Technological Feasibility Analysis

Team ZAM

7 April 2023

Instructor: Michael Leverington

Sponsor: Tim Wojtulewicz

Mentor: Daniel Kramer

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

Table of Contents

Technological Feasibility **1**

Table of Contents **2**

1. Introduction **3**

2. Technological Challenges **5**

3. Technology Analysis **6**

3.1 Search Engine 6

3.2 Updating the Website 9

3.3 Storing Package Information 10

3.4 Integrating back-end with the front-end 11

3.5 Reworking look of the website 12

4. Technology Integration 17

5. Conclusion 17

1. Introduction

As technology continues to evolve and become more innovative, the volume of Internet of Things (IoT) devices has skyrocketed. While the convenience of controlling appliances from a smartphone or no longer relying on conventional meters for tracking utility usage is undeniable, it also brings a host of new potential network vulnerabilities.

Zeek, an open-source network traffic analyzer developed by Corelight, is a powerful tool used by many organizations looking to secure their networks and investigate any suspicious activity. While its primary use case is as a network security monitor (NSM) for identifying and mitigating potential threats, Zeek also supports a wide range of traffic analysis tasks beyond the security domain, such as performance measurement and troubleshooting. With its passive monitoring approach and versatility, Zeek is well-equipped to help organizations optimize their networks while maintaining a high level of security.

As an organization, Corelight is dedicated to providing innovative solutions for its users, so in 2016, they implemented a Zeek Package Manager website to allow developers and users to create and publish their own packages, providing a vast array of tools that can be used to enhance network analysis. However, when it comes to discovering and managing packages, the current workflow can be a challenge for users. With no easy way to discover new packages, inconsistent tagging practices, and a search engine that is currently considered substandard, users may be missing out on valuable tools that could enhance their work. Additionally, developers at Corelight feel that their package manager website has an outdated look and would like for us to elevate the overall look and feel of it, aiming for a more modern, sleek look.

This technology feasibility document will act as a design rationale for this project. In it we will outline our design decisions, technological challenges, and how its functions are relevant

to the overall outcome of the product. We will also discuss the alternatives for addressing the challenges we are anticipating running into and our candidate solutions for them. Furthermore, we will explain how all of the alternatives were evaluated and how we arrived at solutions as a team. This technological feasibility analysis is a great way to make sure we have all of our bases covered and can launch into solution design within a realistic framework for envisioning our final product.

Overall, our solution vision revolves around reworking the search feature, storing every package without the weight of a database, and automating the website to update with every new package added, as well as any documented updates. Along with that, our solution also includes revamping the look and feel of the Zeek package manager to look more modern and appealing to users.

2. Technological Challenges

In the design and implementation of our package management website, we envision a few high-level hurdles. For our proposed solution most of our technological challenges lie in the back-end. The first major hurdle is to create a search engine for the site. This search engine needs to be able to search through packages by name, associated tags, and even external files such as the README file in a package's GitHub repository. A second major hurdle is making updates to our website based on a central metadata file containing information about every single Zeek package available. Our website needs to post updates to package information whenever an update is made to the metadata file. A third major challenge would be to design a simple system for storing and retrieving package information. Our clients have already stated that they think a database would be an over-engineered solution to this problem, so we need to figure out a way to effectively store state without too much complexity. A final challenge is to lint packages. We need to be able to lint packages for information that may be relevant to the user so that we can properly highlight it on the site.

On the other hand, when it comes to the front-end of the package manager website, we have two main obligations to fulfill. One of them being the obvious challenge to any website, integrating the front-end with the back-end. Lastly, our client feels that their current package manager website appears to be outdated, so our goal is to rework it with a modernized, sleek look.

3. Technology Analysis

Through preliminary discussions with our client we have identified the following technological issues as the most pertinent to project success: the search engine, package information storage, and updating the website whenever a change is made to the central metadata file.

3.1 Search Engine

Our search engine will need to function just like any other search engine users have experienced in the past. It needs to work well for the users, as we expect it to be the primary means through which they find new packages to use.

There are a few desired characteristics when it comes to our search engine. The most important characteristic is accuracy of results. Currently, if you search for something on the Zeek package website, the results may not match the search query. For example, if one were to search for ‘ssh’, the third result is the emojifier package: a tool that summarizes Zeek logs as emojis. The goal of the search engine on our site is to return the most relevant results to the user, without the inclusion of any irrelevant results. Another desired characteristic is speed. The search process has to feel snappy and responsive, much in the way that a Google search feels instantaneous. This is important, as users will be less likely to use the search engine if it is slow, and they will have trouble finding packages that may be of use to them. A final desirable characteristic with the search engine, much as with the rest of our website, is ease of maintenance. Eventually, we will have to turn over this project to a team of engineers at Corelight Inc., and they will need to be able to solve any problems that arise as the website matures. So, if we make the search engine easy to maintain, they can update it as Zeek grows and more packages are created.

There are many algorithms used to rank results, one of which is tf-idf (term frequency-inverse document frequency). We were introduced to tf-idf just by searching for document ranking algorithms online, as it is one of the most popular algorithms. The idea behind ranking results based on term frequency has been around for a while, one of the first forms of term weighting can be traced back to Hans Peter Luhn's information retrieval work in the 1950s. It was not until 1972 when Karen Spärck Jones first introduced the idea of inverse document frequency, penalizing documents that contain a high frequency of really common search terms, such as "the". This algorithm has since gained popularity in digital libraries, as most library recommender systems use tf-idf.

Another algorithm used to rank search results is PageRank. Famously developed by Larry Page and Sergey Brin at Stanford in 1996, PageRank became the first ranking algorithm employed by Google. Now, Google's patent has expired, but they still use PageRank in conjunction with other algorithms. PageRank works by ranking results that have more links to the page higher. It is thought that if many web pages link to one webpage, that webpage is very important.

BM25, also known as Okapi BM25, is yet another ranking algorithm. BM25 has many similarities to tf-idf, as when it was being developed by Stephen E. Robertson and Karen Spärck Jones in the 1970s and 1980s, they took much inspiration from tf-idf. It differs from tf-idf in that it takes into account the length of a document, so that a document will not be unduly penalized for containing common words such as "the" much more than other documents if it is substantially longer. BM25 has found use in many of the same areas as tf-idf, with one more famous example being its implementation at London's City University.

In evaluating these ranking algorithms, we had a difficult task before us. It would take much effort to actually implement them in code and try them out with even a limited set of Zeek packages. So, we first decided to think about what each algorithm brought to the table. With this approach, we quickly ruled out PageRank. First, looking at how packages link to one another would take a lot of time and computational effort, most likely for little gain. Currently, there are about 220 Zeek packages, which are unlikely to link directly to one another, so searching each one for links would return a lot of empty results. Additionally, we believed that one package would not be better than another simply because other packages depend on it. This would likely push more generic packages above more specialized ones, yet our users more likely want to find specialized packages. Comparing tf-idf to BM25 was much more difficult. Both algorithms were so similar that it seemed that we could not go wrong with either. Thankfully, this whitepaper[1] by KMW Technology helped us sort out the differences between the two algorithms. It tactfully explains the differences in the implementation of each algorithm, and how BM25 requires a bit more computing power than tf-idf, but it can provide much better results if there are vast differences in the lengths of documents being compared.

So, we have found that while tf-idf is a versatile, widely-used, reasonably fast ranking algorithm, it does not take into account the fact that multiple items may be wildly different in length, affecting their scores. PageRank is a very well known algorithm, and it is relatively easy to implement, and likely easy for someone unfamiliar with it to understand. However, it is not the fastest algorithm, as it requires searching each item to get results for one item. BM25 solves tf-idf's problem when it comes to differing lengths of items, but requires more effort to do so, slowing down the ranking process.

[1] https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25/#:~:text=In%20summary%2C%20simple%20TF%2DIDF,length%20and%20term%20frequency%20saturation.

	Speed	**Maintainability**	**Accuracy**
tf-idf	1	2	2
PageRank	3	1	3
BM25	2	3	1

The table above shows rankings for the three ranking algorithms when it comes to the desired characteristics of speed, maintainability, and accuracy. PageRank will be the slowest, and tf-idf will be slightly faster than BM25, as they are practically the same algorithm, just with some additional steps for BM25. We rank PageRank as the easiest to maintain because of how famous it is, and how easy it is to find explanations of it online. We ranked both tf-idf and BM25 as more accurate than PageRank, as we imagine that links will not be too common in package descriptions.

So, through this analysis, we have decided to implement BM25 as the ranking algorithm in our search engine. 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 tf-idf, we believe that it will still be fast enough to keep our website responsive, while providing the best results to our users.

To prove that our chosen approach is feasible, we will demonstrate a basic webpage, that does not need to be properly styled or formatted with HTML or CSS, that can correctly provide search results while still being responsive.

3.2 Updating the Website

The website will need to update itself anytime a new package is added to the metadata, or any time a package is updated to reflect the new updates within the package. One of the difficulties with this is the need to grab more information about a package if the information in the central metadata is not enough. This will require going to the actual package and parsing additional information not found within the metadata file to accurately reflect the new changes within the package if they are.

This creates a challenge with updating the website every time a package as we would need a way to store that package information without a database as our client has stated that a database would be an over-engineered solution to the problem. The solution that we find needs to serve a similar purpose without the overhead that a database system brings to a project.

The solution to not over-engineer the website with a database comes with the requested maintainability of the website so being able to automate the update process is essential as to not force additional work hours with the provided product/website. A potential solution to this problem could be writing to some type of file within the directory of the package and comparing information in the file, to the newly parsed information. Taking this a step further, we could only make these checks if the version number has been updated, otherwise we can assume that the package has not been updated. The use of Github Actions within a CI/CD pipeline will allow us to check for differences and updates within the metadata file housing important information related to Zeek packages and pushing those updates to the public facing website.

We will prove that this solution works by setting up this pipeline with Github actions, and potentially utilizing webhooks from the suggestion of our client to look for changes within the metadata as to only make changes to the site when absolutely necessary. Proving this change will

work may be difficult when trying to use the actual metadata, so the usage of a fake metadata file that we can change for initial testing may come in handy to prove that it will work within the production stage.

3.3 Storing Package Information

Package information for the Zeek project is stored in an `aggregate.meta` file located in a separate repository dedicated to packages which will aid the Zeek package website's search engine. This file does not contain a standard guideline for all required fields for example dependencies, build/test commands, informational tags, etc. are not a commonality between each package. This makes it difficult to display accurate and helpful information on the Zeek package website. To address this issue, the desired solution should be able to provide a baseline for required fields in package metadata, be simplistic enough for future maintainers and developers to use, and gather missing data by parsing READMEs of the packages with the use of regular expressions.

The implemented parser makes use of a few regular expressions and is designed to act as a TOML parser, where the first regular expression it checks for is package names. These names are stored as section headers within brackets. Once each section header is parsed, the key/value pairs are then checked until the next section header (denoted by the next open bracket [). This is done using either the `get_line` or `next_line` methods. The get_line method parses for the specified regular expression until the end of the line, while the next_line method gets the current line and any subsequent lines until the next key/value pair is recognized. Certain keys, such as tags, version, and credits, have their corresponding values parsed using the `get_line` method while keys such as description or dependencies, on the other hand, use the `next_line` method.

One alternative solution would be to require maintainers of each project to update their metadata, but this would be time-consuming and costly as most of Zeek's packages are stale. Another alternative would be to manually gather missing data from READMEs, but this would also be time-consuming and error-prone. Therefore, a proposed solution is to recommend that a standard be put in place for package metadata, and to use regular expressions to parse READMEs to fill in any missing data.

By setting a standard for package metadata, a baseline of required fields can be established, making it easier for maintainers to provide consistent and complete information. Parsing READMEs with regular expressions is a reliable and efficient way to gather missing data, and simplifies the process for future maintainers and developers. The feasibility of this solution can be proven by implementing it in a test environment and monitoring its performance, and feedback from future maintainers and developers can be used to further improve the solution and ensure its feasibility. Overall, this solution is a feasible and effective way to improve the functionality of the Zeek package website by displaying accurate and helpful information to work with a Zeek package.

3.4 Integrating the back-end with the front-end

Once the back-end is in a stable and testable state, we will need to begin integrating it into the front-end. The back-end can be tested in the back-end itself, however the users will see the results of this on the front-end. We will need to provide this functionality to the users visually while maintaining the accuracy of the results in the back-end.

Ideally, this solution would be quick without adding a lot of overhead on top of the back-end results. As stated before with the search engine, our website needs to feel snappy and responsive to prevent users from leaving the site due to things not loading. The back-end is built

with a focus on speed so this should be reflected in the front-end. Since this project will be maintained by an employee for Zeek, we want this project to be easily maintainable so the overall logic of bridging the front-end and back-end should not be overly complex. This solution should be simple and to the point.

When researching frameworks that we should use, we had three major categories that we looked at for our ideal candidate. Our first category was speed. We wanted our back-end to produce the results quickly and display just as quickly. The second thing that we looked at was maintainability. We did not want an overly complex structure for this project as that would make it more difficult to maintain after we hand over the final product. The third and final category that we looked at to determine the best framework to use was ease of development. We did not want to try to learn an entirely new language unless it was the best solution, but we also needed to ensure that our project can easily be looked at and developed upon by the next developer at Zeek if need be. This means that the framework would ideally be built in a language that is easy to read and common among many programmers.

With these things in mind, we determined that one framework we would look at was Django. We all had previously heard about Django either through using it or through other projects showcased. Django was originally invented by Lawrence Journal-World in 2003, to meet the short deadlines in the newspaper while also meeting the demands of web developers. It was later released to the public in July 2005. Django is a web framework that was built for Python. It is open source with a focus on developing dynamic web applications. It uses an MVT (Model, View, Template) design system. Django does a great job in keeping their system secure and up to date preventing many different types of attacks. Many companies like Mozilla,

Instagram, and Pinterest use or have used Django in the past for their project's back-end and front-end.

Another framework we looked at was FastAPI. We discovered this framework when looking for frameworks with speed in mind. This framework was developed by Sebastián Ramírez and initially released in December 2018. FastAPI is another web framework for 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. FastAPI is utilized by companies like Microsoft, Netflix, and Uber.

Using the above criteria to compare these two frameworks, FastAPI is faster than Django with handling incoming requests as well as serving information from the back-end. As far as maintainability, both frameworks use a similar templating engine of inserting data (typically JSON) into html files using Jinja: a templating language that allows you to insert python code into HTML. This means that we can create assets that are reused within the templates and we can focus on producing a quality back-end with our core focus in mind. With the final category of ease of development, we also found that these two frameworks would be similar in terms of development time as they both use a similar structure and utilize the Python language, so we decided to use FastAPI. Both frameworks provided similar solutions to our problem so we decided to use the faster option of the two.

Framework	**Speed**	**Maintainability**	**Ease of Development**
Django	2	1	1
FastAPI	1	1	1

Looking at the table above, we can see that these two frameworks are very similar for the categories that we looked at. However, since FastAPI was able to handle more requests per second when compared to Django, we decided that FastAPI would be the framework that we would choose to develop in.

Other factors that went into the decision were security and the architecture of the system we wanted to develop. Since the architecture would be the same, security was the heavier choice of these last two. While it appears that Django is safer in some regards, the security features that Django shines in would not be as utilized within this project as we will not use a database for any actions on this project and there are no plans to use forms for Cross Site Request-Forgery (CSRF) attacks. As a result, we figured FastAPI would be secure enough for our needs and focused on our main three categories for our criteria.

To prove that FastAPI will work for our project, we will develop simple pages to show that we can send information from the back-end to the front end using API calls and routing them to pages utilizing HTML templates.

3.5 Reworking look of website

As said previously, the current look of the Zeek package manager website is outdated and needs some reworking for it to meet the standards of our clients. Right now, the contents of the website appear bleak and plain. Users are greeted with a logo, block of text, and a search bar. Our goal is to modernize the website while still preserving the overall look-and-feel of the Zeek.org website.

Preserving the overall design of the Zeek website is a key characteristic of this challenge. This is important because we don't want to make the mistake of making a website that looks completely different from its parent website. At the same time, when comparing the Zeek package manager website to other popular ones, Zeek's outdated look stands out so we want to be able to modernize it so it can be seen in the likes of other well-known package manager websites.

In this case, we don't have many alternatives to choose from. Since we are only making a sister website, our client wants us to follow a specific design scheme. However, we are not too limited on our approach to tackling this solution when it comes to overall design and setup of the website.

When observing competing websites, we noticed that their main focus is not just the search feature, they usually have some type of hook or eye-catching elements to them. We'd like to implement similar elements to our website so that users are greeted with a more eye-catching homepage and intuitive features.

Our solution starts with examining the look of the Zeek website and making note of its key design elements such as color schemes, logos, and overall feel of the website. We then want to take those elements and implement them into our new website, while still following practices of modern websites such as using readable and web-friendly fonts, utilizing negative space, and applying simple and logical page navigation.

After much consideration and analysis our team believes we'll be able to successfully implement our proposed solutions when dealing with the front-end challenges. For the duration of this project our validation process of this challenge will be to periodically check-in with our client and see if the design we're moving forward with fits the mold they desire.

4. Technology Integration

These solutions we have developed will need to come together in order to make a coherent project that we can deliver to our client. The main issues with their current site were accurately parsing through the metadata of the packages to bring relevant information about the packages to the user, as well as a search engine that returns accurate results back to the user. We believe that we have found solid solutions in order to solve these problems, but we will need to bring these deliverables to the front-end. Using FastAPI, we will successfully bridge the API that we create to the front-end in a fast, efficient, and maintainable way. In doing so, we also plan to make the website look more modern and professional, while remaining in line and providing similar functionality as other package manager websites.

5. Conclusion

Currently, Zeek's package management website leaves a lot to be desired. The site looks outdated, discoverability of new packages is a problem, and the website is over engineered for our client's needs. These issues take away from Zeek developers valuable time, which they could be spending solving network security issues. We plan to create a new search engine, dynamically update the site whenever new packages are uploaded, store packages in a simple, efficient manner, rework the look of the site, and integrate all functionality smoothly.

To achieve this, we will create a package for our search engine allowing us to add a form that routes to the required methods to find accurate search results. In addition to this, we will establish webhooks or cron jobs to monitor changes to the aggregate metadata file to automatically add or update packages as needed. Using FastAPI, we will easily be able to create

the necessary packages for parsing and searching which will allow us to insert that data into an HTML file using the Jinja templating language.

In achieving these goals, we will have a website that will allow Zeek developers to find packages easily, meaning they spend less time searching for a package, and more time creating network security solutions. Moving forward, we will implement and test all of the functionality laid out in this document, working towards a new website in the near future.