Preventing SQL injection

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:

Code: java

```java
// 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:

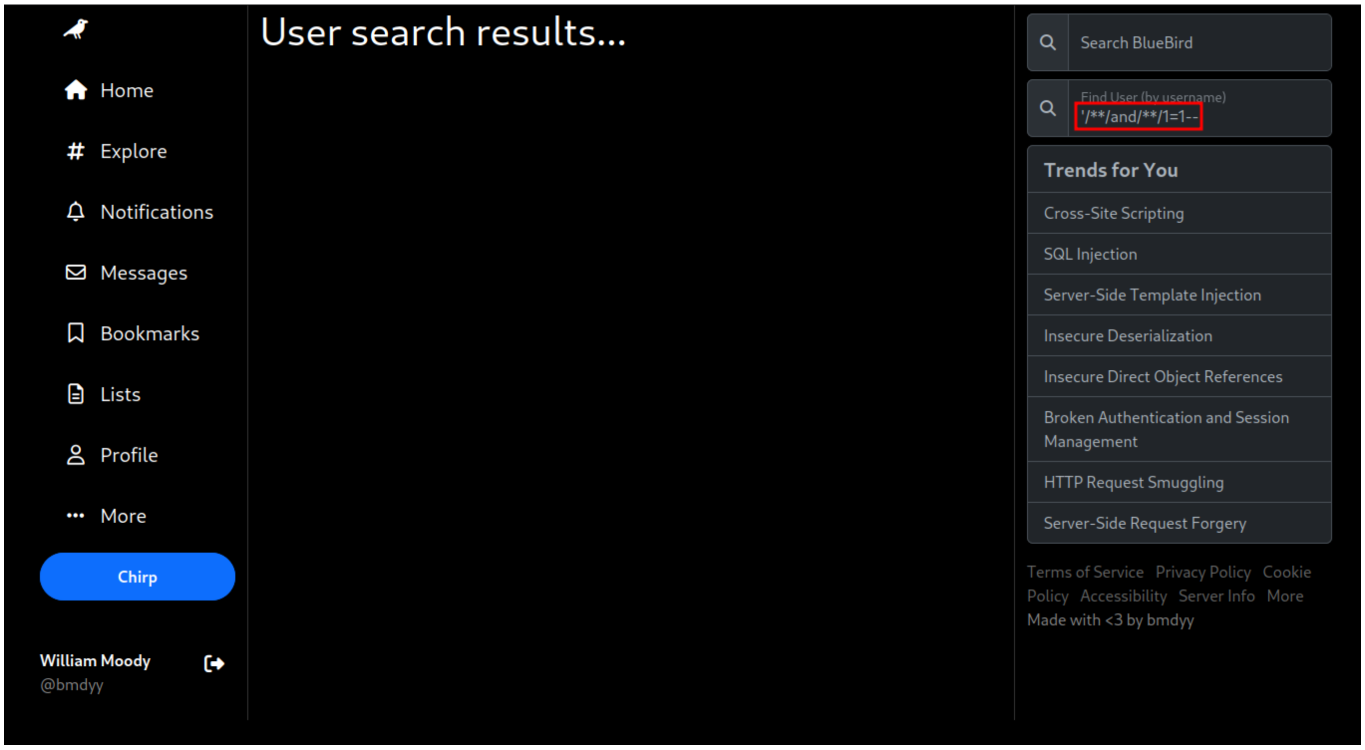
Code: java

```
// 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:



User search results...

'/**/and/**/1=1--

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.