



WHITE PAPER

Common web application security problems and how to identify them

Part 1

Unvalidated input

By

Lee J Lawson
Lead Penetration Tester, **dns**.

Introduction

OWASP, the Open Web Application Security Project produces a top 10 list of common security flaws with web applications. They describe themselves as “...dedicated to finding and fighting the causes of insecure software” and are a great resource for information about common problems with web applications and secure coding practises.

With this series of papers, I intend to go through the most common flaws, explain what they mean and provide hands on instructions on how to identify if your web application is vulnerable to any of them. This paper is not intended to be used for malicious or illegal activities and only describes how to confirm that security risks are present in a web application and offer guidance on fixing the flaws.

Web applications are now being used as the most common client/server implementation for critical business functions including e-commerce, email and data retrieval. These are the flagship applications for any organisation’s Internet point of presence and attacks against them are both common and successful in an alarming number of instances.

Gone are the days of static HTML web content, applications are now very dynamic, utilising powerful programming languages to produce and format content from back end data sources. Due to the nature of these powerful applications, and their access to the back end data sources, they are the target for many nefarious Internet users. They can provide a wealth of information or access to the internal network infrastructure on which the web server resides.

as web applications become increasingly powerful and complex, they become more likely to contain weaknesses.

As web applications become increasingly powerful and complex, they become more likely to contain weaknesses. These weaknesses may be hard to locate as the application could be constructed of multiple pieces of code spread across many web pages.

I hope to explain how to identify the most common problems and provide some real life examples of security weaknesses I have encountered whilst performing penetration testing services.

The list

The OWASP top 10 list of web application security flaws can be found at www.owasp.org and is updated based on current information. It should be noted that the list changes from time to time and the information in the paper reflects the top 10 list at the time of writing.

8 of the 10 entries are down to the development methodology used by the programmers and not simply a check box to check in the web server/ application server or database backend. The developers of applications should consider security during the 3 D’s – Design, Deployment and Default settings of an application. Security should be baked in and not caked on at the end of

development. In spite of the frequency with which these phrases are thrown around in security conscious circles, they are often little heeded.

Here is the list taken directly from the OWASP site at the time of writing.

1. **Unvalidated input**
2. **Broken access control**
3. **Broken authentication and session management**
4. **Cross site scripting**
5. **Buffer overflows**
6. **Injection flaws**
7. **Improper error handling**
8. **Insecure storage**
9. **Application denial of service**
10. **Insecure configuration management**

It should be noted that this paper is not intended to be used as a replacement to professional penetration testing. Rather it is aimed toward raising the level of security knowledge in the community. Professional penetration testing requires years of experience to be able to apply worthwhile interpretation to results from testing tools and gain the skills required to perform manual assurance testing.

Unvalidated Input

all user input is evil until proven otherwise

Information from web requests is not validated before being used by a web application. Attackers can use these flaws to attack backend components through a web application.

All user input is evil until proven otherwise. This tenet should be considered at every stage of the development life cycle. Just because your application asks a user to enter a username, password or other piece of data, does not guarantee that the user actually will. They can enter any data they like as long as the application permits them.

the damage caused by a successful attack is governed by the application and the input types.

The Problem

This type of weakness is commonly found on web applications that pass SQL statements to a database back end but is prevalent on many systems that allow for user input. The damage caused by a successful attack is governed by the application and the input types. For example, inputting scripting code could lead to Cross-Site Scripting attacks which have their own section due to the level of risk associated with them. Being able to inject a Control / Line Feed may alter a log file which is still a vulnerability but at a lower threat level.

This can also apply to URL addresses that are constructed with a series of values. These values could be selected by the application depending on a number of different variables such as a user clicking on a link, the current user's authorisation and location etc. The intended process is the user allowing the application to create the URL, but an attacker may adjust, change or outright corrupt the URL values causing the application to perform some action that it would not otherwise be permitted to.

Here, we will consider the damage caused by attacks to a web application that passes instructions to a database back end.

To understand the weakness, we should first briefly look at some code that is vulnerable. This example is demonstrating a SQL database application weakness, but the principle applies to many application types and data inputs.

```
“SELECT * FROM logins WHERE user='userinput' AND pass='passinput';”
```

The userinput and passinput are populated from the web page; the username and password input fields. The complete SQL statement is constructed on the web application server (Tomcat, ASP etc) and passed to the SQL database as a whole. Once received by the database server (MS SQL, Oracle etc) it is executed and the results passed back to the web application.

This solution would be fine if every user of the application out there only used it in the way the developer intended and entered a valid username or password. Unfortunately we live in a world where intelligent and inquisitive individuals

understand more about computers and programming than the engineers and developers do. Not only do they understand how the code works, they comprehend the most common implementations of the code and they enjoy finding ways to subvert any security.

Should a user enter valid data, the final SQL statement would look like this:

```
“SELECT * FROM logins WHERE user='lee' AND pass='P@ssword';”
```

After receiving the above statement, the database would execute it under the security context of the web application (more on this a little later). It would open the required table (logins) and look at each record in turn searching for the user called 'lee'. After finding a record that has 'lee' in the user column it would attempt to match the password in the record to the password submitted by the user, if it matches then the user is authenticated and some other code executes which, for example, may forward them onto another page.

Great! So the database is looking for a balance between the user submitted data and the data stored in the table. What would happen if the attacker supplied this balance deliberately?

Instead of entering the expected data, the attacker could enter some SQL code that would manipulate and change the way the complete statement executes.

```
Input= ' or 1=1;--
“SELECT * FROM logins WHERE user="" or 1=1;--' AND pass="";”
```

By carefully entering valid SQL commands, the attacker has changed the way the whole statement is executed on the database server.

The injected code displayed above is probably the most common SQL injection script there is. It works like this:

- ' Closes the user input so that it is blank.
- or Allows the code to match one criteria OR another.
- 1=1 A deliberate 'balance'.
- ; Complete the command.
- Comment out the rest of the line so that it does not execute.

Now, when the database server executes the command, it will go to the table called logins and look at each record in turn. It looks at the user field and says “does the data in the user field match the user input?”, if it does then return this record. As shown above, the user input is blank as denoted by the two single quotes “”, but the statement continues... Does the user input match OR does 1=1? 1=1 is a true statement every time it is executed. This way the server sees a balance but is fooled into thinking that the balance is the one it's supposed to be looking for!

In most cases, if the application uses code similar to the above excerpt, the attacker will be logged on and authenticated as the first user in the logins table and be provided access to the application under their security context.

This is obviously bad enough, but database servers have functionality to pass commands to the underlying operating system. Microsoft SQL server has the extended stored procedure called XP_CMDSHELL, which as the name suggests allows the database service to pass shell commands to the OS. By default, only the Administrator (SA) level accounts have access to this stored procedure. Another major problem with database front end applications is that the web page

instead of entering the expected data, the attacker could enter some SQL code that would manipulate and change the way the complete statement executes.

authenticates against the server with overly powerful accounts such as the SA account. This allows an attacker to pass commands to the operating system and retrieve the results. Imagine if the database server service (MSSQL, Oracle, MySQL etc) ran as a powerful operating system account such as root or Administrator? The default setting for MS SQL server is to run as a local Administrator level account! Oracle and MySQL are commonly installed and configured to run as root or Administrator!

This chaining of events allows an attacker to sit on a beach and pass shell commands to the underlying operating system as a local Administrator level account! This is very, very bad! They would be given full access to upload any kind of backdoor they wanted and add Administrator level accounts themselves.

Identification

Identifying all unvalidated inputs to an application can be time consuming and tedious, but is worth the effort in the long term. Developers can aid this process massively as they have access to the source code and may have developed the application themselves.

the problem with automated scanners is the identification of 'false positives'

The speed of identifying ALL inputs to an application can be increased by utilising automated tools. The problem with automated scanners is the identification of 'false positives'. In this scenario the tool will list 100 discovered inputs yet only 15 of them would be considered 'at risk'. The best method is to use a blend of automated and manual techniques.

Inputs to an application can be via:

- Input fields on the web page.
- Hidden values in the HTML code.
- Data values embedded in the URL.

Once you have a list of ALL input fields, you should attempt to identify if they accept commands in whatever language the application is passing data as e.g. SQL, perl, shell commands etc.

One of the best methods for this is to simply enter a single quote. This would effectively disrupt the complete SQL statement and cause an error to be returned. This error can sometimes be very informative and at the very least gives you an error number. The error number 80040E14 is indicative of a SQL injection point, although other errors may also suggest this case.

If you receive error messages similar to the above then it is highly likely that your

```
Microsoft OLE DB Provider for ODBC Drivers
error '80040e14'

([Microsoft][ODBC Microsoft Access Driver] Extra )
In query expression 'User="" AND Pass ='
```

```
Error Type:
(ox80040E14)
/sqltest.asp, line 12
```

input has been passed to the SQL server within the entire statement.

the best method of defending against unvalidated input attacks is to validate the input!

Countermeasures

The best method of defending against unvalidated input attacks is to validate the input! It sounds simple but is rarely executed. Configure a 'white list' for every type of input meaning that you should filter for what data you want. If a username is constructed of a-z, A-Z and 0-9, then make sure they are the only characters accepted by the application. The other option is to configure a 'black list'; this is less effective as you are attempting to filter out all of the potential attack characters used by hackers. This may be large and unwieldy and worse still it may be ineffective should a new attack vector be discovered at a later date.

Protecting against SQL injection attacks can be simpler based on the fact that they tend to use very specific characters and sometimes a black list approach is feasible. The code below demonstrates how to strip out single quotes from a user input:

```
function blacklist(stringtoconvert)
    tmpstring = replace(stringtoconvert,'"','"')
    tmpstring = "'" & tmpstring & "'"
    blacklist = tmpstring
end function
```

This would be effective against a large majority of SQL injection attacks but the single quote is not the only delimiter for user input. Therefore, a combined 'white list' and 'black list' filter could be employed. Remember that if the filtering code is executing on the client side of the application (the web browser), then an attacker can see, alter and bypass that code easily. Client side validation should be used to speed up the process by locally validating that data has been entered correctly, but server side filtering should be used to defend the application.

This is the first instalment of a series of papers explaining the most common security flaws found on web applications. The level of damage caused by allowing a malicious attacker to pass SQL commands to a database server are severe, especially if a chain of weaknesses exist which allow those commands to be processed as a privileged account.

This paper used the scenario of a SQL injection attack to explain the problem with unvalidated input flaws however this is not the only method or result that can be found with this type of vulnerability. Unvalidated input is a flaw that could lead to many other types of attack such as Cross-Site Scripting, buffer overflows etc. Therefore, by validating all input to an application and verifying that it is 'safe', it is possible to prevent a number of attacks from succeeding.

Other more generic countermeasures can be listed as:

- Code reviews (check input against list of allowed values, 'white list')
- Do not accept unnecessary input from a user
- Sanitise/validate all input

The next paper in the series describes weaknesses with **Broken access controls**.