SECURITY

Registration/Login

- Authentication is a well known problem
- Hard to get right
- Don’t roll your own

- Secret information (password)
  - *Don’t store unencrypted on client or server size*
  - *Password managers like Apple’s KeyChain allow information to be stored behind a single password.*
- Biometric information (fingerprint/faceprint)
  - *Can the hardware be subverted so that I can just supply an appropriate string of bits instead.*

History

- Keep a file of passwords where access is limited.
- Points of attack
  - *Anyone with high level access can access, so only need to break one*
  - *Backups/copies of the file need to be protected as well.*

History

- Keep a file of passwords where access is limited and we encrypt the passwords
- Points of attack
  - *Anyone with high level access can access, so only need to break one password.*
  - *Backups/copies of the file need to be protected as well.*

History – Classic Unix

- Never store the password.
- The password was used as a key (8 ASCII characters for 56bits) to an encryption algorithm (DES) that is applied to a standard chunk of text (64bits of zeros). Encrypt the new text again. Repeat 25 times and store the result.
- Authentication: You enter your password and it is used to encrypt the block of text 25 times. If it matches what was stored, authentication the user.
- Relies on the difficulty of decryption.
- Points of Attack
  - *Brute Force. Try all possible passwords with a given user name.*
  - *Dictionary Attack - Try lots of possible passwords and compare the encrypted results to the stored values. Not targeted at one person. Make sure the dictionary has common words.*
  - *Man in the Middle: There is still a password coming in which might be intercepted.*

History – Classic Unix

- DES in software was slow – As much as one second to do the encryption process.
  - *Modern computers can try up to 10 million passwords in 1 second.*
- Using salt. When a password is changed, generate a random string of characters and use that to change the encryption process. The salt is then stored with the encrypted text, so it can be used in the authentication process.
  - *Classic Unix based the salt off of the time of day, so some salt values were more likely than others.*
  - *Classic Unix only used 12bits of salt which does not give much protection at this point.*
- In classic Unix only the first 8 characters of your password were significant.

History – Modern Unix

- Passwd has useful information that is widely needed, but it is too risky to make the encrypted text visible to all. Split the storage so that the sensitive information is hidden in /etc/shadow.
- Use better encryption: Blowfish or MDS
  - *Use more bits for the key*
  - *Use more bits for the salt*
- MD5 is considered to be secure and is often used in Unix/Linix
- SHA can be used. It is a standard for SSL certificates. SHA-1 is considered weak and SHA-2 is in current use.
- An interesting reference.

Secure Hash Codes

- A cryptographic hash is intended to be a one way function.
- Hashing combined with encryption.
- A larger document is reduced to a value in a smaller range.
- Goal is that given a document and its hash, it should be difficult to find a second document with the same hash.
  - *We can use the hash as a digital finger print.*
  - *I publish the file and the fingerprint and then you can verify that the file has not been tampered with by computing the hash.*

One-Time Passwords

- Tokens
  - *Hardware: Passwords are generated on a hardware device on a continuing basis. If a user attempts to log in, they are presented with a challenge. The correct answer is obtained from their hardware.*
  - *Programmatic: Program running on cell phone can generate the codes.*
  - *Text: A text containing the code is sent via text to a registered device*

Passport and JS

- Passport is middle ware that works with node/express apps.
- Authenticate by
  - *Username/password*
  - *Twitter*
  - *Facebook*
  - *Google*
  - *And others*
- Single sign-on using OAuth services.
- Different authorization mechanisms are supported by individual packages.

Configuration

- We need to let Passport know how authentication is going to managed.
- Verify callback
  - *Callback failure*
  - *Callback sucess*

Distinguish between DB failure and authentication denied.

```
var passport = require('passport’);
var LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
    function(username, password, done) {
       User.findOne({ username: username }, function (err, user){
           if (err) { return done(err); } //DB fail
           if (!user) { return done(null, false,
               { message: 'Incorrect username.' }); }
           if (!user.validPassword(password))
               { return done(null, false,
               { message: 'Incorrect password.' }); }
           return done(null, user); }); } ));
```

Configuration – Middle Ware

- We need to add Passport to the stack of packages that are going to be applied to requests.

```
var session = require('express-session');
var bodyParser = require('body-parser');
app.use(express.static("public"));
app.use(session({secret: 'A secret'); // Do before passport session
app.use(bodyParser.urlencoded({ extended: false}));
app.use(passport.initialize()); // needed
app.use(passport.session());    // optional
```

Login

- Login on Post with form data.
- Using local strategy.
- On authentication, user is redirected to a user home page.

```
app.post('/login’,
    passport.authenticate('local’),
    function(req, res) {
        // If this function gets called,
        //authentication was successful.
        // `req.user` contains the authenticated user.
        res.redirect('/users/' + req.user.username);
});
```

Login – More options

- Success goes to a home page and Failure sends you back to login.
- On failure, flash a message. Can specialize it as needed.
- Don’t give too much information about the failure.
  - *Bad: Invalid username*
  - *Better: Invalid username/password*

```
app.post('/login’,
    passport.authenticate('local’,
        { successRedirect: '/’,
        failureRedirect: '/login’,
        failureFlash: true }));
```

Login - Sessions

- Once authentication occurs a persistent session is maintained in a cookie
  - *We can disable if not required.*
- We can provide callbacks if more specialized handling is required on success/failure.

Login - Local

- Passport-local strategy is a standard username and password.
- Standard form with a div that triggers a post to /login

```
<form action="/login" method="post">
    <div>
        <label>Username:</label>
        <input type="text" name="username"/> </div>
    <div>
        <label>Password:</label>
        <input type="password" name="password"/> </div>
    <div>
        <input type="submit" value="Log In"/> </div>
</form>
```

OAuth 2.0

- We can use another app like Google or Twitter to perform the authentication. There is an earlier standard, but you should be using the newest one for the best security.
- When authorized, our app is given a token to use as a credential.
  - *Our app does not need to store usernames and passwords*
  - *The token can indicate access level (e.g. read only.)*

OAuth 2.0 Steps

- The application requests permission from the user to access protected resources.
- A token is issued to the application if permission is granted.
- The application uses the token to authenticate and allow access to the protected resources.

- Need a server. Easier to piggy back.

OAuth 2.0 Tokens

- While there is a framework for specifying kinds of tokens, the most common type is a bearer token.

```
passport.use(new BearerStrategy(
        function(token, done) {
                User.findOne({ token: token }, function (err, user) {
                        if (err) { return done(err); }
                        if (!user) { return done(null, false); }
                        return done(null, user, { scope: 'read' });
                });
        }
));
```

Protecting Endpoints (Session based)

- We can protect endpoints associated with a view by checking to see if req.user is defined.
- If so, we pass control on, otherwise, remember the URL for the endpoint we are protecting and redirect to a login view.

```
const secured = (req, res, next) => {
   if (req.user){ return next();}
   req.session.returnTo = req.originalUrl;
   res.redirect("/login");
}
```

Protecting Endpoints

- We can require authorization to guard protected resources in our API using tokens.

```
app.get('/api/me',
        passport.authenticate('bearer', { session: false }),
        function(req, res) {
                res.json(req.user);
        });
```

Modified Login

- If we do the redirect as in the previous slide, we should modify the login router code to direct back to the saved endpoint URL (if there is one)

```
router.post('/login', passport.authenticate('local’),
    function(req, res) {
        if(req.session.returnTo)
            res.redirect(req.session.returnTo);
        res.redirect('/');
});
```

Endpoints

- Change

```
/* GET update costume page */
router.get('/update',
      costume_controlers.costume_update_Page)


/* GET update costume page */
router.get('/update', secured,
      costume_controlers.costume_update_Page)
```

Setting timeouts

- We can limit the time that a token remains valid (default is 1 year).

- We can limit the time that a session remains valid

```
app.use(session({secret: ‘Cookie secret’,
     cookie:{_expires : 60000000}, // time im ms })
);
```

Logout

- Authentication will typically persist.
- We can call the function logout() from any route handler.
- req.user is removed and the login session is cleared.

```
app.get(‘/logout’,  function(req, res) {
        req.logout();
        res.redirect(‘/’);
});
```

Twitter example

```
passport.use(new TwitterStrategy({
    consumerKey: TWITTER_CONSUMER_KEY,
    consumerSecret: TWITTER_CONSUMER_SECRET,
    callbackURL: http://www.example.com/auth/twitter/callback}
function(token, tokenSecret, profile, done) {
    Account.findOne({ domain: 'twitter.com', uid: profile.id },
        function(err, account) {
            if (err) { return done(err); }
            if (account) { return done(null, account); }

    var account = new Account();
    account.domain = 'twitter.com’;
    account.uid = profile.id;
    var t = { kind: 'oauth', token: token,
        attributes: { tokenSecret: tokenSecret } };
    account.tokens.push(t);
    return done(null, account);
```

Twitter example

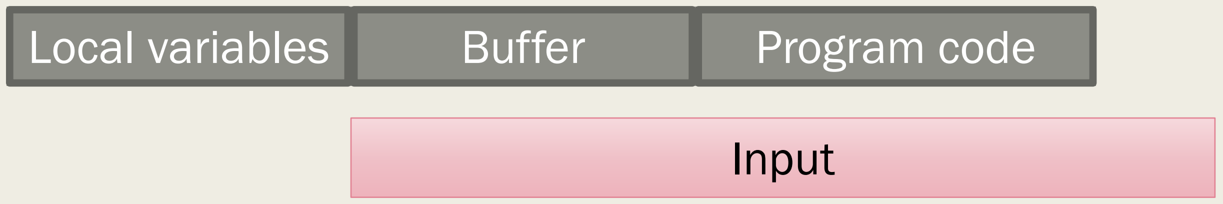
```
app.get('/connect/twitter’,
    passport.authorize('twitter-authz', {
        failureRedirect: '/account' }) );

app.get('/connect/twitter/callback’,
    passport.authorize('twitter-authz', {
        failureRedirect: '/account' }),
    function(req, res) {
        var user = req.user;
        var account = req.account;
        // Associate the Twitter account with the logged-in user.
        account.userId = user.id;
        account.save(function(err) {
            if (err) {
                return self.error(err); }
            self.redirect('/’);
        });
    }
);
```

Scripting Attacks

- Making unwarranted assumptions about the form of input/data leading to foreign code being executed.
- Allow data to be executed as code.
- Scripting languages have a higher degree of vulnerability because of the ease of building strings of code and then executing them.

Buffer Overflow

- Classic you don’t see often any more.
- Input is a string
- We have a fixed size space set aside to hold it. Code and data are not separated.



SQL Injection

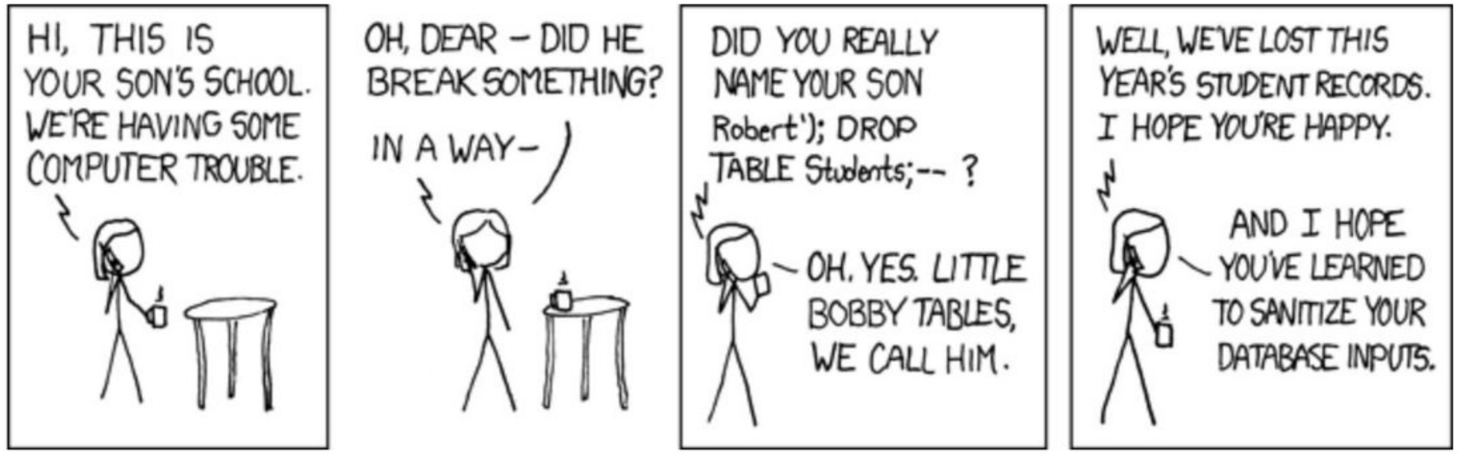
- May access/modify data
- Gain access to back-end networks
- Risk (High)

```
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

```
SELECT * FROM Users WHERE UserId = 74 OR 1=1
SELECT * FROM Users WHERE UserId = 74; DROP TABLE Users
```

SQL Injection

From the webcomic *xkcd*


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.

https://xkcd.com/327/

SQL Injection Prevention

- Layered approach
  - *Validate/Sanitize all user input (HtmlEncode)*
    - Don’t trust DBs and HTML requests
  - *Static analysis tools*
  - *Use Parameterized SQL to distinguish between code and data.*
  - *Concatenation is dangerous*
  - *Stored Procedures can help*
  - *Train to use best practices*

SQL Parameters

- Parameters are cleaned and treated literals, not part of the query.
- txtSQL = "SELECT * FROM Users WHERE UserId = @0"; db.Execute(txtSQL,txtUserId);

HTML Injection

- Use input to construct HTML
- Manipulate tags to start a script section
- Changes are to the local HTML
  - *Can put links onto page of user.*
  - *Can redirect to login page and capture login credentials*

- Not as risky as SQL injection

HTML Injection Example

```
<html>
  <h1>Here are the results that match your query: </h1>
  <h2>{user-query}</h2>
    <ol>
      <li> Result A
      <li> Result B
    </ol> </html>
```

```
</h2>special offer <a href=www.attacker.site>malicious link</a><h2>
```

- Closes

JS Injection

- Introduce data that will be interpreted as code on the client side.
- Introduced via a script tag
- Risks (Moderate)
  - *Change to displayed information*
  - *Leak data*
  - *Leak credentials*
  - *Manipulation of cookie data*

JS Injection Example

document.getElementById(“Thank”).innerHTML=” Thank you for filling our questionnaire, ”+user;

<html> <body> <script> alert('Hello, world!'); </script> </body> </html>

XSS Attack

- Cross site Scripting (XSS).
- Code is injected that will run on the Client side.
- Malicious code can be kept on the server and applied on every page.
- Risks (High)
  - *Change to displayed information*
  - *Leak data*
  - *Leak credentials*
  - *Manipulation of cookie data*
- If the malicious code can leak the session cookie, it can be used by the attacker to login on the original users account.
- Similar to JS injection, but does not need script tags to facilitate the attack.

XSS Example

```
<body onload=“theAttackCode()”> ;
<b onmouseover=“theAttackCode()”>;
```

XSS Attack Main forms

- Via malicious scripts executed on the client side
- Fake page or form displayed to the user that requests credentials from the user
- Via advertisments on the website pages.
- Via malicious emails

XSS Attack Approaches

- Reflected XSS
  - *Malicious code is not stored on the server, inserted into the view*
- Stored XSS
  - *Malicious code is stored on the server permanently*
- DOM
  - *Dom environment is changed, but the code is unchanged.*

XSS Reflected Example

User enters input:

```
        <script>function(send_user_err)
```

Which then is redisplayed as an error.

XSS Stored Example

User enters input that is stored as a value in a data base which contains the code.

Value is retrieved and displayed in page triggering the code

XSS DOM Example

There is no change to the client side code.

Value containing malicious code is in the environment of the DOM. So we could have a query parameter that is supposed to be a number, but we supply something else instead.

Expected

http://site.com/item.html?select=13

Actual

http://site.com/item.html?select=<script>alert(document.cookie)</script>

XSS Prevention

Validate Data – Verify that the input is of the right form

Filter – Search for dangerous text and remove it

<script> tags, JS commands, HTMO

Escape – For any problematic special character, add an escape character so that it is treated as data instead of code.

Example of this is PUG with #{} doing escaping

Check the OWASP cheat sheet for guidelines.

References

- Passport
- Authentication and Authorization with a RESTful API
- An article about Cookies and secrets

- OWASP
- HTML Injection examples.