Secure Code Review

Manual Secure Code Review

Done By: Judy Ammar Hallak

Student Id: CA/S1/5866

Date: September 16 2024

Code 1: The Insecure Code

```python
import os
import json

SECRET_KEY = "my_secret_key"

def vulnerable_function(user_input):
    os.system(f"echo {user_input}")

def insecure_logging(data):
    with open("log.txt", "a") as log_file:
        log_file.write(f"Sensitive data: {data}\n")

def unsafe_deserialization(serialized_data):
    return json.loads(serialized_data)

def insecure_file_download(filename):
    file_path = f"/downloads/{filename}"
    with open(file_path, 'rb') as f:
        return f.read()

def main():
    user_input = input("Enter a command to execute: ")
    vulnerable_function(user_input)

    sensitive_data = input("Enter sensitive data to log: ")
    insecure_logging(sensitive_data)

    serialized_data = input("Enter serialized data (JSON): ")
    data = unsafe_deserialization(serialized_data)
    print(f"Deserialized data: {data}")

    filename = input("Enter the file name to download: ")
    file_content = insecure_file_download(filename)
    print("File content downloaded.")

if __name__ == "__main__":
    main()
```

This Python script performs several operations, including executing shell commands, logging user input, deserializing JSON data, and downloading files. Though the code at first glance may seem okay and run with no errors, there are a couple of security vulnerabilities that could lead to serious issues in a real-world application. The vulnerabilities are:

1. Insufficient Input Validation
2. Logging Sensitive Information
3. Unsafe Deserialization
4. Insecure File Download Mechanism
5. Environment Variables Management
6. Lack of Rate Limiting and Authentication

1.Insufficient Input Validation

The code does not validate user input before using it in a command execution context. This is appears in lines 6-7, **vulnerable_function(user_input)** where **user_input** is passed directly to **os.system()**.
This vulnerability allows an attacker to execute arbitrary commands on the server by injecting malicious input. For example, if a user inputs ; rm -rf /, it could lead to severe damage, deleting critical system files.

2. Logging Sensitive Information

Sensitive data is logged without any form of encryption or masking. This can expose sensitive information if log files are accessed by unauthorized users. This appears in the **insecure_logging(data)** function (line 9)
Logging sensitive user data (like passwords or personal information) in plain text allows attackers to read logs and gain access to private information.

3. Unsafe Deserialization

The **unsafe_deserialization** (line 13) function uses **json.loads** to deserialize user-supplied data **(serialized_data)** directly without any validation. If an attacker crafts malicious JSON data, it could lead to arbitrary code execution or manipulation of the application's state. To mitigate this vulnerability, input should be validated, and potentially unsafe data structures should be avoided in deserialization.

4. Insecure File Download Mechanism

The **insecure_file_download(line 16)** function constructs a file path from user input **(filename)** without validation, which could lead to directory traversal attacks. An attacker could specify paths like ../../etc/passwd, allowing them to access sensitive files outside the intended download directory. To protect against this, proper input validation should be implemented, ensuring that the filename only allows safe, expected input and sanitizing any dangerous characters or patterns.

5. Environment Variables Management

The **SECRET_KEY(line 4)** is hard-coded directly into the source code, exposing sensitive information within the application. This practice can lead to security risks, especially if the code is shared or stored in version control systems. If an attacker gains access to the source code, they can easily find the key and exploit it. To enhance security, sensitive information like keys and configuration settings should be stored in environment variables or secure configuration management systems, which can be accessed at runtime without hardcoding them in the source.

6. Lack of Rate Limiting and Authentication

The main function(line 37) provides direct access to execute commands without any authentication or rate limiting. This design exposes the application to potential abuse, including Denial of Service (DoS) attacks, where an attacker could continuously flood the application with requests. Implementing user authentication and applying rate limiting can help protect against abuse and ensure that only legitimate users can access critical functionalities.

Code 2: The Secure Code

```
import os
import json
import logging
from werkzeug.security import generate_password_hash, check_password_hash

logging.basicConfig(level=logging.INFO, filename='app.log',
                    format='%(asctime)s - %(levelname)s - %(message)s')

SECRET_KEY = os.getenv("SECRET_KEY")

def secure_function(user_input):
    safe_commands = ['list', 'show']
    if user_input in safe_commands:
        os.system(f"echo {user_input}")
    else:
        logging.warning("Unauthorized command attempted.")

def secure_logging(data):
    logging.info("User performed an action.")

def safe_deserialization(serialized_data):
    try:
        data = json.loads(serialized_data)
        if not isinstance(data, dict):
            raise ValueError("Invalid data format.")
        return data
    except json.JSONDecodeError:
        logging.error("Failed to decode JSON.")
        return {}
```

```python
def secure_file_download(filename):
    safe_directory = "/downloads/"
    if '..' in filename or filename.startswith('/'):
        logging.warning("Attempted directory traversal.")
        return "Invalid file name."

    file_path = os.path.join(safe_directory, filename)
    if os.path.isfile(file_path):
        with open(file_path, 'rb') as f:
            return f.read()
    else:
        logging.warning("File not found.")
        return "File not found."

def main():
    user_input = input("Enter a command to execute (list/show): ")
    secure_function(user_input)

    sensitive_data = input("Enter some data: ")
    secure_logging(sensitive_data)

    serialized_data = input("Enter serialized data (JSON): ")
    data = safe_deserialization(serialized_data)
    print(f"Deserialized data: {data}")

    filename = input("Enter the file name to download: ")
    file_content = secure_file_download(filename)
    print("File content downloaded." if isinstance(file_content, bytes) else file_content)

if __name__ == "__main__":
    main()
```

How The Vulnerabilities Were Addressed

1. The **secure_function** now validates user input against a list of **safe_commands** (i.e., list and show). This prevents unauthorized commands from being executed. If the input doesn't match one of the allowed commands, a warning is logged instead of executing the command.
2. In the **secure_logging** function, sensitive user data is not logged. Instead, a generic log message indicates that a user action has occurred. This minimizes the risk of exposing sensitive information in logs, protecting user privacy.
3. The **safe_deserialization** function includes error handling to catch **JSONDecodeError**. It also checks whether the deserialized data is of the expected type (a dictionary). If the format is invalid or if deserialization fails, an error is logged, and the function returns an empty dictionary instead of executing potentially unsafe code.
4. In **secure_file_download**, checks are added to prevent directory traversal attacks. The function verifies that the filename does not contain .. or start with /, which would allow accessing files outside the intended directory. Additionally, it logs a warning for unauthorized attempts and checks if the file exists before attempting to read it, logging a warning if it is not found.
5. The **SECRET_KEY** is now sourced from environment variables using **os.getenv("SECRET_KEY")**. This practice enhances security by preventing sensitive information from being hard-coded into the source code, reducing the risk of exposure if the code is shared or accessed by unauthorized users.
6. Though not implemented in this snippet, best practices for securing the application would involve adding authentication checks before executing sensitive functions, along with rate limiting to prevent abuse. Implementing these would further enhance the security posture of the application.

Conclusion

In the updated code, several critical security vulnerabilities have been effectively addressed, enhancing the overall safety of the application. By implementing input validation, secure logging practices, and safe deserialization methods, the code minimizes the risk of unauthorized access and data exposure. Additionally, the use of environment variables for sensitive information and safeguards against directory traversal in file downloads further solidify the application's defense against potential attacks. These improvements not only protect user data but also foster trust in the application’s reliability and security. Ongoing vigilance and regular security audits remain essential to maintaining a robust security posture in software development. These improvements collectively strengthen the security of the application by addressing the previously identified vulnerabilities.