Teaching with Security in Mind

Orlando Karam
Department of Computer Science and Software Engineering
School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, GA 30060
okaram@spsu.edu

Svetlana Peltsverger
Information Technology Department
School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, GA 30060
speltsve@spsu.edu

ABSTRACT

Security topics have been taught for some time at universities. The most common approach has been to teach a required topic, and then introduce a security module later in the course. We are promoting the notion of teaching security at the same time as main course’s material. This helps students to adopt to the idea of writing secure code at an early stage and encourages them to focus on the security issues before beginning coding. While this method has clear advantages, it is not easy to implement in practice. This is partly a result of the faculty's lack of awareness about security issues, and the perception of security as an advanced topic. We see security as an extension of the basic concept of input validation, and so a very basic topic. We also propose teaching secure practices as the default model presented to the students, facilitating the adoption of those practices.

Categories and Subject Descriptors


General Terms

SQL injection attacks, web development

Keywords

Sql injection, computer security

1. INTRODUCTION

Almost a decade ago cyber security became a broadly recognized priority for both government and private industry. Information assurance became a multi billion industry, but universities are still not producing the graduates that will satisfy the requirements of industry in this area. This paper explores the reasons why there is still lack of security awareness among most undergraduate computing students. The authors will explore the issue of introducing security early in the courses, and why this is important. Security topics are incorporated in existing courses, but not early enough. After students learn how to write code, they have to learn again how to write the same code with security in mind. This situation is similar to what was happening to teaching programming using Object Oriented approach.

According to the Application Security Trends Report for the second quarter of 2008[1] by Cenzic, there were 1,200 unique vulnerabilities, with 73 percent falling under the Web technology vulnerabilities category, with 34 percent of the total Web vulnerabilities being SQL injection vulnerabilities. The report states that the percentage of Web vulnerabilities has risen, and this is an alarming statistics. According to the report, almost 25% of all vulnerabilities are SQL injections. They are also the most commonly exploited vulnerabilities. That is why SQL injection is the primary focus of this paper.

2. BACKGROUND

One of the most common problems programmers create for themselves is accepting unvalidated input, which will cause any number of problems, including SQL injection.

In many web applications, input is received from a form, and then a SQL query string is created which includes that input; this string is then sent to a database management system (DBMS) for execution. In a SQL injection attack, the input is crafted so that the resulting SQL statement changes its meaning. In SQL, character constants are surrounded by apostrophes ('), semicolons (;) usually separate statements, and (--) start of comment, so the mischievous inputs will usually include at least one of those characters. If we want to actually include one of those characters in a string, we can precede them with a backslash (\), a process called escaping.

While it is easy to escape those special characters, input validation cannot easily protect against injection strings that use alternate character encodings, like URL encoding or Unicode. It is even harder to try to detect other patterns in mischievous input strings; the well known '1' = '1' that is commonly used to force the WHERE clause to evaluate to true can be transformed to 'd' = 'd' or even 'd' = (select substring(user,1,1)) that will be evaluated to true for the MS SQL Server dbo user.

Commercial DBMS vendors developed a mechanism, called PREPARED statements, that can be used to protect their products from SQL injection vulnerabilities. "However, retrofitting an application to make use of PREPARED statements requires manual effort in specifying the intended query at every query point, and the effort required is proportional to the complexity of the web application." [2].

Parameterized SQL is not a new concept. There are numerous articles available about parameterized batch writing of SQL statements, or PREPARED statements. It is well known as one
way to increase performance when executing similar queries, as the database does not have to parse each statement on the batch, but executes it as a single statement and a set of bind parameters. Prepared statements also increase security by separating SQL logic from the data. Keeping the logic (structure) of a SQL query separate from data helps to prevent SQL injection attacks. SQL injection attacks work by modifying the logic of the query, to make WHERE clause to evaluate to true (with attacker injected code in bold).

For example, imagine that we have a web application, that receives two fields from a form, and generates a SQL query string from them; using PHP syntax, and assuming that the incoming information is stored in the variables $userId and $userPassword, we would write:

```
$query = "SELECT * FROM users WHERE userId= $userId
AND userPassword = '$userPassword";
```

Notice that in php, variable references are preceded with $, and variables are substituted inside string constants.

When passed expected data, say $userId being 123 and $userPassword being abc, the query will behave as expected; $query would be: SELECT * FROM users WHERE userId=123 AND userPassword= 'abc'.

However, a malicious user could pass, as the value of $userPassword a string containing a single quote character, which would terminate the string constant in the first query; it could then add other conditions, and then use the closing quote. For example, if $userPassword is set to ' OR '1'='1 then the query would become:

```
SELECT * FROM users WHERE userId = 123 AND userPassword = 'abc123 OR '1' = '1';
```

A malicious string could also change number of conditions (notice the use of -- to comment the rest of the string)

```
SELECT * FROM users WHERE userId = 123; -- AND userPassword = 'abc';
```

or use a semicolon to terminate the query and then execute another query, which could even modify the database.

```
SELECT * FROM users WHERE userId = 4; DROP TABLE users; -- AND userPassword = 'abc';
```

Below we will compare parameterized SQL queries with non-parameterized queries and show that most of the time they are easier to understand.

Java Database Connectivity (JDBC) PARAMETERIZED

```
// query logic
prepSt = con.prepareStatement("SELECT lname FROM users WHERE userId = ? AND userPassword = ?");
// query data
users.setInt(1, id);
users.setInt(2, pass);
// query execution
users.executeUpdate();
```

Java Database Connectivity (JDBC) NON PARAMETERIZED

```
// query
Statement users = con.createStatement();
ResultSet result = users.executeQuery("SELECT lname FROM
users WHERE userId = " + id + " AND
userPassword = " + pass + ");
```

MySQLi (MySQL Improved) extension for PHP

```
// query logic
$statement = $db_connection-> prepare("SELECT lname FROM users WHERE userId = ? AND
userPassword = ?");
// query data where "is" - an integer
// followed by a string
$statement->bind_param("is", $id, $pass);
// query execution
$statement->execute();
```

**MySQL/PHP NON PARAMETERIZED**

```
mysql_select_db("db");
$query = "SELECT lname FROM users WHERE
userId = ": $id, " AND
userPassword = " : $pass . " ";
$result = mysql_query($query);
```

SQL Server and .NET PARAMETERIZED

```
// query logic
rdSetCmd.CommandText = "SELECT lname FROM
users WHERE userId = ? AND
userPassword = ?";
rdSetCmd.Prepared = true;
// add int
rdSetCmd.Parameters.Append(rdSetCmd.CreateParameter("id", adInteger, adParamInput, 5,
1234));
// add string
rdSetCmd.Parameters.Append(rdSetCmd.CreateParameter("pass", adVarChar, adParamInput, 12,
"abc123");
set rs = rdSetCmd.Execute();
```

SQL Server and .NET NON PARAMETERIZED

```
rs.Open "SELECT lname FROM
users WHERE userId = " + id + ";
```

While parameterized code is quite different from non-parameterized, there should be no difference for a student who is just learning how to retrieve data from a database table. It might be even easier to explain how to bind parameters, than to explain why they need "" (apostrophe in quotes).

Google search was used to find courses where students start learning how to write database driven applications. Search string used: "web development" database syllabus site:edu. From the first 20 courses with the available on-line course materials, only two used PREPARED statement, majority explained how to escape one or several database reserved characters (apostrophe, semicolon, etc) to avoid errors, not to protect data. Almost all courses listed W3Schools as a reference.

W3Schools is the largest web developers' site on the Internet. They display 80 million pages every month with more than 11 million visitors per month. According to Google search (w3schools uses Google as search engine) they have only one page that mentions SQL injection attack http://www.w3schools.com/php/func_http_header.asp. A code injection is not mentioned either.

The following code was used to perform code injection:

```
<script type="text/javascript">
```
To get a quick estimate of the prevalence of security problems in actual code, we used Google's code search [5], and searched for the strings pg_query (for the function used to execute dynamic SQL queries against a PostgreSQL database), and pg_query_params (used to execute prepared statements). The search for pg_query returned 6,000 hits, while a search for pg_query_params returns only 60 queries; while further analysis would be needed to validate the relevance of those hits, a ratio of 100 to 1 indicates that many more applications are using pg_query and are potentially vulnerable to SQL injection attacks.

Most universities, including SPSU have courses that teach security [3], but most of those courses are upper level electives, rather than entry level courses. Development of special tools that teach security [4] is another common approach in teaching Information Assurance. Most of the time security is included in existing courses as one of the last modules.

PREPARED statements separates SQL logic from the supplied data that prevents from SQL injection attacks, but it does not support all query types for example Data Definition Language (depends on DBMs). Moreover for some queries implementation is tricky. For example when variable number of columns is used in SQL IN clause [6]

```java
PreparedStatement stmt = conn.prepareStatement("SELECT id, name FROM users WHERE id IN (?, ?, ?)");
```

Another disadvantage of PREPARED statement that it takes additional time to prepare the query. This time will be compensated only if the query is executed many times (some authors say > 50). We firmly believe that correctness trumps efficiency in most cases, and so we should use PREPARED statements in most cases.

3. ADVANTAGES OF "SECURITY FIRST" APPROACH

It is reasonable to assume that students by default will keep using the patterns that they were taught at school, although of course they will adopt new patterns and revise their patterns as they become more experienced. Teaching them secure patterns as the default patterns of software should help ensure they produce more secure software in their professional career. For many applications, the secure patterns are not more difficult to understand or apply than the insecure ones.

Teaching some of the basics of computer security, even without getting deep into details, should help them produce better and more secure software in their professional careers and enhance the security of future software in general.

Moreover, many of the secure programming issues are related to software correctness in general; if you’re not validating your inputs, you’re not just letting the door open to security problems, but also to many other kinds of malfunctions. Teaching our students to validate their input should make them produce better software.

4. DISADVANTAGES OF "SECURITY FIRST" APPROACH

The main disadvantage of introducing security in basic classes is that it is an additional topic, and so it can take time and attention away from other important topics; however, we’re advocating not just teaching more information security, but teaching patterns of software development that are more secure by default, even if the students are not any more aware of security per se.

Another possible disadvantage could be that some secure patterns are harder to understand or slower when executing; we believe, however, that most patterns, and in particular those dealing with prepared SQL statements are as easy to understand as the vulnerable patterns, and will result in only slight performance degradation.

5. CONCLUSION

Most students will keep programming the way we teach them. If we teach them secure patterns as the default, they will produce more secure software without any additional effort on their part. These secure patterns are not more complicated that the ones we are currently using.

6. REFERENCES

[1] Cenzic: Top Vulnerabilities
http://www.cenzic.com/cia_research/top_vulnerabilities.php


