remember that you can't access **A[100]**, the highest index you can access is **A[99**] and the lowest is **A[0]**.
- The best defense against off-by-one errors of any kind is a thorough combination of testing and code inspection.

Old code used for new purposes - Problem.

Often old code is reused in new projects. Even if the old code was thoroughly tested and written in a safe manner, it might not have accounted for things that the new code expects it to support, like international character sets.

Old code used for new purposes - Consequences.

"HELLO" in ASCII is 0x48-0x45-0x4C-0x4C-0x4F

"HELLO" in UNICODE (supports international character sets) is 0x00-0x48- 0x00-0x45-0x00-0x4C-0x00-0x4C-0x00-0x4F

- The old code might tell the new code to give it no more than 5 characters because it uses a 5-byte buffer. The new code gives it 5 characters, but in UNICODE instead of ASCII, so they fill 10 bytes. (The assumption that 5 characters = 5 bytes is a dangerous one.)
- This is more common and more easily exploitable than you might think. The Venetian exploit can hijack a program with a reasonably sized buffer overflow even if UNICODE format forces the attacker to have half of his attack code bytes be zeros.

Old code used for new purposes - Recommendations.

- Enumerate and challenge all assumptions you've made about the interaction between old code and new.
- Test thoroughly.
- Test old code when you're using it for new purposes, even if you tested it before. If your software allows the user to use UNICODE then do all of the testing you did for ASCII with UNICODE as well.
- Include the old code in code inspection, even if you inspected it before.
- Test code on every type of platform it will likely be used on. Depending on how the processor arranges memory you might have an off-by-one error of a single byte that has no effect on program execution for a Sun processor but would have a noticeable effect on program execution for an Intel processor.

All sorts of other far-fetched but deadly-serious things you should think about - Problem.

- Sometimes seemingly reasonable assumptions are just not true.
- Attackers have plenty of time and infinite creativity.
- A thoroughly tested and inspected piece of software might still be vulnerable through a series of a half dozen or so clever tricks.

All sorts of other far-fetched but deadly-serious things you should think about - Consequences.

Your software might be a UNIX utility that spawns two processes.

One sets an environment variable to either “CHUCKY” or “CHEESE”, and the second reads it.

The reading process doesn’t bother to check the size before it puts it in a buffer because it is just an environment variable you made up and is guaranteed to have six characters, right? There is no user I/O involved. But an attacker can force a race condition that changes the environment variable between when one process writes it and when the other process reads it. They give the environment variable more than six characters causing a buffer overflow. (Add getenv() to the long list of dangerous library functions.)

All sorts of other far-fetched but deadly-serious things you should think about - Recommendations.

- Challenge all of your assumptions like an attacker would.
- Never assume that a well inspected and thoroughly tested piece of software is absolutely defect free. As long as programmers use C there will always be buffer overflows, hopefully just not as many.

About this Project

This presentation is part of a larger package of materials on buffer overflow vulnerabilities, defenses, and software practices.
For more information, go to: http://nsfsecurity.pr.erau.edu

Also available are:

- Demonstrations of how buffer overflows occur (Java applets)
- PowerPoint lecture-style presentations on an introduction to buffer overflows, preventing buffer overflows (for C programmers), and a case study of Code Red
- Checklists and Points to Remember for C Programmers
- An interactive module and quiz set with alternative paths for journalists/analysts and IT managers as well as programmers and testers
- A scavenger hunt on implications of the buffer overflow vulnerability

Please complete a feedback form at http://nsfsecurity.pr.erau.edu/feedback.html to tell us how you used this material and to offer suggestions for improvements.