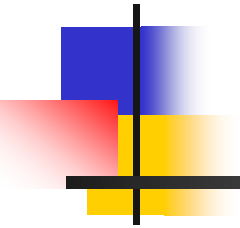
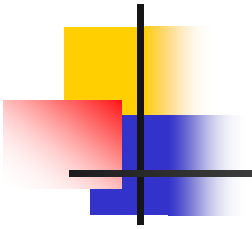
Input Validation Testing



Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu



Introduction

- The most common web application security weakness is the failure to properly validate input coming from the client or from the environment before using it.

- This weakness leads to almost all of the major vulnerabilities in web applications
  - cross site scripting
  - SQL injection
  - interpreter injection
  - file system attacks
  - buffer overflows

Introduction

- Data from an external entity or client should never be trusted, since it can be arbitrarily tampered with by an attacker.
- "All Input is Evil", says Michael Howard in his famous book "Writing Secure Code".
  - That is rule number one.
- Unfortunately, complex applications often have a large number of entry points, which makes it difficult for a developer to enforce this rule.
- Data Validation Testing
  - This is the task of testing all the possible forms of input to understand if the application sufficiently validates input data before using it.

Testing for SQL Injection: Summary

- An SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application.
- A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system.
- SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

Testing for SQL Injection: Summary

- In general, the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data.
- Example:

```
select title, text from news where id=$id
```

  - In the example above the variable $id contains user-supplied data, while the remainder is the SQL static part supplied by the programmer
    - making the SQL statement dynamic

Testing for SQL Injection: Summary

- Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user's choice.

- The example below illustrates the user-supplied data "10 or 1=1", changing the logic of the SQL statement, modifying the WHERE clause adding a condition "or 1=1".

Testing for SQL Injection: Summary

- SQL Injection attacks can be divided into the following three classes:
  - Inband: data is extracted using the same channel that is used to inject the SQL code.
    - This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
  - Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
  - Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

Testing for SQL Injection: Summary

- A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query.
- If the application returns an error message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly.
- However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.

Testing for SQL Injection: Summary

- About the techniques to exploit SQL injection flaws there are five commons techniques.
- Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):
  - Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
  - Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
  - Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
  - Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).
  - Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It is useful when attacker doesn't have some kind of answer (result, output, or error) from the application.

Testing for SQL Injection: How to Test

- Detection Techniques
  - The first step in this test is to understand when the application interacts with a DB Server in order to access some data.
  - Typical examples of cases when an application needs to talk to a DB include:
    - Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
    - Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
    - E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

Testing for SQL Injection: How to Test

- The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error.

- The very first test usually consists of adding a single quote (') or a semicolon (;) to the field or parameter under test.
  - The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query.
  - The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error.

Testing for SQL Injection: How to Test

- The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed
quotation mark before the
character string ''.
/target/target.asp, line 113
```

Testing for SQL Injection: How to Test

- Also comment delimiters (-- or /* */, etc) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query.
- A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error
converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Testing for SQL Injection: How to Test

- Monitor all the responses from the web server and have a look at the HTML/javascript source code.
  - Sometimes the error is present inside them but for some reason (e.g. javascript error, HTML comments, etc) is not presented to the user.
- A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection attack.
- However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques.
- In any case, it is very important to test each field separately:
  - Only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - Consider the following SQL query:

```
SELECT * FROM Users WHERE Username='$username' AND
Password='$password'
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - A similar query is generally used from the web application in order to authenticate a user.
  - If the query returns a value, it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied.
  - The values of the input fields are generally obtained from the user through a web form.
  - Suppose we insert the following Username and Password values:

```
$username = 1' or '1' = '1        $password = 1' or '1' = '1
```

  - The query will be:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND
Password='1' OR '1' = '1'
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

```
http://www.example.com/index.php?username=1'%20or%20
'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1
```

  - After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1).
  - In this way the system has authenticated the user without knowing the username and password.

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - In some systems the first row of a user table would be an administrator user.
  - This may be the profile returned in some cases.
  - Another example of query is the following:

```
SELECT * FROM Users WHERE ((Username='$username') AND
(Password=MD5('$password')))
```

    - In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function.

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):

```
SELECT * FROM Users WHERE ((Username='$username') AND
(Password=MD5('$password')))
```

  - First of all, we resolve the problem of the parentheses.
    - That simply consists of adding a number of closing parentheses until we obtain a corrected query.
  - To resolve the second problem, we try to evade the second condition.
    - We add to our query a final symbol that means that a comment is beginning.
    - In this way, everything that follows such symbol is considered a comment.
    - Every DBMS has its own syntax for comments, however, a common symbol to the greater majority of the databases is /*.
    - In Oracle the symbol is "--".

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - This said, the values that we'll use as Username and Password are:

```
$username = 1' or '1' = '1'))/*
```

```
$password = foo
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - This may return a number of values.
  - Sometimes, the authentication code verifies that the number of returned records/results is exactly equal to 1.

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - In the previous examples, this situation would be difficult (in the database there is only one value per user).
  - In order to go around this problem, it is enough to insert a SQL command that imposes a condition that the number of the returned results must be one.
  - (One record returned) In order to reach this goal, we use the operator "LIMIT <num>", where <num> is the number of the results/records that we want to be returned.
  - With respect to the previous example, the value of the fields Username and Password will be modified as follows:

```
$username = 1' or '1' = '1')) LIMIT 1/*
```

```
$password = foo
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 1 (classical SQL Injection):
  - In this way, we create a request like the follow:

```
http://www.example.com/index.php?username=1'%20or%20
'1'%20=%20'1'))%20LIMIT%201/*&password=foo
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 2 (simple SELECT statement):
  - Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

  - Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

  - When the tester tries a valid value (e.g. 10 in this case), the application will return the description of a product.
  - A good way to test if the application is vulnerable in this scenario is play with logic, using the operators AND and OR.

Testing for SQL Injection: Standard SQL Injection Testing

- Example 2 (simple SELECT statement):
  - Consider the request:

```
http://www.example.com/product.php?id=10 AND 1=2
```

```
SELECT * FROM products WHERE id_product=10 AND 1=2
```

  - In this case, probably the application would return some message telling us there is no content available or a blank page.
  - Then the tester can send a true statement and check if there is a valid result:

```
http://www.example.com/product.php?id=10 AND 1=1
```

Testing for SQL Injection: Standard SQL Injection Testing

- Example 3 (Stacked queries):
  - Depending on the API which the web application is using and the DBMS (e.g. PHP + PostgreSQL, ASP+SQL SERVER) it may be possible to execute multiple queries in one call.
  - Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

  - A way to exploit the above scenario would be:

```
http://www.example.com/product.php?id=10; INSERT INTO
users (...)
```

Testing for SQL Injection: Standard SQL Injection Testing

- Fingerprinting the Database:
  - Even the SQL language is a standard, every DBMS has its peculiarity and differs from each other in many aspects like special commands, functions to retrieve data such as users names and databases, features, comments line etc.
  - When the testers move to a more advanced SQL injection exploitation they need to know what the back end database is.

Testing for SQL Injection: Standard SQL Injection Testing

- Fingerprinting the Database:
  - 1) The first way to find out what back end database is used is by observing the error returned by the application.
  - Follow are some examples:
    - MySql:

      You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '\'' at line 1

    - Oracle:

      ORA-00933: SQL command not properly ended

    - MS SQL Server

      Microsoft SQL Native Client error '80040e14'
      Unclosed quotation mark after the character string

Testing for SQL Injection: Standard SQL Injection Testing

- Fingerprinting the Database:
  - 2) If there is no error message or a custom error message, the tester can try to inject into string field using concatenation technique:

```
MySql: 'test' + 'ing'
SQL Server: 'test' 'ing'
Oracle: 'test'||'ing'
PostgreSQL: 'test'||'ing'
```