Standards for Input Validation

Ayesha Bhatti
Linköpings universitet, Sweden
Email: ayesa593@student.liu.se

Abstract

Almost every computer application processes user data in one form or the other. The data may be from a genuine user or it may come from a malicious source. An application that doesn't have any mechanism to recognize the malicious data may process it and that may result in any thing between a simple crash and a server with data loss. This document states different types of input vulnerabilities and the methods of preventing those. Practical part of the report addresses SQL Injection and cross-site scripting (XSS) attacks. It states different types of attacks in these two categories, and also states the regular expressions that can block the stated attack types.

1. Introduction

Almost every computer application processes the data supplied by the user in one form or the other. Data may be login details, filenames, environment variables, web forms, cookies, URLs etc. The normal behavior of a program is to accept the provided data, process it and generate the output. But due to the fact that all users are not good and don't always provide expected input, programs cannot trust any user supplied information. Due to the multi-user nature of computer programs and the fact that most applications are operating on web, the situation has gotten worse and no input is reliable any more to be accepted directly.

By supplying wrong input into a vulnerable program user can do anything to the extent of taking control of the server or steeling whole databases depending upon the type of target application and attack.

SQL Injection is form of attack in which user inputs meta-characters like single quote or single dash. These single quotes break or change SQL queries resulting in deletion of undesired data from database or display of sensitive data of other users. Take an example of following query.

```
SELECT uid, pwd FROM tblusers WHERE uid = 'uid' AND pwd = 'password'
```

If user supplies the following data

```
Uid= '  OR ' =''
pwd= '  OR ' =''
```

The query will become

```
SELECT uid, pwd FROM tblusers WHERE uid = '' OR ''='' AND pwd = '' OR ''=''
```

Now this query can never be negative, it will pick all records from the table and reveal secret information of all users along with their passwords to some one entering the malicious input.

Various real life examples of SQL injection attack can be seen at web hacking database [29]. One of hack states that Russian hackers stole 53,000 credit card numbers from http://www.ri.gov/ through SQL injection attack. More details can be checked from [30].

Hence it is a must for both developers and security personals to learn how hackers can input wrong data, and what they should block in order to make their systems secure.

Objective of the current work is to search for the existing standards or guidelines provided by different security experts or associations for validating input and present them those in an organized form at one place.

The report is structured in three major parts from here on. The first part (section 2) contains the background information about what are the input types and how many are these? What are the vulnerabilities related with each input type? The next part (section 3) states the standards or guidelines for input validation in general and also related to individual input type. Section (4) concentrates deeply on two of the attack forms that are possible through malicious input. Please note that there are other attacks that are possible but this report only concentrates on two of them. The section describes how these attacks can be made possible and also how to stop them using the guidelines stated in section 4. Rest is conclusion and references.

2. Input Types

Two main questions arise when looking at any input for security concerns, where the input is coming from and how secure is the medium of input? If it is coming from a file then how secure is it? If it is coming from a setting in a web-browser then who can do the setting? etc. That means inputs can be categorized on the basis of source or medium. Section 3.1 to section 3.6 describes various inputs types that are categorized on the basis of source/medium and their related problems.

2.1 Environment Variables

Environment variables are the data bits the program reads from the already specified locations. These can be set manually by the system administrator or by the some program.

UIDs are used to control access in Unix based systems. System calls to set these UIDs are known as setuid programs. Setuid type of programs use environment variables a lot. The child processes can change these variables. They are principally inherited from the parent program but due to the fact that child processes can alter them this inheritance works transitively. A single pointer points to list of pointers that stores the environment variables as key value pairs. The list may contain multiple entries with one name and different values. Different systems read the list in different ways, some check the first matching value, and some check the last. If users can add their own environment variables at the end of the list then checking the last value could be a big problem.

Solutions include erasing the environment after extracting carefully desired set of values, not allowing restricted users changing or adding environment variables. [6] provides a good reason from keeping users to mess with environment, by giving example of telnet daemons that support RFC 1408 or RFC 1572. It is possible for a remote computer connected to these daemons to transfer environment variables that may influence the login program. Check [6,7] for more details.

2.2 Files

Files offer multiple problems. Users create files and choose names. These names may include characters, which cause various kinds of problems. Like a preceding dash may be interpreted by another program as option and result in a crash. Spaces are allowed in windows like systems and are not allowed in Unix like systems. Programming scripts may confuse new-line or carriage return characters in the file names as argument separator. Filenames could be strangely encoded, like a program may believe that filename is UTF-8 encoded, however it may be long UTF-8 encoding. Operating system doesn't enforce encoding and hence this is a problem for developers.

Globbing is a feature that is used to obtain a list of files whose names fulfill a certain criteria like *.txt will return all text file. Globbing is a useful feature but at the same time can be very hard on system resources like CPU, and memory. Hence can a malicious user use it for denial of service. David provides an expression that can be used for this purpose in his book [7] that is as follows.

```
  ftp> ls */../*/../*/../*/../*/../
*/../*/../*/../*/../*/../*/../*/../*
```

File contents should also be trusted only in case when they are in control of only trusted users. Otherwise contents are just like direct input, and they can cause all those problems that can be caused by the direct malicious input.

2.3 Web Content

URLs have special encoding known as URL-encoding. According to URL-encoding characters are replaced with their hex values. This introduces an additional complexity layer. Now applications have to convert the hex values to characters and then check if they are valid. There is no trusted user on web, so URLs are no different than any other input.

Cookies are data bits stored on the client's machines by the websites. As they reside on user's machine so these are in total control of client and thus totally un-trust worthy. There are two types of cookies, persistent cookies and non-persistent cookies. Non-persistent cookies last only for single http session, while persistent cookies may last for several days. For a non-technical user it is a bit hard to change a cookie value during an http session so non persistent cookies are relatively safe to use. But changing a cookie value that stays on the user's machine for several days is very easy to change for any user. Hence programs using persistent cookies are at higher risk of processing malicious data.

HTML forms are used to take input from web users. Normally javascript is used to validate the input. But unfortunately this only helps valid users to fill the forms correctly. It doesn't have any role for security as the same data can be sent to web-server directly.

2.4 UTF 8

UTF 8 is variable length encoding for multilingual characters. It is flexible in the sense that it is backward compatible with ASCII and its variable length. It is becoming a standard for multilingual softwares and websites due to ease of use, portability and backward compatibility. However this is another source of security problems. All byte sequences are not UTF-8 valid encoding. UTF-8 encodes are supposed to use shortest possible encoding, however some earlier standards allowed decoders to accept values that are longer. This is a genuine problem for validation routines as potentially dangerous inputs can be expressed in multiple ways. [10] discusses this problem in more detail while RFC 3629 stated this as follows.

"Implementers of UTF-8 need to consider the security aspects of how they handle illegal UTF-8 sequences. It is

conceivable that in some circumstances an attacker would be able to exploit an incautious UTF-8 parser by sending it an octet sequence that is not permitted by the UTF-8 syntax.

A particularly subtle form of this attack could be carried out against a parser that performs security-critical validity checks against the UTF-8 encoded form of its input, but interprets certain illegal octet sequences as characters. For example, a parser might prohibit the NUL character when encoded as the single-octet sequence 00, but allow the illegal two-octet sequence C0 80 (illegal because it's longer than necessary) and interpret it as a NUL character (00). Another example might be a parser which prohibits the octet sequence 2F 2E 2E 2F ("/../"), yet permits the illegal octet sequence 2F C0 AE 2E 2F."

3. Input Validation

The previous section explained various forms of inputs and their vulnerabilities. This section explains what a developer can do in order to validate the input data.

David Wheeler has done detailed work on specific input types and has described solution to validate almost every kind of input in his book [7], and also in various web articles. Section 4.2 onwards states his work. Each section explains what are the important issues to look for in each input type and how to validate it correctly.

3.1 Environment variables

"For secure setuid/setgid programs, the short list of environment variables needed as input (if any) should be carefully extracted. Then the entire environment should be erased, followed by resetting a small set of necessary environment variables to safe values. There really isn't a better way if you make any calls to subordinate programs; there's no practical method of listing ``all the dangerous values". Even if you reviewed the source code of every program you call directly or indirectly, someone may add new undocumented environment variables after you write your code, and one of them may be exploitable." [7]

The other thing David says in this regard is to keep user from creating his own environment variables with the valid argument that they will cross the boundaries of their restricted accounts if they get this option. He also supported his argument with a Bugtraq discussion back in 2002, that I couldn't actually find. However if you are interested you can find the text of that article in his book. [7].

3.2 Files

Try not to allow users to choose file names. If that doesn't work then file names should not contain certain characters as described in section 2.3. Take those and you can also add your own according to your application, and make a regular expression, and use that to validate the user provided name. David suggests the following in [13].

```
^[A-Za-z0-9][A-Za-z0-9._\-]*$
```

When it comes to using contents of some file, check the contents against valid pattern depending upon the application, here David agrees with Matt that one should check the data against valid input.

3.3 Web Content

Do not use persistent cookies. It is good for security and also good for being lawful. If you accept a cookie value, check that its domain value is what you expected, i.e. your own site, other wise reject it.

URL's should be validated against valid pattern.

```
^(http|ftp|https)://[-A-Za-z0-
9._/]+$
```

The above is good for most purposes. But it may alarm false positives on valid URLs as it doesn't allow characters like tilt, exclamation mark etc.

For letting more complex patterns through and yet block attackers the following is useful.

```
^(http|ftp|https)://[-A-Za-z0-
9._]+(\/([A-Za-z0-9\-
\_\.\!\~\*\'\(\)\%\?]+))*/?$.
```

Each and every field of web forms should be checked against the valid patterns according to the type of field.

Checking email addresses is tricky. The shortest regular expression for checking email is 4,724 bytes long(Jeffrey Friedl's has written that) and According to David even that misses some cases.

3.4 UTF 8

The following table contains the valid patterns of UTF sequences. Any value not matching these patterns is not a valid UTF 8 sequence. The First column contains values that are to be UTF 8 encoded. Second column contains their legal UTF encoding in binary and last column contains the hex equivalents of legal UTF 8.

'x' in binary value represent data bits 0 or 1.

UCS Code (Hex)	Binary UTF-8 Format	Legal UTF-8 Values (Hex)

00-7F	0xxxxxxx	00-7F
80-7FF	110xxxxx 10xxxxxx	C2-DF 80-BF
800-FFF	1110xxxx 10xxxxxx 10xxxxxx	E0 A0*-BF 80-BF
1000-FFFF	1110xxxx 10xxxxxx 10xxxxxx	E1-EF 80-BF 80-BF
10000-3FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F0 90*-BF 80-BF 80-BF
40000-FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1-F3 80-BF 80-BF 80-BF
40000-FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1-F3 80-BF 80-BF 80-BF
100000-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F4 80-8F* 80-BF 80-BF
200000-3FFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	too large; and for most purposes illegal
04000000-7FFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	too large; and for most purposes illegal

Table 3.1 Valid UTF 8 sequences, taken from David's book.

For further details if interested see [7] and if some one need all information about UTF encoding and its security issues then see [10] by Markus Kuhn.

4. Case Studies

I did two case studies. First on SQL Injection attack and other on Cross Site scripting. The work is mainly inspired by the Security Focus article by K. K. Mookhey, Nilesh Burghate, but I have practically confirmed it.

4.1 SQL Injection

Standard Query language is used to query databases for data in desired format. For reader not having background on SQL please check [21].

SQL does have its own meanings to certain characters like single quote semi colon dashes etc and also their combinations. Hence these are meta characters in case of SQL as described in 2.7.2 .

This means that these characters should not be used with in the data at all. And this needs to validated by the application at the top of database server.

Now consider the following query.

```
SELECT uid FROM tblusers WHERE uid =
'' OR ''='' AND pwd = '' OR ''=''
```

This query is written for matching the user provided input with some valid username and password in the database but user provided the following information.

```
uid='  OR ' =''
pwd='  OR ' =''
```

Due to this malicious input the query has turned into an expression that would always be true.

As advised by David the first step should be the detection of meta-characters and writing a regular expression. If I consider stopping the above attack my regular expression would be

```
/(\%27)|(\')|((\%3D)|(=))/ix
```

I have checked for single quote hex equivalent because its an HTML meta character and is encoded into hex by the browser.

A more sophisticated form could be the following.

```
/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((
\%72)|r|(\%52))/ix
```

This regular expression can also block OR with all its hex equivalents.

In mySQL # creates problems and in oracle double dash creates problems [22]. SO the regular expression will works there.

```
/(\%27)|(\')|((\%3D)|(=))|(\-\-
)|(\%23)|(#)/ix
```

Double dash's hex equivalent is not included in the expression as its not a meta character in HTML.

User can use semicolon to complicate things. Consider the following query

```
select uid from tblusers where age=userage;
```

Now if user supplies his age as follows

```
5; delete from tblsusers;
```

Whole tblusers would be deleted as a result of this query. This can be blocked by following regular expression.

```
/((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/I
```

The main idea is to block semicolon in user input but semicolon is a common HTML character and blocking it straightforwardly can generate lot of false positives [22]. User data comes normally in GET request so checking equality sign before semicolon can reduce the false positives to a great extent.

Union keyword is also source of attack. Consider the following query

```
SELECT uid, age, company FROM tblusers WHERE City = '" & strCity & "' AND Country = 'Sweden'"
```

Now an attacker can easily transform it to

```
SELECT uid, age, company FROM tblusers WHERE City = 'NoSuchCity' UNION ALL SELECT * FROM OtherTable tblcreditcard WHERE 1=1 AND Country = 'Sweden'
```

The following regular expression can block this kind of intrusion [22].

```
/((\%27)|(\'))union/ix
```

The above regular expressions are to be used with Snort IDS system . Snort is a free open source Intrusion detection system and is on its way to become a defecto standard for intrusion detection. [24] Snort rules can be easily created with the above expression.

However it is not necessary to use some kind of intrusion detection system to stop SQL Injection. The regular expressions (with small variations) can be used with in the programming environment to suffice the purpose.

4.2 Cross Site Scripting

Many websites allow user to post messages with HTML code and javascript in them. Some times websites gather data from user and pass it to another website to process it. If the data contains malicious code in it then that can be harmful for the second website and it's the fault of first website that hasn't correctly validated the user input on its part. [25, 26]

Most of the crosssite scripting attacks are result of HTML or javascript code hidden in a link. It can be plain text or in for of hex equivalents.

Regular expression for blocking simple attack is below [22]

```
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix
```

It checks for opening and closing angled brackets, alphanumeric strings with in the tad and forward slash used for closing HTML tag.

<img/> tag can download file easily hence can be used in CSS attack. Following Regular Expression can be used to stop this [22].

```
/((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[^\n]+((\%3E)|>)/I
```

Addition in this regular expression as compared to the previous is the letter and hex equivalents for <img> and any character following it except new-line. Rest is same.

If you want to block all attacks and do not care about false positives then use the following regular expression [22].

```
/((\%3C)|<)[^\n]+((\%3E)|>)/I
```

It simply looks for any tag any character following at other than new line and than closing tag. With this as a snort rule any thing that remotely resembles to CSS attack will be caught.

5. Conclusions

I couldn't find any approved standard for input validation, but according to a wide majority of web articles and books on computer security the standard method of validating user input is matching it against a regular expression written according to the users/application/input type. This can be done with in application code and also through an external tool like Snort IDS. Same kind of regular expressions can be used

with slight variations in different programming environments and on Snort or some other Intrusion detection system. There is another very strong advise given by almost all authors that one should check user supplied data against valid input pattern and reject all those don't match that, instead of looking for bad input patterns. There last thing is one should try to keep false positives as low as possible, the application should be secure against malicious users and it should be available to valid users, so one should take certain care when writing regular expression considering the type of application and level of security.

References

[1] S. Greg, "Why Hackers Escape", news.com 'May 14, 2002.
[2] http://www.securityfocus.com/bid/17821
[3] http://www.sei.cmu.edu/news-at-sei/features/2001/1q01/feature-1-1q01.htm (programmers are not trained to write secure programs)
[4] http://www.homeport.org/~adam/setuid.7.html
[5] http://www.canonical.org/~kragen/security-holes.html
[6] http://www.cert.org/advisories/CA-1995-14.html (environment variables that were transferable on telnet)
[7] http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/input.html
[8] http://www.govexec.com/dailyfed/0401/041801h1.htm(forbidden cookies)
[9] RFC 3629 UTF-8 Standard
[10] http://www.cl.cam.ac.uk/~mgk25/unicode.html
[11] http://www.cert.org/archive/pdf/cross_site_scripting.pdf
[12] http://www.cert.org/tech_tips/malicious_code_mitigation.html
[13] http://www-128.ibm.com/developerworks/library/l-sp2.html
[14] http://www.siam.org/siamnews/general/ariane.htm
[15] http://www.cert.org/homeusers/buffer_overflow.html
[16] Mat Bishop, Introduction to Computer Security.
[17] http://www-128.ibm.com/developerworks/linux/library/l-sp1.html(David's First Article)
[18] http://www.cert.org/advisories/CA-2003-18.html(Integer Overflow)
[19] http://www.redhat.com/archives/redhat-watch-list/2000-September/msg00004.html(locale vulnerability)
[20] http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002.pdf (Matt Bishop's slides, integer Over Flow Problem in SendMail)
[21] SQL Tutorial, http://www.w3schools.com/sql/default.asp
[22] Detection of SQL Injection and Cross Site Scripting Attack, http://www.securityfocus.com/infocus/1768
[23] Writing Secure Code, http://www.microsoft.com/mspress/books/5957.asp
[24] Snort Intrusion Detection System, http://snort.org/dl/
[25] Cross Site scripting, http://www.4guysfromrolla.com/webtech/112702-1.2.shtml
[26] Cross Site Scripting FAQ, http://www.cgisecurity.com/articles/xss-faq.shtml
[27] http://www.securityfocus.com/bid/17812
[28] http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf
[29] http://www.webappsec.org/projects/whid/list_class_sql_injection.shtml
[30] http://www.fcw.com/article92132-01-27-06-Web