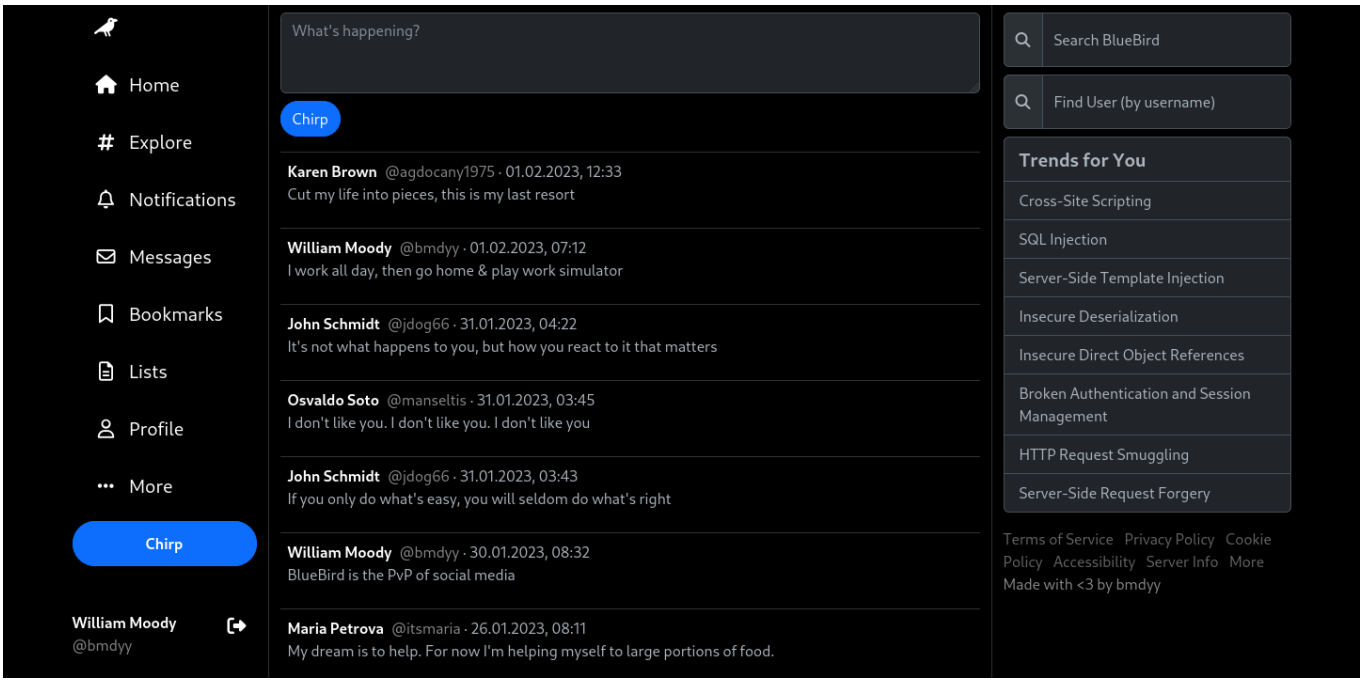
Identifying Vulnerabilities

Decompiling Java Archives

Introduction

Imagine that we were contracted to perform a `white-box` security assessment on a target application named `BlueBird`, a Java Spring Boot web application which uses `PostgreSQL` as its database.



We don't have access to `BlueBird's` source code, but we were given access to the compiled JAR file which, if you aren't familiar with Java, is essentially a Java executable. Let's take a look at two tools we can use to decompile and retrieve `BlueBird's` source code from the `JAR` file so that we can start searching through it for vulnerablities.

Note: It is assumed that you have `Java` installed on your machine. If you don't already have it, head on over to OpenJDK.org and install the latest version.

Testing VM

During this module you are given access to a `testing VM` which has the `BlueBird` JAR file, Java installed, and PostgreSQL installed and initialized.

You may connect via SSH using the username `student` and password `academy.hackthebox.com`.

`BlueBird` application files are located in `/opt/bluebird`, as well as `PostgreSQL` log files (more on that in a later section).

mayala@htb[/htb] `$ /opt/bluebird$ ls -lah total 45M drwxr-xr-x 3 root root 4.0K Feb 28 15:07 . drwxr-xr-x 3 root root 4.0K Feb 28 11:22 .. -rwxr-xr-x 1 root root 45M Feb 28 11:24 BlueBird-0.0.1-SNAPSHOT.jar drwxrwxrwx 2 root root 4.0K Feb 28 15:19 pg_log -rwxr-xr-x 1 root root 319 Feb 28 11:35 serverInfo.sh`

Also in `/opt` is a folder named `Pass2` which contains `Pass2-1.0.3-SNAPSHOT.jar`. You can download this, but ignore it for now, it will be used in the skills assessment.

The `student` user may run the following commands with `sudo`, so that you may restart the services if necessary.

mayala@htb[/htb] `$ sudo -l Matching Defaults entries for student on bb01: env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin User student may run the following commands on bb01: (ALL) NOPASSWD: /usr/bin/systemctl start bluebird (ALL) NOPASSWD: /usr/bin/systemctl stop bluebird (ALL) NOPASSWD: /usr/bin/systemctl start postgresql (ALL) NOPASSWD: /usr/bin/systemctl stop postgresql`

Fernflower

Fernflower is an open-source Java decompiler which is maintained by JetBrains and included in their IntelliJ IDEA IDE. To use this tool, we first need to compile it.

To avoid downloading all the unwanted/extra files in the official repository, we can clone an unofficial mirror of the specific folder containing Fernflower, including:

- github.com/fesh0r/fernflower
- github.com/MinecraftForge/FernFlower

So let's pick one of these and clone it:

mayala@htb[/htb] `$ git clone https://github.com/fesh0r/fernflower.git Cloning into 'fernflower'... remote: Enumerating objects: 12680, done. remote: Counting objects: 100% (2795/2795), done. remote: Compressing objects: 100% (859/859), done. remote: Total 12680 (delta 1435), reused 2541 (delta 1297), pack-reused`

```
9885 Receiving objects: 100% (12680/12680), 6.39 MiB | 1.84 MiB/s, done.
Resolving deltas: 100% (7209/7209), done.
```

Once the repository has been cloned, enter its directory and use Gradle to build `Fernflower`.

mayala@htb[/htb]
```
$ ./gradlew build Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true Welcome to Gradle 7.5.1! Here are the highlights of this release: - Support for Java 18 - Support for building with Groovy 4 - Much more responsive continuous builds - Improved diagnostics for dependency resolution For more details see https://docs.gradle.org/7.5.1/release-notes.html Starting a Gradle Daemon (subsequent builds will be faster) > Task :test Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true BUILD SUCCESSFUL in 20s 4 actionable tasks: 4 executed
```

`Gradle` was able to build `Fernflower` successfully because the `JDK` version on the system matched the one used to develop `Fernflower` (specifically, `JDK 17`), however, when trying to build it on a machine with a non-matching `JDK` version, we will get the following error:

mayala@htb[/htb]
```
$ ./gradlew build Downloading https://services.gradle.org/distributions/gradle-7.5.1-bin.zip <SNIP> Starting a Gradle Daemon (subsequent builds will be faster) > Task :compileJava FAILED FAILURE: Build failed with an exception. * What went wrong: Execution failed for task ':compileJava'. > error: invalid source release: 17 * Try: > Run with --stacktrace option to get the stack trace. > Run with --info or --debug option to get more log output. > Run with --scan to get full insights. * Get more help at https://help.gradle.org BUILD FAILED in 15s 1 actionable task: 1 executed
```

To resolve this issue, we need to use `apt` to install `openjdk-17-jdk`:

mayala@htb[/htb]
```
$ sudo apt install openjdk-17-jdk Reading package lists... Done Building dependency tree... Done Reading state information... Done The following packages were automatically installed and are no longer required: libgit2-1.1 libmbedcrypto3 libmbedtls12 libmbedx509-0 libstd-rust-1.48 libstd-rust-dev linux-kbuild-5.18 rust-gdb Use 'sudo apt autoremove' to remove them. The following additional packages will be installed: openjdk-17-jdk-headless openjdk-17-jre openjdk-17-jre-headless Suggested packages: openjdk-17-demo openjdk-17-source visualvm fonts-ipafont-gothic fonts-ipafont-mincho fonts-wqy-microhei | fonts-wqy-zenhei fonts-indic The following NEW packages will be installed: openjdk-17-jdk openjdk-17-jdk-headless The following packages will be upgraded: openjdk-17-jre openjdk-17-jre-headless 2 upgraded, 2 newly installed, 0 to remove and 106 not upgraded. Need to get 278 MB of archives. After this
```

`operation, 244 MB of additional disk space will be used. Do you want to continue? [Y/n] Y <SNIP>`

Afterward, we need to set it as the default `JDK` for our system, to do so, first we need to know the path to `JDK 17` using `update-java-alternative` along with the `--list` flag:

mayala@htb[/htb] `$ sudo update-java-alternatives --list java-1.11.0-openjdk-amd64 1111 /usr/lib/jvm/java-1.11.0-openjdk-amd64 java-1.13.0-openjdk-amd64 1311 /usr/lib/jvm/java-1.13.0-openjdk-amd64 java-1.17.0-openjdk-amd64 1711 /usr/lib/jvm/java-1.17.0-openjdk-amd64`

The path is `/usr/lib/jvm/java-1.17.0-openjdk-amd64`, thus, we now need to set it as the default with `update-java-alternative` along with the `--set` flag:

mayala@htb[/htb] `$ sudo update-java-alternatives --set /usr/lib/jvm/java-1.17.0-openjdk-amd64`

Trying to build `Fernflower` again, we will notice that the error disappears as the issue got resolved.

Once `Gradle` is done, `Fernflower` should have been compiled into a `JAR` file located at `build/libs/fernflower.jar`. We can use this to decompile `BlueBird` like this (make sure `out` is a real folder):

mayala@htb[/htb] `$ java -jar fernflower.jar BlueBird-0.0.1-SNAPSHOT.jar out Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true INFO: Decompiling class org/springframework/boot/loader/ClassPathIndexFile INFO: ... done INFO: Decompiling class org/springframework/boot/loader/ExecutableArchiveLauncher <SNIP> INFO: Decompiling class com/bmdyy/bluebird/model/Post INFO: ... done INFO: Decompiling class com/bmdyy/bluebird/model/User INFO: ... done`

Once `Fernflower` is done, we can enter `out` and there should be a single `JAR` file containg a bunch of source `.java` files. We can use the following command to extract them all:

mayala@htb[/htb] `$ jar -xf BlueBird-0.0.1-SNAPSHOT.jar Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true`

At this point, we should have the source `.java` files inside the `BOOT-INF/classes` directory.

mayala@htb[/htb] `$ tree . ├── BlueBird-0.0.1-SNAPSHOT.jar ├── BOOT-INF │ ├── classes │ │ ├── application.properties │ │ ├── com │ │ │ └── bmdyy │ │ │ └── bluebird │ │ │ ├── BlueBirdApplication.java │ │ │ ├── controller │ │ │ │ ├── AuthController.java │ │ │ │ ├── IndexController.java │ │ │ │ ├──`

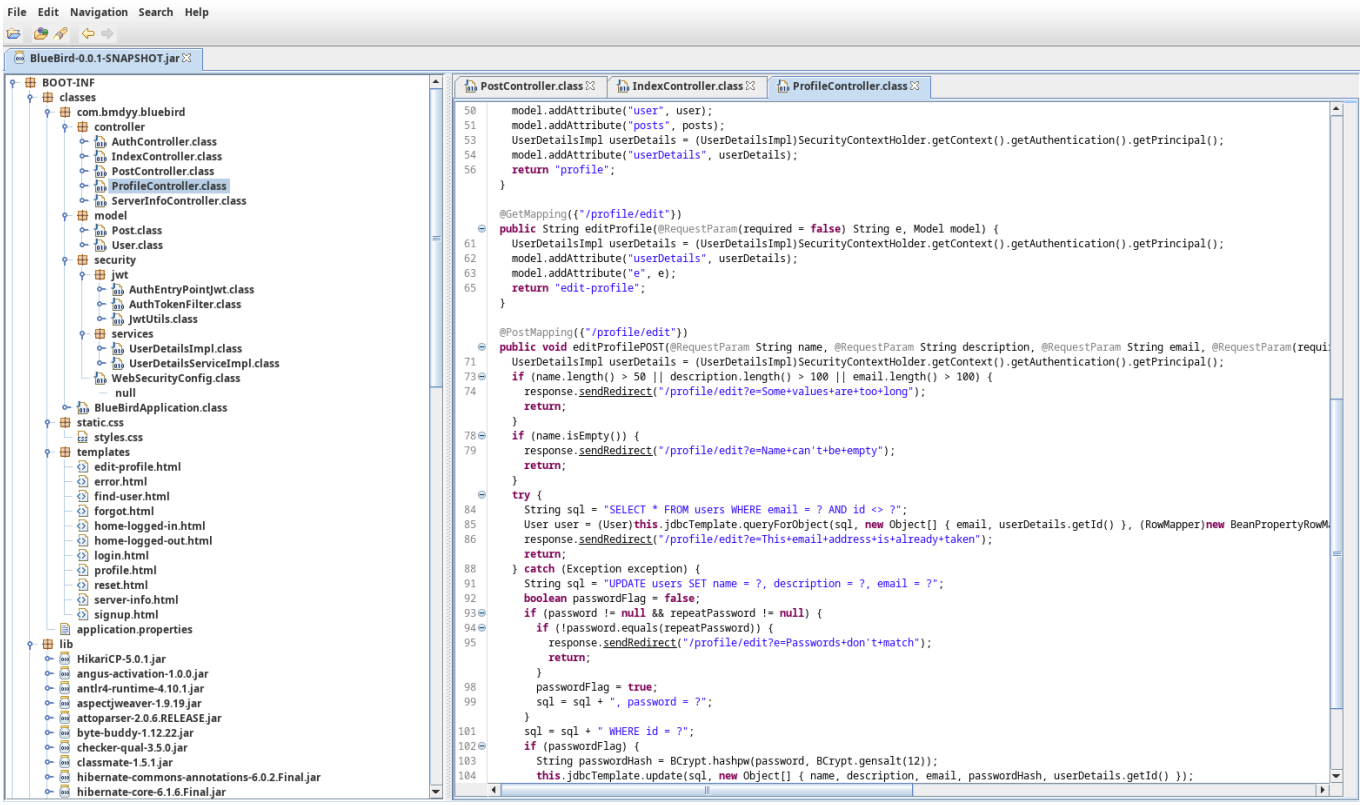
```
PostController.java │ │ │ │ ├── ProfileController.java │ │ │ │ └── ServerInfoController.java │ │ │ ├── model │ │ │ │ ├── Post.java │ │ │ │ └── User.java │ │ │ └── security │ │ │ ├── jwt │ │ │ │ ├── AuthEntryPointJwt.java │ │ │ │ ├── AuthTokenFilter.java │ │ │ │ └── JwtUtils.java │ │ │ ├── services │ │ │ │ ├── UserDetailsImpl.java │ │ │ │ └── UserDetailsServiceImpl.java │ │ │ └── WebSecurityConfig.java │ │ ├── static │ │ │ └── css │ │ │ └── styles.css │ │ └── templates │ │ ├── edit-profile.html │ │ ├── error.html │ │ ├── find-user.html │ │ ├── forgot.html │ │ ├── home-logged-in.html │ │ ├── home-logged-out.html │ │ ├── login.html │ │ ├── profile.html │ │ ├── reset.html │ │ ├── server-info.html │ │ └── signup.html │ ├── classpath.idx │ ├── layers.idx │ └── lib <SNIP> 25 directories, 136 files
```

JD-GUI

Another open-source tool we can use to decompile `JAR` files is JD-GUI. As the name suggests, this one has a graphic interface which we can use to view the decompiled files. It works well, however the last release was in 2019 and the project might even be discontinued.

We can download the latest release `JAR` file from here and run it like so:

mayala@htb[/htb] `$ java -jar jd-gui-1.6.6.jar BlueBird-0.0.1-SNAPSHOT.jar Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true`



We can use the UI to view the `.java` source files and even search for strings, variables or methods. Alternatively, Visual Studio Code can be used to look through the source code. You can save the source files by hitting `File > Save All Sources` and then unzipping the created `ZIP` archive.

Searching for Strings

RegEx

Now that we have the decompiled source files for `BlueBird` we can start searching for vulnerabilities; in this case we are specifically interested in `SQL injection` vulnerabilities.

In most cases, SQL queries are simply `strings` that are passed to a database to be processed. In the case where we want to identify `SQL injection` vulnerabilities, it is necessary to analyze the `SQL queries` being used in the program to see if any are vulnerable. Rather than manually scrolling through lines upon lines of code, we can use `Regular Expressions` (`RegEx`) to significantly speed up our efforts.

In the table below are a few `RegEx` patterns that we can use.

Query	Description
`SELECT|UPDATE|DELETE|INSERT|CREATE|ALTER|DROP`	Search for the basic SQL commands. Injection can occu more than just SELECT statements, exploitation may ju be a bit trickier.
`(WHERE|VALUES).*?'`	Search for strings which include `WHERE` or `VALUES` and then a `single quote`, which could indicate a string concatenation.
`(WHERE|VALUES).*" \+`	Search for strings which include `WHERE` or `VALUES` follc by a double quote and a plus s which could indicate a string concatenation.
`.*sql.*"`	Search for lines which include `sql` and then a `doubl quote`.
`jdbcTemplate`	Search for lines which include `jdbcTemplate`. There various ways to interact

Query	Description
	with `SQL` databases in `Java`. `JdbcTemplate` is one them; others include `JPA` and `Hibernate`.

When analyzing source code, it is useful to take note of the `libraries used` as well as the `coding style` so you can adapt your search queries to be more effective. For example, as we look through the results in `BlueBird`, we will notice that the developer always stores `SQL` queries in a variable named `sql`, so that's something we could look for specifically.

Grep

One way we can use these `RegEx` patterns against the decompiled source files is with grep, which is a `command-line` tool used to search for `patterns` in `files`.

The syntax to use `RegEx` with `grep` is `grep -E <RegEx> <File>`, but we can set a few more arguments to improve our results, such as `--include *.java` to only search for matches in `.java` files, using `-n` to display line numbers, `-i` to ignore case, and `-r` to search recursively through a directory.

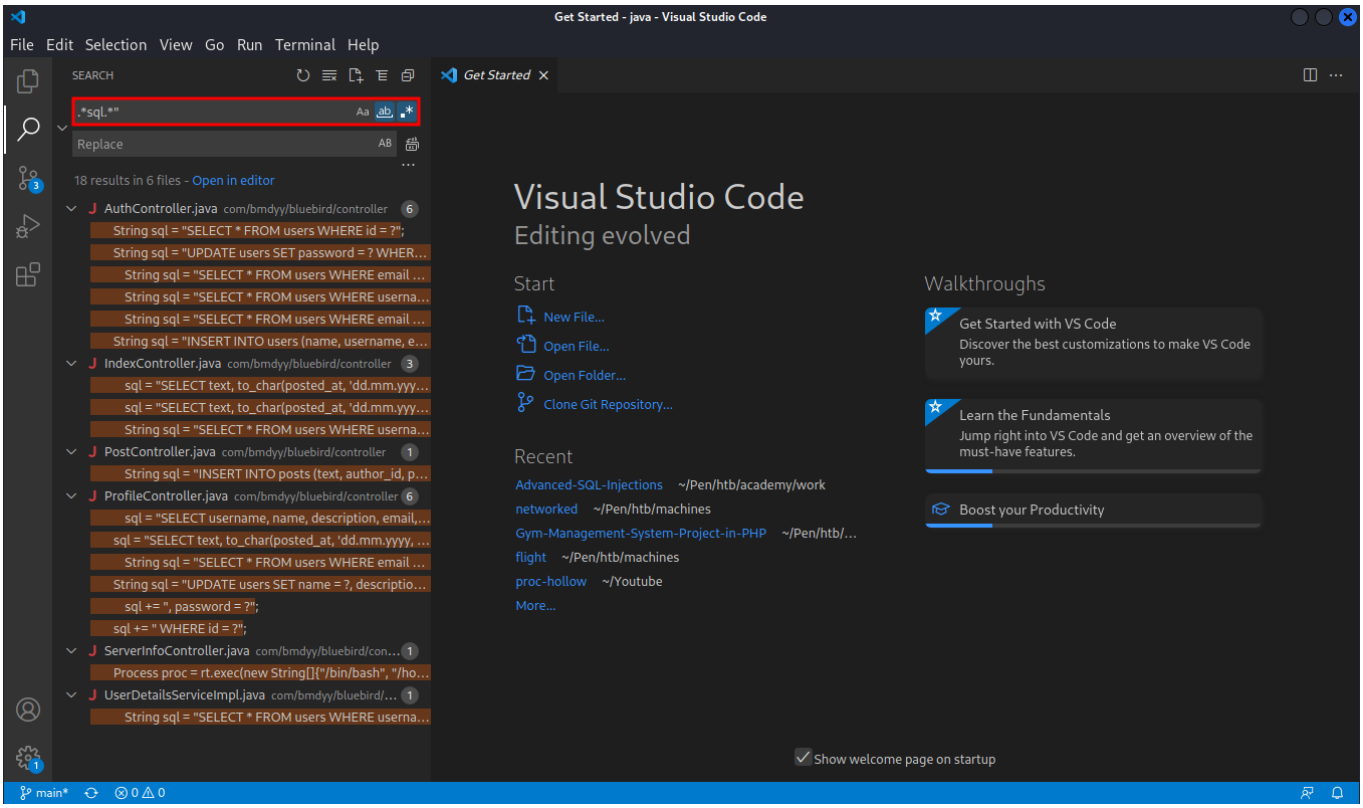
As a result, the command we want to use will look something like this:

mayala@htb[/htb] `$ grep -irnE 'SELECT|UPDATE|DELETE|INSERT|CREATE|ALTER|DROP' . ./com/bmdyy/bluebird/controller/PostController.java:21: public void createPost(@RequestParam String text, HttpServletResponse response) throws IOException { ./com/bmdyy/bluebird/controller/PostController.java:25: String sql = "INSERT INTO posts (text, author_id, posted_at) VALUES (?, ?, CURRENT_TIMESTAMP);"; ./com/bmdyy/bluebird/controller/PostController.java:26: jdbcTemplate.update(sql, text, userDetails.getId()); ./com/bmdyy/bluebird/controller/AuthController.java:78: String sql = "SELECT * FROM users WHERE id = ?"; <SNIP> ./com/bmdyy/bluebird/controller/ProfileController.java:109: response.sendRedirect("/profile/edit?e=Details+updated!"); ./com/bmdyy/bluebird/security/services/UserDetailsServiceImpl.java:21: String sql = "SELECT * FROM users WHERE username = ?";`

Visual Studio Code

Another more visual way to use these `RegEx` patterns is Visual Studio Code. We can use the `Search` feature by clicking on the `magnifying glass` on the left-hand side, entering

the `RegEx` pattern, and then clicking the right-most button to enable `RegEx` search.

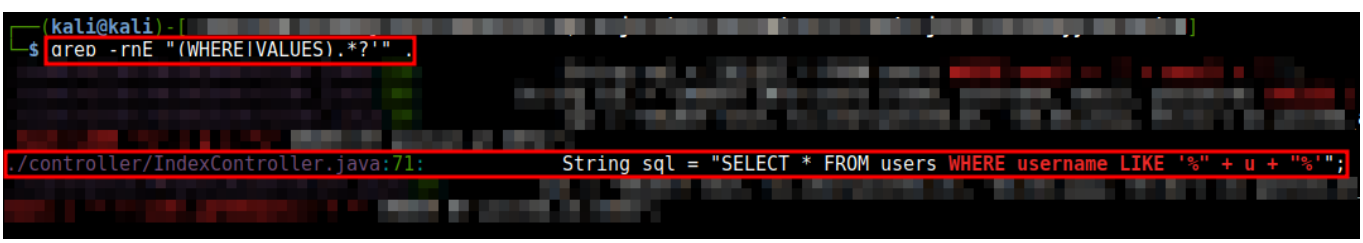


Results

Using either one of these methods, we can quickly identify the functions that use SQL queries. We can then go through them to try and identify ones that lack input sanitization.

Identifying the SQL Injection in /find-user

Using `grep` we can identify the following string concatenation in `IndexController.java`:



Taking a closer look at the code, we can see the function `findUser` which is mapped to `GET` requests to `/find-user`. This function takes a request parameter `u`. If this parameter matches the `RegEx` pattern contained in the variable `p` or contains a `space`, then the error message `"Illegal Search term"` is returned, otherwise `u` is used in a `SQL` query that

populates the `users` list, which is then used as a `model attribute` when rendering `find-user`.

```java
    }

    @GetMapping("/find-user")
    public String findUser(@RequestParam String u, Model model, HttpServletResponse response) throws IOException {
        Pattern p = Pattern.compile("'|(.*'.*'.*)");
        Matcher m = p.matcher(u);

        String u2 = u.toLowerCase();
        if (u2.contains(" ") || m.matches()) {
            model.addAttribute("errorMsg", "Illegal search term");
            return "error";
        }
        try {
            String sql = "SELECT * FROM users WHERE username LIKE '%" + u + "%'";
            List<User> users = jdbcTemplate.query(sql, new BeanPropertyRowMapper(User.class));

            UserDetailsImpl userDetails = (UserDetailsImpl) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
            model.addAttribute("userDetails", userDetails);
            model.addAttribute("users", users);

            return "find-user";
        } catch (BadSqlGrammarException e) {
            System.out.println(e.getSQLException().getMessage());
            model.addAttribute("errorMsg", "Invalid search query");
            return "error";
        } catch (Exception e) {
            e.printStackTrace();
            model.addAttribute("errorMsg", "Invalid search query");
            return "error";
        }
    }
}
```

Looking at `resources/templates/find-user.html` we can see what happens with the `users` attribute; Thymeleaf loops through the list and prints out the `Id`, `Name`, `Username` and `Description` values of the `user` object.

```
find-user.html ×
resources > templates > find-user.html > html > body > div.container > div.row > div.col-3 > form.mt-2 > div.input-group
54
55
56
57
58
59
60
61
62
63
64      <h1>User search results...</h1>
65      <th:block th:each="user: ${users}">
66        <div class="mt-3 pt-2" style="border-top: 1px solid #333">
67          <a style="text-decoration:none; color: white" th:href="${'/profile/' + user.getId()}">
68            <span style="color: white; text-decoration:underline; font-weight: bold" th:text="${user.getName()}"/>
69            <br>
70            <span class="handle" th:text="${'@' + user.getUsername()}"/>
71            <span th:text="${user.getDescription()}"/>
72          </a>
73        </div>
74      </th:block>
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
```

We will want to look into this later, as this is a clear SQL injection vulnerability.

Identifying the SQL Injection in /forgot

Using grep again, we come across the following line in `AuthController.java` where user-input is concatenated into a `SQL` query.

mayala@htb[/htb] `$ grep -nrE '(WHERE|VALUES).*" +' . ./controller/AuthController.java:134: String sql = "SELECT * FROM users WHERE email = '" + email + "'"; <SNIP>`

Opening up `AuthController.java`, we can see that this line happens in the `forgotPOST()` function which handles `POST` requests to `/forgot` and the `email` variable is user-input (`@RequestParam String email`) that is validated against a `RegEx` pattern before being used in the query.

Code: java

```
// AuthController.java (Lines 121-164)

@PostMapping({"/forgot"})
```

```
public String forgotPOST(@RequestParam String email, Model model,
HttpServletRequest request, HttpServletResponse response) throws IOException
{
    if (email.isEmpty()) {
        response.sendRedirect("/forgot?e=Please+fill+out+all+fields");
        return null;
    } else {
        Pattern p = Pattern.compile("^.*@[A-Za-z]*\\.[A-Za-z]*$");
        Matcher m = p.matcher(email);
        if (!m.matches()) {
        response.sendRedirect("/forgot?e=Invalid+email!");
        return null;
        } else {
        try {
            String sql = "SELECT * FROM users WHERE email = '" + email +
"'";
            User user = (User)this.jdbcTemplate.queryForObject(sql, new
BeanPropertyRowMapper(User.class));
            Long var10000 = user.getId();
            String passwordResetHash = DigestUtils.md5DigestAsHex(("" +
var10000 + ":" + user.getEmail() + ":" + user.getPassword()).getBytes());
            var10000 = user.getId();
            String passwordResetLink = "https://bluebird.htb/reset?uid=" +
var10000 + "&code=" + passwordResetHash;
            logger.error("TODO- Send email with link [" + passwordResetLink
+ "]");
            response.sendRedirect("/forgot?
e=Please+check+your+email+for+the+password+reset+link");
            return null;
        } catch (EmptyResultDataAccessException var11) {
            response.sendRedirect("/forgot?e=Email+does+not+exist");
            return null;
        } catch (Exception var12) {
            String ipAddress = request.getHeader("X-FORWARDED-FOR");
            if (ipAddress == null) {
                ipAddress = request.getRemoteAddr();
            }

            if (ipAddress.equals("127.0.1.1")) {
                model.addAttribute("errorMsg", var12.getMessage());
                model.addAttribute("errorStackTrace",
Arrays.toString(var12.getStackTrace()));
            } else {
                model.addAttribute("errorMsg", "500 Internal Server Error");
                model.addAttribute("errorStackTrace", "Something happened on
our side. Please try again later.");
```

```
            }

            return "error";
        }
        }
    }
}
```

Note: If you don't know what `controllers` are, you can imagine them as API endpoints.

In the case of SQL errors, `Spring` writes error messages to `STDOUT`, but in this case we can see explicit exception handling was defined. In most cases we get a typical `500 Internal Server Error` response from the server, but if our client IP address matches `127.0.1.1` then it looks like a stacktrace is passed to the `Thymeleaf` template.

This is something we will want to look into later, as SQL error output can be very useful.

Identifying the SQL injection in /profile

Using another one of the `RegEx` patterns with `grep`, we come across the following `SQL` query. In this case `user.getEmail()` is used in the query unsafely, but we have to take a closer look to see what the value this function returns is and more importantly whether we can manipulate it or not.

mayala@htb[/htb] `$ grep -nrE '.*sql.*"' . <SNIP> ./BOOT-INF/classes/com/bmdyy/bluebird/controller/ProfileController.java:40: sql = "SELECT text, to_char(posted_at, 'dd.mm.yyyy, hh:mi') as posted_at_nice, username, name, author_id FROM posts JOIN users ON posts.author_id = users.id WHERE email = '" + user.getEmail() + "' ORDER BY posted_at DESC"; <SNIP>`

Looking inside `ProfileController.java`, we find that this line is within the `profile()` function which is mapped to `GET` requests to `/profile/{id}`.

Code: java

```
// ProfileController.java (Lines 28-47)

@GetMapping({"/profile/{id}"})
public String profile(@PathVariable int id, Model model, HttpServletResponse
response) throws IOException {
    String sql;
    User user;
    try {
```

```
        sql = "SELECT username, name, description, email, id FROM users
WHERE id = ?";
        user = (User)this.jdbcTemplate.queryForObject(sql, new Object[]{id},
new BeanPropertyRowMapper(User.class));
    } catch (Exception var8) {
        response.sendRedirect("/");
        return null;
    }

    sql = "SELECT text, to_char(posted_at, 'dd.mm.yyyy, hh:mi') as
posted_at_nice, username, name, author_id FROM posts JOIN users ON
posts.author_id = users.id WHERE email = '" + user.getEmail() + "' ORDER BY
posted_at DESC";
    List posts = this.jdbcTemplate.queryForList(sql);
    model.addAttribute("user", user);
    model.addAttribute("posts", posts);
    UserDetailsImpl userDetails =
(UserDetailsImpl)SecurityContextHolder.getContext().getAuthentication().getP
rincipal();
    model.addAttribute("userDetails", userDetails);
    return "profile";
}
```

A quick glance over this code tells us that the `User` object referenced in the vulnerable SQL query is initialized just above with the results from another query. This is good news for us if we can find a way to influence those results, so we'll take a closer look at this later.

Live-Debugging Java Applications

Introduction

Since we have the `JAR` file, we can use Visual Studio Code or Eclipse IDE to remotely debug the application and see how our input is handled in real-time.

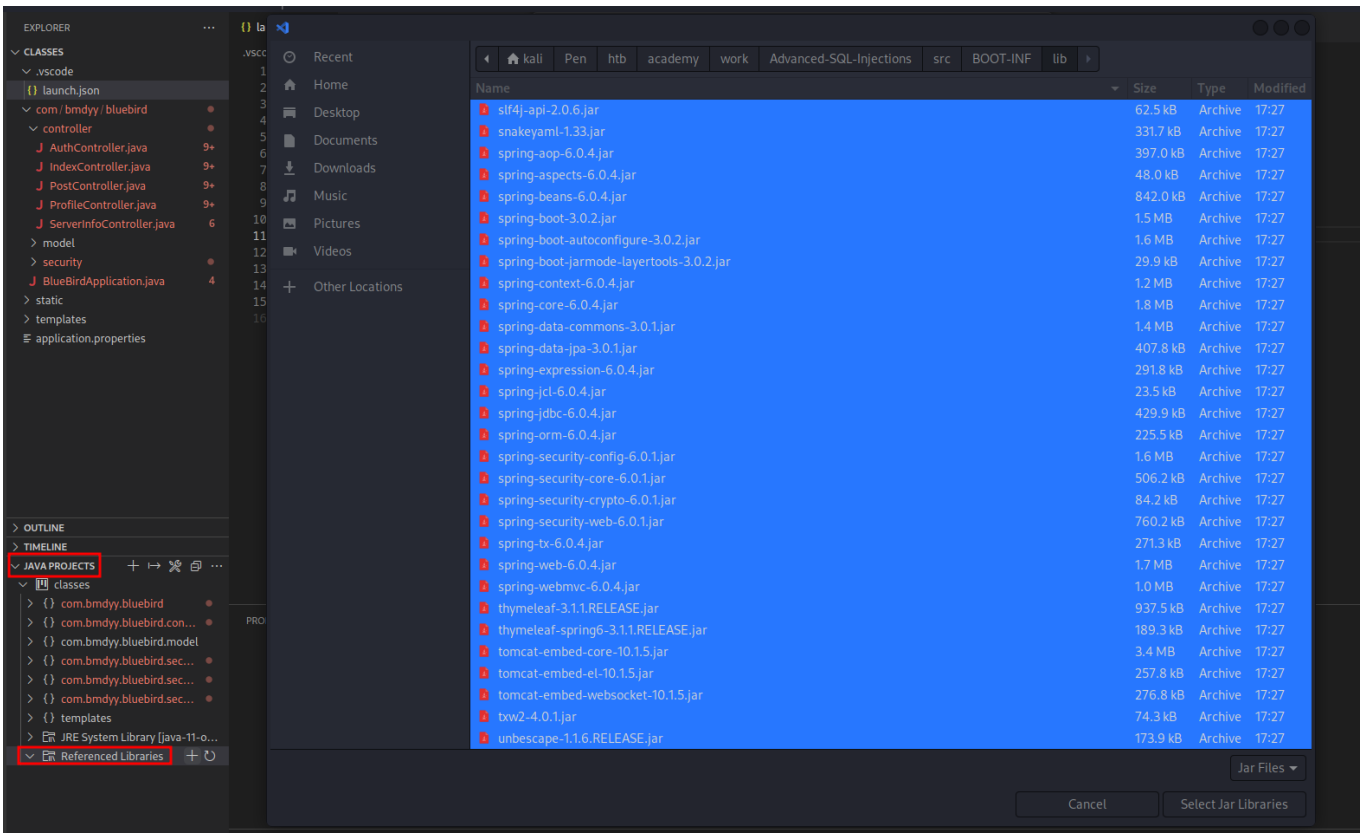
Remote Debugging with Visual Studio Code

The first thing that we want to do, is install the Extension Pack for Java for `VSCode`.

Once that's been installed, we're going to decompile `BlueBird` (if you haven't already) using `Fernflower`. As a reminder, this is what the commands look like:

mayala@htb[/htb] `$ mkdir src $ java -jar fernflower.jar BlueBird-0.0.1-SNAPSHOT.jar src $ cd src $ jar -xf BlueBird-0.0.1-SNAPSHOT.jar`

At this point, we can launch `VSCode` and open the folder `src/BOOT-INF/classes`. We should have all the source files open, but a lot of lines will be underlined in red due to unresolved imports. We can fix this by navigating to `Java Projects > Referenced Libraries` on the lefthand sidebar, clicking the `+` icon and selecting all the `JAR` files from the decompiled `src/BOOT-INF/libs` folder. After this is done, the errors should disappear.



Now, you we want to hit `[CTRL]+[SHIFT]+[D]` to bring up the debug pane, and `create a launch.json file` with the following contents:

Code: json

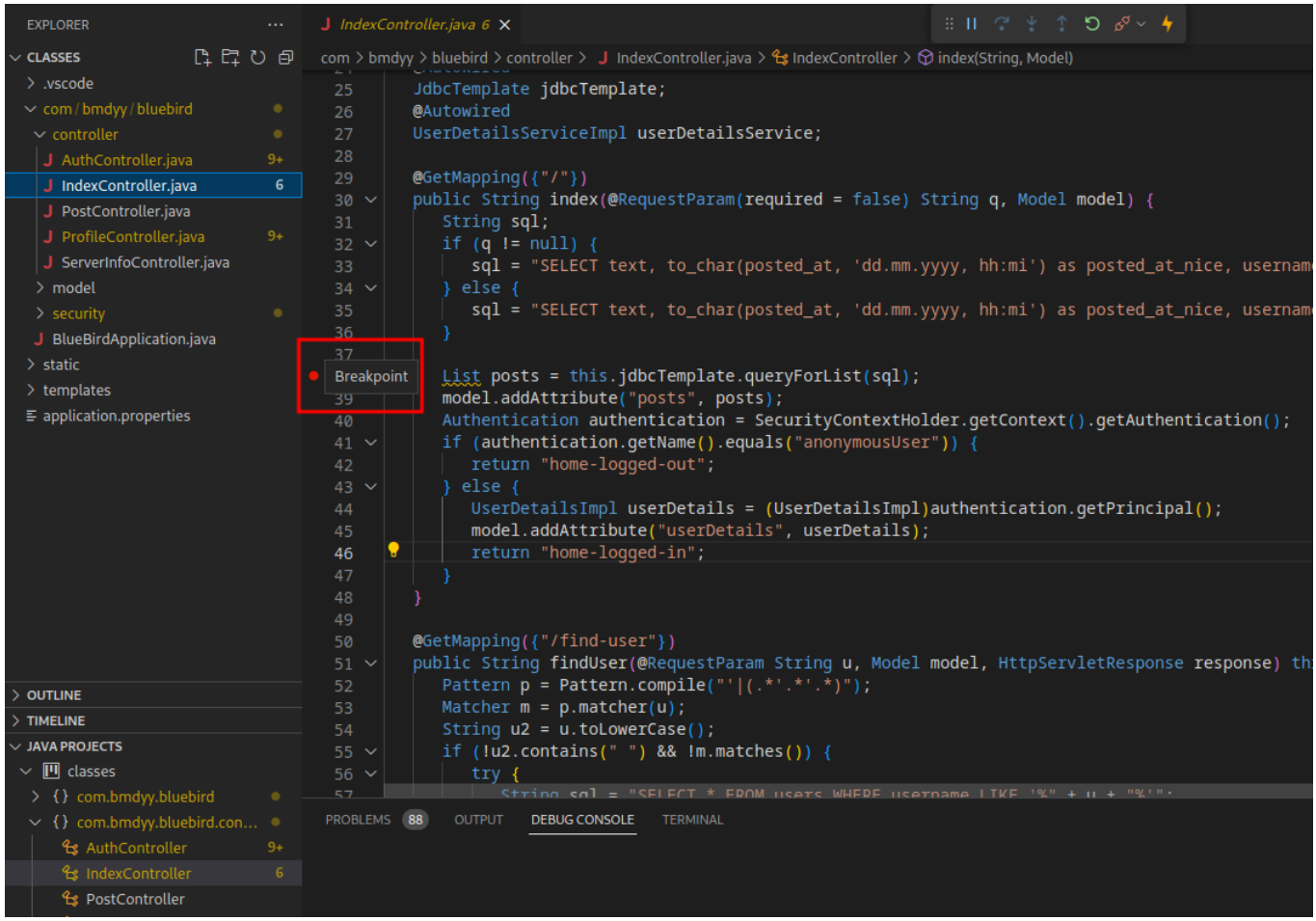
```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Remote Debugging",
            "request": "attach",
            "hostName": "127.0.0.1",
            "port": 8000
        }
```

```
    ]
}
```

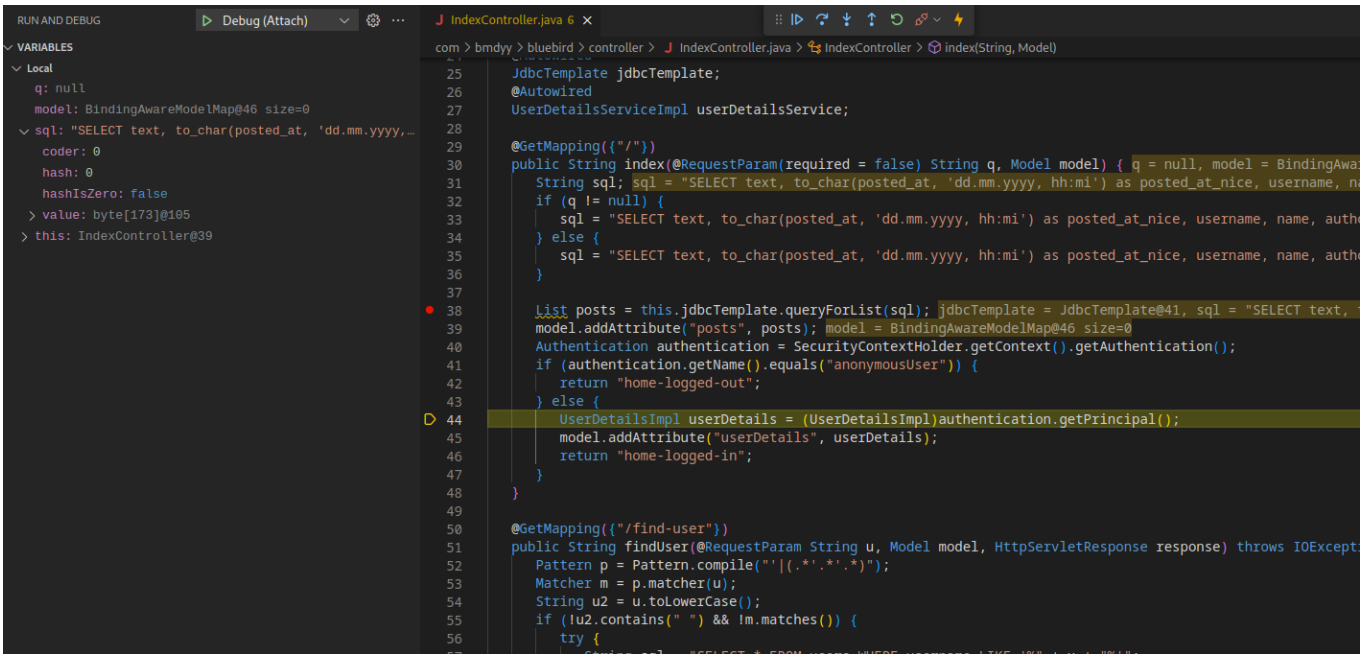
With that all prepared, connect to the VM through SSH with this command which will forward port `8000`, and then run the second command to launch `BlueBird` in remote debugging mode.

mayala@htb[/htb] `$ ssh -L 8000:127.0.0.1:8000 student@x.x.x.x $ java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=y -jar BlueBird-0.0.1-SNAPSHOT.jar`

Finally, we can go back to `VSCode` and hit `[F5]` to start debugging. You can set a breakpoint by left-clicking to the left of a line number like this:

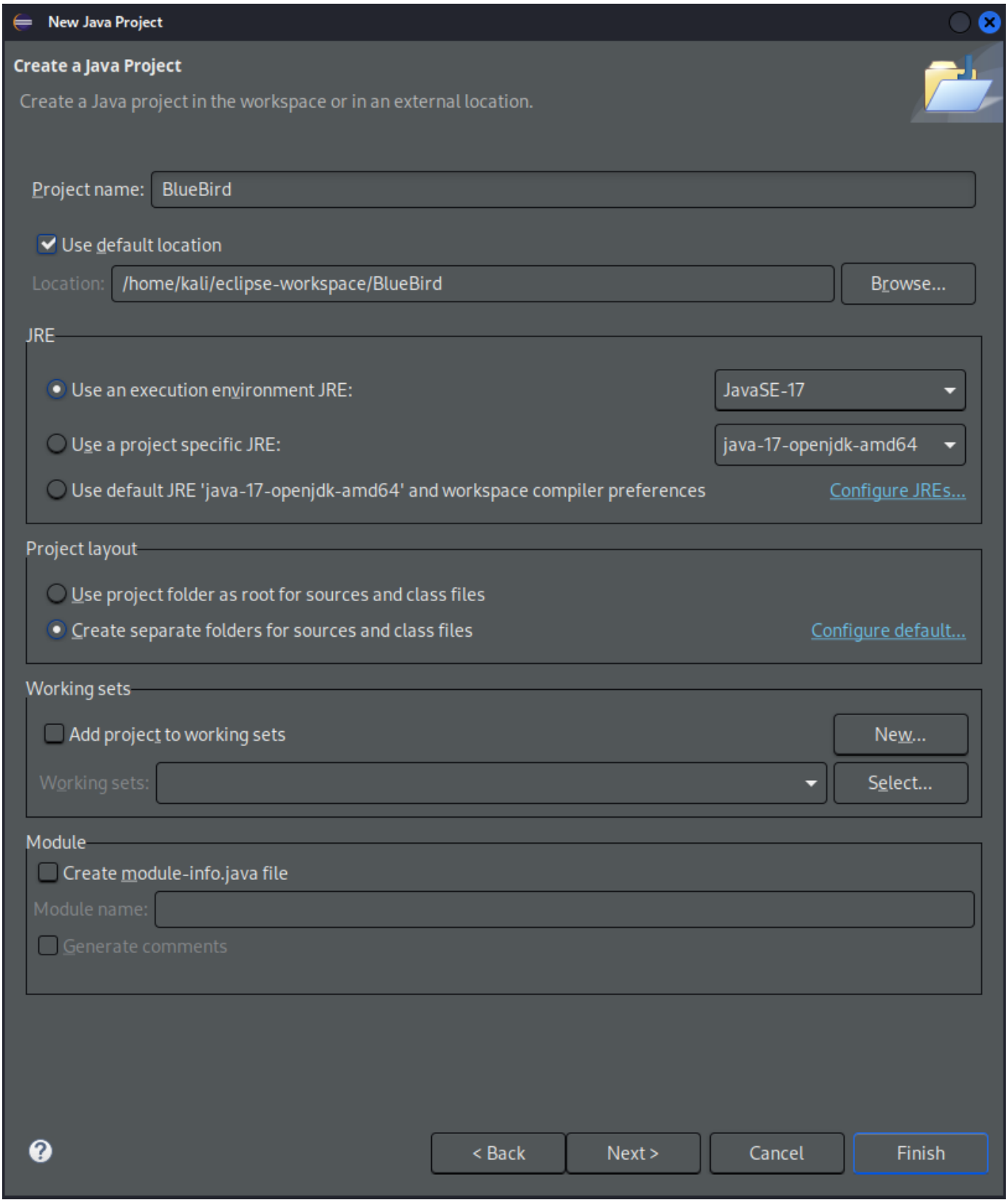


When lines with breakpoints are hit, execution will pause so that you can inspect variable values and control the program's flow by stepping through the lines of code.



Remote Debugging with Eclipse

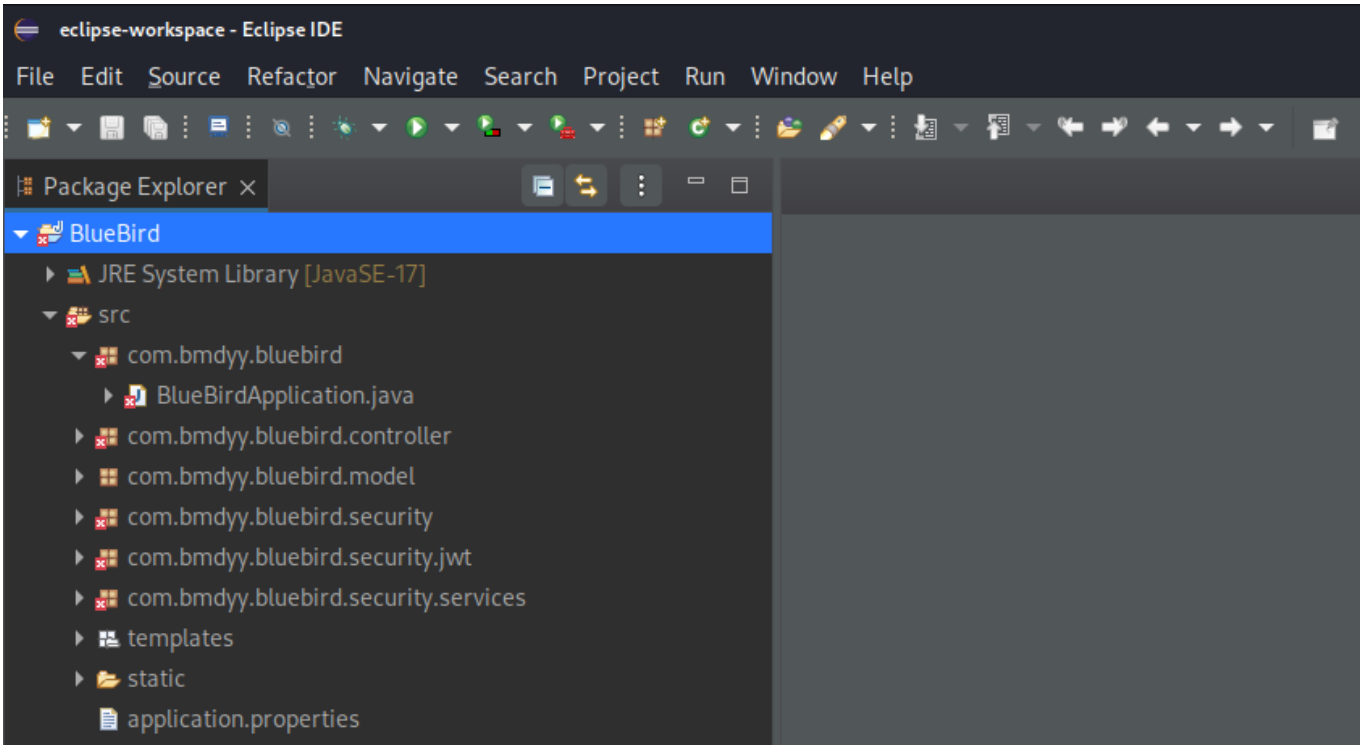
Perhaps you're a fan of `Eclipse`. That's alright, the process is quite similar in this case. Go ahead and create a `new Java Project` with the following settings:



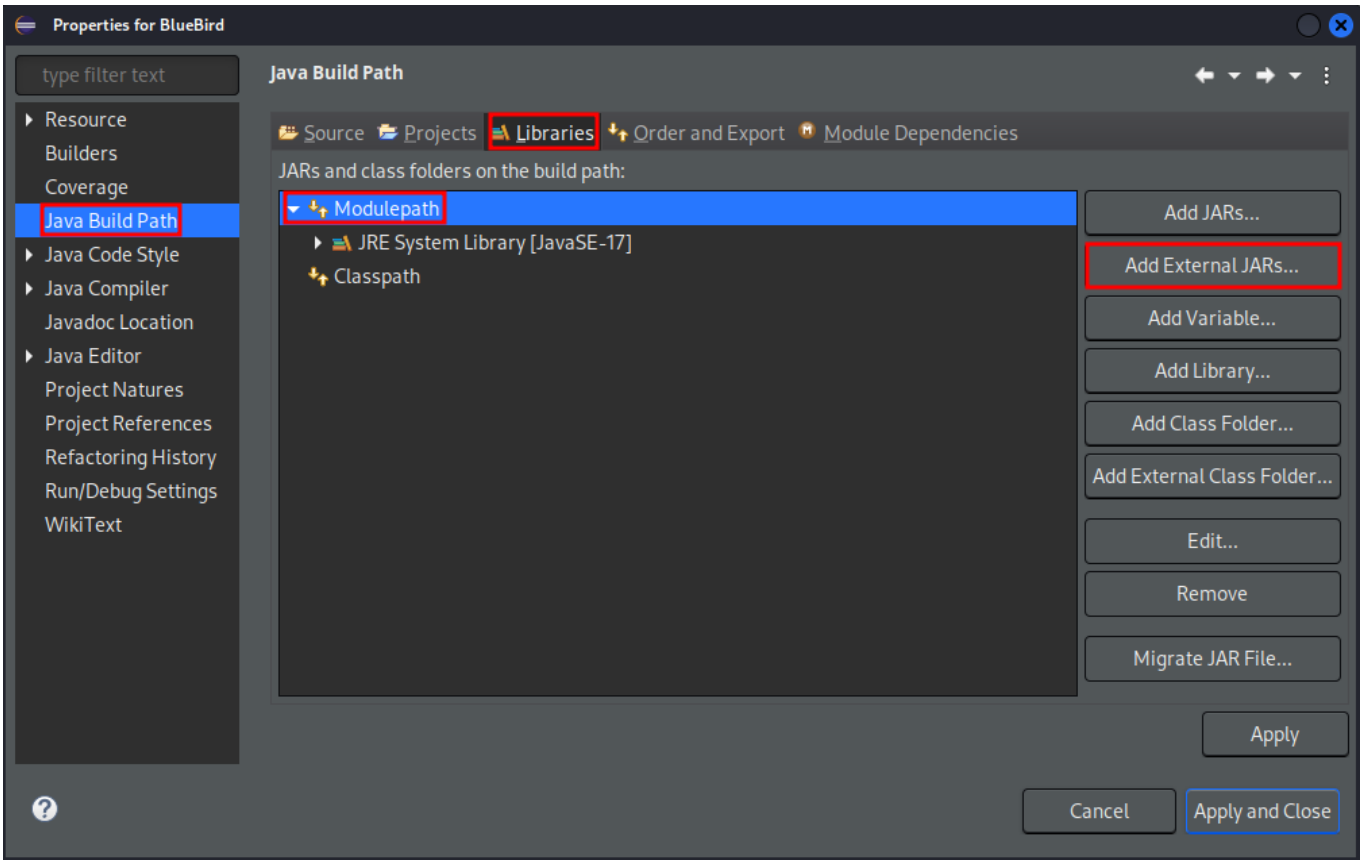
We are going to import the "source" of `BlueBird` into the `Eclipse` project, so if you haven't already, decompile `BlueBird-0.0.1-SNAPSHOT.jar` using `Fernflower` as described in the `Decompiling Java Archives` section. Once that's ready, go ahead and copy the contents of the decompiled `classes/` folder into the `src/` folder for the `Eclipse` project we just made.

mayala@htb[/htb] `$ cp -r src/BOOT-INF/classes/* ~/eclipse-workspace/BlueBird/src`

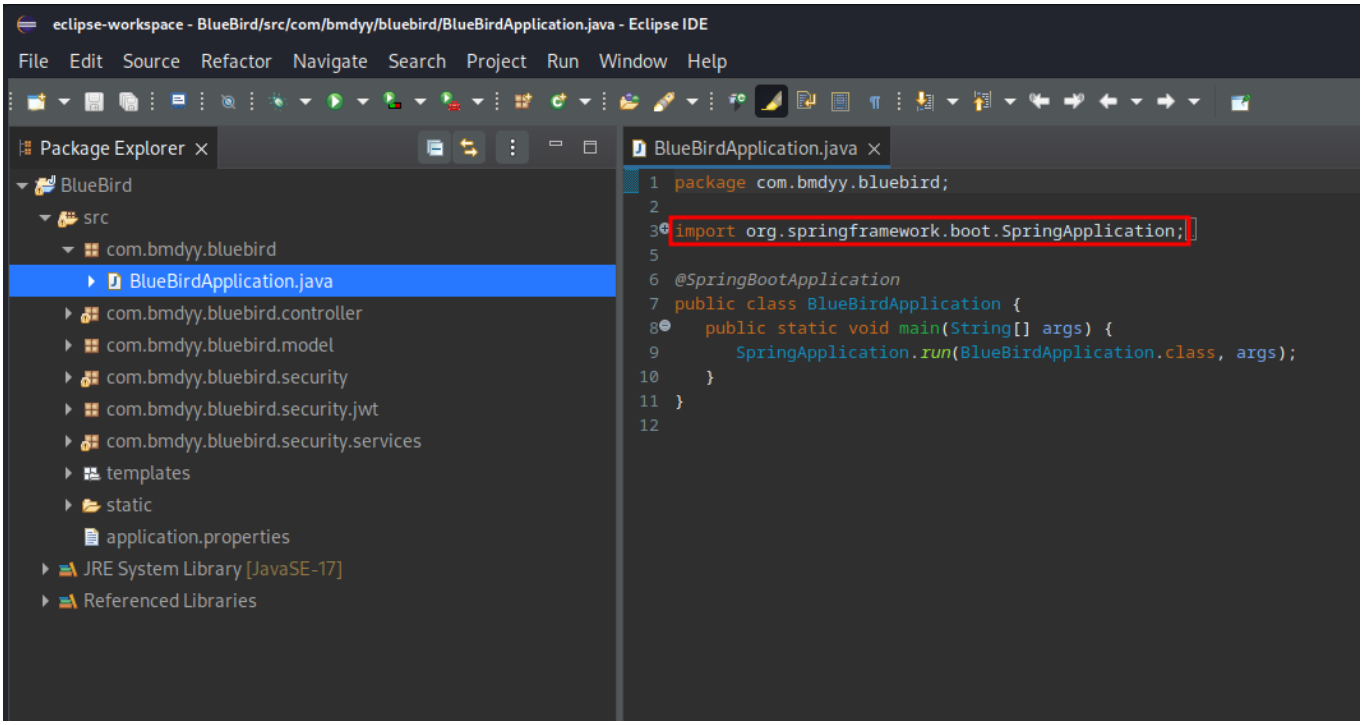
If you did that correctly, right-clicking in the `Package Explorer` and hitting `Refresh` should result in all the packages showing up (with red errors).



The reason the packages have errors is due to missing imports. To resolve this issue, we will import all the dependencies from the decompiled JAR. Go to `File > Properties > Java Build Path > Libraries > Modulepath > Add External JARs` and add all the JAR files from `lib/` (created by `Fernflower` when decompiling). Click `Apply and Close` once imported.



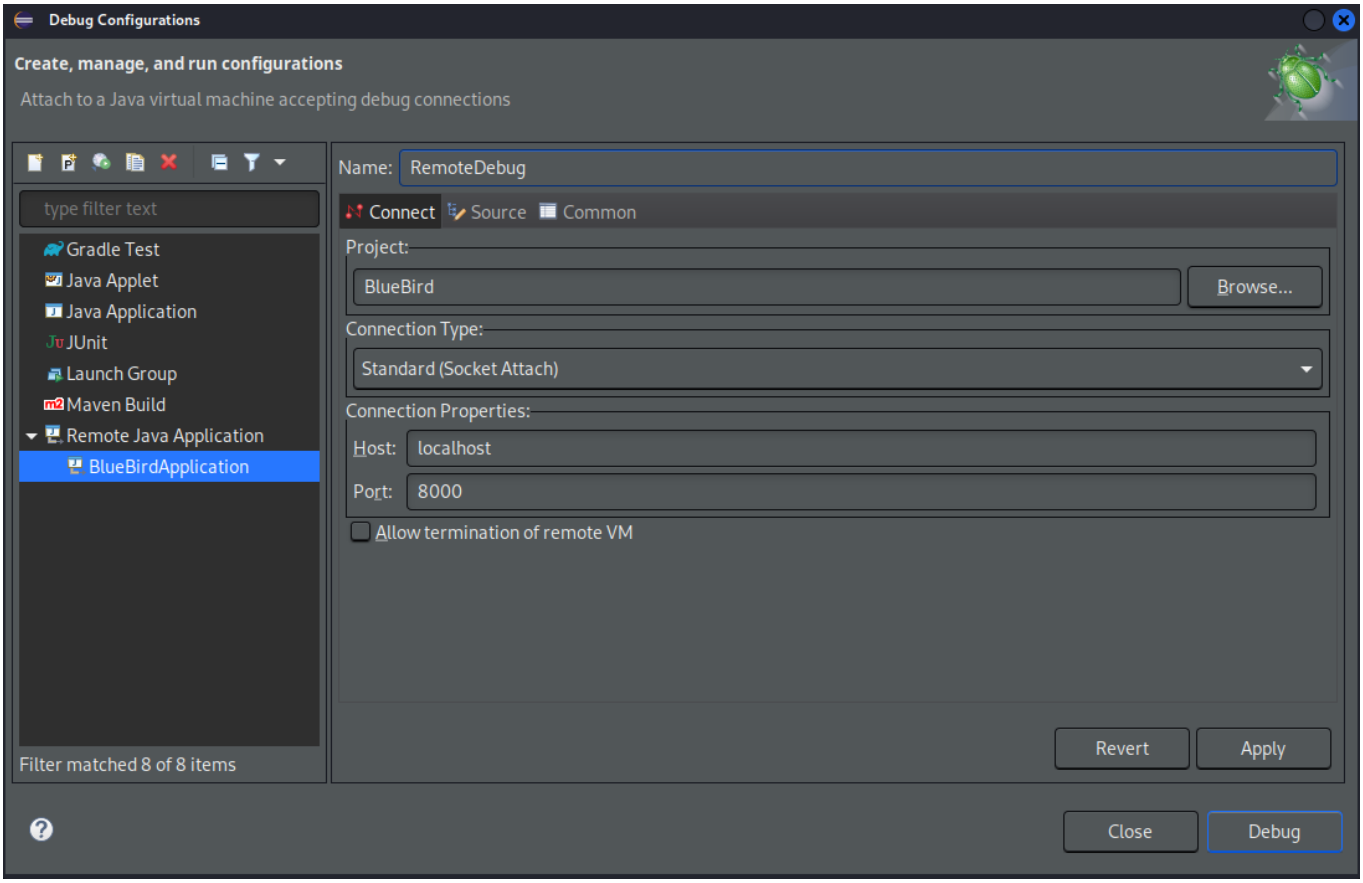
If you did this step correctly, there should be no more red error signs or underlines on import statements as show in the screenshot below.



At this point we can open up a terminal and run the following command to start the JAR file in `remote debugging` mode.

mayala@htb[/htb] `$ java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=y -jar BlueBird-0.0.1-SNAPSHOT.jar Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true Listening for transport dt_socket at address: 8000`

To attach to this, we need to head back to `Eclipse`, go to `Run > Debug Configurations` and create a new `Remote Java Application` with the following settings (should be default):



Click `Apply` and then `Debug`. If you look at the console, you should see the Spring startup log messages.

```
┌──(kali㉿kali)-[~/htb/htb/BlueBird/target]                                  [25/51]
└─$ java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=y -jar BlueBird-0.0.1-SNAPSHOT.jar
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Listening for transport dt_socket at address: 8000

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.2)

2023-02-20T19:01:43.625+01:00  INFO 40250 --- [           main] com.bmdyy.bluebird.BlueBirdApplication   : Starting BlueBirdApplication using Java 17.0.6 with PID 40250 (/home/kali/htb/htb/BlueBird/target/BlueBird-0.0.1-SNAPSHOT.jar started by kali in /home/kali/htb/htb/BlueBird/target)
2023-02-20T19:01:43.629+01:00  INFO 40250 --- [           main] com.bmdyy.bluebird.BlueBirdApplication   : No active profile set, falling back to 1 default profile: "default"
2023-02-20T19:01:44.472+01:00  INFO 40250 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2023-02-20T19:01:44.506+01:00  INFO 40250 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 23 ms. Found 0 JPA repository interfac
es.
2023-02-20T19:01:45.338+01:00  INFO 40250 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-02-20T19:01:45.359+01:00  INFO 40250 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-02-20T19:01:45.359+01:00  INFO 40250 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.5]
```

Inside `Eclipse` click `Window > Perspective > Open Perspective > Debug` to show the debugging windows. At this point we can place breakpoints in the project and live-debug `BlueBird`. To to so, right-click on the line number and select `Toggle Breakpoint`.

When this line will be reached, the application will freeze and we can step through execution line by line to see what happens exactly. You can see the values of variables as they change in the `Variables` window (just make sure the `Debug Perspective` is open).

```
BlueBirdApplication.java   IndexController.java   AuthController.java ×
116         response.sendRedirect("/reset?uid=" + uid + "&code=" + code + "&e=Please+fill+out+all+fields");
117     }
118 }
119
120 @PostMapping({"/forgot"})
121 public String forgotPOST(@RequestParam String email, Model model, HttpServletResponse response) throws I
122     if (email.isEmpty()) {
123         response.sendRedirect("/forgot?e=Please+fill+out+all+fields");
124         return null;
125     } else {
126         Pattern p = Pattern.compile("^.*@[A-Za-z]*\\.[A-Za-z]*$");
127         Matcher m = p.matcher(email);
128         if (!m.matches()) {
129             response.sendRedirect("/forgot?e=Invalid+email!");
130             return null;
131         } else {
132             try {
133                 String sql = "SELECT * FROM users WHERE email = '" + email + "'";
134                 User user = (User)this.jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper(User.class
135                 Long var10000 = user.getId();
136                 String passwordResetHash = DigestUtils.md5DigestAsHex(("" + var10000 + ":" + user.getEmail()
137                 var10000 = user.getId();
138                 String passwordResetLink = "https://bluebird.htb/reset?uid=" + var10000 + "&code=" + passwor
139                 logger.error("TODO: Send email with link [" + passwordResetLink + "]");
140                 response.sendRedirect("/forgot?e=Please+check+your+email+for+the+password+reset+link");
141                 return null;
142             } catch (EmptyResultDataAccessException var10) {
143                 response.sendRedirect("/forgot?e=Email+does+not+exist");
144                 return null;
145             } catch (Exception var11) {
146                 model.addAttribute("errorMsg", var11.getMessage());
147                 model.addAttribute("errorStackTrace", Arrays.toString(var11.getStackTrace()));
148                 return "error";
149             }

Console ×   Problems   Debug Shell

Variables ×   Breakpoints   Expressions

Name                Value
this                AuthController (id=104)
email               "bmdyy@bluebird.htb" (id=11158)
model               BindingAwareModelMap (id=11159
response            HeaderWriterFilter$HeaderWriter
p                   Pattern (id=11161)
m                   Matcher (id=11162)
sql                 "SELECT * FROM users WHERE em

<Choose a previously entered expression>

SELECT * FROM users WHERE email = 'bmdyy@bluebird.htb'
```

Conclusion

`Live debugging` can be a very powerful technique, but it will not always work 100% correctly since we are working with `decompiled` source code and not with the actual source code. In the end, everybody has their own preferred workstyles, so the best thing is to just try it out and see for yourself.

Hunting for SQL Errors

Enabling PostgreSQL Logging

Another way to identify the `SQL` queries which are run, as well as debug your payloads when developing an exploit is to enable `SQL logging`.

To do so in `PostgreSQL`, we first need to find `postgresql.conf`. Usually it is located in `/etc/postgresql/<version>/main/`, but if you can't find it there you can run:

mayala@htb[/htb] `$ find / -type f -name postgresql.conf 2>/dev/null`
`/etc/postgresql/13/main/postgresql.conf`

Once we've located the file, we have to make the following changes to the file:

- Change `#logging_collector = off` to `logging_collector = on`. This enables the logging collector background process [source].
- `#log_statement = 'none'` to `log_statement = 'all'`. This makes it so all statement types (SELECT, CREATE, INSERT, ...) are logged [source].
- Uncomment `#log_directory = '...'` to define the directory in which the logfiles will be saved [source].
- Uncomment `#log_filename = '...'` to define the filename in which logfiles will be saved [source].

Once the changes have been saved, restart `PostgreSQL` like so:

mayala@htb[/htb] `$ sudo systemctl restart postgresql`

At this point, the log file(s) should start appearing in the folder defined by `log_directory`. We can watch the log messages in near-realtime with the following command:

mayala@htb[/htb] `$ sudo watch -n 1 tail <log_directory>/postgresql-2023-02-14_081533.log <SNIP> 2023-02-14 09:06:04.819 EST [22510] bbuser@bluebird LOG: execute <unnamed>: SELECT * FROM users WHERE username = $1 2023-02-14 09:06:04.819 EST [22510] bbuser@bluebird DETAIL: parameters: $1 = 'bmdyy' 2023-02-14 09:06:10.423 EST [22510] bbuser@bluebird LOG: execute <unnamed>: SELECT * FROM users WHERE username = $1 2023-02-14 09:06:10.423 EST [22510] bbuser@bluebird DETAIL: parameters: $1 = 'admin' 2023-02-14 09:06:12.999 EST [22510] bbuser@bluebird LOG: execute <unnamed>: SELECT * FROM users WHERE username = $1 2023-02-14 09:06:12.999 EST [22510] bbuser@bluebird DETAIL: parameters: $1 = 'test' 2023-02-14 09:06:16.688 EST [22510] bbuser@bluebird LOG: execute <unnamed>: SELECT * FROM users WHERE username = $1 2023-02-14 09:06:16.688 EST [22510] bbuser@bluebird DETAIL: parameters: $1 = 'itsmaria'`