A2-Broken Authentication and Session Management

Exploitability: AVERAGE | Prevalence: WIDESPREAD | Detectability: AVERAGE | Technical Impact: SEVERE

Description

In this attack, an attacker (who can be anonymous external attacker, a user with own account who may attempt to steal data from accounts, or an insider wanting to disguise his or her actions) uses leaks or flaws in the authentication or session management functions to impersonate other users. Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.

A2 - 1 Session Management

Description

Session management is a critical piece of application security. It is broader risk, and requires developers take care of protecting session id, user credential secure storage, session duration, and protecting critical session data in transit.

Attack Mechanics

Scenario #1: Application timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #2: Attacker acts as a man-in-middle and acquires user's session id from network traffic. Then uses this authenticated session id to connect to application without needing to enter user name and password.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.

How Do I Prevent It?

Session management related security issues can be prevented by taking these measures:

- User authentication credentials should be protected when stored using hashing or encryption.
- Session IDs should not be exposed in the URL (e.g., URL rewriting).
- Session IDs should timeout. User sessions or authentication tokens should get properly invalidated during logout.
- Session IDs should be recreated after successful login.
- Passwords, session IDs, and other credentials should not be sent over unencrypted connections.

Source Code Examples

In the insecure demo app, following issues exists:

1. Protecting user credentials

password gets stored in database in plain text . Here is related code in `data/user-dao.js` `addUser()` method:

```
// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: password //received from request param
};
```

To secure it, handle password storage in a safer way by using one way encryption using salt hashing as below:

```
// Generate password hash
var salt = bcrypt.genSaltSync();
var passwordHash = bcrypt.hashSync(password, salt);

// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: passwordHash
};
```

This hash password can not be decrypted, hence more secure. To compare the password when user logs in, the user entered password gets converted to hash and compared with the hash in storage.

```
if (bcrypt.compareSync(password, user.password)) {
    callback(null, user);
} else {
    callback(invalidPasswordError, null);
}
```

Note: The bcrypt module also provides asynchronous methods for creating and comparing hash.

2. Session timeout and protecting cookies in transit

The insecure demo application does not contain any provision to timeout user session. The session stays active until user explicitly logs out.

In addition to that, the app does not prevent cookies being accessed in script, making application vulnerable to Cross Site Scripting (XSS) attacks. Also cookies are not prevented to get sent on insecure HTTP connection.

To secure the application:

1. Use session based timeouts, terminate session when browser closes.

```
// Enable session management using express middleware
app.use(express.cookieParser());
```

2. In addition, sets `HTTPOnly` HTTP header preventing cookies being accessed by scripts. The application used HTTPS secure connections, and cookies are configured to be sent only on Secure HTTPS connections by setting `Secure` flag.

```
app.use(express.session({
    secret: "s3Cur3",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

3. When user clicks logout, destroy the session and session cookie

```
req.session.destroy(function() {
    res.redirect("/");
});
```

Note: The example code uses `MemoryStore` to manage session data, which is not designed for production environment, as it will leak memory, and will not scale past a single process. Use database based storage MongoStore or RedisStore for production. Alternatively, sessions can be managed using popular passport module.

3. Session hijacking

The insecure demo application does not regenerate a new session id upon user's login, therefore rendering a vulnerability of session hijacking if an attacker is able to somehow steal the cookie with the session id and use it.

Upon login, a security best practice with regards to cookies session management would be to regenerate the session id so that if an id was already created for a user on an insecure medium (i.e: non-HTTPS website or otherwise), or if an attacker was able to get their hands on the cookie id before the user logged-in, then the old session id will render useless as the logged-in user with new privileges holds a new session id now.

To secure the application:

1. Re-generate a new session id upon login (and best practice is to keep regenerating them upon requests or at least upon sensitive actions like a user's password reset. Re-generate a session id as follows: By wrapping the below code as a function callback for the method req.session.regenerate()

```
req.session.regenerate(function() {

  req.session.userId = user._id;

  if (user.isAdmin) {
    return res.redirect("/benefits");
  } else {
    return res.redirect("/dashboard");
  }

})
```

Further Reading

- Helmet (https://npmjs.org/package/helmet) Security header middleware collection for express
- Seven Web Server HTTP Headers that Improve Web Application Security for Free (http://recxltd.blogspot.sg/2012/03/seven-web-server-http-headers-that.html)
- Passport (http://passportjs.org/guide/authenticate/) authentication middleware
- CWE-384: Session Fixation (http://en.wikipedia.org/wiki/Session_fixation)

A2 - 2 Password Guessing Attacks

Description

Implementing a robust minimum password criteria (minimum length and complexity) can make it difficult for attacker to guess password.

Attack Mechanics

The attacker can exploit this vulnerability by brute force password guessing, more likely using tools that generate random passwords.

How Do I Prevent It?

Password length

Minimum passwords length should be at least eight (8) characters long. Combining this length with complexity makes a password difficult to guess and/or brute force.

Password complexity

Password characters should be a combination of alphanumeric characters. Alphanumeric characters consist of letters, numbers, punctuation marks, mathematical and other conventional symbols.

Username/Password Enumeration

Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code

Additional Measures

- For additional protection against brute forcing, enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed.
- Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception. Enforce the changing of temporary passwords on the next use. Temporary passwords and links should have a short expiration time.

Source Code Example

The demo application doesn't enforce strong password. In routes/session.js `validateSignup()` method, the regex for password enforcement is simply

```
var PASS_RE = /^.{1,20}$/;
```

A stronger password can be enforced using the regex below, which requires at least 8 character password with numbers and both lowercase and uppercase letters.

```
var PASS_RE =/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;
```

Another issue, in routes/session.js, the `handleLoginRequest()` enumerated whether password was incorrect or user doesn't exist.This information can be valuable to an attacker with brute forcing attempts. This can be easily fixed using a generic error message such as "Invalid username and/or password".