|  |
| --- |
| Gopher Industries – Nutrihelp |
| Vulnerability Scanner Report |
|  |

|  |
| --- |
| Lachlan Adams  9-4-2024 |

Statement of Intent

Overview

This document has been developed to assist students within Nutrihelp after their use of “VulnerabilityScannerV1.0”. This report entails all the vulnerabilities the scanner is looking for and entails how the attacks are carried out, the consequences and how to patch this vulnerability. This document has been written with the presumption that readers know how to code based on their submissions and changes to the Nutrihelp API.

Acronyms and Abbreviations

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| OS | Operating System |
| XXS | Cross-Site-Scripting |
| SQL | Structured Query Language |
| Hijack | When an attacker takes control of the session, device, webserver etc |
| Session | In the context of this report, a session is a web browser tab or active online website page where a user is interaction or accessing through an internet browser. |
| RNG | Random Number Generator |

Document History

|  |  |  |  |
| --- | --- | --- | --- |
| **Dates** | **Version** | **Author** | **Comments** |
| 20/08/2024 | 1 | Lachlan Adams | 5 vulnerability patterns identified |
| 04/09/2024 | 2 | Lachlan Adams | Updated to 10 vulnerabilities (Previously 5), need to add potential issues, defences and code identification |
| 25/09/2024 | 3 | Lachlan Adams | Cleaned up document for final submission |

Table of Contents

[**SQL Injection** 5](#_Toc178195950)

[How is this attack found via the Vulnerability Scanner? 5](#_Toc178195951)

[Potential issues if Vulnerability / Attack was successful: 5](#_Toc178195952)

[How to defend against this Vulnerability / Attack: 5](#_Toc178195953)

[**Cross-Site Scripting** 5](#_Toc178195954)

[How is this attack found via the Vulnerability Scanner? 6](#_Toc178195955)

[Potential issues if Vulnerability / Attack was successful: 6](#_Toc178195956)

[How to defend against this Vulnerability / Attack: 6](#_Toc178195957)

[**Command Injection** 6](#_Toc178195958)

[How is this attack found via the Vulnerability Scanner? 6](#_Toc178195959)

[Potential issues if Vulnerability / Attack was successful: 6](#_Toc178195960)

[How to defend against this Vulnerability / Attack: 7](#_Toc178195961)

[**Insecure File Handling** 7](#_Toc178195962)

[How is this attack found via the Vulnerability Scanner? 7](#_Toc178195963)

[Potential issues if Vulnerability / Attack was successful: 7](#_Toc178195964)

[How to defend against this Vulnerability / Attack: 7](#_Toc178195965)

[**Insecure File Upload** 7](#_Toc178195966)

[How is this attack found via the Vulnerability Scanner? 8](#_Toc178195967)

[Potential issues if Vulnerability / Attack was successful: 8](#_Toc178195968)

[How to defend against this Vulnerability / Attack: 8](#_Toc178195969)

[**Code Injection** 8](#_Toc178195970)

[How is this attack found via the Vulnerability Scanner? 8](#_Toc178195971)

[Potential issues if Vulnerability / Attack was successful: 8](#_Toc178195972)

[How to defend against this Vulnerability / Attack: 9](#_Toc178195973)

[**Directory Movement – Transversal Attack** 9](#_Toc178195974)

[How is this attack found via the Vulnerability Scanner? 9](#_Toc178195975)

[Potential issues if Vulnerability / Attack was successful: 9](#_Toc178195976)

[How to defend against this Vulnerability / Attack: 9](#_Toc178195977)

[**Insecure Token Generation** 9](#_Toc178195978)

[How is this attack found via the Vulnerability Scanner? 10](#_Toc178195979)

[Potential issues if Vulnerability / Attack was successful: 10](#_Toc178195980)

[How to defend against this Vulnerability / Attack: 10](#_Toc178195981)

[**Permission Level** 10](#_Toc178195982)

[How is this attack found via the Vulnerability Scanner? 10](#_Toc178195983)

[Potential issues if Vulnerability / Attack was successful: 11](#_Toc178195984)

[How to defend against this Vulnerability / Attack: 11](#_Toc178195985)

[**Redirects** 11](#_Toc178195986)

[How is this attack found via the Vulnerability Scanner? 11](#_Toc178195987)

[Potential issues if Vulnerability / Attack was successful: 11](#_Toc178195988)

[How to defend against this Vulnerability / Attack: 12](#_Toc178195989)

[**Summary** 12](#_Toc178195990)

## **SQL Injection**

This is a web-based attack, where a user manipulates the SQL queries via entering in tehri own malicious code. This is often performed through any input field on a website. For example, if a user is prompted to enter in their username, this input field could be used to coordinate an attack. This allows the attacker to manipulate the query’s execution and lead to further damage or attacks. Dependant on which area was breached and the extent of their malicious code, the attacker may be able to manipulate the database or gain unauthorised access to the system.

### How is this attack found via the Vulnerability Scanner?

“"Sql\_Injection": re.compile(r'\.query\s\*\(.\*\+.\*\)')“ this expression is used to search through the selected file and locate any areas where a user is able to enter something into a input field. Without proper data sanitisation, a user would be able to input SQL code to the input field and begin to breach the application.

### Potential issues if Vulnerability / Attack was successful:

**Privilege Escalation:** A hacker can begin to elevate their privilege. Which could result in access to different areas of the application. This their new privileges, they could make changes to the web server and take control.

**Data Corruption:** An attacker could intentionally cause a system corruption. Where they damage files or areas of the system and cause damage to the integrity and lifespan.

### How to defend against this Vulnerability / Attack:

**Data Sanitisation:** Sanitise all data that is collected from users.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Pre configured parameters and allowed characters**: Allowing only a specific range of characters based on the exact needs for the input field. And example of this is within the name field, numbers and special characters will not be necessary. These should be blocked by default.

## **Cross-Site Scripting**

This is a vulnerability which can be taken advantage of through web application. This vulnerability can be exploited when a web application displays a user’s input without the correct data sanitisation and input validation. This allows for attackers to deploy their malicious code and potentially hijack the session or interfere with the website. An example of this attack is a user could enter in <script>alert(‘The session has been hijacked!!!!! Malware uploaded’)</script>. Which would cause a popup window with “The session has been hijacked!!!!! Malware uploaded” on the user’s screen.

### How is this attack found via the Vulnerability Scanner?

‘ "XSS": re.compile(r'res\.send\s\*\(.\*\+.\*\)') ‘ this line of code is used to detect XSS vulnerabilities within the JavaScript files. This identifies areas of code that an XSS attack can occur. Due to a user’s input not being properly sanitised or without the correct data validation.

### Potential issues if Vulnerability / Attack was successful:

**Session Hijacking:** The active session could be hijacked by an attacker. Which could lead to a range of issues from, unauthorised access, further attacks and damage to the integrity of the web server.

**Malware infection:** The attacker could deploy malware to the users device or the webserver. Dependant on the malware, this could corrupt files, infect other users and potentially give full access to the attacker.

### How to defend against this Vulnerability / Attack:

**Data Sanitisation:** Sanitise all data that is collected from users.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Develop Input Controls:** Development of controls that prevent users from inputting characters that they would not need for the given input field.

## **Command Injection**

As the name suggests this is another injection-based attack, which attackers commonly use if there is an open vulnerability to abuse. When there is little to no validation and poor coding from the developers. Applications can sometimes be configured in a way that user inputs are sent directly to the system. This means that an attacker is able to execute system level commands and pressure test a system in an attempt to cause a disruption or gain access.

### How is this attack found via the Vulnerability Scanner?

 ‘ "Command\_Injection": re.compile(r'exec\s\*\(.\*\+.\*\)') ‘ This line of code catches any attempt to submit data to an input field. If there is no sanitisation for “+” or other characteristics of malicious code. This vulnerability will be identified.

### Potential issues if Vulnerability / Attack was successful:

**System Control**: The hacker can gain control of the system via their malicious code. This could result in immediate damages, users losing access and potentially a ransomware attack.

**Data Integrity:** Users files and information may be compromised. Which could result in legal trouble for Nutrihelp. This is completely avoidable if proper sanitisation and validation is performed.

**System Corruption:** The system could be taken offline by the attacker

### How to defend against this Vulnerability / Attack:

**Data Sanitisation:** Sanitise all data that is collected from users.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

## **Insecure File Handling**

This is a vulnerability within the code of an application when files are uploaded, downloaded, changed, deleted or accessed. Without proper controls preconfigured in the code, attackers can explore and experiment freely based on their codes conditions. If there are little to no forms of validation, authentication or read only permissions set. An attacker can cause dame via deleting files, modifying files, uploading malicious files or accessing files they shouldn’t be able to see. This causes an issue within the systems integrity and can potentially lead to an attacker accessing files that contain sensitive data that was not meant for their eyes to see.

### How is this attack found via the Vulnerability Scanner?

‘ "insecure\_file\_handling": re.compile(r'fs\.unlink\s\*\(.\*\)') ‘ This line of code detects files where the Unlink function has been used. This is dangerous as it is used to delete files from the system. Which mean if the hacker was able to access this area. They would be able to delete files within specific sub directories.

### Potential issues if Vulnerability / Attack was successful:

**File Deletion:** Attackers could delete any files they would like too. This may include system files that are crucial to run the server.

**Manipulation:** The attacker may manipulate files or alter their directories.

### How to defend against this Vulnerability / Attack:

**Account Permissions:** Limit what each user can perform on the webserver. This may include only allowing users low permissions such as uploading and viewing files.

**File Paths:** Set permissions to not allow any changers to file paths and restrict access to viewing the locations of files stored on the server.

## **Insecure File Upload**

Similar to the vulnerability identified above. This vulnerability solely focuses on the file upload area. When an application or webserver allows users to upload files, there should be validation that is performed on the system to ensure those files cannot be executed on the system. Attacks may submit files that cause harm to the system via the input malicious code or malware within them. Dependant on the systems validations, an attack may be able to gain access to the systems, completely wipe it or take control.

### How is this attack found via the Vulnerability Scanner?

‘ "insecure\_file\_upload": re.compile(r'multer\s\*\(\s\*{.\*dest.\*}\s\*\)') ‘ This line of code is responsible for detecting an insecure file upload. It searches for code that does not apply security checks or validation in regards to file uploads.

### Potential issues if Vulnerability / Attack was successful:

**System Control**: The hacker can gain control of the system via their malicious code. This could result in immediate damages, users losing access and potentially a ransomware attack.

**Data Corruption:** An attacker could intentionally cause a system corruption. Where they damage files or areas of the system and cause damage to the integrity and lifespan.

**Data breach:** User’s data could be stolen by the attackers. This may include any sensitive data attached to their file.

### How to defend against this Vulnerability / Attack:

File Directories: Ensure when files are uploaded, they are stored in a quarantine zone that is isolated.

**File Types:** Only allow specific file types.

**File Scanner:** Implement a file scanning system which checks for malware within all files uploaded.

## **Code Injection**

Code injection is where an attacker enters in malicious code into an input field. This data is then executed by the webserver or application. This causes integrity issues as if this vulnerability is not properly identified. An attacker can easily access the system and deploy further attacks.

### How is this attack found via the Vulnerability Scanner?

‘ "Eval": re.compile(r'eval\s\*\(.\*\)'), ‘ this line of code is responsible for identifying code injection vulnerabilities. It searches for ‘eval’ in the selected file. If any lines of code contain this within an input section, the alert is triggered

### Potential issues if Vulnerability / Attack was successful:

**File Deletion:** Attackers could access the system then delete any files they would like too. This may include system files that are crucial to run the server.

**Manipulation:** The attacker may manipulate files

**Data breach:** The attacker may gain access to the system and view individual users sensitive data.

**Malware infection:** The attacker could deploy malware to the users device or the webserver. Dependant on the malware, this could corrupt files, infect other users and potentially give full access to the attacker.

### How to defend against this Vulnerability / Attack:

**Data Sanitisation:** Sanitise all data that is collected from users.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Develop Input Controls:** Development of controls that prevent users from inputting characters that they would not need for the given input field.

## **Directory Movement – Transversal Attack**

This vulnerability is within the file directory. This vulnerability is exploited via an attacker who manipulates the file path/ directory path. This allows users (usually the attacker) to reach areas they are not supposed to be able to access. This may include areas within the web application or potentially where the files are stored off the web application.

### How is this attack found via the Vulnerability Scanner?

‘ "Directory\_Movement": re.compile(r'fs\.readFile\s\*\(.\*\.\.\/.\*\)') ‘ This line searches for any attempt to travel though the file system. This vulnerability hard codes Directory movement into the application.

### Potential issues if Vulnerability / Attack was successful:

**Data Corruption:** An attacker could intentionally cause a system corruption. Where they damage files or areas of the system and cause damage to the integrity and lifespan.

**Data breach:** User’s data could be stolen by the attackers. This may include any sensitive data attached to their file.

**Data Integrity:** Users files and information may be compromised. Which could result in legal trouble for Nutrihelp. This is completely avoidable if proper sanitisation and validation is performed

### How to defend against this Vulnerability / Attack:

**Data Sanitisation:** Sanitise all data that is collected from users.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Whitelisting:** Setting the allowed file paths for users. This ensures that the information is only accessible based on tehri permissions.

## **Insecure Token Generation**

This vulnerability refers to the generation of tokens. Which re used in session management, authentication and verifying a user is accessing the web server from their normal location. If the tokens generated are static or low include low RNG. They are weak and can be easily generated or predicted by attackers. This is also a large issue of the tokens are not hashed and contain no form of cryptographic cover.

### How is this attack found via the Vulnerability Scanner?

‘ "Insecure\_Token\_Generation": re.compile(r'Math\.random\s\*\(\)') ‘ This line searches for an insecure token generation. As “math.random” generates a “random” number between x and y. dependant on how the application is configured. This method can be calculated by advanced hackers. Which means the attacker could gain access to the MFA token and use it themselves.

### Potential issues if Vulnerability / Attack was successful:

**Stolen MFA Token:** An attacker could predict the MFA token generation then use it themselves to gain access to an account or the system.

**Data Exposure:** An Attacker could access all the data that the linked account for the MFAS token could previously access. Dependent on the exact account, the attacker may access more than one user’s sensitive data.

**Password bypass**: Dependant where this insecure token generation occurs. An attacker could predict the MFA token then manually reset that user’s password. This would lock the user out of their own account, and let the attack cause any damage they can.

### How to defend against this Vulnerability / Attack:

**Using Stronger Token Generation Methods:** Utilizing a range of other methods to cryptographically generate an MFA token. This eliminates the prediction aspect from an attacker.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Acknowledgement:** Acknowledging that math.random is insecure, and upgrading any areas that still use it to a cryptographic approach.

## **Permission Level**

This vulnerability is within the code itself. If incorrect permission settings are enabled for users, this could be damaging to the web server. As the user could accidently or intentionally cause damage to the system. Their account could also be compromised by an attacker. This would lead to much further damage and system wide corruption if the permission levels are not set correctly.

### How is this attack found via the Vulnerability Scanner?

‘ "Dangerous\_Permission\_Level": re.compile(r'fs\.chmod\s\*\(.\*\)') ‘ This line searches for code where the permission level is changed. The “fs” stands for file system in this case. The “chmod” is the privilege escalation. This line essentially searches for any time where a privilege escalation occurs that is not performed safely.

### Potential issues if Vulnerability / Attack was successful:

**Privilege Escalation:** An attacker could elevate their privilege and potentially reach other areas of the web API. This may allow them full control or access to sensitive files.

**Code Execution:** As the attacker now has a higher level of access within the system. They could potentially execute code or deploy malware.

**Loss Of Trust:** If client’s information is accessed and stolen by the attacker. This may lead to a distrust between users and the organisation. This may result in users deleting their account and seeking another service provider.

### How to defend against this Vulnerability / Attack:

**Least Privilege Access:** Ensuring all accounts have the least privilege access needed to operate. This limits the potential damages of an attacker if the gain access to a user’s account.

**Monitor/ Logging:** Develop a system that logs all areas each user accesses. This allows data to be collected and verify if any users are within systems they should not have access too.

**Secure coding principles and frameworks:** Utilising online documentation and guides to code with security and data integrity first above all else.

**Whitelisting Of File Paths:** Creation of a whitelist of file paths that are allowed to be accessed / edited

## **Redirects**

This vulnerability as the name suggests, refers to redirects. An attacker can create malicious links and place them in their script. Which could redirect users to malicious URL’s. this type of attack could to a hijacked session, redirected to a webpage owed by the hacker that contains malware or further exfiltration. Without further validation and set URL’s, this could be heavily exploited by an attacker.

### How is this attack found via the Vulnerability Scanner?

‘ "Redirects": re.compile(r'res\.redirect\s\*\(.\*req\.query\..\*\)') ‘ This line searches for an attempt to redirect the user. As this particular method is dynamically determined by the set value within “req.query”. an attacker could intercept this and change where the redirect URL is located.

### Potential issues if Vulnerability / Attack was successful:

**Redirect attacks/ Phishing:** An attacker could attack this area via creating their own malicious link that directs users to their web page instead. This could result in a user entering in sensitive information to the attackers website or potentially downloading malware onto their machine.

**Bypass Security Policy:** when a redirect occurs, pre-configured security policies will not apply on the new URL. Which means that there is little to no protection at this site, as it is not part of the original organisations.

**Malware:** As stated above, the attacker website can automatically download malware onto the users device. As the new URL is configured as per the hacker’s configuration.

### How to defend against this Vulnerability / Attack:

**Reject untrusted/ not verified inputs:** Automatically rejecting any input that has not been previously validated by the system.

**URL Validation:** Validating each URL that is legitimate.

**Security Headers:** Develop informative security headers which specify exactly which domains are allowed and which are not.

## **Summary**

As detailed in this report, it is crucial to ensure there are no vulnerabilities coded into applications. However, if vulnerabilities are found, they can still be patched before an attacker gets the chance to abuse them. All of the vulnerabilities discussed in this report can be updated and patched in a single line of code. Majority of the vulnerabilities are created due to improper data sanitisation, validation or file handling. When applications are developed, Integrity, accuracy and effectiveness should be the top priority to ensure success.

Please reach out to me if you have any questions in regard to patching any vulnerability my program has identified in your code.