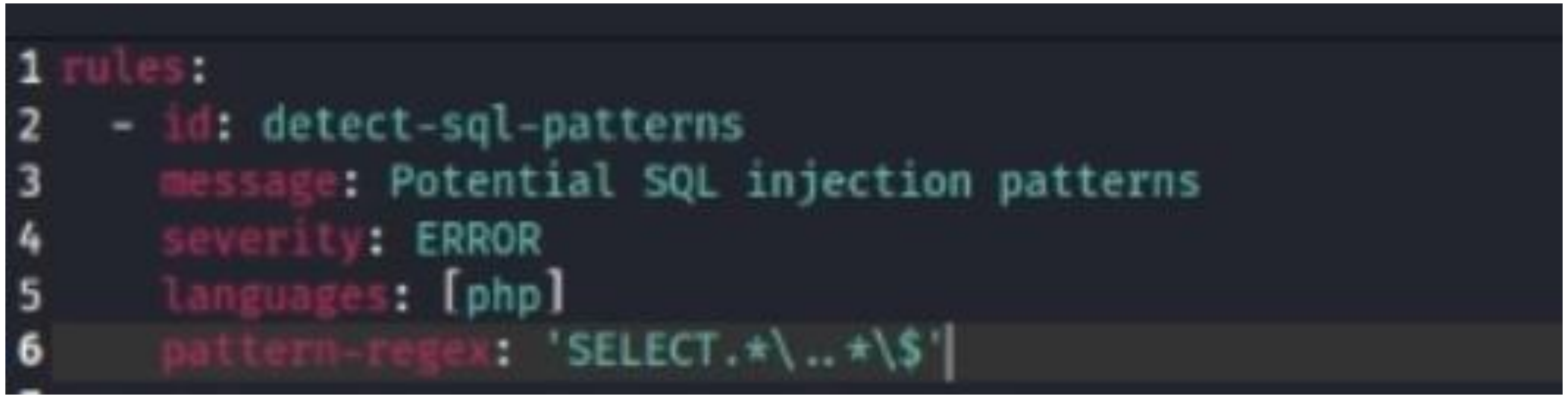
Youssef hassan

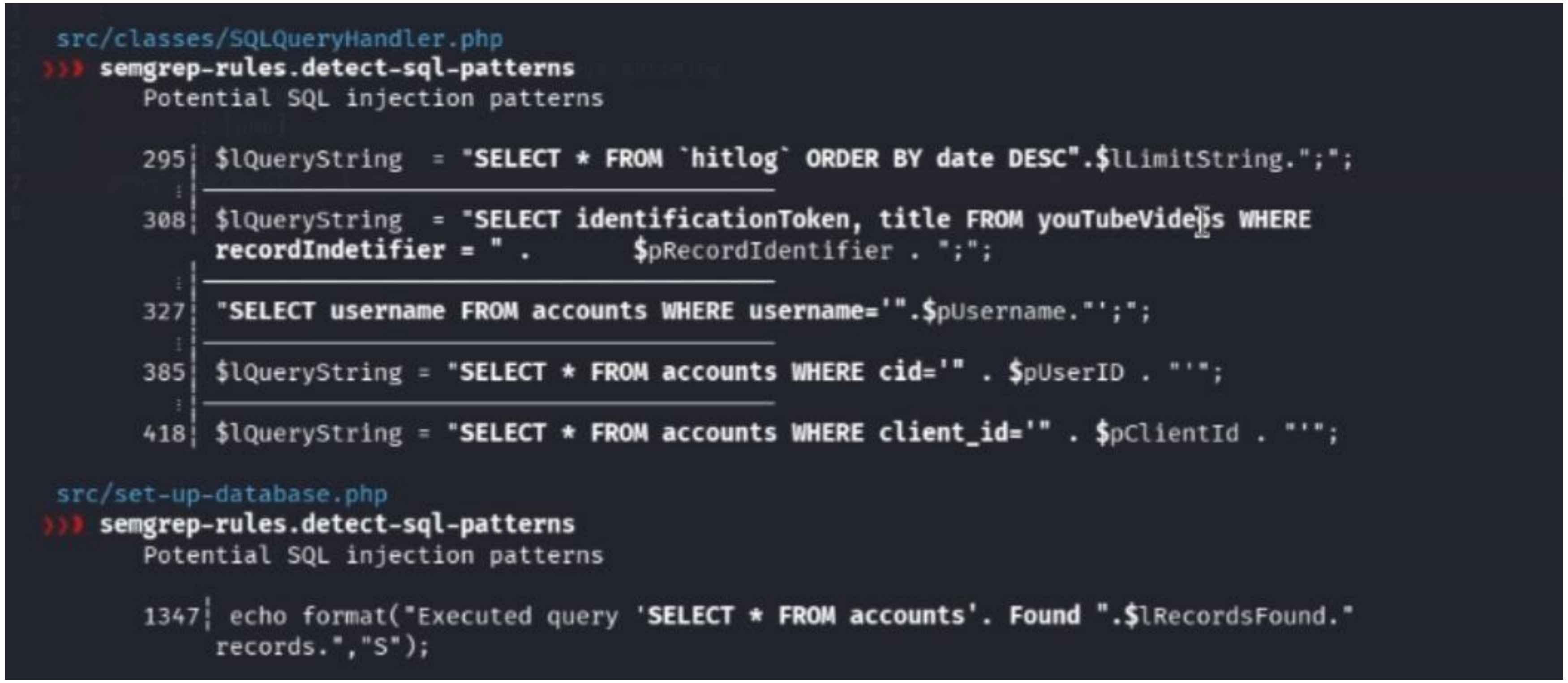
2305049

The rules used in detecting SQL injection vulnerabilities



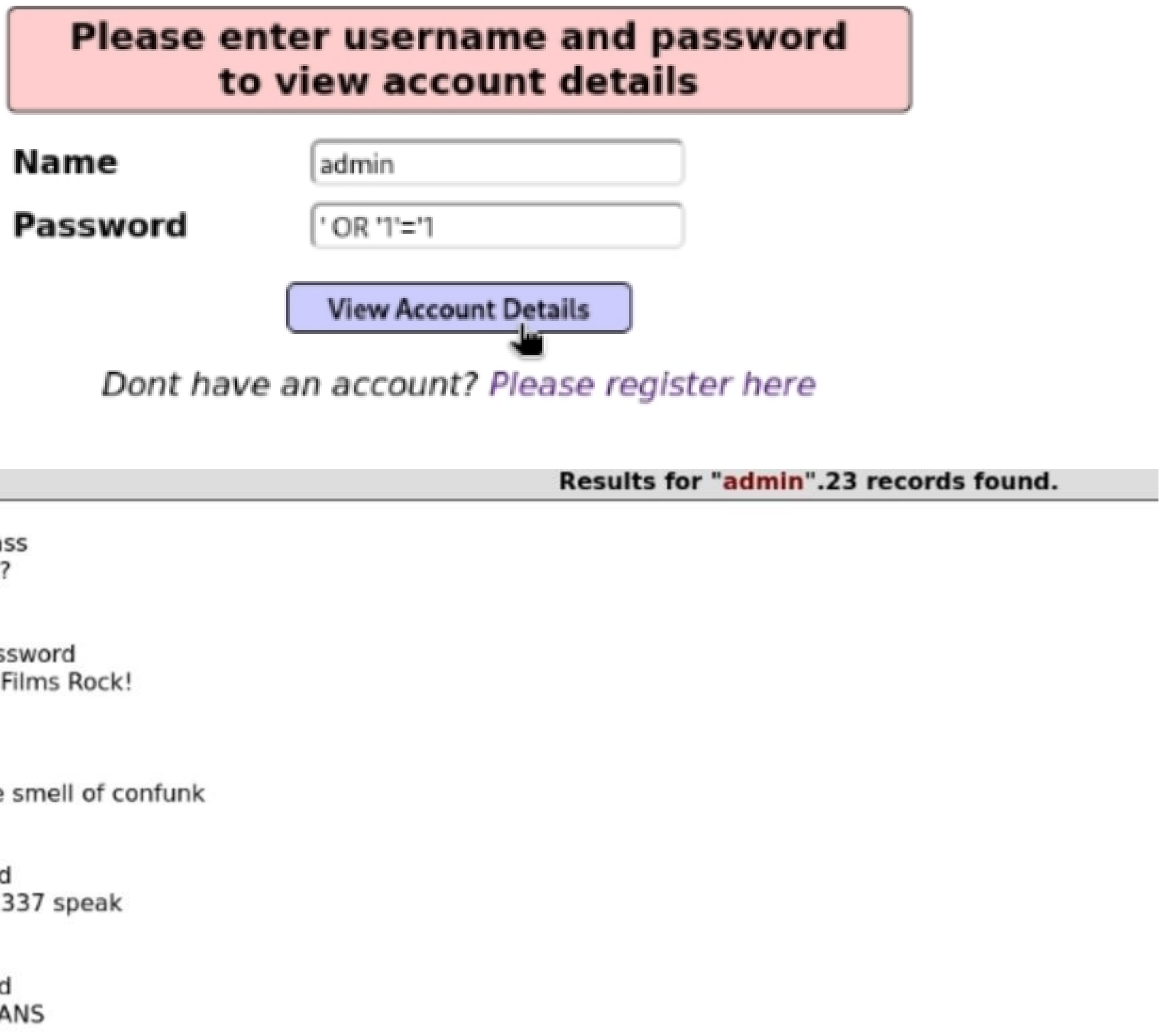
```yaml
rules:
  - id: detect-sql-patterns
    message: Potential SQL injection patterns
    severity: ERROR
    languages: [php]
    pattern-regex: 'SELECT.*\..*\$'
```

Output:



```
  src/classes/SQLQueryHandler.php
 ❯❯❱ semgrep-rules.detect-sql-patterns
        Potential SQL injection patterns

        295┆ $lQueryString  = "SELECT * FROM `hitlog` ORDER BY date DESC".$lLimitString.";";
           ⋮┆----------------------------------------
        308┆ $lQueryString  = "SELECT identificationToken, title FROM youTubeVideos WHERE
             recordIndetifier = " .       $pRecordIdentifier . ";";
           ⋮┆----------------------------------------
        327┆ "SELECT username FROM accounts WHERE username='".$pUsername."';";
           ⋮┆----------------------------------------
        385┆ $lQueryString = "SELECT * FROM accounts WHERE cid='" . $pUserID . "'";
           ⋮┆----------------------------------------
        418┆ $lQueryString = "SELECT * FROM accounts WHERE client_id='" . $pClientId . "'";

  src/set-up-database.php
 ❯❯❱ semgrep-rules.detect-sql-patterns
        Potential SQL injection patterns

        1347┆ echo format("Executed query 'SELECT * FROM accounts'. Found ".$lRecordsFound."
             records.","S");
```

SQL injection in action

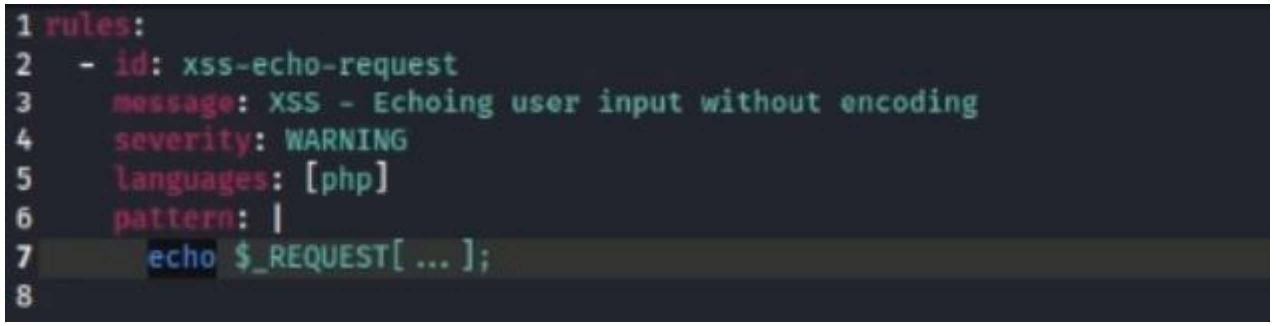


Please enter username and password to view account details

Name	admin
Password	' OR '1'='1

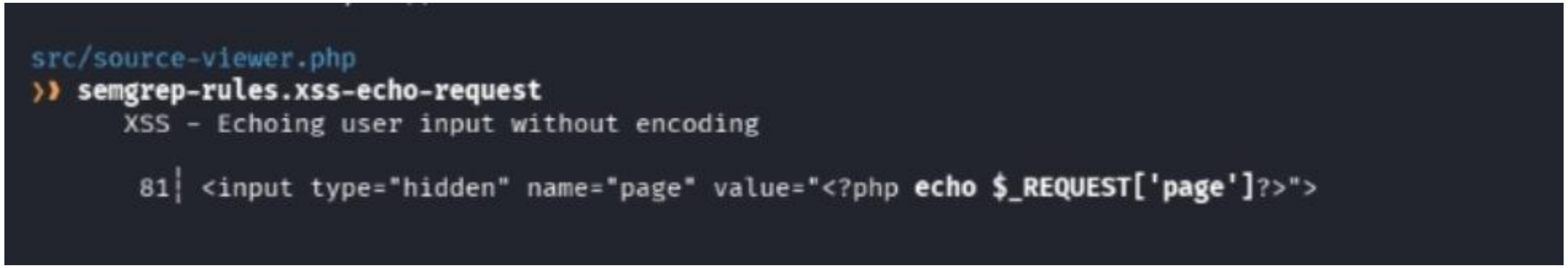
```
Results for "admin".23 records found.
Username=admin
Password=adminpass
Signature=g0t r00t?

Username=adrian
Password=somepassword
Signature=Zombie Films Rock!

Username=john
Password=monkey
Signature=I like the smell of confunk

Username=jeremy
Password=password
Signature=d1373 1337 speak

Username=bryce
Password=password
Signature=I Love SANS
```

The rules used in detecting XSS vulnerabilities



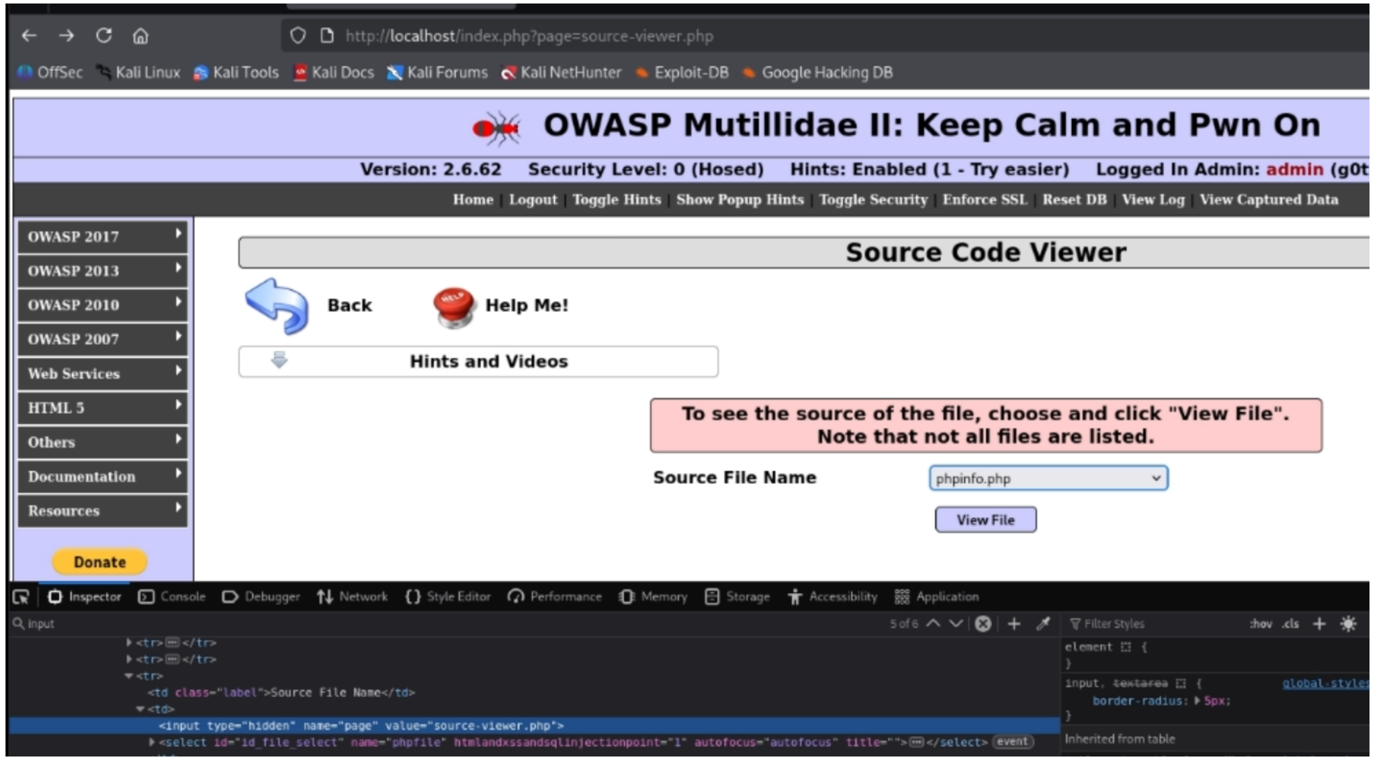
```yaml
rules:
  - id: xss-echo-request
    message: XSS - Echoing user input without encoding
    severity: WARNING
    languages: [php]
    pattern: |
      echo $_REQUEST[...];

```

Output:



```
src/source-viewer.php
❯❯ semgrep-rules.xss-echo-request
      XSS - Echoing user input without encoding

       81┆ <input type="hidden" name="page" value="<?php echo $_REQUEST['page']?>">
```

XSS in action



http://localhost/index.php?page=source-viewer.php

OWASP Mutillidae II: Keep Calm and Pwn On

Version: 2.6.62 Security Level: 0 (Hosed) Hints: Enabled (1 - Try easier) Logged In Admin: admin (g0t

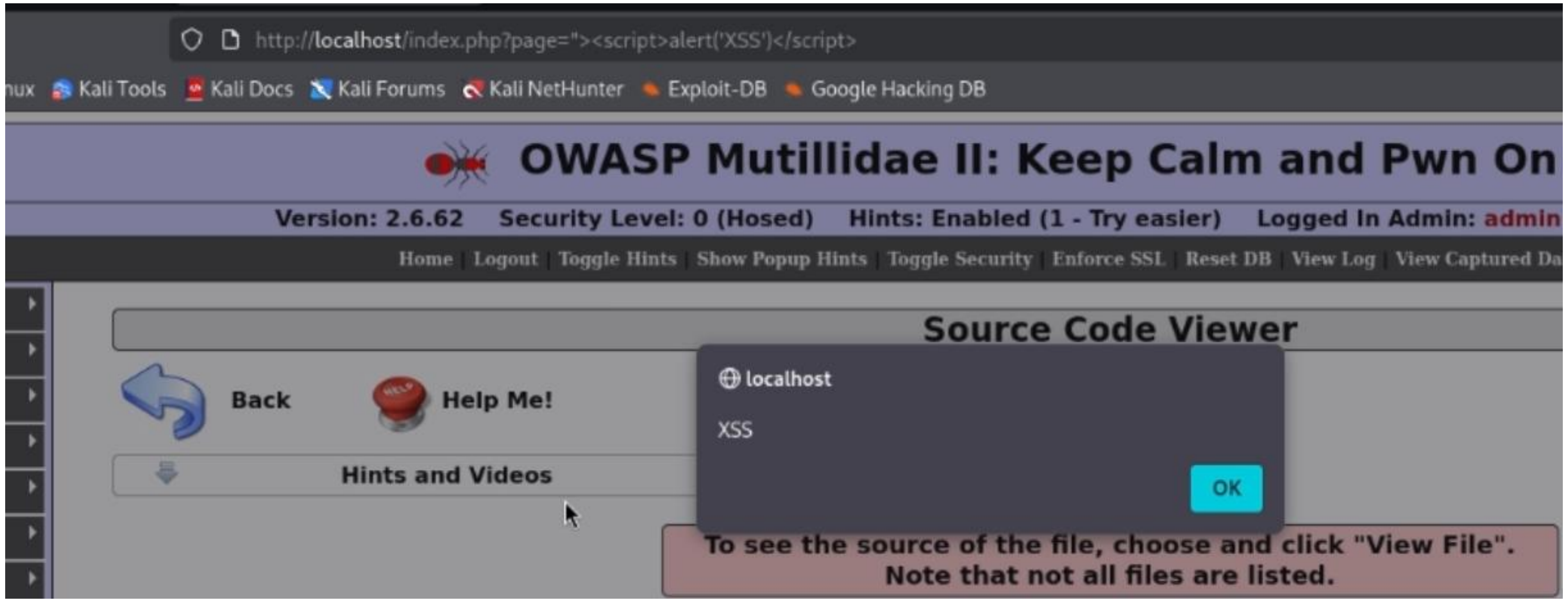
Source Code Viewer

To see the source of the file, choose and click "View File". Note that not all files are listed.

Source File Name: phpinfo.php

```
▸<tr>…</tr>
▸<tr>…</tr>
▾<tr>
   <td class="label">Source File Name</td>
  ▾<td>
     <input type="hidden" name="page" value="source-viewer.php">
    ▸<select id="id_file_select" name="phpfile" htmlandxssandsqlinjectionpoint="1" autofocus="autofocus" title="">…</select> event
```

```
element {
}
input, ~~textarea~~ {
    border-radius: 5px;
}
Inherited from table
```



http://localhost/index.php?page="><script>alert('XSS')</script>

OWASP Mutillidae II: Keep Calm and Pwn On

Version: 2.6.62 Security Level: 0 (Hosed) Hints: Enabled (1 - Try easier) Logged In Admin: admin

Source Code Viewer

localhost

XSS

To see the source of the file, choose and click "View File". Note that not all files are listed.

limitations of what i did

- so the semgrep tool is really good at finding patterns in code but it has some limitations that i noticed
- first it can only find what you tell it to look for like if i make a rule that looks for SELECT statements with variables it will find those but it might miss other types of SQL injection that look different
- also it found the code patterns but it doesnt actually know if the website is really vulnerable it just knows the pattern is dangerous so i had to manually test each one to make sure it actually worked
- another thing is semgrep only looks at the source code it doesnt actually run the website or test it live so it might find code that looks vulnerable but maybe the website has some other protection that i cant see from just reading the code
- and for the XSS it found that one echo statement but there might be other places where user input gets displayed that use different methods that my rule didnt catch

what the attack actually does

- for the SQL injection when i put `' OR '1' = '1` in the username field what happens is the database thinks im asking for all users because that condition `'1' = '1'` is always true no matter what
- so instead of just checking if the username and password match one user it returns every single user account in the system which means i can log in as anyone or see everyones information
- in a real website this could let hackers see all the user data passwords credit card numbers anything stored in the database
- for the XSS attack when i put the javascript in the URL parameter the website just blindly puts it right into the page and the browser thinks its real code and runs it
- so if i can get someone to click my malicious link their browser will run whatever javascript i want i could steal their login session redirect them to fake pages even install malware
- in the real world attackers use this to hijack accounts by stealing cookies or trick people into giving up their passwords

how to fix these problems

- for SQL injection the fix is pretty simple instead of putting user input directly into queries you use something called parameterized queries or prepared statements

- this means you tell the database hey heres the query and heres the user data separately so the database knows not to treat the user input as actual SQL commands

- for the XSS vulnerability you need to encode the user input before putting it on the page which means converting dangerous characters like < and > into safe versions that wont be treated as code

- so instead of just echoing whatever the user gives you you process it first to make sure it cant contain any javascript