Analyzing Vulnerabilities in the Chromium Browser

Fraser Price

Abstract

Web browsers are some of the most widely used software in the world. Security vulnerabilities in browsers open risks for billions of users around the world; quickly discovering these vulnerabilities and uncovering their source is not only a crucial, but also a time sensitive issue if developers are to effectively combat those who would exploit them.

In this paper, we present a visualization tool for a high-level overview of bug statistics in the Chromium browser which may help developers to understand past causes of vulnerabilities, and so may aid in building secure software in the future.

We also present our attempt to build a predictive model for commits to the Chromium repository with the ability to classify insecure code before it is pushed to the codebase. This attempt failed due to the very limited availability of data linking vulnerabilities to their occurrences in source code; for future studies to find more success, collection and maintenance of such a dataset is required.

1. Introduction

Browsers are not only among the world's most important online software systems, but are also host to multitudes of security sensitive operations which involve relaying critical, private information; these two factors combined give rise to huge amounts of potential risks for the public if browser vulnerabilities can be exploited. Tools which can help us to understand or discover vulnerabilities are therefore extremely important for browser developers in the hunt for software that users can rely on as being safe to use.

With the above in mind, the aim for this research has been focused on two goals:

1. To collect, analyze, and effectively display vulnerability data for the Chromium browser.
2. To build a classifier which can identify insecure code in the Chromium browser.

Our first goal is based on the lack of easily accessible, intuitive information regarding vulnerabilities found in Chromium. We hope that displaying vulnerability data in a clear way may help us to understand where vulnerabilities come from in Chromium, and may lead to interesting questions which may have not been asked before.

Our second goal is focused on directly aiding Chromium and other open-source developers in writing more secure code. Due to the sheer size of the codebase along with its open source nature and mass of developers, it is common for bugs to slip through code review, leading to one of the world's largest public bug databases for a single project [1]. We believe that using a predictive model in conjunction with code review may help to narrow down areas of code which are likely to be vulnerable, and may help to reduce the number of developer hours spent finding vulnerabilities.

We hope that finding success with Chromium may prompt similar work for other browsers.

2. Related Work

In this section, we look at significant pieces of literature relating to our work.

Browser Vulnerabilities

As this project revolves around security, it is worthy to mention how the insecurity of a project is measured; both Kaur and Okereke et al. have shown that insecurity status is proportional to vulnerability density and highly influential vulnerable instances [2][3].

Woo et al. suggest that the AML model for projecting the number of vulnerabilities found over time for a certain project fits well for browsers [4], largely leading to an S-shaped cumulative vulnerability curve over time. They reason that as code size and age increases, so to does the complexity in finding vulnerabilities, but also that the popularity of the project increases incentive for finding vulnerabilities. Browsers score highly in both these categories, meaning that there is massive incentive to find vulnerabilities (exemplified by bounties on vulnerability discovery, such as that used by Google for the Chromium browser [5]) but that they are also difficult to identify. Similar findings are shown by Alzhami et al. [6].

These findings lead to an interesting case where the use of automated vulnerability detection is exceptionally desirable for browsers. This is not only because there is a high incentive to find vulnerabilities, but also because it requires a huge number of man hours to identify them manually by developers; in theory, this time could be drastically reduced by automation.
Di Biase et al. explored the effectiveness of code review for the Chromium project [7]. Being extremely relevant to our work, this study provided us with several key pieces of information:

- Just 1% of code review comments addressed potential security issues
- Code reviews which miss security issues mostly miss language specific issues (e.g. C++ buffer overflow) and domain specific issues (e.g. XSS)
- Code reviews which address security specific issues are more likely to address domain specific issues than language specific ones
- Reviews in which more than 2 reviewers are involved tend to raise more security concerns

This information implies some useful insights:

- Code reviews involving 1 or 2 people don't tend to raise security issues, directly implying that more man hours are needed for code reviews involving <3 people to effectively address security issues
- A large proportion of language specific issues are missed, and a small amount are addressed, implying that language specific issues are hardest to pick up during code review

Vulnerability Prediction

Finding vulnerabilities in any software, let alone the huge codebases of browsers, is a difficult task; there have been numerous different studies which attempt to understand and predict vulnerabilities automatically. Static analysis techniques fall short for predicting vulnerabilities in codebases of this size and complexity, and so current research projects focused on creating tools to predict vulnerable code are largely based on statistical models. Classifiers built by these studies vary by feature sets, learners and granularity; we explore numerous approaches to take into consideration when deciding on a strategy of our own.

A comprehensive overview of quantative analyses of software vulnerabilities is given by Hyunchul, in which he covers the lack of available vulnerability data statistical models can be applied to. He states that although statistical modelling to find vulnerabilities is in its infancy, datasets have begun to become large enough to be used by statistical models [8].

Several approaches to vulnerability prediction focus on code metrics, none of which have been particularly successful. Morrison et al. explore the scarcity of vulnerability predictive models, and try to create one using code metrics [9]. They find that predictive models at source level perform poorly when code metrics alone are used as features. Shin et al. find that code metrics such as churn, complexity and developer activity can be indicative of vulnerabilities, but that the precision of solely metric-based models are extremely poor at below 0.05 [10]. Zimmerman et al. looked at the potential for code metrics to be able to predict vulnerabilities in Windows Vista; their model precision was fair at roughly 0.6, whereas their recall was very low at just 0.1-.02 [11].

A technique which has found more success has been to extract features for learning directly from source code. Scandariato et al. find success at file level classification by using bag of word features of source code in two separate studies [12], [13]; they obtain impressive average precision and recall values, both over 0.8, over 20 Java applications. Another similar approach by Perl et al. is to classify individual git commits, using both commit code and commit metadata (e.g. author commits to project, number of commit hunks, file future changes, etc.) [14]. They analyzed 66 C and C++ projects, classifying commits as vulnerable based on official CVE reports. Their model obtained precision and recall values of 0.6 and 0.24 respectively. It is worth noting that they reported that their dataset should work out of the box for training classifiers for other C and C++ projects.

3. Data Visualization

As discussed in the introduction of this paper, the first goal of this study is to effectively and clearly present vulnerability data for Chromium. As of September 2017, the Chromium codebase contains over 16 million lines of code, 200,000 files, and is linked to a bug database containing over 56,000 reports. Being open source, this information is all publicly available; source can be found at [15], bug report database at [1], officially reported CVEs at [16], and stable release information, which maps CVEs to bug reports and details vulnerability fixes, at [17].

Despite this wealth of information being publicly available, our immediate observation was that there is no direct way to map bugs or vulnerabilities to the specific sections of code which they affect, meaning there is no high-level overview of vulnerability statistics for Chromium. We believe a high-level, intuitive, dynamic overview of vulnerabilities and how they relate to the Chromium source code directly could raise questions which have not yet been considered about high-level vulnerability patterns. The answers to these questions may provide new insights as to how and why vulnerabilities are introduced, and how to prevent common patterns in the future.

We considered what the most effective way of communicating this data from a high-level perspective would be, and settled upon building an online visualization tool alongside a JSON API for those who wish to access our raw data. This visualization must be able to represent the entire repository tree at the topmost level, and so scalability was a large factor in deciding which visualization format to use. We took inspiration from Figure 2 in paper [18], and decided that a colour-coded treemap suited our needs well.

Requirements

We considered the requirements necessary for this tool to be effective, and decided the following were necessary.

Scalability

Chromium is one of the largest open source projects in the world, and so to efficiently and quickly display the top-level view in a web browser the system must serve data dynamically based on the user's current view.

Full Source Tree Traversal

The visualization must be able to display statistics at any node in the source tree to be fully explorable and complete. This implies the ability to be able to traverse through the tree to any folder or file in the Chromium source, thus giving the user the ability to "zoom in" to individual sections of the code for a clear view.

Intuitive Presentation and Ability to Find Defects Visually

To be able to convey information effectively, users must be able to look at the visualization and immediately spot areas of interest for the given view.

Multiple Statistics

To be able to represent all our collected data effectively, users should be able to view different statistics, i.e. select from a number different views for the same codebase.

A Functionality Walk-Through

Below is a brief outline of the basic functionality of the visualization app. The app's landing page is shown below in Figure 1, along with the dropdown menu for selecting the various visualizations in Figure 2. The definitions of the various visualization options in Figure 2 are detailed below in Table 1.

KEY	DEFINITION
P:	Pdfium codebase
C:	Chromium codebase
All	View all bugs in codebase
Security	View security bugs in codebase
Security : All	View ratio of security bugs to all bugs in codebase

Table 1: Visualization Options

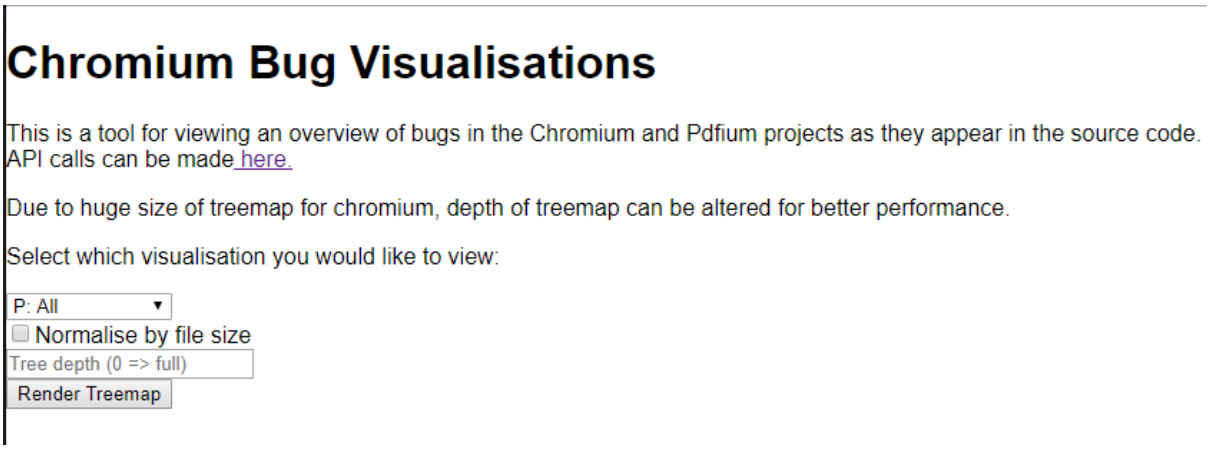
The other customization options which can be seen in Figure 1 are:

Normalize by file size

Bug numbers for selected visualization will be normalized by file size for every file if this value is selected. Value shown by colour scheme for each file will be proportional to $\frac{number\ of\ bugs}{file\ size}$ rather than just $number\ of\ bugs$.

Tree depth

Depth of tree to be loaded at each view, with 0 referring to full directory tree within the source code database. The higher the depth, the more accurate but less responsive the view will be. It is recommended that a limit of 3-5 is set for the Chromium codebase.



Chromium Bug Visualisations

This is a tool for viewing an overview of bugs in the Chromium and Pdfium projects as they appear in the source code. API calls can be made here.

Due to huge size of treemap for chromium, depth of treemap can be altered for better performance.

Select which visualisation you would like to view:

Figure 1: Visualization App Landing Page

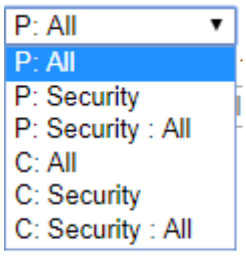


Figure 2: Visualization App View Menu

Once the desired options have been selected, the view can be rendered via the **Render Treemap** button. After querying the API, the treemap will be rendered to specifications; for example, **Error! Reference source not found.** shows the visualization for P: Security with unspecified depth, not normalized by file size. Folders and files in the source tree can be navigated to by clicking on the desired folder or folder header; to return up the tree, users may simply click the "Go Up" button. Data about each folder is displayed via a popup by hovering over the desired file or folder.

To render a different treemap, users may simply select new options and click the Render Treemap button once more.

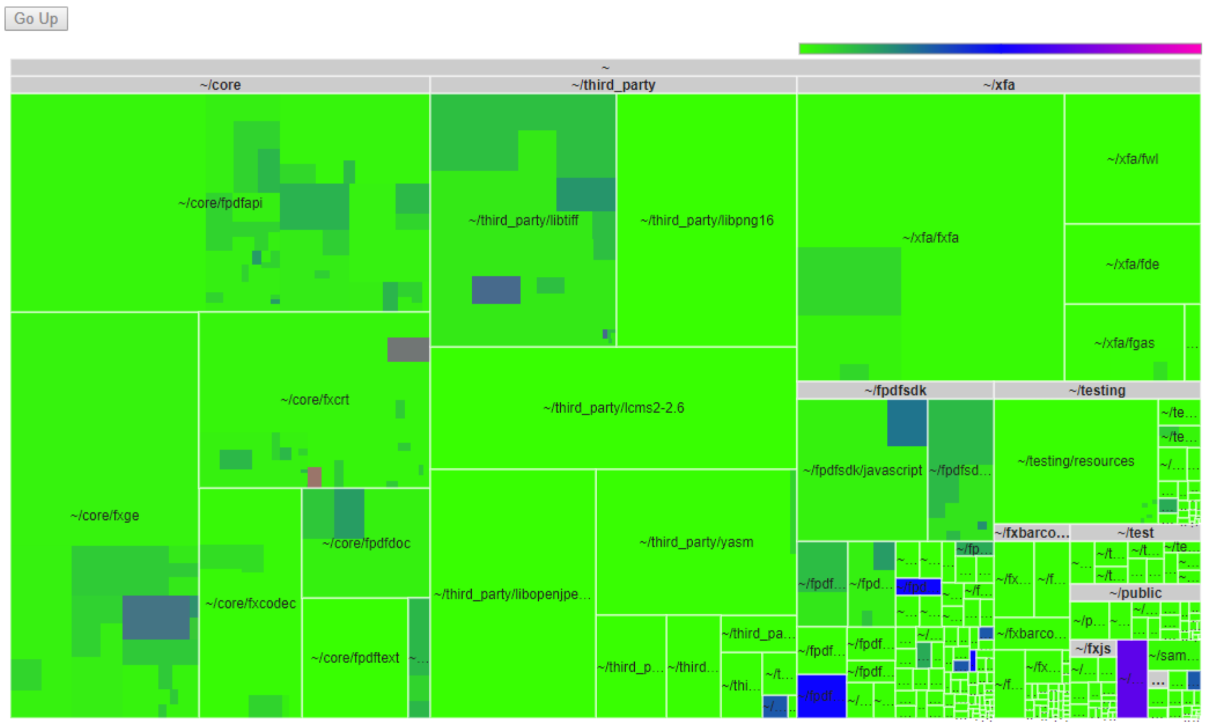


Figure 3: Visualization of P: Security at source root

Case Studies

We believe this visualization tool has multiple potential uses; this section will attempt to provide information as to how this tool may be used, and will detail some hypothetical practical use cases.

High-Level Overview

Error! Reference source not found., which we have already seen above, gives us useful high-level information at a quick glance. A user can immediately see from this top-level view which parts of the Pdfium codebase tend to give rise to security bugs, and can easily traverse to the offending parts to investigate further.
In the case of our **Error! Reference source not found.** example, the user in question may question why security issues have only occurred in half of the subdirectories of the `~/core/fdpfapi` directory, located at the top left corner of **Error! Reference source not found.**. Upon navigating to `~/core/fdpfapi`, they would be able to view the offending directory and files, shown in Figure 4 below, and see that they occur in all subdirectories but `cmaps`.

This sort of high-level overview could be useful for several reasons, most notably:

Targeting of an Analysis Tool or Manual Code Review

Developers may believe this data to be applicable to current vulnerabilities or bugs, and may choose to focus an automated or manual review on sections of code which were prone to bugs or vulnerabilities in the past (or alternatively on those which weren't and may have slipped past review)

Intuitive Discovery of High-Level Patterns

Quickly discovering high level patterns may help to combat insecure coding. In our example detailed above, questions could be raised as to why no security issues appear in `cmaps`, but appear in other subfolders. These questions may not have otherwise been asked due to the lack of available data which makes these patterns obvious

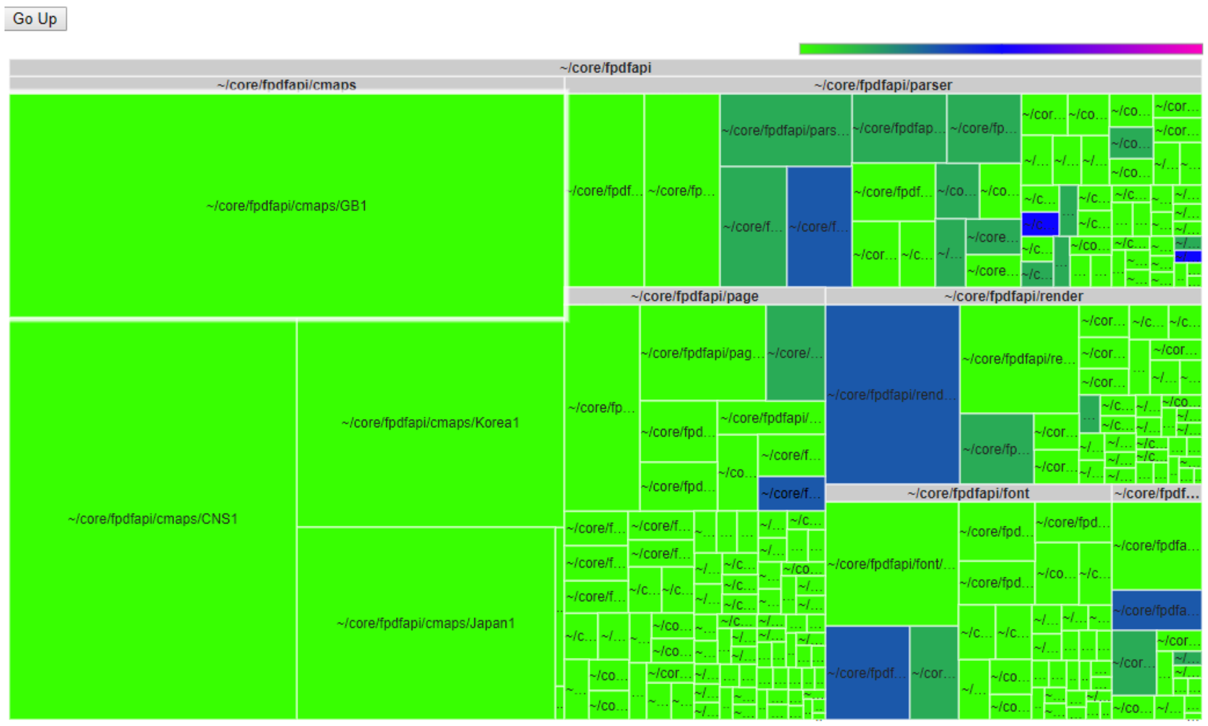


Figure 4: Visualization of P: Security at ~/core/fpdfapi

Practical Functionality Examples

Each available view provides different information about the same codebase, meaning these different views may be useful for completely different purposes. As an example of the versatility of this tool, we will consider a hypothetical situation involving two separate development teams who wish to solve two different problems:

Team 1: Identify Areas of Pdfium to Target for Static Analysis

After consideration of their problem, Team 1 decides that their solution should be one of the following:

- Target areas of code which have been problematic in the past and so may be likely to cause issues in future
- Target areas which have not undergone analysis recently and so may need to be checked
- Combine both above approaches to find areas to target

They decide the latter strategy is best. To identify areas of code which have been buggy in the past, they can either use the visualization tool to quickly identify problematic areas by viewing P: All, or alternatively retrieve raw data for each file directly from our API. Once they have this data, they can combine it with their own analysis history to come to a solution.

Team 2: Identify Areas of Chromium which should be Reviewed by a Security Team

Team 2 decides that areas which the security team would be most valuable reviewing are those where historically security bugs have been most dense. They simply view C: Security, normalized by file size, and direct the team to areas of code which stand out on the visualization.

These are two simple examples which demonstrate the possible versatility of the app, and that each view may have various functionality

Implementation Details

Our goal was to build an interactive treemap visualization of the codebase which could be easily traversed and could also accurately represent both high level (e.g. root directory) and low-level views (e.g. deep subfolder or file level). Our aim was to not only provide a good amount of data at every level of the source, but also an intuitive colour coded view which could allow users to identify at a glance which areas of the codebase have historically been most problematic.

Our requirements lead to a surprisingly difficult task simply due to the size of the Chromium codebase. Creating a static tree in a browser with this many leaves (i.e. files) and many more nodes (i.e. directories) would be too much for a browser to handle, so data retrieval and rendering had to be dynamic depending on the user's current view. No libraries for dynamically rendering treemaps exist publicly (to the best of our knowledge), so we created a wrapper for the Google Charts Treemap library [19] which allowed for dynamic treemap traversal.

Our frontend [20] was built with ReactJS, API [21] with Express and database with MongoDB [22]. All data was scraped from Google's code review site [23] and bug report database [1], and is stored statically and queried by our API as required.

Below is a summary of how we met each project requirement:

- **Scalability:** Use of dynamic rendering of treemaps using an API and static database allowed us to display large source trees without worrying about performance
- **Full Tree Traversal:** Dynamic rendering also allows us to make API calls to retrieve data for any part of the source tree for any depth, and so allows for full tree traversal by the user
- **Intuitive Presentation:** We used a colour coding scheme to allow users to visually identify interesting areas of code with ease
- **Multiple Statistics:** Incorporating multiple views allowed users to easily study various sets of data using the same tool

4. Vulnerability Prediction

The second part of our research concerned creating a vulnerability prediction model for Chromium. There have been several different approaches to vulnerability prediction, discussed in related work [11], [6], [9], [10], [12], [13], [18], but none (to our knowledge) for the Chromium browser specifically. Our goal was to create a model which would not only allow for code granularity smaller than that of files to allow for more useful information for developers, but also one which gave an explanation as to why a certain piece of code was marked as vulnerable. Combining these two goals, we decided to follow the approach discussed by Perl et al. in the VCCFinder paper [14], which uses linear SVM learning to classify individual commits as vulnerable or not. The ability to classify commits rather than files not only allows for a finer granularity, making results more useful for developers, but also allows us to use commit metadata which was shown to aid in classifying commits as vulnerable or not.

Feature Selection

Before starting with data collection, we collected a list of features to use for learning. The VCCFinder project found that commit metadata, such as length of commit or number of other commits by the commit author, was useful for differentiating between vulnerable and non-vulnerable commits along with code content.

Feature Set

Feature Name	Bucket Size	Description
Tokenized Code	N/A	Tokenized C++ source code
Hunk Count	5	Separate sections of code changed in commit
Additions	10	Number of lines added
Deletions	10	Number of lines deleted

Table 2: Feature Set

Feature Representation

We decided to follow the VCCFinder approach of using a bag of words model [24] to represent both numeric metadata features and tokenized code content.

For a bag of words model to work, commits must be represented in the context of a feature set across our entire dataset so that our classifier can identify the same features across different commits. This feature set takes the form of a large hash map of all tokens in our dataset, each mapping to a unique index. Once the set is formed, individual tokenized commits may be represented within the context of the feature set by an array of counts of each individual feature, where the index of each count represents the index this feature maps to in our feature set. As this array will consist mainly of zeroes, we can efficiently represent it as a sparse feature vector. A sparse feature vector takes the form of a hash map, where keys are feature indexes and values represent the number of times that feature appears; we vastly reduce the size of the array by omitting entries which do not appear in the commit.

We also followed the VCCFinder technique of inserting numeric metadata features into discrete sized bins, allowing our classifier to identify values that are close in value rather than viewing, for example, “4.5” and “4.6” as entirely different strings. Bin sizes were chosen for each feature, and are detailed in Table 1 above.

We tokenize our commit code by all C++ punctuation and whitespace rather than following the VCCFinder approach of simply tokenizing by whitespace (i.e. spaces, tabs and newlines). We believe this tokenizing approach to be better, as our classifier can identify function calls to the same function with different arguments as calls to the same function.

As an example, consider Table 3 below which details two calls to the same function and both versions of their corresponding tokenized versions. In this example, a classifier would be able to identify that the same function is called by both pieces of code if using our tokens, whereas it would see `‘max(2,’` and `‘max(4,’` as two separate tokens in the case of VCCFinder. It is worthy to note that Scandariato et al. found success in studies [12], [13] (precision and recall levels over 0.8) by tokenizing by all punctuation rather than simply by whitespace.

Tokenized commits are represented by a map of `feature_name : feature_count`

Function Call	VCCFinder Tokens	Our Tokens
`int m = max(2, 3);`	`{'int':1, 'm':1, '=':1, 'max(2,':1, '3);':1}`	`{'int':1, 'm':1, 'max':1, '2':1, '3':1}`
`int n = max(4, 5);`	`{'int':1, 'n':1, '=':1, 'max(4,':1, '5);':1}`	`{'int':1, 'n':1, 'max':1, '4':1, '5':1}`

Table 3: Function Calls and their Corresponding Tokens

We will now detail a brief example of commit representation. Let us assume our feature set is as below

```
feature_set = {
      '2'        : 0,
      '3'        : 1,
      '4'        : 2,
      '5'        : 3,
      'int'      : 4,
      'm'        : 5,
      'max'      : 6,
      'n'        : 7,
      'foo'      : 8,
      'bar'      : 9,
      '$ACP_0_1$': 10,
      '$ACP_1_2$': 11,
      '$HC_1_3$' : 12,
      '$foo_bin$': 13,
      '$bar_bin$': 14
}
```

where `$ACP_0_1` corresponds to the author commit percentage bin between 0% and 1%, `$ACP_1_2` corresponds to the bin between 1% and 2%, and `'$HC_1_3'` represents the hunk count bin for counts between 1 and 3. All numeric metadata features are represented in the same form, starting and ending with the `$` character.

For our example, let us consider two commits, shown below in Table 4. For the sake of simplicity, we will only consider commit percentage and hunk count as our metadata for these commits.

As we can see, the representation of commits in feature set context (col 4 of Table 4) will always be equal to the size of the feature set. As of September 2017, Chromium contains over 600,000 commits; the feature set for Chromium commits is therefore massive, at over 16 million tokens. Each individual commit will only contain a handful of these tokens, which is why a sparse feature vector is an extremely compact and therefore efficient representation in comparison with our naïve array.

Commit Additions	Commit Metadata	Tokenized Representation	Representation in Feature Set Context	Sparse Feature Vector

`File A:`
`    int m = max(2, 3);`
`File B:`
`    int n = max(4, 5);`
	`ACP: 0.3%`
	`HC: 2`
		`{'2':1, '3':1, '4':1, '5':1, 'int':2, 'm':1, 'max':2, 'n':1, '$ACP_0_1':1, '$HC_1_3':1}`	`[1, 1, 1, 1, 2, 1, 2, 1, 0, 0, 1, 0, 1, 0, 0]`	`{0:1, 1:1, 2:1, 3:1, 4:2, 5:1, 6:2, 7:1, 10:1, 12:1}`

`File A:`
`    int n = max(4, 5);`
	`ACP: 1.8%`
	`HC: 1`
		`{'4':1, '5':1, 'int':1, 'max':1, 'n':1, '$ACP_1_2':1, '$HC_1_3':1}`	`[0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0]`	`{2:1, 3:1, 4:1, 6:1, 7:1, 11:1, 12:1}`

Table 4: Two Commits and their Tokenized Representation

Datasets

To build our classifier we started with the goal of acquiring a dataset of labelled vulnerable and non-vulnerable Chromium commits to use as training and test data. As this dataset does not (to the best of our knowledge) currently exist, our only option was to build it ourselves from publicly available information online. However, due to the absence of a mapping from vulnerabilities to individual commits, we were worried that building an accurate dataset might be difficult or nearly impossible, and that our classifier would therefore produce poor results. As discussed in the section below, Issues with Chromium Dataset Collection, it turned out to be too difficult to create such a dataset with the time and resources available to us.

However as noted in the related work section, Perl et al. state in their VCCFinder paper [14] that their commit dataset, which is known to be accurate, should be capable of training classifiers for commits from C and C++ projects other than ones used to build the dataset. With this information in mind, we resolved to train our classifier using their dataset.

Issues with Chromium Dataset Collection

For our set to provide reliable training information we required a large volume of Chromium commits, preferably all commits in the repository, each accurately labelled as either vulnerable or not vulnerable. Due to the lack of publicly available information mapping vulnerabilities to commits, our options for building this dataset were very limited.
The only mapping we could find from official CVE reports to useful information regarding them is the mapping found on the Chromium stable update blog [17], which maps officially reported vulnerabilities to their corresponding bug reports [1]. These reports often include a link, provided automatically by a commit bot, to the commit which fixes them; with this in mind, we attempted to build a tool which could automatically find vulnerable commits using the following method:

1. Collect a set of all vulnerability bug reports, found using the mapping on the stable update blog
2. Scrape these vulnerability reports to find commits which fix them
3. Use `git blame` on each fixing commit to find the commit(s) which introduced the vulnerabilities

The flaw in our method arose at step 2; bug reports do not reliably link the commit which fixes the issue, and often contain links to several other commits which are related to the bug being discussed but do not fix it. Of the 1200 vulnerability reports, around 300 could provide links to one or more fixing commits, and of these fewer than 100 could provide a definitive link to a single fixing commit; this is simply not enough data for building a reliable classifier.

We considered the possibility of collecting a list of fixing commits manually with the help of our script, but due to our lack of time and non-existent experience with the Chromium codebase we were forced to resign this as an infeasible option. Our inability to build this dataset not only had the consequence that we were unable train our classifier with data collected directly from Chromium, but also meant that we would be unable to test our classifier if we were to train it using the VCCFinder dataset, thus stopping us from gathering any concrete data on the predictions our classifier would make.

Model Building

Despite our inability to fully test a classifier build solely with VCCFinder data, we decided to build and test a VCCFinder-only model to see whether we could reproduce their results with the intention of manually testing some Chromium commit samples. We did this to see whether we could gain information about whether data collected from other projects may have the potential to be able to predict vulnerabilities in Chromium, a project independent of the training set.

We followed the VCCFinder approach of using a linear SVM model with the help of the Liblinear library[25] using the feature set discussed above. The dataset contained a total of 351,409 commits, of which 734 were vulnerable, over 66 different open source C++ projects on GitHub. More information regarding the collection and accuracy of this data can be found in the VCCFinder paper[14]. We tested our model with VCCFinder data to get an idea of whether our features were indicative of vulnerabilities; our precision and recall values were quite poor at around 0.2. After removing a large percentage of non-vulnerable commits from our training set precision and recall rose to 0.75; this indicated that although our features were indeed indicative of vulnerabilities, the correlation between them and commit vulnerability was low. We decided that unfortunately it would not be worth testing our model across Chromium commits, given its poor performance for commits in the same dataset.

5. Conclusions

Summary of Results

Despite the lack of results in the form of concrete data, this study has provided several insights into vulnerabilities in Chromium and the way in which data is reported and presented. Our key findings are detailed below.

Potential Insights from High Level Data

Our visualization tool may provide Chromium developers with the ability to find high level bug and vulnerability patterns in their source which may not have otherwise been discovered. If found to be useful, tools like this which present high-level vulnerability data could open new avenues for the discovery and prevention of vulnerabilities.

Lack of Available Data

Our attempt to build a vulnerability prediction model fell short primarily due to the lack of reliable reported data for Chromium. To successfully build an accurate prediction model which can be tested and verified, we believe data accurately linking vulnerabilities to their occurrences in source code is required.

This data could be collected for past vulnerabilities with enough man hours from developers who are familiar with the Chromium source, and could be easily maintained with more reliable reporting of direct occurrences of vulnerabilities as they appear in source. For reliable vulnerability prediction tools to become the norm, open source projects such as Chromium should seek to enforce more strict reporting of vulnerability information.

Potential for Future Improvements

This study has opened several options for potential improvement and exploration in future studies.

Data Visualization

There is a huge amount of potential for our visualization tool and others like it to provide much more intuitive, high level vulnerability information. Some possible features may include:

- Ability for users to add and analyze their own data from their own projects
- Direct linking of source files and commits from visualization
- Query system for customization of results
- Timeline view, where users can traverse git branches to view project throughout its development

Predictive Model

As stated above, improvements to vulnerability prediction models in future will revolve around the availability of larger amounts of reliable data; along with this, improvements may also stem from testing different learning models and feature sets to find those which work best.

6. Bibliography

[1] “Chromium Bug Database.” [Online]. Available: https://bugs.chromium.org/p/chromium.

[2] P. D. Kaur, “Insecurity Status and Vulnerability Density of Web Applications : A Quantitative Approach,” vol. 15, no. 1, pp. 428–432, 2017.

[3] G. E. Okereke, “Security Metrics Model for Web Application Vulnerability Analysis,” vol. 5, no. 5, pp. 146–150, 2012.

[4] S. Woo, O. Alhazmi, and Y. Malaiya, “An analysis of the vulnerability discovery process in web browsers,” *Proc. 10th IASTED …*, 2006.

[5] “Chrome Rewards.” [Online]. Available: https://www.google.com/about/appsecurity/chrome-rewards/index.html.

[6] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, “Measuring, analyzing and predicting security vulnerabilities in software systems,” *Comput. Secur.*, vol. 26, no. 3, pp. 219–228, 2007.

[7] M. Di Biase, M. Bruntink, and A. Bacchelli, “A security perspective on code review: The case of chromium,” *Proc. - 2016 IEEE 16th Int. Work. Conf. Source Code Anal. Manip. SCAM 2016*, pp. 21–30, 2016.

[8] H. Joh, “Quantitative Analysis of Software Vulnerabilities,” *Dissertation*, 2011.

[9] P. Morrison, K. Herzig, B. Murphy, and L. Williams, “Challenges with Applying Vulnerability Prediction Models,” *Proc. 2015 Symp. Bootcamp Sci. Secur.*, p. 4:1-4:9, 2015.

[10] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, 2011.

[11] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista,” *ICST 2010 - 3rd Int. Conf. Softw. Testing, Verif. Valid.*, pp. 421–428, 2010.

[12] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, 2014.

[13] A. Hovsepyan *et al.*, “Predicting vulnerable software components via text mining,” *Comput. Secur.*, vol. 5, no. 5, p. 4:1-4:9, 2012.

[14] H. Perl, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, and S. Fahl, “VCCFinder : Finding Potential Vulnerabilities in Open-Source Projects to Assist Code,” in *Ccs ’15*, 2015, pp. 426–437.

[15] “Chromium Source Code.” [Online]. Available: https://chromium.googlesource.com/chromium/src.

[16] cvedetails, “Chromium CVEs.” [Online]. Available: https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-15031/Google-Chrome.html.

[17] Google, “Chromium Stable Update Blog.” .

[18] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” *Proc. 14th ACM Conf. Comput. Commun. Secur. CCS 07*, p. 529, 2007.

[19] Google, “Google Charts Treemap API.” [Online]. Available: https://developers.google.com/chart/interactive/docs/gallery/treemap.

[20] F. Price and B. Livshits, “Chromium Visualization Client.” 2017.

[21] F. Price and B. Livshits, “Chromium Visualisation API.” 2017.

[22] “MongoDB.” [Online]. Available: https://www.mongodb.com/.

[23] Google, “Chromium Code Review.”

[24] Wikipedia, “Bag of Words Model.” [Online]. Available: https://en.wikipedia.org/wiki/Bag-of-words_model.

[25] N. T. University, “Liblinear.” [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/liblinear/.