Predicting Sabotaged Open-source Libraries

Alexander Petty[1], William Glisson[1] and Ryan Benton[2]
[1]Louisiana Tech University, Ruston, USA
[2]University of South Alabama, Mobile, USA

ajp028@latech.edu
glisson@latech.edu
rbenton@southalabama.edu

Abstract: Open-source software provides free and publicly available software maintained by the open-source community. The variety of contributors creates an environment conducive to the intentional and unintentional introduction of software bugs by participating organizations. Enemy nation-states and independent hackers can exploit these attack vectors to gain access to industry and government systems. Repositories of known vulnerabilities and tools to check vulnerable versions and analyze code exist, but realistically, reviewers can miss issues within many repositories due to constant updates and technological advances. Hence, this research investigates an alternative, non-code-based method for identifying high-risk repositories using repository metadata and commit history, which, when coupled with machine learning, enables us to identify at-risk repositories at rates above 60%. This was achieved using a dataset composed of 41,710 repositories. The contribution of this research is twofold. First, it presents an empirical evaluation of the viability of a non-code-based analysis approach to detecting high-risk, i.e., potentially compromised code repositories. Second, it provides foundational research for non-code-based filtering of open-source repositories, potentially accelerating software investigations and reducing resource requirements.

Keywords: Vulnerabilities, Supply chain attack, Open source, Machine learning, Malicious code

1. Introduction

Open-source software encourages collaboration and innovation for enterprise-level software development on a global scale SolutionsHub (2023). Mercedes-Benz claims that Free and Open Source Software (FOSS) "has become the foundation of just about every application across the industry" (Mercedes-Benz, 2021). According to a GitHub report, 90% of companies use open-source software (GitHub, 2023). Companies pursue open-source software to enhance their development capabilities and to reduce financial dependencies on licensed software (Millidge, 2018). These enhanced capabilities translate into reduced core code development timelines and an increased ability to focus on custom products and features (Barabakh and Meir, 2023). However, like all software, open-source code can suffer from bugs and glitches. Some software defects lead to holes in security that allow attackers to exploit the software to gain access to a system or network for malicious purposes (Clark, Doran, and Glisson, 2018); (Fitzgerald et al., 2023); (Glisson et al., 2015). A report from the Synopsys Cybersecurity Research Center states that of the 1,703 code bases examined, 96% contained open-source code, and 84% contained at least one vulnerability (Synopsys, 2023).

According to the FBI's Internet Crime Complaint Center (IC3) 2022 Internet Crime Report, cybercrime cost the United States more than $10 billion in losses, where $34 million was from ransomware and $9 million from malware (Federal Bureau of Investigations, 2022). Attempting to detect problems at an early stage is a standard countermeasure in the cybersecurity arena (Ceballos Delgado et al., 2021); (Luckett et al., 2018); (McDonald et al., 2017); (Nguyen et al., 2020). In fact, it has been noted that there are tools that the community uses to identify vulnerable open-source libraries (Koussa, 2021). However, several of these tools use a combination of known vulnerabilities from the NIST Vulnerability Database and library versions to identify known vulnerable code.

In addition, commercial solutions can potentially require a database of reported and discovered vulnerabilities to identify security issues. The reliance on discovered vulnerabilities makes them ineffective against zero days (Ohm et al., 2020). It has been noted that open-source software is susceptible to these supply chain attacks and poisoning (Xu et al., 2022). The escalating cost of cybercrime, as well as the accelerating accessibility and growing adoption of open-source software in corporate and government development projects, suggest that organizations should be able to identify high-risk repositories. Open-source repositories can track changes through commits, which researchers could use to detect and identify poisoning. To investigate this idea, this research investigates the hypothesis that it is possible to identify a high-risk poisoned library from the commit history and library metadata. The hypothesis raises the following questions:

- What information from commits and library metadata contributes to the likelihood of a library being at risk?

- Can enough data be gathered to predict risk accurately?
- Can ML be used to predict if a library is at risk?

The contribution of this research is twofold. First, it presents an empirical evaluation of the viability of a non-code-based analysis approach to detecting high-risk, i.e., potentially compromised code repositories. Second, it provides foundational research for non-code-based filtering of open-source repositories, potentially accelerating software investigations and reducing resource requirements.

The paper is organized as follows; section two provides a background of relevant research. Section three addresses methodology and data acquisition. Section four presents research results and analysis. Section five contains the conclusion and offers future work on this topic.

2. Relevant Work

Malicious actors are increasingly exploiting open-source libraries for software supply chain attacks. Ohm et al. (2020) focus on the programming languages JavaScript, Python, Ruby, and Java. The malicious actors use typo squatting, trojan horses, and infecting upstream libraries to compromise their targets. The study identified 469 malicious packages designed to damage computer systems. By avoiding automated data collection, Ohm attempts to ensure that they did not miss a package due to negligence. Looking at package history shows that malicious packages are available for an average of 209 days before removal. Currently, there is no effective method for quickly identifying these malicious libraries. Security experts use machine learning to identify known, recurring vulnerabilities in code. Xu et al. (2022) use a plethora of open-source repositories to train machine learning on real-world samples. Xu's data is from GitHub and focuses on code analysis. Overall, the model trained from GitHub code had a precision of 84.1% and an accuracy of 86.1%, demonstrating the value of GitHub repositories.

Chan and Chandy (2022) analyze Common Vulnerabilities and Exposure (CVE) databases and GitHub commits to studying how often CVEs document the fixes for corresponding vulnerabilities. Using the CVE database patch records and checking the GitHub repository commits to find the number of CVEs with patches.

Ponta, Plate, and Sabetta. (2018) use a combination of static and dynamic analysis to identify vulnerable applications in open-source software, as results indicated the combination furthers vulnerability detection. Their approach is a code-centric method that combines static and dynamic analysis. After detecting a vulnerable library through abstract syntax trees, the goal is to investigate an attack's ability to reach the vulnerable portion of the library. The problem they highlight is the need for a comprehensive knowledge base for vulnerabilities that security experts can reference alongside specialized detection tools.

Bhandari, Naseer, and Moonen (2021) created a tool to create a vulnerability dataset from the CVE records in the National Vulnerability Database (NVD) by collecting vulnerable code and proposed fixes from CVE records. Using the knowledge base, their tool would also provide helpful information on vulnerabilities such as programming language, security metrics, and patches. This enabled quick identification of vulnerable code and the corresponding patch to fix it, potentially leading to an automated patching tool for known vulnerabilities.

Russell et al. (2018) used open-source libraries to create data for machine learning algorithms using a combination of static analysis, dynamic analysis, and commit-message/bug-report tagging. They used neural network classification combined with an ensemble classifier, random forest, to create their model. Their results showed that their machine learning methods outperformed traditional static analysis tools because they are faster and can be tuned to achieve a desired level of precision.

Chakraborty et al. (2022) discuss whether the advances in Deep Learning improved automated vulnerability detection. They find that Deep Learning detection is less than half as effective as other methods. The team attributes the performance drop to the quality of the training data and model choices. They propose a more systematic and principled way to gather training data, and models showed a 33.57% boost in precision and a 128.38% boost in recall. They highlight the potential of Deep Learning-based vulnerability detection when done correctly.

Research exists that uses static and dynamic code analysis to detect vulnerabilities. While some of the methods utilize GitHub to obtain a richer/more diverse training set, minimal research investigates repository information other than code or patches to identify potentially vulnerable repositories.

3. Method

This section provides the approach used to identify potentially vulnerable repositories. This includes the data gathered from repositories, such as user actions, configurations, and summary statistics; the code is explicitly not utilized. The data cleaning, the labeling of the repositories, and Machine Learning (ML) methods utilized are provided in this section. The software employed in this research is presented in Table 1.

Table 1: Hardware and Software

Software	Version
RapidMiner Studio	10.1.002
RapidMiner XGBoost	0.1.3
RapidMiner Python Scripting	10.0.1
RapidMiner Operator Toolbox	2.16.0
Golang	1.20.5
Python	3.10.6
Python imbalanced-learn	0.11.0

3.1 Data Acquisition

This study focuses on popular GitHub repositories for analysis. Believing that frequently used repositories are more likely to incur vulnerabilities, this research gathers information from actively used repositories. The number of GitHub stars is used as an overall indicator to evaluate a repository's rank. To reduce the dataset size, only repositories with 1,500 or more stars were considered, yielding 41,710 repositories.

The project focuses on data from user actions, configurations, and summary statistics. Any user information gathered from the repositories is saved for future work. Commit history is limited to the last four years to save space and time. The method used to determine if a repository ever contained a vulnerability consisted of two parts. The reported issues from each repository are used to mark any repository with an issue that has keywords indicating security patches or vulnerabilities in the title or body, using regular expressions.

For the second part, the results are combined with the results of MITRE's CVE keyword search. Using MITRE's Application Program Interface (API) for CVEs, a repository is marked when any repository's user and name combination returned results. The list of all positive results from both was labeled as repositories with possible vulnerabilities and given a positive mark in the label category "posvuln"; otherwise, it was given a negative mark (no problem). The split between positive and negative results for this label was 13,732 positive and 27,977 negative.

3.2 Feature Selection and Data Discovery

RapidMiner Studio 10.1.002 was used for all the analysis, cleaning of data, feature selection, and testing of the Machine Learning methods. The feature selection begins with removing useless attributes and correlated attributes operators from the complete set of attributes in Table 2. The first operation removes id-like attributes and the second removes attributes that have a correlation value above 0.95. Ensuring consistent results required setting a seed for all RapidMiner modules that rely on a random seed. The seed used, 123456789, was chosen arbitrarily and was set within every applicable module. The set seed ensured that results were repeatable across runs and that direct comparisons among methods could be made. Feature selection continues with the implementation of forward selection and backward elimination. Each RapidMiner module uses one ML Algorithm, of the tried algorithms, internally to perform the feature selection. The backwards elimination and forward selection allowed for a reduction from the already reduced initial 60 features to fit to the separate ML algorithms. When using the Synthetic Minority Oversampling Technique (SMOTE) or Tomek-link reduction methods in a test, they are applied after the use of reduction techniques and before the machine learning algorithms. The SMOTE up-sampling and Tomek reduction methods are used after feature selection to balance and define the feature areas within the data. They balance the number of positive and negative data points and reduce overlap between the two groups to aid the algorithms' learning by providing more well-defined boundaries. The lack of Tomek-Link reduction in RapidMiner requires the python module with the imblearn v0.11.0 library. When required, the data is normalized using Z-transformation and piped to the following ML algorithms for evaluation: Decision Trees, Random Forest, k-NN with k=5, 11, and

15, SVM with analysis of variance (ANOVA), Generalized Linear model, and XGBoost 0.1.3 using logistic regression and regression with squared loss.

4. Results and Analysis

After removing useless and correlated attributes, 44 attributes remained from the original 60. Initial performance in some of the models was imbalanced. Across all models, recall rates were low for positive classification and extremely high for negative classification, except for k-NN, SVM, and XGBoost. Among the models, the lowest positive recall was 2.52%, and the highest negative recall was 99.66%. Unfortunately, all the models had an inverse relationship between recall and precision.

The better models with more even splits, such as XGBoost, had combined recall of 67.04% with splits of 63.29% for positive and 70.79% for negative but had imbalanced precision of 31.03% for positive and 90.28% for negative. Continued testing showed better performance and balance when using a training set of only 2,000 evenly split observations compared to 20,000 observations also evenly split. An example of the imbalance is the k-NN (k=11) model. The same k-NN (k=11) model went from 28.03% positive classification precision to 46.79% when switching from a training set of 20,000 to 2,000. This showed imbalances in the gathered dataset.

Tables 3 and 4 are the results of the decision tree and random forest methods, split between the forward selection (FS) and backward elimination (BE) selection methods. Even though they show precision at 60%-70% and a high level of learning from the models, they have low positive recall. A bias toward negative can result in the model not detecting a true vulnerbility and is not ideal for an early warning system. Random Forest performs better than the lone decision tree in predicting more true positives, but is slightly inferior in precision by including more false positives. Further analysis showed that the models suffer from data imbalance, as they perform worse rather than better with more training data.

Table 3: Decision Tree

Selection Method	Training Size	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	67.53%	70.36%	13.45%	96.95%
Forward Selection	20000	56.53%	83.76%	7.77%	98.76%
Backwards Elimination	2000	69.29%	69.97%	11.22%	97.65%
Backwards Elimination	20000	56.07%	83.76%	7.80%	98.73%

Table 4: Random Forest

Selection Method	Training Size	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	59.25%	74.24%	34.78%	88.71%
Forward Selection	20000	54.05%	83.84%	8.57%	98.49%
Backwards Elimination	2000	60.02%	72.63%	26.92%	91.54%
Backwards Elimination	20000	51.52%	83.83%	8.65%	98.31%

Table 5 contains all the results of the k-NN iterations of 5, 11, and 15. The results from the model have the best performance so far in positive recall classification, but they suffer with positive precision. The higher recall with k-NN and not with the other classification methods, shows that the data's separation is not well defined.

Table 5: k-NN

Selection Method	Training Size	K	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	5	44.27%	76.95%	58.62%	65.17%
Forward Selection	20000	5	25.87%	88.48%	59.46%	64.92%
Backwards Elimination	2000	5	45.04%	76.51%	55.87%	67.82%
Backwards Elimination	20000	5	26.34%	88.70%	60.02%	65.16%
Forward Selection	2000	11	46.76%	76.85%	55.06%	70.41%

Selection Method	Training Size	K	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	20000	11	27.67%	88.99%	59.73%	67.59%
Backwards Elimination	2000	11	46.79%	77.49%	57.41%	69.18%
Backwards Elimination	20000	11	28.03%	88.97%	59.06%	68.53%
Forward Selection	2000	15	47.47%	76.99%	54.81%	71.37%
Forward Selection	20000	15	27.76%	89.19%	60.77%	67.17%
Backwards Elimination	2000	15	46.77%	76.60%	54.07%	70.95%
Backwards Elimination	20000	15	28.53%	89.06%	59.00%	69.32%

Tables 6 and 7 display the performance of the Generalized Linear Model and SVM. The drops in recall with the larger training sizes further highlight an issue with poorly defined data separation.

Table 6: Generalized Linear Model

Selection Method	Training Size	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	58.07%	73.83%	33.44%	88.60%
Forward Selection	20000	43.61%	83.14%	3.11%	99.17%
Backwards Elimination	2000	54.24%	76.13%	45.63%	81.83%
Backwards Elimination	20000	34.90%	87.72%	44.08%	82.93%

Table 8 contains the results of the XGBoost model. The ensemble method, XGBoost, performs the best results of all the preliminary tests. The recall is split 61.27% to 68.55%, demonstrating that is better than guessing. The precision is lower than others at 47.90% positive precision which shows a lower learning of the data.

Table 7: SVM

Selection Method	Training Size	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	57.20%	72.93%	29.55%	89.57%
Forward Selection	20000	-	-	-	-
Backwards Elimination	2000	50.80%	77.46%	53.38%	75.61%
Backwards Elimination	20000	37.99%	84.83%	19.59%	93.36%

Table 8: XGBoost

Selection Method	Training Size	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	2000	57.23%	71.43%	21.77%	92.32%
Forward Selection	20000	0.00%	82.81%	0.00%	100.00%
Backwards Elimination	2000	47.90%	78.95%	61.27%	68.55%
Backwards Elimination	20000	31.03%	90.28%	63.29%	70.79%

To mitigate the separation issues in the data, we implemented the use of SMOTE upsampling with a 30-70 training-testing split to add new data points for the minority class to improve results. A lack of indication in the open-source libraries of vulnerability patches and discrepancies between CVE reports and these libraries likely led to data imbalance and overlap. Rapid miner does not include SMOTE in the base application, but by installing the operator toolbox extension 2.16 SMOTE becomes an available operator. Comparing results after SMOTE upsampling, there was an overall drop in recall but an increase in precision. The test was set up to perform two separate test each with half of the of the models to reduce overall run time. However, the tests excluded XGBoost because of errors during testing with SMOTE.

Table 9 shows the SMOTE results from the tree models. The Decision Tree model's recall decreased with SMOTE but had a major increase in precision meaning the model learned better with the upsampling. This

shows there is an overlap from the up sampling. The Random Forest model had an increase in positive recall, but a decrease in precision showing an overlap within the data.

Table 9: SMOTE first batch testing results

Selection Method	Model Type	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	Decision tree	71.72%	68.98%	10.22%	98.02%
Backwards Elimination	Decision tree	80.85%	67.44%	1.84%	99.79%
Forward Selection	Random Forest	49.82%	76.95%	55.83%	72.39%
Backwards Elimination	Random Forest	55.13%	74.95%	43.86%	82.47%

Table 10 shows how the k-NN model increased its positive recall and improved with more neighbors, higher k, instead of decreased. SVM shows an improvement in precision and recall for the backwards elimination method with a 12% increase in positive precision and only a 7% decrease in negative precision and a more balanced recall of 58.71% positive and 71.81% negative compared to 19.59% positive and 93.36% negative.

Table 10: SMOTE second batch testing results

Selection Method	Model Type	Precision Positive	Precision Negative	Recall Positive	Recall Negative
Forward Selection	k-NN (k=5)	43.72%	75.17%	59.89%	64.05%
Backwards Elimination	k-NN (k=5)	44.28%	75.72%	58.13%	64.09%
Forward Selection	k-NN (k=11)	44.21%	75.48%	57.33%	64.48%
Backwards Elimination	k-NN (k=11)	45.46%	77.06%	61.24%	63.93%
Forward Selection	k-NN (k=15)	45.71%	77.13%	61.12%	64.36%
Backwards Elimination	k-NN (k=15)	46.05%	77.17%	60.80%	65.03%
Forward Selection	Generalized Linear Model	57.64%	72.91%	33.42%	87.94%
Backwards Elimination	Generalized Linear Model	54.39%	75.36%	46.02%	81.06%
Forward Selection	SVM	32.92%	65.00%	99.83%	0.15%
Backwards Elimination	SVM	50.55%	77.99%	58.71%	71.81%

The other models share a similar trend of favoring backwards selection with an overall increase in positive recall but a decrease in precision. To normalize all feature across the models, further tests used features gathered by combining SMOTE upsampling with the forward selection and backwards elimination methods and choosing the top five recall models.

The five chosen ML models were: k-NN (k=11) using SMOTE and the backwards elimination method with a 61% positive and 63% negative recall, k-NN (k=15) using SMOTE and the forwards selection method with a 61% positive and 64% negative recall, Random Forest using SMOTE and the forward selection method with a 55% positive and 72% negative recall, k-NN (k=5) using the forward selection method with a 58% positive and 65% negative recall, and XGBoost using the backwards elimination method with a 61% positive and 68% negative recall.

These five models had positive and negative recall classifications that was 55% and above while maintaining a reasonably balanced precision. To reduce the total number of evaluated attributes, attributes found in at least three of the top five models were included in the final feature set. The included features were: has downloads, lower quartile removed, has wiki, Fork, total pulls, upper quartile commit inserts, total branches, lower quartile added, avg commit files, total contributors, and median commit files.

The resulting feature set includes attributes corresponding to repository activity levels, indicating a link between user engagement and risk for a repository to have a vulnerability. The last tests achieve the best results when testing the best performing models with the new feature set by adding Tomek-link under-sampling. The tests use Tomek-link under-sampling to separate the data into more defined groups and combine it with the SMOTE upsampling used previously. Using the same data cleaning and normalization to

test SMOTE Tomek-link performance, tests used cross-validation on random forest, k-NN (k=15), XGBoost, SVM (ANOVA), and Generalized Linear Models.

Table 11 contains the results of three models that are split by model into four groups: no SMOTE or Tomek-Link, SMOTE, Tomek-link, and both SMOTE with Tomek-link. All three models had a decrease in precision from the upsampling and under-sampling. The limited feature selection caused lower positive recall rates for the models before SMOTE and Tomek-link, but all models improved in recall with SMOTE upsampling.

Table 12 contains the same test with two other models. The Generalized Linear Model follows the pattern of a decrease in precision after SMOTE and Tomek, but SVM increases in both recall and precision, achieving the most balanced result.

Table 11: SMOTE-Tomek result comparison

Sampling Method	Model Type	Precision Positive	Precision Negative	Recall Positive	Recall Negative
None	Random Forest	71.37%	70.12%	15.90%	96.87%
SMOTE	Random Forest	41.68%	80.29%	76.14%	47.70%
Tomek	Random Forest	64.34%	72.00%	26.46%	92.80%
S&T	Random Forest	48.03%	77.55%	59.73%	68.28%
None	k-NN (k=15)	54.79%	72.87%	34.87%	85.87%
SMOTE	k-NN (k=15)	45.16%	77.01%	61.49%	63.37%
Tomek	k-NN (k=15)	51.53%	74.51%	44.66%	79.38%
S&T	k-NN (k=15)	43.01%	77.22%	65.49%	57.40%
None	XGBoost	62.92%	73.50%	33.72%	90.25%
SMOTE	XGBoost	57.03%	75.23%	43.77%	83.82%
Tomek	XGBoost	58.01%	75.41%	43.93%	84.39%
S&T	XGBoost	54.72%	76.44%	49.94%	79.71%

The results show that the Tomek-link under-sampling method increases precision but decreases recall. However, when balanced with SMOTE's recall increase and precision decrease, the test results get a 'balance' of recall and precision. Models that benefit from better defined data separation, k-NN and SVM, see the most benefits and reach recalls and precisions of 60%.

Table 12: SMOTE-Tomek result comparison

Sampling Method	Model Type	Precision Positive	Precision Negative	Recall Positive	Recall Negative
None	SVM (ANOVA)	42.85%	71.22%	38.31%	74.92%
SMOTE	SVM (ANOVA)	50.84%	77.44%	56.60%	73.14%
Tomek	SVM (ANOVA)	48.06%	71.91%	35.31%	81.27%
S&T	SVM (ANOVA)	46.22%	79.68%	68.32%	60.98%
None	Generalized Linear Model	65.69%	70.93%	20.97%	94.62%
SMOTE	Generalized Linear Model	51.11%	76.31%	52.29%	75.44%
Tomek	Generalized Linear Model	62.08%	72.38%	29.03%	91.30%
S&T	Generalized Linear Model	47.94%	77.50%	59.66%	68.20%

Although the results are not perfect for detecting high-risk libraries, they do show a clear, predictable pattern. The data from testing provides evidence that this pattern is consistent and can identify potential vulnerabilities in software systems.

5. Conclusions and Future Work

The results of the research indicate a link between public GitHub repo data and vulnerability risk indicators. With a recall and precision of around 60%, the models demonstrate that the collected data can predict high-

risk repositories. Moreover, the highest-performing models, SVM and k-NN, reveal that the data contains defined groups and is suited to classification via distance analysis and linear separation. Hence, this investigation provides a foundation for subsequent research into repository analysis, serving as an early-warning or vetting process that can bolster code-analysis solutions.

These findings highlight potentially malicious accounts for additional investigation and initial screenings on code repository platforms such as GitHub. However, there is also a risk that malicious actors could misuse the method to identify vulnerable repositories. Using this method could enhance current vulnerability-search methods by reducing the pool of potentially vulnerable code libraries before an actor feeds code into more computationally expensive skill-based language models to build exploits. Alternatively, repository maintainers or security researchers can pre-screen code flagged using the same method to patch identified vulnerabilities.

Future work will investigate alternative machine learning methods, including neural networks and deep learning, as well as further analysis with distance-specific techniques such as subspace analysis. In addition, research will expand to include data from less frequently used and less popular repositories to compare results across different levels of user activity. Another consideration stems from the timeframe of the data collection. Research will need to account for significant changes in potential results due to the advent of artificially generated code, and it will be closely monitored as a variable in future iterations.

A deeper investigation into analyzing active repository contributors as indicators for compromise could enable the identification of accounts that intentionally commit vulnerable code. Research would begin with using user data to construct maps of all users associated with compromised libraries and associated activities. Furthermore, additional analysis would use commit history patterns, code preference, and repository commit history as the initial feature set for prediction. The end goal is to profile user accounts by commit history and account indicators to identify malicious actor accounts.

Ethics and AI Declaration: No ethical clearance was required, and no Generative AI was used in process of this paper.

References

Barabakh, H. & Meir, A. (2023). How To Build a Successful SaaS Product Using Open Source. [online] Available at: https://blog.payproglobal.com/build-saas-using-open-source

Bhandari, G., Naseer, A. & Moonen, L. (2021). CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering.* Association for Computing Machinery.

Ceballos Delgado, A. A., Glisson, W., Shashidhar, N., Mcdonald, J., Grispos, G. & Benton, R. (2021) Deception Detection Using Machine Learning.

Chakraborty, S., Krishna, R., Ding, Y. & Ray, B. (2022). Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering,* 48**,** 3280-3296.

Chan, N. & Chandy, J. A. (2022). Extracting vulnerabilities from github commits , 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) , IEEE.

Clark, G., Doran, M. & Glisson, W. (2018). A malicious attack on the machine learning policy of a robotic system , 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE) , IEEE.

Federal Bureau Of Investigations. (2022). Federal Bureau of Investigations Internet Crime Report.

Fitzgerald, J., Mason, T., Mulhair, B. & Glisson, W. (2023). Exploiting a Contact Tracing App to Attack Neighboring Devices , Hawaii International Conference on System Sciences (HICSS) , HICSS , ISBN 0998133167 , Conference Proceedings.

Github, (2023). The state of open source software , GitHub.

Glisson, W. B., Andel, T., Mcdonald, T., Jacobs, M., Campbell, M. & Mayr, J. (2015). Compromising a medical mannequin , Americas' Conference on Information Systems (AMCIS) , AMCIS , Conference Proceedings.

Koussa, S. (2021). 13 tools for checking the security risk of open-source dependencies , TechBeacon.

Luckett, P., Mcdonald, J. T., Glisson, W. B., Benton, R., Dawson, J. & Doyle, B. A. (2018). Identifying stealth malware using CPU power consumption and learning algorithms. *Journal of Computer Security,* 26**,** 589-613 , ISSN 0926-227X.

Mcdonald, J. T., Manikyam, R., Glisson, W. B., Andel, T. R. & Gu, Y. X. (2017). Enhanced operating system protection to support digital forensic investigations , 2017 IEEE Trustcom/BigDataSE/ICESS , publisher IEEE.

Mercedes-Benz. (2026). Mercedes-Benz AG - FOSS Manifesto [online] Available at: https://opensource.mercedes-benz.com/manifesto/?utm_medium=referral&utm_source=github

Millidge, S. (2018). Benefits of Open Source vs. Proprietary Software , DZone. [online] Available at: https://dzone.com/articles/benefits-of-open-source-vs-proprietary-software

Nguyen, T., Mcdonald, J. T., Glisson, W. B. & Andel, T. R. (2020). Detecting repackaged android applications using perceptual hashing , Proceedings of the 53rd Hawaii International Conference on System Sciences , Conference Proceedings.

Ohm, M., Plate, H., Sykosch, A. & Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks , Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17 , Springer.

Ponta, s. E., plate, h. & sabetta, a. (2018). Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software , 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) , IEEE.

Russell, r., kim, l., hamilton, l., lazovich, t., harer, j., ozdemir, o., ellingwood, p. & mcconley, m. (2018). Automated Vulnerability Detection in Source Code Using Deep Representation Learning.

SOLUTIONSHUB (2023). The State of Open Source Software: Current Trends, Future Outlook, Benefits, and Challenges. [online] SolutionsHub Available at: <https://solutionshub.epam.com/blog/post/the-state-of-open-source>

Synopsys (2023). Open Source Security and Risk Analysis Report. [online] Available at: https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report/

Xu, r., tang, z., ye, g., wang, h., ke, x., fang, d. & wang, z. (2022). Detecting code vulnerabilities by learning from large-scale open source repositories. *Journal of Information Security and Applications,* 69**,** 103293 , ISSN 2214-2126.