5-1-2020

IMPROVING THE SECURITY OF THE ‘WHATCHAMABUDGET’ BUDGETING APPLICATION THROUGH A SECURITY AUDIT

Robert Kemp

IMPROVING THE SECURITY OF THE 'WHATCHAMABUDGET' BUDGETING APPLICATION THROUGH A SECURITY AUDIT

by

ROBERT J.T. KEMP

Presented to the Faculty of the Honors College of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

HONORS BACHELOR OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2020

ACKNOWLEDGMENTS

I would like to thank my Senior Design instructor, Dr. Shawn Gieser, for giving me guidance on this topic. I would also like to thank Dr. David Levine and Dr. Jiang Ming for providing me with an exemplary background on the subject of computer security.

April 07, 2020

ABSTRACT

IMPROVING THE SECURITY OF THE

‘WHATCHAMABUDGET’ BUDGETING

APPLICATION THROUGH

A SECURITY AUDIT

Robert Kemp, B.S. Computer Science

The University of Texas at Arlington, 2020

Faculty Mentor: Shawn Gieser

Security auditing of software applications is becoming a necessity with a growth in the complexity of attacks and number of attackers. There are many ways to conduct a security audit and many things that may be looked for. In general, a software security auditing process follows three main phases: analysis of the code base for vulnerabilities, categorizing/ranking these vulnerabilities based on the threat they pose, and fixing the vulnerabilities through alterations to the application’s code or design. The purpose of this work was to perform a security audit of the Whatchamabudget budgeting application being designed for the CSE capstone class, Computer System Design Project II. After performing an audit as detailed previously, a number of low-to-high severity vulnerabilities were discovered and subsequently patched, improving the security of the application, thus adding value to the project as a whole.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 A Need for Security Auditing of Web Applications

In today's world, much of what we do happens online. Many applications that we use every day are web-based applications or services, such as Facebook, Amazon, and for UTA students, MyMav. The rise of web applications has been met with the rise of web application hackers – malicious individuals out to steal valuable secrets and information from consumers and corporations. One of the most sure-fire ways to combat these individuals, or 'threat actors', is by conducting continuous and periodic security audits of a web application both during its development and deployment.

1.2 Conducting a Security Audit

A typical security audit has three generic phases [1 – 4]. First, the code base is analyzed, exercised, or otherwise tested to uncover vulnerabilities in the code. Next, the discovered vulnerabilities are categorized and ranked based on their level of severity, a measure highly relative to who will be using the application, the kind of activities the application will perform, and how it will perform those activities. Lastly, these vulnerabilities are fixed in order of most to least pressing/severe by the security team and/or the software development team.

1.3 Conducting a Security Audit of the Whatchamabudget Budgeting Application

The methodology that will be used for conducting a security audit of the CSE capstone team's budgeting application, Whatchamabudget, will essentially be the same as the common methodology mentioned in section 1.2.

1.3.1 Phase I: Code Base Analysis

The first phase of the security audit will involve analyzing and testing the code base of the application for security flaws. This analysis will target the entire application (frontend as well as backend artifacts), with an emphasis on the application backend, as this is where most of the business logic of the application has been implemented. Two methodologies will be used to conduct this source code analysis: static analysis and a dynamic code analysis.

1.3.1.1 Static Analysis

Static analysis, in the context of security auditing of applications, is the process of analyzing source code in order to find potential vulnerabilities in the way that the code is written [5]. For example, many static analysis tools look for such things as hard-coded user credentials (username and password combinations) and misconfigured or non-present settings in a configuration file that could potentially lead to security vulnerabilities.

The application backend is written in Python, utilizing the Django and Django REST web application/API design frameworks. Therefore, the static code analyzer Bandit will be used to conduct a simple static analysis of all backend files. Bandit is an open-source static security analysis tool designed to find common security issues in Python code [6].

The application frontend consists of various HTML files containing cascading style sheets (CSS) and JavaScript code, utilizing several libraries (primarily jQuery). Since it is unlikely to find security vulnerabilities in the HTML or CSS itself, due to its straight-forward usage, the emphasis of static analysis for frontend artifacts will be on those files containing large amounts of feature-rich JavaScript code. Originally, this code was to be analyzed using JSPrime, another open-source static analysis tool used to identify commonplace JavaScript security issues [7]. However, due to some unforeseen complications, this tool could not be run against the codebase, so a different JavaScript static analyzer, JSHint, was used instead [8].

1.3.1.2 Dynamic Analysis

Dynamic analysis in the context of security auditing involves analyzing a program at runtime (while it is executing) [5]. For auditing Whatchamabudget, dynamic analysis will focus on the backend code of the application since it contains the majority of the application’s business logic. However, a small exploit will also be attempted against the frontend code.

Dynamic analysis against the backend portion of the application will involve running the Zed Attack Proxy tool (ZAP), a web application penetration testing tool used to find vulnerabilities through analysis of a web application while it is running [9]. ZAP will be run against Whatchamabudget first in its automatic scanning mode, then in its manual scanning mode, as discussed later.

1.3.2 Phase II: Vulnerability Ranking

The next phase of the auditing process will be the ranking of vulnerabilities discovered in the previous phase by their level of severity. While there are a number of ways (of various levels of complexity) to measure the severity of a vulnerability, it was determined that the DREAD process will be used for this analysis. DREAD is a pneumonic ranking system chosen for its relative popularity as well as its simple approach to vulnerability ranking [10].

1.3.3 Phase III: Fixing Vulnerabilities

Lastly, all vulnerabilities that can reasonably be fixed based on project time constraints, as well as implementation constraints, will be fixed. In concept, this is the simplest phase, however, it may be the most time-consuming as each vulnerability will most likely need to be fixed in a unique way that could take much testing and trial-and-error to ensure that the problem is truly fixed.

CHAPTER 2

LITERATURE REVIEW

2.1 A Typical Security Auditing Lifecycle

According to nist.gov, a vulnerability is defined as a “weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source” [11]. Over the past few decades, the number and complexity of information systems has grown drastically. In turn, so has the number of vulnerabilities present within these systems. Luckily, security specialists have also grown their methods of detecting and swiftly fixing these vulnerabilities. The purpose of this project was to perform a security audit of the Whatchamabudget application, finding and fixing as many vulnerabilities in the application as possible. The processes and techniques used were inspired by those being used in industry today [12]. While different companies and entities each have their own procedures for going about a security audit, this report focuses on three phases that are common to most all of them: discovering vulnerabilities, ranking them, and remediating them.

2.1.1 Phase I: Discovery

In general, a first step to any application security-related auditing process involves some form of discovery to uncover issues in the code or system. For example, the CDC’s Vulnerability Management Lifecycle (Figure 2.1) has a first step of ‘discovery’ [3]. While this diagram is about vulnerability assessment in the context of a company’s network, it is easily translated to the software perspective of vulnerability discovery in source code, which is what this report will focus on. Instead of assessing a computer network composed of clients, a computer program composed of discrete methods and routines is assessed.

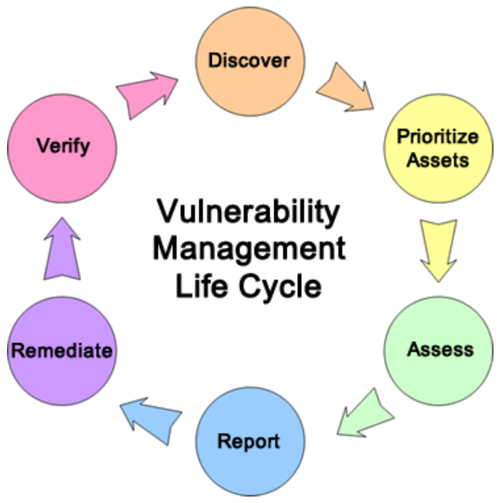


Figure 2.1: CDC Vulnerability Management Lifecycle

Similarly, the SANS institute describes a typical penetration test as following a six-step process; planning, information gathering, and vulnerability detection are the first three steps of the SANS process [4]. The OWASP Vulnerability Management Guide v.1 calls it the “Detection Cycle”, and again it occurs near the beginning of the audit and involves running tools to discover vulnerabilities [1].

2.1.2 Phase II: Ranking Vulnerabilities

A typical security audit has a second major phase consisting of ranking or categorizing known issues/vulnerabilities to facilitate the eventual remediation of these issues. For example, the CDC Life Cycle’s second and third steps are “Prioritize Assets” and “Assess” [3]. In the CDC’s model, prioritizing assets involves categorizing them into business units and assigning a business value to these units based upon how crucial each component is to the business [3]. This same ideology can be used to rank software vulnerabilities with minor tweaking.

Instead of business units, vulnerabilities identified in software may be categorized based on the underlying issue that is causing the vulnerability. For example, this may be a buffer overflow, a cryptographic key that is too short, or a similar technical issue. Next, instead of assigning a risk score based on business value, a software vulnerability ranking could assign a risk score based on a variety of categories related to the impact of an attacker exploiting the vulnerability. The widely used DREAD vulnerability ranking model is a good example of some categories that may be worth considering. Using the DREAD model, the vulnerabilities would be assigned a risk score based upon their potential to cause damage, how easy the vulnerability is for a malicious entity to reproduce, how much work must be done in order to exploit the vulnerability, how many users would be affected by an attack, and how easy it is for an attacker to discover the vulnerability [10].

In the SANS institute guidelines for Implementing a Vulnerability Management Process, vulnerability ranking is performed during the Remediation Phase, wherein different individuals within the business cooperate to rank and prioritize the discovered vulnerabilities [2].

2.1.3 Phase III: Fixing Vulnerabilities

The last major step typically found in a vulnerability management lifecycle is the fixing or remediation of the vulnerabilities in order of most critical to least critical as determined by the previous phase.

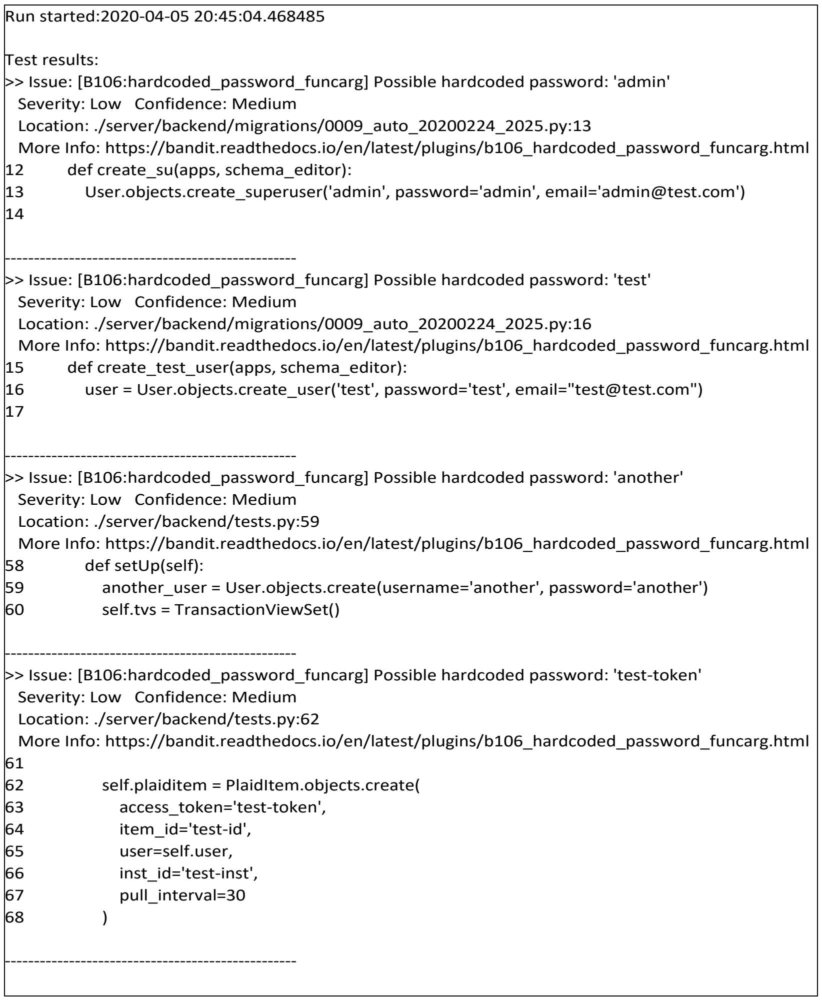
The CDC's model does this as a fifth step: prioritizing and fixing vulnerabilities in the order of their business risk [3]. OWASP's vulnerability management guide also has a remediation step as its final step which it refers to as the Remediation Cycle. During this cycle, vulnerabilities are prioritized utilizing reports, trend analysis, and through the input of business stakeholders [1]. After this prioritization, the vulnerabilities are then patched in order of their rank.

CHAPTER 3

METHODOLOGY

3.1 Static Analysis of the Backend

My first round of static analysis occurred on April 5^{th}, 2020. I waited until a bit later in the lifecycle of the project to do the first round of testing because I wanted to allow the codebase to become not only more stable (in that less major changes were occurring as compared to the project's early phases), but also more complete and indicative of what it would be in its deliverable state. I ran the Bandit security static analysis program recursively on the Django project file hierarchy and got the following results:



```
Run started:2020-04-05 20:45:04.468485

Test results:
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: 'admin'
   Severity: Low   Confidence: Medium
   Location: ./server/backend/migrations/0009_auto_20200224_2025.py:13
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b106_hardcoded_password_funcarg.html
12        def create_su(apps, schema_editor):
13            User.objects.create_superuser('admin', password='admin', email='admin@test.com')
14

--------------------------------------------------
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: 'test'
   Severity: Low   Confidence: Medium
   Location: ./server/backend/migrations/0009_auto_20200224_2025.py:16
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b106_hardcoded_password_funcarg.html
15        def create_test_user(apps, schema_editor):
16            user = User.objects.create_user('test', password='test', email="test@test.com")
17

--------------------------------------------------
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: 'another'
   Severity: Low   Confidence: Medium
   Location: ./server/backend/tests.py:59
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b106_hardcoded_password_funcarg.html
58            def setUp(self):
59                another_user = User.objects.create(username='another', password='another')
60                self.tvs = TransactionViewSet()

--------------------------------------------------
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: 'test-token'
   Severity: Low   Confidence: Medium
   Location: ./server/backend/tests.py:62
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b106_hardcoded_password_funcarg.html
61
62                self.plaiditem = PlaidItem.objects.create(
63                    access_token='test-token',
64                    item_id='test-id',
65                    user=self.user,
66                    inst_id='test-inst',
67                    pull_interval=30
68                )

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

Figure 3.1: Output of Running Bandit on Back End Python Code

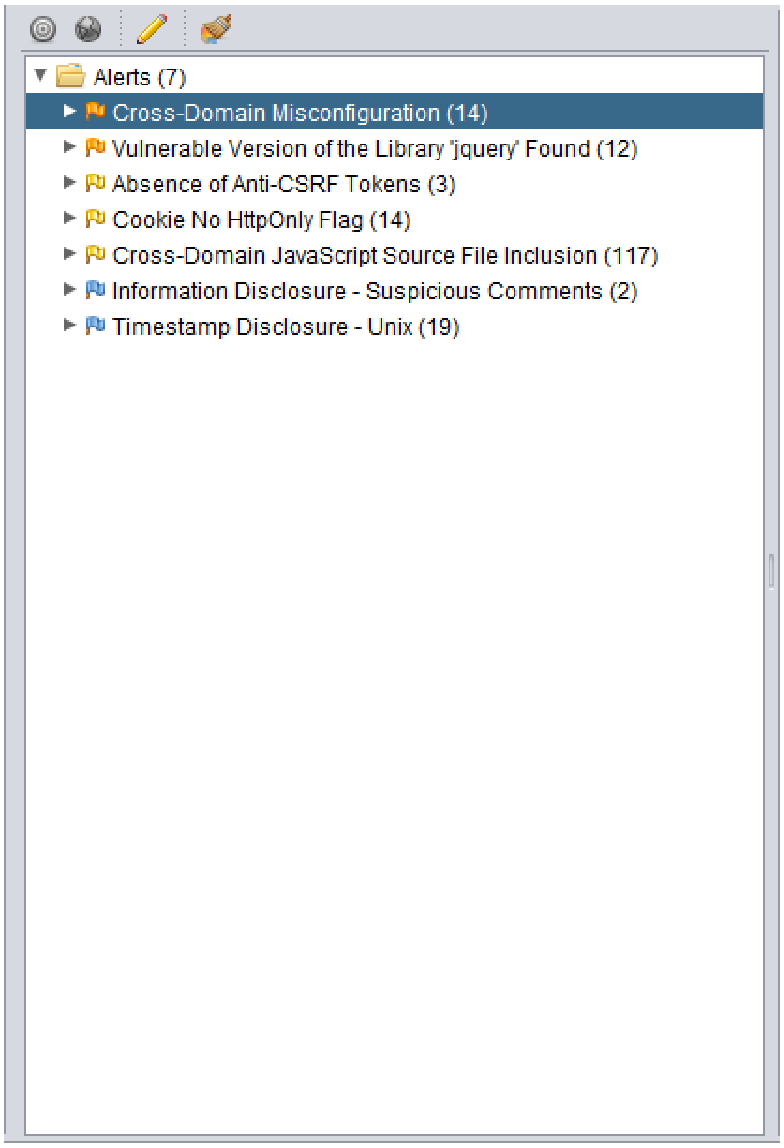
3.2 Dynamic Analysis of the Backend

On April 5th I also ran the first round of dynamic analysis on the backend portion of the application. I configured the ZAP web application scanning tool to do a scan of Whatchamabudget running at the loopback address (127.0.0.1) on my local machine. In configuring ZAP, the two most important settings to set correctly are ZAP's 'mode' and the type of scan that will be performed using this 'mode.'

ZAP has four modes of operation: Safe, Protected, Standard, and ATTACK [9]. In the course of this research, all scans with ZAP were performed in its Standard mode, as the Safe and Protected modes were considered too restrictive in their scanning methods to be useful (Safe mode prevents ‘dangerous’ operations, and Protected mode only scans endpoints in the current scope of the application) [9]. ZAP’s fourth mode, ATTACK mode, was not used as the only difference from standard mode is that it scans new nodes/endpoints immediately with the active scanner, which is not important for the purpose of this paper.

After configuring ZAP to run in Standard mode, ZAP was run against Whatchamabudget using its two main scanning methods: automatic scanning and manual scanning.

Automatic scanning mode spawns a web crawler (spider) to map the URLs of a web application [9]. After this mapping is finished, ZAP sends a variety of specially crafted malicious requests via its Active Scanner to the discovered endpoints [9]. The Active Scanner has a variety of rules that test for many commonly found vulnerabilities in web applications [9]. These include, but are not limited to, buffer overflows, code injection, command injection, reflected and persistent cross site scripting, path traversal, server side include, and SQL injection [9]. ZAP’s vulnerability alerts tab that resulted from running the automatic scan is shown in Figure 3.2.



Alerts (7)
- Cross-Domain Misconfiguration (14)
- Vulnerable Version of the Library 'jquery' Found (12)
- Absence of Anti-CSRF Tokens (3)
- Cookie No HttpOnly Flag (14)
- Cross-Domain JavaScript Source File Inclusion (117)
- Information Disclosure - Suspicious Comments (2)
- Timestamp Disclosure - Unix (19)

Figure 3.2: ZAP Testing Results

Additionally, ZAP was run in its manual mode, which allows for manual exploration of a website while ZAP proxies all requests between the browser used for testing and the application under test [9]. ZAP analyzes the requests and responses to and from the application for security flaws [9]. ZAP can also be called upon at various points during this manual scanning to run its Active Scanner, which will send various requests and payloads that will help to test the app's security, much like in the automatic scan [9].

The results of the manual scan are viewable in the form of an HTML report ZAP creates that enumerates each vulnerability found. Since the manual scan of the application occurred immediately after the automatic scan, without exiting ZAP, the results of the manual scan included those found during the automatic scan run previously. Since the generated HTML report was too large to include as an image, its contents are summarized in Table 3.1 below.

Alert Message	Quantity	Alert Priority Level
Cross Site Scripting (Reflected)	4	High
Buffer Overflow	4	Medium
Cross-Domain Misconfiguration	15	Medium
Vulnerable Version of the Library 'jquery' Found	19	Medium
Absence of Anti-CSRF Tokens	8	Low
Application Error Disclosure	4	Low
Cookie No HttpOnly Flag	20	Low
Cross-Domain JavaScript Source File Inclusion	171	Low
Information Disclosure – Suspicious Comments	15	Informational
Timestamp Disclosure – Unix	57	Informational

Table 3.1: Site Alerts from Manual ZAP Scan

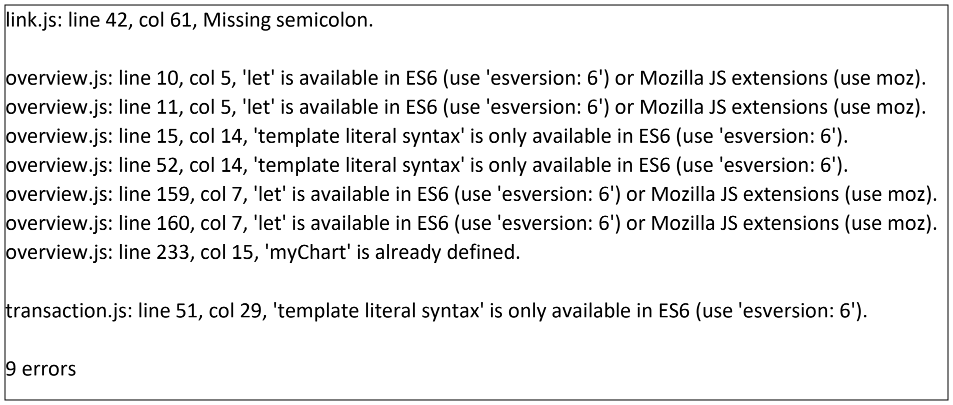
3.3 Static Analysis of the Frontend

The first step in analyzing the frontend code for the application was to determine which code would be most crucial to analyze. This was necessary because all of the JavaScript in the project was embedded in the HTML files, and the tool to be used to perform the static analysis, JSPrime, only operates on JavaScript files, not HTML files. Therefore, manual extraction of the code to a new file was necessary. Due to this, and the fact that most of the crucial frontend logic was present in only a few modules, it was necessary to pick the most crucial modules to analyze. Four JavaScript code modules were selected: expenses.js (responsible for querying the app's backend API for budgetary expenses created by the user and presenting them to the user), link.js (the module responsible for making the connection to the external Plaid banking API), overview.js (holds the logic that gets and displays the user's transaction information in a graphical format), and transaction.js (responsible for getting, displaying, updating and creating user bank transactions).

Once the code to be statically analyzed was determined, JSPrime was run against each file in turn. Unfortunately, for each file run against the tool, the tool threw an internal error. After closer inspection of the tool's source code and the stack trace that was produced on execution, it was determined that the version of NodeJS that was being used to run the tool needed to be an older version to maintain compatibility. Inspecting the last update time of the file throwing the error on GitHub, it was determined that a version of NodeJS before 2015 should be appropriate to try and run the tool with. Several versions of NodeJS were downloaded and used to run the tool in a Linux testing environment, including versions as old as 0.8.27, and as new as 0.11.13 (one of the newer versions from 2014, being that 2014 was the last time the problematic file had been updated).

Unfortunately, despite running the tool with these various versions, problems persisted in the tool's source code itself. Rather than attempting to fix issues with the broken tooling, it was decided that an alternative JavaScript static analyzer would be used; one with less of a focus on security threats, yet would still provide information on how to improve the JavaScript modules.

JSHint was the next tool selected to perform the static analysis of the JavaScript. It was mainly selected due to its popularity, and the wide suite of features it advertised. The same four code files were run against JSHint, producing the output in Figure 3.3.



```
link.js: line 42, col 61, Missing semicolon.

overview.js: line 10, col 5, 'let' is available in ES6 (use 'esversion: 6') or Mozilla JS extensions (use moz).
overview.js: line 11, col 5, 'let' is available in ES6 (use 'esversion: 6') or Mozilla JS extensions (use moz).
overview.js: line 15, col 14, 'template literal syntax' is only available in ES6 (use 'esversion: 6').
overview.js: line 52, col 14, 'template literal syntax' is only available in ES6 (use 'esversion: 6').
overview.js: line 159, col 7, 'let' is available in ES6 (use 'esversion: 6') or Mozilla JS extensions (use moz).
overview.js: line 160, col 7, 'let' is available in ES6 (use 'esversion: 6') or Mozilla JS extensions (use moz).
overview.js: line 233, col 15, 'myChart' is already defined.

transaction.js: line 51, col 29, 'template literal syntax' is only available in ES6 (use 'esversion: 6').

9 errors
```

Figure 3.3: Output of Running JSHint JavaScript Static Analysis on Critical Project Code Modules

3.4 Dynamic Analysis of the Frontend

The first step in performing dynamic analysis of the frontend JavaScript code was to establish a library used by the application that could potentially be exploited, and to search public exploit databases for inspiration/information on an exploit to try. This was the chosen procedure for two main reasons. First, the JavaScript the team had written for the application was not complex enough to potentially introduce significant vulnerabilities. It was reasoned that the likelihood of finding exploitable code in such a small footprint would be lower than attempting to find a vulnerability in the various included libraries in the application. Secondly, vulnerability libraries were used for reference because attempting to write an exploit by hand would be very time-consuming and require more experience in that area than I currently possess.

Research was conducted at several points from mid-April to early May in order to find a potentially exploitable vulnerability across the project's various libraries. The main portion of this research included searching popular vulnerability databases, such as exploit.db and snyk.io's vulnerability database for recently-identified vulnerabilities. It was not until late April that an applicable and sufficiently interesting vulnerability was published. On April 29th, 2020 a Cross-site Scripting vulnerability affecting all versions of the jQuery library less than 3.5.0 was published to snyk.io's vulnerability database. This vulnerability, classified as CVE-2020-11022, allowed for the execution of arbitrary code by a browser when unsanitized (or even sanitized) HTML input was passed to one of jQuery's DOM manipulation methods. Moreover, a second closely-related vulnerability, CVE-2020-11023 allowed for similar execution of untrusted code when passed to one of jQuery's DOM manipulation methods, though this vulnerability was caused by the use of the HTML <option> tag in the malicious code.

An attempt to exploit this vulnerability was performed on the expenses page of Whatchamabudget, as this page was identified as being potentially vulnerable to both of these exploits, as it used jQuery 3.4.1, and contained an <option> list for a dropdown selection menu on the expense creation modal. Though the description of the vulnerability was not explicit on how to exploit it (understandably so), research was performed into performing a general cross-site-scripting (XSS) attack, as this vulnerability fell under that general category. The OWASP XSS Filter Evasion Cheat Sheet by Jim Manico, [13], proved to be a great reference to performing a variety of different XSS attacks, including one that appeared to relate closely to the jQuery vulnerabilities that were under exploitation in Whatchamabudget. This XSS method involved injecting HTML into the page, specifically a <script> tag that would execute arbitrary code and submitting it to the server in a stored XSS attack. Several encoding methods using this strategy were attempted. First, the script tag was injected between the <option> tags and left unclosed in hopes of tricking both Django and jQuery data validation techniques. When this was unsuccessful, a script tag was added as an individual option to the list and left unclosed, ideally to be evaluated and executed by jQuery. Unfortunately, jQuery did not execute the script tag as expected, nor did Django allow the input to be submitted to the backend. A few other methods mentioned on the XSS Cheat Sheet were attempted, but also to no success.

3.5 Ranking Backend Vulnerabilities using DREAD

The vulnerabilities identified in the backend during both static and dynamic analysis are listed in Table 3.4 below. The number of instances of the vulnerability found in the source code is listed in the ‘Quantity’ column. The ‘Found During’ column identifies the vulnerability as being found during either static or dynamic analysis. The ‘Is False Positive’ category indicates whether the vulnerability that was found and reported is in fact a legitimate security vulnerability. If it is a legitimate security vulnerability in need of fixing, then the vulnerability is not considered to be a false positive, otherwise it is.

The vulnerability’s DREAD ranking is specified in the last column. Each of the five DREAD categories for each vulnerability is assigned a value between one and ten. A score of one indicates a lower risk for leaving the vulnerability unfixed, while a ten represents a risk of complete application failure, or disclosure of highly critical data. A total line is included for each vulnerability that sums each DREAD category for the vulnerability. The vulnerability with the highest total will be fixed first, followed by the next, etc.

Vulnerability	Found During	Is False Positive	DREAD Ranking
Cross Site Scripting (Reflected)	Dynamic analysis	No	Damage: 7 Reproducibility: 6 Exploitability: 7 Affected users: 8 Discoverability: 4 **Total:** 32
Cookie No HttpOnly Flag	Dynamic analysis	No	Damage: 2 Reproducibility: 1 Exploitability: 2 Affected users: 2 Discoverability: 5 **Total:** 12
Cross-domain misconfiguration	Dynamic analysis	No	Damage: 1 Reproducibility: 1 Exploitability: 1 Affected users: 3 Discoverability: 4 **Total:** 10
Timestamp disclosure - Unix	Dynamic analysis	Yes	N/A
Hardcoded password	Static analysis	Yes	N/A
Buffer Overflow	Dynamic analysis	Yes	N/A
Application Error Disclosure	Dynamic analysis	Yes	N/A

Table 3.2: DREAD Ranking of Backend Vulnerabilities

3.6 Ranking Frontend Vulnerabilities using DREAD

Since the static analysis performed on the frontend code was not security-oriented, but rather focused around the syntax of the frontend code, it has been decided that giving the issues found by JSHint would be misleading. However, dynamic analysis of the backend revealed several vulnerabilities that related more closely to issues in the frontend code rather than the backend. These are listed below in Table 3.3. The dynamic analysis of the frontend did not identify any additional vulnerabilities; however, it did confirm the validity of the "vulnerable version of the library 'jQuery' found" vulnerability.

Vulnerability	Found During	Is False Positive	DREAD Ranking
Vulnerable version of the library 'jQuery' found	Dynamic analysis	No	Damage: 4 Reproducibility: 3 Exploitability: 3 Affected users: 4 Discoverability: 4 **Total:** 18
Cross-domain JavaScript source file inclusion	Dynamic analysis	No	Damage: 3 Reproducibility: 3 Exploitability: 2 Affected users: 4 Discoverability: 5 **Total:** 17
Absence of anti-CSRF tokens	Dynamic analysis	No	Damage: 3 Reproducibility: 2 Exploitability: 3 Affected users: 2 Discoverability: 4 **Total:** 14
Information disclosure – suspicious comments	Dynamic analysis	Yes	N/A

Table 3.3: DREAD Ranking of Frontend Vulnerabilities

3.7 Fixing Backend Vulnerabilities

All reported vulnerabilities that were not false positives from section 3.4 were fixed successfully such that they were no longer reported by any analysis tools after the following changes were made in this order (the order given by descending DREAD total):

The cross-site scripting vulnerability in the backend code was the most dangerous of all the vulnerabilities identified, mainly because it is one of the main ways for an attacker to load and run arbitrary code in a user's browser. The issue causing the XSS vulnerability in the backend code was a lack of input validation at the API endpoint responsible for returning the user's total income and expenditure upon receipt of a POST request. The fix was to simply use Django Rest Framework's built-in serializer classes to ensure that the incoming data was a date and nothing else, like an HTML script tag, that could be utilized in an XSS attack.

For the "Cookie no 'HttpOnly' flag" vulnerability, the Django server's settings were reconfigured such that CSRF cookies were sent with the 'HttpOnly' flag active. This fix was made by adding the CSRF_COOKIE_HTTPONLY keyword settings into Django's settings file and setting it to 'True.'

Fixing the cross-domain misconfiguration vulnerability required adding a new application that would be run alongside the Django server: "django-cors-headers" [14]. An open-source application for Django, this app allows a Django developer to whitelist origins via the "CORS_ORIGIN_WHITELIST" setting that is introduced to the Django settings file. Adding this setting along with the domain of the app, and the domains of the allowable CDNs fixed this vulnerability.

3.8 Fixing Frontend Vulnerabilities

All reported vulnerabilities that were not false positives from section 3.5 were fixed successfully such that they were no longer reported by any analysis tools after the following changes were made in this order (the order given by descending DREAD total):

The vulnerable jQuery version (3.4.1) identified by both the Retire.js plugin for ZAP [15], and the dynamic analysis of the frontend was upgraded across all application pages to a newer version of jQuery, 3.5.0. The newer version of jQuery was fully backwards compatible with all the features that were utilized by the application, making switching to the newer version very easy.

The "cross-domain JavaScript source file inclusion" vulnerability, while not a false positive, did not necessarily require any type of explicit fix or change to the source code. However, the libraries in question that were identified by the tool were inspected to ensure that the most recent and legitimate version was being used, especially in the case of CDN links.

Lastly, the "absence of anti-CSRF tokens" vulnerability was fixed by including adding Django's special 'csrf_token' flag to the HTML template. This flag, when added to the body of an HTML form that is to be rendered by Django, will automatically include a hidden input field with a value equal to the CSRF token that is to be submitted to the server. This CSRF token prevents cross site requests by a malicious actor using a logged-in user's credentials.

CHAPTER 4

DISCUSSION

This chapter will focus on analyzing the results in Chapter 3: Methodology. It will attempt to explain why the results obtained in the methodology chapter were obtained and what they mean. It will be focused on the results obtained from testing the backend, as there is more to discuss concerning these results since the application logic was implemented almost entirely in the backend code.

4.1 Static Analysis of the Backend

While the results from the analysis using Bandit seemed very minimal, they were not useless. To ensure that the tool was not mistakenly leaving critical files out of the analysis, it was run in its debug mode, which listed this information. Doing so confirmed that the tool was scanning all the files in the target directory, as it was expected to.

As shown in Figure 3.1, Bandit did not return very interesting results from its analysis. The only things that it was able to return that looked ‘suspicious’ in terms of security vulnerabilities were four instances of what appeared to be ‘hardcoded passwords.’ The first two instances it found were in fact instances of hard-coded credentials. However, these were for the purposes of integration testing the app more easily (they kept me from having to create a new user every time the database was destroyed). The latter two were for unit testing and were also known about previously.

While it is unfortunate in the sense that the tool had no interesting vulnerabilities to report on from its analysis, it is also a good thing. Firstly, the tool is known to have scanned the correct files, and to not have encountered any errors while doing so. Moreover, the tool is well-known and widely used, so there is not much concern about the tool's thoroughness. This leads to a reasonable conclusion that the backend code of the application is at the very least relatively secure against common vulnerabilities.

While initially it was believed to be a bit strange that a Python app of this complexity was not identified as having any security vulnerability patterns in the code, this is not entirely surprising when one considers the design of the framework that the code was written on top of. As a widely used and reliable web framework, Django is built to be inherently secure, because it has to be for its wide user base. Visiting djangoproject.com, one will see "Reassuringly secure" is a front-and-center feature. Looking deeper into Django's documentation, on the "Security in Django" page [16], one will find a host of features provided by Django to help ensure secure coding while using the framework, including cross site scripting protection, cross site request forgery protection, host header validation, session security, and many more.

4.2 Dynamic Analysis of the Backend

Dynamic analysis of the backend using the ZAP web application scanner yielded many more results compared to the backend's static analysis phase. The initial automatic scan of the app revealed seven different alerts from ZAP. Some of these were clearly issues in the backend code – e.g., how the server code was setting HTTP headers in the case of the "Cookie No HttpOnly Flag" vulnerability. However, others discovered during this scan related more to frontend artifacts, such as the "Vulnerable version of the library 'jQuery' found" vulnerability and were therefore delegated to the appropriate section in Methodology.

Furthermore, it is worth expanding upon why each vulnerability that was marked as a false positive was marked that way. In the backend, the “Timestamp disclosure” vulnerability was a false positive because the timestamps that appeared to be disclosed were not in fact related to any application logic, or were just random numbers that happened to appear to be a timestamp encoding. Therefore, this was a false positive.

The “Buffer Overflow” and “Application Error Disclosure” vulnerabilities were both false positives for the same reason – the mode that the server was running in during testing. Accidentally, for part of the testing the server had been running in a non-production mode, resulting in invalid input given to the server returning a 500 HTTP error, indicating a fault at the server. Running in production mode, the server would return a 400-client error message, and this vulnerability would not have been reported.

4.3 General Analysis of the Frontend

This section will be a discussion of the results of the frontend testing/attempted exploitation, both front and back end as discussed in the Methodology section.

The failure with the tooling for the frontend static analysis was disappointing; however, the information provided by JSHint was still useful in its own way. The issues identified by JSHint during the frontend static analysis were repaired, adding value to the project even though it was not security-related value.

The ‘information disclosure – suspicious comments’ discovered during the backend dynamic analysis phase and listed as a frontend false-positive vulnerability in Table 3.5 was a false-positive because the ‘suspicious comments’ in question revealed no critical information about the application, and provided nothing to a potential attacker.

Attempting exploitation of the application frontend during the frontend dynamic analysis phase through a known, recent, and real-world vulnerability was fun and exciting, but also very challenging. There is much challenge in knowing exactly how to apply an exploit, including what encoding should be used, when the exploit should be applied and at what exact step it should be expected to execute. Though no great success came from the attempted exploitation of the recently disclosed jQuery vulnerability, it was a fantastic learning experience and good practice in thinking "outside of the box" and analyzing all parts of a software system.

CHAPTER 5

CONCLUSION

In conclusion, after conducting a security audit of the Whatchamabudget budgeting application, a variety of vulnerabilities were discovered, ranked in order of their severity, and repaired in ranked order, from highest to lowest severity. This added value to the application as a whole by both improving the safety of certain portions of the code base while expanding the safety of those lacking it.

REFERENCES

[1] "OWASP Vulnerability Management Guide v.1," *owasp.org*. [Online]. Available: https://owasp.org/www-project-vulnerability-management-guide/owasp-vuln-mngm-guide-v1.txt. [Accessed: 20-Mar-2020].

[2] T. Palmaers, "Implementing a Vulnerability Management Process," *SANS.org*, 23-Mar-2013. [Online]. Available: https://www.sans.org/reading-room/whitepapers/threats/implementing-vulnerability-management-process-34180. [Accessed: 15-Mar-2020].

[3] "Vulnerability Management Life Cycle," *Centers for Disease Control and Prevention*, 24-Oct-2018. [Online]. Available: https://www.cdc.gov/cancer/npcr/tools/security/vmlc.htm. [Accessed: 05-Apr-2020].

[4] C. Wai, "Conducting a Penetration Test on an Organization," *SANS.org*, 2020. [Online]. Available: https://www.sans.org/reading-room/whitepapers/auditing/conducting-penetration-test-organization-67. [Accessed: 05-Apr-2020].

[5] Peng Li and Baojiang Cui, "A comparative study on software vulnerability static analysis techniques and tools," 2010 IEEE International Conference on Information Theory and Information Security, Beijing, 2010, pp. 521-524.

[6] "Bandit," *Bandit documentation*. [Online]. Available: https://bandit.readthedocs.io/en/latest/. [Accessed: 08-Apr-2020].

[7] N. D. Patnaik and S. S. Sahoo , “JSPrime,” *JSPrime*. [Online]. Available: https://dpnishant.github.io/jsprime/. [Accessed: 16-Apr-2020].

[8] A. Kovalyov, “JSHint, A Static Code Analysis Tool for JavaScript,” *About JSHint*. [Online]. Available: https://jshint.com/about/. [Accessed: 20-Apr-2020].

[9] “OWASP ZAP Desktop User Guide,” *OWASP ZAP*. [Online]. Available: https://www.zaproxy.org/docs/desktop/. [Accessed: 05-Apr-2020].

[10] “Security/OSSA-Metrics,” OpenStack. [Online]. Available: https://wiki.openstack.org/wiki/Security/OSSA-Metrics. [Accessed: 06-Mar-2020].

[11] “vulnerability,” NIST - CSRC. [Online]. Available: https://csrc.nist.gov/glossary/term/vulnerability. [Accessed: 16-Mar-2020].

[12] B. Liu, L. Shi, Z. Cai and M. Li, "Software Vulnerability Discovery Techniques: A Survey," 2012 Fourth International Conference on Multimedia Information Networking and Security, Nanjing, 2012, pp. 152-156.

[13] J. Manico, R. "R. S. Hansen, A. Lange, and M. Dhiraj, “XSS Filter Evasion Cheat Sheet,” *OWASP*. [Online]. Available: https://owasp.org/www-community/xss-filter-evasion-cheatsheet. [Accessed: 30-Apr-2020].

[14] “adamchainz/django-cors-headers,” GitHub, 25-Apr-2020. [Online]. Available: https://github.com/adamchainz/django-cors-headers. [Accessed: 26-Apr-2020].

[15] “Retire.js,” Retire.js. [Online]. Available: https://retirejs.github.io/retire.js/. [Accessed: 26-Apr-2020].

[16] “Security in Django,” *Django*. [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/security/. [Accessed: 04-Apr-2020].

BIOGRAPHICAL INFORMATION

Robert (Bobby) Kemp was born in Burlington Iowa in 1998 and graduated salutatorian of the 2016 class of Burlington Community High School. He went on to attend the University of Texas at Arlington (UTA) in the Fall of 2016. While at UTA, Bobby pursued an Honors Bachelor of Computer Science with a minor in Business Information Systems. For the first two-and-a-half years of his undergraduate career, Bobby was an active member and leader in the UTA Game Developer's Club, and enjoyed working on, learning about, and occasionally playing video games in his free time. During his junior year at UTA, Bobby took UTA's introductory computer security course, CSE 4380, and passed the certification exam of the course to earn the Information Systems Security Professionals, NSTISSI No. 4011 certification, along with the System Administrators, CNSSI No. 4013E certification, as certified by The Committee on National Security Systems and The National Security Agency.

Following graduation from UTA in Spring 2020, Bobby plans to become a software engineer for a small-to-medium-sized startup company near Los Angeles, California.