Software Design

Team ZAM

28 September 2023

Instructor: Michael Leverington

Sponsor: Tim Wojtulewicz

Mentor: Vahid Nikoonejad Fard

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

Version: 1.1

Table of Contents

1. Introduction

When deciding what precautionary actions to take to secure an organization's data, network security is a crucial consideration. By prioritizing network security, businesses can save a ton of money and relieve themselves of the fear of falling victim to cyber attacks. In a 2018 report[1] by The Center for Strategic and International Studies (CSIS), they discussed the economic impact of cybercrime and revealed that nearly $600 billion is lost annually. Fortunately, we have Zeek: an open-source network traffic analyzer software that provides a powerful and versatile set of tools for network security monitoring. Zeek also has the capability to measure performance and perform troubleshooting on a given network. These services all provide assistance concerning defending against malicious attacks.

We have partnered with Tim Wojtulewicz, a senior engineer and current release manager for Zeek. Tim and the Zeek team aspire to supply an open network monitoring framework that is able to empower their users by providing them with substantial insights into the activity of their systems. Zeek supports all users in need of their services ranging from simple home offices to the largest and fastest research and commercial networks. Zeek strives to build and support a dynamic community of developers, educators, and users. To achieve this, developers aim to strengthen the Zeek ecosystem, allowing all users to deeply understand and defend their networks.

One of the ways Zeek has grown their ecosystem is by implementing the Zeek Package Manager Website[2], which allows users to create and publish their own third party scripts and plugins. By doing this Zeek is able to consistently provide users a wide variety of tools to enhance their network analysis.

[1] http://csis-website-prod.s3.amazonaws.com/s3fs-public/publication/economic-impact-cybercrime.pdf
[2] https://packages.zeek.org/

2. Implementation Overview

Our team's goal is to take the remnants of the current Zeek Package Manager website and give it a refreshing new look and feel. The current look of the website is deemed to be outdated and gives the impression of being a website that is in need of a modern upgrade.

The first and main problem we want to solve is the current search engine. As of now, the search engine is not up to par with what the developers at Zeek desire as it can oftentimes return search results that are redundant or irrelevant to what the user searched for. Through extensive team analysis of many different ranking algorithms, we plan on doing this by implementing a new search algorithm called Okapi BM25. We have decided to use BM25 because we want packages with long descriptions to score higher, as they are more likely to be easily used by our clients. Although it is slightly slower than other algorithms we analyzed, we believe it will still be fast enough to keep our website responsive, while providing the best results to our users.

Furthermore, we plan on using the FastAPI web framework for integrating the back-end and front-end of our website and developing RESTful API's in Python. FastAPI utilizes type hinting in Python to ensure the data is accurate during validation. This also allows this API to be utilized across front-end different systems if need be. This framework is also open source and well maintained being a secure choice for our system. Along with that, FastAPI utilizes a templating engine, Jinja, for inserting data into html files.

When facing the challenge of revamping the look of the Zeek Package Manager website, we plan on using simple CSS tools to make the website feel more modern and refreshing. However, we still plan on preserving the overall look and feel of the Zeek website so that the Zeek team and its users aren't met with a completely unfamiliar product.

3. Architectural Overview

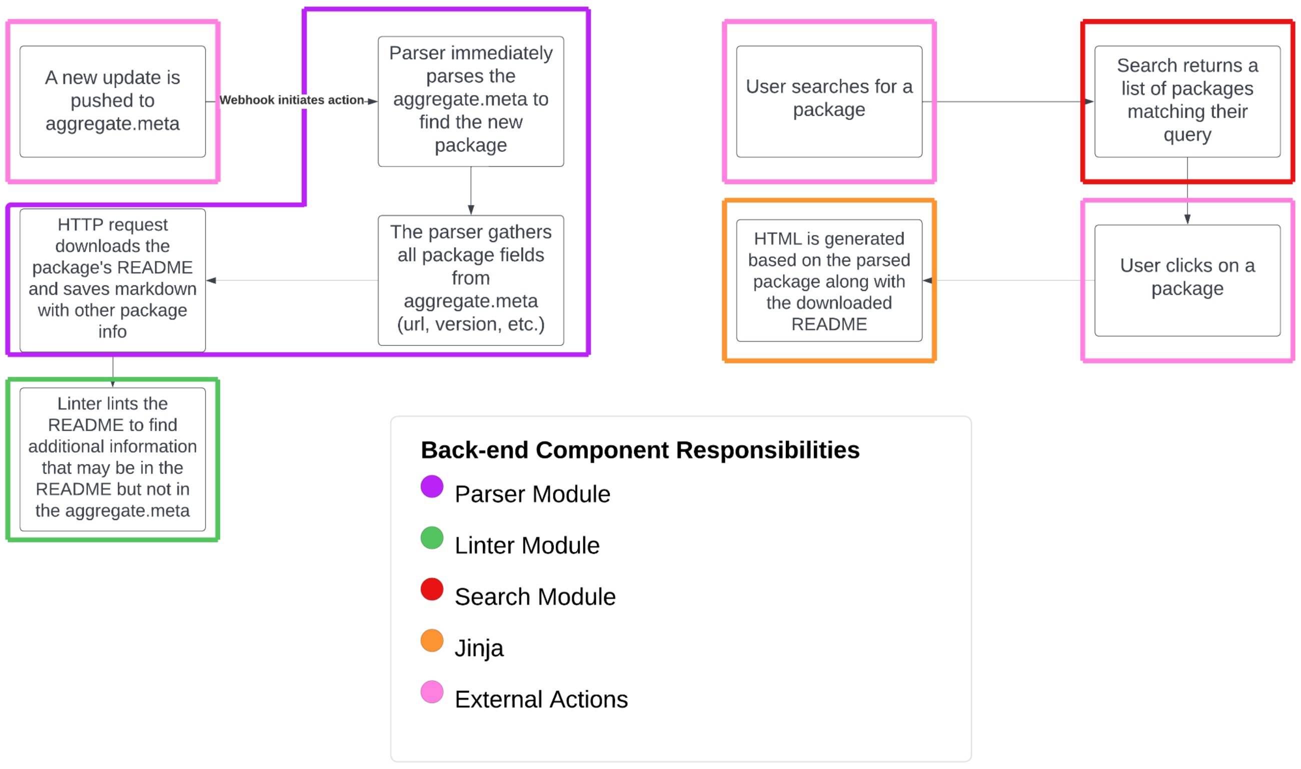

A new update is pushed to aggregate.meta
Webhook initiates action
Parser immediately parses the aggregate.meta to find the new package
The parser gathers all package fields from aggregate.meta (url, version, etc.)
HTTP request downloads the package's README and saves markdown with other package info
Linter lints the README to find additional information that may be in the README but not in the aggregate.meta
User searches for a package
Search returns a list of packages matching their query
User clicks on a package
HTML is generated based on the parsed package along with the downloaded README
Back-end Component Responsibilities
Parser Module
Linter Module
Search Module
Jinja
External Actions

Figure 3.1 Back-end Components

Figure 3.1 shows a high-level design of the various components that we have implemented, along with the responsibilities they will have as updates are made to the website via developers or users alike.

This figure visualizes the key responsibilities and features of each component. In this diagram, we showcase how the various components will communicate and work with each other. These components include the parser. linter, and search module, Jinja, and external actions. The main communication mechanism for this diagram demonstrates two main actions, when a package is updated or a new one is added to the site and the instance of a user searching for a package.

In the case that a new update is pushed to the aggregate metadata, a webhook initiates an action that instantly gets parsed to find the new package. Once the parser gathers all the required information an HTTP request downloads the packages README and writes a markdown with the package info, which finally gets passed to the linter module to find additional information that may be in the README but not the aggregate metadata.

Furthermore, when dealing with the instance of a user searching for a package, our search module will return a list of packages that best match their query. Once the user clicks on a given search result HTML is generated based on the parsed package along with the downloaded README.

4. Module and Interface Descriptions

4.1 - Parser Module

The parser module is the first component of the back-end of the Zeek package website. The goal of this module is to go through all of the packages within Zeek's packages repository to store vital information about each package within a *JSON* file. Each package within the packages repository is listed within the repository's *aggregate.meta* file. This file contains metadata information about each and every Zeek package. The role of the parser is to step through this metadata file, package by package, and pull important information about each package. Information from package to package varies - typically, we will find credits, aliases, GitHub URLs, script directories, versions, build commands, and versions. The parser gleans whatever available information is stored in a package's entry in the metadata file.

After the parser module gets information about each and every package, it then gathers a *README* file from each individual package's GitHub page. While information can vary from entry to entry within the *aggregate.meta* file, each package entry does contain a URL to it's GitHub repository. Using this URL, the parser is able to make a request to GitHub for that package's *README* file. The parser needs this information, as the *README* is where most of the information for a given package can be found.

After getting *README*s for each package, the parser module stores information about each package in its own *JSON* file. *JSON* files are used over a database as there are only about 220 packages, and it has been decided that using a database to store so little would be overkill.

The parser module is the module of the back-end of the website to go into action. The other back-end modules, such as the search and the linter require information stored within the

JSON files created by the parser module in order to operate. As such, the parser module must be called upon whenever the website is first initialized. Additionally, the parser must occasionally run, so that any changes made to packages are reflected by the website.

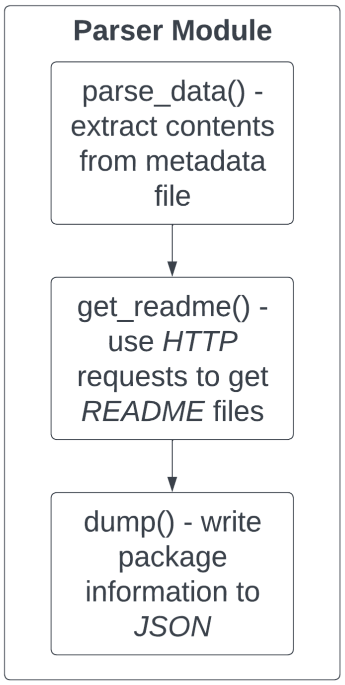

Parser Module
parse_data() - extract contents from metadata file
get_readme() - use *HTTP* requests to get *README* files
dump() - write package information to *JSON*

Figure 4.1.1: The sub-componenets of the parser module

The three actions that are done within the parser module are the gathering of data from the Zeek package repository's *aggregate.meta* file, making *HTTP* requests to gather *README*s for the packages, and finally, dumping all of the information gathered to *JSON* files. These actions all happen linearly, meaning that the *aggregate.meta* must be parsed before *README*s can be gathered, and *README*s must be gathered before the data is dumped.

The public interface for this component consists of two functions: parse_data() and dump(). The function parse_data() does not require any arguments, and simply saves all of the relevant information for each package into a dictionary. It calls the get_readme function during its execution, as these steps should not be split up. The function dump() requires the parse_data() function to be already called so that it can have the information about each package. It simply writes the contents of each package's dictionary to a file. The reason why these two functions are called separately is because there may not be a need to rewrite files for each package whenever the parse_data() function is called. This is because many packages are not frequently updated, so rewriting their *JSON* files when there is no new information would be inefficient. To make use of the parser module, one would just need to import the functions, as they would with any other Python library.

4.2 - Search Module

The search module is the main way through which users of the Zeek package website will find packages. The search module will be called whenever a user inputs a search on the website. The module implements the Okapi BM25 search algorithm to search through each package's *README* to figure out how relevant each package is to the search query. This algorithm is a term frequency-inverse document frequency (tf-idf) algorithm. This means that the algorithm considers how often a term within the query appears within a package's *README* file - the term frequency. It also calculates the inverse document frequency - the measure of how much information this term actually provides. Essentially, the more documents a term within the search query appears in, the less information it provides. This ensures that very common terms do not skew search results and provide irrelevant packages at the top of the search results. Furthermore, since some packages have been deemed rather relevant to certain search queries, we have added an additional bias checking if the search term is contained in the name of the package. Finally, we have created a cutoff point, which will remove particularly irrelevant packages from the search results, so as not to lead the user astray. As searching relies on *README* files gathered by the parser module, the parser module must run before any search can take place. After the parser has been run, and all of the *JSON* files have been populated, the search module can be called upon at any time.

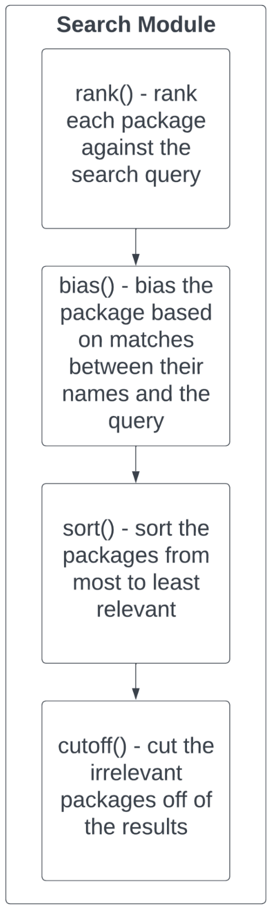

Search Module
rank() - rank each package against the search query
bias() - bias the package based on matches between their names and the query
sort() - sort the packages from most to least relevant
cutoff() - cut the irrelevant packages off of the results

Figure 4.2.1 - The sub-components of the search module

The search module's subcomponents include the rank(), bias(), sort(), and cutoff() functions. The rank function uses the Okapi BM25 algorithm to generate scores for each package. It expects the list of package names along with the search term to be passed as arguments. The rank() function returns a list of the score values for each package. From there, the scores can be sent to the bias() function along with the search query, which will add a bias value to a package's score if it meets the bias criteria. Then, the packages are sorted from highest score to lowest. Finally, the sorted list of packages and scores is sent to the cutoff function, where the line of irrelevance is established. All packages with a score below the line will be omitted from the results. Much like the parser module, these steps are all linear, one must be taken before the next one is taken.

All of this is wrapped in a simple search() function to provide a clean interface for interaction with the front-end of the website. So, all a front-end developer needs to do to make a call to the search module is to call this function and pass along the search query. They will receive a list of results containing the relevant packages, along with their scores as determined by the search algorithm.

4.3 - *README* Scraper module

The *README* scraper is the final module within the back-end of the system. The job of the *README* scraper is to sift through each *README* file gathered by the parser module to find important information to be highlighted on a package's page on the website. Ideally, all of this information would be contained within a package's entry in the *aggregate.meta* file, but through some digging, we have determined that some important information, like build commands, are often omitted from a package's entry in the *aggregate.meta*, and instead included in the package's *README*. We hope to have centralized locations on a package's page for such information so that users do not have to unnecessarily skim through a long *README* file to find one specific piece of information. To do this, the *README* scraper must first identify what was missing from a package's entry in the *aggregate.meta* file by comparing what fields have been filled in to what fields could be filled in but are not. Using this process of deduction, it can decide what it needs to look for in a package's *README* file. Then, it can step through the markdown in the *README* looking for specific terms that would match empty fields from the *aggregate.meta* file. The *README* scraper relies on the parser module, as it needs for the *aggregate.meta* file to be parsed into *JSON* files, and for *README*s to be gathered.

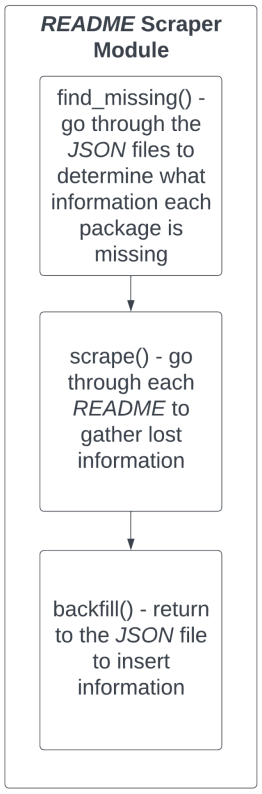
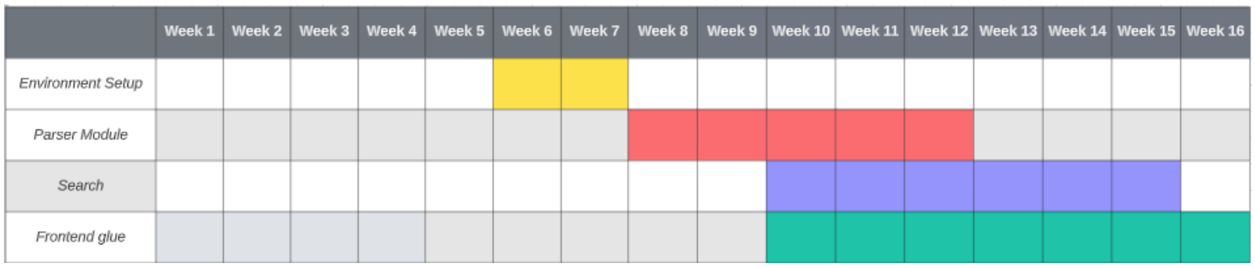

README Scraper Module
find_missing() - go through the JSON files to determine what information each package is missing
scrape() - go through each README to gather lost information
backfill() - return to the JSON file to insert information

Figure 4.3.1 - The sub-components of the *README* scraper module

The find_missing() function goes through the *JSON* files to determine what is missing from each and creates a list of missing fields for each package, which it will pass on to the scrape() function. The scrape() function uses this list of missing fields, along with regular expressions, to find any information about these missing fields that is contained within the *README*, and pass a list of this information to the backfill() function. The backfill() function uses this list to open each *JSON* file, and put the information into the proper field within the *JSON* file.

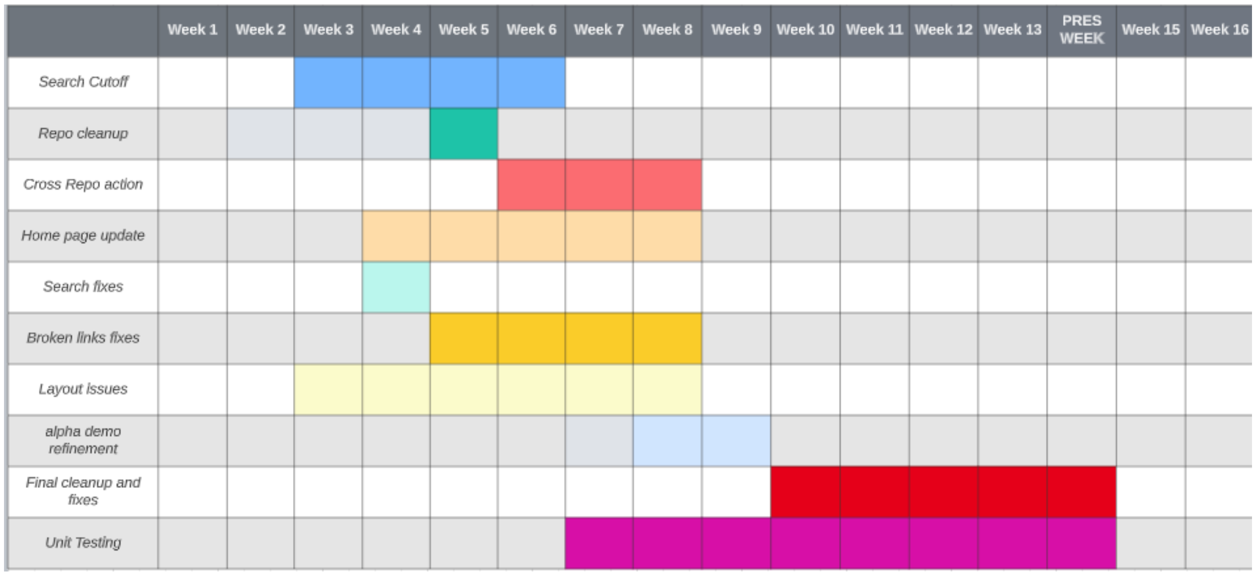
All of this functionality is incorporated into one singular scrape_readme() function that will serve as the interface for the front-end to access this module. This function can be called by the front-end of the system, after the parser module has been called, and it can carry out all of the previously listed functionality of the *README* scraper.

5. Implementation Plan



	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 9 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 backend. 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 is primarily on testing, fixing bugs and issues, and overall refinement in collaboration with our client. Our client, Tim, and his team open new issues and bugs within our github repositories interface for us to easily navigate and track. Our main priority is to monitor those issues and implement solutions to solve them. Another focus is 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 frontend 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. Conclusion

As technology continues to expand, this calls for the network security sector to keep pace and evaluate how new risks can be mitigated. Fortunately, the Zeek Package Manager has been an excellent means to allow users to develop and share packages to enhance network analysis. Nonetheless, developers on the Zeek team feel that the user experience of the site can be reworked and will then alleviate the current technological challenges users face.

The problem revolves around improving the current user experience by making discovering and managing new packages easier. This is detrimental to the workflow because it adds to the time spent filtering through packages and can cause users to lose confidence in the utility of Zeek. Another problem we are faced with is revamping the quality of the website. Zeek wants the package manager website to have a modern up to date look while still preserving the overall look and feel of Zeek.

As a team, we are confident in our plan to manufacture a new website that will retain a modern, sleek design and will provide users with an intuitive experience with regard to discovering packages. On top of that, we plan on implementing a new search engine utilizing the BM25 ranking algorithm. While also developing a new package parser and scraper that allows us to generate the standard information of each package and scrape the captured *README* files to potentially gain more information of a given package.

In this document, we discussed the features and functions of our proposed solution, including the actual implementation and an overview of the architecture. We also highlighted the potential risks of our system and our current plan of execution moving forward. As a result, this solidifies a thorough set of requirements that will guide the fruition of the new Zeek Package

Manager website, and we look forward to providing Zeek and its users a newly enriched user experience.