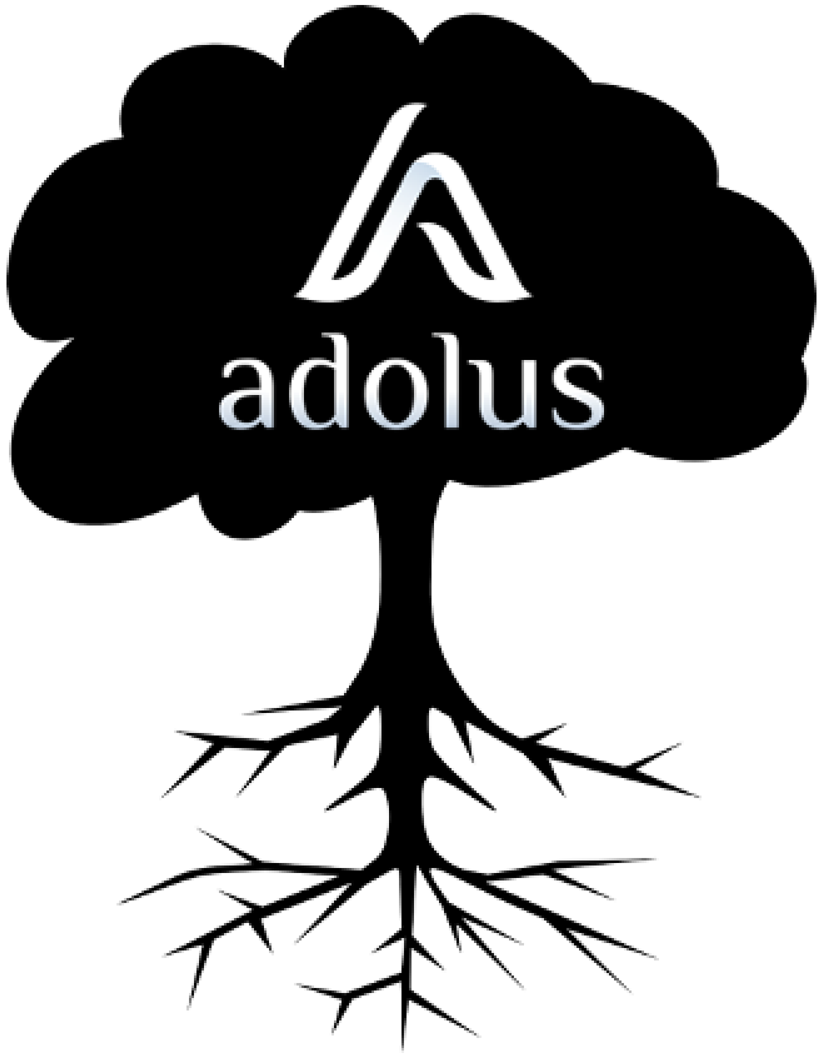


Unearthing Hidden Vulnerabilities

University of British Columbia
Master of Data Science Capstone Project

Partner: aDolus Inc.
Mentor: Tiffany Timbers

James Liu, Jarome Leslie, Derek Kruszewski, Subing Cao

June 23, 2020

Contents

Executive Summary

Matching known vulnerabilities to affected vendor files is an important task which helps aDolus' customers quantify the cybersecurity risk in specific vendor software and firmware. Through the creation of the `periculum` pipeline, aDolus is provided with an automated tool to generate new vulnerability-vendor file matches augmententing the utility of their Framework for Analysis and Coordinated Trust ("FACT") platform. For all vendors cataloged in aDolus' database of product lines, the pipeline may be applied to any collection of corresponding vendor advisory reports saved in an S3 bucket. As demonstrated with Rockwell Automation, the pipeline can find the paths in an advisory up to 73% of the time and a companion report will be produced to address shortcomings by allowing a human to verify the quality of matches, ensuring a fully transparent process. Figure 1 summarizes implementation of the `periculum` pipeline which replaces a slow and manual workflow.

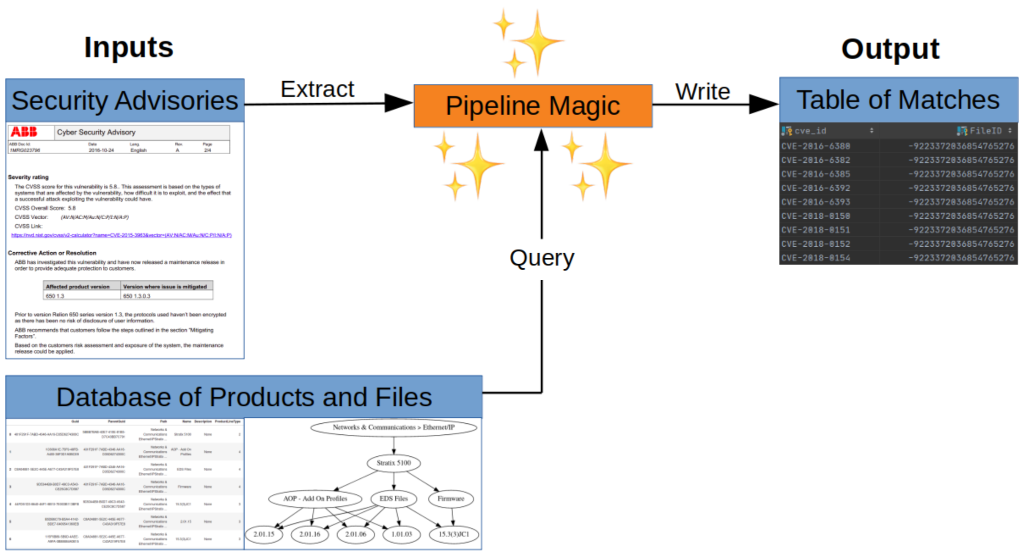


ABB Cyber Security Advisory

ABB Doc Id	Date	Lang.	Rev.	Page
1MRG023796	2016-10-24	English	A	2/4

Severity rating

The CVSS score for this vulnerability is 5.8.. This assessment is based on the types of systems that are affected by the vulnerability, how difficult it is to exploit, and the effect that a successful attack exploiting the vulnerability could have.

CVSS Overall Score: 5.8

CVSS Vector: (AV:N/AC:M/Au:N/C:P/I:N/A:P)

CVSS Link:

https://nvd.nist.gov/cvss/v2-calculator?name=CVE-2015-3963&vector=(AV:N/AC:M/Au:N/C:P/I:N/A:P)

Corrective Action or Resolution

ABB has investigated this vulnerability and have now released a maintenance release in order to provide adequate protection to customers.

Affected product version	Version where issue is mitigated
650 1.3	650 1.3.0.3

Prior to version Relion 650 series version 1.3, the protocols used haven't been encrypted as there has been no risk of disclosure of user information.

ABB recommends that customers follow the steps outlined in the section "Mitigating Factors".

Based on the customers risk assessment and exposure of the system, the maintenance release could be applied.

	Guid	ParentGuid	Path	Name	Description	ProductLineType
0	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	Stratix 5100	None	2
1	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	AOP - Add On Profiles	None	4
2	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	EDS Files	None	4
3	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	Firmware	None	4
4	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	15.3(3)JC1	None	3
5	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	2.01.15	None	3
6	[illegible]	[illegible]	Networks & Communications Ethernet/IP/Stratix ...	15.3(3)JC1	None	3

cve_id	FileID
CVE-2016-6380	-9223372036854765276
CVE-2016-6382	-9223372036854765276
CVE-2016-6385	-9223372036854765276
CVE-2016-6392	-9223372036854765276
CVE-2016-6393	-9223372036854765276
CVE-2018-0150	-9223372036854765276
CVE-2018-0151	-9223372036854765276
CVE-2018-0152	-9223372036854765276
CVE-2018-0154	-9223372036854765276

Figure 1: Periculum pipeline magic

Introduction

The Problem

The use of digital automation control in industrial systems brings with it the risk of firmware and software vulnerabilities in these products. If an entity with malicious intent is able to compromise such a system via its components, such disruption could lead to severe consequences. A poignant example is the 2014 cyberattack of a steel mill in Germany, which led to the explosion of a blast furnace due to shutdown failure[1]. In an attempt to combat potential breaches, firmware and software are regularly updated from vendors and/or third party distributors to patch vulnerabilities; however, this patching may introduce further vulnerability vectors in the form of "man-in-the-middle" attacks[2], increasing the risk of installing malicious software and firmware.

aDolus' Framework for Analysis and Coordinated Trust ("FACT") platform provides a solution to evaluate the security of firmware/software packages and their sub-components before installation occurs in critical equipment[3]. In the evaluation process, FACT uses an internal database of matches between known vulnerabilities and affected files. This database of matches was created and is updated manually by reviewing vendor advisory reports and tagging affected files in aDolus' growing catalog of vendor products. The crux of this problem there is to automate the process of linking the vulnerabilities in the reports from various sources to affected components in software and firmware packages. The automated solution must be robust to handle a diverse set of vulnerability report formats and structures while being transparent and auditable.

Objectives

Our capstone partner's needs are to extract information about vulnerabilities in vendor advisory reports and match it to the ultimate affected components in the software/firmware through an automated pipeline. To be able to locate the affected components, the paths (product names, versions) to those files need to be identified from the advisory reports. Thus, the main objective of this project may be refined as the matching of path information mentioned in the advisory reports to paths in the aDolus database and finding the affected files within the target paths. Essentially, this project is a Natural Language Processing ("NLP") problem.

Data Science Methods

This project presents several unique challenges discussed in the Appendix leading to it being quite different from the traditional supervised learning problem. That said, the `periculum` pipeline discussed below touches on various methods learned throughout the program.

Periculum Pipeline

We developed a software package called `periculum` to implement this pipeline[1]. This package has five main modules. First, `parse_advisories.py` is used to parse all the reports in the S3 buckets into plain texts. Then, `reports_vectorizer.py` vectorizes product line paths into tokens and generates a sparse matrix of path token counts for each advisory report. `path_scorer.py` selects matched paths for each report based on scoring metrics. Finally, `match_summarize.py`, `graphics_gen.py` and a jupyter notebook summarize the results in a `.csv` file, generate visualizations and make a companion report. Throughout this process, intermediary files are passed between scripts using `feather`[4].

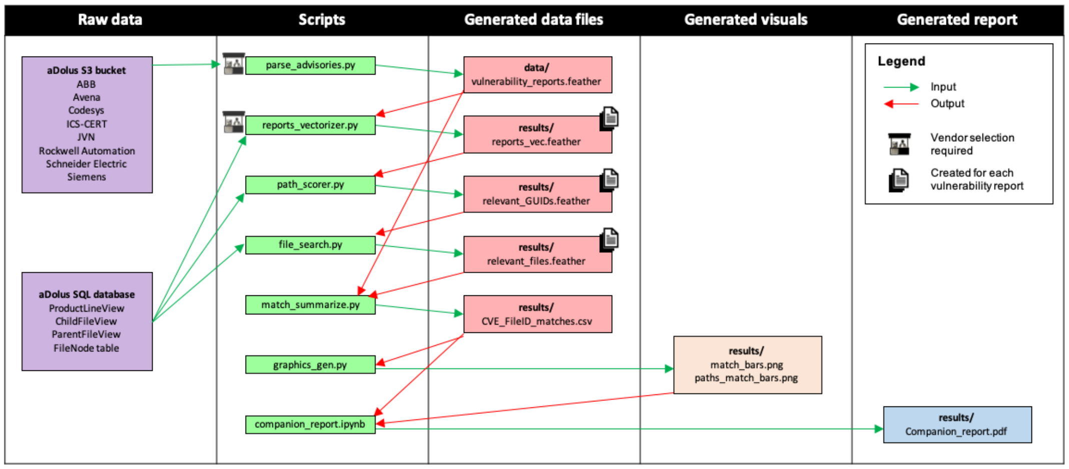


Figure 2: Overview of periculum data pipeline

Text Extraction

Before any work can be done with the text of vendor advisory reports, `parse_advisories.py` reads and extracts the plain text from the raw data files found in an Amazon Web Services S3 bucket using `boto3`[5] and stores it in single aggregate table. Additionally `BeautifulSoup`[6] and `pdfminer`[7] to handle `.html` and `.pdf` respectively, expanding the fuctionality beyond `.txt` files.

[1]`periculum` is the latin word for danger and is the help signalling spell in Harry Potter. Our package embodies the theme of this spell by signalling to aDolus when and where vulnerabilities are found in its catalog of vendor products.

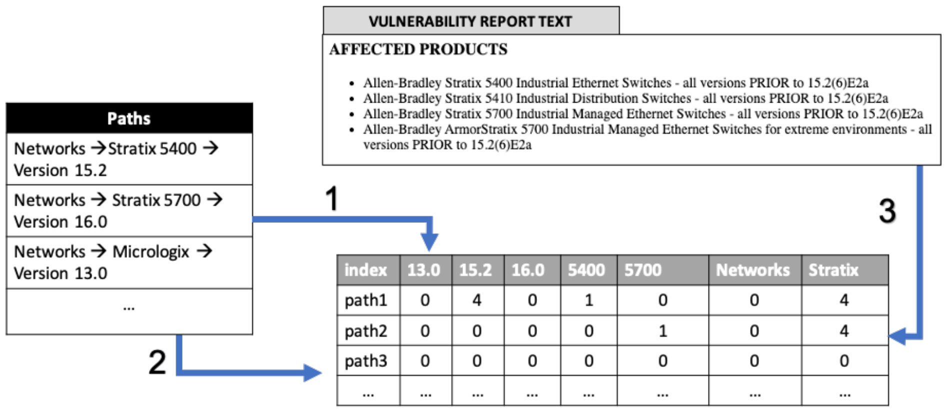
Vectorizing Reports

The crux of our project relies on the concept of tokenizing documents, which is a widely-used technique in Natural Language Processing. The key script for this is `reports_vectorizer.py`, which uses `sklearn.CountVectorizer`[8], a scikit-learn method, to turn texts into a sparse-matrix of token counts.

The overview for the algorithm which we implemented is as follows.

- Call `fit_transform` on the list of paths we obtain from the database using `pyodbc`[9], which generates a `<number of paths>` $\times$ `<number of tokens>` matrix representing the count of each token per path, and uses those same tokens as the vocabulary for the next step.
- Call `transform` on each report, which generates a `<number of tokens>` $\times$ `1` matrix, representing the counts of each vocabulary token in the report.
- For every path that was used in the tokenizer, we compare the counts between the path token matrix row and the report token matrix, and keep only the non-zero values for tokens that are in both the path token row and report matrix. This represents the count of tokens present in each path in the report.

The algorithm produces a `<number of paths>` $\times$ `<number of tokens>` matrix for each report. This matrix is then written to file, and we will use this matrix to generate scores in order to determine the paths that are associated with a report.


VULNERABILITY REPORT TEXT

AFFECTED PRODUCTS

- Allen-Bradley Stratix 5400 Industrial Ethernet Switches - all versions PRIOR to 15.2(6)E2a
- Allen-Bradley Stratix 5410 Industrial Distribution Switches - all versions PRIOR to 15.2(6)E2a
- Allen-Bradley Stratix 5700 Industrial Managed Ethernet Switches - all versions PRIOR to 15.2(6)E2a
- Allen-Bradley ArmorStratix 5700 Industrial Managed Ethernet Switches for extreme environments - all versions PRIOR to 15.2(6)E2a

Paths
Networks → Stratix 5400 → Version 15.2
Networks → Stratix 5700 → Version 16.0
Networks → Micrologix → Version 13.0
...

index	13.0	15.2	16.0	5400	5700	Networks	Stratix
path1	0	4	0	1	0	0	4
path2	0	0	0	0	1	0	4
path3	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...

Figure 3: Overview for Reports Vectorizer

As of the current iteration, we have achieved a final run-time of 2 seconds per report.

Path Scoring

After the scoring matrix is generated, we can now entertain a few methods of scoring to determine the quality of paths associated with a given advisory report.

Using the figure above as an example,

index	13.0	15.2	16.0	5400	5700	Networks	Stratix
path1	0	4	0	1	0	0	4
path2	0	0	0	0	1	0	4
path3	0	0	0	0	0	0	0

we obtain a sparse matrix,

$$S_{paths,tokens} = \begin{bmatrix} 0 & 4 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & & & & & & \end{bmatrix}$$

The most straight-forward approach is by using *simple sum*, where:

$$score = \sum_{i \in tokens} S_i$$

e.g.:

$$S_{simple\_sum\_score} = \begin{bmatrix} 0+4+0+1+0+0+4 \\ 0+0+0+0+1+0+4 \\ 0+0+0+0+0+0+0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 9 \\ 5 \\ 0 \\ \vdots \end{bmatrix}$$

Alternatively, we can use a *simple count* by judging whether or not a token is present in the advisory report, rather than using the number of times it has occurred:

$$score = \sum_{i \in tokens} 1_N(S_i)$$

, where 1_N is the indicator function.

e.g.:

$$S_{simple\_count\_score} = 1_N(S_{paths,tokens}) = \begin{bmatrix} 0+1+0+1+0+0+1 \\ 0+0+0+0+1+0+1 \\ 0+0+0+0+0+0+0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 0 \\ \vdots \end{bmatrix}$$

A more sophisticated method would be using a *weighted score*, where we would apply a weight corresponding to the importance of each token to the score generated by either of the two previous methods:

$$score = \sum_{i \in tokens} w \cdot S_i$$

, where *score* is a score generated by either of the two previous methods, and w is a set array of weights depending on the importance of the token.

e.g.: for $w = [1, 3, 5, ...]$,

$$S_{weighted\_score} = \begin{bmatrix} (5 \cdot 0) + (1 \cdot 4) + (3 \cdot 0) + (5 \cdot 1) + (1 \cdot 0) + (5 \cdot 0) + (1 \cdot 4) \\ (5 \cdot 0) + (1 \cdot 0) + (3 \cdot 0) + (5 \cdot 0) + (1 \cdot 1) + (5 \cdot 0) + (1 \cdot 4) \\ (5 \cdot 0) + (1 \cdot 0) + (3 \cdot 0) + (5 \cdot 0) + (1 \cdot 0) + (5 \cdot 0) + (1 \cdot 0) \\ \vdots \end{bmatrix} = \begin{bmatrix} 13 \\ 5 \\ 0 \\ \vdots \end{bmatrix}$$

Model Selection

The data pipeline provided is designed for flexibility in how paths are chosen (or scored) for an advisory. Within the `path_scorer.py` script is a hyper-parameter in the `score_paths()` function that allows the package user to select new methods of scoring without having to modify the pipeline.

In choosing a default method to implement, and to provide the client with some understanding of the performance of potential scorers, four scoring methods were investigated: `simple sum`, `simple count`, `weighted_sum`, and `weighted_count`. These methods were tested on a manually labelled data set curated for the Rockwell product line in aDolus' database. Key hyperparameters in the scorers tested are `Threshold` and `Weights` (for the weighted methods). `Threshold` was implemented to give the user control of how tolerant the pipeline should be in flagging paths (based on normalized score) and `Weights` to allow for potential model performance improvements based on the domain knowledge that deeper path tokens might be more valuable than shallow ones. For example, version `12.03.05`, a productlinetype level 3 token, is much more specific than `controller`, a productlinetype level 1 token, adding merit to providing more weight to said version tokens.

A summary of some of the scoring methods tested, with said hyperparameters is shown below.

Table 2: Scorer Performance Compared

Scorer	Threshold	Weights	Total.Path.Count	Avg.Recall	Avg.Precision	F1.Score
weighted count	0.9	[0, 0.2, 1, 1, 0.45]	590	23.41%	28.87%	0.259
weighted count	0.8	[0, 0.2, 1, 1, 0.45]	1172	29.39%	25.07%	0.271
weighted count	0.7	[0, 0.2, 1, 1, 0.45]	4334	41.11%	20.67%	0.275
weighted count	0.6	[0, 0.2, 1, 1, 0.45]	14734	53.95%	17.45%	0.264
weighted sum	0.9	[0, 0.2, 1, 1, 0.45]	856	33.03%	30.37%	0.316
weighted sum	0.8	[0, 0.2, 1, 1, 0.45]	3367	38.85%	30.29%	0.340
weighted sum	0.7	[0, 0.2, 1, 1, 0.45]	7318	49.55%	26.48%	0.345
weighted sum	0.6	[0, 0.2, 1, 1, 0.45]	13219	58.80%	23.22%	0.333
simple count	0.9	N/A	811	23.38%	33.42%	0.275
simple count	0.8	N/A	912	25.10%	33.49%	0.287
simple count	0.7	N/A	3744	38.41%	24.75%	0.301
simple count	0.6	N/A	19286	62.55%	16.51%	0.261
simple sum	0.9	N/A	981	31.92%	28.69%	0.302
simple sum	0.8	N/A	1783	38.85%	25.16%	0.305
simple sum	0.7	N/A	4491	48.88%	20.82%	0.292
simple sum	0.6	N/A	8482	63.65%	19.03%	0.293

aDolus has communicated that it is of high importance not to miss target files than if the pipeline accidentally tags incorrect files, i.e false negatives are of more importance than false positives. This is because the potential cost to aDolus' clients should a vulnerable file be left undetected could be catastrophic compared to investigating a miss-labelled file and determining upon closer inspection that is fine. To ensure the model was aligned with this requirement, recall[2] and precision[3] have been used as model evaluation metrics in comparing scorers as opposed to accuracy and false positive rate.

A common trend seen among all scorers is that as recall rises, precision decreases. While it might be tempting to choose a model purely with a high recall, a balance of the two metrics is important. At low precision, the number of recommended paths can balloon to very large volumes (18,000 compared to 3,000), making the task of human verification exponentially difficult and defeating the purpose of the pipeline. Furthermore, the pipeline can easily get a perfect recall of 100% by simply returning all of file paths for Rockwell given Rockwell advisories, but then it isn't doing much value is it. Therefore, to strike a balance between recall and precision, the team utilized F1 Score[4] for choosing a model scorer.

Based on F1 Score, it was seen that summation methods perform better overall than count based methods. This is attributed to there being value in giving weight to tokens that occur more frequently. Tokens like `FactoryTalk` or `Stratix`, which occurring numerous times in an advisory report, can be highly beneficial to correctly building weight on paths containing these terms. However, this assumption is not always the case. By examining poorly performing advisory reports, the team discovered that certain tokens, labelled `trouble_tokens`, can wreak havoc on a scorer's ability to correctly find a path. High occurrence tokens

[2]**Recall** refers to the what percentage of true matches are captured.

[3]Of the model-recommended paths, **precision** refers to the percentage of were correct.

[4]**F1 Score** conveys the balance between precision and recall and reaches its best value at 1.

like `system`, `products`, or `control` commonly occur across many reports and incorrectly shift scores to false paths. To make matters further complicated, these trouble tokens are varied and often spread all through different `productlinetypes` making their identification difficult.

To mitigate against said `trouble_tokens`, the pipeline has been coded to have a filter that can either automatically (using `obtain_trouble_tokens()`) or from a user provided list remove `trouble_tokens`. If ran as automatic, the pipeline benefits from bulk processing of many advisories, because it will look at tokens that frequently occur across the reports and flag them for removal if the occur above a certain percentage. For example, `system` occurs in every Rockwell advisory; hence, it is likely not a useful token to identifying the details of a report.

Table 3: Scorer Performance With Trouble Tokens Removed

	Scorer	Threshold	Max.Token.Freq.	Total.Path.Count	Avg.Recall	Avg.Precision	F1.Score
13	simple sum	0.9	1	981	31.92%	28.69%	0.302
14	simple sum	0.8	1	1783	38.85%	25.16%	0.305
15	simple sum	0.7	1	4491	48.88%	20.82%	0.292
16	simple sum	0.6	1	8482	63.65%	19.03%	0.293
69	simple sum	0.9	0.4	1847	44.93%	30.65%	0.364
70	simple sum	0.8	0.4	2751	58.27%	34.12%	0.430
71	simple sum	0.7	0.4	6838	67.73%	30.70%	0.422
72	simple sum	0.6	0.4	10745	72.37%	27.36%	0.397

An alternative approach to trouble token removal that was tested was to use the structure of reports to filter out non-valuable and potentially miss-leading sections of advisory reports. The `parse_advisories.py` script was written such that it allows for new condensed-report parsers to be added without changing the pipeline script. Testing a condensed-report parser for Rockwell yielded improved results, but not as much as methods to remove `trouble_tokens`. Therefore, a condensed-report parser was not implemented as default because its benefit was marginal for the complexity it added. Combining the two methods also showed no added benefit beyond `obtain_trouble_tokens()`, likely because the additional 'non-useful' text left in the reports actually benefited the pipeline by making it easier for `obtain_trouble_tokens()` to find commonly occurring tokens.

File Searching

After target paths were identified for each report, the next step is to search for the affected files in the target paths through the SQL database. Each path has an unique product line identifier Guid in ProductLineView table. The product line Guids of the path are then used to query ChildFIleView table and VendorFileProductLine table to find the IDs for the files and their child files living in the target paths. Then, the file IDs are used to query the FileView table to get all the information of the files. Those files will be further filtered based on their extensions for executables file, which are usually the targets of attacking. Finally we get all the affected executable files for each target path.

Run-time optimization

In addition to applying various data science methods in the implementation of the periculum pipeline, the team sought to improve the run-time of the various steps. These efforts are discussed in more detail in the Appendix.

Data Product and Results

The output of this pipeline is a `match_summary_report.csv` file which contains the advisory report information(names and S3 bucket links), the matched path information(Paths, ProductLineGuid, Normalized score) and the affected file information(FileID, use `all_columns=True` for match_summarize.py if all the

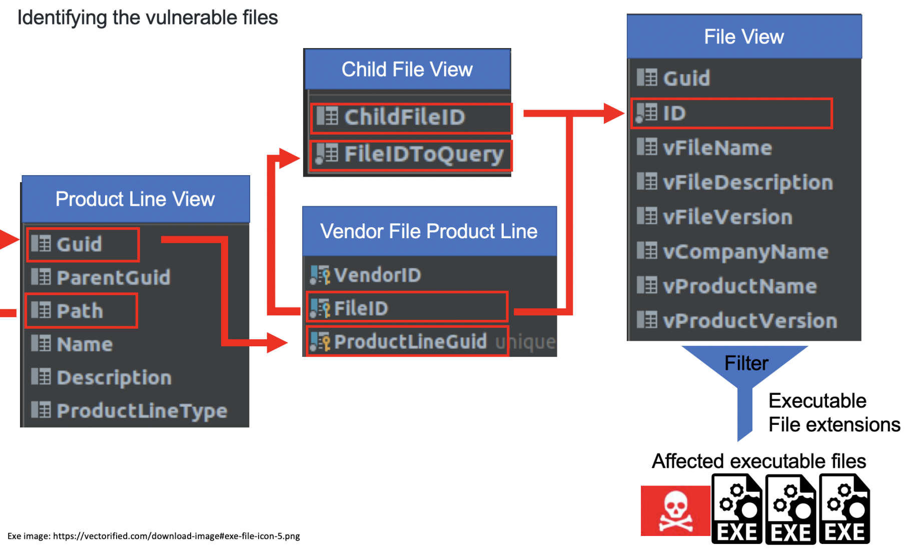


Figure 4: Overview of File Search

features of the files are needed). The included information are selected according to the needs of our capstone partener.

To facilitate the needs of our capstone partner to have human-in-the-loop to verify matches, a companion report in PDF format is generated (See sample here). The companion report contains summary statistics about the number of affected files for each advisory report. It also lists all the paths matched to each report along with the matching scores, allowing a reviewer to quickly scan for erroneous paths in an advisory report of interest.

This pipeline is fully automated from the raw advisory reports to the final output of match_summary_report.csv and companion report. Our capstone partner can run this pipeline in two ways:

1) Install all the dependencies and run the pipeline using the tool Make in local computer or EC2; and
2) Use the Docker image with environment dependencies and the tool Make in local computer or EC2.

These options add more flexibility for our capstone partner in running the pipeline. Details for using periculum are explained in the project GitHub repository.

Conclusion and Recommendations

The pipeline provided to aDolus consumes a collection of vulnerability reports and find products and files within aDolus's database that match the content of the vulnerabilities. The pipeline is able to process documentation of different formats and structures, and in the span of a few minutes, process up to 100 advisory reports. As demonstrated by the Rockwell data set, the pipeline can find the paths in an advisory up to 73% of the time and a companion report will be produced to address shortcomings by allowing a human to verify the quality of matches. In summary, the pipeline provided meets the criteria set by aDolus and can be implemented as a tool to assist in the matching of vulnerability reports to affected files. With further development, it may be possible to integrate the pipeline into an automated system with minimal human over-site but further steps are required.

Some of the recommended actions for future work are as follows:

- **Confirm the manual validation data.** The dataset used to validate scorers was built by the team without domain expertise. An expert at aDolus could likely produce a more accurate dataset.

- **Expand product trees to include vendors beyond Rockwell and Schneider Electric**. Currently the model is limited to processing advisories for only these two vendors because only they have product line trees that can be queried.
- **Test other vectorizers.** Other vectorizers such as tfidf may be able to enhance performance by weighting path tokens differently based on frequency across paths.
- **Perform an optimization of scorer methods.** Due to time constraints, only a few configurations of scorers where able to be tested. Different weighting systems for `weighted_count` and `weighted_sum` methods may yield higher performing scorers if an exhaustive grid search through available hyperparameters was performed. Furthermore, other scorers not considered, such as an ensemble method could improve performances.
- **Add additional parameters to pipeline output.** Time constraints did not allow for it, but adding parameters like **CPE** and **url information** in the companion reports may assist human validation.
- **Add Version context classification.** Often security advisories have various version formats. For example, one advisory might say a vulnerability affects version "5.2 and earlier", whereas another "later that 3.4". Currently, the pipeline is limited to finding versions "5.2" and "3.4" and will not know "earlier" or "later" because these tokens do not exist in path vocabularies. Creating an additional layer of functionality to determine the version context of a report, whether "earlier" or "later", could aid in increasing the precision of the pipeline output by eliminating paths that are out of version context.
- **Add file-properties to database.** Because of the very limited file details, the team was only able to match to files through product paths. If additional details were added to files within aDolus' database, it may become possible to find files directly instead of through paths.

Appendix

Unique challenges

Due to the lack of labeled data, the supervised method is not applicable here. A traditional unsupervised solution is to extract the path information use regular expressions from the advisory reports and then match it to the paths in the database. However, there are three problems making this method unsuitable:

1) Since the advisory reports are in several different formats, this solution needs deep understanding of all the report structures to extract information from the reports;
2) Even with the advisory reports in the same format, the path information is written in different places, making it harder to locate the useful information; and
3) The path information in the advisory reports may not match exactly to the paths in the database, thus certain level of tolerance for mismatches needs to be enabled to identify the paths.

To overcome these problems, we proposed a universal method which can be applied to advisory reports in any format and structure. It can also be tailored to different levels of tolerance for mismatches. This method vectorizes the advisory reports and paths in the database into tokens. Each advisory report will be evaluated for how close it matches to each path in the database. Target paths for advisory report will be selected based on the scoring metrics. Lastly, affected files will be identified within the target paths.

Run-time optimization

Vectorization improvements

While the algorithm worked in terms of business logic, it was extremely time-consuming and inefficient, as the naive approach for this is very computationally expensive. Rockwell Automation ("Rockwell"), for example, has ~7000 paths available, which generated ~1300 tokens. This resulted in a very large, but very sparse matrix; as a result the initial algorithm took 10 minutes to process each report. From there, we made several iterations of improvements to the algorithm.

The first improvement to the algorithm was to consider only the paths that generate an non-zero matrix (i.e: only rows with tokens present), as well as only considering tokens that are present in any document (i.e: only columns with tokens present). This resulted in a 5x speedup, and reduced the per-report run-time from 10 minutes to 2 minutes.

The next improvement was the usage of Boolean Masking for our sparse matrix when comparing values. Instead of iterating through each cell and comparing the values, a Boolean mask was generated for each row of the `fit_transform` token matrix and the `transform` token matrix for each document; only the cell values that are true after performing a bit-wise `and` operations on the masks are kept.

Additionally, we also used vectorized operations. Instead of looking at each row & column to determine if they're non-zero using the naive approach with `np.sum() > 0` while iterating through each row and column ($O(m \cdot n)$, where `m` and `n` are column and row sizes, respectively), we used vectorized operations built into `numpy` (`ndarray.T`), which vastly improved run-time speed. ### Run time optimization

SQL query call minimization

As querying aDolus' SQL database takes time, the team sought to minimize these occurences where possible in the following cases by avoiding SQL calls in loops for multiple advisory reports in `path_scorer.py`. This change reduced run-time by an order of magnitude.

References

1. Die lage der it-sicherheit in deutschland 2014.

2. Wikipedia contributors. Man-in-the-middle attack — Wikipedia, the free encyclopedia. (2004).

3. ADolus - framework for analysis and coordinated trust.

4. Wickham, H. *Feather: R bindings to the feather 'api'.*

5. AWS. Boto3 reference.

6. Richardson, L. Beautiful soup documentation. *April* (2007).

7. Shinyama, Y. Community maintained fork of pdfminer. (2007).

8. Pedregosa, F. *et al.* Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* **12**, 2825–2830 (2011).

9. Kleehammer, M. *Python open database connectivity (odbc) bridge.*