INF226 – Software Security

Håkon Robbestad Gylterud

2019–08–26

Plan for the lecture

- Stack smashing example.
- SQL injections
  - What is the problem?
  - Three solution strategies:
    - Blacklist (bad)
    - Quoting/escaping (difficult)
    - Prepared statements (easy and correct)

Stack smashing

Memory layout of a C program

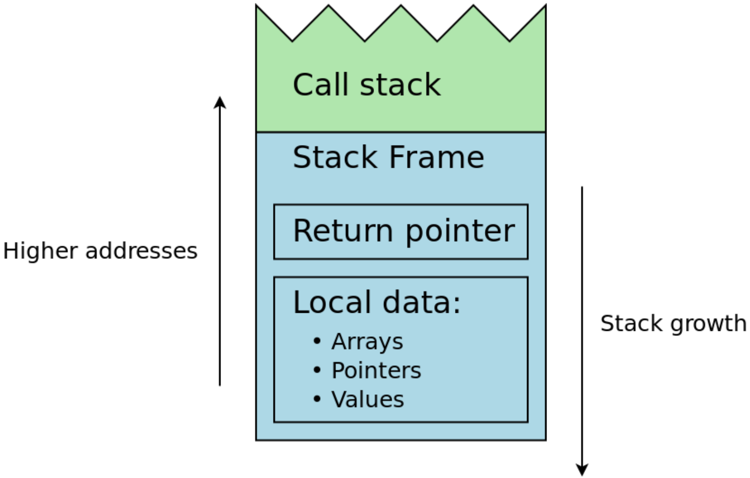
```
/------------------\  lower
|                  |  memory
|       Text       |  addresses
|                  |
|------------------|
|   (Initialised)  |
|        Data      |
|  (Uninitialised) |
|------------------|
|                  |
|       Stack      |  higher
|                  |  memory
\------------------/  addresses
```

The .text section

```
0000000004005b7 <func>:
  4005b7:       55                      push   %rbp
  4005b8:       48 89 e5                mov    %rsp,%rbp
  4005bb:       48 83 ec 10             sub    $0x10,%rsp
  4005bf:       48 8b 15 7a 0a 20 00    mov    0x200a7a(%rip),%r
  4005c6:       48 8d 45 f8             lea    -0x8(%rbp),%rax
  4005ca:       be 00 04 00 00          mov    $0x400,%esi
  4005cf: (···)

0000000000400601 <main>:
  400601:       55                      push   %rbp
  400602:       48 89 e5                mov    %rsp,%rbp
  400605:       b8 00 00 00 00          mov    $0x0,%eax
  40060a:       e8 a8 ff ff ff          callq  4005b7 <func>
  4005cf: (···)
```

The call stack



Higher addresses

Call stack

Stack Frame

Return pointer

Local data:

- Arrays
- Pointers
- Values

Stack growth

Figure 1: The call stack

Return oriented programming example

```
#include <stdio.h>

void func () {
    char buffer[8];
    fgets(buffer, 1024 , stdin);
    printf("You entered: %s \n", buffer);
}

void never() {
    printf("This function is never called.\n");
}

int main() {
    func();
    return 0;
}
```

SQL injection

SQL

- Structured Query Language (SQL) is the dominating language for relational databases.
- It is a **domain specific language**.
- Queries are contructed using other languages.
- Queries are constructed from **user input**.

SQL example

```
SELECT * FROM items WHERE owner='paul' AND
itemname='crysknife'
```

Result:

id	*owner*	*itemname*	*location*
32	paul	crysknife	pocket

Quoting

Problem: Expressions in a language consist of strings. How to represent strings?

Quoting

Problem: Expressions in a language consist of strings. How to represent strings?

First approximation: `'This is a string'`

But what about strings containing the character `'` *itself*?

Quoting

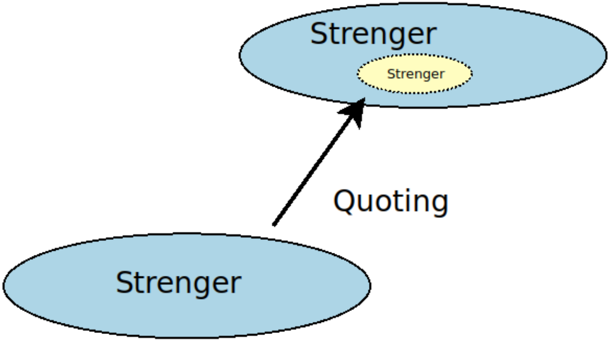


Figure 2: The general problem of quoting.

SQL injection

```
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);
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '" + ItemName.Text + "'";
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
   + userName + "' AND itemname = '" + ItemName.Text + "'";
```

What happens if ItemName.Text comes from user input, and the user inputs the following string?

```
name' OR 'a'='a
```

SQL injection

What if the input was the following?

```
name'; DELETE FROM items; --
```

Preventing SQL injections

First attempt: This is an input sanitation problem. We must blacklist some characters (such as `'` and `--`).

Preventing SQL injections

```
$id = $_COOKIE["mid"];
if (preg_match("/\'/") {
  fail();
} else {
  mysql_query("SELECT MessageID, Subject
               FROM messages WHERE MessageID = $id");
  ...
}
```

Preventing SQL injections

- Hyphens, single quotes and semi-colons are common in natural language.
- Blacklists most often have loop holes.
- Makes for fun CTF challenges, but not great security.

Preventing SQL injections

Second attempt: We must turn all single quotes into double `'`, quotes `''`, which escapes them.

(. . . or into `\'`, which is another way to escape it.)

Escaping data for SQL queries

It is **not enough** to escape single quotes!

- A single quote in a string is represented by `''`
- Thus we can try to double all single quotes in data.
- But this can be worked around by attacker:
  - `\'` becomes `\''` (an escaped quote followed by a quote)

Notoriously difficult to get the escaping right!

SQL injection

```
String query
  = "SELECT * FROM Users WHERE email='"
   + authenticatedUser.getEmail() + "';"
try {
   Statement statement = con.createStatement();
   ResultSet result = statement.executeQuery(query);
   while (result.next()) {
      // ···
   }
} catch (SQLException e) {// ···
```

Then comes a user with e-email address: `eve'or''!='@foo.com`

Prepared statements

A better way to secure against SQL injection:

- **A prepared statement** is a statement with placeholders (?) where the user data will go later.
- Is sent to the SQL server in advance.

Example: In JDBC

```
String query
  = "SELECT * FROM Users WHERE email=? ;"
try {
   PreparedStatement prepared = con.prepareStatement(query);
   prepared.setString(authenticatedUser.getEmail());
   ResultSet result = statement.executeQuery(query);
   while (result.next()) {
      // ···
   }
} catch (SQLException e) {// ···
```

Prepared statements

- Prevents SQL injections.
- Allows type-checking of arguments.
- Could give better performance if a statement is executed many times.

Prepared statements

```
String query = "INSERT INTO order (userid,itemid,address) "
             + "VALUES(" + currentUser + "," + itemId + ","
                         + deliveryAddress ");";

PreparedStatement stmt = connection.prepareStatement(query);
stmt.execute();
```

Haskell sqlite-simple

- There are several DB libraries for Haskell (ex: HDBC).
- Highlighting `sqlite-simple` because it gives type safe protection from most SQL injection pitfals.

Haskell sqlite-simple

```
{-# LANGUAGE OverloadedStrings #-}

(...)
  do
    result <- query conn
                    "SELECT * FROM user WHERE name= ? AND age > ?"
                    ("Boris" :: String, 37 :: Int)

(...)
```

Haskell sqlite-simple

```
{-# LANGUAGE OverloadedStrings #-}

(· · ·)
  do
    result <- query conn
                    "SELECT * FROM user WHERE name= ? AND age > ?"
                    ("Boris" :: String, 37 :: Int)

(· · ·)
```

```
query :: (ToRow q, FromRow r) => Connection -> Query -> q -> IO [r]
```

Haskell sqlite-simple

This would be **ill-typed** (i.e. not compile):

```
{-# LANGUAGE OverloadedStrings #-}

(...)
  do
    result <- query conn
                    "SELECT * FROM user WHERE name="
                    ++ name ++ " AND age > " ++ age
(...)
```

Because `"SELECT * FROM user WHERE name="` has type `Query` and cannot be concatenated with strings.

Testing

The places in the code which cause SQL injections have a clear signature:

- String concatenation on a string which ends up in a query.

Static tools (such as SonarQube) will detect this.

Conclusion

The underlying problem with SQL:

- Confusion between code and data.
- Strings are used to represent both:
  - Data which goes into the database.
  - Queries and code to be executed on the database.
- Languages with type systems can do better!

When designing a program always ask: Is `String` the correct representation of this data?

Other injection attacks

SQL is not the only plase this confusion happens:

- Buffer overflows
- OS command injection
- `eval` injection in scripting languages (ex: Python)
- Cross-site scripting

OS command injection

PHP example:

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

OS command injection

PHP example:

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Now Maleroy enters `;rm -rf /` in the user field:

```
$command = 'ls -l /home/' . $userName;
```

References

- OWASP Top 10: A1
- CWE-89
- JDBC Prepared Statements