Paul Badu Yakubu

Penetration Testing

Professor Anthony D. Candeias

Lab 2: Static Code Analysis

Executive Summary

This report was made after evaluating the security of the source code of two applications being built in Python for your organization. The application is a web-based platform that allows users to interact with various features and functionalities, such as SQL queries, file access, code execution, user profiles, and more.

This report identifies several common vulnerabilities in the application, such as code injection, path traversal, cross-site scripting (XSS), security misconfiguration, sensitive data exposure, cross-site request forgery (CSRF). These vulnerabilities can compromise the confidentiality, integrity, and availability of the application and its data, and pose serious risks to the business and its customers.

This report suggests some ways to improve the security of the system, such as fixing bugs, controlling access, protecting data, and preventing attacks. It also advises to follow some long-term goals, such as reviewing code, checking access, updating software, verifying links, and educating developers and users.

"Lets-be-bad-guys" Application Findings

This application was scanned using Bandit, a static code analysis tool that detects common security issues in Python code. The tool generated a report that listed the issues, their severity, their location, and their description. The report also provided some links to more information and possible solutions. The following table summarizes the main findings of the scan:

Issue	Severity	Description
Code injection	high	The application uses exec to execute user input, which can allow attackers to run arbitrary code on the server.
Path traversal	High	The application uses open to read files based on user input, which can allow attackers to access files outside of the intended scope.
XSS	Medium	The application does not properly escape or sanitize user input, which can allow attackers to inject malicious scripts into the web pages.
Insecure direct	Medium	The application does not check the authorization of the user

object references		before accessing user profiles, which can allow attackers to access or modify other users' data.
Security misconfiguration	Low	The application does not handle exceptions properly, which can expose sensitive information or cause errors.
Sensitive data exposure	Low	The application does not use encryption or hashing to store sensitive data
Missing function level access control	High	The code does not check the authorization of the user before accessing the admin page, which can allow unauthorized users to access or modify sensitive data or functionality.
CSRF	Medium	The code does not use CSRF protection for the image upload view, which can allow attackers to perform unwanted actions on behalf of the user, such as uploading malicious images or stealing cookies.
Using known vulnerable components	Low	The code does not specify the version of Django or other components used by the application, which can expose the application to known vulnerabilities or bugs in the components.
Unvalidated redirects and forwards	Medium	The code does not validate the user input for the redirect and forward views, which can allow attackers to redirect or forward the user to malicious or phishing sites, or to other views that may compromise the security of the application.

Application "Vulpy.py" Findings

- The code registers a blueprint called `mod_hello` that handles the `/hello` endpoint. This endpoint takes a user input as a query parameter and passes it to the `render_template` function without any validation or sanitization. This could lead to a server-side template injection (SSTI) vulnerability, which allows an attacker to execute arbitrary Python code on the server by injecting Jinja2 expressions.

- The code also registers a blueprint called `mod_user` that handles the `/user` endpoint. This endpoint allows users to register, login, logout, and view their profiles. However, the code does not implement any password hashing or salting, which means that the passwords are stored in plain text in the database. This could lead to a data breach if the database is compromised, as well as a broken authentication vulnerability, which allows an attacker to guess or brute-force the passwords of other users.
- The code also registers a blueprint called `mod_posts` that handles the `/posts` endpoint. This endpoint allows users to create, edit, delete, and view posts. However, the code does not properly escape the user input before inserting it into

the database, which could lead to a SQL injection vulnerability, which allows an attacker to execute arbitrary SQL commands on the database by injecting malicious input.

- The code also registers a blueprint called `mod_mfa` that handles the `/mfa` endpoint. This endpoint allows users to enable or disable multi-factor authentication (MFA) for their accounts. However, the code does not verify the user's identity before performing the action, which could lead to a bypass of MFA, which allows an attacker to access the user's account without the second factor.
- The code also registers a blueprint called `mod_csp` that handles the `/csp` endpoint. This endpoint allows users to view the content security policy (CSP) of the application, which is read from a file called `csp.txt`. However, the code does not validate the file name or the file content, which could lead to a file inclusion vulnerability, which allows an attacker to read arbitrary files on the server by manipulating the file name.
- The code also registers a blueprint called `mod_api` that handles the `/api` endpoint. This endpoint allows users to interact with the application through a RESTful API. However, the code does not implement any authentication or authorization for the API, which could lead to an unauthorized access vulnerability, which allows an attacker to perform any action on the application through the API without any credentials.
- The code uses the Flask module to create the web application and the `pathlib` module to handle file paths. However, the code does not import any other modules or libraries that could enhance the security of the application, such as flask-wtf for CSRF protection, flask-bcrypt for password hashing, flask-login for session management, or flask-limiter for rate limiting.
- The code sets the `SECRET_KEY` configuration to a static and predictable value of `'aaaaaaa'`, which is used to sign the session cookies and the CSRF tokens. This could lead to a session hijacking vulnerability, which allows an attacker to forge or steal the session cookies of other users and impersonate them.
- The code uses the `libsession` module to load the session data from the request. However, the code does not check the validity or the expiration of the session data, which could lead to a session fixation vulnerability, which allows an attacker to force a user to use a predetermined session ID and access their session data.
- The code uses the `print` function to output the CSP value to the console. However, the code does not log any other information, such as the requests, the responses,

the errors, or the exceptions, which could lead to a lack of auditability and accountability, as well as a difficulty in debugging and troubleshooting.

- The code uses the `app.run` function to run the application in debug mode, with the host set to `127.0.1.1` and the port set to `5000`. However, the code does not specify any SSL context or certificate, which means that the application does not use HTTPS, which could lead to a man-in-the-middle vulnerability, which allows an attacker to intercept, modify, or tamper with the traffic between the client and the server.

Recommendations

Based on the findings of the scan, the following recommendations are provided to improve the security of the application. The recommendations are divided into two categories: tactical and strategic.

Tactical Recommendations

The tactical recommendations are short-term and specific actions that can be taken to address the immediate vulnerabilities and risks. They include:

- Applying patches and updates to the Python libraries and frameworks used by the application, such as Django, bleach, etc.
- Implementing access control lists (ACLs) to restrict the access and permissions of the users and roles and enforcing the principle of least privilege.
- Sanitizing and validating user input using bleach, validators, or other methods, and escaping or encoding output using html.escape, cgi.escape, or other methods.
- Encrypting and hashing sensitive data using AES, bcrypt, scrypt, or other algorithms, and storing the keys and salts securely.
- Using CSRF protection for POST requests using the @csrf_protect decorator or the csrf_token template tag and using the @csrf_exempt decorator only for views that do not require CSRF protection.
- Handling exceptions properly using try-except-finally blocks, logging errors, and displaying user-friendly messages.
- Implementing access control checks for the admin page, such as using the @user_passes_test or @permission_required decorators or checking the user.is_staff or user.is_superuser attributes.
- Using CSRF protection for the image upload view, such as using the @csrf_protect decorator or the csrf_token template tag and removing the @csrf_exempt decorator.

- Updating the components used by the application, such as specifying the latest version of Django or other libraries in the requirements.txt file or using pip to install or upgrade the packages.
- Validating the user input for the redirect and forward views, such as using the is_safe_url function or the ALLOWED_HOSTS setting to check the validity of the URLs or using the reverse function or the *urltemplate* tag to generate the URLs for the views.
- To mitigate the SSTI vulnerability, the code should validate and sanitize the user input before passing it to the render_template function, as well as enable the auto-escaping feature of Jinja2, which prevents the execution of malicious expressions1.
- To mitigate the data breach and the broken authentication vulnerabilities, the code should implement password hashing and salting using a secure algorithm, such as bcrypt, and a random salt, as well as enforce a strong password policy, such as minimum length, complexity, and expiration.
- To mitigate the SQL injection vulnerability, the code should use parameterized queries or prepared statements, which separate the user input from the SQL commands, as well as escape any special characters in the user input.

These recommendations can help mitigate the current vulnerabilities and reduce the likelihood and impact of potential attacks.

Strategic Recommendations

The strategic recommendations are long-term and general actions that can be taken to improve the security posture and culture of the organization. They include:

- Adopting security best practices, such as the OWASP Top 10, the Python security cheat sheet, and the Django security tips, and following them consistently and rigorously.
- Conducting regular code reviews and audits, both internally and externally, to identify and fix security issues and bugs, and to ensure compliance with standards and regulations.
- Implementing security testing and monitoring tools, such as Bandit, PyLint, ZAP, Nmap, etc., to scan, analyze, and report on the security of the application and its environment, and to detect and respond to any anomalies or incidents.
- Providing security training and awareness to the developers and users, to educate them on the common threats and risks, the best practices and policies, and the roles and responsibilities, and to foster a security mindset and culture.

These recommendations can help improve the security maturity and capability of the organization, and to enhance its reputation and trustworthiness.

Methodology

The methodology used to complete the assignment involved using a static code analysis tool (Bandit), a web search tool (Copilot), and some custom scripts to scan, analyze, and generate the report. The following steps describe the process:

- Step 1: Scaned the code using Bandit, a static code analysis tool that detects common security issues in Python code. The tool was run with the following command: bandit -r -f html <directory_of_sourcecode> -o bandit_report.html.
- Step 2: Analyzed the report generated by Bandit, and identified the main issues, their severity, their location, and their description. The report also provided some links to more information and possible solutions.
- Step 3: Used Copilot, a web search tool that provides relevant and useful responses, to find some resources on Python and Django security, such as articles, tutorials, guides, etc. The tool was invoked with the following query: python django security.

The methodology used automation to improve the efficiency and accuracy of the process, and to reduce the manual effort and errors. The automation was achieved by using tools and scripts that can perform tasks such as scanning, searching, generating, formatting, and checking. The following code examples demonstrate some of the use cases of automation:

References:

- Portantier, F. (2020, December 14). vulpy/bad/vulpy.py. GitHub
- Pirnat, M. (2018, October 13). lets-be-bad-guys/badguys/vulnerable. GitHub
- National Vulnerability Database. (2021, January 19). CVE-2021-21241 Detail. NIST[3]