Security Lab

In production (professional environments), security is one of the most important issues for web applications. Many applications deal with sensitive data: personal information, payment data, and data that users would like not to be destroyed. Any exploited security flaw can quickly damage the reputation of an application, if not the business behind it.

Obligatory notice: some of the things you'll be trying in this lab are potentially dangerous. Even though you'll be learning about vulnerabilities, accessing or attacking a computer system without authorization is illegal, and people go to prison for it.

Basic tenets

The most fundamental rule of web application security is *don't trust user inputs*. It's pretty simple in theory but it manifests itself in many and sometimes quite subtle ways. As a review, here are the primary ways users provide input to your application:

- GET requests (including the request path `/some/page`, request parameters `?foo=1&bar=2`, and HTTP headers like cookies.)

- POST requests (including the request path, request parameters [though this is not typical for a POST request], HTTP headers, and the request body containing form data.)
- Information sent through web sockets.
- Other HTTP verbs like HEAD, TRACE, OPTIONS, PUT, and DELETE (implemented less often)

We'll concern ourselves mostly with the first two.

The client side (the browser) is NOT SECURE

Nothing that happens in the browser is secure: the user can manipulate everything that happens on a page, from its content to the Javascript that it runs to the requests it makes.

The only thing you have full control over as a developer is the server.

That means, for example:

- That validating information (like an email address) with Javascript in the browser is useless from a security perspective. (Although useful perhaps from a user experience perspective!)
- That any kind of authentication performed on the client side (e.g. checking a password with Javascript in the browser) is useless.

Unauthenticated things are NOT SECURE

If a resource (i.e. a handler, a path, an endpoint) does not check that the request being made is authenticated by a user with appropriate permissions then that resource isn't secure. Imagine, for example, the routes of Insecure Banking Inc.:

```
/login
/myaccount
/transfer-money
```

`login` is the login page for the bank. `myaccount` makes sure the user is authenticated. If so, it shows their balance and assorted interfaces, including a link to `transfer-money`. `transfer-money` moves money between two accounts, as specified by the request.

If `/transfer-money` does not explicitly check that the sending account is authenticated, ANY user can send money from anyone's bank account -- even if they're never shown the interface to it. *An attacker can always make a request to any URL (i.e. any resource on your server) with any parameters of their choice*. This means any path, any parameters, and any headers (and therefore any cookies). If you don't check to make sure in some way that they should be allowed to make that request, then they can.

Things saved in plaintext are NOT SECURE

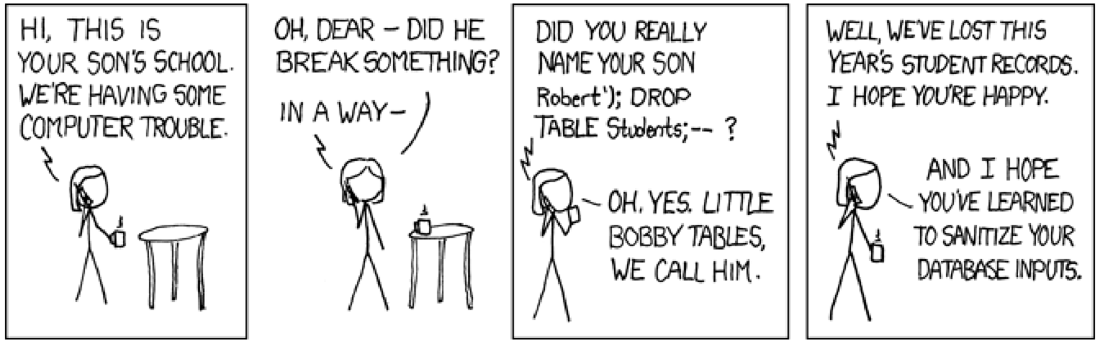
The default protocol for the web, HTTP, sends everything over the internet in plain text. This means that if someone else controls a router between a user and your server (someone else always will) they *can read everything that is sent to your server or sent from it*. This comes up in particular for unsecured Wi-Fi hotspots: everyone in the room can read anything sent via HTTP.

This has a number of implications. First, the user should never be sending sensitive information at all over HTTP: any kind of payment information or other sensitive data could be picked up without a trace. Passwords are a little more complicated because schemes exist to prove that the user has the correct password without ever sending the password to the server.

Second, any authenticated actions the user takes are potentially re-playable if the application doesn't prevent it. An attacker can listen to any request -- even if it's authenticated in a way which makes it impossible to change what the request is -- and simply send it again. This can be addressed by including a token (or nonce) provided by the server in the request, which the server can check for double-submits.

The standard way to address these problems is to switch to HTTPS (HTTP secure). It's a bit of a hassle to set up, and it usually costs a couple bucks, but it more or less transparently solves these issues.

User Inputs


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

This section is where things get interesting. Using this knowledge, you'll exploit security vulnerabilities in a server that you'll be setting up at the end of the assignment.

SQL injection

In a SQL injection attack, an attacker sends a request with input which contains carefully constructed SQL statements. If you aren't careful -- particularly if you use string concatenation to build queries -- attackers are able to run arbitrary commands on your database.

How this works:

Imagine we're writing some code to display user information for their profile page. This will just show the construction of a query string to extract information from the database for that page.

```
//We have their username in some variable user_name:
"SELECT * FROM users WHERE username='" + user_name + "'"

//So if user_name was equal to "Sam" we'd have:
"SELECT * FROM users WHERE username='Sam'"
//(Which would do what we wanted.)

//But what if user_name was equal to:
"'; DROP TABLE users;"

//We would get the following query:
"SELECT * FROM users WHERE username=''; DROP TABLE users;"
//Which is a valid query with predictable consequences.
//(Everything in the users table would be deleted.)
```

NoSQL databases like MongoDB are also vulnerable to SQL injection attacks. Using Mongoose, we could execute a SQL injection with the following:

```
//we have a db with a username and password field
db.users.find({username: username, password: password});

//We don't have strings, so one would think that injection is no longer possible
//However, if the username field comes from a deserialized JSON object,
//we could send that instead...

{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}

//In MongoDB, the field $gt has a special meaning, which is used as the greater
//than comparator. As such, the username and the password from the database will be
//compared to the empty string "", and as a result return a true statement.
app.post('/', function (req, res) {
        db.users.find({username: req.body.username, password: req.body.password}, function (err, users) {
            //db.users.find({username: {"$gt": ""}, password: {"$gt": ""}}
            //this would return everything!
        });
});
```

The best practice to avoid SQL injection is to use stored procedures. Parameterized queries, where the query and data are sent separately, are also relatively secure.

Javascript Injection

Injection occurs when untrusted data is sent to an interpreter as part of a command or a query. The malicious string in the query can trick the interpreter into executing unintended commands, much like SQL injection.

The functions that you should worry about in Javascript are `eval()`, `setTimeout()`, `setInterval()`, and `Function()`. When these are are used to process user provided inputs, it can be exploited by an attacker to inject and execute malicious JavaScript code on server. For instance, let's say we run eval on an input:

```
app.post('/', function (req, res) {
  //using eval to convert the amount ot an integer?!
  var amount = eval(req.body.amount);
  ...
});
```

`eval` will evaluate a Javascript argument as a string. A denial-of-service attack could be executed by sending a wait command,

```
//req.body.amount = 'while(1)'
  eval(while(1))
```

or a kill command.

```
//req.body.amount = 'exit()'
  eval(exit())
```

Perhaps even scarier is that attackers can access contents of the files from the server:

```
//req.body.amount = 'res.end(require('fs').readdirSync('.').toString())'
  eval(res.end(require('fs').readdirSync('.').toString()))
//returns list of files from server
```

Again, preventing server side javascript injection attacks should be done by validating user inputs on the server before processing.

XSS

Cross-site scripting (abbreviated XSS) is an attack where an attacker is able to run some Javascript code in other people's browsers on your site. This allows them to do a number of nasty things:

- Steal the contents of the victim's cookies, and if they are being used for authentication impersonate

the victim.
- Make fraudulent requests from the victim's browser to your site, which will be indistinguishable from a legitimate request by the victim. (This is like CSRF below, but same-site rather than cross-site.)
- Change the content and behavior of your website to anything they want.

XSS is possible when user content is **rendered onto the page directly without escaping HTML**. It's common on website where users share a lot of posts, such as internet forums - users could upload posts or images that contain malicious scripts, and if the server doesn't recognize it as malicious, it is rendered into the HTML and will be parsed by the browser.

Like injection, this can happen when you construct the response to the user with string concatenation. Consider the following example:

I have a route `/search` which takes a query parameter:

```
http://example.com/search?query=Preventing%20XSS
```

(That `%20` is from URL escaping.)

And imagine I construct the title somewhere in my code with my templating agent, where `searchQuery` is a variable with the user's query:

```
<h1>Search for {{searchQuery}} </h1>
```

Then for the above example the resulting HTML would be:

```
<h1>Search for Preventing XSS</h1>
```

But if an attacker chose a malicious URL (hiding the link behind a bit.ly, perhaps) we could have:

```
http://example.com/search?query=<script>/* up to no good here
*/</script>
```

(This would be URL encoded too, but for clarity I haven't encoded it.)

And the result:

```
<h1>Search for <script>/* up to no good here */</script></h1>
```

The attacker has now included Javascript on the search page!

XSS is prevented by carefully controlling the outputs of your application with **escaping** where it's appropriate. *It's important to note that you should not try to escape HTML on your own -- use a templating language which handles this automatically.* (There are many subtle edge cases.)

CSRF

Cross-site request forgery allows attackers to complete a request with the user's own browser.

CSRF is usually done using a user's browser's cookies, especially when they are used to maintain sessions of websites where the user is currently logged in. Normally, browser cookies are sent with every request on a per-domain basis. For instance, If a cookie is set for `example.com`, any request sent to that domain will include the cookies set for `example.com`, *regardless* of how that request was made.

In an attack the victim is somehow lured into clicking a button or link on an attacker-controlled site, (if the attacker also has an XSS vector, "attacker controlled site" means "your site"). That in turn submits a request to another site the user might be logged into. If the user is indeed logged in via cookies those will be sent in the request. This can either be a GET request (through a simple link), or a POST request (though Javascript or a button submitting a hidden form.)

For example, assume a social media site "bookface" has an endpoint `http://bookface.com/make-post` (which we're assuming uses GET parameters, although you shouldn't for actions which are not idempotent). The attacker could include an image in their malicious site:

```
<img src="http://bookface.com/make-post?content=SPAMSPAMSPAMSPAMSPAM"/>
```

Because you're currently logged in, your browser has cookies that maintain your session. Thus, if you prevent users who aren't logged in from using this route by checking the cookie when it's received on the server, an attacker can still maliciously make requests on your behalf. By opening a page with this image, you would end up spamming all your friends on Bookface.

CSRF vulnerabilities can be avoided by associating a unique *token* with the request. The simplest form this takes is the double submit. A token is included in every form submitted and the same token is included in the cookie. On the server side the request is checked to make sure they match. Since the attacker's site can only send cookies through requests, not read the cookies, it's unable to include the correct token.

Your tasks

Your task for these lab days is to setup a server with your project groups, find vulnerabilites in the server, and then modify it so you can defend your server.

You'll be setting up a server on Heroku, since it's free and the TAs are running out of money for the class. Go to this link, and press "Deploy to Heroku" to create the app on Heroku. Alternatively you can see our version of the app here, although you won't be able to make changes to this one.

If you don't have a Heroku account yet, signing up is free. You'll need to provide your credit card during launch, but you shouldn't ever receive a charge for using the service, as it's free for small projects. But if you end up using money and would like to be reimbursed, please let us know.)

Name your project something the TAs can recognize, such as cs132security_group1 (or whatever your group number is). Once you've launched it, you can access it at http://[your project name].herokuapp.com/

To access and update the server code, the easiest way would be to clone the project on Github, and have a local version running on the CS department or your own machine. To sync this git repo with heroku, go to your Heroku Dashboard and select "Deploy", click the Github Deployment method, and fill in the repo that you've cloned in your Github account. This way, you can push changes to the Github account, which Heroku will sync within a few seconds.

The Site

The web application is a retirement savings page, where users login to view their retirement savings and manage their money. There are 3 accounts associated with the site: `user1`, with password `User1_123`, `user2`, with password `User2_123`, and the admin account, `admin`, password `Admin_123`, who has visual access to the benefits page.

The node server may look a little large, but it's quite simple. It uses `express-session` to maintain sessions. `server.js` sets up the dependencies and resources, and routes are handled in `app/routes/index.js` and associated files.

Your job is to find and exploit vulnerabilities in the website. You should be able to find instances of *injection*, *xss*, *csrf*, and other bad security practices. Once you've found vulnerabilities or ways to attack the website, please **document them, including steps required to commit any exploits**, including any code, in a text file. Then, figure out ways to edit the server to defend against these attacks, and upload them to your Heroku site.

Further Resources

Web security is constantly evolving, and don't hesitate to use Wikipedia or other resources to know more about modern web exploits. This lab in particular uses OWASP's Node.js Goat Project, which comes with a tutorial that you can find here. We encourage you to try to find your own exploits and defend against them, but if you're having difficulty, you may read through the tutorial to see what OWASP is doing.