Final Report

Team ZAM

6 December 2023

Instructor: Michael Leverington

Sponsor: Tim Wojtulewicz

Mentor: Vahid Nikoonejad Fard

Team Members: David Knight, Akiel Aries, Cody Beck, Nathan Chan

Version: 1.0

Table of Contents

Table of Contents 1

1. Introduction 2

2. Process Overview 4

3. Requirements 7

4. Architecture and Implementation 10

4.1 Parser 12

4.2 README Scraper 14

4.3 Search Module 16

5. Project Timeline 20

6. Testing 22

6.1 Unit Testing 22

6.2 Integration Testing 23

6.3 Usability Testing 24

7. Future Work 25

8. Conclusion 26

9. Appendix A: Development Environment and Toolchain 28

A.1 Hardware 28

A.2 Toolchain 28

A.3 Setup 28

A.4 Production Cycle 29

1. Introduction

As technology advances, cyber crimes continue to skyrocket every year. The ability to analyze your network traffic is becoming increasingly important as cyber attacks pose a serious threat to users. In the face of this ever-growing challenge allow me to introduce you to a beacon of resilience, Zeek. Zeek is an open-source project dedicated to the innovation the network security sector needs. Zeek allows its users to track network traffic packets to create Zeek logs, which are then used to detect malicious activity within a network. The great thing about this project being open source is that it allows developers to create and publish their own Zeek packages for any user to add to their arsenal of security tools.

The Zeek team implemented a package manager website to allow for a smoother workflow. Although it is a great place for users to find new packages and tools to enhance network analysis, developers at Zeek felt that the user experience and current search engine needed a major update. More specifically, they concluded that the look of the current website is outdated and doesn't fit the look of other modern websites. Furthermore, the search engine is considered subpar and often yields irrelevant or incomplete results. This is caused by the current search protocol in use, which does not parse through package descriptions well enough and utilizes a database that is said to be over-engineered and too complex for the functionality they desire. In addition to the site's inability to render images correctly, these problems make for a substandard user experience as they disrupt the user workflow by adding more time to be spent sifting through more packages than one would need to.

Fortunately for Zeek, our team was able to develop a brand-new website for them. We focused on preserving the look and feel of the original Zeek website while also giving it a more

updated modern look. Additionally, we were able to improve their search engine by implementing a more efficient search algorithm along with upgraded parser and scraper modules.

2. Process Overview

Before we began any development, we decided as a team it would be best to discuss and research what the best technologies would be based on some of the requirements of our client. Firstly, he wanted simple syntax that would not require weeks of reading through the project to understand how it works. That meant that our solution would ideally use a universal language that most developers either know already or can easily read through the official documentation and pick up the language easily. On top of this, we needed to find a solution that would not use a database as this was considered over-engineered for this project. When researching how we could work around this, we decided to use a more simple file system as this requires less space and overhead which is fine for a project revolving around under three hundred total packages. Afterward, we needed to decide on a good way to create the web server and deploy this project in a simple way that can also be used across multiple systems in case the client decides they want to move from one server to another.

With all of this in mind, we decided to use a Python backend as this gave us an easy-to-use yet powerful language that also has a simple syntax. As stated previously, we decided to use a file system over a database as this had less overhead on the system and gave us easy access to the package's data without taking up tons of space on the system due to the lower number of packages. Finally, we agreed on using FastAPI due to its low overhead and seamless integration of our Python backend into our HTML allowing us to create dynamic web pages. Deployment would be done using Docker to containerize the system and allow it to be transferable from one system/server to another.

After deciding on the technologies that would be the best fit for this project, we began to put our plan into action. We decided as a group it would be best to play into our team's strengths

and assign work based on what everyone would be best at, while still ensuring that work is distributed fairly across all team members. As our client decided to provide us with a repository to host this project on their official account, we were all added to this repository and used trunk-based development. Trunk-based development is where our team treats the "main" branch on Github as the main trunk. Any time we plan to develop a feature, make a fix, or any change in general, we create a new branch that is intended to be short-lived before merging it back into the main trunk. This allows us to not only specify what changes are in which branch, but it also ensures that we do not cross over each other's work and mitigates the risk of a merge conflict. A merge conflict will occur when two branches in the repository revolve around the same file and the two changes interfere with each other causing human interaction to resolve it. As this can be a time-consuming process, our development strategy allowed our team to solve an immeasurable amount of time.

Once the repository was created, we were able to begin development based on our team's plan. Our client added the requirements of the project as issues inside of our GitHub repository's issue tracker. This allowed us to not only keep track of what was completed and what still needed to be worked on but also allowed our client to see when issues were resolved and keep track of our progress as we continued development of the website. We decided to play into our team's strengths and ensured that most backend development went to our team architect and release manager as they had the most experience with working behind the scenes and creating APIs to be used within our project. That left us with a strong front-end engineer and our team lead was able to work on both sides as he had experience with full-stack web development. This process of creating a back-end team and a front-end team ensured that all the developers were developing the best possible work they could while remaining comfortable within their work environment.

Whenever the back-end team finished with an API such as the search algorithm, our front-end team would connect that API call to the website and begin styling the site ensuring that the required functionality could be used, while also maintaining the styling requested by our client. This also kept every developer busy so no team member felt as if somebody was not contributing. Once changes were made and approved by our client, we would create a pull request to merge our branch back into the main trunk branch and make deployments to production as needed and/or requested by our client.

Our team architect ensured that our project would work across any system and containerized our project using Docker, which is similar to a sandbox environment where we can set up our project and not have to worry about the system not having dependencies installed.

Staying consistent with this development practice allowed our team to be as successful as possible while still ensuring that no team member was doing less work than another team member. This success was also reliant on strong communication with our client to ensure that the work that we were creating was up to their standard. This communication played a vital role in the success of our project as it allowed us to make all the proper and necessary changes required of us from our clients before merging our work with the completed and approved work and ensuring we gave our client the best possible version of our project.

3. Requirements

The initial website our team was introduced to was a package website with a search engine that was in dire need of reconstruction and a design that was outdated. Our team had multiple discussions with the Zeek team when it came to requirements and decided on a handful of functional and non-functional requirements we would need to satisfy.

As for the functional requirements we agreed on three major components, a fast and precise search engine, an efficient metadata parser, and an accurate *README* scraper. To improve the search engine, we wanted to emphasize the importance of prioritizing relevant results over a simple package listing. Through internal team discussion, we chose to implement the Okapi BM25 ranking algorithm, which would apply scores to certain packages based on the variations of package descriptions and *README*s.

Furthermore, implementing the parser involved working through a metadata file hosted on Zeek's GitHub to extract information about each package. To navigate through the metadata file, we felt that it was easiest to employ regular expressions as the metadata file is TOML format, and data such as script directories, versions, and building and testing commands are organized in brackets. After this, the next step was to obtain each package's *README* file. The parser would request each one and save all the necessary information about each package to a static file. This involved converting the package information to JSON format and writing the formatted data to a file. The end result was the generation of static files containing comprehensive information about each package, utilizing the central metadata file as a source.

Continuing from there, the last functional requirement we needed to fulfill was an accurate *README* scraper. This is significant to our data collection because although our parser grabs this information from the metadata file, oftentimes this information can be found in the

README file of the package too. With this, we must first specify that the scraper is not searching for information the parser has already gathered. We achieve this by again using regular expressions and stepping through the *README* file to look for keywords such as "build command", etc.

Now when it comes to the non-functional requirements we had to ensure that the search engine, metadata parser, *README* scraper, and the front-end performed their specific tasks. To measure the successful implementation of the search engine we evaluated two metrics, speed and relevance. Our goal was to meet the standards of modern search engines so we aimed for results to return in less than half a second. Although speed is significant for a search engine, we believed that the relevance of results is what our clients were most keen on having in the final product. However, defining a benchmark for relevance is a difficult task so instead we decided to manually curate four lists of the five relevant packages for what we think will be common search terms: "ssh", "cve", "ja3", and "spicy". By doing this we believe that the implementation of our search engine successfully met the requirements given to us. As for ensuring an efficient metadata parser we just had to make sure the parsing was correct. Unlike the search engine, the parser had no speed requirements as it would be run in intervals at all times, however, the data being pulled had to be correct one hundred percent of the time. Along with pulling the correct information, we also had to ensure that this data was being correctly saved as it would need to be accessed by other components. In the same light, the *README* scraper also had to be accurate with its data pulling one hundred percent of the time as this is what would be presented to users on the website.

To meet the non-functional requirements of the front end of our website our main goal was revamping the design of the website to match up to its counterparts such as Rust's package

registry Crates.io or Python's Py.pi package website. We also aimed to preserve the specific look and feel of Zeek's main website as this would be most familiar to its users. This meant that we did not want to include big blocks of text that would otherwise be overlooked by users visiting the site and that package pages displayed its information in a visually pleasing way.

Lastly, from an environmental requirements standpoint, we wanted to make sure that the deployment of our website would be easily handled in the case that our clients wanted to modify it. With that being said we thought it was best for our website to be containerized using Docker and be deployed using Amazon Web Services (AWS). We also wanted to utilize modular and simple frameworks such as a Python web server for the back end and utilize technologies like FastAPI, Django, and Flask, which make websites easy to maintain.

4. Architecture and Implementation

When we started this project, our team took the requirements as provided by the client and worked together to create our solution vision. This solution vision is presented in the following diagram.

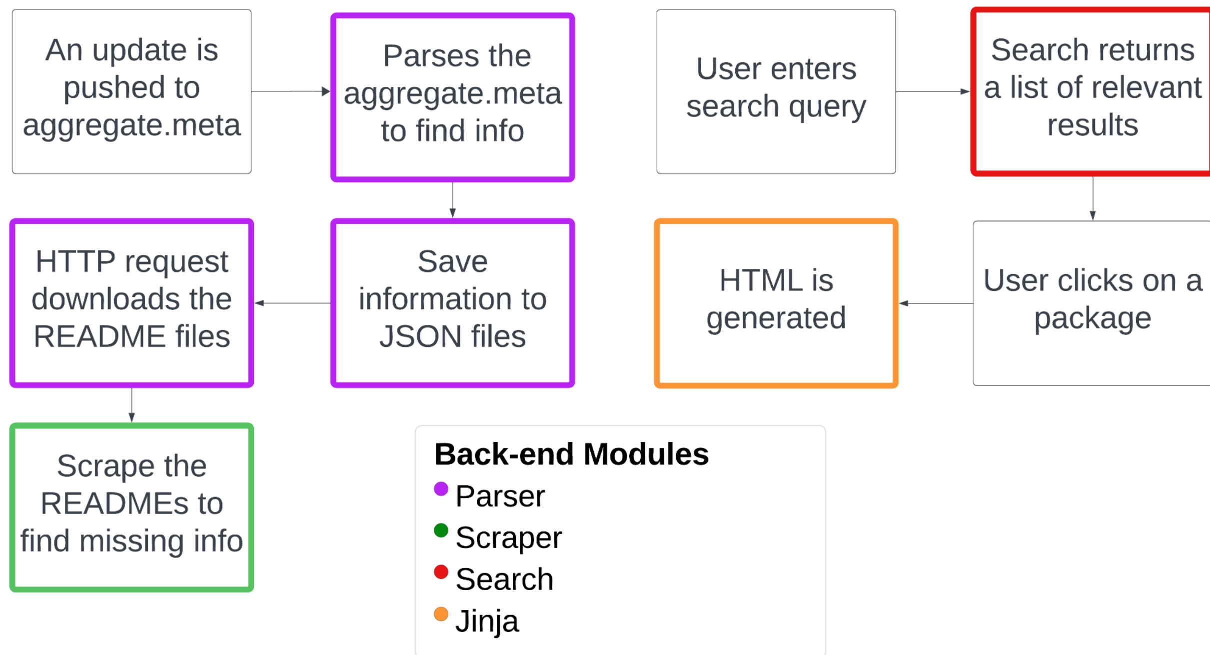

Figure 4.1 Solution Overview

From this proposed solution, we had three main back-end modules to implement: the Parser, *README* Scraper, and Search. These modules would seek to implement the core functionality that was desired from the site: take all of the metadata stored within aggregate.meta, along with package *README* files, to create *JSON* files with all relevant information for a given package, then allow users to search for a given package and present them with the most relevant packages to their search. After this core functionality was implemented, we used Jinja, allowing us to write simple HTML templates that could be filled in with package information by Python code.

The main connective tissue between these back-end modules would be the *JSON* files for the packages. Each package would have a *JSON* file generated by the Parser. The *README* Scraper would add any additional information it can gather from each package's *README* to its respective *JSON* file. The Search module would use these *JSON* files to gain all of the information it would need to appropriately rank packages against queries. Jinja would be able to use the *JSON* files to populate any HTML templates with the proper information. So long as each module handles its interaction with the *JSON* files appropriately, none of the modules would ever need to interact with each other.

After implementing these modules, we had to decide how we would actually run the back-end code. We landed on using FastAPI and Uvicorn to create a Python webserver. This was simply the easiest solution. FastAPI is known for being lightweight and barebones; we did not need too much functionality out of the web server. With this solution, we were able to implement the entire website using Python and HTML, making the whole process very smooth. Additionally, we decided to containerize everything using Docker. We chose this solution because it made standing up the website very simple. All one needs to do is build the container image, and then use docker-compose to run the necessary Uvicorn commands. Essentially, the entire website can be stood up from scratch, using three commands in a terminal. This system was chosen to maximize the portability of our website: it could be run just as easily on AWS as it could be on localhost, or anywhere else for that matter.

Now that we have discussed the main architectural details of our website, let us review the integral parts of each of our modules.

4.1 Parser

The core purpose of the Parser module is to access the aggregate.meta file as provided in Zeek's packages repository, and gather all possible information about each package listed within. This metadata file is a *TOML*-style file, and package entries within it can include the package's author, description, tags, version, internal dependencies, external dependencies, test command, build command, package repository URL, script directory, plugin directory, and user variables. The main challenge in designing this parser was that not each package entry within the aggregate.meta file was standardized. Some package entries contained all possible information, while others only had the package's repository URL (the only required piece of metadata). In designing the Parser, we took care to ensure that it could handle as much or as little metadata for each package as possible.

After getting all of the metadata parsed, the next thing to do was collect each package's *README* file. Generally, the author of any piece of software creates short text files that tell potential users how to use the software. These are known as *README* files. We knew that *README*s would be hosted, along with each package's code, in a package's repository. For all but one package, the code repository was hosted on GitHub. The outlier was hosted on GitLab. Now, we know that GitHub and GitLab have APIs that would allow us to ask for these *README* files. However, one has to pay to be able to use these APIs, and given that there are only about 225 Zeek packages, it did not make sense for us to pay for the API to gather only 225 or so text files. So, we decided to make HTTP requests to GitHub and GitLab for these files. Given that every Zeek package is hosted in a public repository, we knew that each of these requests would receive a response. So, now we had a way to gather all relevant information about each package, and we just needed to store it.

In order to store package information, we decided to use a simple filesystem, with an individual *JSON* file associated with each package. Much like the situation with the APIs, the scale of this project meant that using more traditional approaches, such as an SQL database, was unnecessary. For about 225 packages, a filesystem was manageable, and much simpler to implement than any sort of complex database. Additionally, storing all package information within files helped to enforce the compartmentalization we hoped to achieve within each back-end module.

This functionality is implemented using the following functions, as shown in the diagram below.

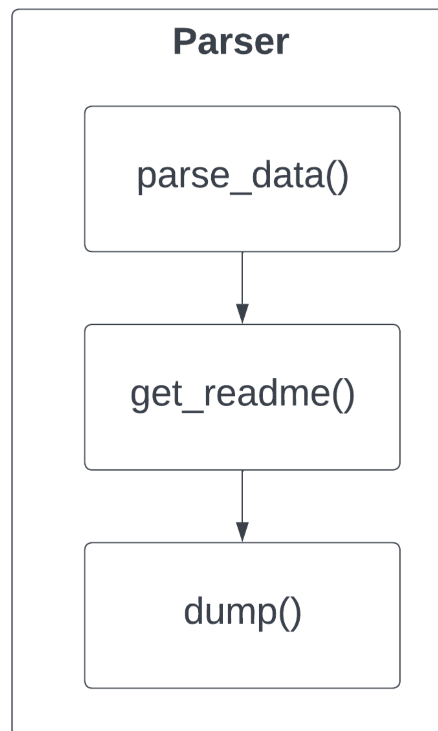

Figure 4.1.1: Parser module functions

4.2 *README* Scraper

The core purpose of the *README* Scraper is to find any information that could be absent in a package's entry in the aggregate.meta file, yet present in its *README* file. This information could include a package's test command, dependencies, or anything that could be in a field in the aggregate.meta file. A secondary functionality of this module was to find any images or links within the *README* and to try to convert their paths to absolute paths. Oftentimes, when a software developer creates a *README*, they will want to use images of their software in action, or link to certain parts of their code. Usually, they will store everything in their software repository, so all file paths for images and links can be relative paths, branching from the root directory of the repository. Since we decided to pull all of the *README* files out of their home repositories, we needed to convert these file paths to absolute paths, stemming from the URL provided in the package's aggregate.meta entry. We decided to do this in the *README* Scraper module because it was already looking at the *README* files for each package.

The first thing the *README* Scraper needed to achieve was to find what information was missing from each package's metadata entry. Fortunately, we already created *JSON* files for each package. If we just looked at what metadata was NULL within a package's *JSON* file, we would have a list of the missing information for that package.

The next task was to take that missing information and scrape through the *README* for each package, stopping to change any relative paths to absolute ones along the way. To do this we would simply look through the *README* for the information we were looking for. If we found it, we could save that information, if not, we would move on to the next field that was missing in the package's metadata. At the end of this process, we would have all of the

information we could find, and we would have solved our problem with broken links and images.

Finally, we could take all of the information we found within the *README*, and fill it back into the package's *JSON* file. Now, we were sure that we could present our users with as much information as possible about each and every Zeek package.

This functionality is implemented using the following functions, as shown in the diagram below.

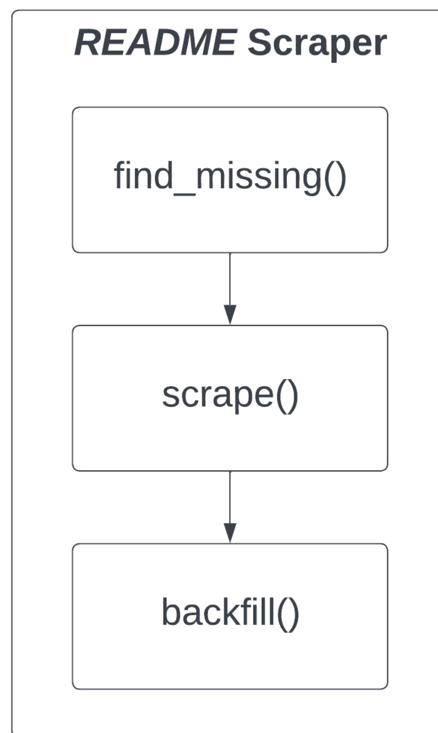

Figure 4.2.1: *README* Scraper module functions

4.3 Search Module

The core functionality of the Search module to be implemented was just that: search. We needed to develop a ranking algorithm that would provide users with relevant results based on their search query. To do this, we turned to the Okapi BM25 ranking algorithm. This algorithm is a modified TF-IDF(term frequency-inverse document frequency) algorithm, a family of algorithms commonly used in the text-based search domain. This algorithm would first rank each package based on the frequency with which each term in the search query appears in its *README*. Essentially, if the search query was “the dog”, the algorithm would sum up the number of instances of “the” and “dog” in each package’s *README*. Then, it would determine the number of *README* files in which each term appeared. Then it would weight the number of instances of a given term for a given *README* against the number of total *README*s in which it appeared, and add each of the weighted values for each term together. This would provide the algorithm with a sense of how much a term in the search query actually mattered. Terms that appeared in many different *README* files would not contribute much to the overall score, as more popular terms would be less likely to be a good differentiator between packages. So in our toy example “the” would likely appear in every single package’s *README*, while “dog” would appear in very few, maybe no, *README*s. So, the algorithm would ensure that “the” would not contribute much to a package’s overall score, while “dog” would contribute much more.

After we generated these scores for each individual package, we would need to bias them. We decided to boost the scores for packages that have names matching the search query. Essentially, if a user wants to find the package “HASSH”, and they search for “HASSH”, we should make sure that is the first package that is returned by the ranking algorithm.

Next, we sorted these results from most to least relevant, as users would want to see the most relevant results first.

Finally, we implemented a cutoff function too. This function would take the sorted list of package scores, and remove the worst-scored packages from the list. We decided which scores would be considered the “worst” based on the behavior of our ranking algorithm. Early on, we noticed that our ranking algorithm had asymptotic behavior. Essentially, all of the least relevant packages would all receive the same exact scores. These scores would not be consistent from search to search: a search for “HTTP” may have 8 packages all receiving the lowest score of -0.88, while a search for “SSH” may have 200 packages all receiving the lowest score of 2.22. But, all we had to do was determine this lowest possible score for the search query and remove all packages that received said score. With this, we were able to ensure that our search results were all somewhat relevant to the query. This functionality is implemented using the following functions, as shown in the diagram below.

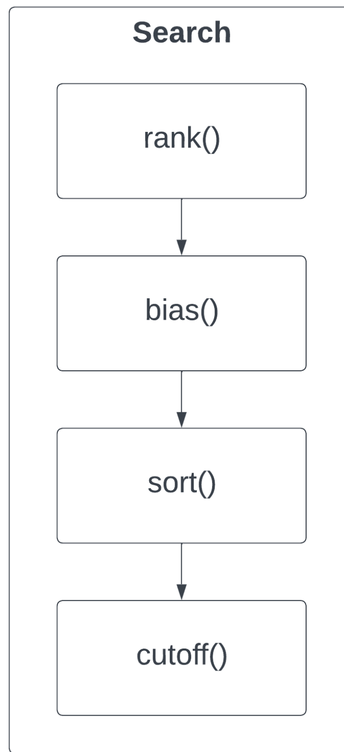

Figure 4.3.1: Search module functions

Throughout the lifecycle of our project, we were able to stick closely to our chosen solution vision and architecture. We only had a few changes from our original vision to the final product, and these were all additions to our original solution vision. Initially, we did not include the ability to properly show all links and images within *README* files. As the project progressed, we realized that this was functionality we would be able to implement much easier than originally thought. Additionally, the search cutoff functionality was not on our radar in the beginning. In demonstrating the Search module to our client, they requested that we find a way to cut off some results, and we landed on our final implementation of the cutoff functionality after some back and forth with them. Overall, the implementation of our website stuck closely to

the agreed-upon requirements with our client, and our vision for solving the problems they brought to our attention.

5. Project Timeline

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15	Week 16
Environment Setup																
Parser Module																
Search																
Frontend glue																

Over the course of the past 11 months, our team has made substantial progress towards a newly refined website for Zeek's available packages. Our focus in the first semester was on implementing a base skeleton of our project in primarily two parts, front and back end. We started with a parser module to gather package information to fill various fields for each package. As the parser module came to completion, the work on our search functionality and frontend environment came together. The integration between the search utility, frontend, and the parser module was seamless and was the first step in tying our project together. Both the search utility and frontend kept user experience as a forethought for development as the existing Zeek package website is severely lacking in those regards.

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	PRES WEEK	Week 15	Week 16
Search Cutoff																
Repo cleanup																
Cross Repo action																
Home page update																
Search fixes																
Broken links fixes																
Layout issues																
alpha demo refinement																
Final cleanup and fixes																
Unit Testing																

During this final semester, our focus was primarily on testing, fixing bugs and issues, and overall refinement in collaboration with our client. Our client, Tim, and his team helped by opening new issues and bugs within our GitHub repositories interface for us to easily navigate and track. Our main priority was to monitor those issues and implement solutions to solve them. Another focus was to implement a suite of unit tests to verify the functionality implemented for this website. Proper checks to ensure the accuracy of our parser module to collect the correct information, assurances around our search functionality to return the proper results, and verification in our front end to display information correctly and reliably. All this in preparation for demonstrations to our client for the last rounds of testing and feature input for final handoff.

6. Testing

6.1 - Unit Testing

Our goals for unit testing are to ensure that functions respect boundary conditions, return proper data types, throw the correct errors when inputs are malformed, and return reasonable values. We want to ensure that the core backend API (parse, scrape, and search) of the website behaves as expected for many different conditions. This ensures that if the Zeek team makes changes to the front end of the website, there will be no problems interfacing with the back end. Essentially, we utilized unit tests to make it as easy as possible for future developers to modify the website without breaking the whole thing. To help us with our unit testing, we used the Pytest Python library to write all of our tests. This allowed us to write simple, readable test cases, furthering our goal of using tests as a way to allow future developers to easily modify the application. Additionally, we were able to test return types, boundary conditions, malformed inputs, and reasonable outputs for every single unit. This means that each unit of code should have at least four test cases written to be considered "wholly covered". As previously mentioned, we will be testing each unit within the backend of our system, which is defined as all of the code within our backend API, which is defined as all code based within the API directory of the codebase. This code contains all of the core modules and functionality that are required to present package information to our users. Properly testing this code helps ensure that users are not presented with strange errors served up by Uvicorn, which is our web server, but instead see what they are expecting to see every single time they interact with the website. The main modules we are testing are the parser, *README* scraper, and search modules.

6.2 - Integration Testing

Our integration test suite has assurances around the data collected from the parser and README scraper modules, searching this data with our search engine, and properly displaying this all to our front end for the user to see. Instead of testing each component individually ensuring it operates on its own, we make sure that these components work in conjunction with each other. Since integration testing is similar to unit testing, except for more components of the system, we will use the same Pytest Python library for assurances throughout our project. In addition to using this framework, we will want to ensure proper error checking and handling is implemented in each module so we can maximize the accuracy of the displayed data. Given the plethora of data we are collecting from our two main methods, metadata file parser and README scraper, we want to create assurances around how we are collecting data and displaying it graphically to our front end. Data collected through the parser should contribute to the various tags that Zeek has for each package allowing for easy browsing on our site. Data collected through the README scraper should provide further details for each package and additional information for users to search for. The main goal behind these two collection modules is to complement each other as far as package information is concerned, meaning the scraper would fill in missing information for the parser for easier browsing. For search, we want to ensure that results are ranked correctly when trying to browse the collected information from the parser and README scraper meaning data transmission from all of these modules in tandem will produce our desired results.

6.3 - Usability Testing

To conduct our usability testing, we had the help of our client as well as other available members of Zeek to conduct a series of tests on the website to ensure it is not only up to their standard but also analyze how the developers of the company will use their future product. Including the Zeek team. This group will allow us to get a good pool of data to ensure there are no usability issues at the time of submitting the project along with the average user flow of the site. Being able to analyze how the average user will interact with the site will allow us to make any necessary changes to the site to make it more usable even in the event of there being no usability issues. The main focus of these tests was to ensure users could browse through every page with no issues on the user end. On top of this, we wanted them to test key features such as searching for packages, finding specific packages and examining their details, and finding a packages' repository to name a few examples. During this testing we analyzed how the users perform each action, taking note of what they do and how they perform on each task. As a result, the data we collected was qualitative with a focus on how the user interacts with the site. We feel this is the most appropriate based on the content of our site being informational about packages and the data within them. There are also not many complex things outside of the backend that users will be able to analyze besides the search feature. As a result, we care mostly about the steps they take to complete the tasks as well as ensuring that relevant packages appear for search queries that we and the Zeek team have determined as popular or relevant.

7. Future Work

Even though our team is fully satisfied with the product we were able to give our clients, there's always some work to be done on any project. One thing we felt that could be done in the future is implementing Github integration. However, this is a paid service so it would be up to the team of developers at Zeek to decide if they would like to apply it. In our case and scope of requirements, we felt that we did not need to integrate with the API directly, but Zeek could use Github integration to enhance their workflow.

Furthermore, more work that could be done on the package manager site is implementing the standardization of package tags. Although it was out of our scope of goals to reach, we believed that setting a standard for package tags could make the package manager more efficient. To implement this it would require communication with developers at Zeek and package authors in general.

In the same light, standardization within package metadata would also greatly increase productivity at Zeek. By doing this Zeek can ensure that every package would provide users with every detail of said package. However again, this would be up to the developers at Zeek to effectively communicate and set standards to their developers and users.

8. Conclusion

To wrap things up, our team's journey to enhance Zeek's network security package website has been a comprehensive and collaborative effort over the past year. The need for advancements in the cyber security world is desperately needed and we are proud to be contributors with our work done for Zeek.

Our focus on revamping Zeek's package manager website stemmed from the acknowledgment that the existing platform had shortcomings, particularly in the user experience and the site's current search feature. The outdated design and subpar search engine prompted Zeek to partner up with our team to develop a new website. We preserved Zeek's familiar look while incorporating a modern aesthetic, addressing issues with the search engine's algorithm and structure.

The chosen technologies including Python, FastAPI, Jinja, and Docker, were carefully selected to ensure simplicity, portability, and efficiency. Trunk-based development facilitated seamless collaboration within our team, allowing us to tackle tasks concurrently without the risk of merge conflicts. Furthermore, requirements were meticulously considered, both functional and non-functional, to meet Zeek's expectations. The implementation of the Okapi BM25 ranking algorithm in the search engine, coupled with efficient metadata parsing and accurate *README* scraping, ensured the retrieval and presentation of relevant information to users.

Our project timeline was divided into two semesters, with the initial focus on establishing th core structure and functionality and the final semester was dedicated to testing, bug fixing, and overall refinement. The collaboration with the Zeek team plays a crucial role in identifying issues, opening bug reports, and providing valuable input for feature enhancements.

As for testing, we performed unit testing, integration testing, and usability testing. These factors played a pivotal role in ensuring the reliability, accuracy, and user-friendliness of the platform. Unit tests were implemented to validate individual modules, while integration testing verified the seamless interaction between them. Usability testing involved collaboration with the Zeek team to assess the user experience and address any other issues.

Although we are satisfied with the product delivered to our clients we've identified areas for potential future work, including the consideration of paid Github integration. Additionally, we suggest implementing standardization for package tags and metadata within Zeek's package manager to enhance overall efficiency and productivity. Which could be done with effective communication and decision-making by Zeek's developers.

Our team's commitment to delivering a refined Zeek package manager website involved overcoming challenges, adhering to best software development practices, and maintaining open communication with the client. The final product reflects a cohesive solution that not only meets the immediate needs of Zeek but also positions the platform for future enhancements and adaptability in the ever-evolving landscape of cyber security.

9. Appendix A: Development Environment and Toolchain

This section will explain how to set up a machine to further the development of this project.

A.1 Hardware

The hardware requirements to be able to develop, test, and run this website are fairly limited. In order to properly test and run this website, any machine should have Docker installed, and basic requirements for a given development machine will need to be the same as the minimum requirements that Docker has. An additional 1GB of disk space is required in order to be able to properly build a Docker container image for the website.

A.2 Toolchain

No special IDEs, plugins, or software were used to develop the software for this project. The only specialized software required for this project is Docker, as previously mentioned, along with git. Once Docker is installed on the development machine. Follow the instructions to stand up the Docker container that can be used for development and testing. To develop outside of the Docker environment, the chosen development environment will need to have a handful of Python libraries installed. These libraries can be found within the requirements.txt file inside of the zeek/package-website Github repository. These Python libraries contain all of the core software needed to develop, test, and run the package website.

A.3 Setup

To set up the development environment within a Docker container, first ensure that the development machine has both Docker and git installed. All commands within this section are

meant to be run from the command line. First, run the command `cd <project_directory> && git clone https://github.com/zeek/package-website`. Next, run the command `cd ~/package-website && docker build -t zeek_website . --no-cache`. Then, run `docker-compose up -d`. Use the command `docker exec -it website /bin/bash`. This will give you access to the container's command line. From here you can make changes to the website's code using and command line editor, and see them reflected at `0.0.0.0:9000`. To shut down the container you can use the command `docker stop website`. To start the container up again, you can use the command `docker start website`. Finally, use the command `docker-compose down` to completely remove the container.

A.4 Production Cycle

As all of the code in this project is written in Python, there is no true "edit-compile-deploy" process. All changes to the code will be reflected in the website whenever they are made. Just note, the Parser and *README* Scraper modules are run every time the website is restarted, along with whenever changes are made to the aggregate.meta file. So, to see changes to these modules, either restart the website, or make a simple change to the aggregate.meta file. Changes to the other modules should be reflected in the local instance of the site, again accessed at `0.0.0.0:9000` immediately.