Code analysis tool comparison

Security in Computer Systems—Project Report

Adrian Panek

16th November 2025

General information: In today's software development vulnerabilities tend to be extremely dangerous. It is very important to locate and eliminate them as soon as possible during product creation, because the later security weakness is discovered the more expensive it is to remediate it. Every day a lot of companies use different security scan tools, very often integrated into their Continuous Integration/Continuous Deployment process during building, testing and deploying their software. Despite world-wide adoption of CI/CD tools many organisations still struggle to identify and mitigate security vulnerabilities throughout software development lifecycle. Vulnerabilities can range from regular coding error, misconfigurations or outdated dependencies to very comples issues such as insecure APIs or poor authentication methods. With integration of CI/CD pipeline it is possible to reduce risk and to ensure that vulnerabilities are addressed all the time, not only after major crash. The following project aims to present and assess different security scanning tools for detecting and solving security weaknesses early in the development process what will lead to more secure software systems.

1 Introduction

Security analysis tools can detect software vulnerabilities before the release of the software. The comparison of SonarCloud, Aikido Security, Snyk, Bandit, and Horusec will help an organization to choose which tool best fits its security requirements. Precise tools help in the early detection of vulnerabilities, reducing the risk of a security breach, along with costs related to remediation. This study compares the tools based on their capability related to the identification of common vulnerabilities and highlights their impact on the security of software.

2 Materials and Methods

For the purpose of comparison a sample Django (1) web application (2) has been created. This intentionally vulnerable app contains three well-known web vulnerabilities that attackers could exploit to manipulate data, extract sensitive information, or execute malicious actions. Below are the key vulnerabilities and a description of their potential impact.

2.1 SQL Injection

SQL injection occurs when unsanitized user input is integrated directly into SQL queries, allowing attackers to manipulate the query structure. In this case, user input is directly embedded into the SQL query using the following piece of code:

```
cursor.execute(f"SELECT email FROM
 ↪  vulnerable_userprofile WHERE username =
 ↪  '{username}'")
```

Listing 1: Code of Django view with vulnerable search in database which can lead to data leakage

Since there is no input validation or parameterization, an attacker can input malicious SQL code to alter the query.

In example, the following Linux terminal command will send a request to webserver to get details about one particular user and email of the user specified in the parameter will be returned, but at the same time the rest of users email will get exposed:

```
curl "http://appurl.com/vulnerable/user_info/
?username=account1'OR'1'='1"
```

Listing 2: Linux command to extract all users from database

2.2 CSRF Token bypass

CSRF (Cross-Site Request Forgery) is a web security vulnerability that tricks authenticated users into unknowingly executing malicious actions on a web application, allowing attackers to perform unauthorized actions on the user's behalf. In created application there is an endpoint for changing each users' password which is vulnerable to CSRF attacks. Attackers can craft a malicious form to trigger this view from a different site. That can have two bad scenarios:

- Unauthorized Actions - attackers could craft a malicious link or form that, when accessed by a logged-in user, automatically submits a request to change their password.

- Account Takeover - if a user is logged into the vulnerable site, an attacker can silently force the user to change their password, locking them out of their own account.

The following piece of code is responsible for exposure to this vulnerability:

```
@csrf_exempt
def update_password(request):
    if request.method == "POST":
        username = request.POST.get("username", "")
        new_password = request.POST.get("password",
         ↪  "")
```

Listing 3: Code of Django view which disables check of CSRF token during web request

The following request created with Linux CURL command will allow unauthenticated attacker to change user's password without their knowledge.

```
curl -X POST -d
 ↪  "username=admin&password=changed_password"
 ↪  http://appurl.com/vulnerable/update_password/
```

Listing 4: CURL HTTP request to change password

2.3 Secret key exposed in source code

Every Django application has its own secret key generated and it is crucial part of the app's security. Key's main purpose is:

- Cryptographic Signing - Django uses the secret key to provide cryptographic signing, ensuring that data shared between the server and clients has not been manipulated. It can be used to sign cookies, sessions etc.

- Hashing Passwords - Django uses the secret key in combination with other techniques to hash and securely store user passwords. This ensures that passwords are not stored in plain text and remain secure.
- Session Security - secret key is used to sign session cookies, ensuring that a malicious user can not tamper with them.

The following piece of code is responsible for exposure to this vulnerability:

```
SECRET_KEY = 'django-insecure-w(s53t6h8i=ide7zv=
a%g87@lp$460ua_520y0z_6ad%chp3=7'
```

Listing 5: Secret key stored as plain text variable in application's settings

2.4 Cross-Site Scripting

Cross-Site Scripting (XSS) is a type of vulnerability in web applications which is based on injection of malicious code into a website, which is then being executed by the victim's browser. Vulnerable code in examined Django application is responsible for reflecting user input directly in the response without escaping (process of using a special character, to change the way the following character or sequence of characters is interpreted by a system), which allow attacker to inject JavaScript or other malicious code. In this case *username* parameter is reflected directly in the HttpResponse, lacking escape. The following scenario can be exploited by crafting a URL containing malicious code, when victim accesses this URL, the browser will execute injected code.

Utilizing this vulnerability is extremely simple, but effective, as entering the following link in browser will lead to execution of JavaScript statement injected as parameter:

```python
def user_info(request):
username = request.GET.get("username", '')

return HttpResponse(f"No username found for:
 ↪  {username}")
```

Listing 6: User input directly returned as HttpResponse

```
http://appurl.com/vulnerable/user_info/
?username=<script>alert("XSS Test")</script>
```

Listing 7: User input directly returned as HttpResponse

3 Compared tools

The following list of tools has been used to scan the source code of mentioned application:

- SonarCloud (3) - a cloud-based code quality and security service developed by SonarSource. In given case a free trial version was used in order to perform analysis. After integration with GitHub repository a scan has been carried out automatically. Afterwards a full report with all vulnerabilities was presented.

- Aikido security (4) - this tool is designed to help organizations efficiently discover and address security vulnerabilities in their software and infrastructure. It helps organizations stay proactive by identifying vulnerabilities early and offering actionable insights to reduce security risks. As previously it is used on a trial basis, and after authentication with GitHub a scan was performed. The results came up as report in tool's graphical user interface (GUI).

- Snyk (5) - a developer-focused security platform that helps organizations identify and remediate vulnerabilities in their open-source dependencies and code. As in two previous examples, usage of this tool was based on a free trial basis and after successful integration with source code stored in GitHub scan has been performed and full list of vulnerabilities were showed on screen.

- Bandit (7) - a security-focused static analysis tool designed specifically for Python applications. In that case tool comes in by another Python package installed via pip in dedicated virtual environment. Running this software requires only presence of source code on the machine and invocation of command:

  ```
  bandit -r insecure_app/ vulnerable/
  ```

Listing 8: Command that allows to run bandit tool for directory called project

  After successful run a whole report appeared in terminal with all found vulnerabilities in the source code.

- Horusec (6) - an open-source security platform designed to assist developers in identifying and mitigating vulnerabilities in their applications. In given case it was ran with Docker container and after succesful scan a full report has been printed out in the console, as well as written into a text file.

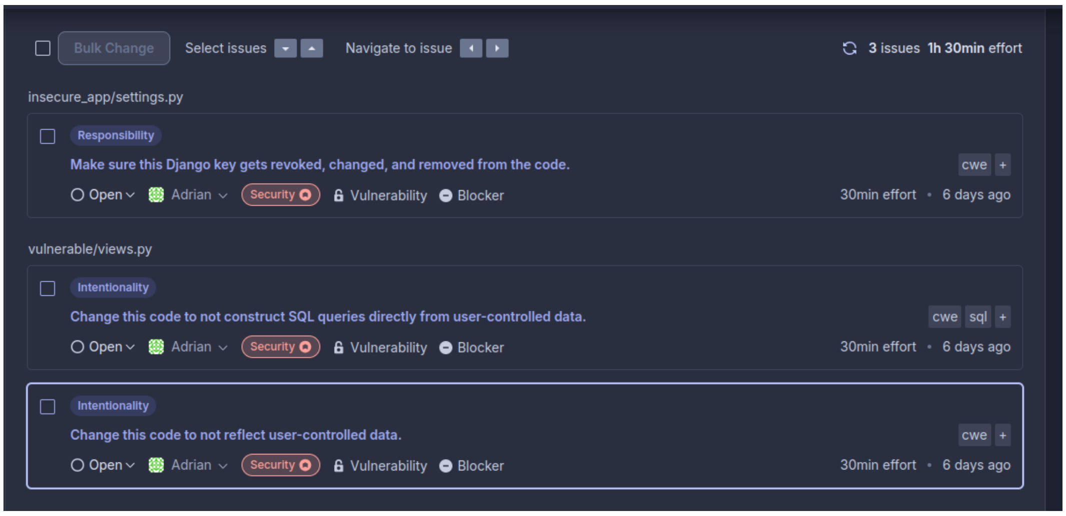
  ```
  docker run --rm -v
   ↪  /var/run/docker.sock:/var/run/docker.sock
   ↪  -v $(pwd):/src/horusec
   ↪  horuszup/horusec-cli:v2.9.0-beta.3 horusec
   ↪  start -p /src/horusec -P $(pwd)
   ↪  --ignore="**/venv/**"
  ```

Listing 9: Command for invoking horusec tool

4 Findings of study

4.1 SonarCloud

SonarCloud managed to find majority of presented vulnerabilities, the only exception being CSRF token. Screenshot from tool's GUI can be found on fig 1.



insecure_app/settings.py

Responsibility
Make sure this Django key gets revoked, changed, and removed from the code.
Open · Adrian · Security · Vulnerability · Blocker · cwe · 30min effort · 6 days ago

vulnerable/views.py

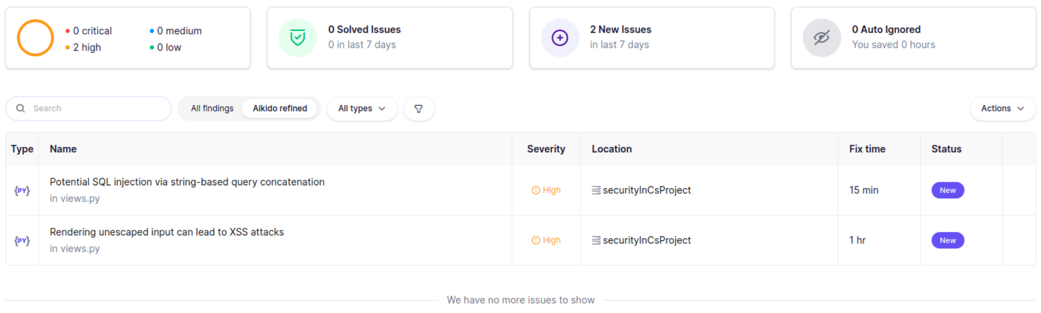
Intentionality
Change this code to not construct SQL queries directly from user-controlled data.
Open · Adrian · Security · Vulnerability · Blocker · cwe sql · 30min effort · 6 days ago

Intentionality
Change this code to not reflect user-controlled data.
Open · Adrian · Security · Vulnerability · Blocker · cwe · 30min effort · 6 days ago

Figure 1: GUI screen from SonarCloud

4.2 Aikido Security

Aikido Security was able to find only 50% of vulnerabilities, identifying the least vulnerabilities among all the tools examined in the competition. Raport from scan can be found on image 2.


● 0 critical ● 0 medium
● 2 high ● 0 low

0 Solved Issues
0 in last 7 days

2 New Issues
in last 7 days

0 Auto Ignored
You saved 0 hours

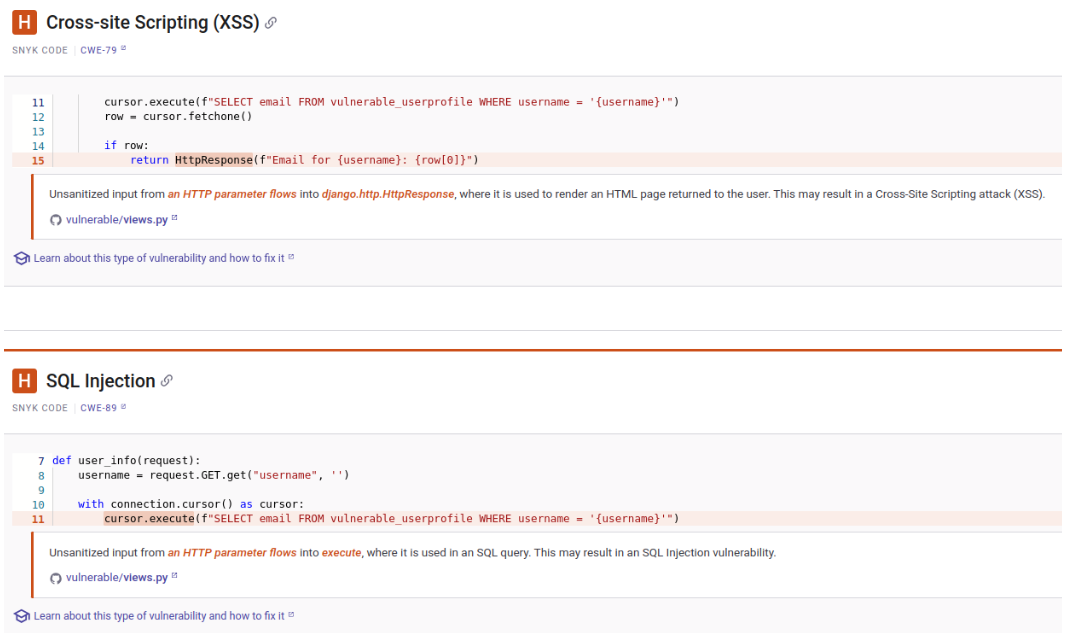
Type	Name	Severity	Location	Fix time	Status
{PY}	Potential SQL injection via string-based query concatenation in views.py	High	securityInCsProject	15 min	New
{PY}	Rendering unescaped input can lead to XSS attacks in views.py	High	securityInCsProject	1 hr	New

We have no more issues to show

Figure 2: GUI screen from Aikido Security

4.3 Snyk

Snyk software was able to detect all potential security gaps. All vulnerabilities found are shown on the images 3-4.


H Cross-site Scripting (XSS)

SNYK CODE | CWE-79

```
    cursor.execute(f"SELECT email FROM vulnerable_userprofile WHERE username = '{username}'")
    row = cursor.fetchone()

    if row:
        return HttpResponse(f"Email for {username}: {row[0]}")
```

Unsanitized input from *an HTTP parameter flows* into *django.http.HttpResponse*, where it is used to render an HTML page returned to the user. This may result in a Cross-Site Scripting attack (XSS).

vulnerable/**views.py**

Learn about this type of vulnerability and how to fix it

H SQL Injection

SNYK CODE | CWE-89

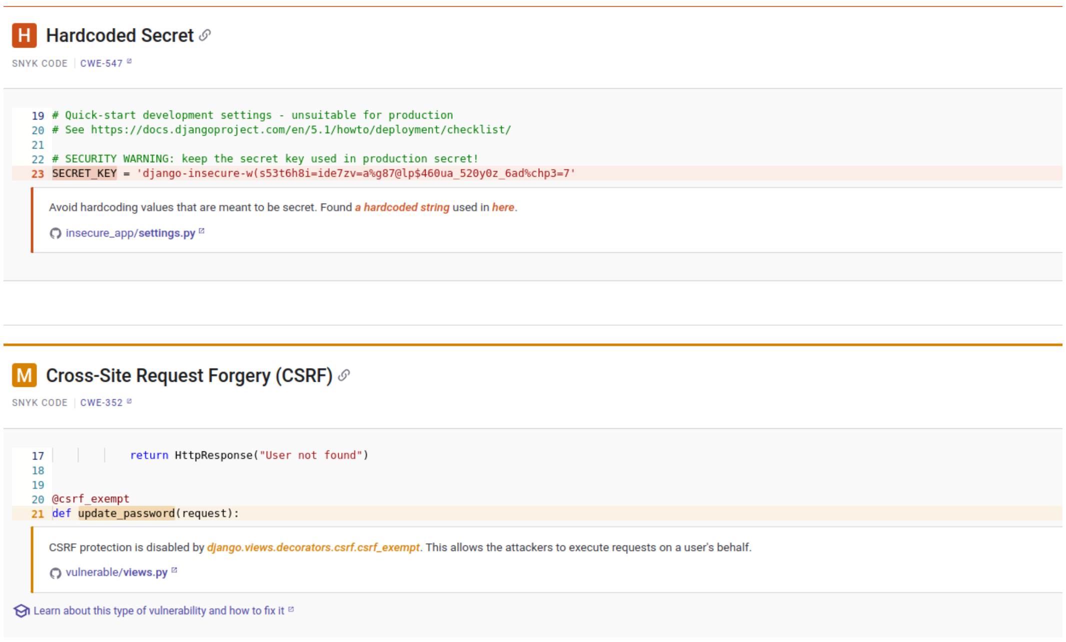
```
def user_info(request):
    username = request.GET.get("username", '')

    with connection.cursor() as cursor:
        cursor.execute(f"SELECT email FROM vulnerable_userprofile WHERE username = '{username}'")
```

Unsanitized input from *an HTTP parameter flows* into *execute*, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

vulnerable/**views.py**

Learn about this type of vulnerability and how to fix it

Figure 3: GUI screen from Snyk


H Hardcoded Secret

SNYK CODE | CWE-547

```
# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/5.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-w(s53t6h8i=ide7zv=a%g87@lp$460ua_520y0z_6ad%chp3=7'
```

Avoid hardcoding values that are meant to be secret. Found *a hardcoded string* used in *here*.

insecure_app/settings.py

M Cross-Site Request Forgery (CSRF)

SNYK CODE | CWE-352

```
            return HttpResponse("User not found")


@csrf_exempt
def update_password(request):
```

CSRF protection is disabled by *django.views.decorators.csrf.csrf_exempt*. This allows the attackers to execute requests on a user's behalf.

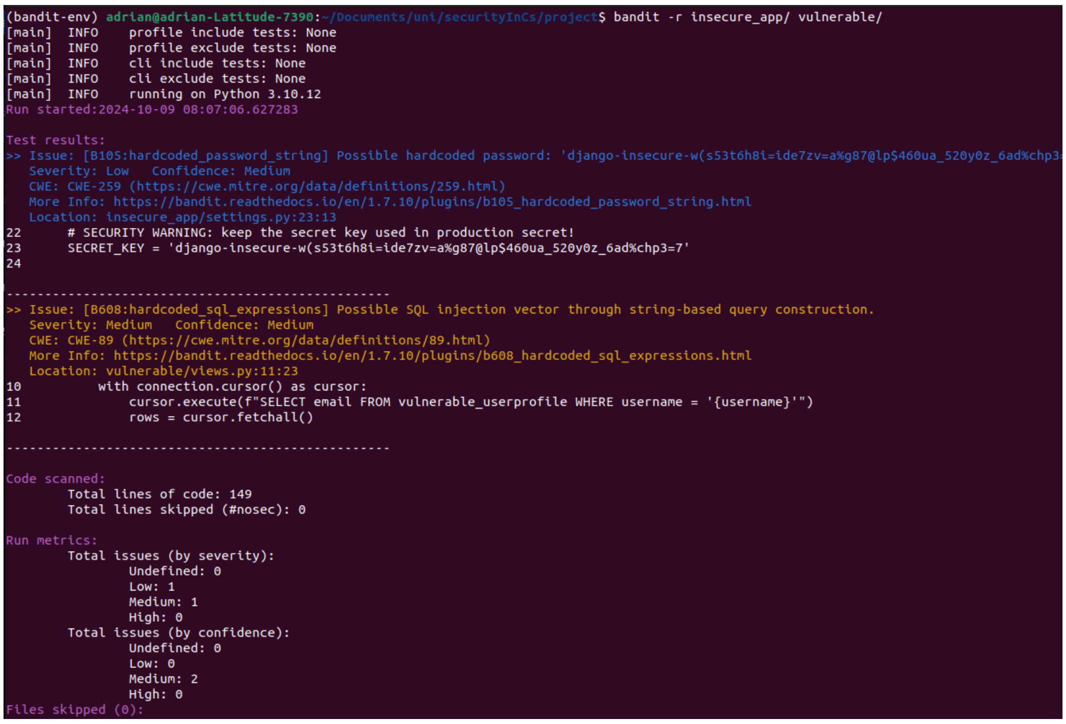
vulnerable/views.py

Learn about this type of vulnerability and how to fix it

Figure 4: GUI screen from Snyk

4.4 Bandit

Bandit software, despite being strictly for the Python application, managed to find only the most common vulnerabilities created in the application. Results of Bandit's scan are shown on figure 5.

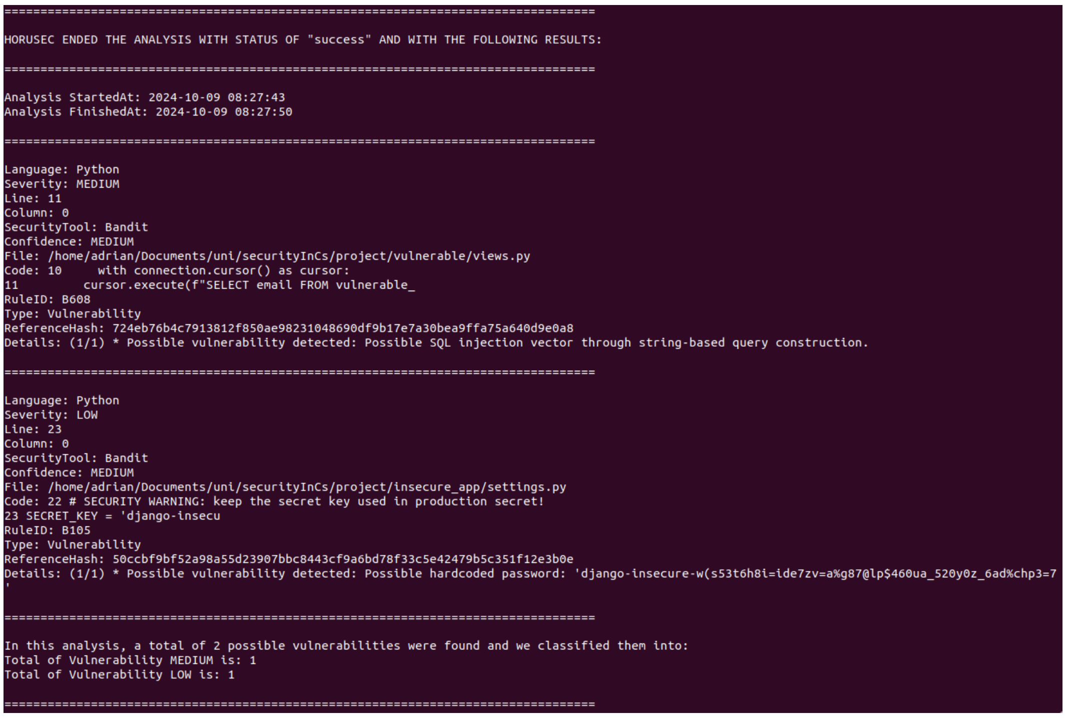


```
(bandit-env) adrian@adrian-Latitude-7390:~/Documents/uni/securityInCs/project$ bandit -r insecure_app/ vulnerable/
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.10.12
Run started:2024-10-09 08:07:06.627283

Test results:
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'django-insecure-w(s53t6h8i=ide7zv=a%g87@lp$460ua_520y0z_6ad%chp3
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b105_hardcoded_password_string.html
   Location: insecure_app/settings.py:23:13
22      # SECURITY WARNING: keep the secret key used in production secret!
23      SECRET_KEY = 'django-insecure-w(s53t6h8i=ide7zv=a%g87@lp$460ua_520y0z_6ad%chp3=7'
24

--------------------------------------------------
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
   Severity: Medium   Confidence: Medium
   CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
   More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b608_hardcoded_sql_expressions.html
   Location: vulnerable/views.py:11:23
10          with connection.cursor() as cursor:
11              cursor.execute(f"SELECT email FROM vulnerable_userprofile WHERE username = '{username}'")
12              rows = cursor.fetchall()

--------------------------------------------------

Code scanned:
        Total lines of code: 149
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 1
                Medium: 1
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 2
                High: 0
Files skipped (0):
```

Figure 5: Results of Bandit scan

4.5 Horusec

Horusec, similarly to Bandit, was unable to find majority of weaknesses in Django application. The most crucial ones like plan-text stored password, as well as SQL injection were detected, but less obvious ones were skipped. Figure 6 presents the results of Horusec's scan.



```
==================================================================================

HORUSEC ENDED THE ANALYSIS WITH STATUS OF "success" AND WITH THE FOLLOWING RESULTS:

==================================================================================

Analysis StartedAt: 2024-10-09 08:27:43
Analysis FinishedAt: 2024-10-09 08:27:50

==================================================================================

Language: Python
Severity: MEDIUM
Line: 11
Column: 0
SecurityTool: Bandit
Confidence: MEDIUM
File: /home/adrian/Documents/uni/securityInCs/project/vulnerable/views.py
Code: 10     with connection.cursor() as cursor:
11         cursor.execute(f"SELECT email FROM vulnerable_
RuleID: B608
Type: Vulnerability
ReferenceHash: 724eb76b4c7913812f850ae98231048690df9b17e7a30bea9ffa75a640d9e0a8
Details: (1/1) * Possible vulnerability detected: Possible SQL injection vector through string-based query construction.

==================================================================================

Language: Python
Severity: LOW
Line: 23
Column: 0
SecurityTool: Bandit
Confidence: MEDIUM
File: /home/adrian/Documents/uni/securityInCs/project/insecure_app/settings.py
Code: 22 # SECURITY WARNING: keep the secret key used in production secret!
23 SECRET_KEY = 'django-insecu
RuleID: B105
Type: Vulnerability
ReferenceHash: 50ccbf9bf52a98a55d23907bbc8443cf9a6bd78f33c5e42479b5c351f12e3b0e
Details: (1/1) * Possible vulnerability detected: Possible hardcoded password: 'django-insecure-w(s53t6h8i=ide7zv=a%g87@lp$460ua_520y0z_6ad%chp3=7
'

==================================================================================

In this analysis, a total of 2 possible vulnerabilities were found and we classified them into:
Total of Vulnerability MEDIUM is: 1
Total of Vulnerability LOW is: 1

==================================================================================
```

Figure 6: Results of Horusec scan

Each examined tool had strengths and weaknesses related to the detection of vulnerabilities. For example, SonarCloud detected most but failed to find vulnerabilities related to CSRF. Aikido Security identified only few vulnerabilities, such as SQL Injection and Cross-Site Scripting (XSS) exposures. Snyk was flawless in detecting all types of weaknesses, while Bandit and Horusec detected only well-known Python-related issues; both failed to detect common web application weaknesses that usually are more complex, like CSRF and exposed keys.

Moreover, the UI of each of these tools varied: SonarCloud and Snyk both provided modern dashboard interfaces that showed vulnerabilities sorted by severity, thus helping speed up action that needed to be taken. Aikido Security did provide a graphical interface, although less verbose. Horusec and Bandit, in turn, are command-line tools, hence requiring more technical familiarity. Their ease of use may not be uncomfortable for developers not used to working with such command-line tools on a daily basis.

Snyk and SonarCloud are based on a freemium model, with premium features for enterprise users. Horusec and Bandit are open-source tools; they are easy to use, however, they seem to be not as detailed as their counterparts. Aikido Security is subscription-based, which may be a determining factor for smaller teams on a tight budget, considering the lack of performance shown in this study.

5 Conclusions

The following work has demonstrated different efficiencies of SonarCloud, Aikido Security, Snyk, Bandit, and Horusec. Each of the tested tools was run against the vulnerable code in the sample Django web application with very common security weaknesses: SQL Injection, CSRF Token Bypass, secret key disclosure, and Cross Site Scripting. The results provided by Snyk were the most detailed and showed all the vulnerabilities, while Aikido Security combined with open source CLI tools identified the least number, leading to the conclusion that software development tools should be very careful during tool selection.

The results of the following research point out that, even though security tools can be useful in risk mitigation tasks, their limitations need to be known by the teams. A dependence on a single tool may result in leaving gaps in security, which can result in significant breaches. Thus, it is essential for an organization to follow a multi-layered approach by putting several tools and practices together for robust protection against software vulnerabilities.

References

[1] https://www.djangoproject.com/

[2] https://github.com/adrian-panek/securityInCsProject

[3] https://www.sonarsource.com/

[4] https://www.aikido.dev/

[5] https://app.snyk.io/

[6] https://horusec.io/

[7] https://bandit.readthedocs.io/en/latest/