Global Information Assurance Certification Paper

Deploying a Secure Web Application: From a Coding Perspective
Jaime Spicciati
May 6, 2003
GSEC Practical v.1.4b

Abstract:

The purpose of this document is to give a developer a very detailed and reproducible guideline for the development of a typical web application. The focus will be on common flaws that recently emerged in popular web applications. This guide will summarize and detail information regarding login page flaws, SQL injection, cross-site scripting/tracing, session ID hijacking and input validation. All of these vulnerabilities will be discussed from a coding perspective and will contain examples of secure implementations that avoid vulnerabilities. The focus is specifically on the coding aspect of development and can be used as a how-to guide for developing a secure web application.

Introduction:
More often then not most all software development is performed in a time critical situation where functionality is the main premise. This very common but unfortunate occurrence causes security to be an "add-on" feature to what may be a very complicated and already fragile application. In any realm of software development there are numerous vulnerabilities released every day. This type of occurrence is primarily not due to laziness of the programmer but instead the product of time critical development. Regardless of the catalyst it is still possible to develop a secure web application if the vulnerabilities are well understood. From this point we will jump into what makes web applications so vulnerable, which of course is user interaction.

1. INPUT VALIDATION: HANDLING WHAT THE USER CAN GIVE

1.1. INTRODUCTION: WHY INPUT VALIDATION IS NECESSARY

Any application that does not require user input is not going to be a very useful product. In the realm of web applications user interaction typically comes in the form of text based input. This input is for the most part the front door of the application, or the typical location a hacker would first look at for vulnerabilities. Typical attacks will include everything from abusive use of login values, hidden values, text boxes or any other item that ever reaches the server. It is only safe to assume that any information that comes from the client is already compromised and cannot be trusted. This first section will lay the foundation for many of the sections that follow.

On the surface input validation is a very simple concept, remove any characters that would interfere with the SQL backend, the HTML interface or any intermediate languages that make the web application run. The majority of the attacks that will be discussed in this paper are the result of missing input validation. Keep in mind that

this front line of defense in no way replaces any other level of defense. A single textbox input that is left unchecked represents the weakest link of the application and will be attacked to the fullest extent. For this reason rely on input validation as step one in achieving defense in depth.

To introduce this section the following table is presented that details some of the more common characters that can be flagged as dangerous

Ascii Representation	Hexadecimal equivalent	Description of attack
<	%3C	Attacks range anywhere from HTML attacks to <script> attacks
>	%3E	Attacks range anywhere from HTML attacks to <script> attacks
“	%22	Used in many cross site scripting attacks to alter the HTML returned to the user (See Cross Site Scripting)
%	%25	Can be used in conjunction to a Hexadecimal value so that it is translated into its equivalent. IE %20 is the same as ‘ ‘, or a space.
‘	%27	May be used to alter HTML returned to user or for SQL attacks (See SQL Injection)
-	%2D	By using ‘- -‘ SQL code can be escaped causing any number of attacks to be run (See SQL Injection)
(	%28	Can be used to call javascript functions or alter server side code (See Cross Site Scripting)
)	%29	Can be used to call javascript functions or alter server side code (See Cross Site Scripting)

Table 1.0 Common characters that should be avoided.
(http://www.mikezilla.com/exp0012.html is an excellent reference for ascii conversions)

As can be seen from the table there are two representations for each character, the ascii value and the hexadecimal equivalent. The middle column includes characters that have not yet been decoded. Once the server or client decodes these items they will become the ascii representation found in the left column. The user can submit the characters presented in two different ways, by the straightforward method

```
<script>alert("hello")<script>
```

or by the much more complex method of encoding the submission

```
%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%22%68%65%6C%6C%6F%22%29%3C%73%63%7
2%69%70%74%3E%20
```

Both of these submissions have the same affect on the server but the second method requires that the text be decoded. The important fact regarding these two representations is that depending on the implementation of the backend the hacker has the ability to submit attacks in many different forms. The following section will discuss the correct manner in which user input should be handled to avoid any such attack.

1.2. IMPLEMENTATIONS FOR INPUT VALIDATION

There are two basic approaches to limiting user submission, or rather validating input. The first approach is simply checking the input for specific characters, in the event the input contains invalid characters, return the error to the user and explain what characters are not allowed. The alternative to this approach is removing the characters and simply not notifying the user. When removing characters keep in mind that the user may notice inconsistencies and that new commands may appear from the manipulated user input. For example, a hacker submits the following

Password: '<dr>"o%p> u&sert>able<->-

Under normal circumstances this password would be very secure. The problem arises when the special characters are removed, resulting in a password of

Password:"'drop usertable"

If this input is not handled correctly then the command has the potential of dropping the "usertable" from the database (See SQL Injection). This example is very specific to SQL but rest assured the possibilities are endless and no backend implementation is completely safe. For this reason it is best to implement security via simplicity, return the string to the user and let the user fix their own mistakes. By avoiding a complex set of tests and logic statements a very simplistic validation can be performed, accept or deny.

The implementation of this type of defense is very simple and should at no costs be forgotten. A very scalable solution to this problem is creating a static method that searches for any disallowed characters (Be sure to include '%' in the characters that are not allowed, this will remove the chance of attacks that use hexadecimal values). Attach this function call to all text input and the result is a very solid first line of defense. Notice that the characters being searched for in our example include characters not previously mentioned. The reason for this is because there is simply no reason to allow characters that are not necessary, such as non-alphanumeric characters.

Depending on backend implementations and new vulnerabilities found everyday it is better to avoid any problems down the road then have to go back and make changes as the vulnerabilities are discovered. Another important note regarding this implementation is that the verification process is done server side instead of on the client. It is very easy to manipulate these parameters on the client side and avoid any restrictions that might be placed in the HTML sent to the client. Client side validation can be performed for server efficiency but always be sure that the server implements a function that is similar to the following:

```
InvalidChars = "~`!@#$%^&*()-+={}[]['";:/?.>,<";

      Vector validateInput(String str2Test)
        {
            Vector errors = new Vector(0);
            for(int i =0; i < invalidChars.length(); i++)
            {
               if(str2Test.indexOf(invalidChars.charAt(i)) != -1)
                {
                  errors.add("Input may not contain the character " +
                  invalidChars.charAt(i) + ", please change before
                  submitting");
                }
            }
            return errors;
        }
```

During the development of a web application it is very possible that at least one input will not be verified. For this reason it is best to implement a safety feature to protect innocent users from script attacks (See Cross Site Scripting). When transmitting the HTML back to the user, call a method similar to HTMLEncode. This method will take all dangerous characters such as '<' or '>' and convert them to their not so effective counterparts, '<' and '&gt'. Therefore "<script>" will be converted to "<script>", which does nothing more then display "<script>" to the user. This method will avoid any HTML defacing or cross site script attacks that might be launched. This also applies to URLs, HTMLEncode has a similar method called URLEncode, which will foil hackers attempts at launching attacks on trusting users by adding a malicious URL to their submission.

In summary input validation is not the solution for security, but can be thought of as a very solid front line of defense.

2. GIVE THE HACKER AS LITTLE INFORMATION AS POSSIBLE

Now that the input is being filtered the next security item that will be discussed has to do with user login. The login page is typically the page that a hacker will first scan for flaws, such as left over comments, significant hidden values, or just about anything else that would show improper coding. Any application that has a login page is going to be susceptible to brute force or dictionary attacks. To further complicate the matter there are numerous applications out there that can be configured to attack these login pages. The simplicity of these applications allow just about anyone to attack your site.

For this reason just assume that everyone with a computer and a mouse is going to be attacking your site. Generally when a hacker is trying to gain access there are three key flaws that will quickly gain the hackers attention, something that should be avoided at all costs. The following section will highlight these flaws and detail how they may be avoided.

2.1. REVEAL AS LITTLE AS POSSIBLE DURING LOGIN

A common mistake during development is trying to make the login page more user friendly. A typical login page consists of a username and a password field. If a user enters an invalid username the following appears:

> Invalid Username, please check for typos or misspelling.

Once the user has determined their username they then move on to the password, which is also incorrect:

> Invalid password, please check for typos, misspelling or caps Lock.

This very user-friendly interface is heaven for hackers. First of all security through obscurity is not security at all, but in this case the level of obscurity grows exponentially. From ground zero the hacker does not have a username or a password. Therefore if the restriction is that the username must be 4 characters(A-Z) and the password must be 4 characters(A-Z), then there are 8^26 minimum possibilities that the hacker must brute force, which is a very large number. If the hacker is able to determine a valid username then he can attack the username and password independently, meaning that there are only 4^26 possibilities for the password and 4^26 possibilities for the username, 4^26 + 4^26 combinations. This is not obvious at first but the hacker that is able to determine a valid username and then attack the password now has to go through 302231445896458038935552 fewer brute force attacks. Keep in mind that it may be an accident that the invalid username page is different from the valid username page. If say for example the hacker finds a valid username and does not visually see a difference between the invalid username page and valid username page. But upon closer inspection through a comparison tool the invalid username page contains an additional space in one of the HTML tags. This additional space, provided through the logic of the application, has just opened up the doors to a brute force attack that greatly diminishes the number of brute force attacks necessary to gain access to the application. In totality, keep things simple and don't give the user more then need to get their job done.

2.2. AVOIDING ATTACKS THROUGH ACCOUNT LOCKING

The second means of securing a login page is through account locking. This is when a user tries to login and after any number of invalid passwords the account is locked. This type of approach is very powerful but at the same time has many drawbacks. In

order for the account to be locked the account name must be known. This can stem from three distinct options 1) It is a valid user who has simply forgotten their password 2) It is a hacker who knows that usernames are of a particular format, say employee IDs or lastname_firstname 3) It is a hacker who used the method described above (Differences in invalid username page and valid username page). A majority of the time the legitimate user is going to be the typical person that is locked out. For this reason it is best to implement a timeout mechanism such that the account will be automatically reactivated after a given amount of time.

When implementing the account locking feature be sure not to use hidden values or store any other important information in the HTML that is sent to the user (An example would be storing invalidLoginsCount in the HTML of the page sent to the user, which keeps track of the number of invalid user submissions). Again, it is best to assume that anything that comes from the client has already been compromised.

The most effective means of limiting a hacker is to not give the number of invalid login attempts a timeout value. What this means is that if the last three login attempts are invalid then invalidate the account. This is in contrast to invalidating the account if there are three invalid login attempts in a 1 minute time span. When attaching a time span to invalid login attempts the hacker can determine the time threshold and effectively set the automated hacking application to waste no time in attacking the login page. On the otherhand, if the account locks after 3 failed attempt regardless of time, then the automated applications effectiveness will be greatly diminished. If a lockout timespan must be implemented then make sure that a brute force application could not guess the username/password before the next time the user's password expires.
Keep in mind that by locking the account the hacker can easily run a Denial of Service attack on the server. This is especially easy in the event the usernames are employee IDs and the lockout time is extremely long. By simply scanning through employee numbers and submitting however many number of invalid login attempts the hacker has effectively denied service to all legitimate users (See "A Guide to Building Secure Web Applications" for more details).

Another major vulnerability to this approach is very comparable to the attack that precedes this. By telling a hacker the account has been locked it is possible for the hacker to gain a list of legitimate usernames. Once the list is sufficient the hacker can then run a dictionary attack on the server with a list of valid usernames. From this point the hacker will know the time threshold for account locking and thus can run the dictionary attack until a user with a weak password is found.

2.3. THE FORGOTTEN PASSWORD 'FEATURE'

The last major vulnerability of the login page is the very common forgotten password link. Just about any login page will have a link that can either reset the password or email the new password to the user. Depending on the implementation the forgotten password link can be a very dangerous feature for many reasons. First of all

the password that resides on the server should always be encrypted in some way, generally these passwords should be hashed using a secure algorithm such as SHA-1. For this reason the password will not be capable of being decrypted, and thus cannot be mailed back to the user. When using a two way encryption algorithm it would be possible to email the password to the user, but keep in mind that it is going to be sent in clear text format across the network. An easy attack on this system consists of sniffing packets on the network after the hacker has asked the server to email the password out. Because most users use the same password on many systems the hacker has just opened up the door to hacking every site the user may interact with. Similar to the Denial of Service attack mentioned earlier, user accounts can be restricted by simply requesting that each employee ID would like to have their password reset, forcing all users to log back in and reset their password. The best solution to all of these flaws is simply storing three pieces of information during account creation, the username, password, and the email address. In the event the user forgets their password have the user first verify their email address and then send a link to that address allowing the user to change their password. This approach will not only stop Denial of Service attacks but will also limit the problems associated with a hacker sniffing a cleartext password out of an email and using that password on other sites the client user might visit. If the hacker can get a hold of a users password then generally they have access to all of the users other accounts.

In summary these three guidelines will greatly enhance the security and robustness of the web application. Another important point to keep in mind has to do with strong passwords. Most dictionary attacks are going to take weak passwords and append numbers or special characters to the end, this is exactly what most users will do as well. The following section will address what occurs after the login page, which has to do with generating a unique session ID.

3. SESSION ID HIJACKING

Once the user has been identified as a legitimate user the next step is to uniquely identify them with a session ID. The session ID is basically an ID that allows the user to avoid authenticating all future actions. Session ID hijacking is going to usually be a brute force attack on guessing any ID that may be running at the time of the attack. As can be thought the only means of security of a session ID is through obscurity. For this reason the session ID must be chosen very well in order to prevent a hacker from guessing it. In the following section we will address 5 steps to creating a secure session ID, all of which will not remove the possibility of session ID hijacking, but will do a decent job in preventing it.

3.1. STEP ONE: GENERATING LARGE SESSION IDS

The first step towards a secure session is very simple; make the range of session IDs very large. By increasing the size of session IDs the hacker will be required to cover more NULL session IDs in the same amount of time. Say for example the session ID is

only 5 digits, this means there are 5^9 possible session IDs, 1953125 total. In contrast to this imagine a 10 digit session ID, 10^9 possible session IDs, 1000000000 total, 5 additional digits equate to 998046875 more possibilities. As more and more users log into the system more session IDs will be live at any given time. For this reason be sure that the chosen session ID size is large enough to support future growth. A system with 10 users is going to have a much smaller session ID size then say an application with 10000 users. Another point to keep in mind when generating session IDs is the importance of the information being transmitted. A banking application is going to use a larger session ID base then an application that simply allows users to send greeting cards.

3.2. STEP TWO: MAKE SESSION IDS AS RANDOM AS POSSIBLE

On a computer that uses binary as its logic device, there is really no way to generate a fully random session ID. The idea behind randomizing is that a random session ID is not a session ID that is incremented sequential by X number of digits. This type of generating can easily be hacked with tools such as Web Sleuth, capable of generating session IDs of any form in the order of X to X+N. Therefore when generating session Ids use some random function that the language supports. Never hard code an algorithm that is going to statically change session IDs by some predetermined amount.

3.3. STEP THREE: TWO FORMS OF SESSION TIMEOUT

A unique session ID has been generated and we have great confidence that the IDs are not sequential and are large enough for a hacker not to easily guess them. The next step is setting a time limit on how long the session ID is good for. Any session ID can be guessed if there is no expiration on the ID. Again, we are creating security through obscurity, this should be avoided at all costs but in this context we have no other choice. Therefore if the session ID generated never expires the hacker can run session ID scanners all day and all night, eventually coming across all users that have ever interacted with the system.

There are 3 basic instances that should cause the session ID to be invalidated. The first is when the account is inactive for a given amount of time. Again this is going to depend on the context of the application, a banking application should have a much shorter timeout session then that of a greeting cards application.

The second instance in which the session ID should be invalidated is the most obvious, when the user logs out. Once the user has logged out their login information should be purged.

A third option that can be implemented specifically deals with systems that have their users logged on for prolonged periods of time. Such a system could simply invalidate the session ID and then generate an entirely new ID. All of this would happen without the user ever knowing. This type of approach would not be appropriate for short-term

session IDs. This is because the application will be giving the hacker more opportunities to guess the session ID as it is continually changing. In a system where users are constantly logging on and off there would be a decent chance that a the new session ID might fall in the range of the session IDs being scanned at that exact time. By continually changing each users session ID while they are logged in there would be a higher probability that the scanner happens to scan the new session ID while it is live. If the application has users that are logged on for days at a time then the hacker can run multiple scanners on the web server and consistently retrieve session IDs.

3.4. STEP FOUR: REQUIRE REAUTHENTICATION

The last measure that should be taken to help avoid devastating session ID hijacking is revalidating the user during important actions. An example of this is when a user is transferring 1000 dollars from account A to account B. This type of action should not rely on a session ID alone, the application must take it upon itself to insure that the user is really who they say they are. In order to provide this level of security require the user to revalidate the username/password combination. In the event a hacker is able to guess the session ID they are limited to only seeing the information, but cannot in any way interact or view secure items such as passwords or PIN numbers.

3.5. SUMMARY OF SESSION ID HIJACKING PREVENTION

From this section we have gathered 5 key points regarding session Ids, these include 1) Choosing a large set of numbers from which the session ID is generated 2) Use truly random session IDs and not sequential algorithms 3)Timeout of session ID when user is inactive 4) Timeout of session ID when user logs out 5) Force the user to revalidate during important transactions. From these guidelines it is possible to avoid session ID hijacking but not to stop it. The idea here is that session ID hijacking is very easy to perform given the set of tools available on the market. For this reason use defense in depth to achieve a secure session, and do not rely only a few of these guidelines, use them all. In the event the hacker is able to guess a valid session ID then stop them in their tracks by making them revalidate the username/password. Think of this process as security through obscurity with a safe fallback in the event the hacker beats the obscurity.

An additional note regarding session IDs is that of booby-trap session IDs. When developing an application it is possible to designate specific IDs as invalid, basically using them as honey-pots. In the event one of these invalid IDs is requested the corresponding IP address would be prohibited from future requests. This type of approach should be a last step implementation following the 5 guidelines presented earlier.

The next few sessions will go into detail about specific attacks that may be launched against your application. Keep in mind that much of this can be avoided by simply performing thorough input validation.

4. CROSS SITE SCRIPTING, CROSS SITE TRACING AND COOKIES

With sufficient input validation cross-site scripting is an attack that can very easily be avoided. Cross-site scripting does not attack the server but instead launches an attack on the users that will be interacting with the server, the clients. These attacks include: 1) Executing of malicious code on innocent user's machines 2) Extracting innocent users session ID 3) Redirect users to a malicious server that mimics expected server (such as login page of bank application) 4) Execute malicious code with innocent user's permissions. By placing specifically crafted function calls into the parameters that are sent to the server it is possible for other users to load these cross scripted pages that will relinquish personal information regarding the client machine. If the client machine has scripts enabled the hacker will be able to execute a script on the client machine that manages to violate the browser's domain restrictions and allows cookie content to be sent back to the attackers computer. For this reason it is extremely imperative that no important content such as passwords be stored into the cookie on the client machine. Session IDs can also be stolen from client side cookies, another reason to revalidate the user when performing important transactions on the server (See Session Hijacking). Cross site scripting is easily avoidable as long as the parameters submitted to the server are thoroughly checked for all of the characters that are mentioned in the section detailing input validation. It is also possible for a hacker to send an email to an innocent user that contains a URL that executes when the page is loaded. This type of attack cannot be avoided from a coding perspective but users should be made aware of the fact that they should never trust emails that do not come from the correct domain.

A new vulnerability that has recently been exposed is called cross site tracing. Cross-site tracing uses the TRACE functionality that is built into the HTTP 1.1 protocol. It is possible for a hacker to create a web page that steals passwords and other important information. The strength of this attack is that all servers support the TRACE utility in order to meet RFC standards. There is not much that can be done at this time for this vulnerability but it simply reemphasizes that no important data should be stored on the client machine via cookies or any such device. Also keep in mind when writing HTML pages that all secure items such as passwords or user IDs should use PUT in the form and never use GET. This only pertains to forms but an ill placed GET will store secure information in the browser history or add such items to the URL, allowing easy extraction for a hacker.

5. SQL INJECTION

SQL injection is a very powerful attack that can be devastating to the security of any web application. Any input that that reaches the database is capable of SQL injection attacks. Such attacks can easily be avoided if the appropriate actions are taken, but to better understand SQL injection it is best to look at an example. This example has been adapted from OWASP's guide to building a secure web application.

5.1. SQL INJECTION EXAMPLE

A Web application includes functionality that enables users to change their passwords. To do so, the server presents an HTML form to the user with four blank fields:

```
Username:
Old password:
New password:
Confirm new password:
```

When the user enters the requested information in the HTML form fields, the browser translates the supplied data into an HTTP request, which it sends to the server application:

```
http://www.server.mil/changepwd?pwd=O!dP@sswD&newpwd=5Q1!nject&newconfirmpwd=
5Q1!nject&uid=testuser
```

The server application extracts the four parameters from the HTTP request:

```
Pwd=              O!dP@sswD
Newpwd=           5Q1!nject
Newconfirmpwd=    5Q1!nject
Uid=              testuser
```

It then checks to make sure that *Newpwd* matches *Newconfirmpwd*. After verifying that the two match, the application discards *Newconfirmpwd* and builds an SQL query. The SQL query that is constructed consists of those attributes that are passed into the server. The constructed SQL function consists of the following attributes, where INPUT[XXX] will be the string that is extracted from the request:

```
UPDATE usertable SET pwd='INPUT[Newpwd]' WHERE uid='INPUT[Uid]';
```

Therefore the SQL statement that will be generated and run will be the following:

```
UPDATE usertable SET pwd='5Q1!nject' WHERE uid='testuser'
```

A hacker who knows how this process works will also know he can inject a custom tailored function that will run within the valid SQL statement. He could modify the HTTP request generated by the browser (before that request is transmitted to the server), to include an additional function that changes all account passwords that have a uid that contains the string 'admin'. Here is an example:

```
http://www.server.mil/changepwd?pwd=O!dP@sswD&newpwd=5Q1!nject'%20where%20uid%
20like%20'admin';-- &newconfirmpwd=5Q1!nject'%20where%20uid%20like%20'admin';-
-
```

The server application will then extract the four parameters from the HTTP request:

```
Pwd=              O!dP@sswD
Newpwd=           5Q1!nject'  where uid like 'admin';--
Newconfirmpwd=    5Q1!nject'  where uid like 'admin';--
Uid=              Does not exist
```

Without any input validation the resulting SQL query that is generated will be the following:

```
UPDATE usertable SET pwd='5Q1!nject' where uid like 'admin';--' WHERE
uid='testuser'
```

This very powerful attack will change the password of all users that have "admin" in their uid. The ";--" portion of the submission allows the hacker to ignore any additional SQL code that exists. By commenting out the last section of the SQL code the SQL function will execute without a problem. The result is that the hacker gains unlimited access to the Web server, whereas the legitimate administrators are locked out.

5.2. SQL INJECTION SOLUTION

Many developers will have the initial assumption that by using convoluted names for the table columns, the hacker stands no chance in directly manipulating any stored data. This assumption will attempt at creating security through obscurity, a method that does nothing more then throw a speed bump in front of the hacker. By submitting various queries to the server the hacker can easily determine the structure of the tables that make up the database (www.nextgenss.com/papers/advanced_sql_injection.pdf). This type of approach will use error messages to obtain critical information that may allow the database to be compromised. With specifically crafted SQL injections the hacker can force the SQL server to present column names and column data. Ultimately there is but one solution to avoiding SQL injection attacks, do not generate the SQL function from user input. By using a repository of stored SQL functions it is virtually impossible to compromise the database. These stored SQL functions should use user input as nothing more then a variable of which the selected database item will be compared to. This type of approach will remove any means by which the hacker can run harmful code on the web server. The SQL command should be changed to read the following:

UPDATE usertable SET pwd=@pwd WHERE uid=@uname

Where the pwd and uname attribute are passed into the SQL function as parameters. This type of approach with canned SQL functions will avoid an attack from a hacker where the SQL command is generated from user input. As a second line of defense it is best to implement input validation that will keep special SQL characters from ever reaching the database. Remember to never trust user input, in this case it is very important that the user input be treated as data and nothing more.

6. BUFFER OVERFLOWS

Depending on the backend implementation many web applications may be susceptible to buffer overflows. All languages will have constant interaction with the system memory, heaps, and stacks. Each function call or memory access has a very specific scope for which access is allowed. These boundaries can easily be violated in

such languages as C/C++. Generally when a memory violation occurs there is an access violation exception or a segmentation fault. In order for the invalid access to become a security issue there must be two conditions met:

1) The attacker must be able to control the data written to the buffer.
2) There must be security sensitive variables stored after the buffer in memory. (www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html)

In order for buffer overflows to be a significant security threat there has to be a great deal of time spent by the hacker determining where the buffer overflow may exist and where security related information is stored. Another complicated approach to buffer overflows deals with the position of the instruction pointer. By inserting a small piece of code into memory and relocating the instruction pointer it is possible to run that piece of code within the application address space. There are two options to avoiding buffer overflows. The first option is to use a language that performs bounds checking itself, such as java, Perl, Python or any of the .Net languages. An alternative to this solution is to avoid using specific vulnerable functions in the language. Such functions ignore size limitations or would use a format string for determining what is returned (Note: Never use a format string that is taken from user input, always hard code these items). These functions can be exploited in one of two ways, by sending invalid format strings to the function or by sending overly large strings to the function. In whichever case the application opens the doors to segmentation faults and in some cases security breaches. The following table details those functions that should be avoided:

Function	**Reason to Avoid**
Scanf,sscanf,fscanf,vscanf,vsscanf,vfscanf	There is no check on input length or content
Strcpy	May cause string to no longer be null terminated, causing segmentation fault.
Sprintf	Buffer bounds are never checked, use snprintf
Strcat	Does not checks on bounds, may cause buffer overflow
Gets	Never use, performs no bounds checking

Table 6.0 Dangerous C Function Calls.

In the event security is a last thought there are tools that will automatically scan C source to determine if vulnerabilities do exist. The following tools are available as open source and provide decent recommendations on how to better secure the applications C code.

FlawFinder: http://www.dwheeler.com/flawfinder/
RATS: http://www.securesoftware.com/download_form_rats.htm

For more information about the details of buffer overflows please see the following reference:

www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html

7. SUMMARY

As can be seen each piece of the puzzle fits together to produce a fairly secure web application. By implementing each step there exists a fallback in case one of the items happens to fail or is forgotten. This can best be illustrated through a diagram that breaks down each step of securing a web application. The diagram can be found in the Appendix, Figure 1.0. As can be seen the first level of security is input validation, insuring that cross-site scripting and SQL injection cannot be performed. If the hacker submits data that could cause buffer overflows they are handled appropriately by not using vulnerable function calls. The first level of defense also includes hard to guess session IDs that keep the hacker from getting into the system. If the hacker does manage to guess a session ID he must revalidate before performing important transactions on the secure database. The last step towards a secure web application is sanitizing the HTML sent to the trusted client, this is done with HTMLEncode that will remove all chances of running malicious code on the client machine.

REFERENCES:

1) OWasp. "Top Vulnerabilities in Web Applications."
URL: www.owasp.org (May 4, 2003).

2) Owasp. "The Ten most Critical Web Application Security Vulnerabilities." Version 1.
URL: www.owasp.org (May 4, 2003).

3) Curphey, Mark et al. "A Guide to Building Secure Web Applications." September 11, 2002. Version 1.1.1.
URL: www.owasp.org (May 4, 2003)

4) Advosys Consulting Inc. "Writing Secure Web Applications." June 22, 2002.
URL: http://advosys.ca/papers/web-security.html (May 4, 2003).

5) Lee, Paul. "Cross-Site Scripting".
URL: www-106.ibm.com/developerworks/security/library/s-csscript (May 4,2003).

6) Rafail, Jason. "Cross-Site Scripting Vulnerabilities."
URL: www.cert.org/archive/pdf/cross_site_scripting.pdf (May 4, 2003).

7) Newsham, Tim. "Format String Attacks." Sep 11 2000.
URL: www.lava.net/~newsham/format-string-attacks.pdf (May 4, 2003).

8)ISS. "Alerts". February 1, 2000.
URL: www.iss.net/issEn/delivery/xforce/alertdetail.jsp?id=advise42 (May 4, 2003).

9) Peteanu, Razvan. "Best Practices for Secure Web Development". September 23, 2000.
URL: www.fi.upm.es/~flimon/secure_web_development.pdf (May 4, 2003).

10) Spett, Kevin. "SQL Injection: Are Your Web Applications Vulnerable?"
URL: www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf.
(May 4, 2003).

11) Guninski, George. "Best Practices for Secure Web Development: Technical Details."
URL: www.developer.com/tech/article.php/640891 (May 4, 2003).

12) Klein, Amit. "Hacking Web Applications Using Cookie Poisoning".
URL: www.cgisecurity.com/lib/CookiePoisoningByline.pdf (May 4, 2003).

13) Owen, Tom. "Cross-Site Tracing Attacks". URL: www.lwn.net/Articles/21364/
(May 4, 2003).

14) Mircrosoft. " HowTo: Prevent Cross-site Scripting Security Issues".
URL: http://support.microsoft.com/default.aspx?scid=kb;en-us;Q252985&sd=tech
(May 4, 2003).

15) Frykholm, Niklas. "Countermeasure Against Buffer Overflow Attacks". November 30, 2000.
URL: http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html
(May 4, 2003).

Appendix
Figure 1.0