Building Secure Applications: SQLi Training Session

Marian Ventuneac
marian.ventuneac@gmail.com

Author

- Security Architect, PhD, MEng, CISM, CISA
- OWASP Ireland Limerick Chapter Leader

  https://www.owasp.org/index.php/Ireland-Limerick
- Member of OWASP and ISACA Ireland Chapters
- Senior Security Researcher

  Data Communication Security Laboratory, University of Limerick

  http://www.ventuneac.net

  http://secureappdev.blogspot.com

Objectives

- Impact of SQL Injection Attacks
- How an SQL Injection Attack works
- Testing for SQL Injection Vulnerabilities
- Why is my App vulnerable to SQL Injection Attacks?
- SQL Injection Prevention
  - Secure Code Review
  - Secure Coding Best Practices

Application Security Myths

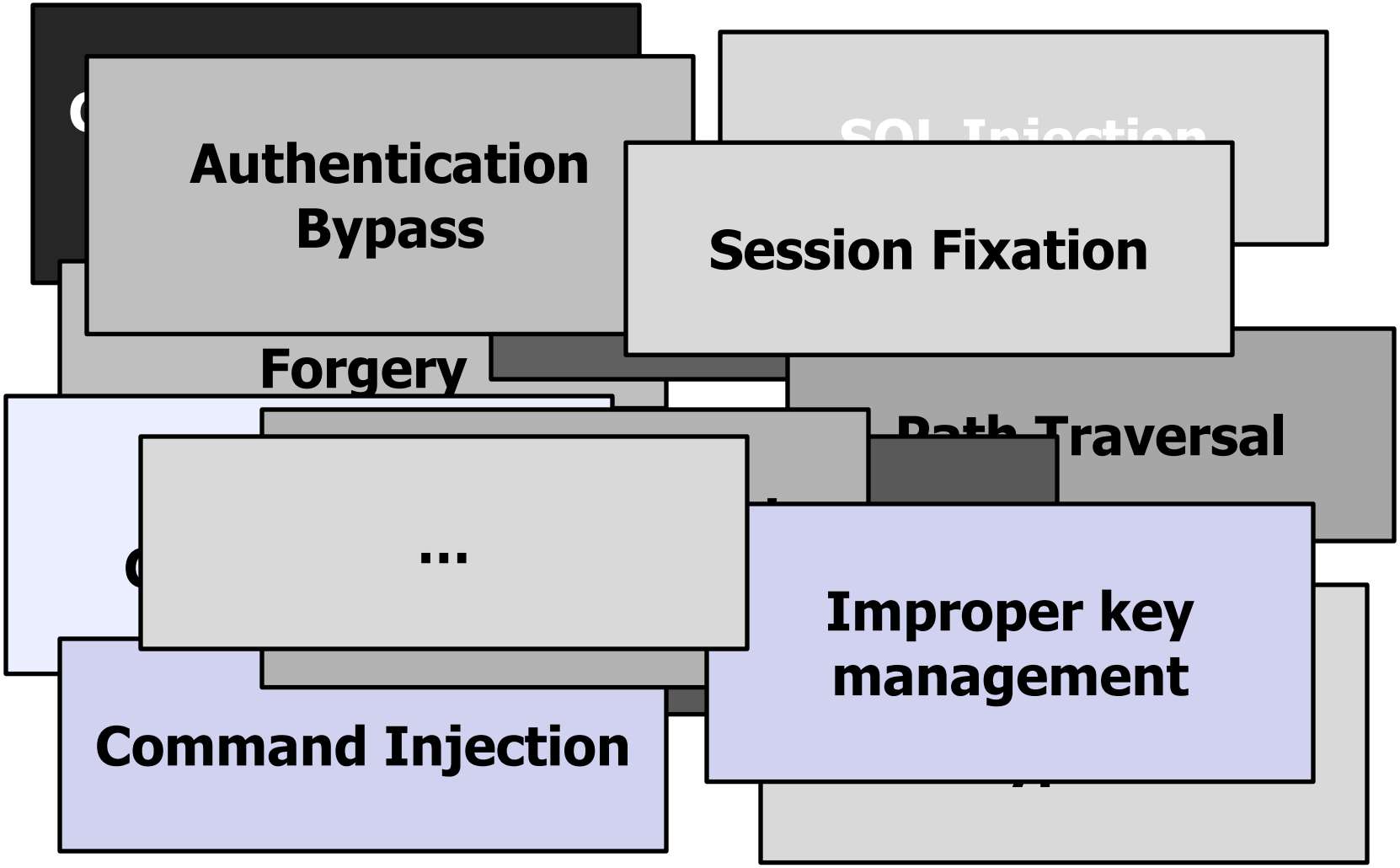
- Network firewalls do protect against Web application security attacks
- Our Web application is totally secure because we use HTTPS
- Strong approach to application security - we perform an annual application security review
- Bullet-proof application security - we use automated vulnerability scanners to identify all vulnerabilities

…

… and the Myth Busters

- Network firewalls don't protect against common application security attacks
- Usage of HTTPS allows encrypting the communication channel. However, exploitation of common application vulnerabilities works just fine over HTTPS
- Security vulnerabilities are discovered on a daily basis – an annual security review could be too little, too late
- An application vulnerability scanner is doing only what is designed for + false positives/negatives
  - There is no automated solution which guarantees the discovery of all known vulnerabilities…
  - For black-box testing, not all types of attacks can be effectively automated

Web Application Security Risks



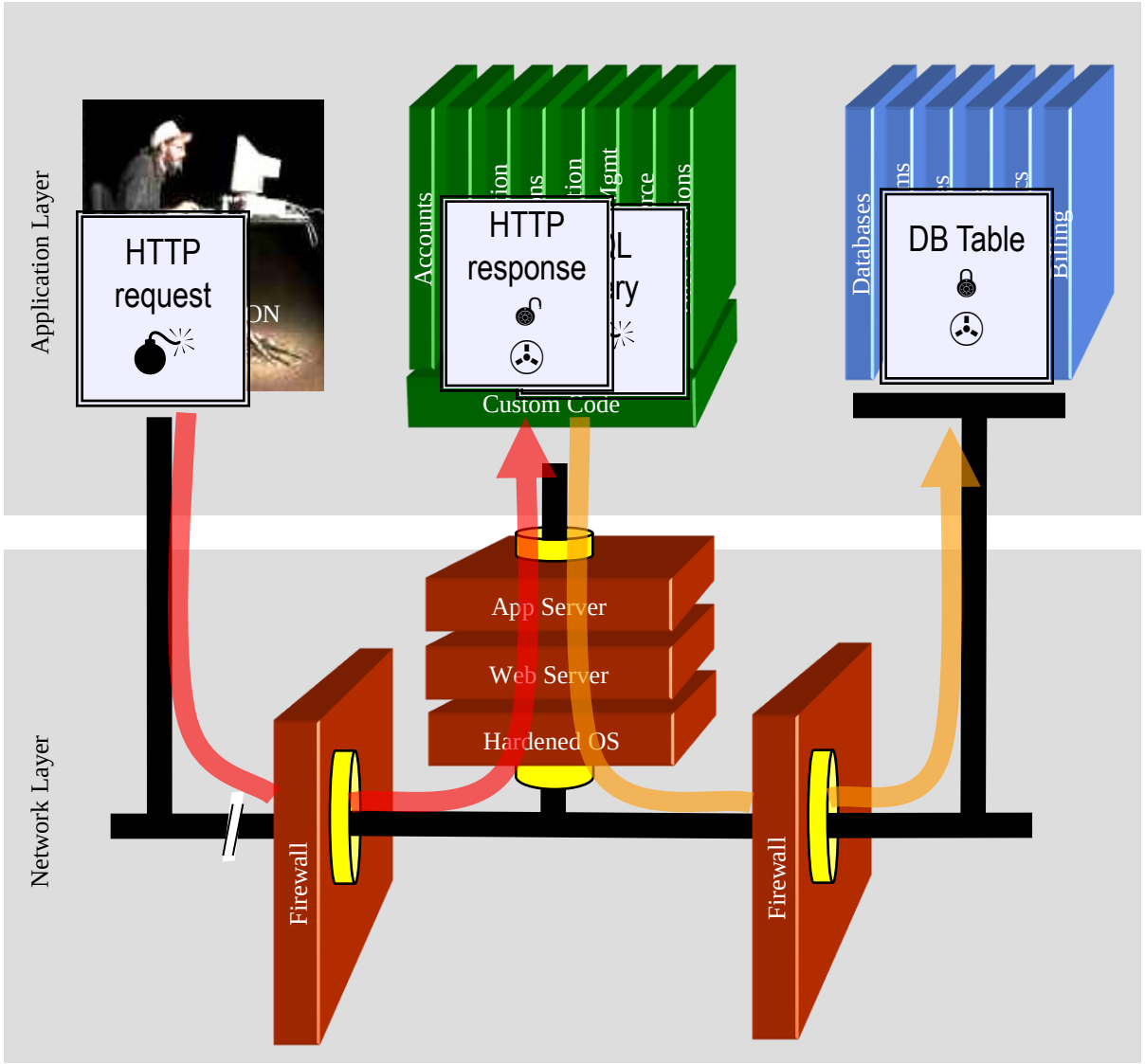
Classifying Security Risks

- OWASP Top 10 Security Risks
  https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- MITRE Common Weaknesses Enumeration (CWE)
  http://cwe.mitre.org/
- CWE/SANS Top 25 Software Errors
  http://www.sans.org/top25-software-errors/
- WASC Threat Classification
  http://projects.webappsec.org/w/page/13246978/Threat Classification
- …

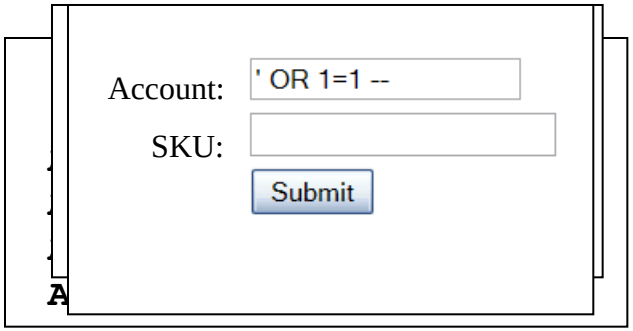
OWASP Top 10: A1 – Injection

- Inject unintended commands in the data sent to an interpreter (SQL, OS Shell, LDAP, XPath, Hibernate, etc…)
- SQL Injection - one of the most common injection attacks
- Impact of SQL Injection attacks
  - The database can be fully read and potentially modified
  - Database schema access
  - Account access
  - OS level access
  - Severe compromise of data confidentiality, integrity and availability

SQL Injection Explained



From OWASP Top 10



1. Application presents a form to the attacker
2. Attacker sends an attack in the form data
3. Application forwards attack to the database in a SQL query
4. Database runs query containing attack and sends encrypted results back to application
5. Application decrypts data as normal and sends results to the user

Testing for SQL Injection Vulnerabilities

- Example: search for valid string *mydata*
  - Test case: *mydata*’**.** If the application returns an error, the applications could be vulnerable to SQL Injection attacks

> [Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark
> after the character string ‘mydata".

> Oracle Automation error '800a01b8'
> SQL execution error, ORA-29902: error in executing ODCIIndexStart() routine ORA-20000:
> Oracle Text error: DRG-50901: text query parser syntax error on line 1, column 34
> /search/search.jsp, line 822

> 500: Internal Server Error

> Custom error page

  - Test case: *mydata*’’**.** If no errors are returned now, the applications is more likely vulnerable to SQL Injection attacks
  - Demo 1

Testing for SQL Injection Vulnerabilities (cont)

- Example 2 – numeric SQL injection: query for records matching a numeric value *myvalue*
  - Test case for *myvalue or 1=1*. If the application returns more results than when querying with *myvalue*, then your application is most likely vulnerable to SQL Injection attacks
  - Demo 2
- Example 3 – blind SQL injection: query for numeric *myvalue* to determine if valid or not
  - Test case for *myvalue and (my_condition is FALSE)*. This will label a valid myvalue as invalid (since second condition is false)
  - Test case for *myvalue and (my_condition is TRUE)*. This will label a valid myvalue as valid (since second condition is true)
  - Demo 3

Testing for SQL Injection – Resources & Tools

- Testing for SQL Injection
  https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OWASP-DV-005)
- Blind SQL Injection
  https://www.owasp.org/index.php/Blind_SQL_Injection
- SQL Injection Cheat Sheet
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Advanced SQL Injection in SQL Server Applications
  http://www.thomascookegypt.com/holidays/pdfpkgs/931.pdf
- Black-box testing:
  - FOOS tools: Burp Suite (manual testing), OWASP ZAP, Absinthe, SQLMap, SQLNinja, SQlBrute, etc
  - Commercial tools and services

Why is My Application Vulnerable?

- Secure Code Review
  - Identify what user provide input is used to query the backend
  - Assess how SQL queries are built and executed
- Common programming errors leading to SQL Injection vulnerabilities
  - Usage of dynamic queries where user provided data is unsafely concatenated to build the query
  - Lack of proper encoding for user provided data
  - Improper database user privileges
  - Lack of input data validation

Why is My Application Vulnerable? (cont)

- Unsafe usage of Dynamic SQL Queries
  - Flawed Java code examples

```
String userid = request.getParameter(“username");
String query = "SELECT * FROM users WHERE username = ‘" + userid + "’";
                                                                    Presentation Layer
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                                    Data Access Layer
PreparedStatement pstmt = connection.prepareStatement( query );
try {
      ResultSet users = pstmt.execute( );
    …
}
```

```
                                                                    Data Access Layer
String userid = request.getParameter(“username");
String query = "SELECT * FROM users WHERE username =  ‘" + userid + "’ ";
Statement stmt = connection.createStatement();
try {
      ResultSet users = stmt.executeQuery( query );
    …
}
```

Why is My Application Vulnerable? (cont)

- Unsafe usage of Dynamic SQL Queries
  - Flawed C# code example

```
…
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName +
        "' AND itemname = '" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

  - Flawed PHP code example

```
$username = trim($_POST['username']);
$query = "INSERT into
tests(id,status,username,urls) VALUES($nextId,'Creating profile...',$username,'')";
$result = $db->query($query);
```

Why is My Application Vulnerable? (cont)

- Unsafe usage of Dynamic SQL Queries
  - Flawed HQL code example (Hibernate)

```
…
Query unsafeHQLQuery = session.createQuery("from Inventory where
    productID='"+userSuppliedParameter+"'");
...
```

  - Flawed SQL Server stored procedure code

```
Create procedure get_report @columnamelist varchar(7900) As
Declare
@sqlstring varchar(8000)
Set @sqlstring = ‘ Select ‘ + @columnamelist + ‘ from ReportTable‘
exec(@sqlstring)
Go
```

Preventing SQL Injection Flaws (cont)

- Use an interface that supports bind variables (e.g., prepared statements, or stored procedures)
  - Avoid using Dynamic SQL queries
  - When using Dynamic SQL, make sure that you bind all variables to allow the interpreter to distinguish between code and data
- Escape all user input before passing it to the interpreter
- Perform ‘white list’ input validation on all user supplied input
- Minimize database privileges to reduce the impact of a flaw

Preventing SQL Injection Flaws (cont)

- Use Prepared Statements
  - Safe Java code

```
String username = request.getParameter(“userName");
// This needs to be validated
String query = "SELECT account FROM users WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, username);
ResultSet results = pstmt.executeQuery( );
```

  - Safe C# .NET code

```
String query = "SELECT account FROM users WHERE user_name = ?";
try {OleDbCommand command = new OleDbCommand(query, connection);
command.Parameters.Add(new OleDbParameter("customerName", CustomerName
    Name.Text));
OleDbDataReader reader = command.ExecuteReader(); // …
} catch (OleDbException se) { // error handling }
```

  - Safe Hibernate code using named parameters

```
Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Preventing SQL Injection Flaws (cont)

- Query parametrisation PHP example

```
$stmt = $dbh->prepare(“SELECT * FROM users WHERE username=:name");
$stmt->bindParam(':name', $username);
```

- Safe Hibernate code using named parameters

```
Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Preventing SQL Injection Flaws (cont)

- Use Stored Procedures
  - Safe Java Stored Procedure code

```
String username = request.getParameter(“userName"); // This needs to be validated
try {
      CallableStatement cs = connection.prepareCall("{call sp_getAccount (?)}");
      cs.setString(1, username);
      ResultSet results = cs.executeQuery();
} catch (SQLException se) { … }
```

  - Safe VB .NET Stored Procedure code

```
Try
      Dim command As SqlCommand = new SqlCommand("sp_getAccount", connection)
      command.CommandType = CommandType.StoredProcedure
      command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
      Dim reader As SqlDataReader = command.ExecuteReader()
Catch se As SqlException
…
 End Try
```

Where no stored procedure is used, Parameters Collection can be safely used with Dynamic SQL

Preventing SQL Injection Flaws (cont)

- Bind Variables when using Stored Procedures
  - Safe PL/SQL Stored Procedure code

```
create or replace procedure myPLSQLProcedure(account_no in number) as
begin
      execute immediate
      'update myAccounts set account_no = :n'
      using n;
commit;
end;
```

  - Safe TSQL Stored Procedure code using Parameters

```
Create Procedure spSelectAccount (@Account INTEGER)
      AS
      SELECT * FROM Users
      WHERE account_no=@Account
Go
```

Where no stored procedure is used, Parameters Collection can be safely used with Dynamic SQL

Preventing SQL Injection Flaws (cont)

- Escape all user provided input before using it to query the backend
  - Useful when refactoring to code to use parameterized queries is not viable
  - Not as strong as using parameterized queries
- Use database specific character escaping schemes
- OWASP Enterprise Security API (ESAPI)
  - Database encoders for Oracle and MySQL, while work is being done to add SQL Server and PostgreSQL encoders
- Validate all user provided input using a ‘while list’ approach (server-side is recommended)

Preventing SQL Injection Flaws - Resources

- SQL Injection Prevention Cheat Sheet
  http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- SQL Injection
  http://msdn.microsoft.com/en-us/library/ms161953%28v=SQL.105%29.aspx
- Reviewing Code for SQL Injection
  https://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection
- OWASP ESAPI
  https://www.owasp.org/index.php/ESAPI
- OWASP WebGoat J2EE Project
  https://www.owasp.org/index.php/WebGoat
- OWASP WebGoat .NET Project
  https://www.owasp.org/index.php/Category:OWASP_WebGoat.NET

Q&A