Denial of Service Attack Report

Disclaimer: All presented attacks were executed against a test application for educational purposes only. The author of this report is not responsible for any illegal actions that readers of this report may take against real (production) systems. Use the information provided responsibly and with care!

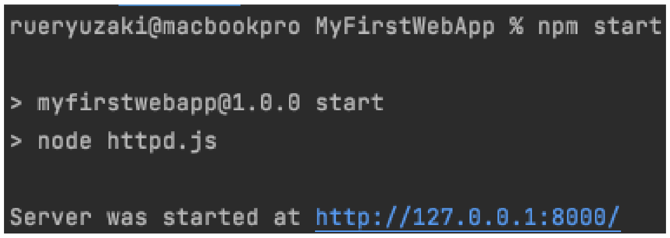
Author: Adil Nurmanov (a.nurmanov1998@gmail.com).

Abstract: A denial of service (DOS) attack is a type of attack that targets the availability of the service and makes a server reject all requests from legitimate clients. The main goal of this attack is to make a resource unavailable for the purposes it was designed for [1]. In this report, several DOS attacks are presented with a step-by-step guide in order to reproduce them. After exploiting and understanding the roots of the vulnerabilities patches will be presented in order to eliminate them. Code samples will be run on the NodeJS platform but the same vulnerabilities may present in other platforms as well.

Prerequisites:

- Source code for target applications (includes both unpatched and patched versions) [4].
- NodeJS runtime.
- Npm.

To start a sample application, download source code for an unpatched app, open the terminal, go inside the *MyFirstWebApp* directory and run the "*npm start"* command. If command was successful than output should look like this:



```
rueryuzaki@macbookpro MyFirstWebApp % npm start

> myfirstwebapp@1.0.0 start
> node httpd.js

Server was started at http://127.0.0.1:8000/
```

Picture 1 - Application run

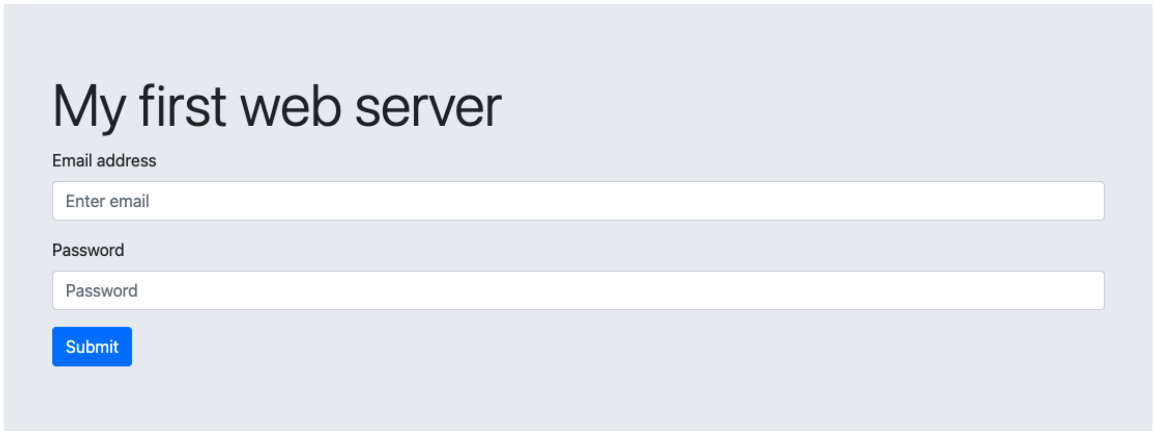
1. **Improper error handling**

Sometimes applications do not work as intended by programmers. Sooner or later inside every application occurs an error. This error can be either expected or unexpected.

The expected errors are errors for which a programmer has created a piece of logic to handle it correctly. For example, if the requested file was not found on the server it should return a 404 (NOT FOUND) response to tell the client that the requested file does not exist on the server (we assume a web server).

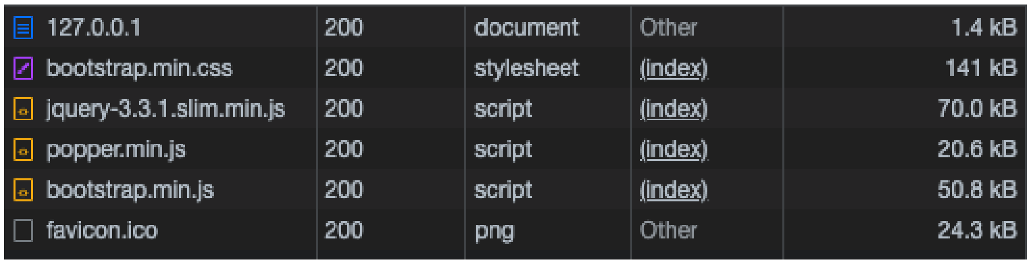
The unexpected errors are errors for which no handling logic was created by a programmer. For example, if the user uploaded file exceeds the amount of memory that server has then server can crash and no future client requests will be processed. This error can happen if a programmer did not check the size of a file before processing it on the server.

Exploit: If we try to access a target app at http://127.0.0.1:8000/ we should see the page like below.



Picture 2 - Main page of the app

We can see that this is a web server which serves some static content like html, css and javascript. The list of all resources that the browser is fetching is presented below.

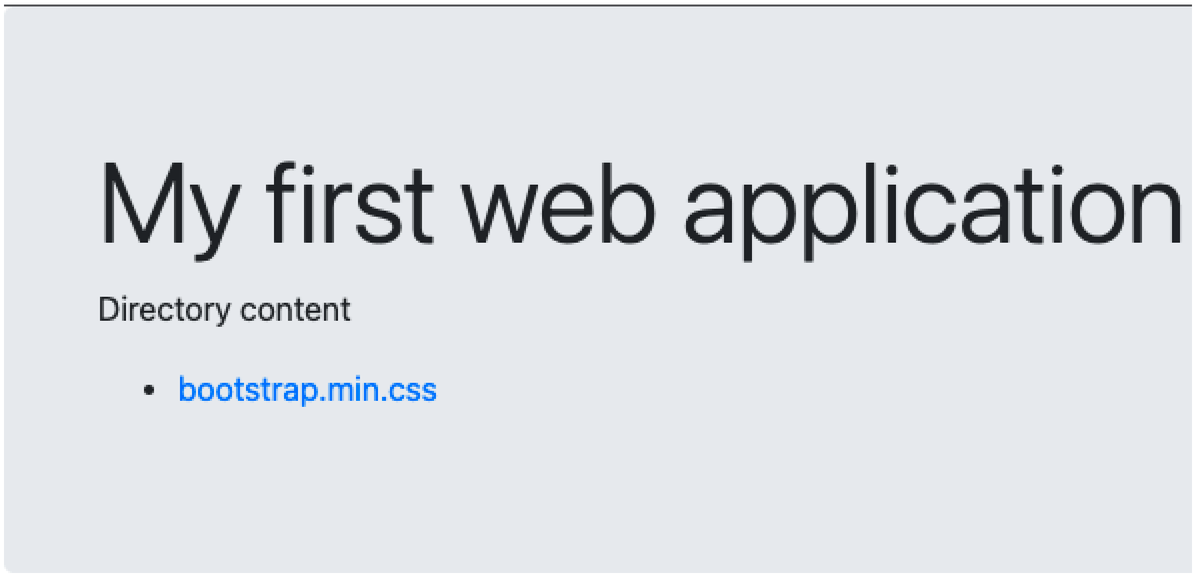


127.0.0.1	200	document	Other	1.4 kB
bootstrap.min.css	200	stylesheet	(index)	141 kB
jquery-3.3.1.slim.min.js	200	script	(index)	70.0 kB
popper.min.js	200	script	(index)	20.6 kB
bootstrap.min.js	200	script	(index)	50.8 kB
favicon.ico	200	png	Other	24.3 kB

Picture 3 - List of resources that the browser has fetched

In order to render the main page browser needs to fetch all the declared resources in the HTML file. Browser then tries to get each resource from the server sequentially one by one.

We can access the directory where css files are stored like this http://127.0.0.1:8000/css

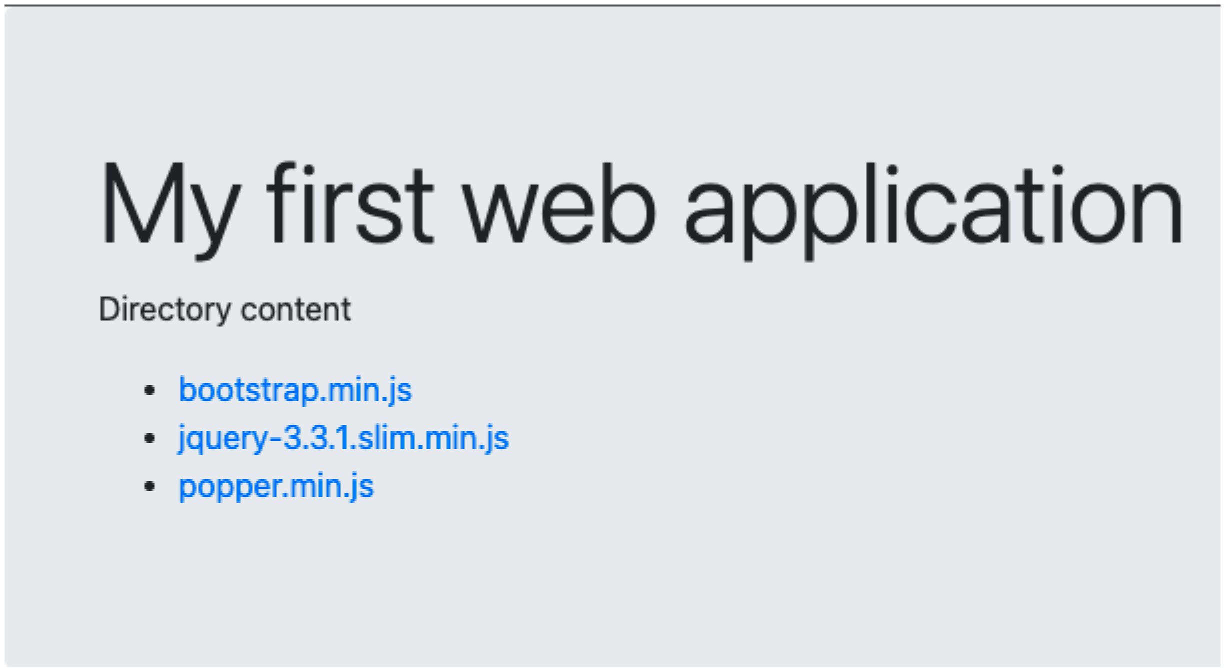

My first web application

Directory content

- bootstrap.min.css

Picture 4 - List of directory where css files are stored

Javascript files can be accessed like this http://127.0.0.1:8000/js


My first web application

Directory content

- bootstrap.min.js
- jquery-3.3.1.slim.min.js
- popper.min.js

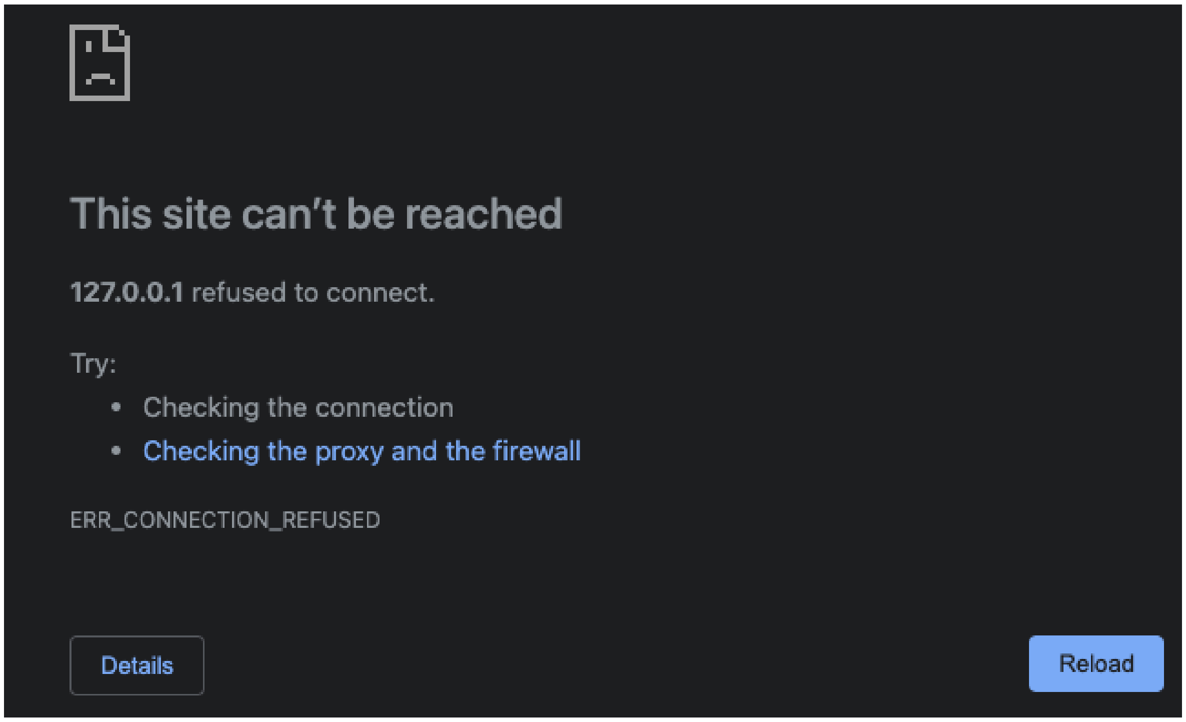
Picture 5 - List of directory where javascript files are stored

We can even access a single file from the browser like this http://127.0.0.1:8000/js/jquery-3.3.1.slim.min.js

All those files are presented at the server as static content which means that they are not changed dynamically and must be presented on the server unmodified.

But what happens if we try to access a non-existing resource?

Let’s try to request http://127.0.0.1:8000/js/not-existing-file.js


This site can’t be reached

127.0.0.1 refused to connect.

Try:

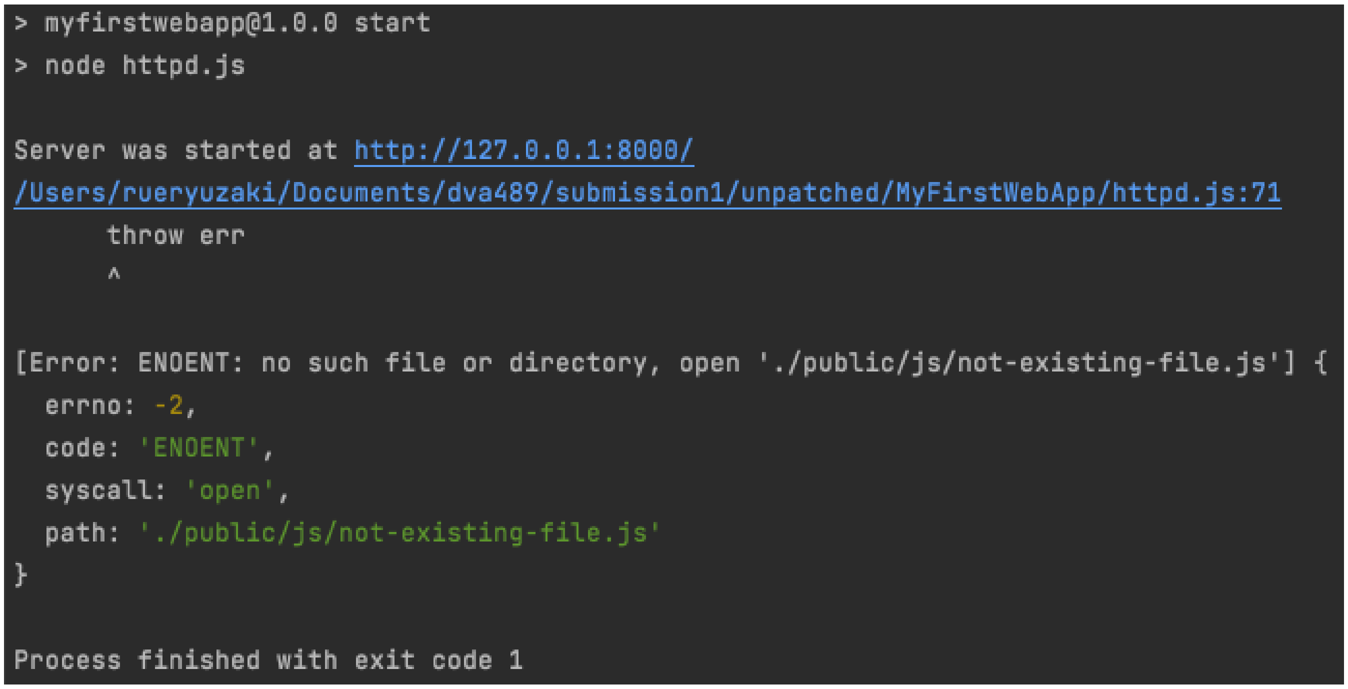
- Checking the connection
- Checking the proxy and the firewall

ERR_CONNECTION_REFUSED

Picture 6 - Trying to access a non-existing resource

Server rejected our request and no file content was given because the requested file does not exist on the server. If we try to access the main page of our application we will get the same response as on the above picture. It looks like our application is not working any more!

If we look at the server logs we should find the following error message.

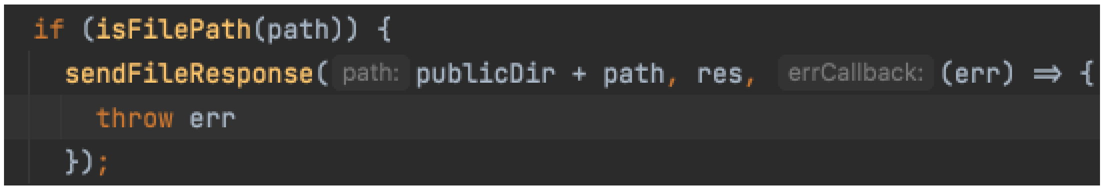


```
> myfirstwebapp@1.0.0 start
> node httpd.js

Server was started at http://127.0.0.1:8000/
/Users/rueryuzaki/Documents/dva489/submission1/unpatched/MyFirstWebApp/httpd.js:71
      throw err
      ^

[Error: ENOENT: no such file or directory, open './public/js/not-existing-file.js'] {
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './public/js/not-existing-file.js'
}

Process finished with exit code 1
```

Picture 7 - Error message

We can see that the error has happened because the requested file was not found in the public directory of our target application which resulted in the error with code ENOENT which means that the file was not found in the file system of the server.

We can even find at which line the error has occurred (71st line of the script).



```
if (isFilePath(path)) {
  sendFileResponse( path: publicDir + path, res,  errCallback: (err) => {
    throw err
  });
```

Picture 8 - Script logic for serving user requested file

Looks like this segment of code tries to resolve file path and serve it to the client and if it fails to do so it raises an error. The problem with this logic is that if the user requested file does not exist on the server then an error will be thrown and if no handling logic is defined then the application will exit with an error code(like on picture 7).

Generally, it means that if someone requests a non-existing resource on the server it will crash with an error and no future client requests will be served which results in a DOS attack.

Mitigation: To eliminate this vulnerability we need to add some validation checks. Those validation checks must look up the requested resources and if the requested resource was not found then answer with appropriate message and status code. Usually a server will respond with a 404 (NOT FOUND) message back to the user if the requested resource was not found on the server. Statuses that start with 4XX usually indicate client side errors, so clients can change their requests appropriately.

The first step is to create *notfound.template* template file to return error message back to the user.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <title>Not found</title>
</head>
<body>
<p>The requested path was not found {{message}}</p>
</body>
</html>
```

The second step is to create a new function *sendNotFoundResponse(res, path)* which will be used to send a response with 404(NOT FOUND) status code and processed *notfound.template* content.

```
/**
* Send not found response
*
* @param res response
* @param path path
*/
function sendNotFoundResponse(res, path) {
 fs.readFile(templateDir + '/notfound.template', (err, data) => {
   if (err) {
     console.error(err);
     res.statusCode = 500;
     res.end();
   } else {
     res.statusCode = 404;
     res.end(data.toString().replace('{{message}}', path));
   }
 });
}
```

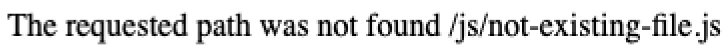
If *notfound.template* file is not found in the directory then it is considered a programming error because a developer did not put it in the expected place. Accordingly, the server must return 500 (INTERNAL SERVER ERROR) status code and log an error message to the console.

This function must be called every time when the requested path was not found on the server. So we need to detect all places in our script where user requested files are tried to be found on the server. As you can see all static content is served from the *staticContent(req, res)* function.

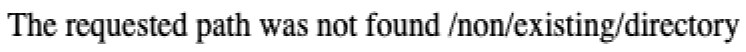
```
/**
* Send static content
*
* @param req request
* @param res response
*/
function staticContent(req, res) {
 let reqUrl = url.parse(req.url, true);
 let path = decodeURIComponent(reqUrl.pathname);
 if (isFilePath(path)) {
   sendFileResponse(publicDir + path, res, () => sendNotFoundResponse(res, path));
 } else {
   if (path === '/') {
     sendFileResponse(publicDir + '/index.html', res, (err) => {
       console.error(err);
       res.statusCode = 500;
       res.end();
     });
   } else {
     sendFileResponse(publicDir + path + '/index.html', res, () => {
       fs.readdir(publicDir + path, (err, files) => {
         if (err) {
           sendNotFoundResponse(res, path);
         } else {
           let dirContent = '';
           if (files && files.length > 0) {
             files.forEach(file => {
               dirContent += `<li><a href="${'http://' + host + ':' + port + path + '/' +
file}">${file}</a></li>`
             });
           } else {
             dirContent = '<li>Directory is empty!</li>';
           }
           fs.readFile(templateDir + '/dircontent.template', (err, data) => {
             if (err) {
               console.error(err);
               res.statusCode = 500;
               res.end();
             } else {
               res.statusCode = 200;
               res.end(data.toString().replace('{{dirContent}}', dirContent));
             }
           });
         }
       });
```

```
            });
          }
        }
      }
```

If we run the server with the patched code and try to access a non-existing resource again we should get the correct error messages like below.


The requested path was not found /js/not-existing-file.js

Picture 9 - Trying to access non-existing file


The requested path was not found /non/existing/directory

Picture 10 - Trying to access non-existing directory

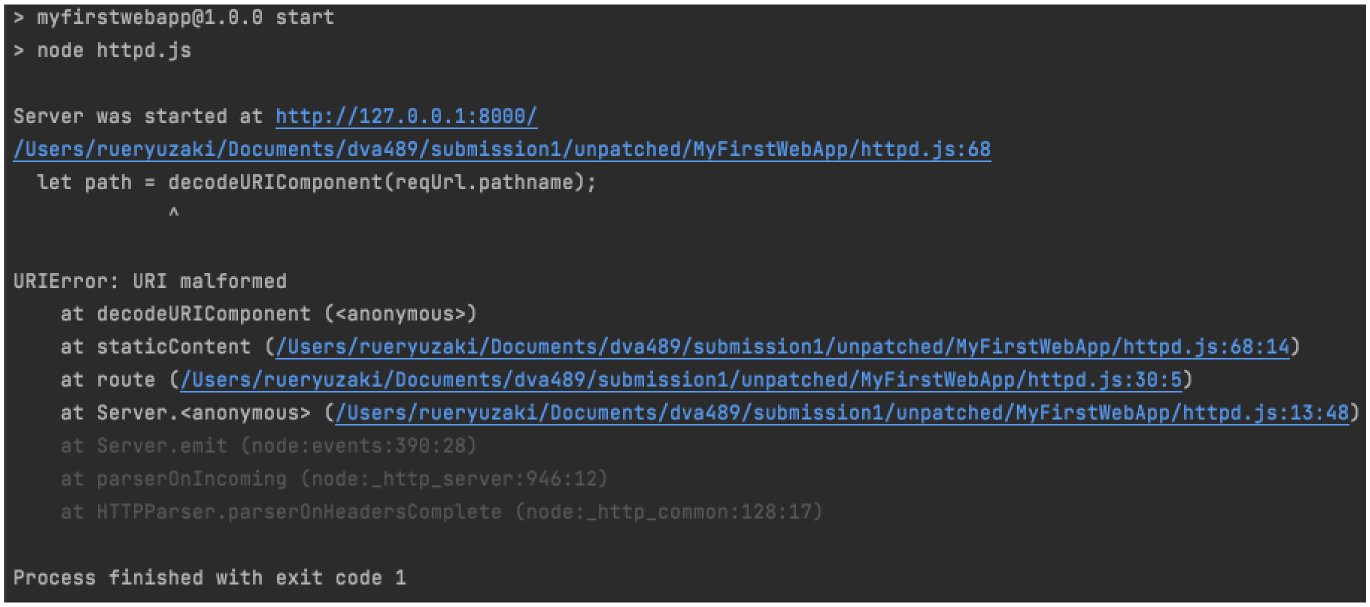
Even after accessing the non-existing resource we can go back to the main page. But at this time the application did not crash!

2. **Malformed URI sequence**

When it comes to sending links to different parts of your application care should be taken while parsing and normalizing user supplied links. A URL is composed from a limited set of characters belonging to the US-ASCII character set. These characters include digits (0-9), letters(A-Z, a-z), and a few special characters ("-", ".", "_", "~"). ASCII control characters (e.g. backspace, vertical tab, horizontal tab, line feed etc), unsafe characters like space, \, <, >, {, } etc, and any character outside the ASCII charset is not allowed to be placed directly within URLs [2].

If an attacker manages to successfully provide a malformed URL to the vulnerable part of the application it can crash the server enabling DOS attack.

Exploit: If we try to access our test application using this url http://127.0.0.1:8000/%E0%A4%A we should see that it crashed with an error.

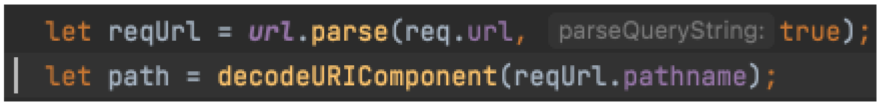


```
> myfirstwebapp@1.0.0 start
> node httpd.js

Server was started at http://127.0.0.1:8000/
/Users/rueryuzaki/Documents/dva489/submission1/unpatched/MyFirstWebApp/httpd.js:68
  let path = decodeURIComponent(reqUrl.pathname);
             ^

URIError: URI malformed
    at decodeURIComponent (<anonymous>)
    at staticContent (/Users/rueryuzaki/Documents/dva489/submission1/unpatched/MyFirstWebApp/httpd.js:68:14)
    at route (/Users/rueryuzaki/Documents/dva489/submission1/unpatched/MyFirstWebApp/httpd.js:30:5)
    at Server.<anonymous> (/Users/rueryuzaki/Documents/dva489/submission1/unpatched/MyFirstWebApp/httpd.js:13:48)
    at Server.emit (node:events:390:28)
    at parserOnIncoming (node:_http_server:946:12)
    at HTTPParser.parserOnHeadersComplete (node:_http_common:128:17)

Process finished with exit code 1
```

Picture 11 - Error message

Looks like our application failed to parse the pathname that was provided by the client and it resulted in an unhandled URIError exception. The same example can be found in the official documentation for JS [3].

Vulnerable part of the code is shown below.



```
let reqUrl = url.parse(req.url, parseQueryString: true);
let path = decodeURIComponent(reqUrl.pathname);
```

Picture 12 - Vulnerable part of code

Mitigation: According to the documentation [3] decodeURIComponent(string) function can throw an error if the provided string contains a malformed URI sequence. In the exploit the malformed part of the URI was %E0%A4%A. The last part of this sequence contains only one letter after % symbol which results in error. It is also clear that this type of error is a client error which means that the server should respond with a 400(BAD REQUEST) response.

Because the logic for parsing user requested URI is used in more than one place it makes sense to wrap this logic in a dedicated function.

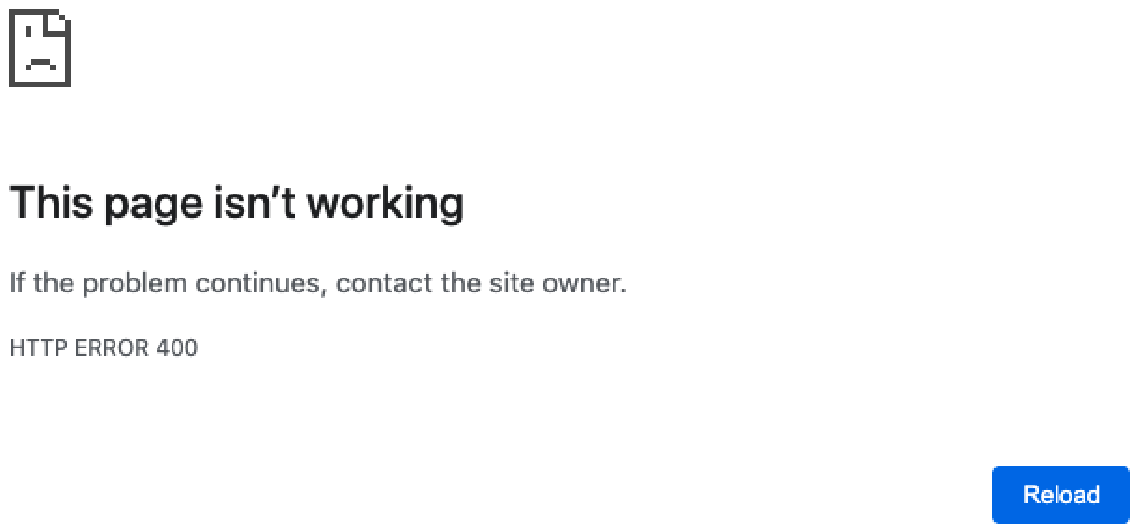
```
/**
* Parse URI
*
* @param req request
* @returns parsed URI
*/
function parseUri(req) {
 let reqUrl = url.parse(req.url, true);
 let path = decodeURIComponent(reqUrl.pathname);
 return {reqUrl, path};
}
```

Call to this function should be wrapped in try catch statement because decodeURIComponent function can throw an error so we need to handle it to process it appropriately.

```
/**
* Send static content
*
* @param req request
* @param res response
*/
function staticContent(req, res) {
 try {
   const {path} = parseUri(req, res);
   if (isFilePath(path)) {
     sendFileResponse(publicDir + path, res, () => sendNotFoundResponse(res, path));
   } else {
     if (path === '/') {
       sendFileResponse(publicDir + '/index.html', res, (err) => {
         console.error(err);
         res.statusCode = 500;
         res.end();
       });
     } else {
       sendFileResponse(publicDir + path + '/index.html', res, () => {
         fs.readdir(publicDir + path, (err, files) => {
           if (err) {
             sendNotFoundResponse(res, path);
           } else {
             let dirContent = '';
             if (files && files.length > 0) {
               files.forEach(file => {
                 dirContent += `<li><a href="${'http://' + host + ':' + port + path + '/' + file}">${file}</a></li>`
               });
             } else {
               dirContent = '<li>Directory is empty!</li>';
             }
             fs.readFile(templateDir + '/dircontent.template', (err, data) => {
               if (err) {
                 console.error(err);
                 res.statusCode = 500;
                 res.end();
               } else {
                 res.statusCode = 200;
                 res.end(data.toString().replace('{{dirContent}}', dirContent));
```

```
          }
         });
        }
       });
      });
     }
    }
   } catch (e) {
    res.statusCode = 400;
    res.end();
   }
  }
```

If we try to request a test application with a malformed URL we get the following error http://127.0.0.1:8000/%E0%A4%A.


This page isn't working

If the problem continues, contact the site owner.

HTTP ERROR 400

Picture 13 - Error message

But at this time no error will be generated and an appropriate message will be sent back to the client.

References:

1. https://owasp.org/www-community/attacks/Denial_of_Service
2. https://www.urlencoder.io/learn/
3. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent
4. https://github.com/Shazam2morrow/dva489