CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 10th, 2011

Web security

Subtitle

On the web, you've got a **client** and then you've got a **server**, and there's a **web app** running on top of a **web server**. Nowadays, there can also be the **JavaScript** code running on the **client's browser**.

The client-side JavaScript and the server-side web app work together to build a single application, part of which runs on the client.

This introduces a **whole new twist** on what you can trust since part of your web application runs on another person's computer. Therefore, the attacker can subvert your security on his client side.

Security goals

There are **three goals in web security**:

1. Protect web apps from malicious clients
2. Protect users from malicious web apps
3. Enable secure mash-up of web applications

Protecting web applications from malicious clients

Since the browser is not inside of our security domain we cannot trust the browser to enforce security checks.

Client-side checking of security constraints

Imagine you have a web app and it connects which has an input box where a user can type in a quantity (like the numbers of items they want to order).

One of your security goals is that $quantity \leq stock\ quantity$, which is displayed by the browser. You might have some JavaScript in this page which checks whether the number is in range.

The user can disable JavaScript and send any $quantity$ value or he can download the page, modify the JavaScript and still send an invalid request with a bad $quantity$.

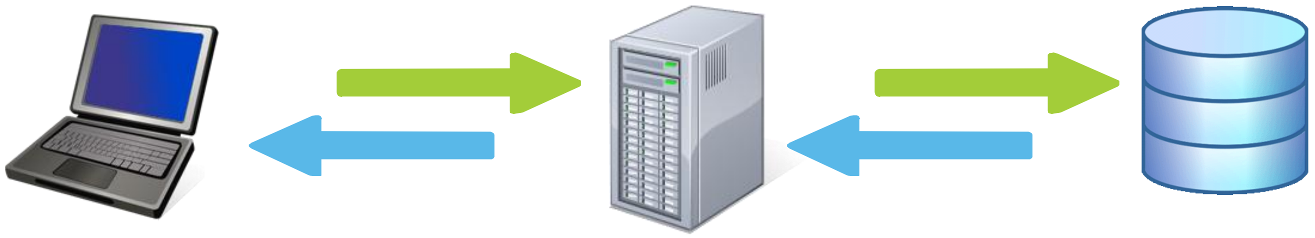
This can get worse when the input is something more sensitive.

SQL injection attacks

SQL injection attacks don't really have anything to do with the division between the trusted and the untrusted. It just has to do with the fact that users can be **evil**.

In most web applications, you have the browser, who talks to a web server, who talks to a database, as illustrated below:

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 10th, 2011



On the **webserver,** there is some PHP or Java code which translates requests from the browser into SQL requests for the database. This code often looks like the following:

```
int loginuser(String name, String password)
{
    String query = "SELECT * FROM users WHERE username=\'" + name +
        "\' AND password=\'" + password + "\';";

    sql.execute(query);

    if(sql.resultCount >= 1)
        start_session(name);
}
```

Example of well generated query: `SELECT * FROM users WHERE username='ROB' and PASSWORD='open sesame'`

SQL injection attack

What if the attacker picked a password like this one: `' OR 1==1`?

The **generated query** will be: `SELECT * FROM users WHERE username='ROB' and PASSWORD='' OR 1==1'`

- The SQL parser will error because of the last unmatched quote

A better choice for the evil password would be: `' OR 1==1;--`

- The two dashes at the end will comment out the rest of the query, and will create a valid query

The **generated evil query** will be: `SELECT * FROM users WHERE username='ROB' and PASSWORD='' OR 1 ==1;--'`

- This query will always return one result since `PASSWORD='' OR 1 ==1` will always evaluate to true and the `loginuser` function will be tricked into logging Rob in without his password

Another thing you can do is craft an input that will **force the username to match anything** and set the password to some common password. Then you can log in as that user who had that common password. You can get his username, you have his password so you can go from there and try and hack this users email or bank account using these acquired credentials.

Depending on other present bugs, you might be able to **print tables in the SQL database**.

Also, sometimes the webserver prints **error messages for SQL parsing errors** and it tells you the exact SQL string sent to the server. That gives you a lot of information about the query so you can fine-tune your SQL injection attack to do what you want it to do.

Avoiding SQL injection bugs

There is a really good system for making sure that this incorrect parsing never occurs: **prepared statements.**

You should always, always, always use prepared statements. They have a pretty nice simple API that varies from language to language. In a prepared statement you first specify the **parameterized query** and then you give the parameter values.

```
PreparedStatement ps = new PreparedStatement("SELECT * FROM users WHERE username=$1 AND
password=$2");
ps.setStringParam(1, name);
ps.setStringParam(2, password);

sql.executePreparedStatement(ps);
```

When you do `setStringParam(1, name)`, this will ensure that the name will get parsed as a single token that will have the type string, so it will never mess up your query. The way it does that is by escaping whatever it needs to escape in the name parameter.

- Another way of ensuring the parameter will not mess up the query is by having the prepared statement API generate the parse tree for the query, fill in *parameter placeholder nodes* in the parse tree with the parameter values and finally send the parse tree instead of the query string to the server.

You can simply `grep` your source code for `sql.execute` statements (as opposed to the good `sql.executePrepared` statements) to find SQL injection bugs.

Cross-side scripting bugs (XSS)

In order to understand XSS attacks some background information about the HTTP protocol is needed.

Web authentication and access control

In a web app, you log into a websites and then you can browse around. How is this done? Once you log in, the webserver **somehow remembers** that you are authenticated.

The HTTP protocol is stateless, so it does not remember anything from one HTTP request to the next one. The way it remembers information is using **cookies**.

A **cookie** is a small amount of data stored on the client-side and submitted with each HTTP request to the web-server. A cookie stores key-value pairs for a certain web-domain like bank.com

Example: $(bank.com, userid, rob)$

How do cookies work to provide authentication?

You've got the browser and you've got the **example.com web-server**.

1. The browser will say `GET login.html` to the server.
2. The server will send the login page to the browser.
3. The browser displays the page.
4. The user types in the name and password.
5. The browser will send a `POST login.html?name=rob&password=opensesame` to the web-server.

If the user provided the right username and password, the server will send the `accountinfo.html` page and will do a `SetCookie` for example.com with a bunch of name-value pairs: `(example.com {(loginid, "rob")})`.

Every time the browser requests another page from example.com, it will also send the cookie along with the request so that example.com knows who he is and whether he is logged in.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 10th, 2011

Problems:

- The user can fake a cookie with the loginid set to `rob` and then visit the server as `rob`.
- The user can steal cookies from another user and impersonate him

Firesheep is a program that allows an attacker to sniff unencrypted HTTP connections for cookies and then impersonate the users whose cookies he stole.

We need **unforgeable cookies** (use Message Authentication Codes) and **unstealable cookies** (encrypt connection via SSL).

A **MAC** is a tuple $(Gen, Mac, Vrfy)$, where:

- Gen – generate a key of length n
- $Mac_k(m)$ – generates a tag (or a MAC) on the message m using the key k generated by Gen as input
- $Vrfy_k(m, t)$ – used to verify the integrity of the message m that was tagged with the tag t and key k

```
$t' = Mac_k(m)$
if $t' = t$ then output 1 else output $\perp$
```

  - returns 1 when the message is authentic
  - returns $\perp$ when the message has been forged

MAC Security goal: No poly-time adversaries should be able to generate a correct (m, t)-pair such that $Vrfy_k(m, t) = 1$

- We can use a MAC to make the cookies unforgeable. The server can MAC every cookie it sends to the client. The cookies are just messages from the server to itself (since the client just sends them back)
- Cookies can have **an SSL only flag**, which will enforce the transmission of the cookie only over SSL-encrypted channels.

Mashups

People visit many websites: their bank's website or a software piracy site. Some of the sites might not be good sites, so the browser has a policy as to which websites it sends your cookies to.

Each cookie has a domain associated with it, like bank.com so that cookie will only be sent to bank.com. This is known as **the same origin policy**.

Also, when your browser has JavaScript running on it, that JavaScript code came from a particular website like `evil.com`. JavaScript can read cookies, so you don't want `evil.com` JavaScript reading your `bank.com` cookies.

The **same origin policy** covers both of these scenarios:

- to whom cookies get sent
- what cookies JavaScript can read
- also, it limits what domains JavaScript can open AJAX requests to
  - JavaScript code that came from example.com can only open HTTP requests to example.com