Software Testing Plan

Team ZAM

30 October 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

1. Introduction

Ensuring network safety is becoming progressively more important as technology advances and cyber-attacks pose a serious threat to organizations. Our team has partnered with Tim Wojtulewicz, senior engineer and Zeek's current release manager, intending to develop a newly refreshed package manager website with an improved search engine.

Software testing is the process of ensuring that your software's implementation demonstrates the necessary functional and non-functional qualities to execute the intended purpose. When it comes to our package manager website we have a plan that consists of performing unit, integration, and usability testing to validate the intended functionality of our software. The main modules we plan to perform unit testing on are our parser module, readme scraper, and search feature; this is to guarantee that the modifications done to the front-end side of the website do not interfere with the back-end functionality. In terms of integration testing, our goals are to verify that the data gathered by our parser and README scraper modules is accurate, accessible through our search engine, and successfully displayed to users on the front-end. Regarding usability testing, we will collaborate with our clients, the Zeek team, to run several tests on the website to make sure that it meets their standards and to examine how each feature is being used.

By following this software testing plan we will be able to assess the readiness of our product and ensure that the features on our website will exhibit the necessary functionality to be efficiently utilized by our clients.

2. Unit Testing

Unit testing is the process of testing the smallest parts, called units, of the software application. Often, the units that are tested are functions. Unit testing ensures that functions respect boundary conditions, return proper data types, throw the correct errors when inputs are malformed, and return reasonable values. Our goals for unit testing are essentially the same. We want to ensure that the backend API of the website behaves as expected for many different conditions. This will ensure that as the Zeek team makes changes to the front-end of the website, there will be no problems interfacing with the back-end. Essentially, we hope to utilize 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 will be utilizing the Pytest library to write all of our tests. This will allow 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. We will be using test coverage as a metric to determine how well-tested our codebase is. We hope to achieve one hundred percent coverage on the backend code of our application. Additionally, we aim 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 will 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 hope to test are the parser, *README* scraper, and search modules.

2.1 - Parser Testing

The parser module has a few functions that let it operate. They are listed in the diagram below:

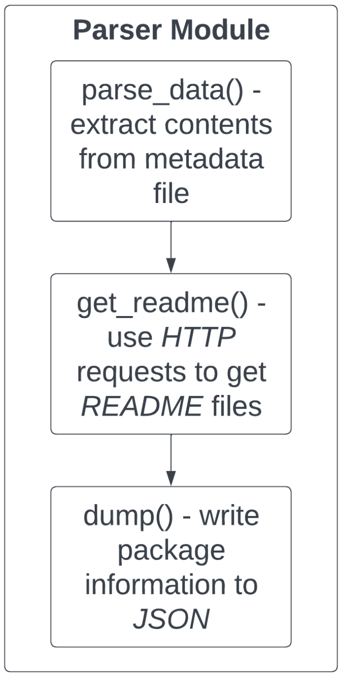
Parser Module
parse_data() - extract contents from metadata file
get_readme() - use HTTP requests to get README files
dump() - write package information to JSON

Figure 2.1.1 - Parser module functionality

The first function to test is the *parse_data* function. This function looks for the aggregate.meta file containing information about each Zeek package, and parses it to find important information. We need to ensure that this function can respect the boundary conditions of the beginning and end of a new entry in the metadata file, and the beginning and the end of the file. Fortunately, the file is formatted using the *TOML* format, so we just need to test that it appropriately reads the *TOML* for these boundary conditions. Additionally, we need to ensure that it performs correctly when information is missing, as not all packages have the same information. We need to test that the parser returns reasonable results by creating some samples to test it against. Finally, we need to ensure that it correctly returns dictionaries filled with the proper information, as the future functions expect to be able to find the data formatted in this way.

The second function to test is the *get_readme* function. This function uses URLs stored in the metadata to make requests to GitHub for a package's *README* file. We need to ensure that it can handle the boundary condition of packages not being hosted on GitHub; one package is hosted on GitLab. Additionally, it needs to be able to handle if a package is missing the URL field, or if the package does not have a *README* file. We need to check if the *README* found is reasonable, again we will create a sample to compare to, and we need to ensure that the package's dictionary has a new *README* field that is correctly populated upon the function's return.

Finally, we need to test the parser's *dump* function. This will dump each package's information into *JSON* files for later access. To test this, we need to first test the boundary condition if the file path to dump does not exist. Then we can test for the package information being *NULL* in a package's dictionary. Next, we will create some sample *JSON* to compare to *JSON* returned by the *dump* function, to ensure that the results are reasonable. Finally, we need to test whether or not the function can correctly write files to the proper location, to ensure that once the parser has been run, we can access the *JSON* files from where we expect to be able to access them.

2.2 - *README* Scraper Testing

The *README* scraper module has several functions that let it operate correctly. They are listed in the following diagram:

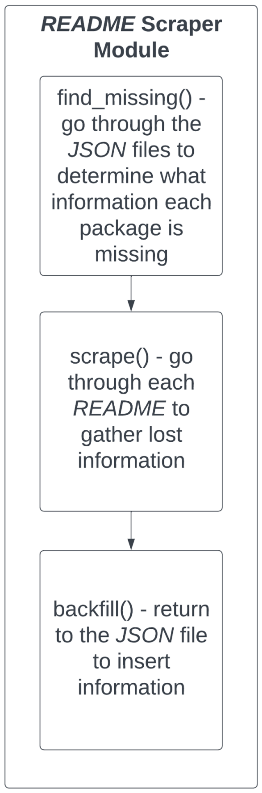
README Scraper Module
find_missing() - go through the *JSON* files to determine what information each package is missing
scrape() - go through each *README* to gather lost information
backfill() - return to the *JSON* file to insert information

Figure 2.2.1 - *README* scraper module functionality

To ensure the proper functionality of the *README* scraper, we need to ensure that each function within the module is properly tested.

The first function we need to test is the *find_missing* function, which identifies any fields that are missing a given package's metadata. This function lacks any real boundary conditions, as the input is rather binary; either a package field has information, or it is *NULL*. We must check for malformed *JSON*, and handle it properly, just in case data is corrupted on the web server for any reason. We also will create some sample packages with missing fields, just to check that the function is performing correctly. Finally, we will ensure that it is properly returning the missing fields, as a list of which fields are missing, to the calling function.

The next function we need to test is the *scrape* function. This function goes through a package's *README* file to find information to fill in fields that were missing in the package's metadata. The boundary condition we need to check for is a *NULL* input, essentially the condition of a package has no missing fields, and where there are missing fields but no relevant information in the *README*. We then need to check that it can handle strange inputs, such as lists of fields that are incorrect. Then, we can test it for correctness, by seeing if it can identify information in packages that we know contain missing fields, and information to fill said fields in the metadata. Finally, we can check that it correctly returns a dictionary with each field and its associated information.

The final function we need to test in the *README* scraper module is the *backfill* function. This function fills the information found within the *README* into the proper fields of a package's *JSON* file. The boundary conditions we need to test against are if a package's *JSON* file does not exist if the fields within the *JSON* have already been filled, and if the input to the function is *NULL.* We also need to check for incorrect input, such as fields that are slated to be filled in with *NULL* values. Again, we will use some sample packages that we know have missing information that can be found within the *README* to test if the *backfill* function gives reasonable results. Finally, we need to check that upon the function's completion, each package still has a correctly formatted *JSON* file in the correct location.

2.3 - Search Testing

The final module we need to test is the search module. The functions for this module are found below:

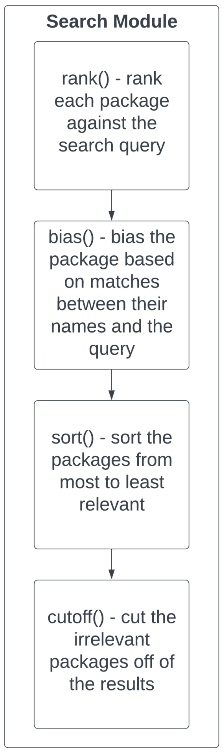
Search Module
rank() - rank each package against the search query
bias() - bias the package based on matches between their names and the query
sort() - sort the packages from most to least relevant
cutoff() - cut the irrelevant packages off of the results

Figure 2.3.1 - Search module functionality

The first function to test is the *rank* function, which ranks each package based on information found within its *JSON* file. We need to ensure that this function handles the boundary condition of the *JSON* files for packages being missing. We also need to ensure that the function can handle the boundary condition of fields, most notably the *README* field, being missing within a package's *JSON*. So long as those boundary conditions are handled, it should be able to handle any input. To test its correctness, we will have some sample rankings saved to test the function against. Finally, we need to ensure that it returns a list of packages and their associated scores upon its completion.

The next function is the bias function, which biases packages based on hyperparameters. We need to ensure that the bias function can handle the boundary conditions of the input being *NULL*, for both the rankings and the search query. We also need to ensure that it does not bias packages based on file extensions, such as *.git* and *.json*. To ensure that it returns correct results, we will save some biased rankings to compare the function against. Finally, we need to ensure that it returns the same list of ranked packages that it got as input, just with some of the scores changed based on the biasing procedure.

We need to test the *sort* function to ensure that it properly sorts packages based on their biased scores, from most to least relevant. It needs to be able to handle the boundary conditions of the rankings being *NULL* and of packages all having the same score. If it can handle these boundary conditions, the *sort* function should be able to handle any input. We will have some sample packages and scores saved in a sorted list to ensure that it properly sorted in the correct order. Finally, we need to ensure that it returns the same list of packages and their associated scores that it received as input, just sorted.

The final function we need to test is the *cutoff* function. This function finds the minimal score and cuts off any packages that received said score. We need to ensure that it can handle the boundary condition of the input being *NULL*, as that is the only real boundary condition or incorrect input that the function could receive. We need to ensure its correctness by saving some simple rankings and cutting off the lowest scores, keeping these to compare this function's results against. Finally, we need to ensure that it can correctly return the sorted list of packages and their associated scores, with the lowest scores being removed.

If we have correctly implemented these test cases, we can consider the backend units of the application to be "wholly covered" for testing purposes. This will ensure that the backend will always perform as expected, and make modifying the codebase simpler for future developers.

3. Integration Testing

Integration testing is the process of verifying the correct interactions between a project's components and modules. In our case, it is 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.

The parser and README scraper modules serve as the main entry points of our project largely in charge of the collection and storage of each package's data. The parser collects package information from Zeek's central metadata file with this information while our README scraper browses each package's README for additional information. Testing the behavior of these two collection modules with our search engine is vital to the functionality of our website and assuring all collected information is stored neatly for our search engine to return accurate and meaningful results. This means the data communicated between these modules must be verified for accuracy and ensure no loss of data to minimize unexpected results on our website.

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.

Correctly implementing our suite of unit and integration tests which will ensure the operation of each component with our integration tests that ensure modules work correctly together will provide some safety and durability for the operation of our website. This also aids future maintainers, developers, and even users of our website so that additional features can be implemented easily and our users have a satisfactory experience.

4. Usability Testing

Usability testing is the process of evaluating a product being built through a group of users testing said product. The goals of usability testing are to identify any potential usability problems and being able to analyze how the users will use the product. The general process of this is to get a group of users, typically the intended audience for the product, to perform a set of tasks set by the developers. The developers will have the users perform the task one at a time while analyzing and taking notes of how they perform these actions, as well as any usability issues that may appear. The results from each user are compared and any issues found are reported and fixed.

To conduct our usability testing, we will enlist 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, we also plan to form a group of university students majoring in Computer Science and/or Cybersecurity to analyze how potential users of the app would navigate the website. These two groups 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 will be to ensure users can browse through every page with no issues on the user end. On top of this, we will want 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.

The first task that we will have users accomplish is to find the instructions on how to use Zeek found on the About page. Next, we will have users go to the package page and click on 3 random packages and tell us the version number as well as click on the repository. After this, we will have the user search for ‘http’ and tell us the first 5 results. Based on the names we will ask how relevant the packages seem to be. The last test will be to have the user find the ‘add-json’ package and list off the build and test command for this package.

During this testing we will analyze 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 will be collecting will be 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.

5. 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 in terms of its current search engine and the look and feel of the UI. By developing a newly refreshed website we intend to alleviate the current technological challenges users face.

In this document, we've discussed how we plan to validate the features we developed and how safe implementations can be made in the future so Zeek can continue to ensure their customers have a positive and beneficial experience with their product. In regards to unit testing, we explained how we plan to test the parser, README scraper, and search modules by utilizing the Python Pytest library. Similarly, we will use the same library to perform integration testing, which focuses on the seamless interaction between each module with emphasis on data accuracy and properly displaying to the front-end of the website. Furthermore, the usability testing will help to ensure these users receive the best user experience possible and that everything is easy and seamless to find. By following this plan, our team should be able to give the Zeek team and their users a positive experience on their site, which then will encourage them to come back as the demand for enhanced safety and security rises.