Static Code Analysis

Table of Contents

Summary

I conduct static code analysis for two applications built in Python. Static code analysis for source codes of each application is performed primarily using the Bandit tool, and supplemented by the Pylint and Pyflakes tools. To provide a holistic analysis, the Wapiti tool is finally used on each application to evaluate deployed web application web pages. While this tool does not scan source code, it provides further security insights which the enterprise may wish to take note of.

I then summarize key findings from each scan, and reference sources which can be consulted for more information concerning specific vulnerabilities. As an issue is introduced in each “Findings” section, some suggestions for remediation are provided. Major recommendations are then provided, followed by a methodology of each scan. Appendices are included to provide the output of each scan, which the enterprise may wish to consult to identify additional vulnerabilities to address.

Application 1 findings

Bandit

The commands and methodology used to generate these findings are detailed in the "Methodology" section to follow. The first scan run is a simple Bandit command, which does not yield any high severity issues - but which does generate high confidence issues. The two medium severity, high confidence issues generated were the use of "exec". The Bandit scan references CWE-703, which attributes the riskiness of using the exec command to enabling arbitrary code execution. This arbitrary code may cause security issues if it is not implemented aptly, allowing attackers to execute "unexpected, dangerous" commands directly on the OS. Hence, it is highly recommended to use an alternative approach, or to ensure any user input is validated before the exec() command is used (Bandit developers). These issues were specifically noticed in column 12 of line 72, and column 16 of line 76.

The low-severity, high-confidence issue detected in three instances was that of try_except_pass, a potential improper check or handling of exceptional conditions. The presence of this check is a risk because a raised number of errors from a service may be indicative of an attempt being made to disrupt or interfere with the system. This may lead to potential failures and system crashes. To prevent these outcomes, two steps are recommended. First, rather than proceeding with a "pass," it is recommended for developers to specify potential exceptions in order to secure the code further. In order to achieve this heightened security, developers may configure the code to ignore the try_except_pass sequence when an exception is inputted. At the very least, errors should be logged even if they are passed.

More importantly, however, it is important to keep in mind not all exceptions can be accounted for, and dependable systems can not adequately tolerate all unexpected conditions. Hence, the second step is to implement a method of graceful degradation, that may prove handy in the event that the system was to crash - without causing drastic hazards (Shelton). While one instance of this try_except_pass command would not be regarded as a major cause for concern, the presence of three instances - in line 65, column 8, line 77, column 12, and line 79, column 8, of the file views.py. For more information on the potential risks associated with this issue, refer to CWE 703, or B110 of Bandit documentation.

Pylint

The Pylint scan for Application 1 generates insights into the application's code security, with an evaluation of compliance with PEP 8 guidelines. The score reported for the application was 5.79/10, with some potential concerns. Some notable insights from the results, presented in Appendix 1.2, are as follows, in order of descending importance:

1) Unused modules imported: the Pylint scan brings up several modules, models, patterns and functions that are imported but unused, represented by W0611 (For instance: unused patterns, unused includes, unused models imported from django.db). Unused imports may increase resource usage, which can affect load times. Higher bandwidth and memory is used, in addition to adding performance overhead during runtime and risking creating import cycles. To avoid these consequences, the imports should either be used or removed entirely.

2) Some `no_name_in_module` errors, and related import errors; these can be fixed by ensuring module names are correct, and the path to each module is also present and correct. Dependencies should not be omitted when necessary.
3) General formatting oversights in the code (such as missing docstring modules). While not a major security concern, these oversights can be addressed to produce more accessible code and can be completed swiftly. This can be done by adding a comment at the top of each module, providing a brief overview of code capability.

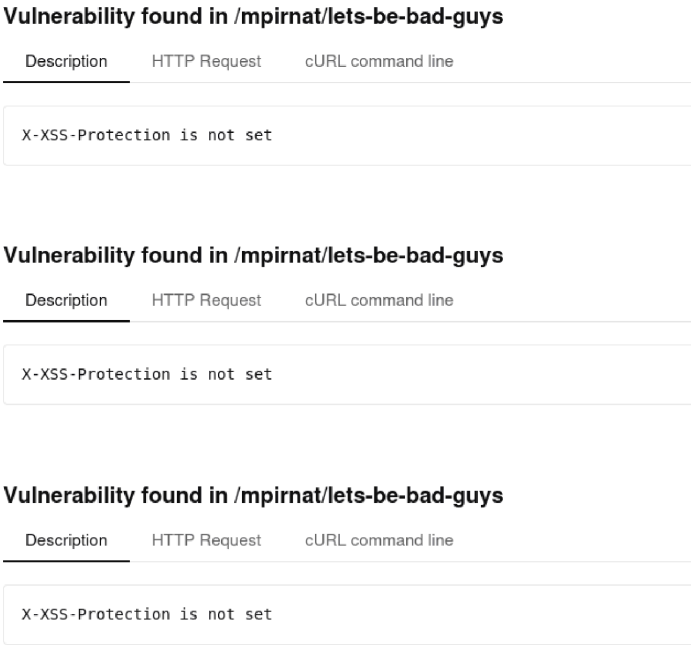
Reviewing these three points and implementing the suggested changes will not only improve the readability and formatting of existing code, but will also make the code more accessible to future developers.

Pyflakes

The Pyflakes results generated from a scan of this application reaffirm some of the concerns brought up from the Pylint scan, and specifically highlighted imports which have not been used. The importance of reviewing these imports are outlined above, and only necessary imports should be included in the code. Supposing a vulnerability was found associated with an external model, pattern, or module which was imported or unused, the application would be unnecessarily increasing its own risk of exposure to this vulnerability, heightening the probability of a zero-day attack. Hence, in addition to avoiding additional resource, memory, and bandwidth usage, and minimizing performance overhead, imports which are not used should be eliminated from the code to limit security threats.

Wapiti

Upon running the Wapiti scan for this application, three major concerns are detected. First, eleven vulnerabilities relating to HTTP secure headers were identified. HTTP security headers indicate how to behave to the browser when handling the website's content. The specific vulnerability identified was X-XSS protection not set in all eleven instances:



X-XSS protection prevents pages from loading when they detect cross-scripting attacks (MDN Web Docs). These vulnerabilities should be examined, but can reasonably be ignored. X-XSS protection is non-standard, and is unnecessary when sites implement a strong content security policy which disables the use of inline JavaScript. Furthermore, in some instances XSS protection can create XSS vulnerabilities in websites otherwise considered safe.

The second major vulnerability identified is eleven instances of a lack of the HttpOnly flag in a cookie, specifically the `_octo` cookie. Using this flag when generating a cookie helps

reduce the likelihood of client-side script accessing the protected cookie, so this flag should be included in the `_octo` cookie.

The third vulnerability is the occurrence of two internal server errors, preventing the server from processing requests.

The 500 HTTP error code is a generic error response, indicating that the server is unable to fulfill the request due to some unexpected condition. Examining these conditions further on the server-side may lend insight into the severity of these vulnerabilities.

Application 2 findings

Bandit

The Bandit scan run on Application 2 reveals many more vulnerabilities in comparison to Application 1. 49 issues in total were identified. In terms of severity, four issues were considered high severity, 36 were considered medium severity, and nine were considered low severity. In terms of confidence, four issues were deemed high confidence, 39 were deemed medium, and six were considered low confidence. Every issue generated by the Bandit scan should closely be examined, referring to each vulnerability code provided. Some vulnerabilities to pay close attention to are hardcoded password strings, hardcoded temporary directories, and hardcoded SQL expressions. Hardcoded password strings leave passwords exposed to any attacker who obtains access to the code. Typically, hardcoded passwords can result in authentication failures. Hardcoded temporary directories can render the application vulnerable to attacks, and the risk associated with hardcoded SQL expressions can be mitigated with proper input validation.

Some high confidence issues are associated with the execution of untrusted inputs. Again, these can be addressed with input validation and sanitization, which should not be overlooked at any point in code development. These issues can be resolved by ensuring such safeguards are in place. Try-except-pass instances are also detected, a shared commonality with Application 1, for which a solution has been provided above. Hardcoded passwords and hardcoded temporary directories, along with try-except-pass instances, comprise most of the medium-to-high severity and confidence issues generated by this Bandit scan. These issues should be resolved swiftly, to reduce Application 2's attack vector. For more insight into each of these vulnerabilities, refer to CWE-377, CWE-259, and CWE-89.

Pylint

The Pylint scan for Application 2 indicates several issues with the code, and a score of 3.36/10 - significantly lower than that of Application 1. Many of these issues relate to code readability and formatting; however, a number of these issues have potential security implications as well.

Several unused imports are used throughout the code, and these imports should either be avoided or used for aforementioned reasons, such as performance overload and vulnerability exposure. All imports are identified by E0401. These import errors are the highest severity amongst issues identified by Pylint.

Another issue occurring on several occasions in Application 2 are unreachable code errors, represented by code W0101. Unreachable code should be avoided, as it creates clutter in the program, increases code complexity, and opens up security vulnerabilities to unexecuted code. Unreachable code can also in some instances lead to unexpected behavior in the design; eliminating these codes will increase the efficiency, and possibly security, of the application (ChipVerify).

Pyflakes

The Pyflakes scan reiterates Pylint findings, and outlines every instance where an unused import in the application has been made. Only import errors were generated by this scan. Again, reviewing unused imports, and ensuring imports are either performed correctly or avoided when unnecessary will ensure better performance and a lower security risk in the application.

Wapiti

The Wapiti output for Application 2 indicates four vulnerabilities in total. Two vulnerabilities are associated with HTTP secure headers, specifically the lack of X-XSS protection, a shared commonality with Application 1. Again, these vulnerabilities can be ignored within reason, as implementing X-XSS protection can in fact open up new cross scripting vulnerabilities.

The other two instances of vulnerabilities are the HttpOnly flag not set in two cookies, which are also shared vulnerabilities with Application 1. Recommendations for this vulnerability are listed in the Wapiti scan for Application 1. The next section summarizes some of the recommendations covered above, and a few more to consider.

Recommendations

Based on the Bandit scans of each application, certain steps may be taken to ensure security vulnerabilities are lowered. Firstly, try-except-passes are present in both applications and should be avoided; rather than employing passes, developers may consider adding conditions for failure, and implementing graceful degradation systems. “exec” commands should also be evaluated to prevent arbitrary code execution. In Application 2, several instances of hardcoded passwords, hardcoded temporary directories, and hardcoded SQL expressions are detected. To avoid hardcoding, implement input validation at every stage which requires input for the program to proceed.

It is highly recommended for developers to review the entirety of the Pylint scan results, documented in the appendices section, to review the codes of each application for usability and security. Avoiding unused imports of models, or ensuring imports are formatted correctly, will provide a reliable, fail-proof code. In addition, unreachable code should be reviewed and avoided, unless it is intentionally included as a honeypot in source code to deter attackers.

While the Pyflakes scans do not provide many more insights into vulnerabilities of each application when compared to each Pylint scan, they reiterate the importance of avoiding unused imports, lowering code complexity, and maintaining efficiency. Addressing unused imports would resolve a majority of issues associated with both applications.

The insights generated by the Wapiti tool effectively calls for including the Httponly flag in cookies to prevent client-side scripting attacks. While X-XSS protection may not be necessary, including some measure to protect against cross-scripting attacks, such as input validation, will ensure a lower likelihood of such attacks being attempted.

Methodology

Four tools were run on each application evaluated. These tools were Bandit, Wapiti, Pylint, and Pyflakes. Bandit was used to gather initial information about code vulnerabilities, by testing for common security issues associated with the source codes of each application. Pylint was used to test the code against conventional Python standards, and point to potentially insecure aspects of each application. Pyflakes was then used to verify and expand upon information generated by Bandit, and provide some additional security challenges to be considered. Finally, since previous tools were used to examine source code, Wapiti was used to scan deployed web application webpages. The output of this query was summarized in an HTML report. The primary focus of this investigation was evaluating source code, and Wapiti does not evaluate source code - rather, it performs "black-box" scans, scanning a deployed web application's webpages for scripts and fonts where data can be injected. Wapiti injects payloads in order to determine whether a web page is vulnerable; these insights can be used by developers to secure the application further (Gomez). Hence, while not the primary focus of this investigation, recommendations and findings from Wapiti were also included to provide holistic security insights into each application.

Bandit

For both applications evaluated, the first command run was a simple Bandit scan:

```
bandit -r lets-be-bad-guys
```

```
bandit -r vulpy
```

The results generated three levels of severity in potential code vulnerabilities: low severity, medium severity, and high severity. The results are, of course, interpreted in the

"Application findings" sections, with results from each code scan presented in respective appendices (1.1, 2.1).

Pyflakes

The Pyflakes project evaluates Python source code, and was used to verify - and possibly expand upon - information gathered from the initial Bandit scan. Pyflakes primarily checks for simpler errors, such as missing imports and references of undefined names (Buildbot). The command used to run Pyflakes3 to analyze each application was as follows:

```
pyflakes3 lets-be-bad-guys
```

```
pyflakes3 vulpy
```

Pylint

The second tool utilized was Pylint, a static code analyser for Python. Pylint evaluates code for errors, and compliance with the PEP 8 Python coding conventions. The tool also looks for "code smells," surface-level indications of deeper problems existing in a system (Fowler).

Since the tool is not pre-installed in Kali Linux, contrary to Wapiti, the following command was executed to install Pylint:

```
(sudo) pip install pylint
```

To see all options, the command `pylint` can be executed on its own. The queries used to evaluate each application using Pylint were as follows:

```
pylint lets-be-bad-guys
```

```
pylint vulpy
```

The output of the Pylint query is included in Appendices 1.2 and 2.2.

Wapiti

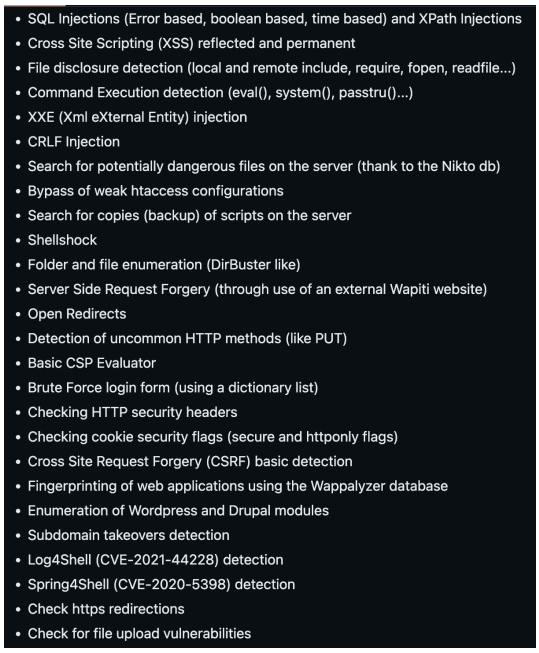
The final tool used was Wapiti. This tool is a web application vulnerability scanner, which performs a general audit for web apps. The scan took over half an hour to complete for Application 1, and over an hour on Application 2. The tool did not identify as many vulnerabilities as initially assumed given the duration of each scan. However, some useful insights were generated, resulting in more vulnerabilities being found and reported.

In order to utilize the Wapiti tool, the following query was used (note the tool requires a complete URL with a protocol scheme):

```
wapiti -v2 -u https://www.github.com/mpirnat/lets-be-bad-guys
```

```
Wapiti -v2 -u https://www.github.com/fportnatier/vulpy
```

After each scan was complete, from start to finish, the tool generated an HTML report included in Appendices 1.4 and 2.4, covering the vulnerabilities found for each of the following attacks (github.com):

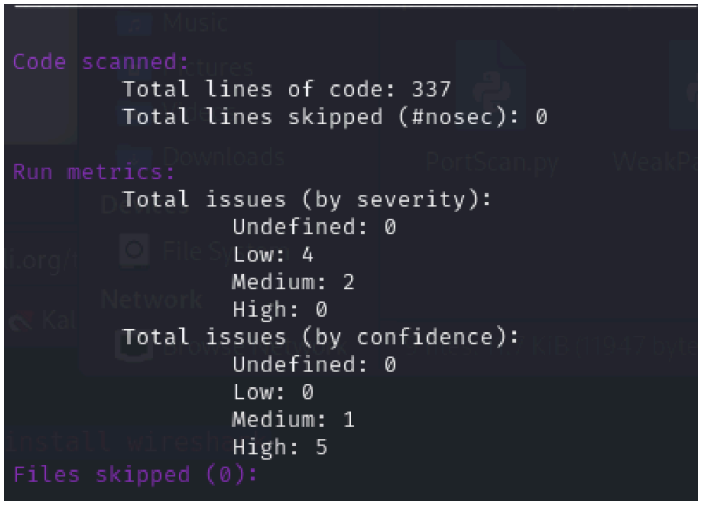


Appendices

Appendix 1.1: Application 1 Bandit results

```
┌──(kali㉿kali)-[~/bandit]
└─$ bandit -r lets-be-bad-guys
[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 3.11.6
Run started:2024-02-26 06:12:30.925820

Test results:
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'h++j
szpm)i@p%ay_b=cp#()^od!qns14)h%@qm3)p=cuo+st^a'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b105_hardco
ded_password_string.html
   Location: lets-be-bad-guys/badguys/settings.py:90:13
89      # Make this unique, and don't share it with anybody.
90      SECRET_KEY = 'h++jszpm)i@p%ay_b=cp#()^od!qns14)h%@qm3)p=cuo+st^a'
91

--------------------------------------------------
>> Issue: [B110:try_except_pass] Try, Except, Pass detected.
   Severity: Low   Confidence: High
   CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b110_try_ex
cept_pass.html
   Location: lets-be-bad-guys/badguys/vulnerable/views.py:65:8
64                  os.unlink('p0wned.txt')
65              except:
66                  pass
67

--------------------------------------------------
>> Issue: [B102:exec_used] Use of exec detected.
   Severity: Medium   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b102_exec_u
sed.html
   Location: lets-be-bad-guys/badguys/vulnerable/views.py:72:12
71                  # Try it the Python 3 way...
```

```
>> Issue: [B102:exec_used] Use of exec detected.
   Severity: Medium   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b102_exec_u
sed.html
   Location: lets-be-bad-guys/badguys/vulnerable/views.py:76:16
75                  try:
76                      exec(base64.decodestring(first_name))
77                  except:

--------------------------------------------------
>> Issue: [B110:try_except_pass] Try, Except, Pass detected.
   Severity: Low   Confidence: High
   CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b110_try_ex
cept_pass.html
   Location: lets-be-bad-guys/badguys/vulnerable/views.py:77:12
76                      exec(base64.decodestring(first_name))
77                  except:
78                      pass
79              except:

--------------------------------------------------
>> Issue: [B110:try_except_pass] Try, Except, Pass detected.
   Severity: Low   Confidence: High
   CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b110_try_ex
cept_pass.html
   Location: lets-be-bad-guys/badguys/vulnerable/views.py:79:8
78                      pass
79              except:
80                  pass
81

--------------------------------------------------
```



Appendix 1.2: Application 1 Pylint results

For brevity, a sample of results are shown below, along with the generated score.

```
┌──(kali㉿kali)-[~]
└─$ pylint lets-be-bad-guys
************* Module manage
lets-be-bad-guys/manage.py:1:0: C0114: Missing module docstring (missing-modu
le-docstring)
************* Module badguys.wsgi
lets-be-bad-guys/badguys/wsgi.py:23:0: C0413: Import "from django.core.wsgi i
mport get_wsgi_application" should be placed at the top of the module (wrong-
import-position)
************* Module badguys.settings
lets-be-bad-guys/badguys/settings.py:21:0: C0301: Line too long (101/100) (li
ne-too-long)
lets-be-bad-guys/badguys/settings.py:1:0: C0114: Missing module docstring (mi
ssing-module-docstring)
lets-be-bad-guys/badguys/settings.py:180:0: C0116: Missing function or method
 docstring (missing-function-docstring)
************* Module badguys.urls
lets-be-bad-guys/badguys/urls.py:1:0: C0114: Missing module docstring (missin
g-module-docstring)
lets-be-bad-guys/badguys/urls.py:1:0: E0611: No name 'patterns' in module 'dj
ango.conf.urls' (no-name-in-module)
lets-be-bad-guys/badguys/urls.py:20:0: C0413: Import "import badguys.vulnerab
le.views as exercises" should be placed at the top of the module (wrong-impor
t-position)
lets-be-bad-guys/badguys/urls.py:1:0: W0611: Unused patterns imported from dj
ango.conf.urls (unused-import)
lets-be-bad-guys/badguys/urls.py:1:0: W0611: Unused include imported from dja
ngo.conf.urls (unused-import)
************* Module badguys.vulnerable.tests
lets-be-bad-guys/badguys/vulnerable/tests.py:11:0: C0115: Missing class docst
ring (missing-class-docstring)
************* Module badguys.vulnerable.models
lets-be-bad-guys/badguys/vulnerable/models.py:1:0: C0114: Missing module docs
tring (missing-module-docstring)
lets-be-bad-guys/badguys/vulnerable/models.py:1:0: W0611: Unused models impor
ted from django.db (unused-import)
************* Module badguys.vulnerable.views
lets-be-bad-guys/badguys/vulnerable/views.py:194:0: C0305: Trailing newlines
(trailing-newlines)
lets-be-bad-guys/badguys/vulnerable/views.py:1:0: C0114: Missing module docst
ring (missing-module-docstring)
```

```
lets-be-bad-guys/badguys/vulnerable/views.py:128:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:143:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:144:4: W0719: Raising too genera
l exception: Exception (broad-exception-raised)
lets-be-bad-guys/badguys/vulnerable/views.py:149:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:149:19: W0613: Unused argument '
request' (unused-argument)
lets-be-bad-guys/badguys/vulnerable/views.py:155:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:165:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:176:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:181:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:191:0: C0116: Missing function o
r method docstring (missing-function-docstring)
lets-be-bad-guys/badguys/vulnerable/views.py:5:0: W0611: Unused reverse impor
ted from django.core.urlresolvers (unused-import)

-------------------------------------
Your code has been rated at 5.79/10
```

Appendix 1.3: Application 1 Pyflakes results

```
┌──(kali㉿kali)-[~]
└─$ pyflakes3 lets-be-bad-guys
lets-be-bad-guys/badguys/urls.py:1:1: 'django.conf.urls.patterns' imported but unused
lets-be-bad-guys/badguys/urls.py:1:1: 'django.conf.urls.include' imported but unused
lets-be-bad-guys/badguys/vulnerable/models.py:1:1: 'django.db.models' imported but unused
lets-be-bad-guys/badguys/vulnerable/views.py:5:1: 'django.core.urlresolvers.reverse' imported but unused
lets-be-bad-guys/badguys/vulnerable/views.py:58:5: local variable 'msg' is assigned to but never used
```

Appendix 1.4: Application 1 Wapiti HTML report:

Summary

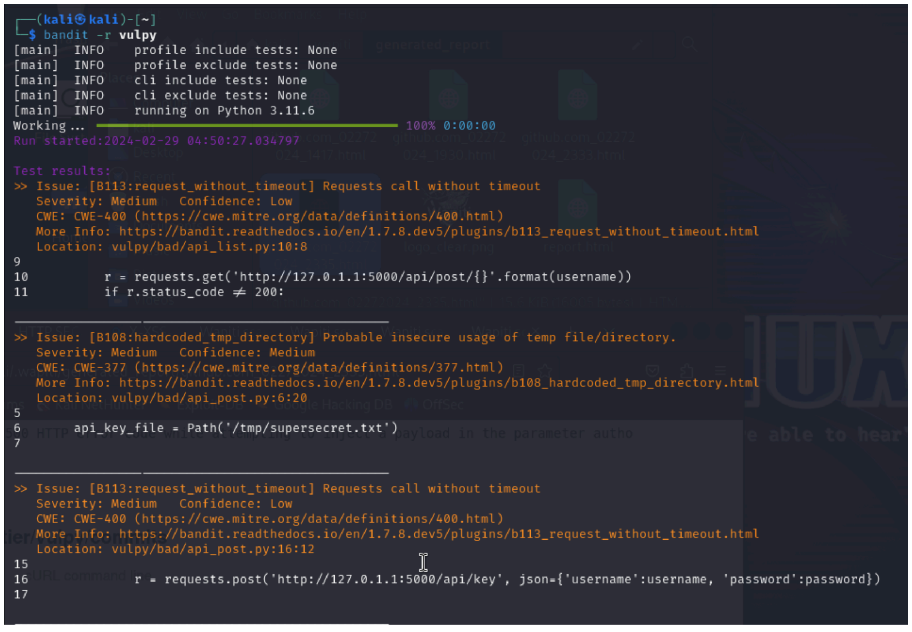
Category	Number of vulnerabilities found
Backup file	0
Blind SQL Injection	0
Weak credentials	0
CRLF Injection	0
Content Security Policy Configuration	0
Cross Site Request Forgery	0
Potentially dangerous file	0
Command execution	0
Path Traversal	0
Htaccess Bypass	0
HTTP Secure Headers	11
HttpOnly Flag cookie	11

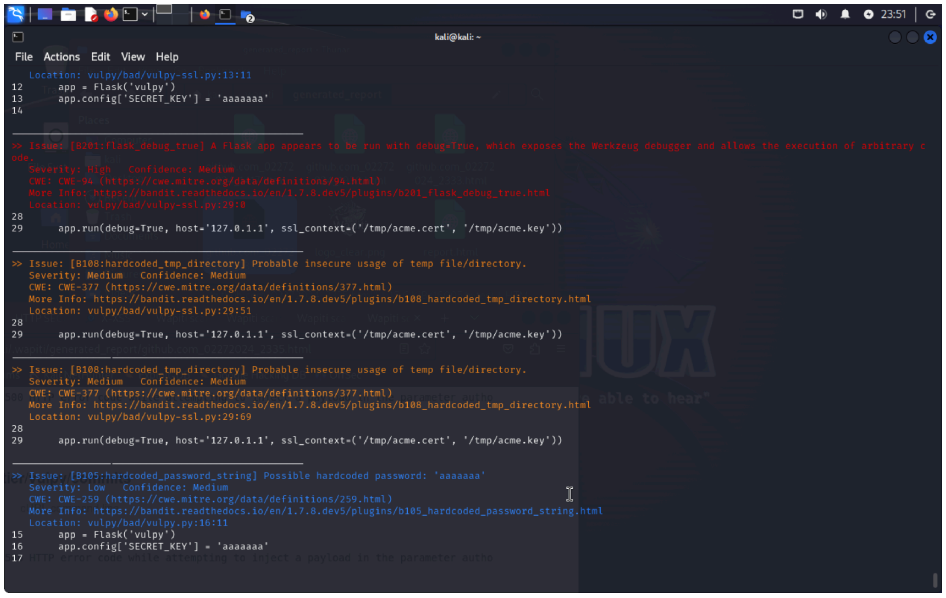
Wapiti

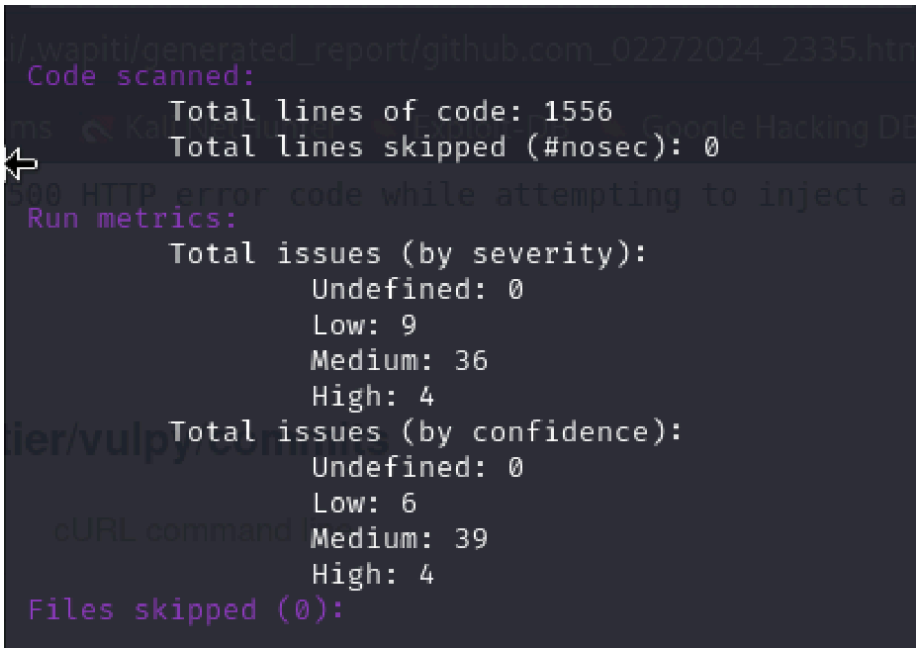
Secure Flag cookie	0
SQL Injection	0
Server Side Request Forgery	0
Cross Site Scripting	0
XML External Entity	0
Internal Server Error	2
Resource consumption	0
Fingerprint web technology	0

Appendix 2.1: Application 2 Bandit results

Not all output is included due to a large number of issues; a sample of results are included.

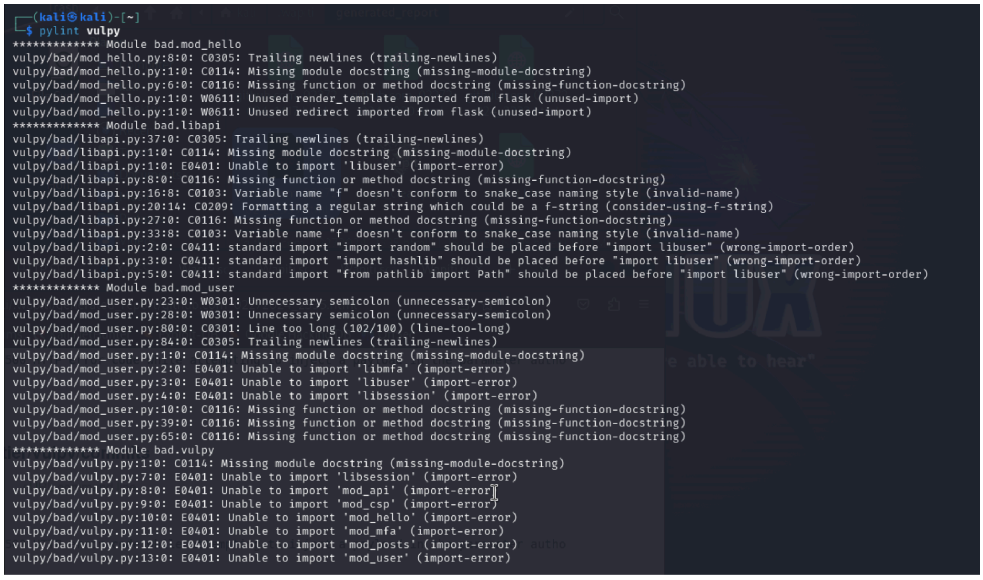



kali@kali: ~
File Actions Edit View Help
Location: vulpy/bad/vulpy-ssl.py:13:11
12 app = Flask('vulpy')
13 app.config['SECRET_KEY'] = 'aaaaaaa'
14
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with debug=True, which exposes the Werkzeug debugger and allows the execution of arbitrary code.
Severity: High Confidence: Medium
CWE: CWE-94 (https://cwe.mitre.org/data/definitions/94.html)
More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b201_flask_debug_true.html
Location: vulpy/bad/vulpy-ssl.py:29:0
28
29 app.run(debug=True, host='127.0.1.1', ssl_context=('/tmp/acme.cert', '/tmp/acme.key'))
>> Issue: [B108:hardcoded_tmp_directory] Probable insecure usage of temp file/directory.
Severity: Medium Confidence: Medium
CWE: CWE-377 (https://cwe.mitre.org/data/definitions/377.html)
More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b108_hardcoded_tmp_directory.html
Location: vulpy/bad/vulpy-ssl.py:29:51
28
29 app.run(debug=True, host='127.0.1.1', ssl_context=('/tmp/acme.cert', '/tmp/acme.key'))
>> Issue: [B108:hardcoded_tmp_directory] Probable insecure usage of temp file/directory.
Severity: Medium Confidence: Medium
CWE: CWE-377 (https://cwe.mitre.org/data/definitions/377.html)
More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b108_hardcoded_tmp_directory.html
Location: vulpy/bad/vulpy-ssl.py:29:69
28
29 app.run(debug=True, host='127.0.1.1', ssl_context=('/tmp/acme.cert', '/tmp/acme.key'))
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'aaaaaaa'
Severity: Low Confidence: Medium
CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
More Info: https://bandit.readthedocs.io/en/1.7.8.dev5/plugins/b105_hardcoded_password_string.html
Location: vulpy/bad/vulpy.py:16:11
15 app = Flask('vulpy')
16 app.config['SECRET_KEY'] = 'aaaaaaa'
17


Code scanned:
Total lines of code: 1556
Total lines skipped (#nosec): 0
Run metrics:
Total issues (by severity):
Undefined: 0
Low: 9
Medium: 36
High: 4
Total issues (by confidence):
Undefined: 0
Low: 6
Medium: 39
High: 4
Files skipped (0):

Appendix 2.2: Application 2 Pylint results

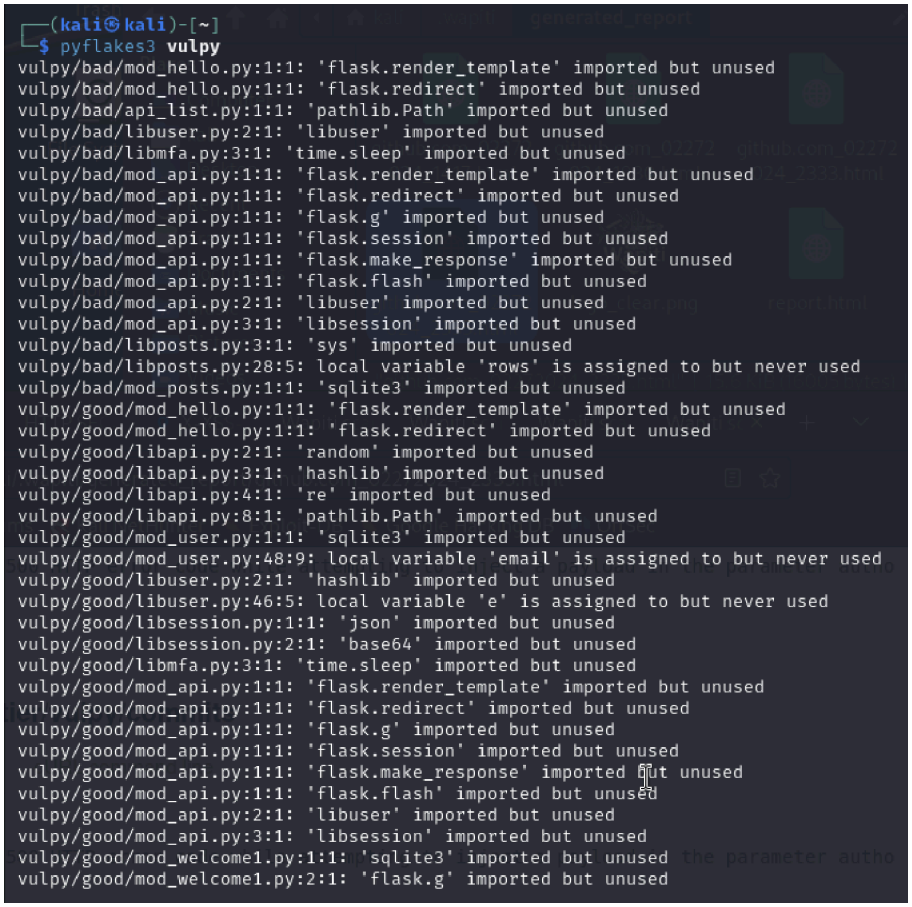
A sample of the results is included



```
┌──(kali㉿kali)-[~]
└─$ pylint vulpy
************* Module bad.mod_hello
vulpy/bad/mod_hello.py:8:0: C0305: Trailing newlines (trailing-newlines)
vulpy/bad/mod_hello.py:1:0: C0114: Missing module docstring (missing-module-docstring)
vulpy/bad/mod_hello.py:6:0: C0116: Missing function or method docstring (missing-function-docstring)
vulpy/bad/mod_hello.py:1:0: W0611: Unused render_template imported from flask (unused-import)
vulpy/bad/mod_hello.py:1:0: W0611: Unused redirect imported from flask (unused-import)
************* Module bad.libapi
vulpy/bad/libapi.py:37:0: C0305: Trailing newlines (trailing-newlines)
vulpy/bad/libapi.py:1:0: C0114: Missing module docstring (missing-module-docstring)
vulpy/bad/libapi.py:1:0: E0401: Unable to import 'libuser' (import-error)
vulpy/bad/libapi.py:8:0: C0116: Missing function or method docstring (missing-function-docstring)
vulpy/bad/libapi.py:16:8: C0103: Variable name "f" doesn't conform to snake_case naming style (invalid-name)
vulpy/bad/libapi.py:20:14: C0209: Formatting a regular string which could be a f-string (consider-using-f-string)
vulpy/bad/libapi.py:27:0: C0116: Missing function or method docstring (missing-function-docstring)
vulpy/bad/libapi.py:33:8: C0103: Variable name "f" doesn't conform to snake_case naming style (invalid-name)
vulpy/bad/libapi.py:2:0: C0411: standard import "import random" should be placed before "import libuser" (wrong-import-order)
vulpy/bad/libapi.py:3:0: C0411: standard import "import hashlib" should be placed before "import libuser" (wrong-import-order)
vulpy/bad/libapi.py:5:0: C0411: standard import "from pathlib import Path" should be placed before "import libuser" (wrong-import-order)
************* Module bad.mod_user
vulpy/bad/mod_user.py:23:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
vulpy/bad/mod_user.py:28:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
vulpy/bad/mod_user.py:80:0: C0301: Line too long (102/100) (line-too-long)
vulpy/bad/mod_user.py:84:0: C0305: Trailing newlines (trailing-newlines)
vulpy/bad/mod_user.py:1:0: C0114: Missing module docstring (missing-module-docstring)
vulpy/bad/mod_user.py:2:0: E0401: Unable to import 'libmfa' (import-error)
vulpy/bad/mod_user.py:3:0: E0401: Unable to import 'libuser' (import-error)
vulpy/bad/mod_user.py:4:0: E0401: Unable to import 'libsession' (import-error)
vulpy/bad/mod_user.py:10:0: C0116: Missing function or method docstring (missing-function-docstring)
vulpy/bad/mod_user.py:39:0: C0116: Missing function or method docstring (missing-function-docstring)
vulpy/bad/mod_user.py:65:0: C0116: Missing function or method docstring (missing-function-docstring)
************* Module bad.vulpy
vulpy/bad/vulpy.py:1:0: C0114: Missing module docstring (missing-module-docstring)
vulpy/bad/vulpy.py:7:0: E0401: Unable to import 'libsession' (import-error)
vulpy/bad/vulpy.py:8:0: E0401: Unable to import 'mod_api' (import-error)
vulpy/bad/vulpy.py:9:0: E0401: Unable to import 'mod_csp' (import-error)
vulpy/bad/vulpy.py:10:0: E0401: Unable to import 'mod_hello' (import-error)
vulpy/bad/vulpy.py:11:0: E0401: Unable to import 'mod_mfa' (import-error)
vulpy/bad/vulpy.py:12:0: E0401: Unable to import 'mod_posts' (import-error)
vulpy/bad/vulpy.py:13:0: E0401: Unable to import 'mod_user' (import-error)
```

```
@mod_user.route('/create', methods=['GET', 'POST'])
def do_create():

    session.pop('username', None)

    if request.method == 'POST':

        username = request.form.get('username')
        password = request.form.get('password')
        #email = request.form.get('password') (duplicate-code)
vulpy/utils/crack-hash.py:1:0: R0801: Similar lines in 2 files
==bad.mod_api:[8:14]
==good.mod_api:[8:14]
mod_api = Blueprint('mod_api', __name__, template_folder='templates')

key_schema = {
    "type" : "object",
    "required": [ "username", "password" ],
    "properties" : { (duplicate-code)
vulpy/utils/crack-hash.py:1:0: R0801: Similar lines in 2 files
==bad.libmfa:[16:26]
==good.libuser:[50:60]
        return False

def mfa_disable(username):

    conn = sqlite3.connect('db_users.sqlite')
    conn.set_trace_callback(print)
    conn.row_factory = sqlite3.Row
    c = conn.cursor()
 (duplicate-code)

------------------------------------------------------------------
Your code has been rated at 3.36/10 (previous run: 3.36/10, +0.00)
```

Appendix 2.3: Application 2 Pyflakes results



Appendix 2.4: Application 2 Wapiti HTML report

Category	Number of vulnerabilities found
Backup file	0
Blind SQL Injection	0
Weak credentials	0
CRLF Injection	0
Content Security Policy Configuration	0
Cross Site Request Forgery	0
Potentially dangerous file	0
Command execution	0
Path Traversal	0
Htaccess Bypass	0
HTTP Secure Headers	2
HttpOnly Flag cookie	2
Open Redirect	0
Secure Flag cookie	0

SQL Injection	0
Server Side Request Forgery	0
Cross Site Scripting	0
XML External Entity	0
Internal Server Error	0
Resource consumption	0
Fingerprint web technology	0

Bibliography

Bandit Developers. (n.d.). *B102: Exec_used*. Bandit . https://bandit.readthedocs.io/en/latest/plugins/b102_exec_used.html

Gomez, Brian. “Complete Guide to Using Wapiti Web Vulnerability Scanner to Keep Your Web Applications & Websites Secure.” *Linux Security*, Linux, 2022, https://linuxsecurity.com/features/complete-guide-to-using-wapiti-web-vulnerability-scanner-to-keep-your-web-applications-websites-secure

Shelton, Charles. “Exception Handling.” *Topic: Exception Handling*, Carnegie Mellon University, 1999, users.ece.cmu.edu/~koopman/des_s99/exceptions/.

Thalmann, Bruno. “Python-Security/PYT: A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications.” *GitHub*, GitHub, github.com/python-security/pyt.

“Unreachable Code Analysis.” *ChipVerify*, ChipVerify, www.chipverify.com/verification/unreachable-code-analysis#:~:text=Unreachable%20code%20analysis%20is%20a,dead%20code%20or%20redundant%20code.

“Wapiti-Scanner/Wapiti: Web Vulnerability Scanner Written in Python3.” *GitHub*, GitHub, github.com/wapiti-scanner/wapiti.

“X-XSS-Protection - Http: MDN.” *MDN Web Docs*, Mozilla, July 2023, developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection.