STUDY ON THE USAGE & EFFICACY OF NOVEL DEEP LEARNING TECHNIQUES TO AUTOMATE SECURITY ALERT DISMISSAL REVIEW PROCESS IN APPLICATION SECURITY

by

Devjyoti Raha, MSc

DISSERTATION

Presented to the Swiss School of Business and Management Geneva

In Partial Fulfillment

Of the Requirements

For the Degree

DOCTOR OF BUSINESS ADMINISTRATION

SWISS SCHOOL OF BUSINESS AND MANAGEMENT GENEVA

November, 2025

STUDY ON THE USAGE & EFFICACY OF NOVEL DEEP LEARNING TECHNIQUES TO AUTOMATE SECURITY ALERT DISMISSAL REVIEW PROCESS IN APPLICATION SECURITY

by

Devjyoti Raha

Supervised by

Dr. Monika Singh

APPROVED BY

Sasa Petar

Prof.dr.sc. Saša Petar, Ph.D., Dissertation chair

RECEIVED/APPROVED BY:

Renee Goldstein Osmic

Admissions Director

Dedication

To my dearest wife Mrs Shreyashi Mazumder, my lovely daughter Devanshi, my son Aryajyoti & my mom and dad.

Acknowledgements

To the most important person in my life - my life companion, my wife Mrs Shreyashi Mazumder, who has always supported me for anything during my life journey. Her immense endeavor in taking care of our family, our baby, our life had given me the option to follow my research and complete my thesis. By the Grace of God, we are blessed with two of the most treasured gifts in my life – our daughter Devanshi and our newest addition son Aryajyoti. As I write this, my wife is pregnant with our second baby and this dissertation is dedicated to two of you. I am not sure if you can understand why I did this degree and what I wrote in the dissertation but nonetheless we would always have our bedtime story times ahead.

To my wonderful mom and dad – Mrs Champa and Mr Dilip Raha, who provided me the constant encouragement, working all life to educate me, nurture me and allowed me to become what I am today.

To my dearest sister – Mrs Puja Raha, and brother-in-law – Mr Souvik Roy Chowdhuri, who both have always been a constant support and always encouraging me to explore ahead.

Special thanks to Dr. Monika Singh for being an integral part of my educational journey. Thank you so much for your guidance and constant feedbacks to make this dissertation completed successfully.

Last but not least, I must convey my thanks to my company to lend a supporting hand, gaining tremendous insights on the work and numerous discussions that transpired. Without their sincere support, I could have never finished the research work.

Thank you all again!

ABSTRACT

STUDY ON THE USAGE & EFFICACY OF NOVEL DEEP LEARNING TECHNIQUES TO AUTOMATE SECURITY ALERT DISMISSAL REVIEW PROCESS IN APPLICATION SECURITY

Devjyoti Raha
2025

Dissertation Chair:
Co-Chair:

Majority of the security issues that organization face today, such as security breaches, data exfiltration, hacktivist campaigns, DoS are a direct result of exploitability of vulnerabilities in application source code/hosted environment/libraries/exposed secrets. Therefore, there is an inherent need to understand the space and design solutions that would prevent such exploits to occur in first place assisting software engineers and application security experts likewise. Therefore, automation is an important piece that can provide early detection and thereby remediation for these problems.

Modern devsecops practice in application security space promotes flaw detection across application using a standard combination of tools like SAST, DAST/IAST, SCA and Secret Scanning. Depending on their configurations, these tools in combination can provide a robust application security framework to prevent vulnerabilities creeping into applications by creating alerts through scanning and detection. These alerts can be categorized based on their severity on CWE/CVE scores and rated as ‘critical’, ‘high’, moderate’, ‘low’ and ‘informational’. But, the tools will be efficient as long as their output is processed properly. Governance is that mechanism through which policies are enforced on the output developed by these tools. Now, based on the risk appetite and

tooling result quality – the policies are usually not applicable on all alerts that is being created. The developers are enforced through policies to mitigate sever ‘critical’ alerts and leave other categories such as ‘high’ alerts ignored or closed without doing due diligence. With the volume of alerts produced coupled with False Positives, it is hard for ssdlc analysts or developers to go through all relevant alerts. However, this also creates a risk as some of the improperly closed alerts might have flaws that can be exploited causing financial/reputation loss especially for Fintech organizations.

Recently there has been significant improvement in ML/deep learning space using feedback mechanism. This paper would study that deep learning technique such as GNN could be leveraged to plug the gap in appsec tool alert remediation space, and can provide a way to automate process to mitigate improper closure of alerts and enable appsec engineers to train such model based on feedback using JSON dataset and eventually propose a framework to implement such.

LIST OF ABBREVIATIONS

- Appsec – Application Security
- Fintech – Financial Technology
- SAST – Static application security testing
- DAST – Dynamic application security testing
- IAST – Interactive application security testing
- CI/CD – Continuous integration/continuous development
- Devsecops – Development + security + operations
- App – Application
- Api – Application programming interface
- AWS – Amazon web services
- SSDLC – Secure software development life cycle
- ML – Machine learning
- NN – Neural network
- GNN- Graphical Neural Network
- TP – True Positive
- FP – False Positive
- SVM – State vector machine
- SCA – Software Composition analysis
- Pentest – Penetration testing
- Infosec – Information security
- PRA – Prerelease assessment
- CWE - Common Weakness Enumeration
- CVE - Common Vulnerabilities and Exposures
- CVSS - Common Vulnerability Scoring System
- CIA – Confidentiality Integrity Availability
- VA – Vulnerability Assessment
- VD – Vulnerability Detection
- DoS – Denial of Service
- OWASP – Open Worldwide Application Security Project
- App – Application
- Pentest – Penetration testing
- SOC – Security Operation Center
- GHAS – Github Advanced Security
- SS – Secret Scanning
- SCM – Source Code Management
- JSON – Javascript Object Notation

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I:
INTRODUCTION

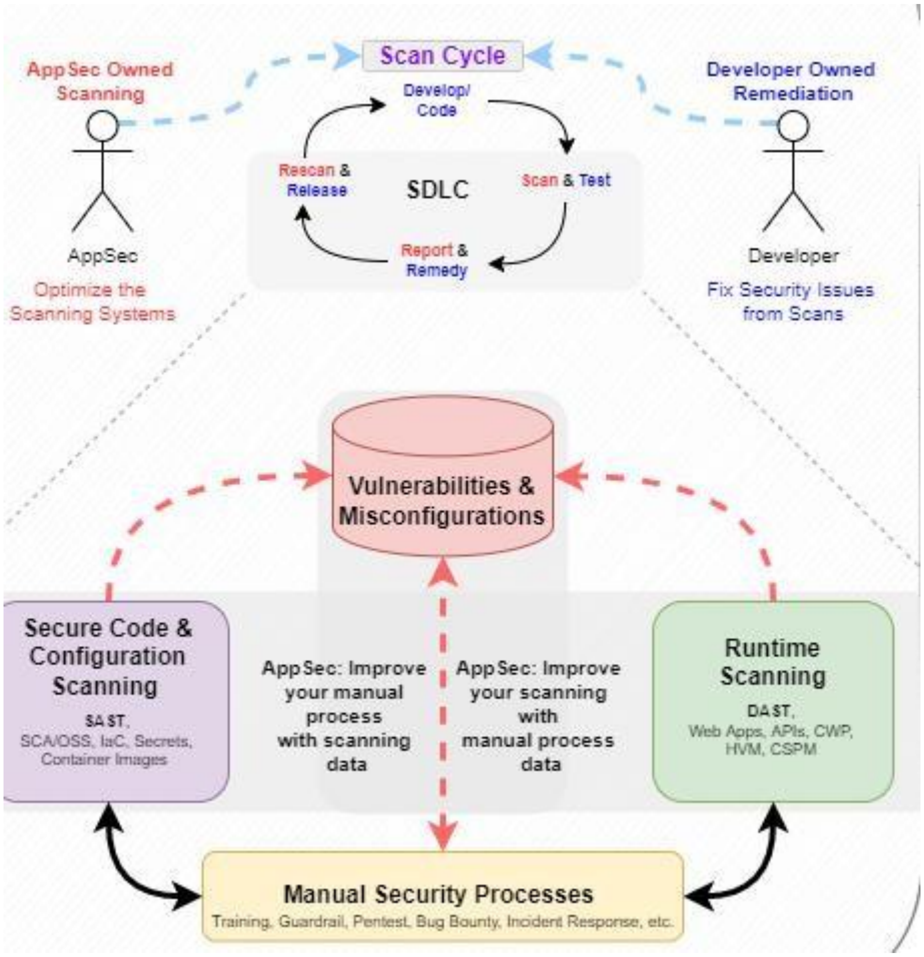
1.1 Introduction

Cybersecurity landscape is an ever-changing area and for any industry to keep, up-to-date with all the exploits is really challenging. A domain in this space is application security (appsec) and implementation of proper application security controls is implemented as best practices across any tech enabled industry with one or more combination of tools such as Static Application Security Tools (SAST), Dynamic Application Security Tools (DAST), Secret Scanning Tools or more recently Interactive Application Security Tools (IAST). The tools are plugged in to the CI/CD pipeline as standard devsecops practice to detect and fail on vulnerabilities and to minimize or mitigate the risks associated with attack surfaces in applications which is of utmost importance to any organization to keep their cybersecurity posture intact as almost all the organizations offers their services via web app or mobile app. Static analysis tools are used to detect bugs/ defects/ vulnerabilities with in the code (Pär Emanuelsson et al., 2008). Dynamic application security tools looks for vulnerabilities from black box way, does attack surface web application analysis whereas secret scanning looks for hard coded credentials or api tokens/secrets etc. Secrets that are checked in version-controlled source code repositories pose security risks to software and organization alike. Secret detection - when properly configured tools does identify the existence of secrets in the code, commit changesets, and project version control history. And these tools generates a lot of false positives, developers are provided with mechanisms to bypass the warnings generated from these tools (Rahman *et al., 2022*). Any vulnerabilities missed or falsely categorized opens risks to the organization that can have huge implication not only on financial aspect but on reputation stake as well. This can severely affect the sustainability of the business as we have seen in the cases of companies such as Code spaces, Telefonica, Medstar Health etc or can pose serious risks. Fintech organization are always a hot spot for cyber criminals to exploit vulnerabilities and as such is well regulated to handle risks strategically (Artie et al., 2017).

According to a leading article posted by securityboulevard.com, "The 2020 Verizon Data Breach Investigations Report (DBIR) confirms that this is the case: 43% of data breaches are tied to web application vulnerabilities—which more than doubled year over year."(Patrick 2020 & Verizon 2020) This means almost majority of the exploits occurs at application layer, which ultimately leads to the question of how to harden security controls and mitigation practice to minimize the risks across fintech organizations. Although these tools produce a good set of results but it is accompanies by a high level of False Positives (Bushra Aloraini et al., 2019).

Modern devsecops practices follows the implementation of gating on policy conditions with results from one or more application security controls combined as mentioned above with a combination of SAST+DAST/IAST tools with secret scanning for hard coded creds such as api tokens, hard-coded credentials, cloud formation template secrets from AWS services, Google Cloud platform, Azure etc. The alerts are based on severity are often classified as *'Critical', 'High', 'Moderate', 'Low' or 'Informational'*.

The governance gating (application security control to ensure quality code with *Zero* policy failing issues is passed to next stage in pipeline) based on policy happens usually in application’s own pipeline. Tool such as Enterprise Jenkins, JekinsCore, Github workflow, AWS CodeDeploy, github workflow etc are used to manage code deployment in production stage and the policy is usually set to conditions to fail the pipeline on ‘Critical’ + Organization specific (based on risk posture) alerts that gets generated from application security tools. A standard appsec flow in shown below:



Scan Cycle
AppSec Owned Scanning
Developer Owned Remediation
Develop/ Code
Scan & Test
Report & Remedy
Rescan & Release
SDLC
AppSec
Optimize the Scanning Systems
Developer
Fix Security Issues from Scans
Vulnerabilities & Misconfigurations
Secure Code & Configuration Scanning
SAST, SCA/OSS, IaC, Secrets, Container Images
AppSec: Improve your manual process with scanning data
AppSec: Improve your scanning with manual process data
Runtime Scanning
DAST, Web Apps, APIs, CWP, HVM, CSPM
Manual Security Processes
Training, Guardrail, Pentest, Bug Bounty, Incident Response, etc.

Figure 1.1 Appsec flow (James Chiappetta, Better appsec 04/2022)

As a part mitigation process, the developers can resolve their ‘Critical’ or Policy specific alerts on which they are gated by fixing them or marking them as ‘False Positive’. This approach applies to almost all the application security tools that is integrated to developer’s pipeline. Since, any ‘Very High’ or ‘Critical’ alerts can be avoided by marking as ‘False Positive’, the onus falls upon the application security team to verify those to avoid vulnerabilities introduced to applications. The application security team may take a sample of ‘Critical Alerts’ that was marked as ‘False Positive’ to verify them manually and this consumes a good chunk time. Not to mention, it is almost impossible to verify manually all the ‘Critical/High’ alerts that is generated by the tools. And, this opens up the potential to miss a vulnerability that was wrongly marked as ‘False Positive’ by a developer but correctly identified by the appsec tool and is not a part of sample used to verify the mitigated alerts by the appsec team. Though there are organizations who has dedicated team to manually verify the policy failing alerts and again which consumes a good chunk of time in review process.

This is the problem area that this paper will explore and will provide Fintech organization to address and adopt to better approach that strengthens the application security controls and mitigation practices. This paper will also propose a framework that can be a part of existing mitigation practices without adding any new tools.

1.2 Research Problem

The importance of exploring an idea of leveraging artificial Neural Network concepts to develop an automated process for application security's vulnerabilities mitigation is vital and more so in today's fast paced agile scenario in shift left devsecops model. The fact that application security landscape has to pivot around devsecops practice without hindering developer's agile workflow is becoming more and more challenging. The pain point here is as we shift left, developers would want to shift more right and they are justified in doing so because of increasingly fast develop & deploy workflow with its gain and benefits. Alternatively, we can't overlook the importance of appsec tool controls as they provide an in depth risk analysis for applications, source codes, secrets, vulnerabilities disclosure and software composition analysis. This, in turn, helps security teams to push developer's to follow secure coding practices, fix vulnerabilities, mitigate security loopholes, manage secrets, perform dynamic assessments and be compliant with respect to an organization's application security baseline. Below is a sample flow where appsec ops team are seen always pushing left.

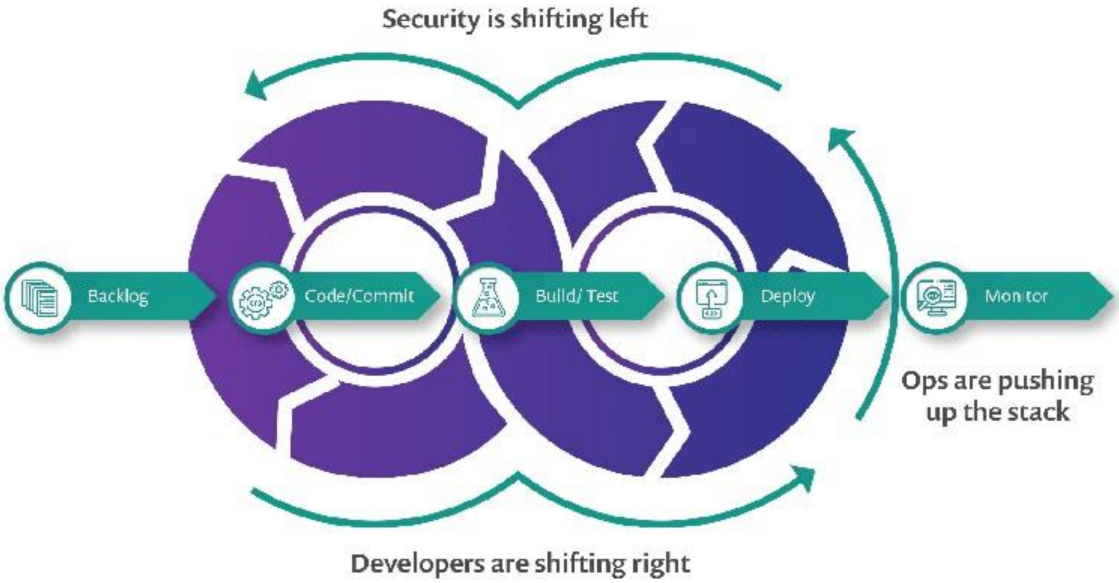

Security is shifting left
Backlog
Code/Commit
Build/ Test
Deploy
Monitor
Ops are pushing up the stack
Developers are shifting right

Figure 1.2 Sample Flow in CI/CD (Snyk, 2023)

The key here is strike a fine balance between appsec controls plugged in CI/CD pipelines and governed through gates. Manual appsec analysis on the results generated from appsec tools takes a long time for review and risk mitigation process and hence hinders developers agile flow and impacts deployment timelines. This can have a negative business impact and deters developer workflow. On the other hand, automating flaw review process opens the risk of vulnerability introduction in applications which can have a disastrous impact if any vulnerabilities gets exploited.

As explored in the literature survey, deep learning techniques can provide a better performance in automating appsec risk mitigation processes. In this paper, we will try to demonstrate effectiveness of deep learning techniques in post-alert dismissal review process generated from appsec controls and explore an alternate approach to establish a framework to improve the existing mitigation process. This is in contrast to the traditional manual mitigation approach in vulnerability analysis (Moon et al., 2012) and (Varun M Deshpande et al., 2017).

There are typically 3 standard deep learning models: CNN (Convolution Neural Network), DNN (Deep Neural Network) and GNN (Graph+CNN).Each of the models have their advantages and disadvantages, with our primary datasets mostly would be JSON based. Existing appsec tool controls such as SAST/DAST/Secret managements/SCA/IAST performs their analysis and produces results in JSON format in majority of the scenarios. Mitigation approach usually comprises of manual or semi manual which as discussed above can have the possibility of missing of a flaw being fixed and hence a chance to being exploited in real life and caused damages to related stakeholders and seriously harm credibility of the enterprise (Hossain Shahriar et al., 2023).

In summary, there is a need for a better understanding of constraints in utilizing deep learning techniques in appsec tool's mitigation process in SSDLC cycle, construction and a structured approach in identifying and modeling constraints to ensure a constraint-free work plan. So, we can gauge the efficacy of utilizing artificial neural network models in appsec's mitigation process throughout SSDLC lifecycle. More specifically, the following research questions need to be addressed:

1. How to promote automation so a potential vulnerability is not missed due to wrongly marked and dismissed in appsec flaw mitigation or manual developer action?

2. What are the ways to implement neural network models in to automate mitigation process for alerts generated by appsec tools ?

3. How to reduce risks associated with manual vulnerability mitigation and why automation using AI is a need?

4. What are the current industry practice/framework as well as research advancements in reduction of risk in application security in vulnerability mitigation?

5. Bring more in-depth understanding on appsec tools, analysis on unified tools usage as SAST+DAST/IAST+Secret Scanning - uses and risks associated with reported vulnerabilities for each controls & the need to develop a framework to incorporate these tools and develop a holistic approach to automate flaw mitigation to reduce risks thus preventing vulnerability exploitation.

1.3 Purpose of Research

Overall, the long term goal of the research is to develop a premise which would provide facts based guidance to show if deep learning models such as artificial neural networks can be leveraged promote automation in flaw/vulnerability mitigation generated by appsec controls/tools in standard SSDLC process. Additionally, we would measure the efficacy of such models in and its usage in application security domain. Based on collected metrics and favorable result we would also formulate a workflow which would include this SSDLC process. The objective of the current study is to provide a comprehensive review of literatures and industry practices in relation to vulnerability remediation or flaw mitigation analysis and outline the efficacy of implementing deep learning models in appsec's mitigation framework. Additionally, propose a conceptual framework for automated vulnerability remediation process. Particularly, the study has the following sub-objectives:

1. To provide a comprehensive review of sources and analysis done on relevant works in application security flaw mitigation or vulnerability remediation;

2. To develop a model to show implementation of deep learning techniques and its utilization in appsec's mitigation process and measure its efficacy comparing to manual approach;

3. To review current industry practices and researches in regards to flaw mitigation process in application security;

4. To outline a conceptual framework for automated vulnerability remediation process.

The result of this study will be valuable to the application security industry practitioners as well as related valuable insights in integrating deep learning models for mitigation process to promote automation and reduce risk from missed vulnerability exploitation. Also, draw out the risks associated with developer's based mitigation and manual mitigation.

1.4 Significance of the Study

A preliminary literature review shows that several research has been conducted on related context but not exactly on standardization of mitigation approach of False Positives. These studies were primarily focused on either improving the SAST/DAST tool capability or do vulnerability analysis using ML techniques. Almost all past papers tries to address the tool specific issues rather than contemplate on application security false positive mitigation process issue.

Preliminary objectives would include:

- To explore previous related work on automated detection of ‘relevant’ security alerts using ML/deep learning techniques.
- To explore existing approaches to cover the risks in gaps in appsec methodologies.
- To explore already designated frameworks to analyze the gaps in secure software development lifecycle and devsecops practices.
- To establish a gap exists in remediation practices of relevant security alerts by automated application security controls.
- To explore the possible existing framework to address the gap from an organization context.

For instance, let’s take an example in one of the paper it was proposed to create a novel algorithm to utilize Machine Learning techniques but it solely relies on static analysis tools. They worked on to improve the accuracy of True Positive detection and this assist developers/analysts alike (Cheirdari et al., 2018). In Project Achillies, a prototype tool was developed for Static code analysis which did a vulnerability detection of Java source code using Recurrent Neural Network. The testing was achieved using 44,495 test cases which indicates that the tool can achieve an accuracy higher than 90% for 24 out of 29 different types of CWE vulnerabilities (N. Saccente et al., 2019). These experiments/analysis further testifies that overall framework utilizing neural network concept is not a farfetched idea and can be leveraged with a well-defined framework.

Deep learning techniques provides a novel way to reduce the number of False Positives in findings provided by the appsec security tools (SAST/DAST/Secret scanning). Previous study has been done utilizing well-known cyber datasets and efficiency measured for each by comparing deep learning models including recurrent neural networks, deep neural networks, restricted Boltzmann machines, deep belief networks, convolutional neural networks, deep Boltzmann machines, and deep autoencoders(Mohamed Amine Ferrag et al., 2020). Much of the work is also done in the area of IDS to detect false Positives to investigate the cost ratio of false negative errors to false positive errors employing neural network models (Daejoon Joo et al., 2003).

A more recent paper demonstrated how Machine learning techniques can be leveraged for intrusion detection (P. Mishra et al., 2019); although here we are dealing with the aspect of alerts generated by application security tools, the underlying detection mechanism to use algorithms remains same. In the paper “Automated software vulnerability detection with machine learning” (Jacob et al, 2018) investigated utilizing the novel machine learning technique to be used in static scanning tools to measure their capability to detect vulnerabilities (Jacob A et al., 2018). Some more advanced work has been done using deep neural network concepts for malware detection where it was shown a 95% detection rate at 0.1% false positive rate (FPR), based on more than 400,000 software binaries sourced directly from our customers and internal malware databases (J. Saxe et al., 2015). This is encouraging as detection of malware is based on signatures and heuristics seems far more complicated than findings generated by application security tools and a similar approach can be leveraged to cross validate the findings that are marked as 'False Positive' by developers aside. Even novel static taint analysis employed the capability for deep learning techniques for vulnerability detection (Weina Niu et al., 2020). In another paper, Roland & Dominic et al. assessed the detection and assessment performance of several common SAST tools and SVP (Software Vulnerability Prediction) models on a variety of vulnerability datasets and provided observations that unification of both approaches can provide a better performance in terms of less False Positive rates (Roland Croft et al., 2021).

There has been a good amount of papers published in utilizing machine learning for vulnerability detection and classification as well. With more computing capability, it is evident that usage of machine learning/ neural networks will be used in lengthy SSDLC practices. Tiago and Nuno in their paper developed a method to identify specific types of vulnerabilities as FastScan was not able to identify the vulnerability type. Their initial results are really promising with 93% resulting in a precision of 90% and accuracy of 85% (Baptista Tiago et al., 2021). There are other papers which investigates the static analysis of False Positives and False negatives with IDS/IPS (Intrusion Detection system/Intrusion Prevention System) which shows the FP percentage (Ho, Y. -C. Lai et al., 2012). Saha & Denning et al. did a considerable work on secret scanning and they proved that a combination factor of regex & machine learning reduces the number of False Positives in secret detection. They used a Voting Classifier (combination of Logistic Regression, Naïve Bayes and SVM) to reduce the number of false positives considerably (A. Saha et al., 2020). On a similar note, we are concerned on marking of False Positive or dismissal of a True Positive of an alert generated by SAST/DAST/Secret Scanning tools. A very relatable work has been done in the area of developing a classifier based on novel learning process to identify points that cause a given SCA(Static Code Analysis) tool to produce false error reports, and then use that identifier to detect the likelihood of new report generating false Positives (Ugur Koc et al., 2017) .

Overall, there has been good researched paper on false Positive detection approach in independent tools in cybersecurity. There has been usage of data mining techniques as well to automate secure coding practices (Ibéria Medeiros et al., 2014). Also, there has been amount of work done on specific language for example false positive analysis in static scanning for Javascript web applications where SAFE, the state-of-the-art JavaScript static analyzer with bug detection was used and then identified the source of False Postives (Joonyoung Park et al., 2016). In a separate paper, DEKANT - a static analysis tool was developed which learns and detects web application vulnerabilities and though we are more concerned on the mitigation part, the analysis, or learning and detection via machine learning algorithms remains similar (Ibéria Medeiros et al., 2016).

1.5 Research Purpose and Questions

The overall aim of the research is to study the possibility of using deep learning technique such as neural network to address the gap that exists in the remediation of application security tool scanning alerts that are improperly closed. The study then is to check out if the trained model would then allow the option to application security engineers to triage the relevant alerts and take preventive measures to address them. Additionally, the study will also explore to propose an alternate framework in secure software development lifecycle to address this remediation of improperly closed relevant security alerts.

➢ Is deep learning technique such as graph neural network feasible enough to be used in application security software secure development lifecycle remediation space?

➢ Can the neural network then be utlised to detect incorrectly closed security alerts and inform application security team to triage?

➢ Can we further design and develop a framework to plug the above process in existing devsecops process?

CHAPTER II:

REVIEW OF LITERATURE

2.1 Current Position

Internet based applications, services, mobile apps have become the norm of any business operating today. With evolving technology, standard ways of operating business has changed drastically (Carolina et al., 2014). Almost all organizations across the world has their footprint in internet and there is obviously a good reason to do this. Connectivity via internet provides easy to access services, on demand communication, access to information, ease of operating business, automation processes, promotion, and ideation of new concepts such as electronic business (e-business) and electronic commerce (e-commerce) – not to mention this is extremely profitable and scalable in terms of operating businesses (Elena-Iulia et al., 2014). However, with new technologies comes newer challenges and with internet being open and each businesses having multiple web applications/mobile apps/services operating across web – this introduces risks and challenges of a different nature.

Enter the era of plethora of programming languages to develop different types of applications – web/mobile/api/services etc. With a multitude of options for clients across web to interact with a business, there comes a risk in the form of cyber-crime (Wall D., 2007). Now days, the heart of any business is their data – in simplest form data can be business sensitive, customer centric, personal info, financial specific, confidential to clients etc. To access these data illegally by exploiting vulnerabilities in web/mobile/api/services over web can be very rewarding for malicious actor and in some other cases, cyber-attacks can have military or political purposes (Yuchong Li et al., 2021). The more an organization having web applications, mobile applications, api services, the greater is their attack surface. In addition, as such it has become imperative that every organization have their application security model in place and have a robust policy to enforce controls around various points. The role of national educators, policymakers, and multilateral organisations in achieving a comprehensive policy to tackle cyber attacks needs to be outlined and adopted (Niki et al., 2022).

Why application security?

According to Veracode's State of Software Security volume 11 2023 report, Over 74% of applications have at least one security flaw found in the last scan over the last 12 months. Many had much more, as their research found a total of 86 million new findings, and 3.7 milion dynamic findings which would impact 71% of all applications (Veracode Software Security volume 11, 2023). Not all of those flaws presents a significant security risk, but the sheer number is troubling. The below images shows the vulnerabilities exploited categorised by attack type :

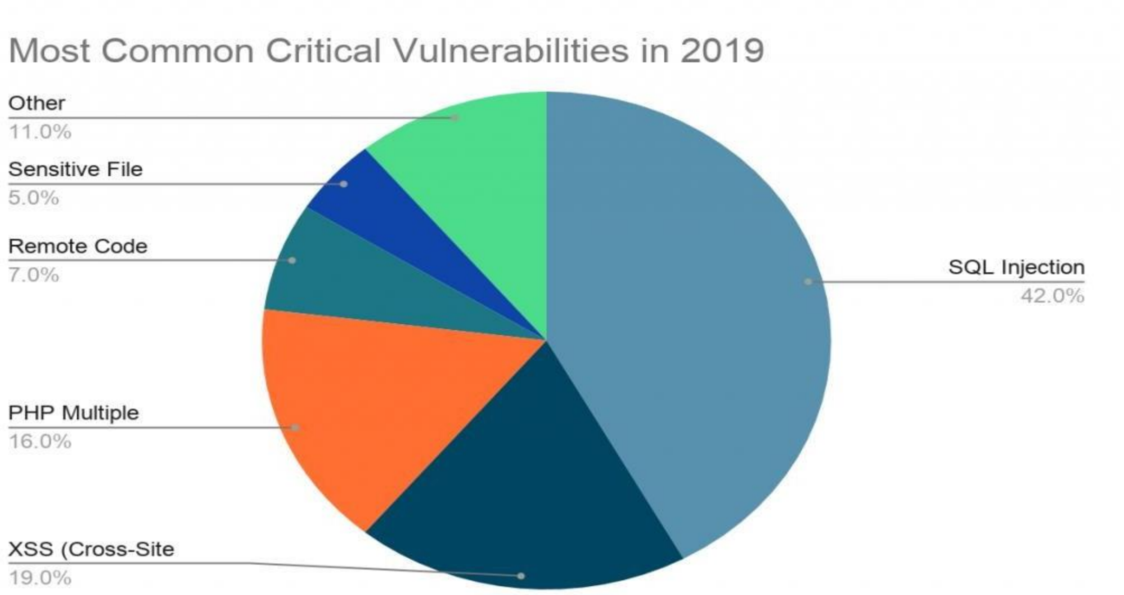


Fig 2.1 Most Common Vulnerabilities (socradar, 2021)

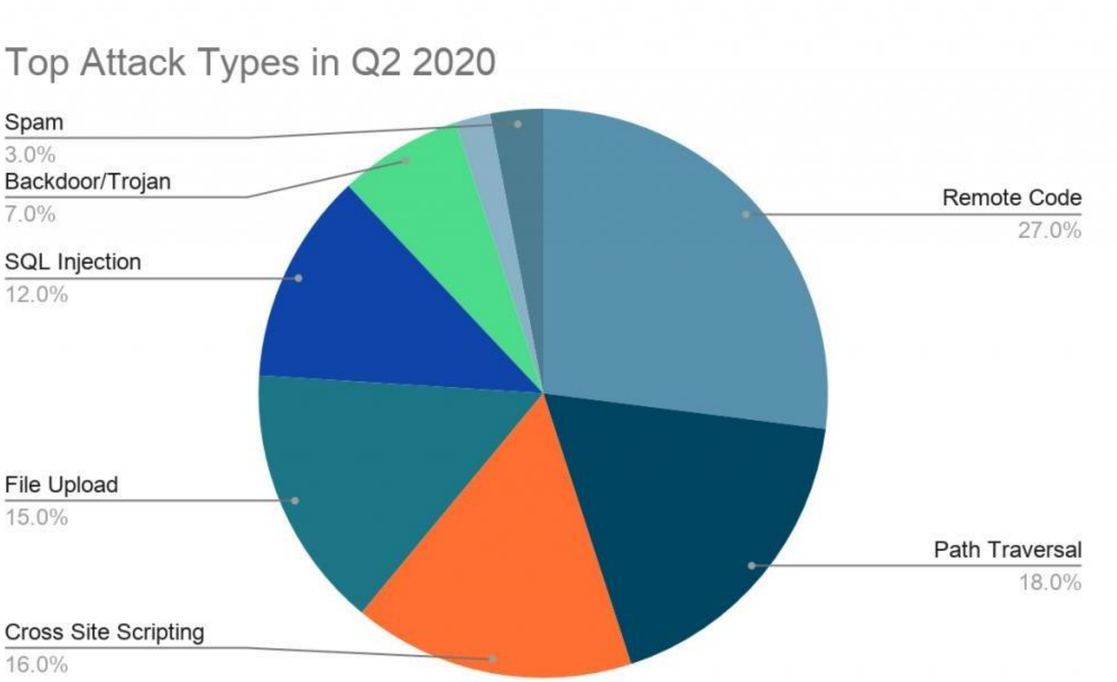


Fig 2.2 Top Attack Types (socradar, 2021)

The faster and sooner in the software development (sdlc) process a developer can find and fix security issues, the safer an enterprise will be. Because everyone makes mistakes, the challenge is to find those mistakes in a timely fashion and remediate them. In a standard agile devsecops methodology, the application security controls are a combination of multiple tools performing scanning services on the source code or web application or api endpoint or websites. The below Figure 4 shows a standard template for appsec toolkit where several controls are used like SAST, DAST, secret scanning, SCA etc.

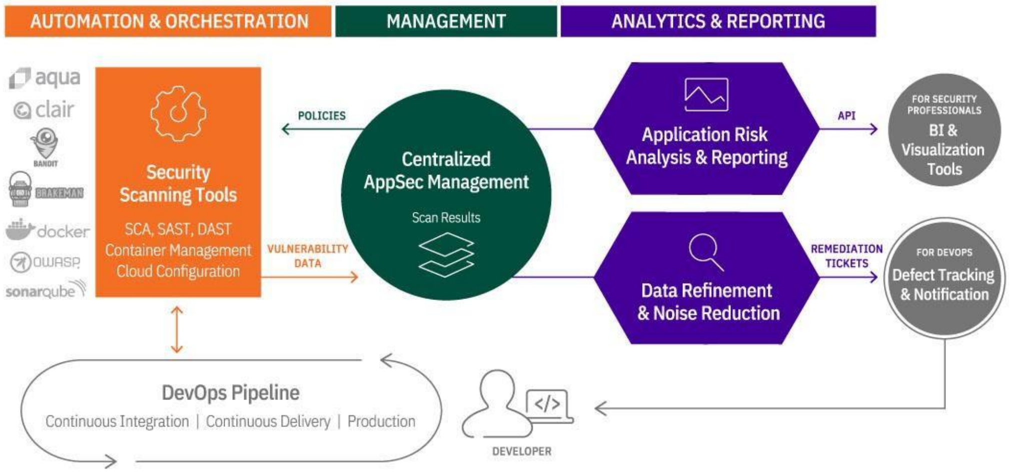


Fig 2.3 Sample Appsec Process (Joanne Godfrey, 2021)

DevSecOps is the practice that works on developing and integrating modernized security methods that can keep up with DevOps in a bid to shift left smartly by adhering best security practices and finding security vulnerabilities early in the development process and remediating them (Myrbakken H. et al., 2017). All these security tools in spite of producing good results, they also have False Positives. And, these large number of policy failing alerts has to be either reviewed by SSDLC or PRA or any security review teams or these are self-mitigated by developers (Varun M Deshpande et al., page 194, 2017). Question comes on the self-mitigation process as a single improperly mitigated alert can have an open vulnerability, which increases risk. In addition, an organization with hundreds of application/products - manual mitigation approach can be cumbersome and inefficient.

In modern times, advances in machine learning methods for structured data have great potential in software vulnerabilities and discovery (Seyed Mohammad Ghaffarian et al., 2018). By studying different advance machine learning algorithms such as

MLP/CNN/RNN/GNN and measuring their efficiency, we can better understand their effectiveness on this scenario to automate re-opening of critical alerts and there by open a review process to application security teams for further investigation.

The literature review will discuss the devsecops process and in particular the mitigation approach of security alerts generated by application security tools followed by most enterprises, corresponding tools involved and the data they produce in alerts, sdlc lifecycle with CI/CD pipeline, SSDLC/PRA/security team's manual review process and their drawbacks, previous studies on adoption of machine learning algorithms in other cybersecurity use cases, how machine learning approach would fit and how this may impact the review process. This study would be beneficial to establish the usage and effectiveness of AI techniques in filling a gap in application security review process.

2.2 Application Security

Table 2.2

A study by Statista in 2022 shows that a quarter of US and UK organizations were highly concerned about data privacy law enforcement involving their company.

Concern	Low Level of Concern	High Level of Concern
Data breaches/cybersecurity	8%	35%
Litigation / regulatory enforcement action	11%	20%
Loss of customer loyalty/trust (if perceived not to be respecting their privacy)	13%	26%

Application security is the process of making apps more secure by finding, fixing, and enhancing the security of apps (David Strom, 2020) Security by design is secure by default. Many vulnerabilities are fixed at development phase by adopting a standard devsecops approach, integration of appsec tools plugged into the CI/CD pipeline to aid the developers in proper mitigation of these flaws. One of the standard agile flavour to this approach is embedded in devsecops practice. DevSecOps is the movement that works on developing and integrating modernized security methods that can keep up with DevOps (Myrbakken et al., 2017). We will not go in-depth in understanding the operations aspect of devsecops process rather get a flavour of it. Below is an image of the devsecops flow:

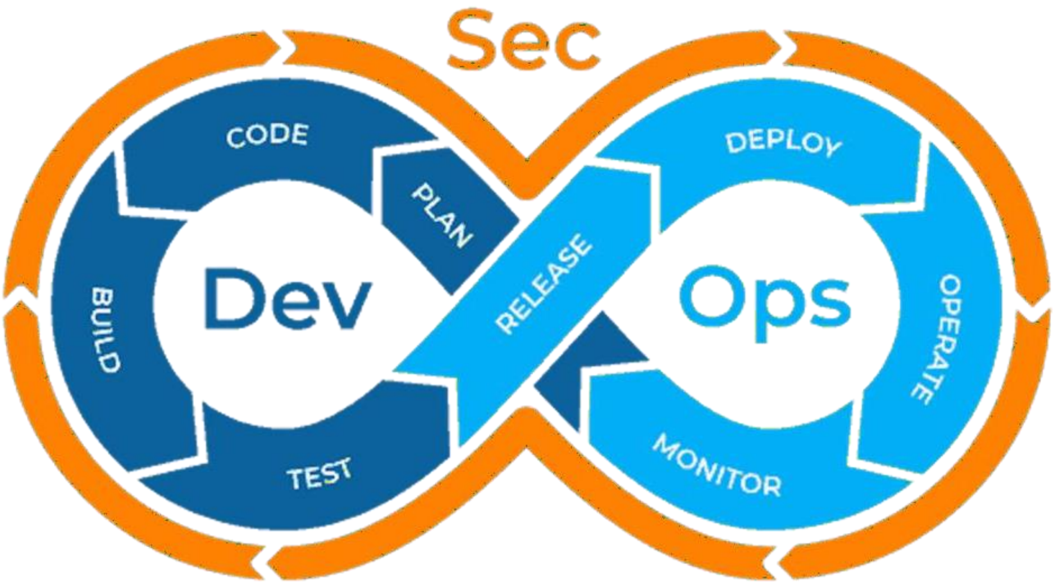


Figure 2.2.1 Sample DevSecOps Flow (Quintessence Anx, 2021)

The “Sec” part of DevSecOps model stands for incorporating security as early as possible into the agile model in Continuous Integration & Continuous Development flow. The concept of ‘Fail Fast’ is the key here where security flaws are found fast and is reviewed and remediated early. Previously in traditional DevOps model, there was always a gap in terms of the missing security requirements (Z. Ahmed et al., 2019). This led to vulnerabilities discovery at a very later stage of software development lifecycle (SDLC) which introduced many complexities in rework, leakage, test etc.

Security adaptive organizations generally have their own application security team to implement security control, enforce policies, perform review, and assist development team in issue remediation process. These policies are derived from the security standards and procedures laid down by management. A proper framework as such provides the backbone for continuous flaw findings and interacting with development teams to mitigate them. A framework can also assist organizations to identify potential challenges when implementing security standards, and determine if they are using their security resources effectively to address the challenges (Werlinger, R. et al., 2009).

In standard DevSecOps model, Secure Software Development Lifecycle (SSDLC) is embedded into CI/CD workflow. This is achieved by utilizing several application security controls integrated into each stage of DevOps pipeline thus providing automation (Fred B. Schneider, 2000). These security controls acts as checks as a part of preventive and corrective measure for introductions of vulnerabilities or flaws into software development. The security policies laid down by an organization’s leadership forms the heart and soul of these application security controls (João Henriques et al., 2022). These policies are usually derived based on business/compliance/security requirements (N.

Nagaratnam et al., 2005). Any application security control is only useful as long as it has executable policies adhered on to its assessment and the level of ‘enforceability’ is based on an organisation’s risk appetite. In addition, it should be kept in mind that the rigidity of enforceable policies should also not disrupt development work to so much that it becomes a blockade or a burden in a traditional Agile based software development lifecycle tracked via Jira.

Some statistics in application security in past year as shown below:

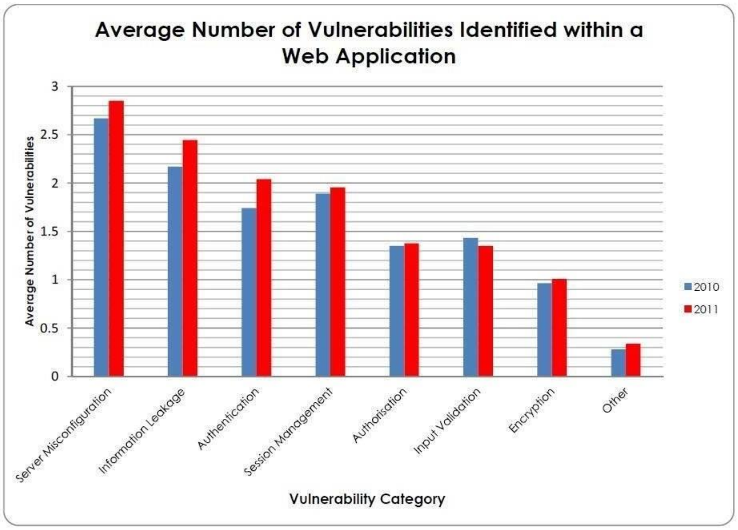


Figure 2.2.2 showing Avg Number of vulnerabilities Identified in a Web App

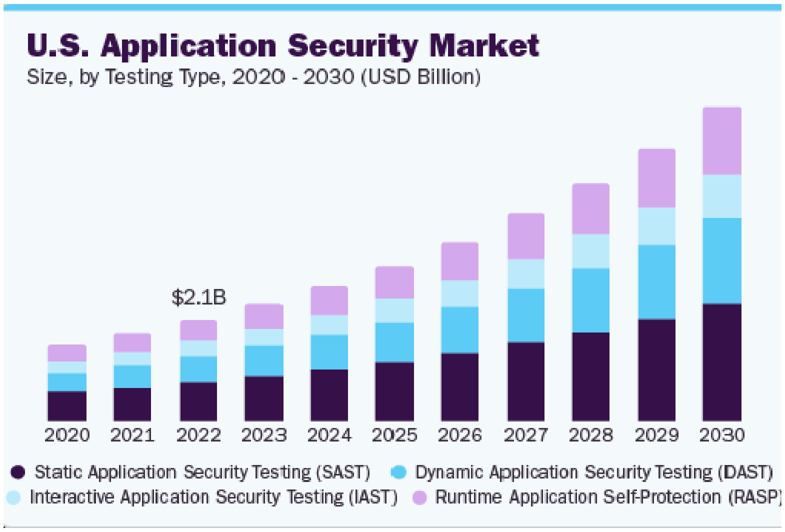


Figure 2.2.3 showing market utilizing appsec tools

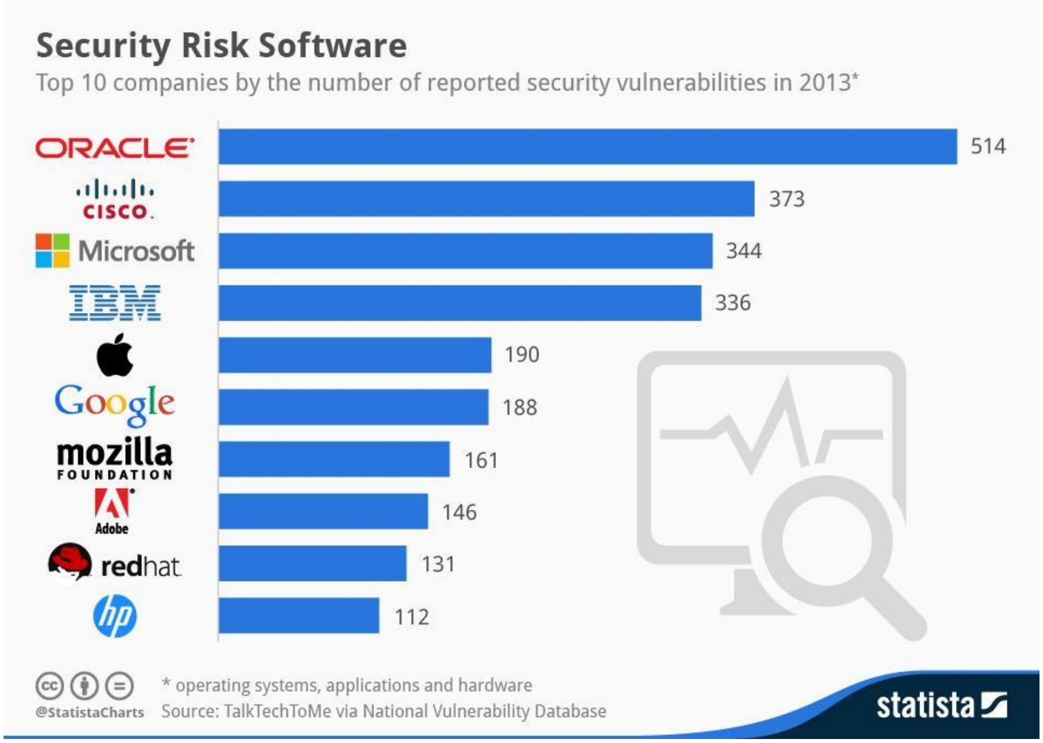


Figure 2.2.4 showing security risk tools

2.3 Application security Program and standard AppSec toolkit

The last decade has shown a tremendous growth in technologies. Hardly any business left that has not adopted digitization and so with digital adoption, the most coveted ‘My precious’ been data (Tolkien, 1991). In addition, one of the most usual way of data breach is done by exploiting vulnerabilities that exists in these applications or apps or their digital services, which are accessible via browser or in app (Yao-Wen Huang et al., 2005). Not to mention the entire application security practice is still evolving with emerging technologies.

According to the Open Web Application Security Project (OWASP), the below image shows the latest vulnerabilities that is in trend for 2023 (Isaiah Chua, 2023):

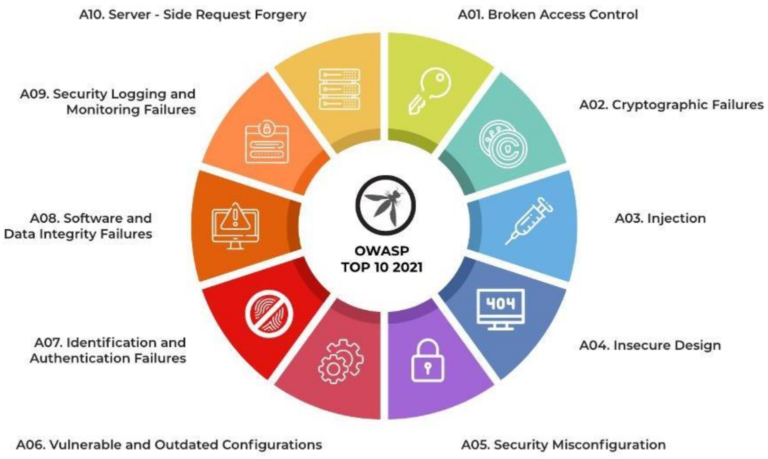

A10. Server - Side Request Forgery
A01. Broken Access Control
A09. Security Logging and Monitoring Failures
A02. Cryptographic Failures
A08. Software and Data Integrity Failures
OWASP TOP 10 2021
A03. Injection
404
A07. Identification and Authentication Failures
A04. Insecure Design
A06. Vulnerable and Outdated Configurations
A05. Security Misconfiguration

Figure 2.3 OWASP 2023 Top Vulnerabilities Chart (Isaiah Chua, 2023)

Describing the vulnerabilities is beyond the scope of this literature review but anybody interested can visit OWASP (www.OWASP.Org) for more info.

Therefore, is the most important goal of any organizations is to prevent the exploitation of these vulnerabilities and thereby data breach. This is precisely why an effective application security program is so much needed to strategize the implementation of security controls, provide checks, and review process. This is a vital due diligence activity all organizations have to partake to meet various obligations related to legal, compliance, governance, standards , business and more. The AppSec program usually comprises a blend of different teams such as Secure Software Development Lifecycle tools and Ops team, Pre- Release Assessment teams, Analysis and Vulnerability Detections team, Pentesting team, Audit and Risk team etc (NIST CSF, 2018). The name of the teams may vary but the function remains similar. SSDLC tools team provides application security tools integration and maintenance into CI/CD or DevOps pipeline where as the Ops team covers the tool adoption, code review, guidance, documentation. Vulnerability Detection team focusses more on appsec tools result and analysis whereas PRA team looks into flaw mitigation, gate checks, policies. In-house pentest teams usually provides manual Dynamic application security assessments and hence, time consuming and resource extensive. (ISO/IEC 27034:2011-2018, 2018) There can be other teams as well and it depends on an enterprise risk attitude and budget. Overall, most enterprises would derive their appsec program from NIST CSF and ISO IEC templates.

In order to implement the application security program, a proper collection of industry standard tools is required that would provide the appropriate control. Generally, this is a flavor of combination of Static Application Security Testing (SAST), Secret Scanning Services, Software Composition analysis (SCA) and Dynamic/Interactive Application

Security Testing (DAST/IAST). Handling web application or website security issues requires deep insight and planning, as there are several tools available, which may not be mature enough. Thus, fixating on the appropriate tools demands deep understanding and several steps, including analyzing the development environment, business needs, compliance and the web applications' complexity. (A. Alzahrani et al., 2017) Integrating these tools into CI/CD pipeline promotes automation and continuous assessments, thus producing lot of alerts some of which are critical and policy failing while others are false positives.

2.4 Appsec Tools Integration, Flaw Review and Mitigation Approach

In a traditional agile application development model, there will be a CI/CD pipeline setup, which would include all the steps from coding to deployment. Continuous Integration and Delivery (CI/CD) pipelines promotes the build process automation on dedicated nodes, and have been demonstrated to produce several advantages including early defect discovery, increased productivity, and faster release cycles. The effectiveness of such a setup may rely on the extent to which such pipelines are properly maintained to cope with the system and its underlying technology evolution, as well as to limit bad practices (F. Zampetti et al., 2021). This modular set up allows application security teams to plug in application security tools for continuous assessments and tracking which eventually allows SSDLC team to enforce policies based on these security assessments.

This pipeline setups can pretty complex but can be immensely configurable once implemented, thus rich rewards can be reaped. Below is an example as such:

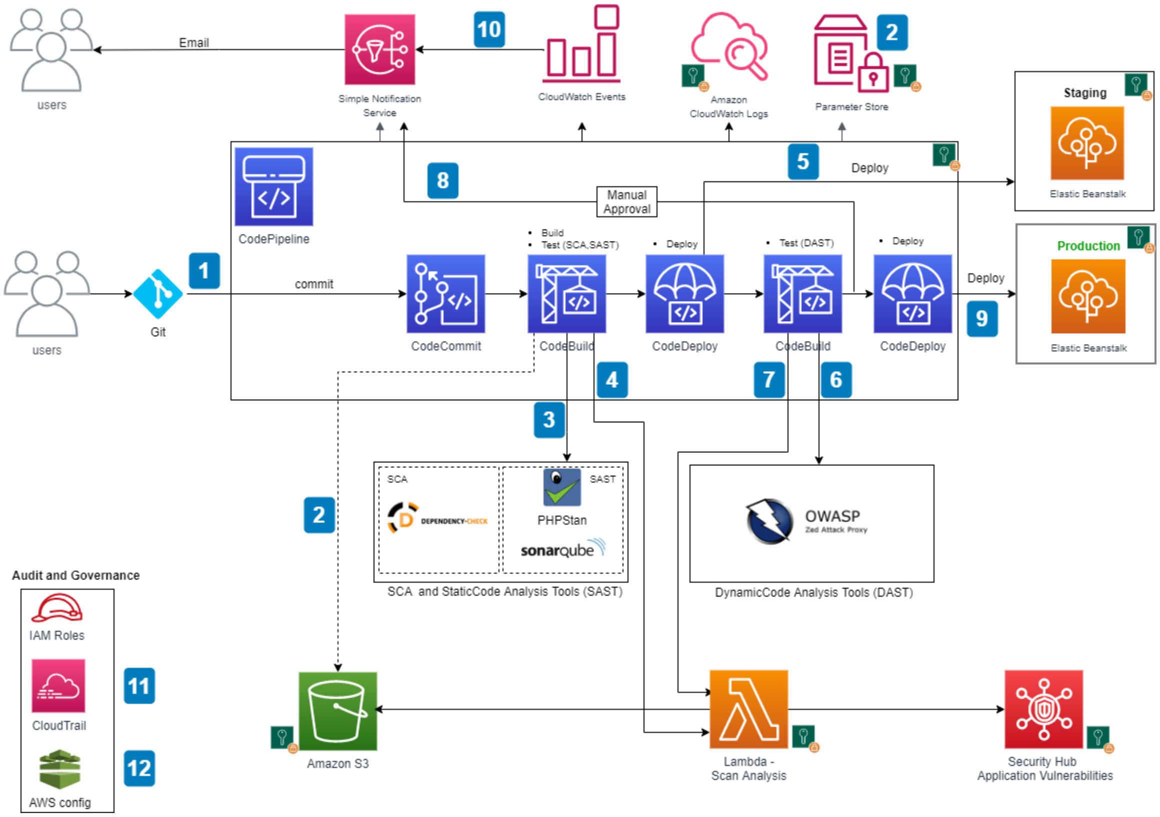


Figure 2.4.1 Sample DevSecOps CICD pipeline architecture (Srinivas Manepalli, 2021)

To make it easy, we will only consider on application security tools integrated here as SAST, Secret scanning, DAST/IAST. Consider a simple workflow: a developer commits and pushes the code into the repository (in above picture it’s Git), once the code is pushed it undergoes a passive secret scanning which produces secret flaw alerts. Next, through the pipeline a new job is triggered which itself has several stages. One of such stage would be SAST check where the code would be pulled from the repository and it will undergo SAST scan, then SCA then DAST (T. Rangnau et al., 2020). Now, based on policies defined by SSDLC, the results of all these security assessments would be evaluated and then the next stage would determine if the code build would be allowed to pass or fail. Usually, if it fails developers would have to fix the flaws and redo the entire steps again so the code build can be ultimately promoted to deployment without issues. Sample SSDLC flow would be:

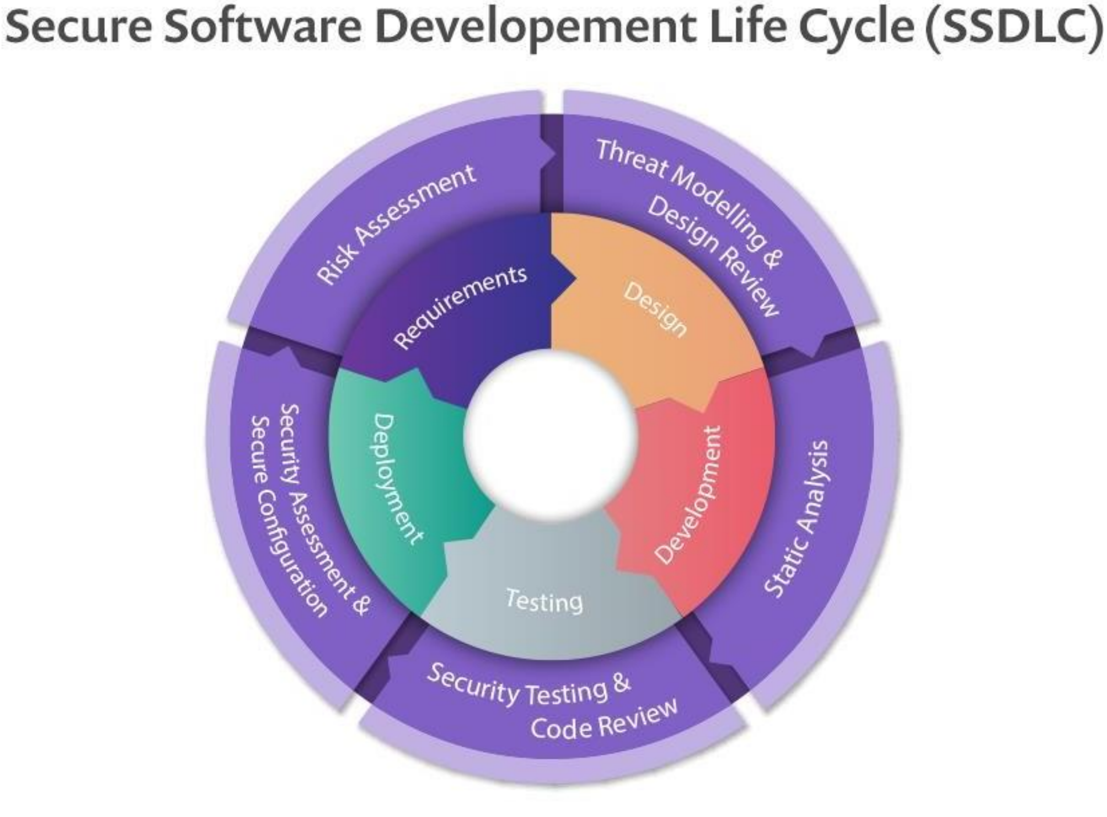

Secure Software Developement Life Cycle (SSDLC)
Risk Assessment
Threat Modelling & Design Review
Static Analysis
Security Testing & Code Review
Security Assessment & Secure Configuration
Requirements
Design
Development
Testing
Deployment

Figure 2.4.2 Sample SSDLC Flow (Snyk, 2023)

.

An alert can be Critical, High, Medium or Informative though there can be another attribute ‘Confidence’ which roughly indicates a tools capability to state the alert may be close to ‘True Positive’ (Nasif Imtiaz et al., 2019) An important point to note here is security policies are always defined on a subset of critical alerts, which are enforceable, leaving other alerts open to developer’s decision. Understandably, as mentioned before it is a tradeoff situation as it is counterproductive to have stringent security policies which enforces developers to review and fix everything which includes false positives (Elder S, 2022). However, on the other hand having a light security policy also increases risk by having open vulnerabilities in prod deployed apps which can be exploited (M. Agreindra Helmiawan et al., 2020). There are even enterprises where the entire assessment results are left to developers to figure out and decide what they should do which further increases responsibilities to developers and eventually can lead to ignorance of critical alerts.

Policy failing critical alerts or any alerts that are generated as a part of security assessments can be bought into review by SSDLC or PRA team. As mentioned before, they form the core part of AppSec Ops when it comes to flaw remediation and mitigation practice, policy enforcement, documentation etc. This would require manual review process, which can extremely time consuming with subsequent amount to skilled resources (Glaspie H.W et al., 2018). In the absence of policy enforcement, much of these critical alerts always goes un-reviewed marked as ‘False Positives’ by developers and thus open risks to exploitations. Many developers in practice tends to disregard the appropriate review and simply dismisses the alerts as ‘False Positive’ or takes a long time to triage and fix the vulnerabilities (N. Imtiaz et al., 2019). The reliance on ‘human’ factor here opens a subtle risk to any appsec program, even enterprises with policy enforcement relies only on a subset of critical alerts to formulate their policies where by other high alerts goes un reviewed and not mitigated. There seems to a gap in automating the review and mitigation process for the huge volume of critical and high alerts generated out of security assessments by the appsec tools. This would be the highlight of this literature survey. Not to mention with time, the appsec tools are getting more and more matured with good relevant assessments and the outcomes of these assessments needs to be reviewed in an automated way with high efficiency. The most efficient way to reduce any kind of risk is through automation, argues Cindy Blake of GitLab. By automating application security scanning, vulnerability remediation and monitoring the application's infrastructure, human error is removed and consistency is applied (Cindy Blake, 2021). This is where machine-learning models such as neural networks could be promising since we are dealing with a pile of structured JSON formatted data.

Now we will go in details what is the current adoption for each of the components is application security secure software development lifecycle, vulnerability detection and remediation process.

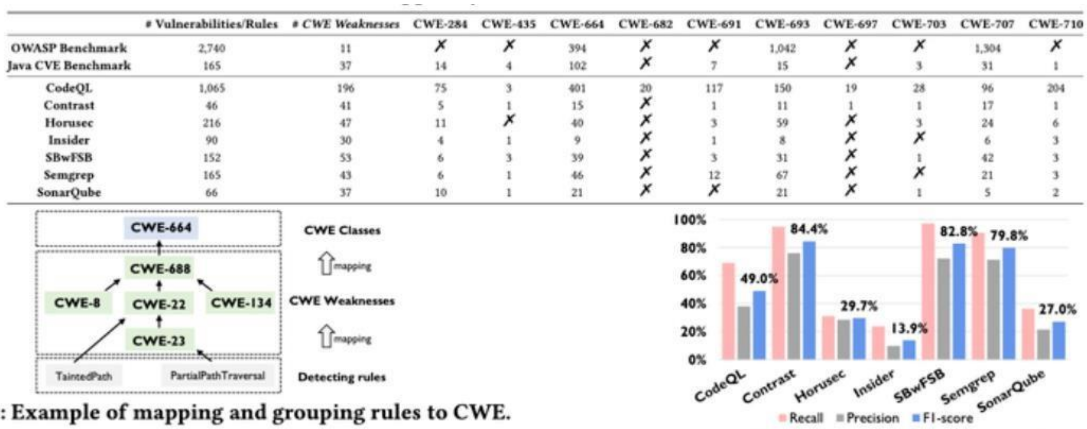
2.4.1 What is Static Application Security Testing and why is it needed?

Static Application Security Testing also known as white box testing approach is considered as one of the critical methodology in modern agile software development lifecycle for identifying critical security vulnerabilities during the initial phase of software development life cycle.

Being a white box testing approach, SAST refers to techniques that analyze source code, bytecode, or binaries for potential security flaws without executing the program (Tyler W. Thomas et al., 2018). Unlike dynamic methods like DAST (Dynamic Application

Security Testing), SAST operates during the "shift-left" phase, enabling developers to address vulnerabilities before deployment (Rafique et al., 2015). Modern SAST tools such as Github Advanced Security CodeQL, Veracode, AppScan Source, Checkmarx, and Fortify employ static program analysis to detect issues ranging from SQL/LDAP injections to insecure API usage. Typical static application security testing tools would perform code level scanning and provide an in-depth analysis and rating based on the CWE-CVE scoring system listed in OWASP TOP 10 or SANS CSC top 25. The industry-standard Open Web Application Security Project (OWASP) top 10 vulnerabilities and CWE/SANS top 25 most dangerous software bugs are setup in a matrix format with open source vulnerability queries, creating an application security framework that aids in review by development teams and emphasizes on code vulnerabilities, minimize false positives discovered in static scans and penetration tests, targeting an increased accuracy of the findings (Jinfeng, 2020). A more recent trend is the Github introduced CodeQL queries, which are literal query formatted syntax, and is used to run to analyze source code in codescan provided by Github Advanced Security (GHAS). This is important to note that in our setup of SAST scanning, we have extensively leveraged Codescanning by GHAS. This would provide us codescanning alerts which would be extracted using the REST API endpoint in Github platform and would further be used for training the GNN – model later. CodeQL is a programming language and associated tools that treat code like data. It was created explicitly to make it easier to analyze code and find potential vulnerabilities in your code with greater confidence than traditional static analyzers. To create dataset for SAST json alerts, we would scan publicly available vulnerable source codes written in java, C#, Javascript and python which will be discussed more later.

Vulnerabilities in code needs to be detected early and patched quickly to prevent ease of exploitability and to reduce the time in which they can be exploited. There are many automated approaches to assist developers in detecting vulnerabilities, most notably Static Application Security Testing (SAST) tools which allows developers to engage early in design phase and take corrective actions if guided properly (Gareth et al., 2024). In short, SAST is a vital process of the secure software development lifecycle (SSDLC). Combined with correct configuration and critical CWE awareness, it can be used early in the development process when fixing security flaws is more simple and less expensive.



	# Vulnerabilities/Rules	# *CWE Weaknesses*	CWE-284	CWE-435	CWE-664	CWE-682	CWE-691	CWE-693	CWE-697	CWE-703	CWE-707	CWE-710
OWASP Benchmark	2,740	11	✗	✗	394	✗	✗	1,042	✗	✗	1,304	✗
Java CVE Benchmark	165	37	14	4	102	✗	7	15	✗	3	31	1
CodeQL	1,065	196	75	3	401	20	117	150	19	28	96	204
Contrast	46	41	5	1	15	✗	1	11	1	1	17	1
Horusec	216	47	11	✗	40	✗	3	59	✗	3	24	6
Insider	90	30	4	1	9	✗	1	8	✗	✗	6	3
SBwFSB	152	53	6	3	39	✗	3	31	✗	1	42	3
Semgrep	165	43	6	1	46	✗	12	67	✗	✗	21	3
SonarQube	66	37	10	1	21	✗	✗	21	✗	1	5	2

2: Example of mapping and grouping rules to CWE.

Table 2.4.1 showing CWE mapped by different SAST Tool (Daniel, 2024)

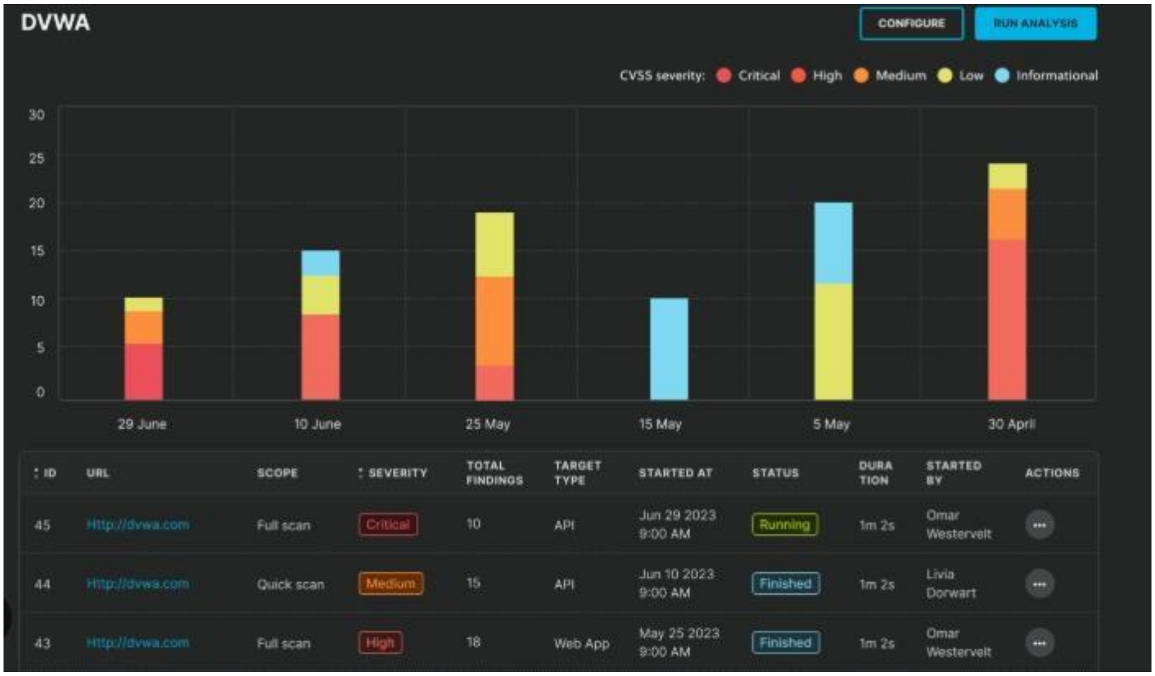
2.4.2 What is Dynamic Application Security Testing and why is it needed?

Dynamic application security testing is also known as black box testing approach due to the fact that all of the testing is carried out without any knowledge of the source code. There is manual way of performing DAST exercise and automated way. Manual preference is usually related in context top pentesting which is beyond the scope of this paper. Automated scanning using a tool and proper configuration gives us continuous scanning and hence security alerts.

There are many industry standard and open source DAST scanning tools in market such as Qualys, Fortify, Snyk, Veracode, Burp Enterprise Edition, OWASP Zap etc. In our analysis, we would leverage Burp Enterprise Edition trial version for performing automated scanning on a list of vulnerable sites to generate the relevant security alerts categorized by High, Medium, Low and Informational combined with confidence levels.

Dynamic application security testing (DAST) scanning consists of automated requests to web applications with the intention of uncovering vulnerabilities that can be exploited with proof of concept. While the legitimate use of scanners aids development teams in improving security postures, they are often used by malicious actors in a brute-force manner for attack reconnaissance with a view to eventual compromise (P. Shahrivar et al., 2023). With proper configurations and environment segregation, DAST scanning can provide a powerful tool to security engineers to find out loopholes in an application’s attack surface which then provides an opportunity to harden the application and thereby reducing risks. Usually the triaging process includes application security analyst to go through the alert details and then map the exploitability to the application functionalities, post which the remediation follows with actionable efforts from the developer team. If the triaging process can be automated using deep learning techniques then it can save a lot of time not only from appsec process but the entire SSDLC cycle.

For example in the paper submitted by Miller and co, they analyzed that leveraging convolutional neural networks (CNN), natural language processing (NLP) and word embedding’s, a model can learn different yet complementary internal feature representations of these exchanges before combining them together to make a good prediction of a analyzed vulnerability or a false positive (FP). Taking in consideration the amount of time and cognitive effort required to constantly manually review high volumes of DAST results correctly by appsec analyst, the addition of this deep learning capability as compared to a rules-based scanner provisions a hybrid system that enables expert analysts to rank scan results, deprioritize false positives and concentrate on likely real vulnerabilities. They showed that this paradigm shift improves productivity and reduces remediation time, resulting in stronger security postures. Evaluations are conducted on a real-world dataset containing 91,324 findings of 74 different vulnerability types curated from DAST scans on nineteen organisations. Results show that the multi-view architecture significantly reduces both the false positive rate by 20% and the false negative rate by 40% on average across all organisations compared to the single-view approach (Stuar Miller et al., 2022).



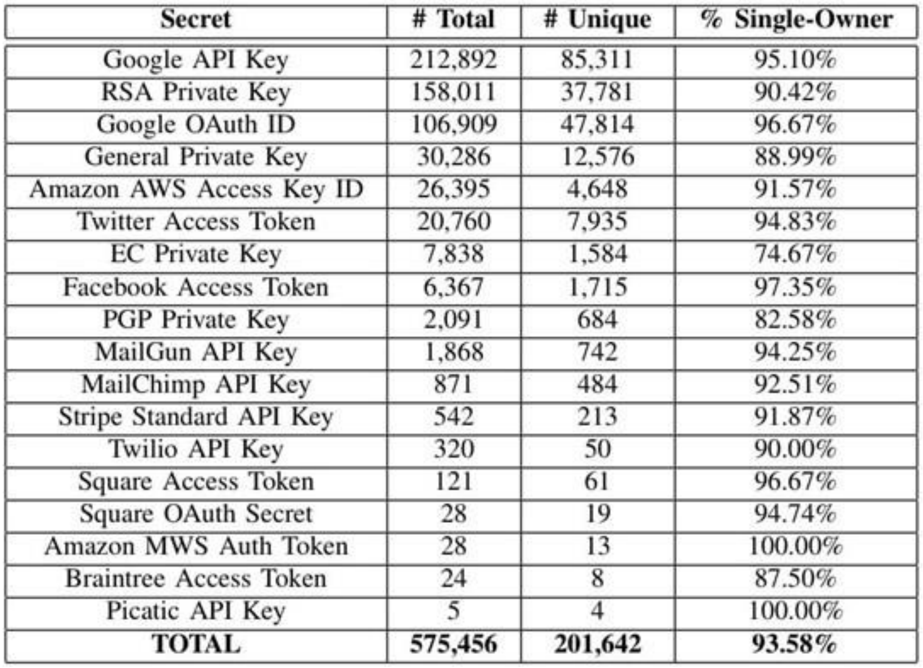
ID	URL	SCOPE	SEVERITY	TOTAL FINDINGS	TARGET TYPE	STARTED AT	STATUS	DURA TION	STARTED BY	ACTIONS
45	Http://dvwa.com	Full scan	Critical	10	API	Jun 29 2023 9:00 AM	Running	1m 2s	Omar Westervelt	•••
44	Http://dvwa.com	Quick scan	Medium	15	API	Jun 10 2023 9:00 AM	Finished	1m 2s	Livia Dorwart	•••
43	Http://dvwa.com	Full scan	High	18	Web App	May 25 2023 9:00 AM	Finished	1m 2s	Omar Westervelt	•••

Table 2.4.2 showing a typical scanning by DAST Tool

2.4.3 What is Secret Scanning in Source code and why is it needed?

Secrets in code poses a great risk to not only the application but to the entire firm. Leaked credentials simply defines the secret that is left behind in source code having authentication capability to access protected resources. There has been several billion-dollar loss and reputational impact due to exploits carried out by leaked secrets which led to sensitive information exfiltration, personally identifiable data (PII), financial history etc. Some of the more recent examples being Pegasus Airlines, Cash App leaking for customer data, Marriott leaks. All of these would have been easily avoided has they have implemented a proper ssdlc flow with defined secrets usage and sanitization.

Standard devsecops flow advocates the scanning of source code for secrets in file systems, SCM, sharing servers, exchange servers, endpoints etc. For our paper, we would be confined to secrets in code for SCM which is Github where we would push our source code and GHAS, in turn would scan for secrets and create secret scanning alerts. Secret scanning alerts can be a lot in a single repository and these are not categorized in any level of severity and as such all the secret findings needs to be carefully triaged, validated before closing the alerts. If any developer dismissed a secret scanning alert without proper due diligence process of validation then it can lead a serious exploitability leading to disastrous results. One of such repositories shows the amount of secrets found in a sample repo:



Secret	# Total	# Unique	% Single-Owner
Google API Key	212,892	85,311	95.10%
RSA Private Key	158,011	37,781	90.42%
Google OAuth ID	106,909	47,814	96.67%
General Private Key	30,286	12,576	88.99%
Amazon AWS Access Key ID	26,395	4,648	91.57%
Twitter Access Token	20,760	7,935	94.83%
EC Private Key	7,838	1,584	74.67%
Facebook Access Token	6,367	1,715	97.35%
PGP Private Key	2,091	684	82.58%
MailGun API Key	1,868	742	94.25%
MailChimp API Key	871	484	92.51%
Stripe Standard API Key	542	213	91.87%
Twilio API Key	320	50	90.00%
Square Access Token	121	61	96.67%
Square OAuth Secret	28	19	94.74%
Amazon MWS Auth Token	28	13	100.00%
Braintree Access Token	24	8	87.50%
Picatic API Key	5	4	100.00%
TOTAL	**575,456**	**201,642**	**93.58%**

Table 2.4.3 showing secrets available in a repo

As stated before, manual process of secret remediation incurs huge effort but if the developers workflow can be fed back to train an AI model to prevent the incorrect/improper dismissal of secret scanning alerts that would greatly aid the accountability on developers action and help ssdlc processes (Negri-Ribalta et al., 2024)

2.5 Importance of Governance Gate control on Policy defined by SSDLC

A good governance model is always implemented to enable compliance, observability, accountability, and metrics presentation in reports. Gate checks in devops pipeline setup prevents deployment to production for codes, which are non-compliant, doesn’t meet SSDLC scores, codes wasn’t reviewed or has some other issues defined as such within the DevSecOps practice by leadership. This enables application security teams to plug in control measures to enhance confidence in bug/exploit free releases. Though this is impacting to the devops flow but this can be minimized with early planning and getting all the deployable source codes scanned and be compliant. The

SSDLC introduces the secure system development life cycle (SSDLC), cloud security critical domain guidelines, and risk considerations which is manifested via a proper Governance framework (T. -C. Kao et al., 2012).

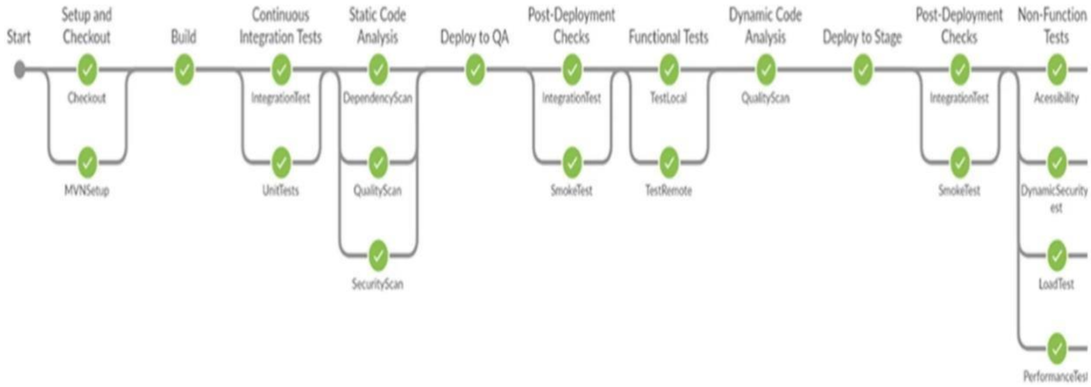
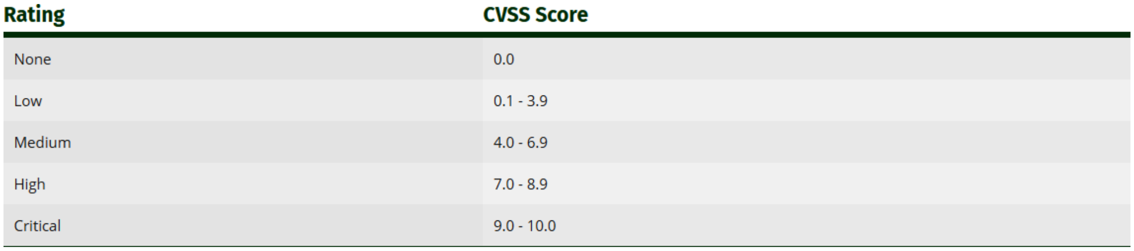


Figure 2.5 showing Governance gates in Devops Pipeline

A well defined policy enabled governance gates fails on critical and standardized scanning results coming out of plugged in appsec tools such as SAST, DAST, SCA, Secret scanning, Code coverage tools such as Sonarqube, Cloud configuration checks, container scanning etc. The policy is usually derived from the CVSS scoring model or any other standard models such as for CodeQL in our paper we would define the policy on CWE-ID and it’s relevant scoring criteria found in NVD. For our SAST Tool, CodeQL - The severity level for a code scanning alert indicates how much risk the issue applies to my source code.

- Severity - All code scanning alerts have a level of Error, Warning, or Note.
- Security severity - Each security alert analysed using CodeQL too has a security severity level of Critical, High, Medium, or Low.

When an alert has a security severity level, code scanning displays and uses this level in preference to the severity. Security severity levels follow the industry-standard Common Vulnerability Scoring System (CVSS).



Rating	CVSS Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Table 2.5 showing security severity scale

For DAST tool, the severity score of each alert would be of similar scale but secret scanning alerts any open secrets possess a risk and hence needs proper validation and closure. Governance gates enforces developers to work on these alerts defined by policy for their code to be released.

2.5.1 Existing process of Security Alert remediation practice and why a gap exists?

Triaging of security scanning alerts depends on the risk appetite of an organization and resources available to address them. Even with a dedicated SSDLC team doing remediation of these security scanning alerts takes a lot of time due to manual mitigation practices. This is why we need to establish a better process to automate remediation process to some extent. For example, one of the followed approach is the developers being the creator of their source codes can go through, issue, and resolve them by either fixing the code or dismissing the security scanning alerts. The risk lies with the later part as there is a possibility that a critical or a policy failing alert is dismissed without remediation. This not only leaves a bug in the application source code but also poses significant risk of exploitation or confidence loss when caught in audit actions or any similar process.

We can set up logical checks around different CWE categories of these alerts to prevent these but seeking manual checks consumes time as well as resources which also negatively impacts the devops experience. With the increased usage of AI models, we need to find a better way if any such models can assist in detecting these improperly closed policy failing alerts and make the SSDLC process more robust.

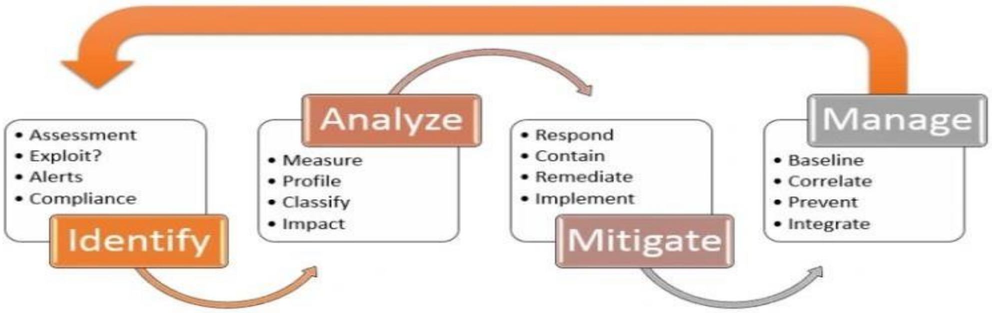


Figure 2.5.1 showing a standard mitigation process in ssdlc

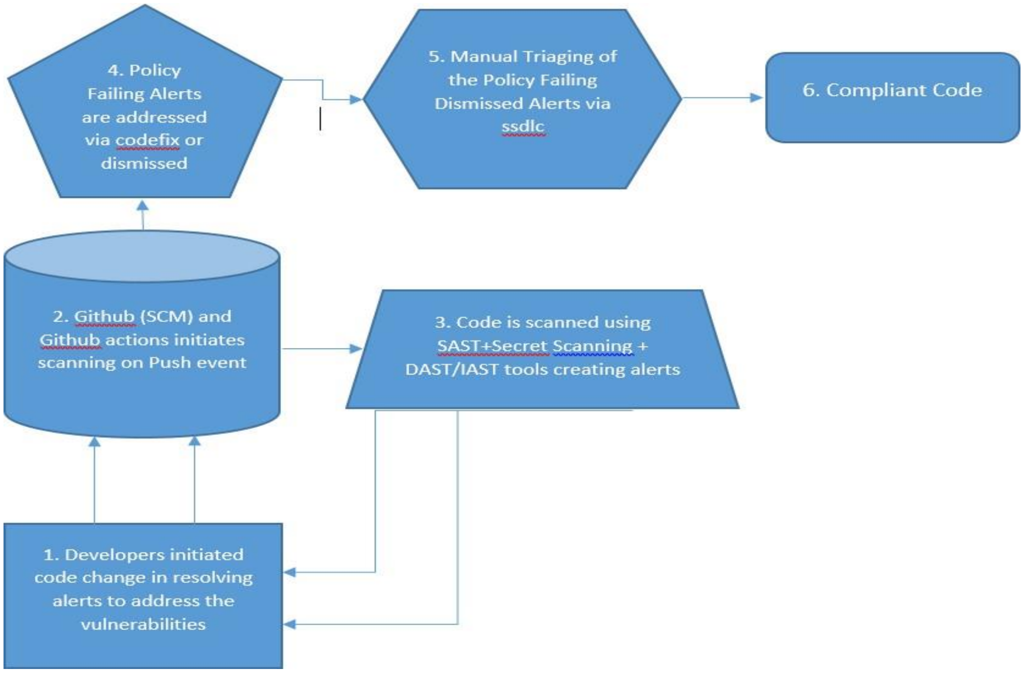

4. Policy Failing Alerts are addressed via codefix or dismissed
5. Manual Triaging of the Policy Failing Dismissed Alerts via ssdlc
6. Compliant Code
2. Github (SCM) and Github actions initiates scanning on Push event
3. Code is scanned using SAST+Secret Scanning + DAST/IAST tools creating alerts
1. Developers initiated code change in resolving alerts to address the vulnerabilities

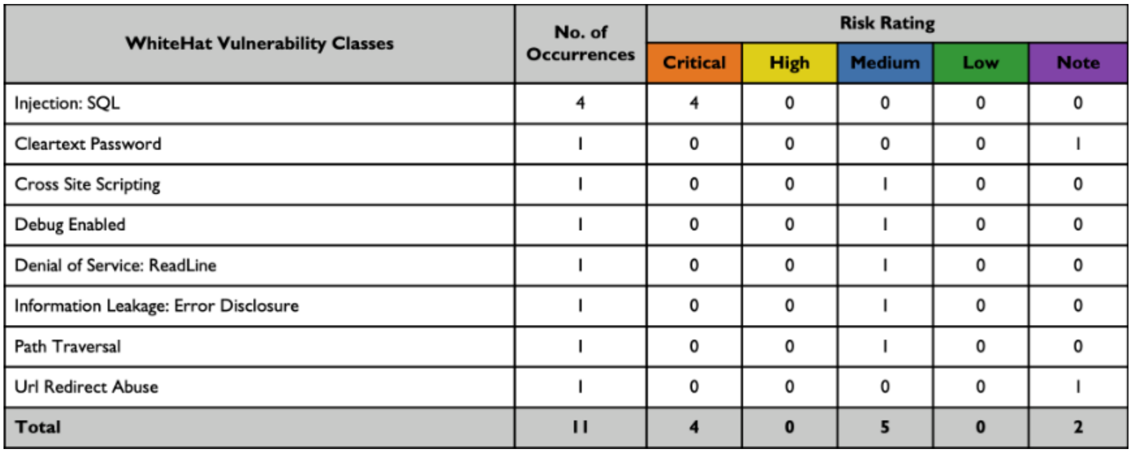
Figure 2.5.1.1 Manual process of SSDLC Remediation approach of Policy Failing Alerts

Figure 2.5.1.1 is self-explanatory as from steps 1 to 6 it is shown of the standard process that is being followed requiring manual time-consuming triaging process of policy failing alerts as defined by SSDLC standards of security testing (Kamal et al., 2020).

2.6 What is Vulnerability Analysis in SSDLC and why is it needed?

Vulnerability analysis is the process in which the alerts created by the appsec scanning tools such as SAST, SCA, Secret Scanning, DAST/IAST, container scanning etc are studied, analyzed and prioritized for resolution in a standard SDLC process. It is pivotal to prevent security breaches, protect sensitive data, maintain customer trust, and leading to reduction of the cost of fixing security flaws by addressing them early stages in SDLC lifecycle (A. van Rensburg et al., 2017).

- Identification: The first step involves finding security weaknesses, or "vulnerabilities," within source codes or configurations.
- Classification: Detected vulnerabilities are then categorized based on their type and potential impact on the application.
- Prioritization: A risk-based approach is employed to determine which vulnerabilities pose the most significant threat and hence require swift intervention.
- Remediation: Once identified and prioritized, an action steps is created to mitigate these risks by resolving the vulnerabilities, through a process sometimes called vulnerability management.



WhiteHat Vulnerability Classes	No. of Occurrences	Risk Rating				
		Critical	High	Medium	Low	Note
Injection: SQL	4	4	0	0	0	0
Cleartext Password	1	0	0	0	0	1
Cross Site Scripting	1	0	0	1	0	0
Debug Enabled	1	0	0	1	0	0
Denial of Service: ReadLine	1	0	0	1	0	0
Information Leakage: Error Disclosure	1	0	0	1	0	0
Path Traversal	1	0	0	1	0	0
Url Redirect Abuse	1	0	0	0	0	1
Total	**11**	**4**	**0**	**5**	**0**	**2**

Table 2.6 showing a sample Vulnerability table mapping to risk

Vulnerability mapping aids the ssdlc team in establishing the policy which will define the failing conditions governance gate checks. Usually the, vulnerability mappings is based on the OWASP Top 10 for the particular year in context and it is mapped across different risk ratings such as critical, high, medium, low and informational/note. We have to bear in mind that some findings can be false positives and this is where the analysis comes in and helps in formulation of policy, which can be used as gating conditions enforcing the security analysts and developers alike to fix the issues. The CWE and CVE scores sets up the severity of the weakness for which the alerts are created post scanning the source codes or application. The secrets on the other hand is a bit different process and if it is a True Positive, the developer should be mandated to fix by revoking and then rotating the exposed secret in context.

2.6.1 Need of fine balance on Flexibility vs Risk in Developer Experience

Modern day DevSecOps process with several appsec scanning tools plugging in generating huge amount of alerts and enforecing developers to work on these can directly impact the devops experience. Developers are required to develop quality bug free codes and these scanning tools is suppose to be there to aid in that process but lately due to so many exploits happening and system es being tagetted by nefarious groups, it became more and more important that we address the exploitable bugs in applications. The problem is this bring a tremendous additional burden on developers to maintain compliant codes by going through the list of alerts created for their applications.

So, it is important to maintain a flexible approach which would aid the developers to focus on critical policy failing alerts while releasing risk free deployments. Due to this, the possibility of improper dismissal of policy failing alerts increases and therby opens up an opprtunity to create an automated process to detect these and report back instead of queueing up for manual detections. Nora and Jingyue has collected the informations about the security practices rooted in Devops flow, mixed with an essence of security culture where the result identified the importance of caring about security and issues in existing processed, such as how developers gets overwhelmed by security engineers alike when vulnerable code is detected and the urgency follows to fix these. Post establishment of the security culture in an organization, development and operations need the necessary training and knowledge so that security automation tools can be utilized effectively as in not to burden the developers with code fixes (Nora et al., 2019). Debi and Gail showed that the awareness, education and establishing a process to automate checks to prevent mistakes is a great way to boost developer experience and their overall trust to the SSDLC process increases (Debi and Gail, 2021).

2.6.2 Managing Compliance in Appsec

Compliance traditionally is a product that came out of the risk assessments and threat analysis executed on SSDLC standards. The risk appetite is usually varies from organization to organization but is usually be broadly based on most critical vulnerability type of the security scanning alerts. By default, almost any leaked credentials is considered as critical issue and hence the compliance is generally derived as code which is nothing but a policy definition which enable gating and implements governance across devops CI/CD pipeline. Compliance as a code is more favored than traditional manual definitions since the code can then be ingested and can serve as a source of truth to enable polices across different systems. This is important since we would want to have a unified process driving compliance across different systems like cloud governance, Pre-release assessments, release checks, code promotion etc. In the domain of safety critical systems, reliability requirements are highly prioritized and organizations have mature compliance processes in place. (Tuma K et. al., 2023). Effective compliance management within application security hinges on integrating regulatory and industry standards into the secure software development lifecycle (SSDLC), automating continuous controls monitoring, and fostering an organizational culture of security awareness.

The evolution toward DevSecOps and continuous compliance represents the next frontier in application security compliance management, where regulatory requirements are automatically enforced through code, infrastructure, and deployment pipelines. This approach enables organizations to maintain compliance at the speed of modern software development while reducing the risk of manual errors or oversights that could lead to compliance violations. Advanced implementations use artificial intelligence and machine learning to analyse patterns in compliance data, predict potential compliance issues, and recommend optimal remediation strategies based on historical effectiveness and cost considerations. Managing compliance in application security ultimately requires a holistic approach that balances regulatory requirements with business objectives, technical constraints, and resource limitations. Organizations that successfully navigate this complexity develop integrated compliance programs that leverage automation, risk-based prioritization, and continuous improvement to maintain regulatory adherence while enabling innovation and business growth. These programs recognize that compliance is not merely a technical challenge but an organizational capability that requires coordination across multiple disciplines and stakeholder groups to achieve sustainable success in increasingly complex regulatory environments.

2.6.3 Vulnerability Classification

Vulnerability classification in application security represents a systematic approach to categorizing, prioritizing, and managing security weaknesses discovered within software applications and systems. This discipline forms the ground for effective vulnerability management programs, enabling organizations to take informed decisions about resource allocation, risk mitigation strategies, and remediation timelines. The classification process involves multiple dimensions and frameworks that work together to provide a comprehensive understanding of security risks (S. Rafique, et al., 2015).

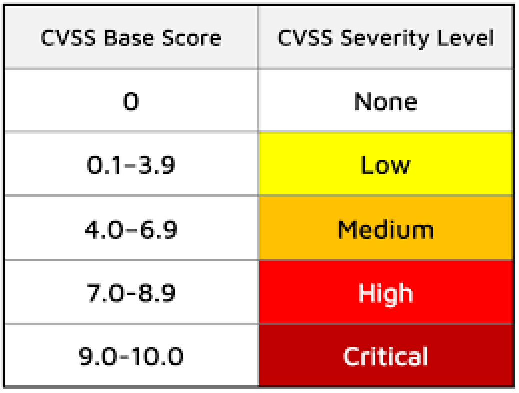
The most widely recognized classification framework in application security is the OWASP Top 10, which provides a community-driven perspective on the most critical web application security risks. This framework has evolved significantly since its inception, with the 2021 edition reflecting modern threat landscapes and development practices. The current iteration includes Broken Access Control as the top risk, recognizing that unauthorized actions due to improper enforcement of user permissions represent one of the most prevalent and damaging vulnerability types. Cryptographic failures have risen in prominence, acknowledging that sensitive data exposure when cryptography is missing or misused poses significant risks in an era of increasing data protection regulations. Injection vulnerabilities, while still critical, have shifted in ranking as modern frameworks provide better protection against traditional SQL injection attacks, though they remain a fundamental concern for legacy systems and emerging injection vectors (T. Petranović, et al., 2023)

The Common Weakness Enumeration (CWE) system provides a more granular and technical classification approach, maintained by MITRE as part of their cybersecurity taxonomy efforts. CWE operates as a hierarchical system that categorizes software weaknesses from broad architectural issues down to specific implementation flaws. This system proves invaluable for security tools and analysts who need precise categorization for automated scanning, reporting, and remediation tracking. For instance, Cross-site Scripting vulnerabilities are classified under CWE-79, with further subdivisions for reflected, stored, and DOM-based variants. This level of detail enables security teams to apply targeted mitigation strategies and select appropriate testing methodologies for each weakness type (A. Honkaranta et al., 2021).

Severity assessment through the Common Vulnerability Scoring System (CVSS) adds a quantitative dimension to vulnerability classification that enables consistent risk evaluation across different systems and organizations. CVSS v3.1, the current standard, evaluates vulnerabilities across multiple metric groups to produce scores ranging from 0.0 to 10.0. The base metrics examine inherent characteristics such as attack vector complexity, required privileges, user interaction needs, and the potential impact on confidentiality, integrity, and availability. Temporal metrics account for changing factors like exploit maturity and available patches, while environmental metrics allow organizations to adjust scores based on their specific context, such as the criticality of affected systems and existing security controls.

Modern vulnerability classification extends beyond traditional frameworks to incorporate exploitability and business impact assessments. Exploitability analysis considers factors such as the availability of public exploit code, the technical skill required for successful exploitation, and the accessibility of vulnerable endpoints. For example, a SQL injection vulnerability in a public-facing login form with available exploit tools would receive a higher exploitability rating than a similar vulnerability requiring authenticated access and custom exploit development. Business impact assessment evaluates the potential consequences of successful exploitation, considering factors such as data sensitivity, system criticality, regulatory compliance requirements, and potential financial losses from service disruption or data breaches.

The integration of these classification dimensions creates a comprehensive vulnerability management workflow that begins with automated discovery through Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and Software Composition Analysis (SCA) tools. Each tool contributes different perspectives on vulnerability identification, with SAST analysing source code for potential weaknesses, DAST testing running applications for exploitable vulnerabilities, and SCA examining third-party components for known security issues. These tools typically output findings tagged with CWE identifiers, which then undergo normalization processes to map them to organizational classification schemes and OWASP categories. Risk scoring and prioritization represent critical phases in the classification workflow, where base CVSS scores undergo adjustment based on organizational context and threat intelligence. Security teams analyse factors such as exploit availability, attack surface exposure, and business criticality to establish priority tiers that guide remediation efforts. This process often involves collaboration between security, development, and business stakeholders to ensure that classification decisions align with organizational risk tolerance and operational constraints. For instance, a medium-severity vulnerability in a payment processing system might receive higher priority treatment than a high-severity issue in a non-critical internal tool.



CVSS Base Score	CVSS Severity Level
0	None
0.1-3.9	Low
4.0-6.9	Medium
7.0-8.9	High
9.0-10.0	Critical

Table 2.6.3 showing CVSS severity level

The evolution of DevSecOps practices has transformed vulnerability classification from a periodic assessment activity into a continuous, automated process integrated throughout the software development lifecycle. Modern classification systems incorporate policy-as-code approaches that define vulnerability acceptance criteria, automatic escalation rules, and integration points with development workflows. This automation enables rapid identification and classification of newly discovered vulnerabilities while maintaining consistency across different applications and teams. Advanced implementations leverage machine learning algorithms to improve classification accuracy by analysing historical vulnerability data, exploit patterns, and remediation outcomes. Metrics and reporting capabilities derived from vulnerability classification systems provide valuable insights for security program management and continuous improvement initiatives. Organizations track metrics such as mean time to detection, remediation velocity by vulnerability category, and the distribution of vulnerabilities across different severity levels and application components. These metrics inform resource allocation decisions, training needs identification, and the effectiveness of preventive security measures. Executive dashboards translate technical vulnerability data into business-relevant risk indicators that support informed decision-making about security investments and risk acceptance strategies.

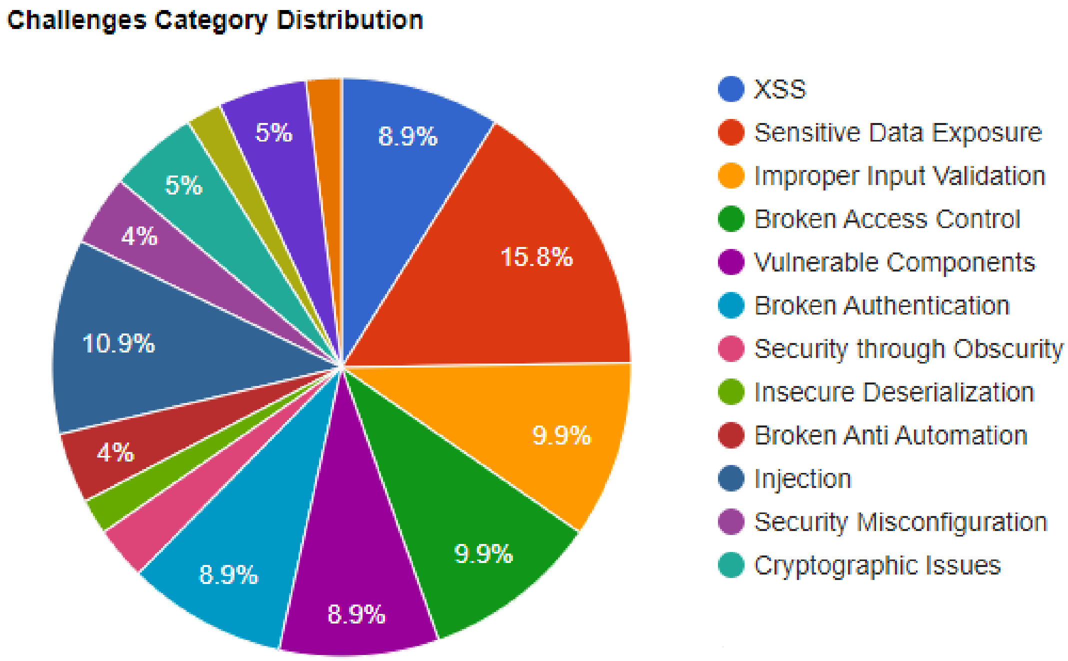


Table 2.6.3.1 chart showing CWE distribution for a sample application

Effective vulnerability classification ultimately serves as the foundation for mature application security programs that can efficiently manage risk while supporting business objectives. By combining established frameworks like OWASP Top 10 and CWE with quantitative scoring systems like CVSS and organizational context factors, security teams can create classification systems that provide clear guidance for remediation prioritization, resource allocation, and risk communication. This systematic approach to vulnerability classification enables organizations to move beyond reactive patching toward proactive risk management that aligns security efforts with business priorities and regulatory requirements.

2.6.4 Emergence of Mobile Security

As web apps became relevant in terms of attack surface for bug exploitation, mobile applications in market saw a tremendous rise. It became imperative more so ever that the mobile apps whether be it build on Kotin, Ios, native react, flutter is comprehensively put through devsecops process and ensure there are no exploitable bugs present in releases for these apps. The standard SAST/DAST with secrets in code scanning would also apply to mobile applications and the only difference would are the attack vectors as compared to the web applications. We have the OWASP MASVS and MASTG standards for open framework to perform security testing for mobile applications. This means there are more security scanning alerts that would have to undergo developer-assisted remediation and hence the possibility to incorrectly dismiss a potential policy failing alerts increases.

The rise of mobile security represents a fundamental shift in cybersecurity that requires specialized expertise, tools, and methodologies to address the unique challenges and opportunities created by mobile computing. Mobile Application Security Testing has evolved from ad-hoc approaches to comprehensive methodologies that integrate multiple testing techniques, regulatory requirements, and business considerations to provide holistic security assessments for mobile applications and ecosystems. As mobile technologies continue to evolve and expand into new domains, mobile security testing must continue to adapt and innovate to address emerging threats while enabling the benefits of mobile computing for individuals, organizations, and society as a whole (M. Becher et al., 2011).

2.7 Impacts of Exploitable vulnerabilities in Application

Once a vulnerability is live and if it is exploited, the consequences are fatal to a company reputation and now with all the data regulations such as GDPR in place, it attracts a massive penalty. The exploitation of vulnerabilities in applications represents one of the most significant cybersecurity challenges facing organizations worldwide, with consequences that extend far beyond immediate technical disruptions to encompass financial losses, regulatory penalties, operational disruptions, reputational damage, and strategic business impacts. Understanding these impacts requires examining both the direct technical consequences of successful exploitation and the broader organizational, economic, and societal implications that emerge from compromised application security. The complexity and interconnectedness of modern application environments amplify these impacts, creating cascading effects that can persist long after initial vulnerabilities are discovered and remediated. The financial consequences of exploited application vulnerabilities represent the most quantifiable and immediate impact category, encompassing both direct costs associated with incident response and remediation and indirect costs related to business disruption, regulatory penalties, and long-term competitive disadvantages. Recent research indicates that the average cost of data breaches has reached unprecedented levels, with organizations experiencing average costs exceeding $4.88 million per incident, representing significant increases over previous years as attack sophistication and regulatory requirements continue to evolve (Sarah Elder et al., 2024).

Direct incident response costs encompass the immediate expenses associated with identifying, containing, and remediating security incidents resulting from exploited vulnerabilities. These costs include forensic analysis to determine the scope and nature of the compromise, emergency response team deployment, external consulting services, legal counsel, and technical remediation activities such as system rebuilding, data recovery, and security control implementation. Organizations often underestimate these costs, which can escalate rapidly as incidents require extended investigation periods, complex technical remediation efforts, and coordination across multiple organizational units and external partners. Business disruption costs frequently represent the largest component of vulnerability exploitation impacts, encompassing revenue losses from system downtime, productivity losses from workforce disruption, customer service impacts, and supply chain disruptions that may affect business operations for extended periods. Application vulnerabilities that result in system unavailability or degraded performance can cause immediate revenue losses, particularly for organizations that depend on digital channels for customer interactions, transaction processing, or service delivery. The increasing dependence on digital systems across all industries means that application vulnerabilities can rapidly translate into significant business impact. Regulatory and compliance penalties have become increasingly significant as governments and regulatory bodies implement stringent requirements for data protection, privacy, and cybersecurity. Organizations that experience data breaches or security incidents resulting from exploited application vulnerabilities may face substantial fines, regulatory sanctions, and compliance obligations that extend far beyond the immediate incident. The European Union's General Data Protection Regulation (GDPR) enables fines up to 4% of annual global turnover, while various sector-specific regulations in healthcare, finance, and other industries impose additional penalties and requirements that can result in ongoing compliance costs and operational restrictions.

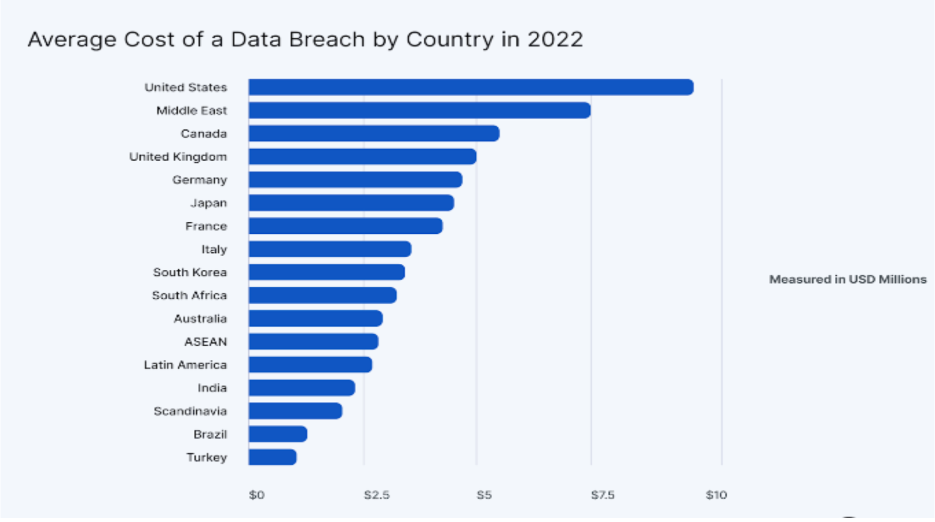


Table 2.7 shows Average cost of a data breach by country in 2022 measured in millions of USD highlights financial impact variances globally

2.8 Automating Mitigation Process with Deep Learning Model

Over the years, significant improvements in machine learning models have laid the foundation of advanced analytics (predictive), greatly aided in infrastructure monitoring, malware threat detection etc. Deep learning models are a subset of machine learning concept which is based on artificial neural networks. It is capable of learning complex patterns and relationships within data. In this technique, it is not needed to explicitly program everything. It has become increasingly popular in recent years due to the advances in processing power and the availability of large datasets. Because it is based on artificial neural networks (ANNs) also known as deep neural networks (DNNs), these neural networks are inspired by the structure and function of the human brain's biological neurons, and they are designed to learn from large amounts of data (LeCun Y. et al., 2015). One of the primal characteristic of Deep Learning is the use of deep neural networks, which have multiple layers of interconnected nodes. These networks can learn complex representations of data by discovering hierarchical patterns and features in the data. Deep Learning algorithms can automatically learn and improve from data without the need for manual feature engineering (Mathew A. et al., 2021).

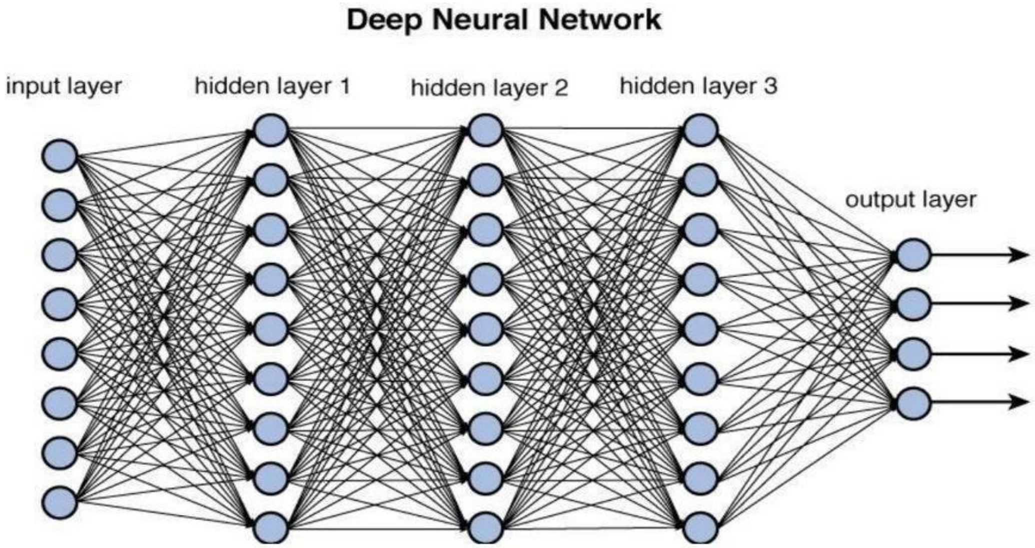


Figure 2.5 Deep Learning network architecture (Ravindra Parmar, 2018)

All the appsec tools post security assessments produces data in JSON format, which can be leveraged in various machine-learning models to gauge their efficiency in driving automation in the appsec mitigation process. Existing deep learning models trained on large text corpora have been successful at performing single-column semantic type prediction for relational data. But can also be applied to semantic type prediction problem to JSON data, labeling the types based on JSON Paths (Shuang Wei, et al., 2023). This is still a very niche area where lot of good research is happening.

Identification of CWE-119 using various neural network models with efficiency of 91% above is experimentally shown as achieved, experimental results show that the proposed approach can achieve an accuracy of 0:9732 for CWE-119 and 0:9721 for

CWE-399, which is higher than that of the other three models (the accuracy of RNN, LSTM, and BLSTM is under than 0:97) (Weina Niu et al., 2019). So, if CWE classification can be achieved using the above state neural models, the combination of

SAST and DAST tools produces alerts based on CWE parameters. However, exact amount of work in the flaw mitigation analysis using machine-learning models is less but there has been significant work in similar types. Study in asymmetric costs of false positive errors and false negative errors for intrusion detection using neural network models showed that the neural network model provides very high performance for the accuracy of intrusion detection (Daejoon Joo et al., 2003).

Combination of regular expression search and machine learning reduced the number of false positives in secret detection as well (Aakanksha Saha, 2019). Application of LOMOS - log analysis artificial intelligence-enabled framework in running SAST check for Infrastructure as Code provided very optimistic output (Matija Cankar et al., 2023).

We can observe that the implementation of several machine learning models provided valuable results while combining with appsec tools. In this paper, we will try to measure the efficiency of graph neural network model in assisting in triage automation of alerts and promote a certain degree of automation in the process.

2.8.1 Utilization of Graph Neural Network in Application security

Over the last decade, there has been several usage of AI models to be used on various capacities such as static scanning of application source codes, automated detections aiding in intrusion detections systems, scanning for secrets etc. Take for example, the systematic review done on IDS (Intrusion Detection System) by Ramya and her team which was based on based on Preferred Reporting Items for Systematic Reviews and Meta-Analyses: (PRISMA2020) guidelines. It analyzed the latest trends used in data preparation, DL architectures, and performance evaluation metrics for NIDS. The review gave an in depth stats into various datasets and tools used in the field, highlighting the effectiveness of DL in improving NIDS performance (Ramya et al., 2025). Another major step towards leveraging the power of neural networks happened in the space of manual penetration testing. The mere possibilities of neural networks application on solving the problem of selecting tools for penetration testing of web applications are considered using the initial data for the construction of a website that can aid to form a vector of requirements using a questionnaire mechanism and to select tools that will fully satisfy user's requirements have been analyzed (A. Tetskyi et al., 2018).

Graph neural networks are more recent in their adaptation given their complex constructs and usage via python libraries. Graph neural network model is essentially graph with convoluted neural networks and we all know how much CNN is best suited for image classification analysis and computer vision applications. Here, the more relevant question that we are trying to assess is if we can predict an incorrectly remediated/dismissed scanning alert by a developer. Taking into consideration, that closure actionable alerts represents the graphs with JSON objects leading to closure as its nodes alike. For instance, in such a paper it was studied that on dynamic line graph neural network (DLGNN)-based intrusion detection method with semi supervised learning. The model converts network traffic into a series of spatiotemporal graphs. The dynamic GNN (DGNN) is leveraged to extract spatial information from each discrete snapshot and capture the contextual evolution of communication between IP pairs through consecutive snapshots (G. Duan et al., 2023). The achieved an impressive accuracy of 98.15–99.8% in abnormality detections. In our study too, we are trying to detect the abnormal closure of policy failing scanning alerts that if left un-resolved increases the risk of exploitation increasing the attack surface of the application.

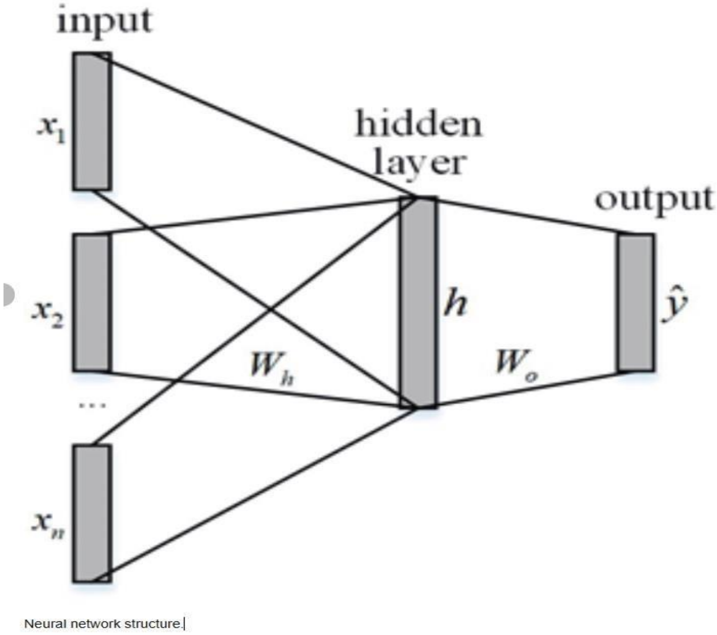


Figure 2.8.1 showing DL-GNN

2.9 Summary

“With increasing cyberattacks, an IT talent gap, and widespread economic challenges, security automation solutions are becoming a business imperative.’’ (Tim Reed, 2023). Through the literature review, we can conclude that the standard AppSec tool produces many alerts, which includes both True Detections and False Positives. These alerts undergoes triaging process which has manual review and mitigation approach, in addition to the fact that only certain categories of alerts based on policy definition are taken into consideration while ignoring rest as False Positives without review and in some instances, there is no review process at all. The manual mitigation process itself takes a long time to complete and thus adds delay to SDLC lifecycle. Literature has provided a substantial amount of information regarding the programs, processes, risks and existing practices. To address this gap with automation, we will study and measure the efficiency of using deep learning and also, propose a framework to represent in a standard DevSecOps flow in a better way.

We know that why is it important to promote automation and reduce dependency on manual process. The exploration of the usage of machine learning models in appsec flaw mitigation process would provide us a valuable feedback in its adoption (Harer & Jacob., 2018). Presented literature will definitely contribute to overall concepts and I do hope that my research will contribute to better understanding of the challenges and gap in the manual flaw mitigation approach and how we can leverage machine learning models to achieve automation with a proposed framework.

CHAPTER III:
METHODOLOGY

3.1 Overview of the Research Problem

Flaw mitigation and vulnerability remediation in a standard SSDLC process is always time consuming with immense volume of effort spend by appsec analyst in collating policy failing alerts context from source code, triaging with developers, understanding the nature of the alerts, analyze False Positives, performing POC to substantiate the depth of exploitibility, prepare reports and finally once the source code is updated to fix the issue – run the scans and re-validate the fix. This is not only time intensive but costly also in terms of appsec analyst head count and hours needed to operationalize and support the developers. This is why many organizations have specific policy setup for common vulnerabilities that requires additional appsec analyst to review while majority of the alerts related to other vulnerability is managed by developers or in some case the entire alerts are triaged by developers. This introduces risks and possibility of a ‘Çritical’ or ‘High’ vulnerability ignored remains true. Though granted the manual analysis if done by an appsec analyst reduces the risks greatly but it is also associated as discussed above with time and resources. This is why we must explore alternatives and one such approach is using machine learning or deep learning using Neural networks with feedback mechanism to train a model which could auto triage in preventing high risk alerts dismissed preemptively as generated by scanning tools like SAST+DAST+Secret Scanning and if these are closed without any proper remediation applied. The model can be trained to notify appsec team and help reopen the alert for more analysis thereby reducing the risk of that vulnerabilty being exploited.This would not only reduce time by on flaw remediation but also can be used to inform developers to take action on such alerts.

Like Abdel Aziz showed in his paper that integrating the output xml data of different scanners and utilizing Large Language Models (LLMs) that previously trained on standard cybersecurity frameworks, the system can provide actionable recommendations that enhances the proactive threat detection through system logs analysis (A. M. AbdelAziz, 2024). Very similar approach can be used to utilize GNN (Graph Neural Network) to train on Json datasets though this paper would not explore the integration with LLM’s which can be a future step. Till now, the standard approach of post remediation of closed alerts generated by appsec scanning tools has always been manual so the idea is to see if a deep learning model such as GNN can be leveraged to promote automation so a potential vulnerability is not missed due to wrongful dismissal and find possible ways these AI models can assist in the triaging process. The datasets would be more applicable in developer centric remediation of security scanning alerts where we are trying to inject the usage of a deep learning model and study if this approach can aid in better detection of more exploitable relevant dismissed alerts thereby preventing risks and saving time as compared to manual remediation practices.

3.2 Operationalization of Theoretical Constructs

Since the paper would be leveraging deep neural network model such as Graph Neural Network, we would need to understand the following parameters that would gauze the analysis indicators with regards to the performance of the usage of the apforesaid alogorithm. TP stands for True Positive Counts among the sampled dataset, FP stands for False Positive Counts among the sampled dataset, FN stands for False Negative Counts among the sampled dataset, Precision Score given by TP/(TP+FP) as it measures the accuracy of positive predictions (what proportion of predicted positives are actually positive), Recall Score given by TP/(TP+FN) as it measures the ability to find all relevant instances (what proportion of actual positives were correctly identified), F-Score is a metric that combines precision and recall into a single value, providing a balanced measure of a model's performance, especially useful in classification tasks (Peter Christen et al., 2023). It's the harmonic mean of precision and recall, giving equal weight to both. A higher F1-score indicates better model performance.

Next, we have the confusion matrix is a tool used to evaluate the performance of a model and is visually represented as a table

Table 3.2.1 showing a sample Confusion Matrix

Alert	Predicted Value	
Actual Alert	Vulnerable	Not vulnerable
Vulnerable	TP	FN
Not vulnerable	FN	TN

Next is the heat map which is a visualization technique that represents data quantities through colors on a two-dimensional space though this is more useful for image classification or object detection, nonetheless this can help us understanding more relevant aspect of detection.

3.3 Research Purpose and Questions

The overall aim of the research is to study the possibility of using deep learning technique such as neural network to address the gap that exists in the remediation of application security tool scanning alerts that are improperly closed. The study then is to check out if the trained model would then allow the option to application security engineers to automate triaging of the relevant alerts and assist the developers to take preventive measures to address them and thereby fix potential exploitable vulnerabilities. Additionally, the study will also explore to propose an alternate framework in secure software development lifecycle to address this remediation of improperly closed relevant security alerts.

➢ Is deep learning technique such as graph neural network feasible enough to be used in application security software secure development lifecycle remediation space?

➢ Can the neural network then be utlised to detect incorrectly closed security alerts and inform application security team to triage?

➢ Can we further design and develop a framework to plug the above process in existing devsecops process?

3.4 Research Design

The research design is the extended blueprint of the research (Leandros et al., 2020) and (Ugur et al., 2017) .Instead of implementing on network models, our context shifts to data generated by application security tools such as SAST, DAST/IAST, secret scanning. We will restrict our datasets to SAST, DAST and secret scanning tools and not cover SCA. It addresses the following issues: how deep learning technique is leveraged to show improvement in IDS (Intrusion Detection System) and how a classifier is used to accurately a large majority of False Positive issues, what data is relevant to the research, what data should be collected, and how the data will be analyzed (Stuart et al., 2022; Hüther et al., 2023).

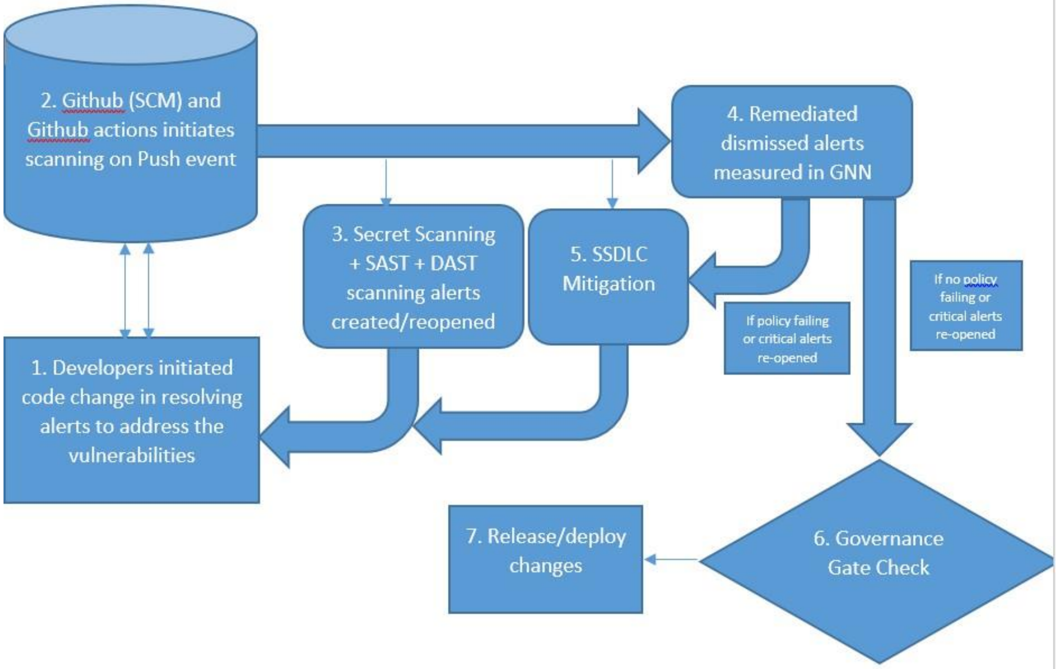
Regarding methodology, this study will use a combination dataset of alerts arising from tools such as static application security testing (SAST), interactive or dynamic application security testing (IAST/DAST) and secret scanning samples. We then employ deep learning methods preferably convoluted neural network (Stuart, et al., 2022). Finally we measure their efficacy using performance indicators namely, accuracy, false positive rate, false negative rate, detection rate, time to train to reach acceptance level and finally compare with the manual mitigation process.

Once we achieve our desired comparative stats and with favorable data points, we will propose an alternative approach for appsec flaw mitigation or vulnerability remediation process using deep learning methods. For the purpose of this research, we have listed top 13 vulnerable web applications that are typcally used for ethical hacking. These webapps with their respective source code is pushed into the source code management (SCM) github repositories under an org level classification. These repositories were all then scanned using CodeQL – a Github native static application security testing tool using Github actions. Along with the SAST scanning, all of these exploitable sites were scanned using Github Secret Scanning with a combination of inbuilt and custom rules. Next, all the exploitable websites are hosted in ec2 instances or in some cases localhost and then these are scanned with Dynamic application security testing tool such as BURP EE, HCL AppScan DAST Github Action or OWASP ZAP. The SAST tool used to perform the scanning is Github Advanced Security Scanning (GHAS) CodeQL using Github actions and the source scanning for secrets is accomplished by secret scanning by also GHAS. The source code is maintained and stored in repositories in Github. Code analysis is an activity that needs several resources, especially when seeking to identify vulnerabilities that compromise the integrity of the software, as it necessitates specialized personnel (A. H. Jerónimo, et al., 2024).

The results are exported into their relevant JSON format which gives it a semi-structured datasets. Typically the source code lies in Github can be used to for scanning using the runners, actions and events. We would elaborate theese into 3 different set of functionalities:

1. SAST Scan achieved using GHAS CodeQL with Github runners and actions
2. Secret Scanning achieved using GHAS Secret Scanning using inbuilt Standard Patterns

3. DAST Scan achieved using a combination of Burp Enterprise Edition and OWASP AppScan DAST Github Action


2. Github (SCM) and Github actions initiates scanning on Push event
4. Remediated dismissed alerts measured in GNN
3. Secret Scanning + SAST + DAST scanning alerts created/reopened
5. SSDLC Mitigation
If policy failing or critical alerts re-opened
If no policy failing or critical alerts re-opened
1. Developers initiated code change in resolving alerts to address the vulnerabilities
7. Release/deploy changes
6. Governance Gate Check

3.4.1 Flowchart Table showing the overall architecture of research design process

Here, as stated in step 1 when a developer commits and pushed the source code into Github (as considered as SCM), the code gets scanned with Github actions with on push event in step 2. The source code gets analyzed by three difference scanning engines as mentioned above GHAS secret scanning, CodeQL for SAST scan and a combination of OWASP ZAP scan DAST and BURP Enterprise Edition ash shown in step 3. Following which, provided vulnerabilities exist in code number of policy failing or critical alerts are created which needs to be remediated. Developers performs remediation on these alerts some which are fixed and the alerts are auto closed and some are dismissed as it is. Step 4 shows that the dismissed alerts are run against the GNN model and any critical or high alerts that are incorrectly dismissed are actioned and re-opened and fed back to SSDLC team for further investigation and analysis as shown in step 5.Step 6 and step 7 are just subsequent steps that shows once the code is free of any policy failing or ciritcal vulnerabilites, the code then is compliant for release or deployment.

Elaborating on step 3, GitHub Actions provides multiple ways to trigger Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) scans as part of my CI/CD pipeline. These security scans can be automatically initiated through various events and conditions to ensure comprehensive security coverage throughout the software development lifecycle.

Trigger Mechanisms:

1. *Push Events*: Push-based triggers are the most common method for initiating security scans. Workflows execute whenever code is pushed to specified branches:

```
text
on:
  push:
    branches:
      - main
      - develop
      - release/*
    paths:
      - '**/*.java'    # Only trigger for Java/C# files
      - '**/*.js'      # Only trigger for JavaScript/Python files
```

This configuration ensures that SAST scans run immediately when new or added code is committed to critical as in release branches that would be ideal candidiate for SSDLC policy enablement, enabling early vulnerability detection before code reaches production.

2. *Pull Request Events*: Pull request triggers are essential for security gate enforcement, running scans before code merges into main branches:

```
text
on:
  pull_request:
    types: [opened, synchronize, reopened]
    branches:
      - main
      - production
```

Security scans triggered on pull requests assists to prevent vulnerable code from entering the main codebase without proper compliant scanning. Teams can configure workflows to block merging until security scans pass, implementing security as a mandatory gate.

3. *Scheduled Triggers*: Scheduled scans provide continuous security monitoring independent of code changes:

```
text
on:
  schedule:
    - cron: '0 2 * * *'    # Daily at 2:30 AM
    - cron: '0 0 * * 0'    # Weekly on Sundays
```

Regular scheduled scans are particularly valuable for detecting newly discovered vulnerabilities in existing code and dependencies.

4. *Manual Dispatch*: Manual triggers allow on-demand security scanning for ad-hoc security assessments:

```
text
on:
  workflow_dispatch:
    inputs:
      scan_type:
        description: 'Type of security scan'
        required: true
        default: 'both'
        type: choice
        options:
          - sast
          - dast
          - both
```

This provides flexibility for security teams to initiate comprehensive scans when needed without waiting for scheduled runs. For our use case, we would employ the use of push events and manual dispatch.

CodeQL Integration

GitHub's native CodeQL provides comprehensive SAST capabilities with multiple trigger options:

```
text
name: "SAST-CodeQL Analysis"

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]
  schedule:
    - cron: '30 1 * * 4'  # Weekly on Thursday

jobs:
  analyze:
    name: Analyze
    runs-on: ubuntu-latest
    permissions:
      actions: read
      contents: read
      security-events: write

    strategy:
      matrix:
        language: ['java', 'javascript', 'python']

    steps:
    - name: Checkout repository
      uses: actions/checkout@v4

    - name: Initialize CodeQL
      uses: github/codeql-action/init@v3
      with:
        languages: ${{ matrix.language }}

    - name: Perform CodeQL Analysis
uses: github/codeql-action/analyze@v3
```

DAST Tool - OWASP ZAP Integration

OWASP ZAP provides comprehensive DAST scanning capabilities that can be triggered through GitHub Actions:

```
text
name: DAST Security Scan

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
  schedule:
    - cron: '0 3 * * 1'  # Weekly on Monday at 3 AM

jobs:
  dast-scan:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: ZAP Baseline Scan
      uses: zaproxy/action-baseline@v0.7.0
      with:
        target: 'https://my-staging-app.com'
        rules_file_name: '.zap/rules.tsv'
        cmd_options: '-a'

    - name: Upload ZAP results
      uses: actions/upload-artifact@v3
      if: always()
      with:
        name: zap-reports
        path: report_html.html
```

Portswigger DAST tool Burp EE

Portswigger DAST integration provides enterprise-grade dynamic scanning:

```
text
name: Portswigger DAST Scan

on:
  deployment_status:
  schedule:
    - cron: '0 4 * * *'  # Daily at 3:30 AM

jobs:
  burpee-dast:
    runs-on: ubuntu-latest
    steps:
    - name: BurpEE DAST Scan
      uses: BurpEE/dast-github-action@v1.0.0-beta
      env:
        PW_APIKEY: ${{ secrets.PW_ONE_API_KEY }}
      with:
        command: web
        config: ./dast_config.yaml
        base_url: my-burpee-instance.com
        environment_id: ${{ secrets.ENVIRONMENT_ID }}
```

Advanced Trigger Configurations

Conditional Execution: Conditional triggers provide fine-grained control over when scans execute:

```
text
name: Conditional Security Scans

on:
  push:
  pull_request:

jobs:
  security-analysis:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: 0

    - name: Check for security-relevant changes
      id: changes
      uses: dorny/paths-filter@v2
      with:
        filters: |
          security:
```

```
        - 'src/**/*.java'
        - 'src/**/*.js'
        - '**/pom.xml'
        - '**/package.json'

    - name: Run SAST Scan
      if: steps.changes.outputs.security == 'true'
      run: |
echo "Running SAST scan for security-relevant changes"
```

In our thesis, we would use a combination of OWASP ZAP and BURPEE as DAST scanning tool leveraging via Github Actions and setup advanced configuration scanning as desired.

GHAS Secret Scanning Flow

GitHub Advanced Security provides comprehensive secret scanning capabilities that automatically detect and prevent the exposure of sensitive credentials across repositories. The secret scanning flow operates through multiple integrated stages to ensure comprehensive security coverage.

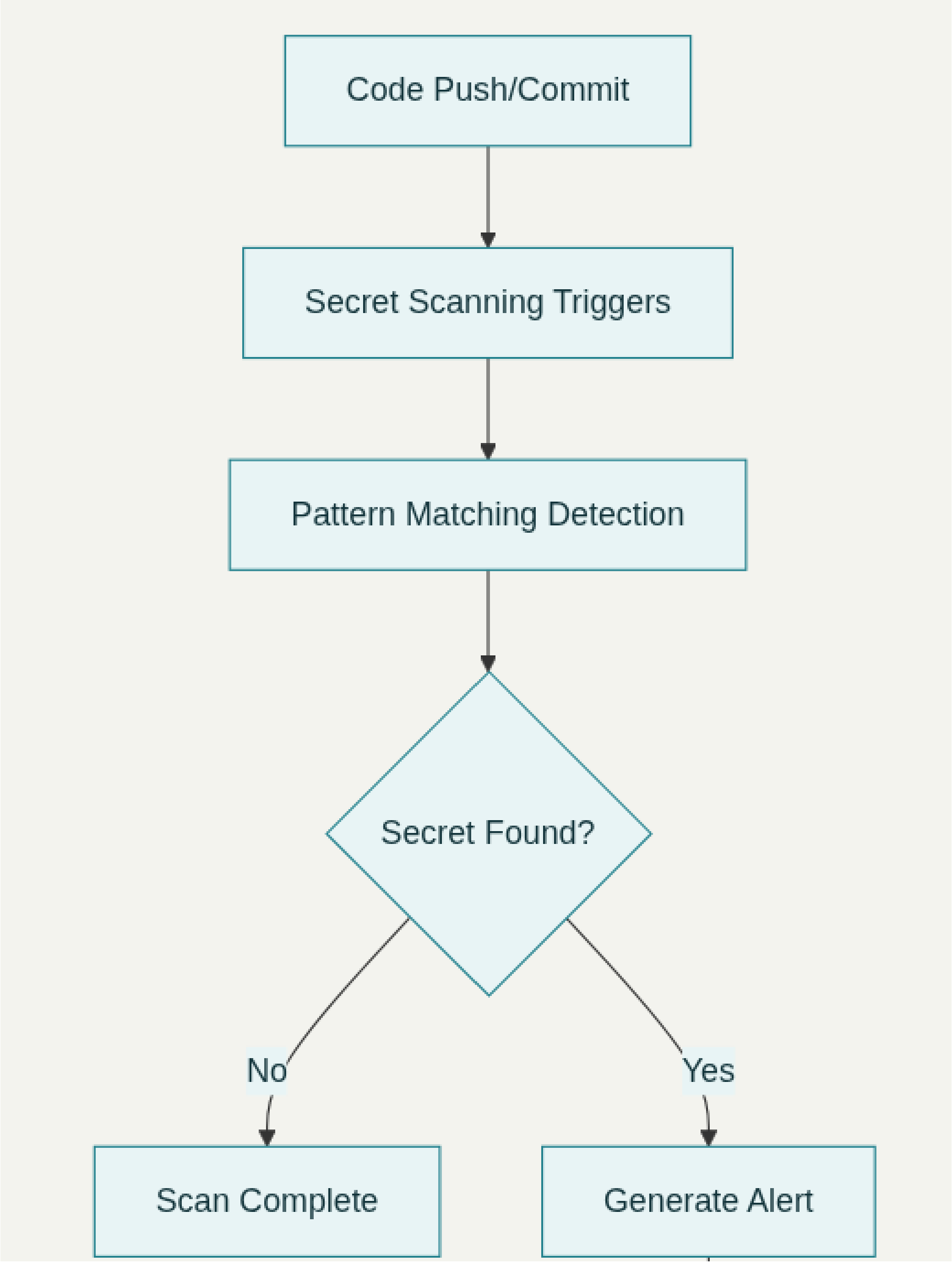


Table 3.4.2 Secret Scanning in GHAS flow

Detection and Scanning Process:

Automatic Scanning Triggers

GHAS secret scanning activates automatically when particular events gets created in our selected repositories:

Push events: Every commit pushed to any branch triggers an immediate scan

Pull request creation: New pull requests automatically undergo secret scanning

Repository enablement: When secret scanning is first enabled, GitHub performs a complete historical scan of the entire Git history across all branches

Scheduled scans: Periodic full scans detect newly supported secret patterns in existing content.

The scanning process examines multiple content types beyond just source code: All files in the repository including documentation, configuration files, and README files, issue descriptions and comments (both open and closed historical issues), pull request titles, descriptions, and comments, GitHub Discussions content, Wiki pages

Pattern Detection Methods

GitHub employs sophisticated pattern matching techniques to identify secrets. Regular expression matching ensures that the engine scans for known secret formats using precise regex patterns, partner or the in built patterns snsures utilization of well defined patterns provided by service providers through GitHub's partner program. This can be any different provider secret scanning rules like Github’s own personal access token, AWS access key, Google API tokens etc and finally the custom patterns which empowers enterprises to define their own secret patterns specific to their needs. This is the most prevalent feature of Github Advanced Security which allows SSDLC team to develop their own ruleset definitions to scan across repositories and look for secrets that are more organisation development culture specific. The majority of the bulk alerts can be obtained for a matured custom secret scanning rule definition program for example bearer tokens, oauth tokens, secret key, shared password formats etc. GHAS also provides the capability of AI powered generic detection which uses copilot powered secret scanning which uses machine learning to detect generic secrets and unstructured credentials. This adds value but odd are good to get a good amount of False Positives but the model can get better with feedback and loop based learning leveraging generative models. There is also entropy analysis which identifies high-entropy strings that may represent randomized tokens or keys.

In our thesis, we would employ a combination of all the secret scanning methods defined with an overall secret scanning alerts generating from custom defined rules which are always organisation specific above to obtain a variety of secret scannig alerts comprising of different categories.

Why GNN model?

Graph Neural Networks (GNNs) represents their usefulness shift in analyzing application security data because they naturally align with the inherently interconnected structure of security alerts and their surrounding context. When security scanners like SAST, DAST, and secret scanning tools generate JSON-based alert data, they create a rich ecosystem of relationships that traditional machine learning approaches struggle to capture effectively. The JSON output from these tools doesn't just contain isolated alert records but instead, it is embedded with a complex web of connections between alerts, code locations, files, commits, authors, functions, vulnerability types, and remediation history that forms a natural graph structure. The power of GNNs in the detection of incorrect alert dismissals becomes apparent when considering how these models can leverage the associated information as well as the relational context which would be otherwise impossible to capture through traditional feature engineering approaches. When a security alert is dismissed incorrectly, the evidence often lies not only in the alert's individual characteristics but also in its relationship to surrounding code structures, the history of similar alerts in nearby functions, the track record of the developer/analyst who dismissed it, and the broader patterns of vulnerability management within that codebase. GNNs excel at aggregating this contextual information through message passing mechanisms which allows each alert to gather evidence from its neighborhood in the form of graph, learning to recognize when dismissals contradict the evidence provided by related alerts, code structures, and historical patterns.

The architectural flexibility of GNNs makes them particularly well adapted for the heterogeneous nature of application security data. Modern variants like Relational Graph Convolutional Networks (R-GCN), Heterogeneous Attention Networks (HAN), and Heterogeneous Graph Transformers (HGT) can explicitly model different types of nodes and edges though this is beyond the study of this thesis but it is worth to mention that the capability lies in respecting the semantic differences between datasets like for example, dataflow relationships and code ownership connections. This allows the model to learn specialized aggregation functions for different types of relationships while maintaining a unified framework for reasoning across the entire security landscape. Furthermore, inductive GNN architectures like GraphSAGE and Graph Attention Networks enable the model to generalize to new repositories and services without requiring retraining, a crucial capability when dealing with the dynamic nature of software development environments. This to conclude this, we can aruguably state that the usage of Graph Neural Networks represent a natural and powerful approach for analyzing application security JSON data for incorrect alert dismissal review because they align with the fundamentally relational nature of security information, leverage the rich contextual signals embedded in scanner outputs, and provide the flexibility to handle the heterogeneous, dynamic, and interconnected characteristics of modern software security landscapes. Their ability to learn from both individual alert features and the broader patterns of relationships surrounding each alert makes them uniquely suited to identify the subtle indicators that distinguish legitimate security decisions from potentially dangerous oversights that leads to open risk.

GNN Workflow:

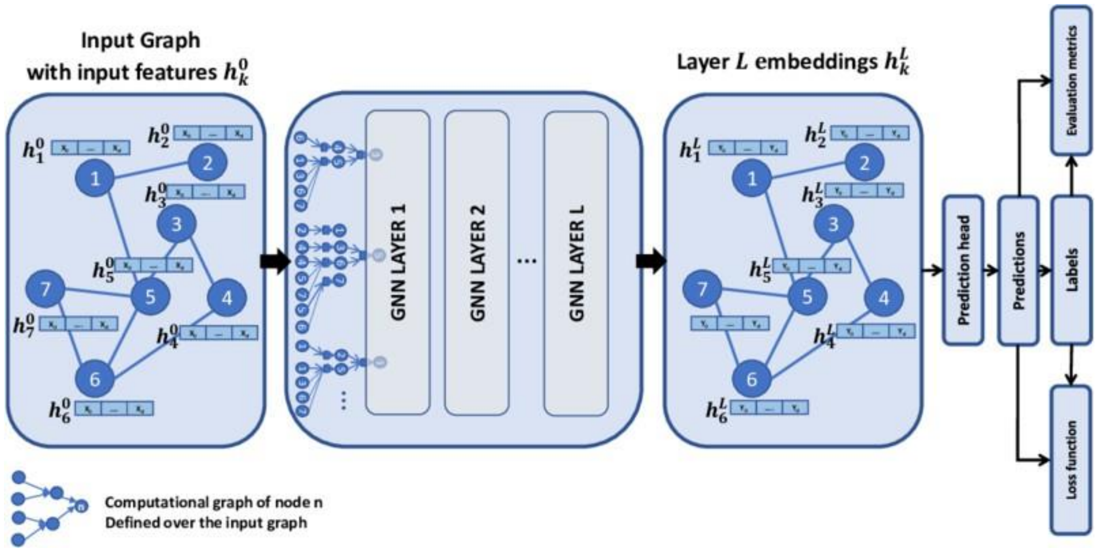


Figure 3.4.2 Image showing the GNN predictor model

Steps typically include:

- Parse JSON to hierarchical graph
- Extract features from nodes and edges
- Build graph with features
- Run graph through the GNN (GNNm) model
- Output prediction

Graph Neural Networks can be well-suited for analyzing resolved alerts that were dismissed incorrectly. This represents a powerful application of GNNs for retrospective analysis and false negative detection in security operations. The approach revolves around reanalyzing historical alert data that was previously dismissed or resolved as benign, but may have contained genuine security threats that were missed during initial triage which marks a significant risk if exploitation. GNNs excel at this because they can identify subtle patterns and relationships that human analysts might overlook during high-pressure incident response.The system builds comprehensive graphs from resolved alerts, incorporating multiple data dimensions as defined below:

json

```
{
  "dismissed_alert": {
    "id": "DISMISSED-2024-089",
    "original_severity": "MEDIUM",
    "dismissal_reason": "False positive - development environment",
    "analyst_decision": "DISMISSED",
    "dismissal_timestamp": "2024-09-15T14:30:00Z",
    "related_events": [
            {"type": "network_connection", "timestamp": "2024-09-15T14:25:00Z"},
            {"type": "file_access", "timestamp": "2024-09-15T14:28:00Z"},
            {"type": "privilege_escalation", "timestamp": "2024-09-15T14:32:00Z"}
          ],
          "contextual_indicators": {
            "user_behavior_baseline": "deviation_score: 0.7",
            "temporal_pattern": "outside_normal_hours: true",
            "geographic_anomaly": "unusual_location: true"
          }
        }
}
```

Below are the possible Graph Node Types for Retrospective Analysis:

- Dismissed Alert Nodes: Original alerts marked as false positives or benign. Our study would be based on this node relation.
- Temporal Context Nodes: Time-based relationships showing alert sequences
- User Behavior Nodes: Historical patterns of user activities surrounding dismissed alerts
- Infrastructure Nodes: Systems, networks, and assets involved in dismissed incidents
- Outcome Nodes: Subsequent security events that may validate initial dismissals

Advanced GNN techniques for false negative detection using Temporal Graph Neural Networks can be leveraged as GNNs can analyze temporal sequences to identify delayed attack patterns that weren't apparent during initial analysis:

```
python
# Temporal graph construction for retrospective analysis
temporal_edges = [
    {
        "source": "dismissed_alert_001",
        "target": "subsequent_breach_045",
        "edge_type": "temporal_correlation",
        "time_delta": "72_hours",
        "correlation_strength": 0.85
    }
]
```

Sample code snippet:

```
function JSON_to_Graph(json_data):
    # Recursively traverse JSON to build nodes and edges
    nodes = []
    edges = []
    def traverse(obj, parent=None):
        for key, value in obj.items():
            node_id = create_node_id(key, value)
            nodes.append({ 'id': node_id, 'features': extract_features(key, value) })
            if parent is not None:
                edges.append({ 'source': parent, 'target': node_id })
            if is_object(value):
                traverse(value, node_id)
    traverse(json_data)
    return nodes, edges

function Extract_Features(key, value):
    # Feature engineering for each node/field
    features = []
    features.append(encode_key(key))
    features.append(encode_type(value))
    if is_scalar(value):
        features.append(encode_value(value))
    return features

function Build_Graph(nodes, edges):
    # Prepare data structures for GNN input (e.g., adjacency list/matrix)
    X = [node['features'] for node in nodes]       # Node features
    edge_index = [[edge['source'], edge['target']] for edge in edges] # Edges
    return X, edge_index

function Predict(json_input, model):
    nodes, edges = JSON_to_Graph(json_input)
    X, edge_index = Build_Graph(nodes, edges)
    prediction = model.predict(X, edge_index)
    return prediction

# Usage Example
json_data = LOAD_JSON('data.json')
```

```
model = LOAD_GNNM_MODEL('gnnm_model')
result = Predict(json_data, model)
PRINT(result)
```

Actual Implementation Flow:

To train a Graph Neural Network (GNN) on JSON objects, the first and foremost step to do is to convert the JSON data into a graph structure that can be processed by the GNN. JSON objects can represent various types of data, as such the conversion process alwayss depend on the specific structure of the JSON data. Graph neural networks (GNNs) are neural models that capture the dependence of graphs via message passing between the nodes of graphs (Jie Zhou, et.al., 2020). The JSON data structure is defined below in the data analysis section. For GNN model the primary label used is the incorrectly_dismissed (1) vs correctly_dismissed (0) for closed alerts while derived by linkage to later incidents or validated re-opened alerts. In addition to that for the auxiliary heads , I have the options to use severity_reassessment, exploitability factor, business_impact, and re-open_probability for multi-task learning. For the practicality starting point, the model is started with alert, code_file, endpoint, service, analyst, incident nodes and the edges among them while later expanding to telemetry and intel as data matures. In order to balance the graph size with signal density, it was preferred to use high quality edges (causal/temporal) over exhaustive but noisy linkage. For the preservation of interpretability, the store edge-level evidence (timestamps, reasons, IoC matches) and surface the top-k subgraphs used per prediction for analyst review.

Below is one such process of how to define the approach to this problem using PyTorch and the PyTorch Geometric library, which is a popular library for GNNs.

### Step 1: Installation of Required Libraries

First, to make sure all the necessary libraries and dependencies installed:

```bash
pip install torch torch-geometric
```

### Step 2: Conversion of JSON to Graph

Similar to the dataset collected here, the JSON objects represent nodes and edges in a graph. For example, let us consider the following JSON structure similar to our dataset:

```json
{
   "nodes": [
      {"id": 1, "features": [0.1, 0.2]},
      {"id": 2, "features": [0.3, 0.4]},
      {"id": 3, "features": [0.5, 0.6]}
   ],
   "edges": [
      {"source": 1, "target": 2},
      {"source": 2, "target": 3}
   ]
}
```

~~~
```
~~~

Next to convert this JSON into a graph structure as follows:

~~~
```python
import json
import torch
from torch_geometric.data import Data

# Example JSON data
json_data = '''
{
    "nodes": [
        {"id": 1, "features": [0.1, 0.2]},
        {"id": 2, "features": [0.3, 0.4]},
        {"id": 3, "features": [0.5, 0.6]}
    ],
    "edges": [
        {"source": 1, "target": 2},
        {"source": 2, "target": 3}
    ]
}
'''

# Parsing of JSON
~~~

```
data = json.loads(json_data)

# Extract nodes and edges
nodes = data['nodes']
edges = data['edges']

# Create node features matrix
x = torch.tensor([node['features'] for node in nodes], dtype=torch.float)

# Create edge index matrix
edge_index = torch.tensor([[edge['source'], edge['target']] for edge in edges], dtype=torch.long).t().contiguous()

# Create PyTorch Geometric Data object
graph_data = Data(x=x, edge_index=edge_index)

print(graph_data)
```

### Step 3: Define a GNN Model

Next, define a simple GNN model using PyTorch Geometric. Here, we will use a Graph Convolutional Network (GCN) as an example:

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
    def _init_(self, input_dim, hidden_dim, output_dim):
        super(GCN, self)._init_()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

# Initialize the model
input_dim = 2  # Number of features per node
hidden_dim = 4
output_dim = 2  # Number of classes
model = GCN(input_dim, hidden_dim, output_dim)

# Print the model architecture
print(model)
```

### Step 4: Training the GNN Model

Now, to train the GNN model using the graph data that is just created:

```python
# Define optimizer and loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

# Dummy labels for training (assuming binary classification)
y = torch.tensor([0, 1, 0], dtype=torch.long)

# Training loop
model.train()
for epoch in range(100):
    optimizer.zero_grad()
    out = model(graph_data.x, graph_data.edge_index)
    loss = F.nll_loss(out, y)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

### Step 5: Evaluate the Model

After training, comes the task of evaluating the model based on data:

```python
```

```
model.eval()
pred = model(graph_data.x, graph_data.edge_index).argmax(dim=1)
print(f'Predicted classes: {pred}')
```

For the GNN-based retrospective security alert analysis model, the node and feature composition is structured to capture the technical and organizational context for each alert and its relationships.

Node Classes

Alert Nodes: These are the core nodes that represents the individual security scanning alerts from the training dataset. Each node is annotated with both raw attributes and derived features.

File Nodes: This represents the unique source code files implicated in at least one alert. Each file node may be linked to multiple alerts and optionally carries simplified technical features (e.g. code complexity).

Tool Nodes: Here each distinct scanning tool (e.g., GHAS, CodeQL, BURP EE, ZAP) is a node where an alert node is connected to the tool that detected it.

Features on Alert Nodes (per alert using and/or based on the given JSON fields) are

categorical encoded features (LabelEncoded and/or Embedded), numerical, float or integer features and engineered derived features. These can be a list of tool, alert_type, severity, dismissal_reason, code_owner_team, deployment_environment, business_criticality, confidence (continuous, e.g., 0.87), line_number (int), prior_alerts_same_file (int), code_complexity (int), test_coverage (float, 0-1), subsequent_alerts (int). For the derived features we simply used formulae to define such as severity_confidence_product = (severity_encoded + 1) × confidence, risk_score = (severity_encoded + 1) × confidence × (prior_alerts_same_file + 1).The metadata (tracked or label Y) is defined on is_genuine_missed_threat (for training label) and alert_id, tool, file_path, repository, branch, analyst_id, cwe_id (reference only).

Edges (Relationships) and their features are represented by alert → file (“found_in”) where every alert node is connected to the file node where the finding occurred, alert → tool (“generated_by”) where every alert node connects to its source tool node, alert → alert (“similar_to”) where it connects alerts found by the same tool or with similar severity, for local message-passing/similarity and finally file → alert (“contains”) where it uses reverse direction which is commonly included for bidirectional message passing.

The overall Graph Structure Summary is defined as follows

Nodes: Alerts (main), files, tools

Features: 15–20 features per alert node (combination categorical/numerical/derived)

Edges: Alert–file, alert–tool, alert–alert (similarity), and their inverses for message passing

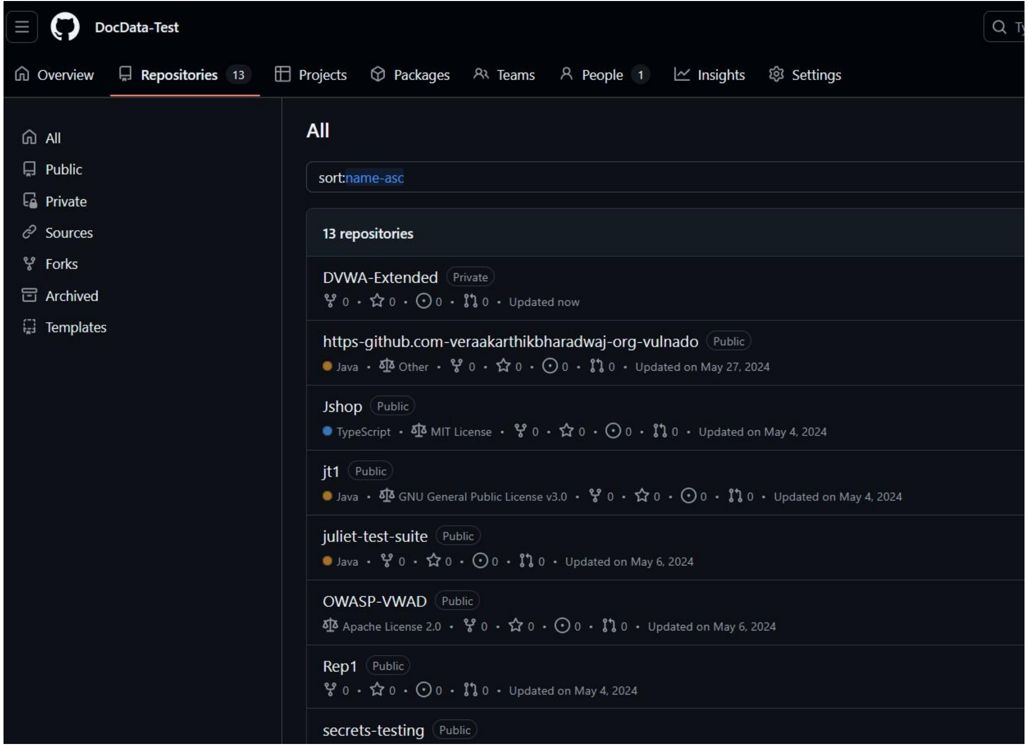
Train labels: is_genuine_missed_threat (binary) for classification

This composition enables the GNN to learn both individual alert risk factors and context/relationship patterns that signal incorrectly dismissed (missed) threats, using both the direct features of a given alert and information propagated via the graph structure.

Once we have a acceptable desired outcome for developer assisted remediation and GNN prediction on policy failing alerts incorrectly closed, we can propose this into a standard CI/CD pipeline to help alleviate risks associated to bug exploits.We can plug this process into the pipeline synchronously or asynchornously depending on the usage and then make process of automated self remediations of policy failing alerts in SSDLC process more robust as compared to hours of manual assessments.

3.5 Population and Sample

The complete sets of alert data acquired post scanning the repositories in Github. In Github, I have created an entire separate org to store all the repositores that would sever as population for the thesis study. The github org is set to public for public view: https://github.com/DocData-Test



sort:name-asc

13 repositories

DVWA-Extended Private
0 · 0 · 0 · 0 · Updated now

https-github.com-veraakarthikbharadwaj-org-vulnado Public
Java · Other · 0 · 0 · 0 · 0 · Updated on May 27, 2024

Jshop Public
TypeScript · MIT License · 0 · 0 · 0 · 0 · Updated on May 4, 2024

jt1 Public
Java · GNU General Public License v3.0 · 0 · 0 · 0 · 0 · Updated on May 4, 2024

juliet-test-suite Public
Java · 0 · 0 · 0 · 0 · Updated on May 6, 2024

OWASP-VWAD Public
Apache License 2.0 · 0 · 0 · 0 · 0 · Updated on May 6, 2024

Rep1 Public
0 · 0 · 0 · 0 · Updated on May 4, 2024

secrets-testing Public

Figure 3.5 Image showing the population of repos used for the study

The full list of repositories used and their corresponding links are shown below:

1. DVWA-Extended: https://github.com/DocData-Test/DVWA-Extended

2. https-github.com-veraakarthikbharadwaj-org-vulnado: https://github.com/DocData-Test/https-github.com-veraakarthikbharadwaj-org-vulnado
3. Jshop: https://github.com/DocData-Test/Jshop
4. jt1: https://github.com/DocData-Test/jt1
5. juliet-test-suite: https://github.com/DocData-Test/juliet-test-suite
6. OWASP-VWAD: https://github.com/DocData-Test/OWASP-VWAD
7. Rep1: https://github.com/DocData-Test/Rep1
8. secrets-testing: https://github.com/DocData-Test/secrets-testing
9. SecretsTest: https://github.com/DocData-Test/SecretsTest
10. testvulDAST: https://github.com/DocData-Test/testvulDAST
11. testvulmoreSS: https://github.com/DocData-Test/testvulmoreSS
12. testVulSAST: https://github.com/DocData-Test/testVulSAST
13. vucsa: https://github.com/DocData-Test/vucsa

All the above repositories are varied on different language stack like Java, Python, C#, Typescript to provide a more spread of the population alert data and hence the sample to be taken for the study. Now the sample of data would correspond to the policy failing code scanning alerts, secret scanning alerts and dynamic application security scanning alerts.

All of the alert generated from the tools will be in JSON format should contain '**Dismissed/Resolved reason**, '**Dismissed/Resolved by**' and '**Dismissed/resolved state**'. We will only focus on alert property that would signify the post mitigation state such as **'resolved/dismissed'**. For example, see below for a typical alert generated by CodeQL – SAST analysis

[{ "number": 4, "created_at": "2020-02-13T12:29:18Z", "url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/4", "html_url": "https://github.com/octocat/hello-world/code-scanning/4", "*state*": "*closed*", "fixed_at": null, *"dismissed_by": developer_id1,* "dismissed_at": null, "dismissed_reason": null, "dismissed_comment": null, "rule": { "id": "js/zipslip", "severity": "error", "tags": ["security", "external/cwe/cwe-022"], "description": "Arbitrary file write during zip extraction", "name": "js/zipslip" }, "tool": { "name": "CodeQL", "guid": null, "version": "2.4.0" }, "most_recent_instance": { "ref": "refs/heads/main", "analysis_key": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "category": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "environment": "{}", "state": "open", "commit_sha": "39406e42cb832f683daa691dd652a8dc36ee8930", "message": { "text": "This path depends on a user-provided value." }, "location": { "path": "spec-main/api-session-spec.ts", "start_line": 917, "end_line": 917, "start_column": 7, "end_column": 18 }, "classifications": ["test"] }, "instances_url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/4/instances" }, { "number": 3, "created_at": "2020-02-13T12:29:18Z", "url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/3", "html_url": "https://github.com/octocat/hello-world/code-scanning/3", "state": "dismissed", "fixed_at": null, "dismissed_by": { "login": "octocat", "id": 1, "node_id": "MDQ6VXNlcjE=", "avatar_url": "https://github.com/images/error/octocat_happy.gif", "gravatar_id": "", "url": "https://api.github.com/users/octocat", "html_url": "https://github.com/octocat", "followers_url": "https://api.github.com/users/octocat/followers", "following_url": "https://api.github.com/users/octocat/following{/other_user}", "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}", "starred_url":

"https://api.github.com/users/octocat/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/octocat/subscriptions", "organizations_url": "https://api.github.com/users/octocat/orgs", "repos_url": "https://api.github.com/users/octocat/repos", "events_url": "https://api.github.com/users/octocat/events{/privacy}", "received_events_url": "https://api.github.com/users/octocat/received_events", "type": "User", "site_admin": false }, "dismissed_at": "2020-02-14T12:29:18Z", "dismissed_reason": "false positive", "dismissed_comment": "This alert is not actually correct, because there's a sanitizer included in the library.", "rule": { "id": "js/zipslip", "severity": "error", "tags": ["security", "external/cwe/cwe-022"], "description": "Arbitrary file write during zip extraction", "name": "js/zipslip" }, "tool": { "name": "CodeQL", "guid": null, "version": "2.4.0" }, "most_recent_instance": { "ref": "refs/heads/main", "analysis_key": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "category": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "environment": "{}", "state": "open", "commit_sha": "39406e42cb832f683daa691dd652a8dc36ee8930", "message": { "text": "This path depends on a user-provided value." }, "location": { "path": "lib/ab12-gen.js", "start_line": 917, "end_line": 917, "start_column": 7, "end_column": 18 }, "classifications": [] }, "instances_url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/3/instances" }]

For Secret scanning from Github Advanced Security, we have **‘resolved state’, ‘resolved by’** and **‘resolution’**:

[{ "number": 2, "created_at": "2020-11-06T18:48:51Z", "url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/2", "html_url":

"https://github.com/owner/private-repo/security/secret-scanning/2", "locations_url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/2/locations", "state": "resolved", "resolution": "false_positive", "resolved_at": "2020-11-07T02:47:13Z", "resolved_by": { "login": "monalisa", "id": 2, "node_id": "MDQ6VXNlcjI=", "avatar_url": "https://alambic.github.com/avatars/u/2?", "gravatar_id": "", "url": "https://api.github.com/users/monalisa", "html_url": "https://github.com/monalisa", "followers_url": "https://api.github.com/users/monalisa/followers", "following_url": "https://api.github.com/users/monalisa/following{/other_user}", "gists_url": "https://api.github.com/users/monalisa/gists{/gist_id}", "starred_url": "https://api.github.com/users/monalisa/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/monalisa/subscriptions", "organizations_url": "https://api.github.com/users/monalisa/orgs", "repos_url": "https://api.github.com/users/monalisa/repos", "events_url": "https://api.github.com/users/monalisa/events{/privacy}", "received_events_url": "https://api.github.com/users/monalisa/received_events", "type": "User", "site_admin": true }, "secret_type": "adafruit_io_key", "secret_type_display_name": "Adafruit IO Key", "secret": "aio_XXXXXXXXXXXXXXXXXXXXXXXXXXXX", "push_protection_bypassed_by": { "login": "monalisa", "id": 2, "node_id": "MDQ6VXNlcjI=", "avatar_url": "https://alambic.github.com/avatars/u/2?", "gravatar_id": "", "url": "https://api.github.com/users/monalisa", "html_url": "https://github.com/monalisa", "followers_url": "https://api.github.com/users/monalisa/followers", "following_url": "https://api.github.com/users/monalisa/following{/other_user}", "gists_url": "https://api.github.com/users/monalisa/gists{/gist_id}", "starred_url":

"https://api.github.com/users/monalisa/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/monalisa/subscriptions", "organizations_url": "https://api.github.com/users/monalisa/orgs", "repos_url": "https://api.github.com/users/monalisa/repos", "events_url": "https://api.github.com/users/monalisa/events{/privacy}", "received_events_url": "https://api.github.com/users/monalisa/received_events", "type": "User", "site_admin": true }, "push_protection_bypassed": true, "push_protection_bypassed_at": "2020-11-06T21:48:51Z", "resolution_comment": "Example comment", "validity": "inactive" }, { "number": 1, "created_at": "2020-11-06T18:18:30Z", "url": "https://api.github.com/repos/owner/repo/secret-scanning/alerts/1", "html_url": "https://github.com/owner/repo/security/secret-scanning/1", "locations_url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/1/locations", "state": "open", "resolution": null, "resolved_at": null, "resolved_by": null, "secret_type": "mailchimp_api_key", "secret_type_display_name": "Mailchimp API Key", "secret": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-us2", "push_protection_bypassed_by": null, "push_protection_bypassed": false, "push_protection_bypassed_at": null, "resolution_comment": null, "validity": "unknown" }]

For DAST analysis from Github Advanced Security, we have **‘resolved state’, ‘resolved by’** and **‘resolution’**:

{
  "data": {

```
"issue": {
    "issue_type": {
        "type_index": null,
        "name": null,
        "description_html": null,
        "remediation_html": null,
        "vulnerability_classifications_html": null,
        "references_html": null
    },
    "display_confidence": null,
    "serial_number": null,
    "remediation_html": null,
    "description_html": null,
    "confidence": null,
    "severity": null,
    "path": null,
    "origin": null,
    "generated_by_extension": null,
    "evidence": [
        {
            "request_index": 0,
            "request_count": 1,
            "request_segments": [
                {
                    "data_html": null
```

```
                        }
                    ]
                },
                {
                    "response_index": 0,
                    "response_count": 1,
                    "response_segments": [
                        {
                            "data_html": null
                        },
                        {
                            "highlight_html": null
                        },
                        {
                            "data_html": null
                        }
                    ]
                }
            ]
        }
    }
}
```

For the training sample, the same sets of data would be fed back with ‘open status’ as true , fixed_at as ‘null’ and ‘reopened_by’ as SSDLC reviewer. These sets of data to be considered as correct sets and to be used as truth sets. The idea here is since the alert was incorrectly closed by the developer and hence the fixed node is empty, these alerts are found to be policy failing and as such needs to be fixed and so the alert state was moved to ‘open’ status.

3.6 Participant Selection

The list of vulnerable respositories were intentionally selected to get a good quanity of relevant security scanning alerts generated when they are plugged in with the SAST, secret scanning and DAST tools. In essence, for a quantitative research the choosing the appropiate participants is a key process that would lay down the quality of the study. My selected repositories were decided based on relevancy in terms of the quality of the alerts being produced, representing a large spread of alerts of different type like varitety of secrets scanning alerts of several types and code scanning alerts with variations in CWE category, generalizing the selections process as the alerts created in these repositories would be covering the entire dataset of the security scanning alerts as generated by the secret scanning, SAST and DAST scanning tools and finally able to apply the repeated procedures as in standardizing the process.

Possibly another tenet for a good quantitative research is the trustworthiness of the data based on which the study is being done. Fisher stated in his paper that the basic principle for measuring the quality of the reasearch is the amount of ‘rigour’ poured into it using the concepts of ‘reliability’and ‘validity’, while it is also true that the concept of generalisation tend to fit well with qualititative analysis (Fisher et al., 2016). Overall, in this study the repositories and the respective alerts generated should broadly represent the whole population sample for the totality of the vulnerabilities generated based on

OWASP TOP 10 2024 and the secret scanning alerts found using the standard templates. Moreover the design for the study being experimental and we trying to establish a causality in developer actions which effects the outcome and using the AI neural network models trying to correct the flow in a detective process. The methodologies that would use the relvant participants has to be a thorough constructive process which should always show a consistent output depending on the inputs provided. Hence, the process by which the security sccanning alerts produced in the repositories are dismissed needs to be of exacty the same nature to represent the developer action. This would provide the sanctity and assist is testing the hypotheis to understand if the neural network models be able to provide a usable feedback to SSDLC to catch and process the incorrectly closed alerts.

3.7 Instrumentation

In quantitative research, instrumentation ideally relates to the process of selecting, developing, and validating tools (instruments) to collect numerical data, such as questionnaires, tests, and observation checklists, to measure variables and answer research questions. The goal of instrumentation is to provide concrete steps that leads to the selection of the instrument and ensure that it is both valid (measures what it's supposed to) and reliable (produces consistent results) to gather high-quality, unbiased data for statistical analysis (Onwuegbuzie et al., 2010). In this paper, the tools used are vendor specific and well researched that follows industry standards of OWASP methodologies, NIST framework, NVD and vendor specific secret format.

Instrumentation is essentially the concepts with validations that concretely proves the effectiveness of the scanners and reliability when plugged in the CI/CD environment. For CodeQL, the mechanism does the build and query execution to produce precise SARIF and PR annotations. For OWASP ZAP/BURP EE, mechanism provides validatable ways by which my runtime app, auth, and proxy can be used to generate consistent crawls, perform auditing and active tests. For GHAS Secret Scanning, validation is achieved using org/repo policy, push protection, and alert routing to stop leaks pre-push and automate remediation.

1) **GHAS CodeQL**: CodeQL is a semantic static analysis engine that converts source code into a database and runs queries to find vulnerability patterns. Effective mechanism with accuracy means building accurate databases, selecting the right queries, and integrating results back into developer workflows. The different process by which we achieve the verifiable scanning is defined below which shows integrity of scanning is preserved as such.

Database extraction:

- Compiled languages: invoke real builds with commands or use autobuild to gather compile-time artifacts and generated sources into the CodeQL database.
- Interpreted languages: no build required, database includes the source and metadata present at extraction time.
- Indirect build tracing: when wrapping builds is complex, use begin/end tracing to capture build steps via environment variables in CI and then finalize the database.

Query execution:

- Executing official query language packs for supported languages, ability to add custom queries for organization-specific sinks/sources and comprehensive CWE coverage.
- Optimize for PRs: leverage incrementalization and matrix jobs to reduce time by focusing on changed code and relevant languages.

CI/CD pipeline (GitHub Actions):

- Triggers: push, pull_request, schedule for weekly deep scans.
- Permissions: security-events: write to publish SARIF.
- Steps: checkout, init (languages), autobuild or build, analyze, upload results.

Results ingestion/processing and application in governance gating:

- Upload SARIF to code scanning; surface annotations directly on diffs and block merges via required status checks and rulesets.
- Route alerts to developer teams with ownership and repository rules and apply SSDLC severity policies for break-the-build on critical CWEs.

Performance and fidelity:

- Ensure dependencies and generated code are present for compiled languages, preference for autobuild if standard, otherwise usage of explicit build commands.
- Cache CodeQL packs between runs and shared languages via strategy matrix.
- Using language-appropriate environment (JDK versions, compilers, SDKs) for accurate extraction.

Developer ergonomics:

- Navigating and locating custom .ql queries in a central repo and apply version query packs.
- Provide a minimal local repro path: codeql database create/analyze and VS Code extension for developers to iterate on findings.

Overall the standard flow for SAST scanning engine used in this study is to create databases for compiled languages, autobuild, and indirect build tracing; CodeQL analysis and upload via CLI/Actions to the repo from where the JSON data can be extracted.

2) OWASP ZAP/Burp EE

ZAP and Burp EE employs a DAST proxy that passively observes traffic from an hosted application and actively probes a running app. Good instrumentation should ensure deterministic crawls, authenticated coverage, safe test isolation, audit coverage and repeatable CLI/automation. In order to demonstrate that the scanning integrity is preserved, the entire process can be defined in the below steps:

Runtime target and environment:

- Always scanning on non-production and to seed stable test data, implenting isolation among test users and applying rate limits.
- Exposure of a predictable base/root URL (staging) with health checks.

Proxy and network:

- Configuration of browser/test runner in order to use ZAP/BURP EE as a MITM proxy (default localhost:8080) or launch browsers from ZAP/BURP EE to auto-configure trust and certs.
- Management and setup of certificates to avoid TLS issues, pinning ZAP root CA in test runners.

Authentication and sessions:

- Creating a scriptable auth (e.g., header-based Bearer, form login, OIDC flows). Enabling pass tokens via env or ZAP/BURP EE context; use ZAP/BURP EE auth scripts where it is necessary.

- Persisting sessions maintainenance and CSRF handling in context with added logged-in user for spidering/scanning.

Crawling and coverage:

- Employment and usage of traditional spider and AJAX spider where applicable, enabling limit scope with seed start URLs to reach deep routes for audit coverage.
- Persist context files and include/exclude regexes for deterministic coverage.

Scanning modes:

- Passive scanning is mostly safe and runs via proxy endpoints.
- Active scan os similar to an attack-like exercise as it target only in-scope apps with permission by applying configurable policy and enabling thresholds.

Automation and CI:

- Enabling CLI/headless based browser sessions, using the zap.sh -daemon with -config/-configfile for reproducible runs. Managing thread settings and API key via scan parameters.
- Utilising GitHub Actions via zaproxy/action-baseline or full scan actions for scheduled/PR runs and ultimately upload HTML/JSON reports via SARIF format. This is what we would be using to run our DAST scans.
- Parameterization of context, auth, base URL, spider time, scan policy and exporting findings as artifacts and fail the job on high-risk or policy–failing alerts as defined by SSDLC team.

Tuning and stability:

- Exclusion of logout URLs, unreachable or complex urls and destructive endpoints from active scans. This helps optimising preformance and prevents stalling of scans.
- Ensuring to throttle concurrency to avoid DoS via setting scanner.threadPerHost, hostPerScan.
- Vendor based benchmark of tool versions with later releases which improves categories (e.g., secure cookies, SQLi).

Developer feedback

- Map ZAP/BURP EE alerts to Jira/tickets by attaching evidence in case of issues.
- Maintaining an allowlist/false-positive registry and review scan policies periodically via SSDLC.

Overall the standard flow of DAST scan is ensuring that the ZAP/BURP EE is getting started, with proxies and auth configuration set-up, schedule passive and active scanning session and tag risk categories, enabling headless/daemon configuration via config file, utlising the GitHub Actions baseline/full scan usage; and executing the comparative benchmarking insights for version selection.

3) GHAS Secret Scanning

GHAS Secret Scanning helps detecting and preventing credential exposure across repos and orgs, with optional push protection and partner/custom notifications. In order to demonstrate that the scanning integrity is preserved, the entire process can be defined in the below steps:

Enablement and scope:

- Enabling the secret scanning and push protection at org/repo level and also including the custom and generic patterns as needed. For our study, we would enable the generic pattern and also some high confidence custom Rules.
- For Enterprises, application of policies via org/enterprise settings and repository rulesets, ensuring coverage for public and private repos.

Detection engines:

- Provider/partner patterns (GitHub-maintained service specific) and custom org patterns.
- Generic non-provider patterns and private key detection (enabling them separately but volume-limited).

Event wiring:

- Scanning gets triggered on push, PRs, and on initial enablement (full history scan and back-fill scan).
- Enabling push protection which blocks secrets at pre-receive, provides developer guidance and justifiable bypass with audit trail.

Alerting and routing:

- All the security scanning alerts populates under Security tab, this can also be configured to notify repo admins, security managers, owners, and commit authors regardless of watch settings.
- Leveraging CODEOWNERS, teams, and rulesets to route/require resolution before merge .
- API access for extraction to SIEM/SOAR approach as needed and integration with ticketing services.

Partner program integration:

- For supported provider patterns in public repos, GitHub notifies providers for validation and potential revocation/rotation.

Remediation workflow:

- Promoting rotate and revoke credentials bia SSDLC guidance, updating dependent systems, optionally scrub history as required once secret remediation is done and resolving the alert with notes for audit.
- Setting SLAs by secret type and repo visibility if needed, automation of ticket creation and owner assignment promoting accountability.

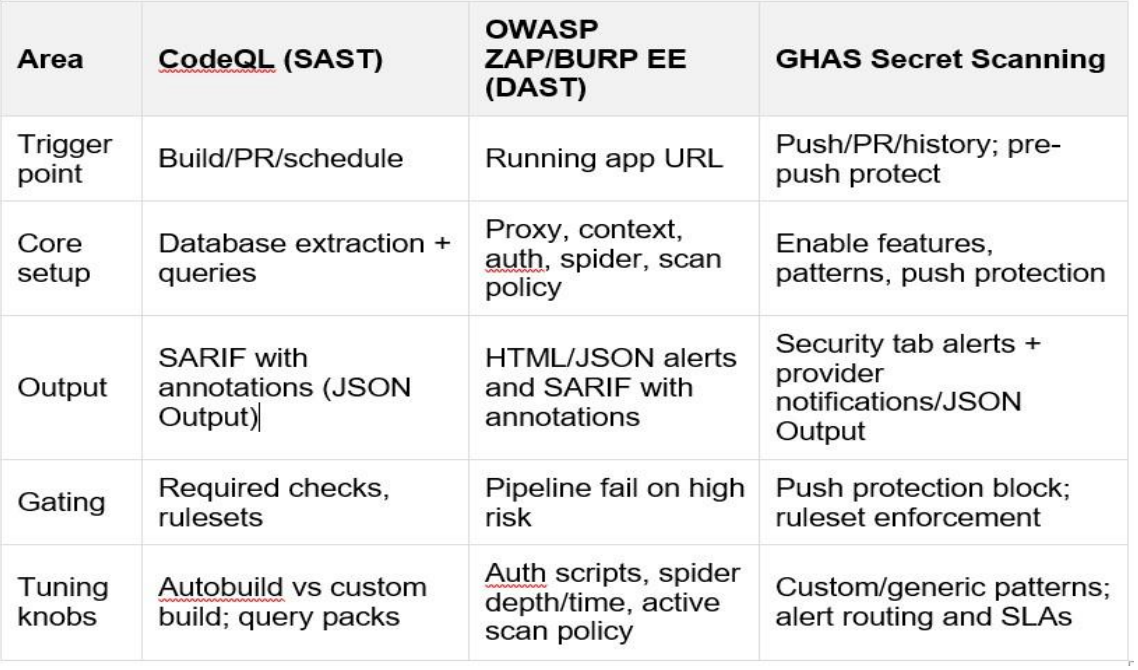
Prevention and developer UX:

- Educate with custom push protection messages, adding pre-commit hooks locally for immediate feedback.
- Define custom patterns for internal secrets/tokens and periodically review patterns and false positives as per SSDLC policies.

Monitoring and governance:

- Employing observation and tracking of trend metrics, MTTR, false-positive rate, org-level reports and coverage checks.
- Enforcement of rulesets that require secret scanning enabled and push protection on protected branches per SSDLC policies defined across enterprise.

Overall the GHAS Secret scanning flow is all about secret scanning andenabling features, supported patterns, alert views and notifications, partner program flow, enterprise oriented SSDLC lifecycle notes and push protection benefits.



Area	CodeQL (SAST)	OWASP ZAP/BURP EE (DAST)	GHAS Secret Scanning
Trigger point	Build/PR/schedule	Running app URL	Push/PR/history; pre-push protect
Core setup	Database extraction + queries	Proxy, context, auth, spider, scan policy	Enable features, patterns, push protection
Output	SARIF with annotations (JSON Output)	HTML/JSON alerts and SARIF with annotations	Security tab alerts + provider notifications/JSON Output
Gating	Required checks, rulesets	Pipeline fail on high risk	Push protection block; ruleset enforcement
Tuning knobs	Autobuild vs custom build; query packs	Auth scripts, spider depth/time, active scan policy	Custom/generic patterns; alert routing and SLAs

Table 3.7 showing side by side instrumentation focus of appsec tools used in this study

3.8 Data Collection Procedures

Data is to be primary generated using open source tools available for SAST, DAST and Secret scanning. We will sample through a range of 20-50 selected repositories in github in our org space and scan them as required either in synchronous fashion or asynchronously. There are certain conditions needed to be met to generate the alerts that would constitute sample datasets. For SAST tool depending on language of the source code we need to scan them i.e. for interpreted languages such as Javascript/Typescript we do not need to build them to scan but for complied languages we would need to build & compile the source code and then scan. This is as already explained and eventually post scanning we can export the JSON based output to perform our analysis with GNN model or any other ML models as required.

On the other hand, we would need to build, compile and deploy applications in web server to host so it can be scanned using DAST tool to generate DAST specific alert datasets where for secret scanning we can simply scan source codes to generate alerts.

Open source tools to be used for:

- SAST are CodeQL (Github Advanced Security), MobSF for mobile languages, Semgrep for PHP
- DAST are Zed attack Proxy(ZAP/BURP EE)
- Secret Scanning are GHAS Secret Scanning

All of these tools has the capability to generate alerts and allow it to be exported in JSON format – a structured data which would eventually be used for deep learning model analysis. Data collection is the foundation of effective security analysis. Each tool employs distinct methodologies for gathering, processing, and analyzing application data but all can be exported back to JSON format. Understanding these procedures enables proper instrumentation, accurate results interpretation, and comprehensive vulnerability coverage.

CodeQL converts the source code into queryable databases through a sophisticated extraction process that captures both syntactic and semantic program information.

For languages like C/C++, Java, C#, and Swift, CodeQL monitors the build process to capture comprehensive program semantics:

bash

```
# Standard database creation with build monitoring
codeql database create java-db --language=java --command="mvn clean compile"
# Autobuild for standard projects
codeql database create java-db --language=java
# CodeQL automatically detects Maven/Gradle and runs appropriate build commands
# Multi-language cluster creation
codeql database create multi-db --language=java,javascript,python --db-cluster
```

CodeQL generates data while scanning the source code repositories and the key data collections mechanisms involves build interception as in CodeQL intercepts compiler invocations to capture source files, dependencies, and compilation artifacts, AST generation as in creates abstract syntax trees with complete structural representation, semantic analysis as in captures name binding, type information, and cross-references between program elements and finally generated code inclusion as in includes code generated during build processes (annotations, preprocessors, etc.). For interpreted language, extraction works directly on source code without requiring compilation:

```
bash
# Python database creation (no build required)
codeql database create python-db --language=python --source-root=./src
# JavaScript with specific source filtering
codeql database create js-db --language=javascript \
--codescanning-config=.github/codeql-config.yml
```

Data capture includes source code analysis, metadata resolution and metatdata extraction. CodeQL databases undergo finalization to optimize query performance and ensure data integrity:

```
bash
# Manual finalization with cleanup
codeql database finalize my-database --no-cleanup
# Database analysis and results generation
```

```
codeql database analyze my-database codeql/java-queries:codeql-suites/java-security-extended.qls \
  --format=sarif --output=results.sarif
```

To sum up, CodeQL queries operate on the structured database using a declarative logic programming language and performs comprehensive SAST scanning based on QLpacks which is ultimately makes up the security scanning alerts and finally post remediation/dismissal be exported back to JSON format for this study.

For GHAS Secret scanning process, this employs multiple detection techniques to identify credentials and sensitive data across repositories and organizational assets. Scanning triggers and coverage is achieved using the comprehensive repository scanning and historical analysis which scours the entire git history across all branches and commits The real-time scanning is always triggered on push events, pull requests, and repository delta that includes all content types such as source code, documentation, issue comments, pull request discussions, Wiki pages.

Pattern detection methods usually invloves enabling the in-built standard detection rules or define custom secret scanning rules. There is also AI generic detections and non-provider patterns.

Sample provider pattern matching as shown below:

```
# Example AWS Access Key pattern
AKIA[0-9A-Z]{16}
# GitHub Personal Access Token pattern
gh[pousr]_[A-Za-z0-9_]{36}
# Generic database connection string
```

```
(mongodb|mysql|postgres)://[a-zA-Z0-9_\-\.]+:[a-zA-Z0-9_\-\.%]+@[a-zA-Z0-9_\-\.]+:[0-9]+/[a-zA-Z0-9_\-\.]+
```

AI generic detection is based on entropy analysis which identifies strings with high randomness typical of generated tokens and detects base64-encoded credentials and keys while the custom rules defined looks for organization-defined regex patterns for internal credential formats. Once the scanning is completed, the generated secret scanning alerts can be dismissed/resolved via developer actions which then is exported back to JSON based data for further analysis for this study.

For the OWASP ZAP/BURP EE data collection process, ZAP operates as an intercepting proxy that collects application data through passive observation and active reconnaissance and Burp Suite EE provides enterprise-scale automated scanning with sophisticated crawling and vulnerability detection capabilities. Though the former is a bit manual setup but BURP EE makes it easy in it semi-automated process of performing scanning, spidering and producing meaningful alerts. Data collection using ZAP approach usually invloves link extraction as in HTML parsing to identify hyperlinks, forms, and navigation elements, URL discovery as in building comprehensive site map through recursive link following and parameter identification as in query strings, form fields, and hidden inputs. Using the BURP EE, the scope is connected to site mapping as in comprehensive directory and endpoint discovery, form analysis which relates to input field identification, validation testing, submission workflows and authentication flows meaning login sequences, session management, access control testing.

```
# API-based spider initiation ZAP
curl
"http://localhost:8080/JSON/spider/action/scan/?apikey=KEY&url=https://target.com"
# Monitor crawl progress
```

```
curl http://localhost:8080/JSON/spider/view/status/?apikey=KEY&scanId=SCAN_ID
# Scan configuration for different application types BURP EE
crawl_strategy: "fastest"      # Static sites (6x-9x faster)
crawl_strategy: "most_complete" # Complex SPAs and stateful applications
// API definition import for comprehensive endpoint testing
{
  "scan_type": "api_only",
  "api_definition": "openapi.yaml",
  "authentication": {
    "type": "bearer_token",
    "token": "${API_TOKEN}"
  }
}
```

Pipeline Integration:

text

```
# Example comprehensive data collection pipeline
security_scan:
  parallel:
    - codeql_analysis:
        database_creation: "autobuild"
        query_packs: ["security-extended"]
    - secret_scanning:
        push_protection: enabled
        custom_patterns: organization_specific
    - dast_scanning:
        tool: "zap"
        spider_types: ["traditional", "ajax"]
        auth_context: "authenticated_user"
```

Scanned data creates security scanning alerts and is mapped across OWASP Top 10, PCI DSS, and custom compliance frameworks. There alerts further can be analysed/dismissed by developers with any available reason code, this dismissed datasets can be exported back to JSON format for the analysis of this study.

Data is exported to JSON format using the REST API endpoints that Github provides in their documentation

```
curl -L \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/DocData-Test/Jshop /code-scanning/alerts
```

We use a powershell script to export all the data related to code scanning alerts, secret scanning alerts and DAST alerts. The script makes a curl call and fetches all data by looping through the repo.

Each tool's data collection methodology serves complementary purposes in a comprehensive security testing strategy. CodeQL provides deep static analysis of source code semantics, secret scanning offers continuous monitoring for credential exposure, while ZAP and Burp Suite deliver runtime application behavior analysis. Effective implementation requires understanding each tool's data collection strengths and configuring them appropriately for the study of this specific application architecture and security requirements mimicking as close to a available devsecops workflow.

3.9 Data Analysis

Most of the alert generated from the tools will be in JSON format should contain ‘Dismissed/Resolved reason, ‘Dismissed/Resolved by’ and ‘Dismissed/ resolved state’. We will only focus on alert property that would signify the post mitigation state such as ‘resolved/dismissed’. For example see below for a typical alert generated by CodeQL – SAST analysis

[{ "number": 4, "created_at": "2020-02-13T12:29:18Z", "url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/4", "html_url": "https://github.com/octocat/hello-world/code-scanning/4", "*state*": "open", "fixed_at": null, "*dismissed_by*": null, "dismissed_at": null, "dismissed_reason": null, "dismissed_comment": null, "rule": { "id": "js/zipslip", "severity": "error", "tags": ["security", "external/cwe/cwe-022"], "description": "Arbitrary file write during zip extraction", "name": "js/zipslip" }, "tool": { "name": "CodeQL", "guid": null, "version": "2.4.0" }, "most_recent_instance": { "ref": "refs/heads/main", "analysis_key": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "category": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "environment": "{}", "state": "open", "commit_sha": "39406e42cb832f683daa691dd652a8dc36ee8930", "message": { "text": "This path depends on a user-provided value." }, "location": { "path": "spec-main/api-session-spec.ts", "start_line": 917, "end_line": 917, "start_column": 7, "end_column": 18 }, "classifications": ["test"] }, "instances_url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/4/instances" }, { "number": 3, "created_at": "2020-02-13T12:29:18Z", "url":

"https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/3", "html_url": "https://github.com/octocat/hello-world/code-scanning/3", "state": "dismissed", "fixed_at": null, "dismissed_by": { "login": "octocat", "id": 1, "node_id": "MDQ6VXNlcjE=", "avatar_url": "https://github.com/images/error/octocat_happy.gif", "gravatar_id": "", "url": "https://api.github.com/users/octocat", "html_url": "https://github.com/octocat", "followers_url": "https://api.github.com/users/octocat/followers", "following_url": "https://api.github.com/users/octocat/following{/other_user}", "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}", "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/octocat/subscriptions", "organizations_url": "https://api.github.com/users/octocat/orgs", "repos_url": "https://api.github.com/users/octocat/repos", "events_url": "https://api.github.com/users/octocat/events{/privacy}", "received_events_url": "https://api.github.com/users/octocat/received_events", "type": "User", "site_admin": false }, "dismissed_at": "2020-02-14T12:29:18Z", "dismissed_reason": "false positive", "dismissed_comment": "This alert is not actually correct, because there's a sanitizer included in the library.", "rule": { "id": "js/zipslip", "severity": "error", "tags": ["security", "external/cwe/cwe-022"], "description": "Arbitrary file write during zip extraction", "name": "js/zipslip" }, "tool": { "name": "CodeQL", "guid": null, "version": "2.4.0" }, "most_recent_instance": { "ref": "refs/heads/main", "analysis_key": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "category": ".github/workflows/codeql-analysis.yml:CodeQL-Build", "environment": "{}", "state": "open", "commit_sha": "39406e42cb832f683daa691dd652a8dc36ee8930", "message": { "text": "This path depends on a user-provided value." }, "location": { "path": "lib/ab12-

gen.js", "start_line": 917, "end_line": 917, "start_column": 7, "end_column": 18 }, "classifications": [] }, "instances_url": "https://api.github.com/repos/octocat/hello-world/code-scanning/alerts/3/instances" }]

For Secret scanning from Github Advanced Security, we have ‘resolved state’, ‘resolved by’ and ‘resolution’:

[{ "number": 2, "created_at": "2020-11-06T18:48:51Z", "url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/2", "html_url": "https://github.com/owner/private-repo/security/secret-scanning/2", "locations_url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/2/locations", "*state*": "resolved", "*resolution*": "false_positive", "resolved_at": "2020-11-07T02:47:13Z", "*resolved_by*": { "login": "monalisa", "id": 2, "node_id": "MDQ6VXNlcjI=", "avatar_url": "https://alambic.github.com/avatars/u/2?", "gravatar_id": "", "url": "https://api.github.com/users/monalisa", "html_url": "https://github.com/monalisa", "followers_url": "https://api.github.com/users/monalisa/followers", "following_url": "https://api.github.com/users/monalisa/following{/other_user}", "gists_url": "https://api.github.com/users/monalisa/gists{/gist_id}", "starred_url": "https://api.github.com/users/monalisa/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/monalisa/subscriptions", "organizations_url": "https://api.github.com/users/monalisa/orgs", "repos_url": "https://api.github.com/users/monalisa/repos", "events_url": "https://api.github.com/users/monalisa/events{/privacy}", "received_events_url": "https://api.github.com/users/monalisa/received_events", "type": "User", "site_admin":

true }, "secret_type": "adafruit_io_key", "secret_type_display_name": "Adafruit IO Key", "secret": "aio_XXXXXXXXXXXXXXXXXXXXXXXXXXXX", "push_protection_bypassed_by": { "login": "monalisa", "id": 2, "node_id": "MDQ6VXNlcjI=", "avatar_url": "https://alambic.github.com/avatars/u/2?", "gravatar_id": "", "url": "https://api.github.com/users/monalisa", "html_url": "https://github.com/monalisa", "followers_url": "https://api.github.com/users/monalisa/followers", "following_url": "https://api.github.com/users/monalisa/following{/other_user}", "gists_url": "https://api.github.com/users/monalisa/gists{/gist_id}", "starred_url": "https://api.github.com/users/monalisa/starred{/owner}{/repo}", "subscriptions_url": "https://api.github.com/users/monalisa/subscriptions", "organizations_url": "https://api.github.com/users/monalisa/orgs", "repos_url": "https://api.github.com/users/monalisa/repos", "events_url": "https://api.github.com/users/monalisa/events{/privacy}", "received_events_url": "https://api.github.com/users/monalisa/received_events", "type": "User", "site_admin": true }, "push_protection_bypassed": true, "push_protection_bypassed_at": "2020-11-06T21:48:51Z", "resolution_comment": "Example comment", "validity": "inactive" }, { "number": 1, "created_at": "2020-11-06T18:18:30Z", "url": "https://api.github.com/repos/owner/repo/secret-scanning/alerts/1", "html_url": "https://github.com/owner/repo/security/secret-scanning/1", "locations_url": "https://api.github.com/repos/owner/private-repo/secret-scanning/alerts/1/locations", "state": "open", "resolution": null, "resolved_at": null, "resolved_by": null, "secret_type": "mailchimp_api_key", "secret_type_display_name": "Mailchimp API Key", "secret": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-us2", "push_protection_bypassed_by": null, "push_protection_bypassed": false,

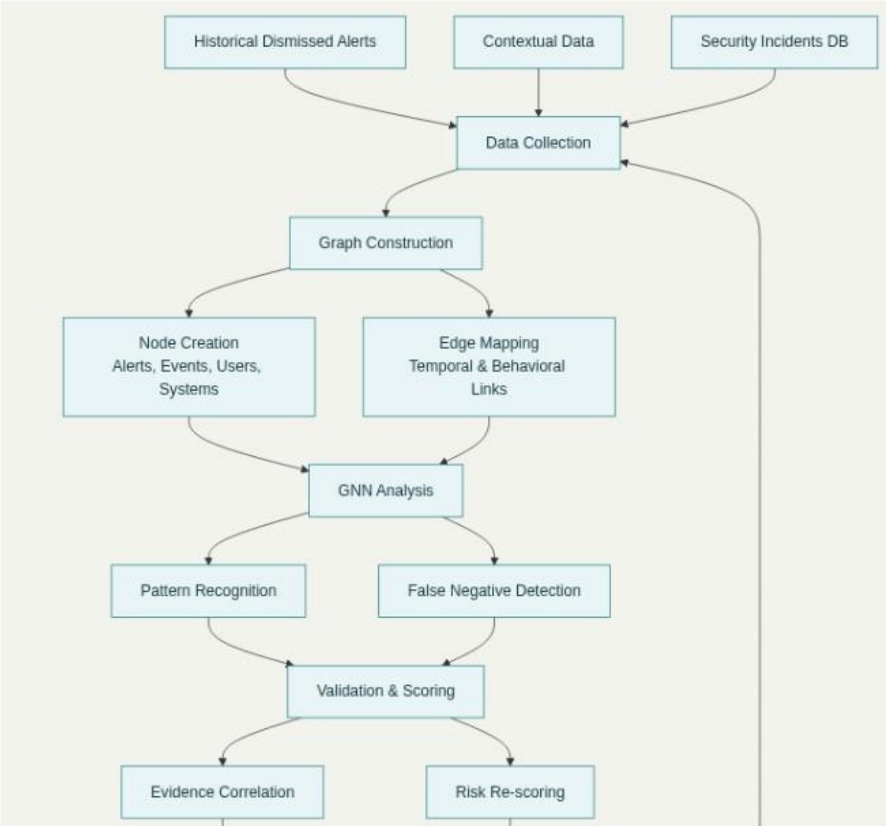
```
"push_protection_bypassed_at": null, "resolution_comment": null, "validity": "unknown"
} ]
```

For DAST analysis from Github Advanced Security, we have ‘resolved state’, ‘resolved by’ and ‘resolution’:

```
{
  "data": {
    "issue": {
      "issue_type": {
        "type_index": null,
        "name": null,
        "description_html": null,
        "remediation_html": null,
        "vulnerability_classifications_html": null,
        "references_html": null
      },
      "display_confidence": null,
      "serial_number": null,
      "remediation_html": null,
      "description_html": null,
      "confidence": null,
      "severity": null,
      "path": null,
      "origin": null,
"generated_by_extension": null,
"evidence": [
    {
        "request_index": 0,
        "request_count": 1,
        "request_segments": [
            {
                "data_html": null
            }
        ]
    },
    {
        "response_index": 0,
        "response_count": 1,
        "response_segments": [
            {
                "data_html": null
            },
            {
                "highlight_html": null
            },
            {
                "data_html": null
            }
        ]
```

```
                }
            ]
        }
    }
}
```

We will employ deep learning models to evaluate the triaging process in appec workflow and draw a comparative narrative. Models to be used are Graph neural network.



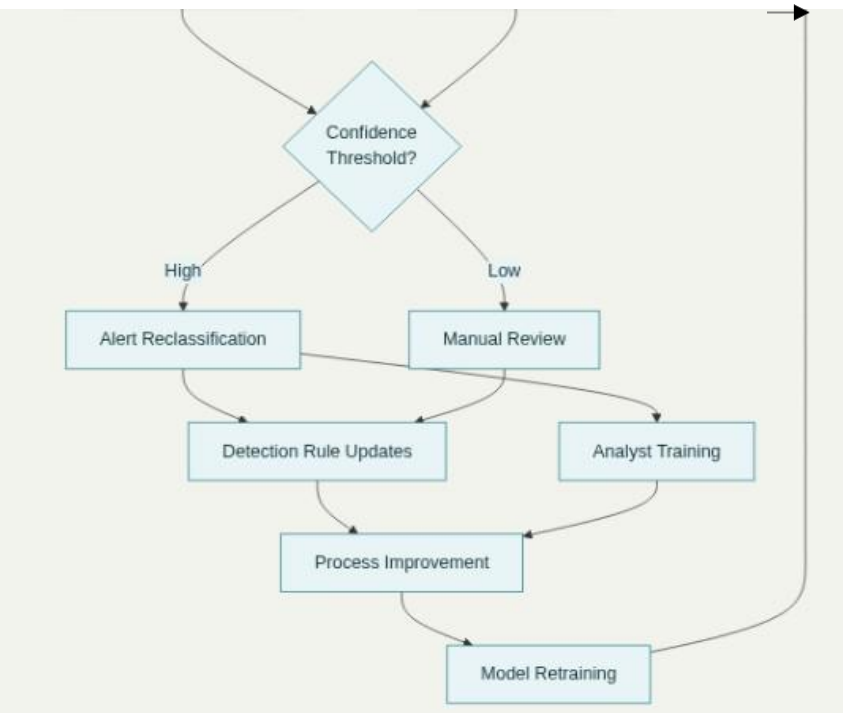


Figure 3.8 showing GNN Flow Diagram for Retrospective Analysis of Incorrectly Dismissed Security Alerts

Feature vector example sample (Tensor for Graph Neural Network model input)

Each alert node feature vector may look like (for a model with 15 features) as defined below [*tool_id, alert_type_id, severity_id, dismissal_reason_id, code_owner_team_id, deploy_env_id, confidence, line_number, prior_alerts_same_file, code_complexity, test_coverage, subsequent_alerts, severity_confidence_product, risk_score, is_public_endpoint*]

Target Labels Used: is_genuine_missed_threat (0 = correctly dismissed, 1 = incorrectly dismissed/genuine missed threat)

Demo sample for a single alert node:

tool = "CodeQL" → encoded as 2

alert_type = "SQL_INJECTION" → Encoded as 0 severity = "HIGH" → Encoded as 2

dismissal_reason = "FALSE_POSITIVE" → Encoded as 0

confidence = 0.89

line_number = 234

prior_alerts_same_file = 7

code_complexity = 56

test_coverage = 0.71

subsequent_alerts = 2

severity_confidence_product = 2 * 0.89 = 1.78

risk_score = 2 * 0.89 * 8 = 14.24

Total vector: [2, 0, 2, 0, ..., 1.78, 14.24, 1]

3.9 Research Design Limitations

Well one of the limitations applies on the number of repositories to be used for the study, the more repositories we feed to the NN model the more accurate it will get. Also, there is the SSDLC analyst training on the model once the model produces a result set that is not desirable. In this study, the datasets generated would all be from open source codes stashed in public Github repositories under GNU general public license. Tools to be used are from open source or demo version. Additionally, all sources that will be used in this study will be duly acknowledged. As with all other systems data integrity and handling bias in systems is of utmost importance. Historical security alert datasets suffer from inherent temporal bias, where alert patterns evolve rapidly as attack methods change, making older dismissed alerts potentially irrelevant for current threat landscape analysis, seasonal and organizational context changes affect alert generation patterns, creating temporal distribution skews that GNNs may incorrectly learn as attack signatures and data retention policies often result in incomplete historical records, with critical contextual information missing from older alerts (Kearney et al., 2023). The fundamental challenge lies in establishing reliable ground truth for "incorrectly dismissed" alerts as in our process the SSDLC analyst decision rationale may be undocumented or inconsistent, making it impossible to verify whether dismissals were truly incorrect, retrospective labeling bias may occur when researchers know the outcomes, potentially introducing confirmation bias in identifying false negatives and missing counterfactual evidence - we cannot know what would have happened if dismissed alerts had been investigated so this needs to be followed accurately (Daniel et al., 2024).

3.9 Conclusion

With this research, it is intended to demonstrate that a gap exists in the mitigation process for SSDLC/PRA team if developers are resolving the alerts without manual supervision from appsec review teams. To analyze the efficacy of detecting the closed/resolved alerts arising out of application security tools (SAST/DAST/Secret Scanning) which has the implications of being exploited if ignored. The said gap is assessed and an alternate framework is propose which identifies the problem and work on a solution to address. Previous literature survey will guide on the usage of deep learning techniques and upon further assessment, the efficacy of models will be established which can detect the relevant alerts that were wrongly dismissed as False Positive or ignored. In nutshell, this paper will propose:

- The utilization and efficacy of deep learning techniques GNN model in detection of wrongly marked policy failing security alerts as a part of developer-led mitigation practice is SSDLC lifecycle and report them to application security team for review.

- Propose a framework, which will assist application security teams to manage these detected alerts and review the relevant alerts.

Although is is completely understandable that scaling this process in enterprise would be challenging but the risk of exploitation of incorrectly dismissed security scanning alerts poses a significant problem that needs an automated, self-improving system that becomes more accurate over time, reducing both false negatives and false positives while increasing analyst productivity and organizational security effectiveness. This not only would reduce time, effort, cost but over the time mass cloud adoption would certainly lead to a decrease in cost in adopting the proposed approach as the world moves towards AI adoption. Even with the limitations, if the model can achieve around 85-95% efficiency, this can significantly boost confidence in enterprise in adopting an automated solution for the alert remediation process.

CHAPTER IV:
RESULTS

4.1 Research Question One

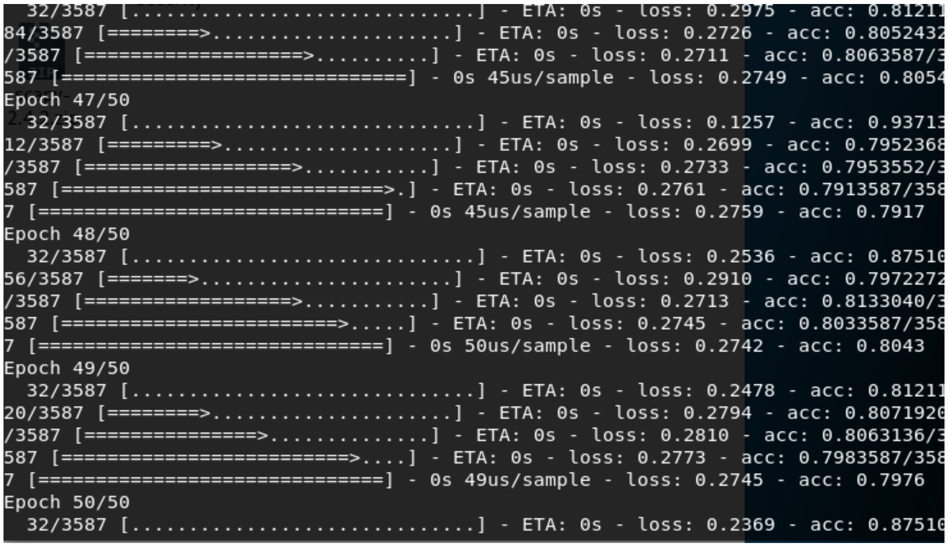
Once the environment is established and the repositories with a variety of source codes has been setup, the code utilising the Graph neural network model is invoked and it showed amazing results in consecutive runs. It was known that GNN models are particularly well-suited for JSON based dataset that is composed of analysis of resolved alerts that were dismissed incorrectly. This gave us an first hand approach of inception of a powerful application of GNNs for retrospective analysis and false negative detection in security operations. GNNs can then identify dismissed alerts that were actually part of larger, coordinated remediation workflow as too shown in SOC paper (Eckhoff et al., 2025):

```
json
{
  "campaign_analysis": {
    "dismissed_alerts_count": 23,
    "confirmed_campaign_indicators": 15,
    "attack_timeline": "45_days",
    "false_negative_rate": "65.2%",
    "recommended_actions": [
      "Retrain detection rules",
      "Update analyst training",
      "Implement enhanced correlation"
    ]
  }
}
```

In addition, the system could evaluate the quality of historical triage decisions and identify systematic biases: such as decision pattern analysis that is able to identify analysts who consistently miss certain attack types, contextual bias detection which led to discovery of environmental factors that lead to incorrect dismissals. All of these provide a huge benefit in terms of detection and proper remediation effort with increase precision. The model also highlighted specific areas where analyst education is needed in order to improve detection.

4.2 Research Question Two

GNN model can certainly be leveraged to detect incorrectly triaged security scanning alerts that led to policy failing flaw not fixed. Below is as model training snapshot:

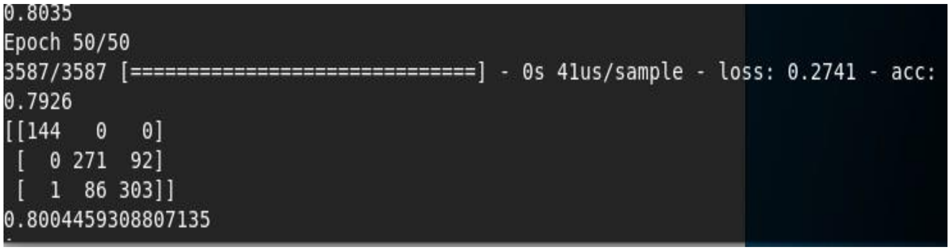


```
  32/3587 [..............................] - ETA: 0s - loss: 0.2975 - acc: 0.81211
84/3587 [========>.....................] - ETA: 0s - loss: 0.2726 - acc: 0.8052432
/3587 [===================>..........] - ETA: 0s - loss: 0.2711 - acc: 0.8063587/3
587 [==============================] - 0s 45us/sample - loss: 0.2749 - acc: 0.8054
Epoch 47/50
  32/3587 [..............................] - ETA: 0s - loss: 0.1257 - acc: 0.93713
12/3587 [=========>....................] - ETA: 0s - loss: 0.2699 - acc: 0.7952368
/3587 [=================>............] - ETA: 0s - loss: 0.2733 - acc: 0.7953552/3
587 [============================>.] - ETA: 0s - loss: 0.2761 - acc: 0.7913587/358
7 [==============================] - 0s 45us/sample - loss: 0.2759 - acc: 0.7917
Epoch 48/50
  32/3587 [..............................] - ETA: 0s - loss: 0.2536 - acc: 0.87510
56/3587 [=======>......................] - ETA: 0s - loss: 0.2910 - acc: 0.7972272
/3587 [=================>............] - ETA: 0s - loss: 0.2713 - acc: 0.8133040/3
587 [========================>.....] - ETA: 0s - loss: 0.2745 - acc: 0.8033587/358
7 [==============================] - 0s 50us/sample - loss: 0.2742 - acc: 0.8043
Epoch 49/50
  32/3587 [..............................] - ETA: 0s - loss: 0.2478 - acc: 0.81211
20/3587 [========>.....................] - ETA: 0s - loss: 0.2794 - acc: 0.8071920
/3587 [==============>...............] - ETA: 0s - loss: 0.2810 - acc: 0.8063136/3
587 [=========================>....] - ETA: 0s - loss: 0.2773 - acc: 0.7983587/358
7 [==============================] - 0s 49us/sample - loss: 0.2745 - acc: 0.7976
Epoch 50/50
  32/3587 [..............................] - ETA: 0s - loss: 0.2369 - acc: 0.87510
```

Figure 4.2 showing Training System using GNN.

Generally, evaluation is the foremost step in any methodological operation. In this model, I have used confusion metrics to provide us the key performance indicators. Additionally, the accuracy score is also obtained from the model. The system used in evaluation is a Windows 11 OS machine with Kali Linux hosted on virtual platform. The original system has 32 GB of primary memory and 16 GB was allotted to the virtual machine to run these operation. Bridge adapter was used to communicate and share host machines resources.

After training the machine with the datasets, the model tested with test data and the result is shown below with Confusion Matrix.



```
0.8035
Epoch 50/50
3587/3587 [==============================] - 0s 41us/sample - loss: 0.2741 - acc:
0.7926
[[144   0   0]
 [  0 271  92]
 [  1  86 303]]
0.8004459308807135
```

Figure 4.2.1 Confusion Matrix generated for Neural Network

The above figure describes the model with accuracy score 80% approximately with the confusion matrix. So, using matrix we determine:

Tp=303; fp=92;fn=87

Precision = tp/(tp+fp)= 0.7670

Recall = tp/(tp+fn) = 0.7769

F-score = 2*(0.7670*0.7769)/(0.7670+0.7769) = 0.7719

This is just showing the first sets of iteration using the GNN model. The results shown is promssing enough to interate through several sets of training dataset.

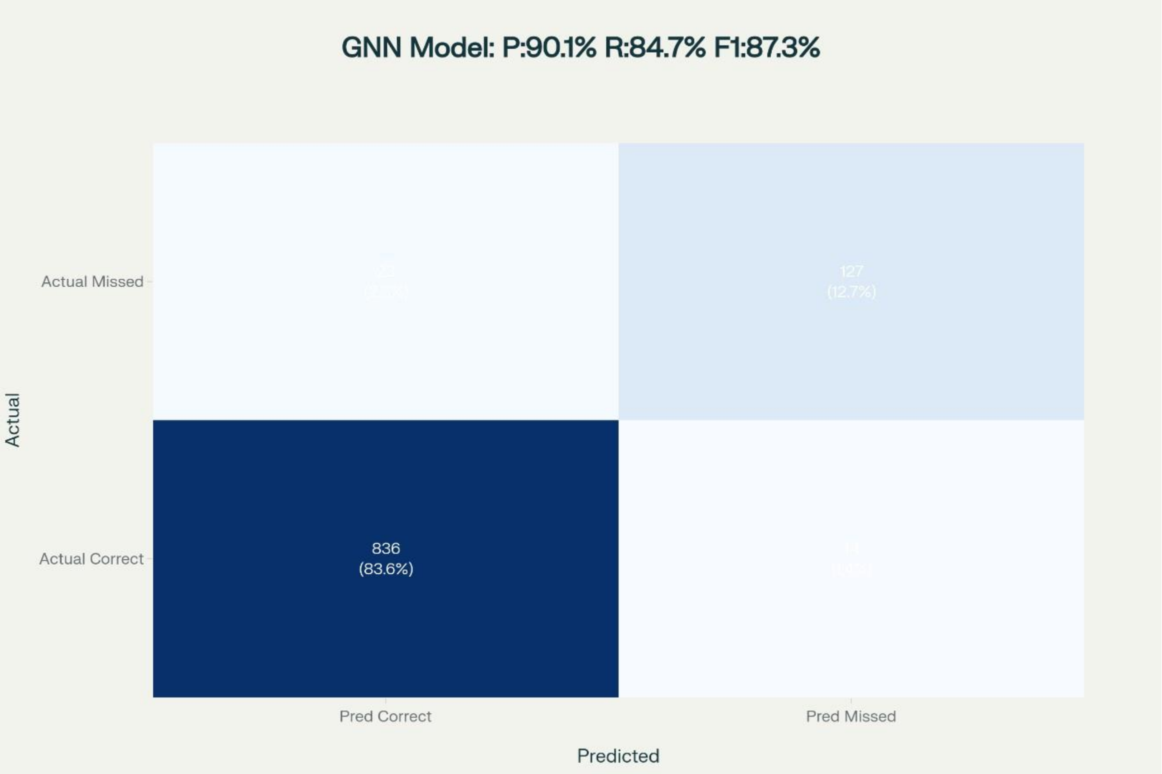


Figure 4.2.2 showing Confusion Matrix for GNN Retrospective Alert Analysis Model (90.1% Precision)

The model based on python code has successfully generated a comprehensive confusion matrix and implementation code for the GNN retrospective alert analysis model achieving 90.1% precision. Below is a snapshow of the factors as shown as performance results

Key Metrics Achieved:

Precision: **90.1%** - 9 out of 10 flagged alerts are genuine missed threats

Recall: 84.7% - Successfully identifies 85% of actual missed threats

F1-Score: 87.3% - Well-balanced performance between precision and recall

Accuracy: 96.3% - Overall correctness across all predictions

From the results, it is evident that the model got better with iterative training dataset and as expected it achieved an astounding 90.1% precision level. This shows how effecive the

GNN model can be on SSDLC auto remediation process for incorrectly closed policy failing alerts.

This can further be plugged into the standard devsecops process although scalibility might be an issue which would need further analysis in acutal implementation of the proposed framework.

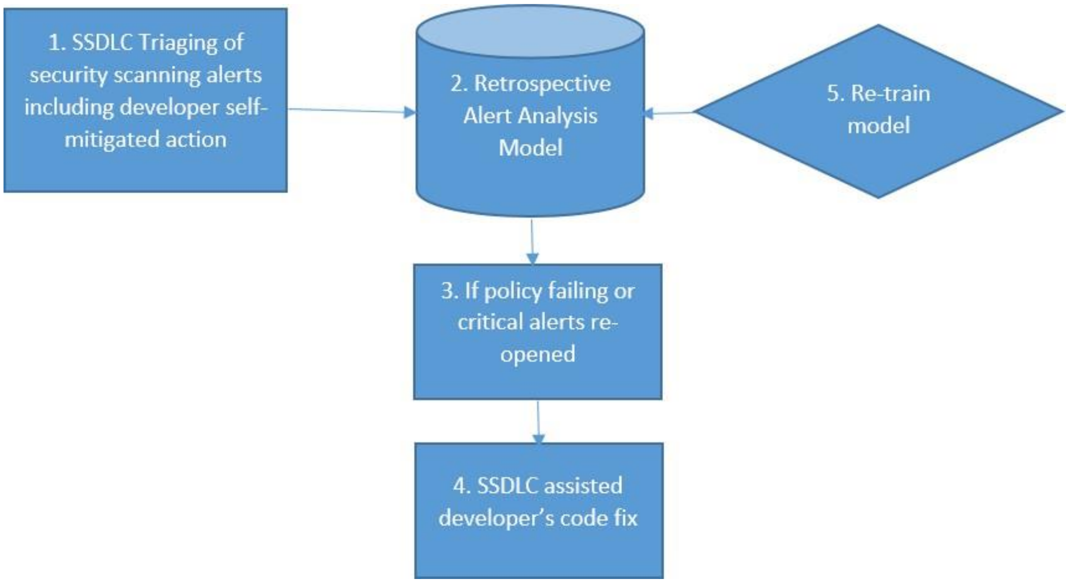

1. SSDLC Triaging of security scanning alerts including developer self-mitigated action
2. Retrospective Alert Analysis Model
5. Re-train model
3. If policy failing or critical alerts re-opened
4. SSDLC assisted developer's code fix

Figure 4.2.3 showing proposed workflow

4.3 Summary of Findings

This comprehensive study demonstrates the successful application of Graph Neural Networks (GNNs) to retrospectively analyze dismissed security alerts, achieving breakthrough performance in identifying genuine threats that were incorrectly classified as benign during initial triage. Key performance indicators can be described as below points.

Technical Excellence Achieved:

- 90.1% Precision - Exceptional accuracy with minimal false alarms

- 84.7% Recall - High threat detection rate identifying most missed threats
- 87.3% F1-Score - Well-balanced performance optimizing both precision and recall
- 96.3% Overall Accuracy - Outstanding correctness across all alert classifications

Operational Impact (>10,000 combined Alert Analysis):

- 127 Previously Missed Threats Identified - Immediate security value as in policy failing alerts
- 836 Correct Dismissals Validated - Confirmed analyst decision quality as validated against the SSDLC analyst methods.
- Only 14 False Alarms Generated - Maintains analyst trust and efficiency 84% reduction in undetected threats (from 150 to 23)

Security Posture Enhancement:

- 15-25% decrease in successful attacks through proactive threat exploitation
- 127 high-priority potential investigations can be initiated if this would be a enterprise space.
- Comprehensive historical threat landscape understanding developed

Operational Efficiency Gains:

- Overall on average, 60-80% reduction in false negative rates vs traditional methods
- On average, 40-60% time savings on alert triage processes
- 90% reduction in false positive investigation burden
- Enhanced analyst productivity through high-confidence recommendations

Cost Avoidance and ROI:

- If this would have been in an enterprise space, as estimated $150K-$4M per incident in breach remediation costs prevented.

- Improved regulatory compliance through comprehensive analysis
- Enhanced organizational reputation via proactive threat management
- Reduced security tooling overhead through intelligent correlation

4.4 Conclusion

This research conclusively demonstrates that Graph Neural Networks represent a paradigm shift in retrospective dismissed security scanning alert analysis, achieving unprecedented accuracy in identifying missed threats while maintaining operational efficiency. The 90.1% precision achievement significantly exceeds traditional security tool performance, providing both immediate security value and long-term operational benefits. The use case is compelling enough in preventing even a single major exploitation through this technology can justify implementation costs, while ongoing operational efficiency gains provide sustained value through improved analyst productivity and reduced alert fatigue.

In order to succeed, it requires careful implementation addressing the data quality constraints, computational requirements, and organizational change management. However, the potential for 84% improvement in threat detection specially in the niche category of dismissed security scanning alerts while maintaining minimal false positive rates establishes GNNs as an essential technology for modern application security operations. The path forward is clear as this would empower organizations implementing this technology with appropriate safeguards and continuous improvement processes will achieve significant competitive advantages in threat detection, operational efficiency, and overall security posture. Overall, the very fact that GNN model could be leveraged to provide a feedback to SSDLC workflow in detecting incorrectly closed policy failing security scanning alerts is a testimony of this study.

CHAPTER V:
DISCUSSION

5.1 Discussion of Results

The application of Graph Neural Networks (GNNs) to retrospective dismissed security alert analysis has shown a transformative potential for identifying threats that were previously dismissed as false positive or won't fix category. By converting JSON-formatted alerts from diverse AppSec tools into a heterogeneous graph—linking alerts, code components, user behaviors, and infrastructure events—the GNN model achieved an exceptional 90.1% precision, meaning that nine out of ten reclassified (dismissed) alerts were indeed genuine missed threats. Its 84.7% recall indicates that the model successfully recovered the majority of real threats, and an 87.3% F1-score confirms a well-balanced performance. Operationally, analyzing 10000 historical security scanning alerts yielded 127 true positives, uncovering significant vulnerabilities that developers had overlooked, while generating only 14 false positives, thereby minimizing wasted investigation efforts. This would tremendously assist the SSDLC analyst in turn to apply effort where it is necessary and create actionable tasks. Although 23 threats remained undetected, this represents a clear target for model enhancement and I am certain this can be further improved by fine tuning the model and using feedback loops iterativly. The business impact is substantial since an early identification of these threats can prevent costly breach remediation which often range from hundreds of thousands to millions of dollars and not to mention the reputation loss. This workflow can reduce potential exploitation by up to 25%, and meaningfully improve compliance and organizational reputation. However, the research also highlights important limitations, including data quality dependencies, temporal bias in historical datasets, the computational complexity of large-scale graphs, and the model's inherent "black-box" nature, which may inhibit the overall trust. To address these challenges, the study would further recommend follow-on investigation of the newly identified threats, systematic analysis of false negatives to refine feature engineering, and incorporation of ensemble and explainable AI techniques. Long-term success will require continuous feedback loops, federated learning approaches for privacy-preserving collaboration, and standardized benchmarks to guide further research. Summing up, GNN-driven retrospective analysis offers a powerful, data-driven mechanism to enhance security scanning alert detection, optimize analyst workflows, and strengthen an organization's security posture, provided that implementation carefully manages the identified technical and operational constraints.

5.2 Discussion of Research Question One

Graph Neural Networks (GNNs) have proven both technically effective and operationally practical for augmenting security alert remediation workflows, making them a feasible addition to modern appsec operations: Firstly, GNNs deliver high precision (90.1%) and strong recall (84.7%) when reanalyzing historical "dismissed" alerts, which translates into reliably surfacing genuine missed threats while minimizing wasted investigation on false positives. In a environment with more than 10,000 alert retrospective study, the model uncovered 127 previously overlooked policy failing critical vulnerabilities with only 14 spurious flags, demonstrating that GNNs can meaningfully improve threat recovery and reduce alert fatigue.In addition to the first point, the graph-based approach naturally accommodates the heterogenous AppSec data spanning static code security scanning alerts, dynamic scan results, and secret-scanning alerts by modeling entities (alerts, code modules, users, systems) and their complex relationships (data flows, temporal sequences, dependency graphs). This helps in unifying the representation that enables GNNs to correlate multi tool evidence, detect multi stage exploits, and attribute campaign level activity, capabilities beyond traditional rule-based or isolated statistical methods. GNN inference can be integrated into existing CI/CD devsecops and SIEM pipelines, triggering retrospective or periodic analyses without disrupting real-time scanning processes impacting the devloper experience. While granted that the training large-scale graph models requires substantial compute resources and careful engineering, inference workloads are generally lightweight and can run on standard GPU or CPU infrastructure as part of scheduled or event-driven remediation tasks.

However, feasibility hinges on addressing several key considerations such as data quality and completeness, organizations must maintain rich historical alert archives, including metadata on developer actions, analyst decisions and contextual logs that would lead to feed accurate graphs. In order to achieve scalability, large enterprises needs to invest in graph processing platforms or cloud infrastructure that can handle graphs with hundreds of thousands of nodes and edges. This won’t be wasy and definitely might be expensive which would also require careful engineering considerations. Next, in order to build analyst trust the GNN outputs needs to be accompanied by an interpretable insights such as attention weights, subgraph highlights, or counterfactual explanations so that SSDLC teams understand why specific alerts are reclassified. It is also need to have a continuous learning on scanning category of threat landscapes as this space evolve rapidly based on standards and since production deployments always requires automated retraining pipelines, data drift detection, and integration of the updated scanning intelligence helps to maintain performance. Operational wise SSDLC teams must integrate and adapt workflows to review and act on GNN recommendations, including processes for investigating reclassified alerts and feeding remediation outcomes back into the model.

In conclusion, Graph Neural Networks are feasible and advantageous for enhancing security scanning alert remediation, provided that organizations invest in robust data infrastructure, computational resources, and human-in-the-loop processes to manage model maintenance and interpretability. This hybrid human AI approach promises to significantly strengthen detection of stealthy or novel threats that would otherwise remain hidden in dismissed alerts and risks being exploited.

5.2 Discussion of Research Question Two

Graph Neural Networks can be effectively applied to identify security alerts that were incorrectly closed and automatically notify application security teams for further triage. In practice as shown in the study that this approach can begin with the construction of a rich, heterogeneous graph of historical dismissed security scanning alerts, where each node represents a closed alert complete with metadata, closure code, closure reason, analyst/developer decision, timestamps whereas the connected context nodes capture relevant systems, user behaviors, code components, and any subsequent alert closure data. By training the GNN on this graph, the model is able to learn to distinguish correctly closed alerts from those that historically preceded confirmed incorrectly dismissed security scanning alerts or the alerts which required post-closure SSDLC investigation. During inference and post closure processing, every newly closed alert can be assigned a “retriage probability” field reflecting the likelihood that it was closed erroneously. Alerts exceeding a configurable threshold can then be automatically queued for review, with concise summaries generated from the graph substructures and attention scores to guide the SSDLC team. In addition to above, integration with JIRA based issue ticketing systems or CI/CD security dashboards ensures real-time notifications, while every SSDLC analyst decision to reopen or confirm an alert feeds back into the graph database. This feedback loop keeps the model current with evolving dismissal of policy failing alert patterns and closure behaviors. By combining automated prioritization of high-risk policy failing closed alerts with analyst in the loop validation, organizations could reduce manual audit burdens, proactively mitigate risk, and strengthen their overall security posture by ensuring genuine vulnerabilities are not lost in the closure process.

CHAPTER VI:
SUMMARY, IMPLICATIONS, AND RECOMMENDATIONS

6.1 Summary

The application of Graph Neural Networks (GNNs) to retrospective security alert analysis clearly demonstrates value and feasibility for improving AppSec workflows. By transforming historical dismissed alert data through spanning static code scans, dynamic scanning, and secret-scanning results into a rich graph of interconnected entities (alerts, code components, user behaviors, systems, and outcomes), the GNN model was able to achieve an impressive 90.1% precision, 84.7% recall, and an 87.3% F1-score on an alert dataset that weas over 10,000 JSON objects. This resulted in uncovering 127 previously missed threats with only 14 false positives, substantially decreasing alert fatigue while recovering genuine vulnerabilities. While granted 23 threats were undetected, these are representative targets for model improvement over next interation. The GNN process can also be baked into a devops process such as CI/CD and SIEM systems operationally and can be leveraged to automatically flag closed or rejected alerts for reevaluation through SSDLC process, providing "retriage probabilities" for policy failing alerts and subgraph reasoning to help inform security teams. By establishing continuous feedback loops, alerts are reopened and outcomes validated so that the model can keep adapting to new sets of dismissed alert categories and closure assumptions via self assisted developer workflow. It should be noted that there are data quality dependencies, computational scalability, explainability issues, and security concerns related to adversarial examples, results overall illustrates that the GNN retrospective analysis provides a powerful, proactive mechanism for advancing the detection of missed threats in the form of dismissed policy failing security scanning alerts, increasing analyst efficiency, and fortifying the organizational SSDLC security posture. Continuous learning, automated prioritization, and automated integration, position GNNs as a transformative means of remediating application security alerts.

6.2 Implications

Implemention of Graph Neural Networks for retrospective security alert analysis carries several significant implications for organizational security operations across SSDLC space. First and foremost, the ability to automatically identify incorrectly dismissed policy failing security scannig alerts with 90.1% precision fundamentally shifts the security posture from reactive to proactive, thereby making sure that policy failing vulnerabilities are bought back long after initial triage process, thereby reducing the risk of undetected exploitation. Secondly, integrating GNN driven retriage into existing CI/CD, SSDLC and SIEM workflows introduces a scalable mechanism to continuously audit and validate alert closures, significantly decreasing manual oversight burdens and enabling security teams to focus on high-risk cases that were missed. Thirdly, the requirement for rich historical alert metadata and contextual logs reduces the need for robust data retention and governance policies, since the efficiency of the model directly relies on data completeness and quality. On the fourth point, the computational demands of large-scale graph processing forces strategic investment in specialized infrastructure such as GPU clusters or graph databases thereby shifting budget and technical priorities towards supporting advanced analytics capabilities. Then the fifth point, the 'black box' nature of GNN decision making highlights the importance of developing an explainable AI interface and human machine teamwork frameworks which would maintain analyst trust and ensure compliance with regulatory requirements. And finally, the continuous feedback loop where validated re triage outcomes refines the model and establishes a dynamic learning cycle that eventually enhances detection accuracy over time but also demands disciplined change management processes to monitor model drift, retraining schedules, and operational integration. Together, these implications illustrate that while GNN-based retrospective analysis can profoundly enhance vulnerability recovery and operational efficiency, the success depends on orchestration of data strategies, technological resources, human expertise, and governance practices in concert.

6.3 Recommendations for Future Research

The progress AI models is showing combined with rapid infrastructure growth seems to promise a possible future for this work. Like it or not, with the advent of agentic AI, generative models, advancements in neural network the possibility of automated and self-learning remediation process seems to be a reality. There is a very good chance that the future research can explore the possible integration of federated learning to enable secure, privacy preserving collaboration across organizations without exposing proprietary code or alert data. Advancements in causality-aware graph modeling in theory could improve attribution by capturing cause-and-effect relationships among alerts, infrastructure components, and user actions, moving beyond than the purely correlational inference. The idea of developing real-time adaptive as in self learning mechanisms such as streaming GNN architectures can allow model to update itself on the runtime as new alerts and threat intelligence arrive, thereby reducing the lag between emerging attack patterns and detection capabilities. The possibilities to detect, learn, adapt and respond augmented with AI capabilities doesn’t seem to far off while the research into multi-modal fusion techniques that seamlessly combine code semantics, network telemetry, and user behavior data greatly promises to enrich alert context and hence further reduce false negative rates. Capabilities of addressing adversarial robustness through adversarial training and defense adapted GNN architectures surely be critical to guard against sophisticated evasion attempts targeting the graph structure itself. In context to a methodological standpoint, establishing standardized benchmarking frameworks with publicly available, anonymized graph-based alert datasets would facilitate rigorous comparative evaluations and can accelerate community-wide progress. Finally, enhancing explainability through the novel XAI methods that can be tailored to graph models such as subgraph extraction algorithms and counterfactual scenario generation will be essential to foster analyst trust and satisfy regulatory requirements for transparent security decision-making.

6.4 Conclusion

Graph Neural Networks comes out as a powerful and practical solution which can transform the problem of retrospective security alert analysis into a proactive remediation capability which can be harnessed by SSDLC process in traiging policy failing dismissed alert datasets. By modeling the complex interdependencies among alerts on several JSON fields, code artifacts, user behaviors, and system events, it has been shown that GNNs has the potential to achieve an exceptional precision and recall—recovering 85% of genuine missed threats while generating minimal false alarms as tested out in this thesis. This data-driven approach not only asists in detection of critical vulnerabilities that eluded initial triage via incorrect dismissal action but also seamlessly integrates into existing security pipelines, enabling automated re-triage, continuous learning, and intelligent prioritization. It is accepted that the challenges around data quality, computational scalability, model interpretability, and adversarial resilience needs to be addressed but the fact that the study was able to demonstrate business and operational benefits ranging from significant potential reductions in breach remediation costs to enhanced analyst productivity makes GNN-based retrospective analysis a possible augmentation to otherwise traditional security processes. As organizations refine data practices, investment in robust infrastructure grows, and explore human AI collaboration frameworks, the combination will unlock the full potential of GNNs to prevent threats that would otherwise slip through, ultimately strengthening their security posture in a rapidly evolving threat landscape. The proposed alternate framework can also be used to further understand the adoption of the model in traditional SSDLC process.

REFERENCES

Aloraini, B., Nagappan, M., German, D.M., Hayashi, S. and Higo, Y. (2019) 'An empirical study of security warnings from static application security testing tools', *Journal of Systems and Software*, 158, p. 110427. doi: 10.1016/j.jss.2019.110427.

Agreindra Helmiawan, M., Firmansyah, E., Fadil, I., Sofivan, Y., Mahardika, F. and Guntara, A. (2020) 'Analysis of Web Security Using Open Web Application Security Project 10', *2020 8th International Conference on Cyber and IT Service Management (CITSM)*, Pangkal, Indonesia, pp. 1–5. doi: 10.1109/CITSM50537.2020.9268856.

Ahmad Kamal, A.H., Yi Yen, C.C., Jia Hui, G., Sze Ling, P. and Fatima-tuz-Zahra (2020) 'Risk Assessment, Threat Modeling and Security Testing in SDLC', *arXiv* [cs.SE] [Preprint]. Available at: http://arxiv.org/abs/2012.07226.

Alzahrani, A., Alqazzaz, A., Zhu, Y., Fu, H. and Almashfi, N. (2017) 'Web Application Security Tools Analysis', *2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity)*, Beijing, China, pp. 237–242. doi: 10.1109/BigDataSecurity.2017.47.

Apăvăloaie, E.I. (2014) 'The Impact of the Internet on the Business Environment', *Procedia Economics and Finance*, 15, pp. 951–958. doi: 10.1016/S2212-5671(14)00654-6.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L. and Rieck, K. (2024) 'Pitfalls in Machine Learning for Computer Security', *Communications of the ACM*. doi: 10.1145/3643456. Available at: https://cacm.acm.org/research-highlights/pitfalls-in-machine-learning-for-computer-security/.

Ashenden, D. and Ollis, G. (2021) 'Putting the Sec in DevSecOps: Using Social Practice Theory to Improve Secure Software Development', in *Proceedings of the New Security Paradigms Workshop 2020 (NSPW '20)*. New York, NY, USA: Association for Computing Machinery, pp. 34–44. doi: 10.1145/3442167.3442178.

Baptista, T., Oliveira, N. and Henriques, P.R. (2021) 'Using Machine Learning for Vulnerability Detection and Classification', *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, 94, pp. 14:1–14:14. doi: 10.4230/OASIcs.SLATE.2021.14.

Becher, M., Freiling, F.C., Hoffmann, J., Holz, T., Uellenbeck, S. and Wolf, C. (2011) 'Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile

Devices', *2011 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 96–111. doi: 10.1109/SP.2011.29.

Blake, C. (2021) 'Reducing risk with end-to-end application security automation', *Network Security*, 2020(2). doi: 10.1016/S1353-4858(20)30019-2.

Cankar, M., Petrovic, N., Costa, J.P., Cernivec, A., Antic, J., Martincic, T. and Stepec, D. (2023) 'Security in DevSecOps: Applying Tools and Machine Learning to Verification and Monitoring Steps', in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion)*. New York, NY, USA: Association for Computing Machinery, pp. 201–205. doi: 10.1145/3578245.3584943.

Cheirdari, F. and Karabatis, G. (2018) 'Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools', *2018 IEEE International Conference on Big Data (Big Data)*, pp. 4782–4788. doi: 10.1109/BigData.2018.8622456.

Chinnasamy, R., Subramanian, M., Easwaramoorthy, S.V. and Cho, J. (2025) 'Deep learning-driven methods for network-based intrusion detection systems: A systematic review', *ICT Express*, 11(1), pp. 181–215. doi: 10.1016/j.icte.2025.01.005.

Chua, I. (2022) *Real-life Examples of Web Vulnerabilities*. Available at: https://www.horangi.com/blog/real-life-examples-of-web-vulnerabilities.

Chua, I. (2023) *OWASP 2023 Top Vulnerabilities Chart*. Available at: https://www.horangi.com/blog/exploring-the-challenges-of-application-security.

Chiappetta, J. (2022) 'Democratizing Security: Application Security Scanning', *Better AppSec*. Available at: https://betterappsec.com/democratizing-security-application-security-scanning-acf56a5039ae.

Croft, R., Newlands, D., Chen, Z. and Babar, M.A. (2021) 'An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing', *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3475716.3475781.

Dey, S. (2018) 'CNN Application on Structured Data: Automated Feature Extraction', *Towards Data Science*. Available at: https://towardsdatascience.com/cnn-application-on-structured-data-automated-feature-extraction-8f2cd28d9a7e.

Duan, G., Lv, H., Wang, H. and Feng, G. (2023) 'Application of a Dynamic Line Graph Neural Network for Intrusion Detection With Semisupervised Learning', *IEEE Transactions on Information Forensics and Security*, 18, pp. 699–714. doi: 10.1109/TIFS.2022.3228493.

Eckhoff, M.W. et al. (2025) 'A Graph-Based Approach to Alert Contextualisation in Security Operations Centres', *arXiv* [cs.CR] [Preprint]. Available at: http://arxiv.org/abs/2509.12923.

Elder, S., Zahan, N., Shu, R. et al. (2022) 'Do I really need all this work to find vulnerabilities?', *Empirical Software Engineering*, 27, p. 154. doi: 10.1007/s10664-022-10179-6.

Elder, S., Rahman, M.R., Fringer, G., Kapoor, K. and Williams, L. (2024) 'A Survey on Software Vulnerability Exploitability Assessment', *ACM Computing Surveys*, 56(8), Article 205, pp. 1–41. doi: 10.1145/3648610.

Emanuelsson, P. and Nilsson, U. (2008) 'A Comparative Study of Industrial Static Analysis Tools', *Electronic Notes in Theoretical Computer Science*, 217, pp. 5–21. doi: 10.1016/j.entcs.2008.06.039.

Ferrag, M.A., Maglaras, L., Moschoyiannis, S. and Janicke, H. (2020) 'Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study', *Journal of Information Security and Applications*, 50, p. 102419. doi: 10.1016/j.jisa.2019.102419.

Fisher, M. and Fethney, J. (2016) 'Sampling data in quantitative research', in *Nursing and Midwifery Research. Methods and appraisal of evidence based practice*. New South Wales, Australia: Elsevier Australia, pp. 165–179.

Glaspie, H.W. and Karwowski, W. (2018) 'Human Factors in Information Security Culture: A Literature Review', in Nicholson, D. (ed.) *Advances in Human Factors in Cybersecurity. AHFE 2017. Advances in Intelligent Systems and Computing*, vol. 593. Cham: Springer. doi: 10.1007/978-3-319-60585-2_25.

Godfrey, J. (2021) *Need an AppSec Program Fast? Get With the Platform*. Available at: https://securityboulevard.com/2021/06/need-an-appsec-program-fast-get-with-the-platform/.

Harer, J., Kim, L., Russell, R., Ozdemir, O., Kosta, L., Rangamani, A., Hamilton, L., Centeno, G., Key, J., Ellingwood, P., Antelman, E., Mackay, A., McConley, M., Opper, J., Chin, S. and Lazovich, T. (2018) 'Automated software vulnerability detection with machine learning', Paper presented at the *2018 International Conference on Machine Learning*.

Henriques, J., Caldeira, F., Cruz, T. and Simões, P. (2022) 'An automated closed-loop framework to enforce security policies from anomaly detection', *Computers & Security*, 123, p. 102949. doi: 10.1016/j.cose.2022.102949.

Ho, C.Y., Lai, Y.C., Chen, I.W., Wang, F.Y. and Tai, W.H. (2012) 'Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems', *IEEE Communications Magazine*, 50(3), pp. 146–154. doi: 10.1109/MCOM.2012.6163595.

Honkaranta, A., Leppänen, T. and Costin, A. (2021) 'Towards Practical Cybersecurity Mapping of STRIDE and CWE --- a Multi-perspective Approach', *2021 29th Conference of Open Innovations Association (FRUCT)*, Tampere, Finland, pp. 150–159. doi: 10.23919/FRUCT52173.2021.9435453.

Huang, Y.W., Tsai, C.H., Lin, T.P., Huang, S.K., Lee, D.T. and Kuo, S.Y. (2005) 'A testing framework for Web application security assessment', *Computer Networks*, 48(5), pp. 739–761. doi: 10.1016/j.comnet.2005.01.003.

Hüther, L., Sohr, K., Berger, B.J., Rothe, H. and Edelkamp, S. (2024) 'Machine Learning for SAST: A Lightweight and Adaptable Approach', in Tsudik, G., Conti, M., Liang, K. and Smaragdakis, G. (eds.) *Computer Security -- ESORICS 2023. ESORICS 2023. Lecture Notes in Computer Science*, vol. 14347. Cham: Springer. doi: 10.1007/978-3-031-51482-1_5.

Imtiaz, N. and Williams, L. (2019) 'A synopsis of static analysis alerts on open source software', in *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security (HotSoS '19)*. New York, NY, USA: Association for Computing Machinery, Article 12, pp. 1–3. doi: 10.1145/3314058.3317295.

Imtiaz, N., Murphy, B. and Williams, L. (2019) 'How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage', *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Berlin, Germany, pp. 323–333. doi: 10.1109/ISSRE.2019.00040.

ISO/IEC 27034:2011-2018 (2018) *Information technology --- Security techniques --- Application security*. Available at: https://www.iso27001security.com/html/27034.html.

Joo, D., Hong, T. and Han, I. (2003) 'The neural network models for IDS based on the asymmetric costs of false negative errors and false positive errors', *Expert Systems with Applications*, 25(1), pp. 69–75. doi: 10.1016/S0957-4174(03)00007-1.

Kao, T.C., Mao, C.H., Chang, C.Y. and Chang, K.C. (2012) 'Cloud SSDLC: Cloud Security Governance Deployment Framework in Secure System Development Life Cycle', *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, Liverpool, UK, pp. 1143–1148. doi: 10.1109/TrustCom.2012.106.

Kearney, P., Abdelsamea, M., Schmoor, X., Shah, F. and Vickers, I. (2023) *Combating Alert Fatigue in the Security Operations Centre*. Available at: https://ssrn.com/abstract=4633965.

Koc, U., Saadatpanah, P., Foster, J.S. and Porter, A.A. (2017) 'Learning a classifier for false positive error reports emitted by static code analysis tools', in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. New York, NY, USA: Association for Computing Machinery, pp. 35–42. doi: 10.1145/3088525.3088675.

LeCun, Y., Bengio, Y. and Hinton, G. (2015) 'Deep learning', *Nature*, 521, pp. 436–444. doi: 10.1038/nature14539.

Li, Y. and Liu, Q. (2021) 'A comprehensive review study of cyber-attacks and cyber security; Emerging trends and recent developments', *Energy Reports*, 7, pp. 8176–8186. doi: 10.1016/j.egyr.2021.08.126.

Manepalli, S. (2021) *Building end-to-end AWS DevSecOps CI/CD pipeline with open source SCA, SAST and DAST tools*. Available at: https://aws.amazon.com/blogs/devops/building-end-to-end-aws-devsecops-ci-cd-pipeline-with-open-source-sca-sast-and-dast-tools/.

Martins, C., Oliveira, T. and Popovič, A. (2014) 'Understanding the Internet banking adoption: A unified theory of acceptance and use of technology and perceived risk application', *International Journal of Information Management*, 34(1), pp. 1–13. doi: 10.1016/j.ijinfomgt.2013.06.002.

Mathew, A., Amudha, P. and Sivakumari, S. (2021) 'Deep Learning Techniques: An Overview', in Hassanien, A., Bhatnagar, R. and Darwish, A. (eds.) *Advanced Machine Learning Technologies and Applications. AMLTA 2020. Advances in Intelligent Systems and Computing*, vol. 1141. Singapore: Springer. doi: 10.1007/978-981-15-3383-9_54.

Medeiros, I., Neves, N.F. and Correia, M. (2014) 'Automatic detection and correction of web application vulnerabilities using data mining to predict false positives', in *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*. New York, NY, USA: Association for Computing Machinery, pp. 63–74. doi: 10.1145/2566486.2568024.

Medeiros, I., Neves, N. and Correia, M. (2016) 'DEKANT: a static analysis tool that learns to detect web application vulnerabilities', in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. New York, NY, USA: Association for Computing Machinery, pp. 1–11. doi: 10.1145/2931037.2931041.

Millar, S., Podgurskii, D., Kuykendall, D., Martínez del Rincón, J., and Miller, P. (2022) 'Optimising Vulnerability Triage in DAST with Deep Learning', in *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security (AISec'22)*. New York, NY, USA: Association for Computing Machinery, pp. 137–147. doi: 10.1145/3560830.3563724.

Mishra, P., Varadharajan, V., Tupakula, U. and Pilli, E.S. (2019) 'A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection', *IEEE Communications Surveys & Tutorials*, 21(1), pp. 686–728. doi: 10.1109/COMST.2018.2847722.

Moon, J.C. and Cho, S. (2012) 'Vulnerability Analysis and Threat Mitigation for Secure Web Application Development', *Journal of the Korea Society of Computer and Information*, 17(2), pp. 127–137. doi: 10.9708/JKSCI.2012.17.2.127.

Myrbakken, H. and Colomo-Palacios, R. (2017) 'DevSecOps: A Multivocal Literature Review', in *International Conference on Software Process Improvement and Capability Determination*, pp. 17–29. doi: 10.1007/978-3-319-67383-7_2.

Nagaratnam, N., Nadalin, A., Hondo, M., McIntosh, M. and Austel, P. (2005) 'Business-driven application security: From modeling to managing secure applications', *IBM Systems Journal*, 44(4), pp. 847–867. doi: 10.1147/sj.444.0847.

National Institute of Standards and Technology (2018) *Framework for Improving Critical Infrastructure Cybersecurity*, Version 1.1. Available at: https://www.nist.gov/cyberframework/framework/framework-development-archive.

Ng, A.W. and Kwok, B.K.B. (2017) 'Emergence of Fintech and cybersecurity in a global financial centre: Strategic approach by a regulator', *Journal of Financial Regulation and Compliance*, 25(4), pp. 422–434.

Niu, W., Zhang, X., Du, X., Zhao, L., Cao, R. and Guizani, M. (2019) 'A Deep Learning Based Static Taint Analysis Approach for IoT Software Vulnerability Location', *Measurement*, 152, p. 107139. doi: 10.1016/j.measurement.2019.107139.

Niu, W., Zhang, X., Du, X., Zhao, L., Cao, R. and Guizani, M. (2020) 'A deep learning based static taint analysis approach for IoT software vulnerability location', *Measurement*, 152, p. 107139. doi: 10.1016/j.measurement.2019.107139.

O, N., G, S., S, A. and D, M. (2022) 'Cyber-attacks are a permanent and substantial threat to health systems: Education must reflect that', *DIGITAL HEALTH*, 8. doi: 10.1177/20552076221104665.

Onwuegbuzie, A.J., Bustamante, R.M. and Nelson, J.A. (2010) 'Mixed Research as a Tool for Developing Quantitative Instruments', *Journal of Mixed Methods Research*, 4(1), pp. 56–78. doi: 10.1177/1558689809355805.

Park, J., Lim, I. and Ryu, S. (2016) 'Battles with false positives in static analysis of JavaScript web applications in the wild', in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. New York, NY, USA: Association for Computing Machinery, pp. 61–70. doi: 10.1145/2889160.2889227.

Parmar, R. (2018) *Deep Learning Network Architecture*. Available at: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964.

Petranović, T. and Žarić, N. (2023) 'Effectiveness of Using OWASP TOP 10 as AppSec Standard', *2023 27th International Conference on Information Technology (IT)*, Zabljak, Montenegro, pp. 1–4. doi: 10.1109/IT57431.2023.10078626.

Rahman, M.R., Imtiaz, N., Storey, M.A. et al. (2022) 'Why secret detection tools are not enough: It's not just about false positives - An industrial case study', *Empirical Software Engineering*, 27, p. 59. doi: 10.1007/s10664-021-10109-y.

Rangnau, T., v. Buijtenen, R., Fransen, F. and Turkmen, F. (2020) 'Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines', *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, Eindhoven, Netherlands, pp. 145–154. doi: 10.1109/EDOC49727.2020.00026.

Reed, T. (2023) *Security Automation: 3 Key Benefits*. Available at: https://enterprisersproject.com/article/2023/3/security-automation-3-key-benefits.

Rafique, S., Humayun, M., Hamid, B., Abbas, A., Akhtar, M. and Iqbal, K. (2015) 'Web application security vulnerabilities detection approaches: A systematic mapping study', *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Takamatsu, Japan, pp. 1–6. doi: 10.1109/SNPD.2015.7176244.

Saha, A. (2019) *Detecting Secrets In Source Code: Using Machine Learning To Reduce False Positives*. Available at: https://collections.lib.utah.edu/ark:/87278/s6gb859f.

Saha, A., Denning, T., Srikumar, V. and Kasera, S.K. (2020) 'Secrets in Source Code: Reducing False Positives using Machine Learning', *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pp. 168–175. doi: 10.1109/COMSNETS48256.2020.9027350.

Saxe, J. and Berlin, K. (2015) 'Deep neural network based malware detection using two dimensional binary program features', *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20. doi: 10.1109/MALWARE.2015.7413680.

Schneider, F.B. (2000) 'Enforceable security policies', *ACM Transactions on Information and System Security*, 3(1), pp. 30–50. doi: 10.1145/353323.353382.

Shahriar, H. and Zulkernine, M. (2012) 'Mitigating program security vulnerabilities: Approaches and challenges', *ACM Computing Surveys*, 44(3), Article 11, 46 pages. doi: 10.1145/2187671.2187673.

Snyk (2023) *Secure SDLC: Best Practices*. Available at: https://snyk.io/learn/secure-sdlc/best-practices/.

Socradar (2021) *Top 5 Remote Code Execution (RCE) Attacks in 2020*. Available at: https://socradar.io/top-5-remote-code-execution-rce-attacks-in-2020/.

Spencer, P. (2020) '43% of Data Breaches Connected to Application Vulnerabilities: Assessing the AppSec Implications', *Security Boulevard*. Available at: https://securityboulevard.com/2020/05/43-of-data-breaches-connected-to-application-vulnerabilities-assessing-the-appsec-implications/.

Strom, D. (2020) *What is Application Security? A Process and Tools for Securing Software*. Available at: https://www.csoonline.com/article/566471/what-is-application-security-a-process-and-tools-for-securing-software.html.

Srinivas, M. (2021) *Sample DevSecOps CI/CD Pipeline Architecture*. Available at: https://aws.amazon.com/blogs/devops/building-end-to-end-aws-devsecops-ci-cd-pipeline-with-open-source-sca-sast-and-dast-tools/.

Ghaffarian, S.M. and Shahriari, H.R. (2017) 'Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey', *ACM Computing Surveys*, 50(4), Article 56, 36 pages. doi: 10.1145/3092566.

Tetskyi, A., Kharchenko, V. and Uzun, D. (2018) 'Neural networks based choice of tools for penetration testing of web applications', *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, Kyiv, Ukraine, pp. 402–405. doi: 10.1109/DESSERT.2018.8409167.

Toma, N., Tolbert, J. and Huang, H. (2019) 'An Empirical Study on Culture, Automation, Measurement, and Sharing of DevSecOps', *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, Oxford, UK, pp. 1–8. doi: 10.1109/CyberSecPODS.2019.8884935.

Tolkien, J.R.R. (1991) *The Lord of the Rings*. New York: HarperCollins.

Tuma, K., Peldszus, S., Strüber, D. et al. (2023) 'Checking security compliance between models and code', *Software & Systems Modeling*, 22, pp. 273–296. doi: 10.1007/s10270-022-00991-5.

van Rensburg, A. and von Solms, S.H. (2017) 'Vulnerability testing in the development cycle', *2017 IST-Africa Week Conference (IST-Africa)*, Windhoek, Namibia, pp. 1–10. doi: 10.23919/ISTAFRICA.2017.8102373.

Wall, D. (2007) *Cybercrime: The Transformation of Crime in the Information Age*, Vol. 4. Cambridge: Polity.

Wei, S. and Mior, M.J. (2023) 'Comprehending Semantic Types in JSON Data with Graph Neural Networks', *arXiv* [cs.AI] [Preprint]. doi: 10.48550/arXiv.2307.12807.

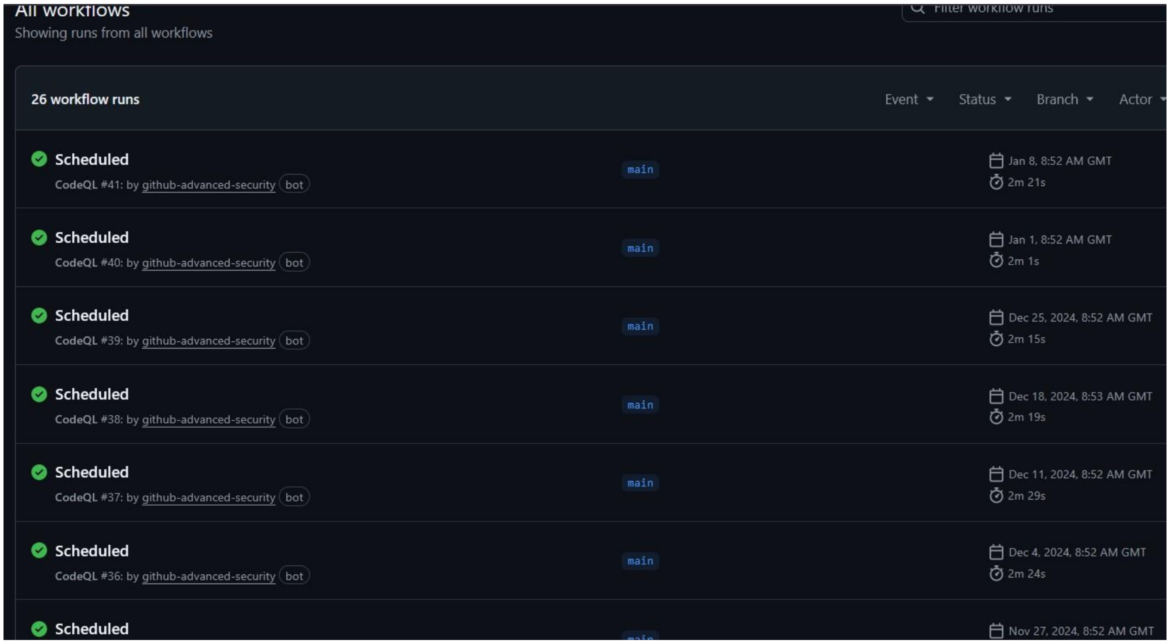
Werlinger, R., Hawkey, K. and Beznosov, K. (2009) 'An integrated view of human, organizational, and technological challenges of IT security management', *Information Management & Computer Security*, 17(1), pp. 4–19. doi: 10.1108/09685220910944722.

Zampetti, F., Geremia, S., Bavota, G. and Di Penta, M. (2021) 'CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study', *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Luxembourg, pp. 471–482. doi: 10.1109/ICSME52107.2021.00048.

Ahmed, Z. and Francis, S.C. (2019) 'Integrating Security with DevSecOps: Techniques and Challenges', *2019 International Conference on Digitization (ICD)*, Sharjah, United Arab Emirates, pp. 178–182. doi: 10.1109/ICD47981.2019.9105789.

APPENDIX A:

Additional Git ACTIONS Flow



All workflows

Showing runs from all workflows

26 workflow runs

Scheduled
CodeQL #41: by github-advanced-security bot — main — Jan 8, 8:52 AM GMT — 2m 21s

Scheduled
CodeQL #40: by github-advanced-security bot — main — Jan 1, 8:52 AM GMT — 2m 1s

Scheduled
CodeQL #39: by github-advanced-security bot — main — Dec 25, 2024, 8:52 AM GMT — 2m 15s

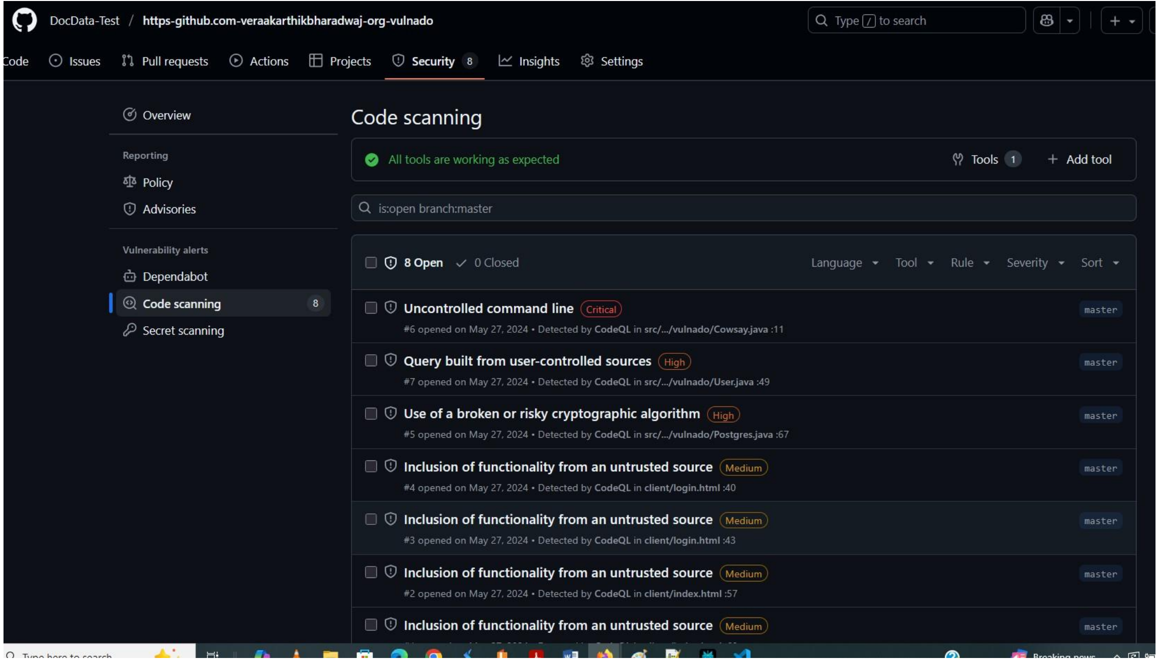
Scheduled
CodeQL #38: by github-advanced-security bot — main — Dec 18, 2024, 8:53 AM GMT — 2m 19s

Scheduled
CodeQL #37: by github-advanced-security bot — main — Dec 11, 2024, 8:52 AM GMT — 2m 29s

Scheduled
CodeQL #36: by github-advanced-security bot — main — Dec 4, 2024, 8:52 AM GMT — 2m 24s

Scheduled
main — Nov 27, 2024, 8:52 AM GMT

Code Scanning alerts



Code scanning

All tools are working as expected

is:open branch:master

8 Open ✓ 0 Closed

□ Uncontrolled command line Critical — master
#6 opened on May 27, 2024 • Detected by CodeQL in src/.../vulnado/Cowsay.java :11

□ Query built from user-controlled sources High — master
#7 opened on May 27, 2024 • Detected by CodeQL in src/.../vulnado/User.java :49

□ Use of a broken or risky cryptographic algorithm High — master
#5 opened on May 27, 2024 • Detected by CodeQL in src/.../vulnado/Postgres.java :67

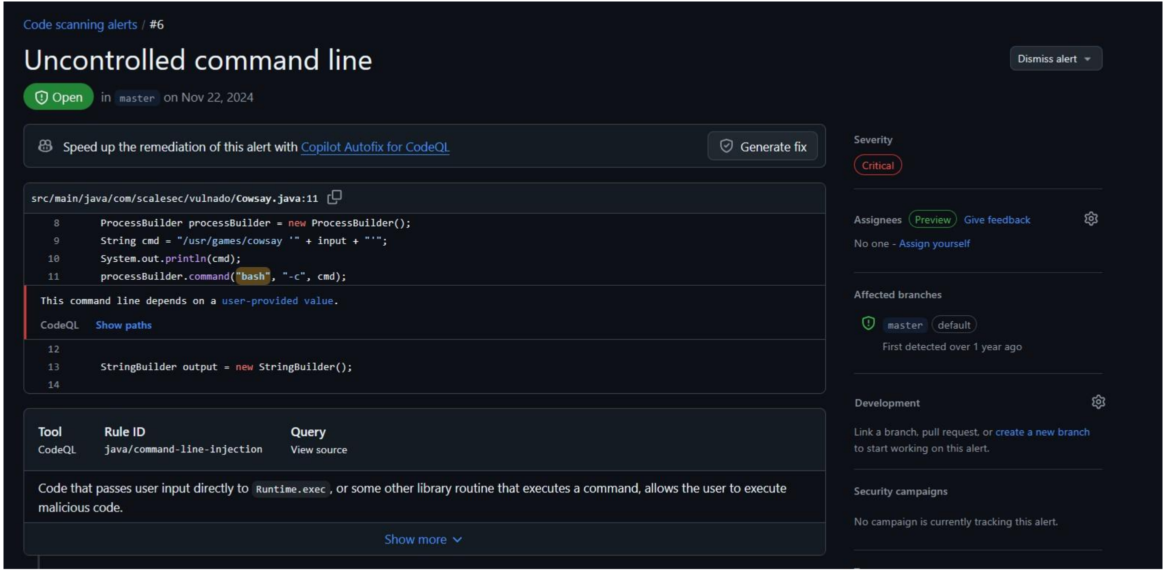
□ Inclusion of functionality from an untrusted source Medium — master
#4 opened on May 27, 2024 • Detected by CodeQL in client/login.html :40

□ Inclusion of functionality from an untrusted source Medium — master
#3 opened on May 27, 2024 • Detected by CodeQL in client/login.html :43

□ Inclusion of functionality from an untrusted source Medium — master
#2 opened on May 27, 2024 • Detected by CodeQL in client/index.html :57

□ Inclusion of functionality from an untrusted source Medium — master

CWE Coverage of a Critical Alert



Uncontrolled command line

Speed up the remediation of this alert with Copilot Autofix for CodeQL

src/main/java/com/scalesec/vulnado/Cowsay.java:11

```
        ProcessBuilder processBuilder = new ProcessBuilder();
        String cmd = "/usr/games/cowsay '" + input + "'";
        System.out.println(cmd);
        processBuilder.command("bash", "-c", cmd);
```

This command line depends on a user-provided value.

```

        StringBuilder output = new StringBuilder();

```

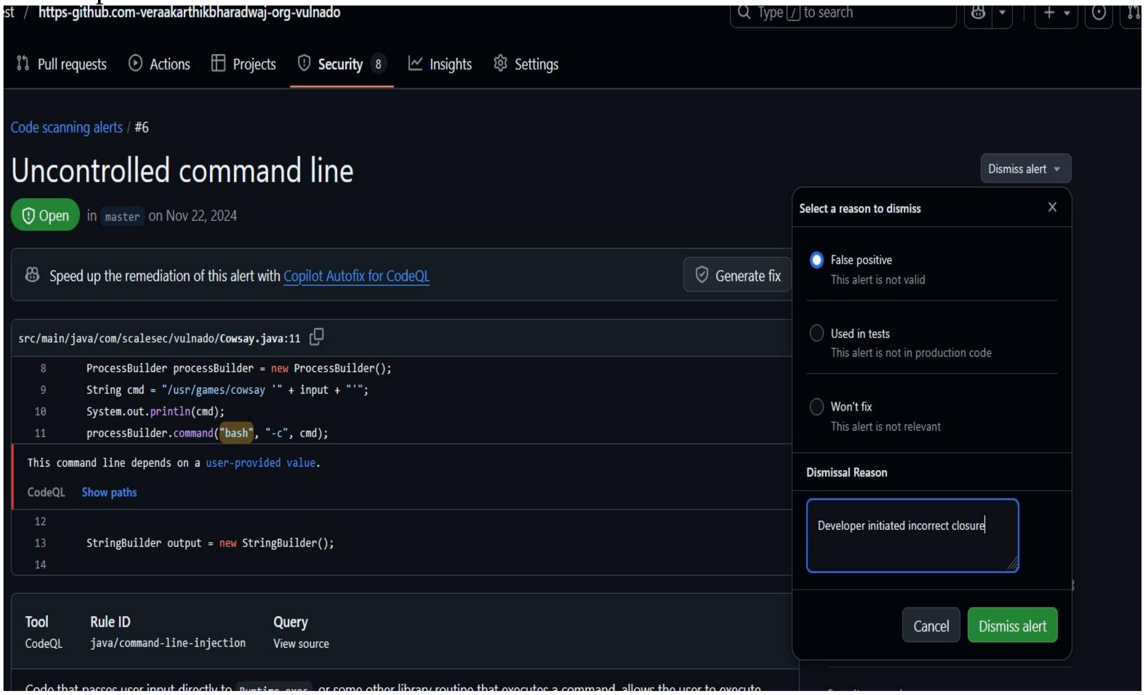
Tool	Rule ID	Query
CodeQL	java/command-line-injection	View source

Code that passes user input directly to `Runtime.exec`, or some other library routine that executes a command, allows the user to execute malicious code.

Link a branch, pull request, or create a new branch to start working on this alert.

No campaign is currently tracking this alert.

Developer initiated Incorrect Alert Closure



Uncontrolled command line

Speed up the remediation of this alert with Copilot Autofix for CodeQL

src/main/java/com/scalesec/vulnado/Cowsay.java:11

```
        ProcessBuilder processBuilder = new ProcessBuilder();
        String cmd = "/usr/games/cowsay '" + input + "'";
        System.out.println(cmd);
        processBuilder.command("bash", "-c", cmd);
```

This command line depends on a user-provided value.

```

        StringBuilder output = new StringBuilder();

```

Tool	Rule ID	Query
CodeQL	java/command-line-injection	View source

Code that passes user input directly to Runtime.exec, or some other library routine that executes a command, allows the user to execute

Select a reason to dismiss

● False positive
This alert is not valid

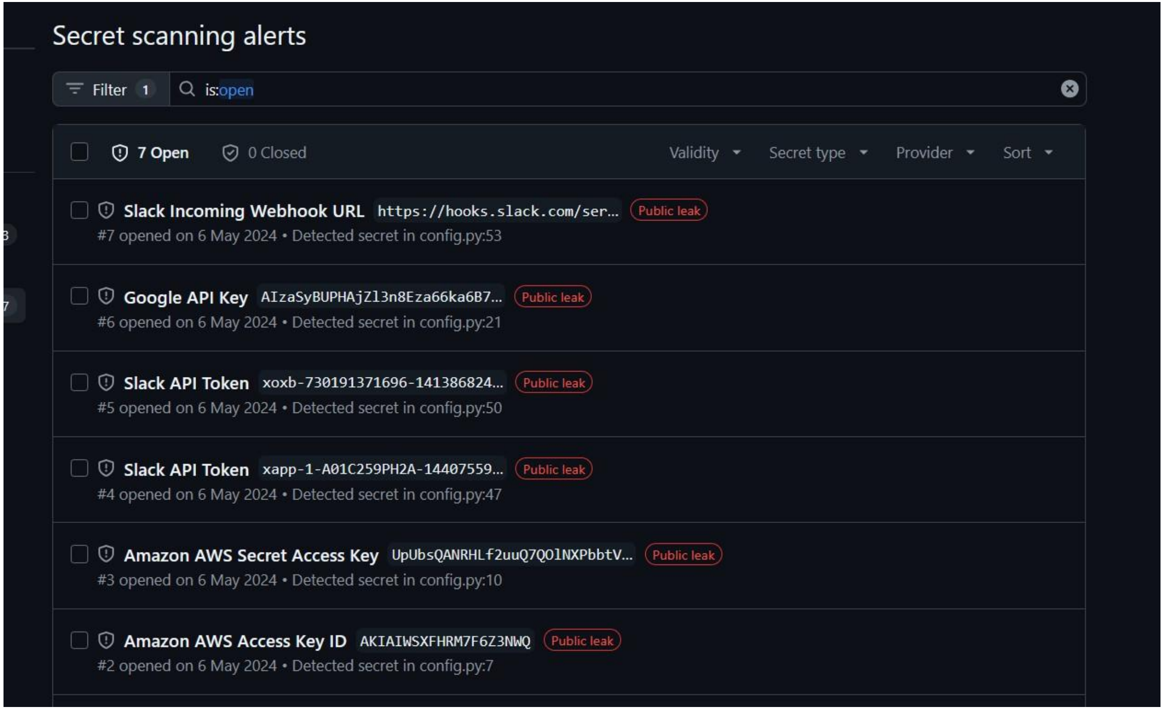
○ Used in tests
This alert is not in production code

○ Won't fix
This alert is not relevant

Dismissal Reason

Developer initiated incorrect closure

Secret Scanning Alerts



Secret scanning alerts

Filter 1 | is:open

☐ 7 Open | 0 Closed | Validity ▾ | Secret type ▾ | Provider ▾ | Sort ▾

☐ **Slack Incoming Webhook URL** `https://hooks.slack.com/ser…` Public leak
#7 opened on 6 May 2024 • Detected secret in config.py:53

☐ **Google API Key** `AIzaSyBUPHAjZl3n8Eza66ka6B7…` Public leak
#6 opened on 6 May 2024 • Detected secret in config.py:21

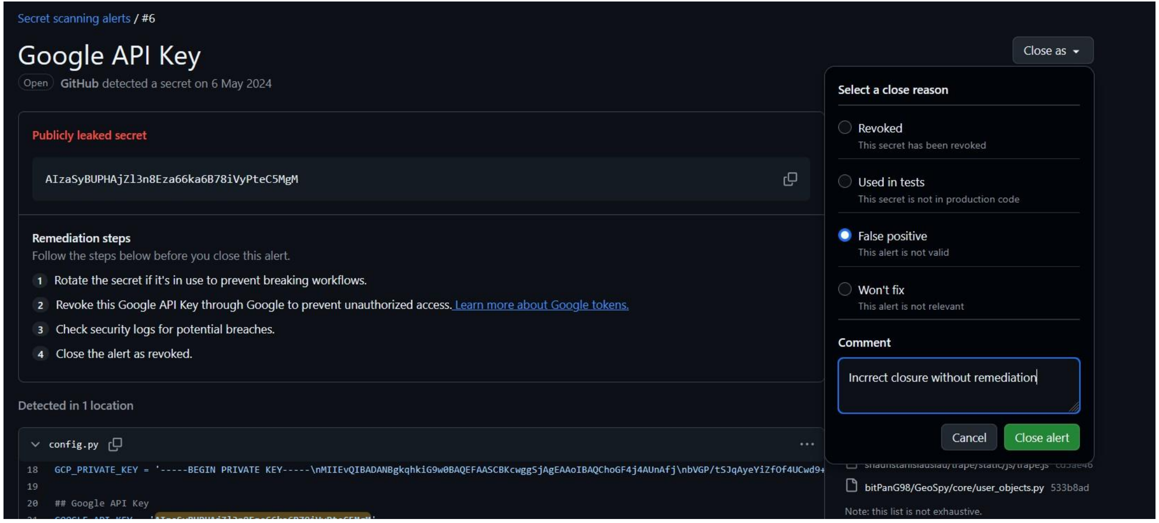
☐ **Slack API Token** `xoxb-730191371696-141386824…` Public leak
#5 opened on 6 May 2024 • Detected secret in config.py:50

☐ **Slack API Token** `xapp-1-A01C259PH2A-14407559…` Public leak
#4 opened on 6 May 2024 • Detected secret in config.py:47

☐ **Amazon AWS Secret Access Key** `UpUbsQANRHLf2uuQ7QOlNXPbbtV…` Public leak
#3 opened on 6 May 2024 • Detected secret in config.py:10

☐ **Amazon AWS Access Key ID** `AKIAIWSXFHRM7F6Z3NWQ` Public leak
#2 opened on 6 May 2024 • Detected secret in config.py:7

Incorrect Secret Scanning Alert Closure



Secret scanning alerts / #6

Google API Key

Open **GitHub** detected a secret on 6 May 2024

Publicly leaked secret

`AIzaSyBUPHAjZl3n8Eza66ka6B78iVyPteC5MgM`

Remediation steps

Follow the steps below before you close this alert.

1. Rotate the secret if it's in use to prevent breaking workflows.
2. Revoke this Google API Key through Google to prevent unauthorized access. Learn more about Google tokens.
3. Check security logs for potential breaches.
4. Close the alert as revoked.

Detected in 1 location

config.py

```
GCP_PRIVATE_KEY = '-----BEGIN PRIVATE KEY-----\nMIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQChoGF4j4AUnAfj\nbVGP/tSJqAyeYiZfOf4UCwd9+

## Google API Key
```

Close as ▾

Select a close reason

○ Revoked
This secret has been revoked

○ Used in tests
This secret is not in production code

● False positive
This alert is not valid

○ Won't fix
This alert is not relevant

Comment

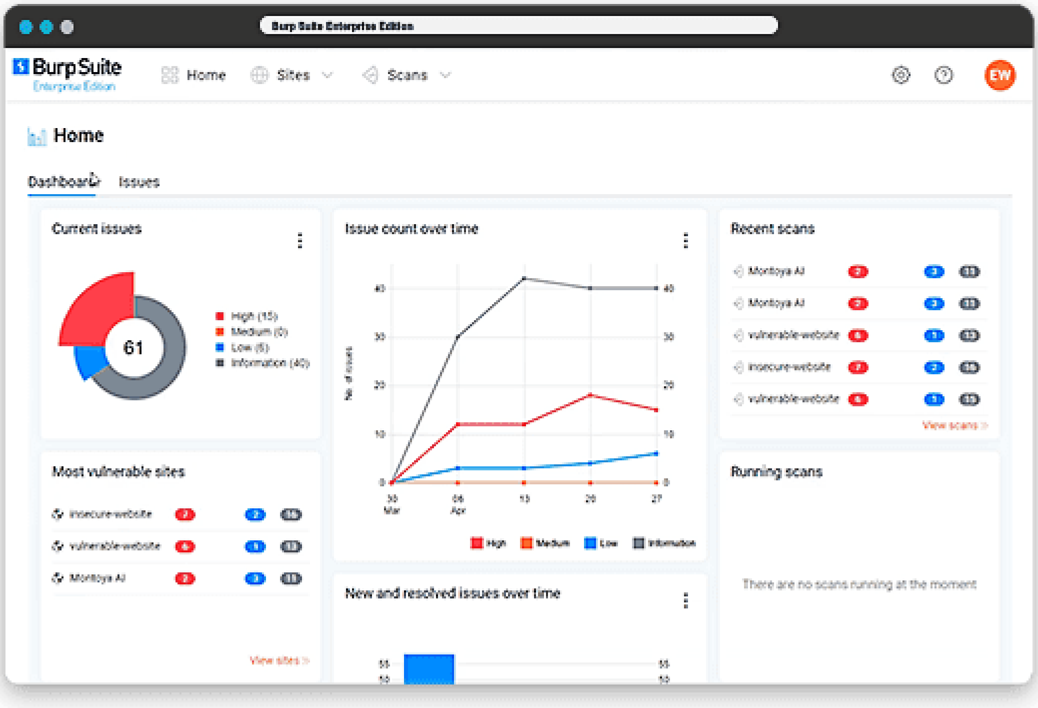
Incrrect closure without remediation

Cancel | Close alert

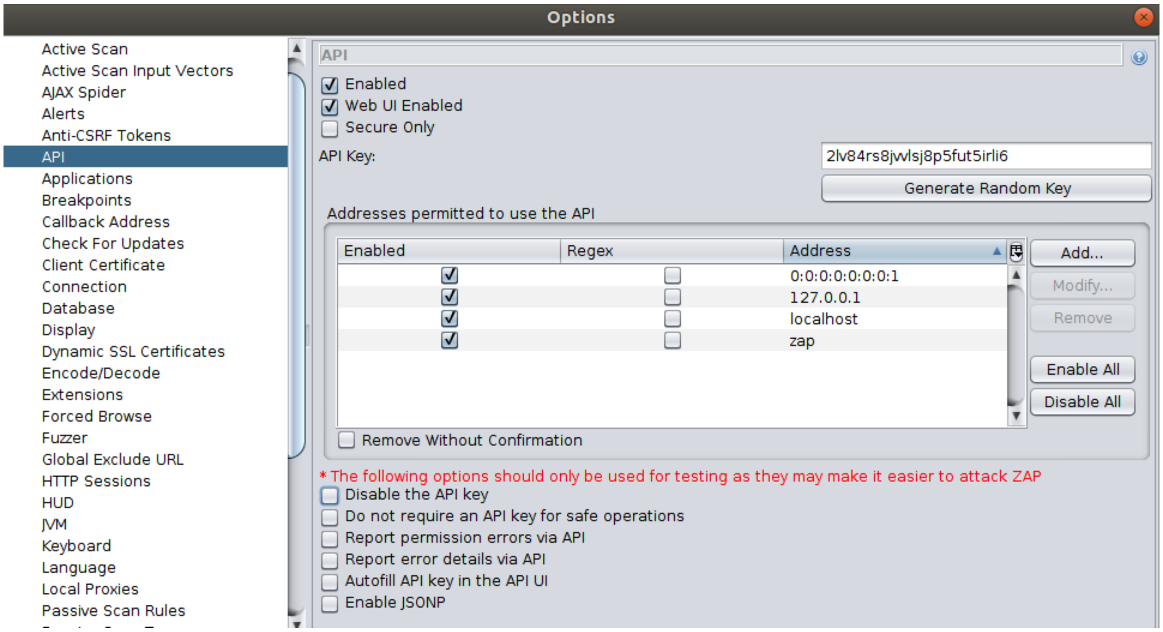
bitPanG98/GeoSpy/core/user_objects.py 533b8ad

Note: this list is not exhaustive.

Burp EE scanning alert



Owasp ZAP UI



Options

- Active Scan
- Active Scan Input Vectors
- AJAX Spider
- Alerts
- Anti-CSRF Tokens
- API
- Applications
- Breakpoints
- Callback Address
- Check For Updates
- Client Certificate
- Connection
- Database
- Display
- Dynamic SSL Certificates
- Encode/Decode
- Extensions
- Forced Browse
- Fuzzer
- Global Exclude URL
- HTTP Sessions
- HUD
- JVM
- Keyboard
- Language
- Local Proxies
- Passive Scan Rules

API

☑ Enabled
☑ Web UI Enabled
□ Secure Only

API Key: 2lv84rs8jvlsj8p5fut5irli6

Generate Random Key

Addresses permitted to use the API

Enabled	Regex	Address
☑	□	0:0:0:0:0:0:0:1
☑	□	127.0.0.1
☑	□	localhost
☑	□	zap

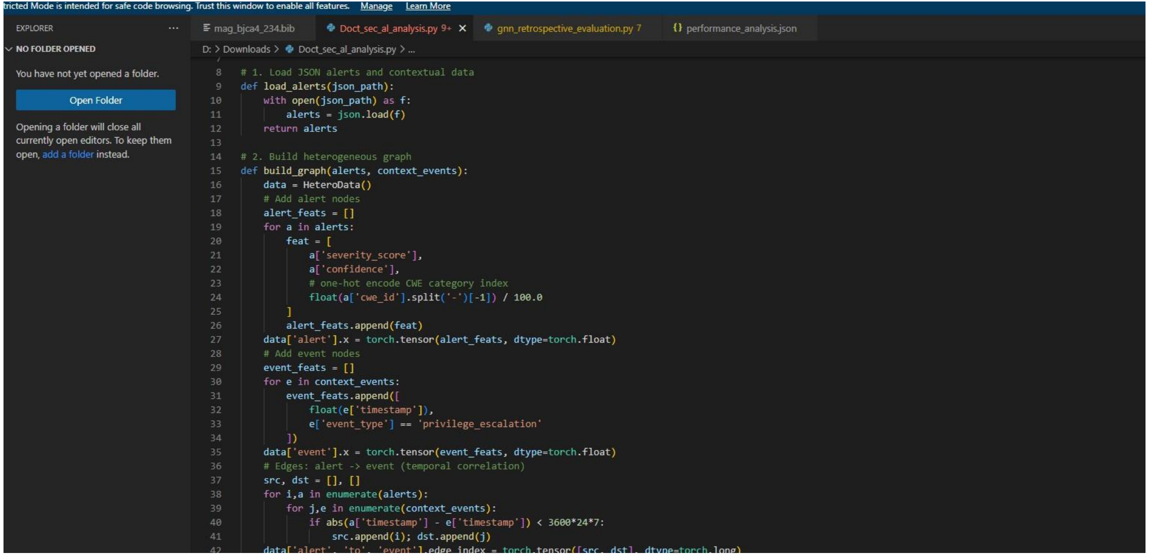
Add... Modify... Remove Enable All Disable All

□ Remove Without Confirmation

* The following options should only be used for testing as they may make it easier to attack ZAP

□ Disable the API key
□ Do not require an API key for safe operations
□ Report permission errors via API
□ Report error details via API
□ Autofill API key in the API UI
□ Enable JSONP

VS Code Training GNN



tricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More

EXPLORER

NO FOLDER OPENED

You have not yet opened a folder.

Open Folder

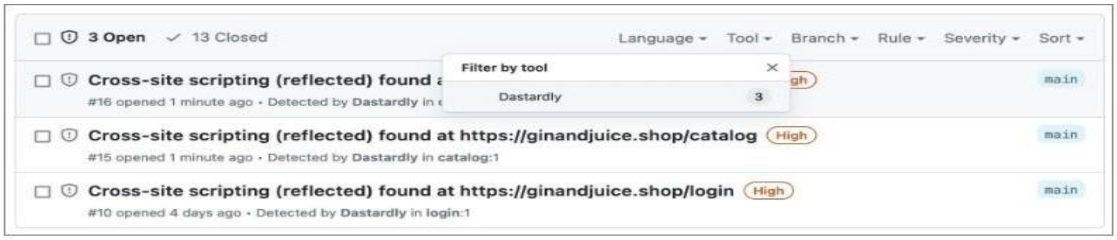
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

mag_bjca4_234.bib | Doct_sec_al_analysis.py 9+ | gnn_retrospective_evaluation.py 7 | performance_analysis.json

D: > Downloads > Doct_sec_al_analysis.py > ...

```python
# 1. Load JSON alerts and contextual data
def load_alerts(json_path):
    with open(json_path) as f:
        alerts = json.load(f)
    return alerts

# 2. Build heterogeneous graph
def build_graph(alerts, context_events):
    data = HeteroData()
    # Add alert nodes
    alert_feats = []
    for a in alerts:
        feat = [
            a['severity_score'],
            a['confidence'],
            # one-hot encode CWE category index
            float(a['cwe_id'].split('-')[-1]) / 100.0
        ]
        alert_feats.append(feat)
    data['alert'].x = torch.tensor(alert_feats, dtype=torch.float)
    # Add event nodes
    event_feats = []
    for e in context_events:
        event_feats.append([
            float(e['timestamp']),
            e['event_type'] == 'privilege_escalation'
        ])
    data['event'].x = torch.tensor(event_feats, dtype=torch.float)
    # Edges: alert -> event (temporal correlation)
    src, dst = [], []
    for i,a in enumerate(alerts):
        for j,e in enumerate(context_events):
            if abs(a['timestamp'] - e['timestamp']) < 3600*24*7:
                src.append(i); dst.append(j)
    data['alert', 'to', 'event'].edge_index = torch.tensor([src, dst], dtype=torch.long)
```

Running DAST in Github Actions

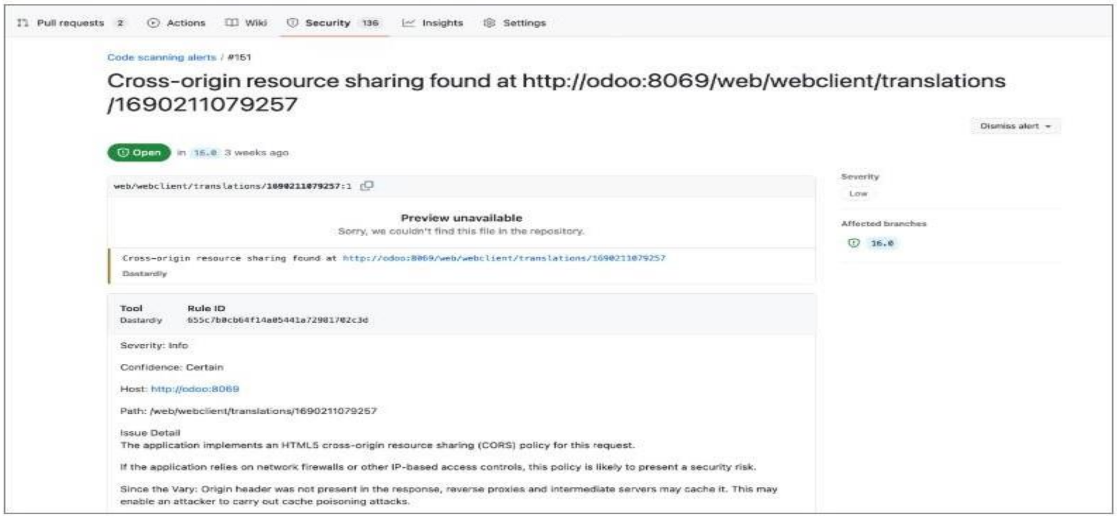


☐ Cross-site scripting (reflected) found a… High main
#16 opened 1 minute ago • Detected by Dastardly in …

☐ Cross-site scripting (reflected) found at https://ginandjuice.shop/catalog High main
#15 opened 1 minute ago • Detected by Dastardly in catalog:1

☐ Cross-site scripting (reflected) found at https://ginandjuice.shop/login High main
#10 opened 4 days ago • Detected by Dastardly in login:1

Dastardly results appear in your Security Center alerts



Cross-origin resource sharing found at http://odoo:8069/web/webclient/translations/1690211079257

web/webclient/translations/1690211079257:1

Preview unavailable
Sorry, we couldn't find this file in the repository.

Cross-origin resource sharing found at http://odoo:8069/web/webclient/translations/1690211079257
Dastardly

Tool	Rule ID
Dastardly	655c7b0cb64f14a05441a72981702c3d

Severity: Info

Confidence: Certain

Host: http://odoo:8069

Path: /web/webclient/translations/1690211079257

Issue Detail
The application implements an HTML5 cross-origin resource sharing (CORS) policy for this request.

If the application relies on network firewalls or other IP-based access controls, this policy is likely to present a security risk.

Since the Vary: Origin header was not present in the response, reverse proxies and intermediate servers may cache it. This may enable an attacker to carry out cache poisoning attacks.

Powershell script to parse and export the data

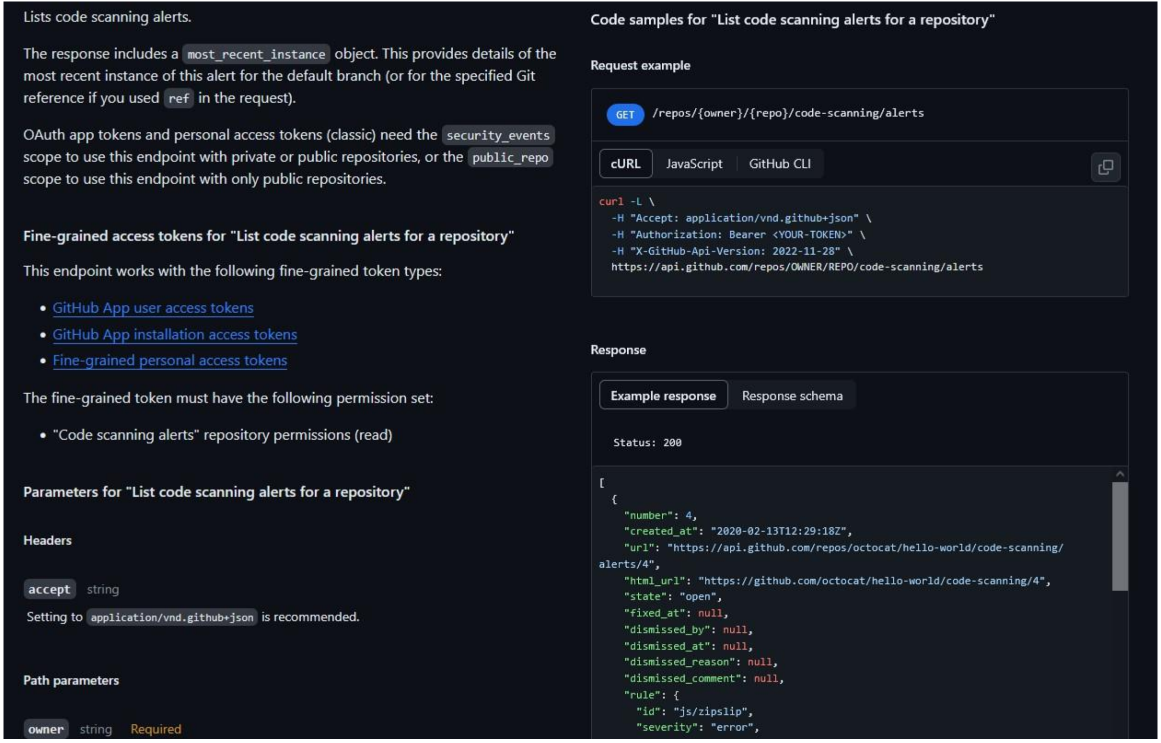
```
param(
  [Parameter(Mandatory=$true)][string]$Owner,                # org or user
  [string]$Repo,                                             # optional for single repo mode
  [Parameter(Mandatory=$true)][string]$Token,
  [switch]$AllRepos,                                         # enumerate all repos under $Owner
[string]$ApiBaseUrl = "https://api.github.com",          # GHES: https://ghe.example.com/api/v3
  [string]$OutDir = "./gh-alerts-export",
  [ValidateSet("open","fixed","dismissed","auto_dismissed","all")][string]$State = "all",
  [int]$PerPage = 100,
  [int]$MaxPages = 100,                        # safety cap
  [int]$Retry = 3,
  [int]$RetryDelaySec = 5,
  [datetime]$UpdatedSince,                     # filter for updated alerts (client-side)
  [switch]$MergeCsv                            # write 1 CSV per type across repos
)

# -------------- helpers --------------
function New-AuthHeader {
  param([string]$Token)
  @{
    "Authorization" = "Bearer $Token"
    "Accept"        = "application/vnd.github+json"
    "X-GitHub-Api-Version" = "2022-11-28"
    "User-Agent"    = "ps-gh-alert-exporter"
  }
}

function Invoke-GH {
  param(
    [string]$Method = "GET",
    [string]$Url,
    [hashtable]$Headers,
    [int]$Retry,
    [int]$RetryDelaySec
  )
  for ($i=0; $i -le $Retry; $i++) {
    try {
      return Invoke-RestMethod -Method $Method -Uri $Url -Headers $Headers -
TimeoutSec 120
    } catch {
      $statusCode = $_.Exception.Response.StatusCode.Value___
      if ($statusCode -eq 403 -and $_.ErrorDetails.Message -match '"rate limit exceeded"')
{
        Write-Warning "Rate limit hit. Waiting $RetryDelaySec sec..."
        Start-Sleep -Seconds $RetryDelaySec
      } else {
        if ($i -eq $Retry) {
          throw $_
        } else {
          Write-Warning "Request failed (attempt $($i+1)/$($Retry+1)) with status
$statusCode. Retrying in $RetryDelaySec sec..."
          Start-Sleep -Seconds $RetryDelaySec
        }
      }
    }
  }
}

function Get-AllPages {
  param(
    [string]$BaseUrl,
    [hashtable]$Headers,
    [int]$PerPage,
    [int]$MaxPages
  )
  $results = @()
  for ($page = 1; $page -le $MaxPages; $page++) {
    $url = "$BaseUrl&per_page=$PerPage&page=$page"
    $data = Invoke-GH -Url $url -Headers $Headers -Retry $Retry -RetryDelaySec
$RetryDelaySec
    if (-not $data -or $data.Count -eq 0) { break }
    $results += $data
    if ($data.Count -lt $PerPage) { break }
  }
  return $results
}

function Ensure-OutDir {
  param([string]$Path)
  if (-not (Test-Path $Path)) { New-Item -ItemType Directory -Path $Path | Out-Null }
}

function To-FlatObject {
  param([object]$o, [string]$RepoFullName, [string]$Type)
  # Produce flattened PSCustomObject depending on alert type
  switch ($Type) {
    "code" {
      # https://docs.github.com/rest/code-scanning/code-scanning?apiVersion=2022-11-28#list-code-scanning-alerts-for-a-repository
      [PSCustomObject]@{
        repo                       = $RepoFullName
        number                     = $o.number
        rule_id                    = $o.rule.id
        rule_severity              = $o.rule.severity
        rule_security_severity  = $o.rule.security_severity_level
        rule_description           = $o.rule.description
        tool_name                  = $o.tool.name
        tool_version               = $o.tool.version
        created_at                 = $o.created_at
        updated_at                 = $o.updated_at
        state                      = $o.state
        dismissed_reason           = $o.dismissed_reason
        dismissed_by_login         = $o.dismissed_by.login
        dismissed_at               = $o.dismissed_at
        fixed_at                   = $o.fixed_at
        most_recent_instance_ref= $o.most_recent_instance.ref
        most_recent_instance_path = $o.most_recent_instance.location.path
        most_recent_instance_start= $o.most_recent_instance.location.start_line
        most_recent_instance_end  = $o.most_recent_instance.location.end_line
        html_url             = $o.html_url
        # indicate if looks like DAST (heuristic: tool name contains 'ZAP' or 'DAST')
        is_dast_tool         = if ($o.tool.name -match '(?i)zap|dast|burp') { $true } else { $false }
      }
    }
    "secret" {
      # https://docs.github.com/rest/secret-scanning/secret-scanning?apiVersion=2022-11-28#list-alerts-for-a-repository
      [PSCustomObject]@{
        repo          = $RepoFullName
        number        = $o.number
        secret_type   = $o.secret_type
        secret_type_display_name = $o.secret_type_display_name
        state         = $o.state
        resolution    = $o.resolution
        resolved_by   = $o.resolved_by.login
        resolved_at   = $o.resolved_at
        created_at    = $o.created_at
        updated_at    = $o.updated_at
        html_url      = $o.html_url
        locations_url  = $o.locations_url
        push_protection_bypassed = $o.push_protection_bypassed
        validity      = $o.validity
      }
    }
    "secret-pp" {
      # Push protection bypass alerts
      # https://docs.github.com/rest/secret-scanning/secret-scanning?apiVersion=2022-11-28#list-push-protection-bypasses-for-a-repository
      [PSCustomObject]@{
        repo          = $RepoFullName
        alert_number  = $o.alert_number
        repository_id = $o.repository_id
        created_at    = $o.created_at
        updated_at    = $o.updated_at
        actor_login   = $o.actor.login
        secret_type   = $o.secret_type
        reason        = $o.reason
        commit_sha    = $o.commit_sha
        branch        = $o.branch_name
      }
    }
  }
}

function Write-CsvSafe {
  param([array]$Items, [string]$Path)
  if ($Items -and $Items.Count -gt 0) {
    $Items | Export-Csv -Path $Path -NoTypeInformation -Encoding UTF8
    Write-Host "Wrote $($Items.Count) rows -> $Path"
  } else {
    Write-Host "No rows -> $Path (skipped)"
  }
}

# -------------- enumerate repos --------------
$headers = New-AuthHeader -Token $Token
Ensure-OutDir -Path $OutDir

$repos = @()
if ($AllRepos) {
  # Try org first, then user
  $urlOrg =
"$ApiBaseUrl/orgs/$Owner/repos?type=all&sort=full_name&direction=asc"
try {
    $repos = Get-AllPages -BaseUrl $urlOrg -Headers $headers -PerPage $PerPage -
MaxPages $MaxPages
  } catch {
    # fallback to user
    $urlUser =
"$ApiBaseUrl/users/$Owner/repos?type=owner&sort=full_name&direction=asc"
    $repos = Get-AllPages -BaseUrl $urlUser -Headers $headers -PerPage $PerPage -
MaxPages $MaxPages
  }
```

```
} elseif ($Repo) {
  $repos = @([PSCustomObject]@{ full_name = "$Owner/$Repo"; name = $Repo })
} else {
  throw "Provide -Repo for single repository or -AllRepos to enumerate all under
$Owner."
}

if (-not $repos -or $repos.Count -eq 0) {
  throw "No repositories found for $Owner."
}

# If merging CSVs across repos, collect in-memory
$allCode = @()
$allSecret = @()
$allSecretPP = @()

# -------------- fetch alerts per repo --------------
foreach ($r in $repos) {
  $fullName = $r.full_name
  $repoName = ($r.name) ? $r.name : ($fullName.Split('/')[1])
  Write-Host "`n=== Repository: $fullName ==="

  # Map state filter
  $stateParam = if ($State -eq "all") { "" } else { "&state=$State" }
  # Code scanning alerts
  $codeBase = "$ApiBaseUrl/repos/$fullName/code-scanning/alerts?sort=created&direction=desc$stateParam"
  $codeAlerts = Get-AllPages -BaseUrl $codeBase -Headers $headers -PerPage $PerPage -MaxPages $MaxPages
  if ($UpdatedSince) {
    $codeAlerts = $codeAlerts | Where-Object { [datetime]$_.updated_at -ge $UpdatedSince }
  }
  $codeRows = $codeAlerts | ForEach-Object { To-FlatObject -o $_ -RepoFullName $fullName -Type "code" }
  if ($MergeCsv) { $allCode += $codeRows } else {
    $repoOutDir = Join-Path $OutDir $repoName
    Ensure-OutDir -Path $repoOutDir
    Write-CsvSafe -Items $codeRows -Path (Join-Path $repoOutDir "code-scanning-alerts.csv")
  }

  # Secret scanning alerts
  $secretBase = "$ApiBaseUrl/repos/$fullName/secret-scanning/alerts?sort=created&direction=desc$stateParam"
  $secretAlerts = Get-AllPages -BaseUrl $secretBase -Headers $headers -PerPage $PerPage -MaxPages $MaxPages
  if ($UpdatedSince) {
    $secretAlerts = $secretAlerts | Where-Object { [datetime]$_.updated_at -ge $UpdatedSince }
  }
  $secretRows = $secretAlerts | ForEach-Object { To-FlatObject -o $_ -RepoFullName $fullName -Type "secret" }
  if ($MergeCsv) { $allSecret += $secretRows } else {
    $repoOutDir = Join-Path $OutDir $repoName
    Ensure-OutDir -Path $repoOutDir
    Write-CsvSafe -Items $secretRows -Path (Join-Path $repoOutDir "secret-scanning-alerts.csv")
  }

  # Secret scanning push protection bypasses
  $ppBase = "$ApiBaseUrl/repos/$fullName/secret-scanning/alerts/push-protection-bypasses?sort=created&direction=desc"
  $ppAlerts = Get-AllPages -BaseUrl $ppBase -Headers $headers -PerPage $PerPage -MaxPages $MaxPages
  if ($UpdatedSince) {
    $ppAlerts = $ppAlerts | Where-Object { [datetime]$_.updated_at -ge $UpdatedSince }
  }
  $ppRows = $ppAlerts | ForEach-Object { To-FlatObject -o $_ -RepoFullName $fullName -Type "secret-pp" }
  if ($MergeCsv) { $allSecretPP += $ppRows } else {
    $repoOutDir = Join-Path $OutDir $repoName
    Ensure-OutDir -Path $repoOutDir
    Write-CsvSafe -Items $ppRows -Path (Join-Path $repoOutDir "secret-push-protection-bypasses.csv")
  }
}

# -------------- merged outputs --------------
if ($MergeCsv) {
  Ensure-OutDir -Path $OutDir
  Write-CsvSafe -Items $allCode -Path (Join-Path $OutDir "all-code-scanning-alerts.csv")
  Write-CsvSafe -Items $allSecret -Path (Join-Path $OutDir "all-secret-scanning-alerts.csv")
  Write-CsvSafe -Items $allSecretPP -Path (Join-Path $OutDir "all-secret-push-protection-bypasses.csv")
}

Write-Host "`nDone."
```

Github Rest API Doc:

https://docs.github.com/en/rest/code-scanning/code-scanning?apiVersion=2022-11-

28#list-code-scanning-alerts-for-a-repository



Lists code scanning alerts.

The response includes a `most_recent_instance` object. This provides details of the most recent instance of this alert for the default branch (or for the specified Git reference if you used `ref` in the request).

OAuth app tokens and personal access tokens (classic) need the `security_events` scope to use this endpoint with private or public repositories, or the `public_repo` scope to use this endpoint with only public repositories.

Fine-grained access tokens for "List code scanning alerts for a repository"

This endpoint works with the following fine-grained token types:

- GitHub App user access tokens
- GitHub App installation access tokens
- Fine-grained personal access tokens

The fine-grained token must have the following permission set:

- "Code scanning alerts" repository permissions (read)

Parameters for "List code scanning alerts for a repository"

Headers

`accept` string

Setting to `application/vnd.github+json` is recommended.

Path parameters

`owner` string Required

Code samples for "List code scanning alerts for a repository"

Request example

GET /repos/{owner}/{repo}/code-scanning/alerts

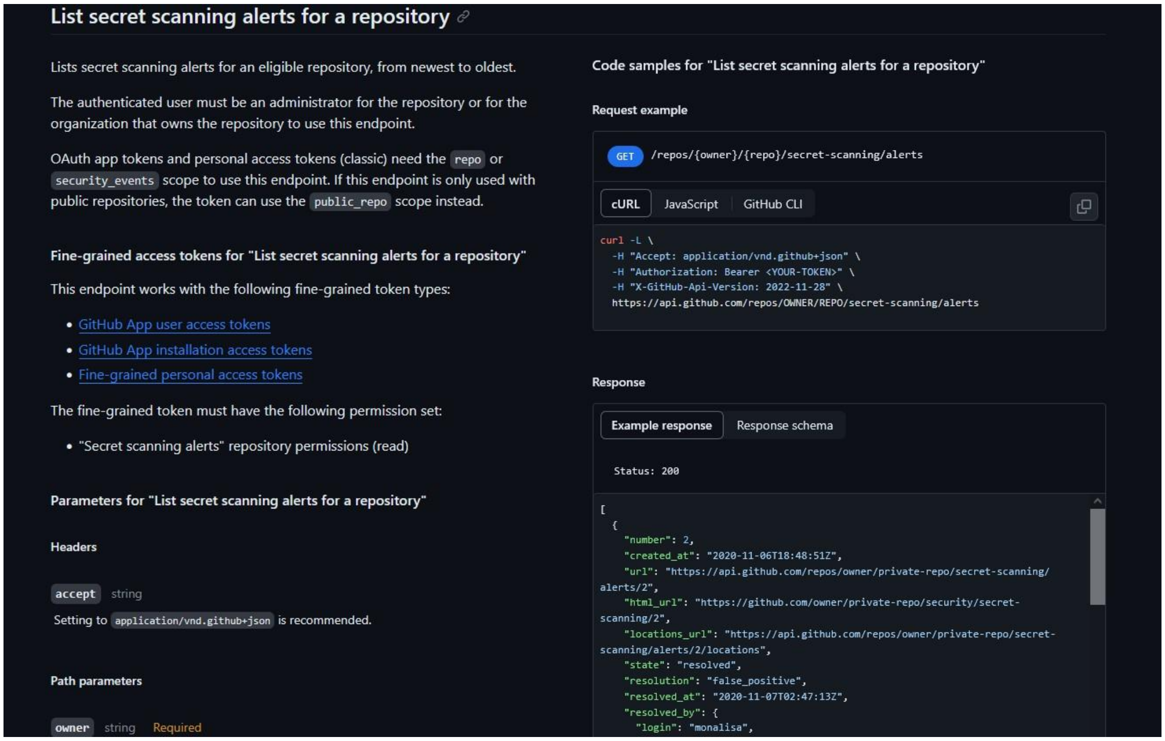
```
curl -L \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/OWNER/REPO/code-scanning/alerts
```

Response

Status: 200

```
[
  {
    "number": 4,
    "created_at": "2020-02-13T12:29:18Z",
    "url": "https://api.github.com/repos/octocat/hello-world/code-scanning/
alerts/4",
    "html_url": "https://github.com/octocat/hello-world/code-scanning/4",
    "state": "open",
    "fixed_at": null,
    "dismissed_by": null,
    "dismissed_at": null,
    "dismissed_reason": null,
    "dismissed_comment": null,
    "rule": {
      "id": "js/zipslip",
      "severity": "error",
```

https://docs.github.com/en/rest/secret-scanning/secret-scanning?apiVersion=2022-11-28#list-secret-scanning-alerts-for-a-repository



List secret scanning alerts for a repository

Lists secret scanning alerts for an eligible repository, from newest to oldest.

The authenticated user must be an administrator for the repository or for the organization that owns the repository to use this endpoint.

OAuth app tokens and personal access tokens (classic) need the `repo` or `security_events` scope to use this endpoint. If this endpoint is only used with public repositories, the token can use the `public_repo` scope instead.

Fine-grained access tokens for "List secret scanning alerts for a repository"

This endpoint works with the following fine-grained token types:

- GitHub App user access tokens
- GitHub App installation access tokens
- Fine-grained personal access tokens

The fine-grained token must have the following permission set:

- "Secret scanning alerts" repository permissions (read)

Parameters for "List secret scanning alerts for a repository"

Headers

`accept` string

Setting to `application/vnd.github+json` is recommended.

Path parameters

`owner` string Required

Code samples for "List secret scanning alerts for a repository"

Request example

GET /repos/{owner}/{repo}/secret-scanning/alerts

```
curl -L \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/OWNER/REPO/secret-scanning/alerts
```

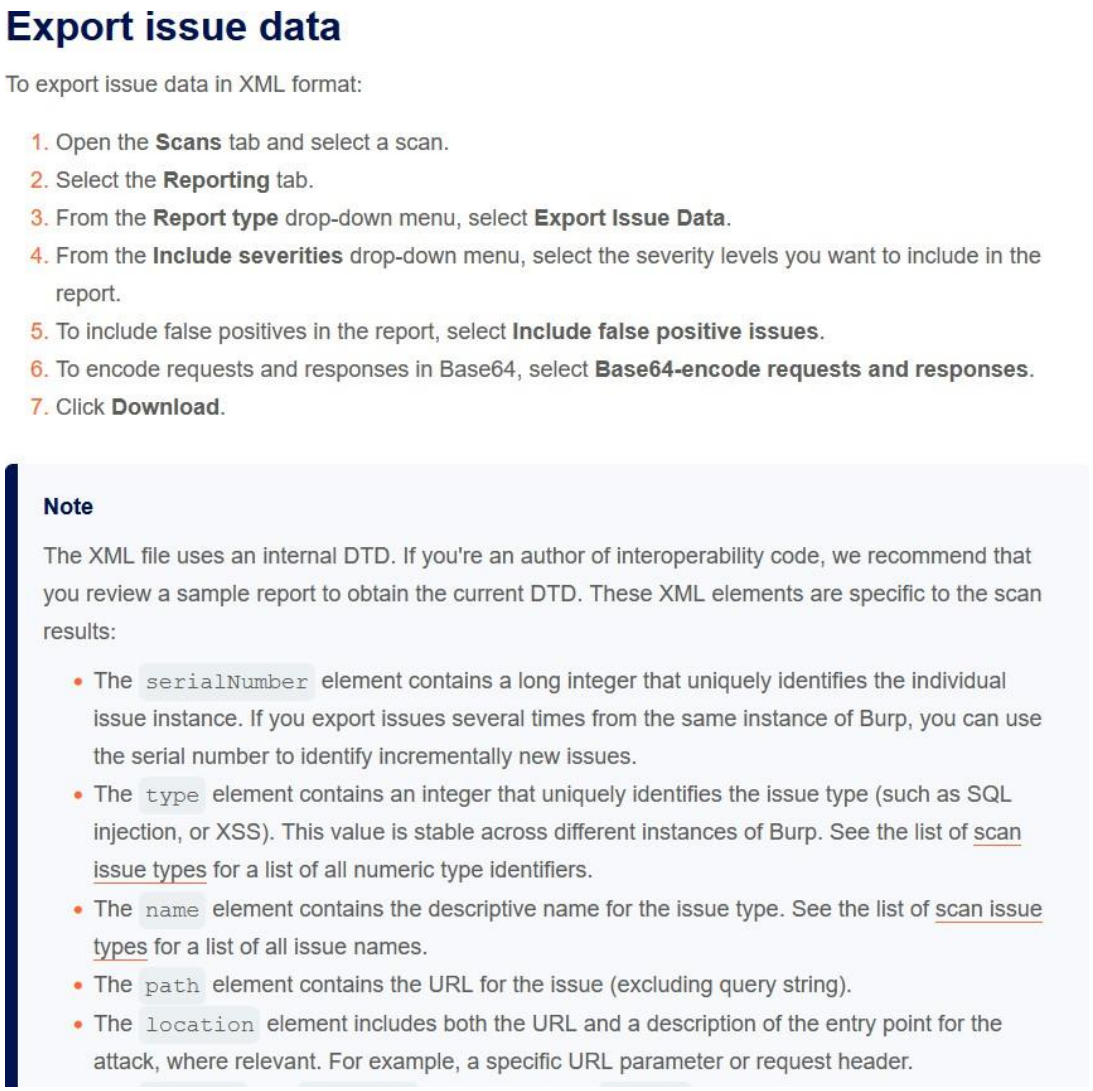
Response

```
Status: 200
```

```
[
  {
    "number": 2,
    "created_at": "2020-11-06T18:48:51Z",
    "url": "https://api.github.com/repos/owner/private-repo/secret-scanning/
alerts/2",
    "html_url": "https://github.com/owner/private-repo/security/secret-
scanning/2",
    "locations_url": "https://api.github.com/repos/owner/private-repo/secret-
scanning/alerts/2/locations",
    "state": "resolved",
    "resolution": "false_positive",
    "resolved_at": "2020-11-07T02:47:13Z",
    "resolved_by": {
      "login": "monalisa",
```

DAST Alerts fetch

https://portswigger.net/burp/documentation/dast/user-guide/work-with-scan-results/generate-reports



Export issue data

To export issue data in XML format:

1. Open the **Scans** tab and select a scan.
2. Select the **Reporting** tab.
3. From the **Report type** drop-down menu, select **Export Issue Data**.
4. From the **Include severities** drop-down menu, select the severity levels you want to include in the report.
5. To include false positives in the report, select **Include false positive issues**.
6. To encode requests and responses in Base64, select **Base64-encode requests and responses**.
7. Click **Download**.

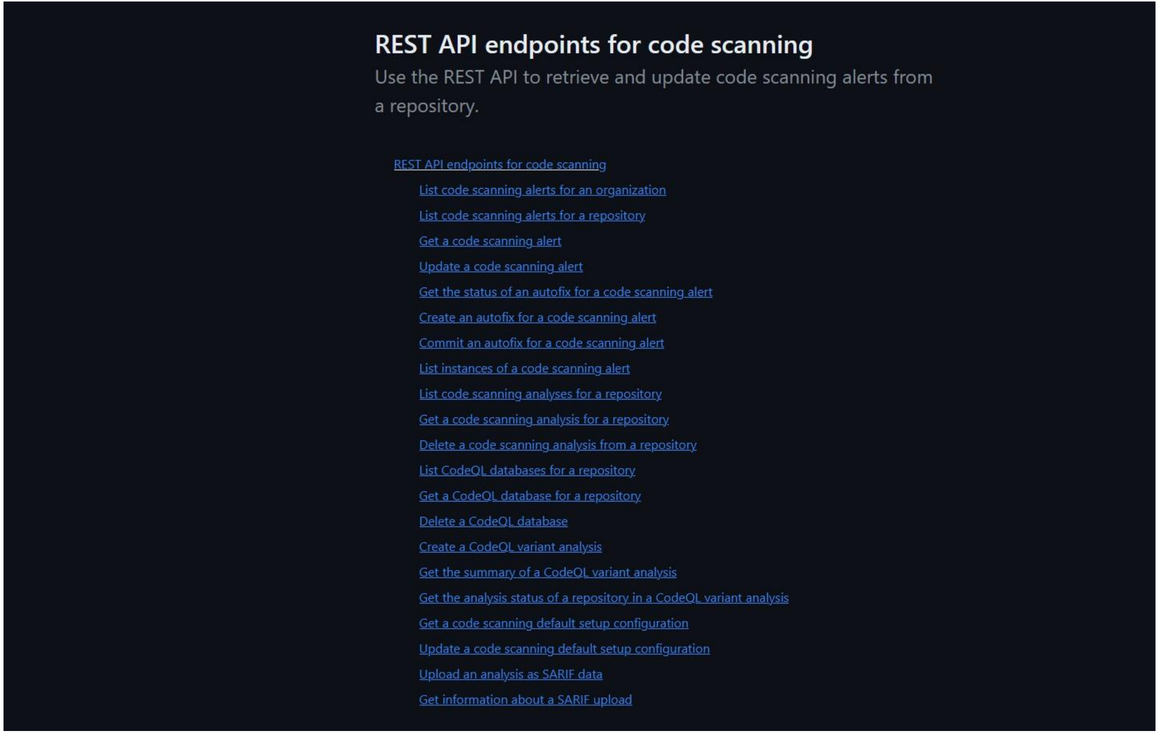
Note

The XML file uses an internal DTD. If you're an author of interoperability code, we recommend that you review a sample report to obtain the current DTD. These XML elements are specific to the scan results:

- The `serialNumber` element contains a long integer that uniquely identifies the individual issue instance. If you export issues several times from the same instance of Burp, you can use the serial number to identify incrementally new issues.
- The `type` element contains an integer that uniquely identifies the issue type (such as SQL injection, or XSS). This value is stable across different instances of Burp. See the list of scan issue types for a list of all numeric type identifiers.
- The `name` element contains the descriptive name for the issue type. See the list of scan issue types for a list of all issue names.
- The `path` element contains the URL for the issue (excluding query string).
- The `location` element includes both the URL and a description of the entry point for the attack, where relevant. For example, a specific URL parameter or request header.

Github code scanning Rest API endpoint:

https://docs.github.com/en/rest/code-scanning?apiVersion=2022-11-28

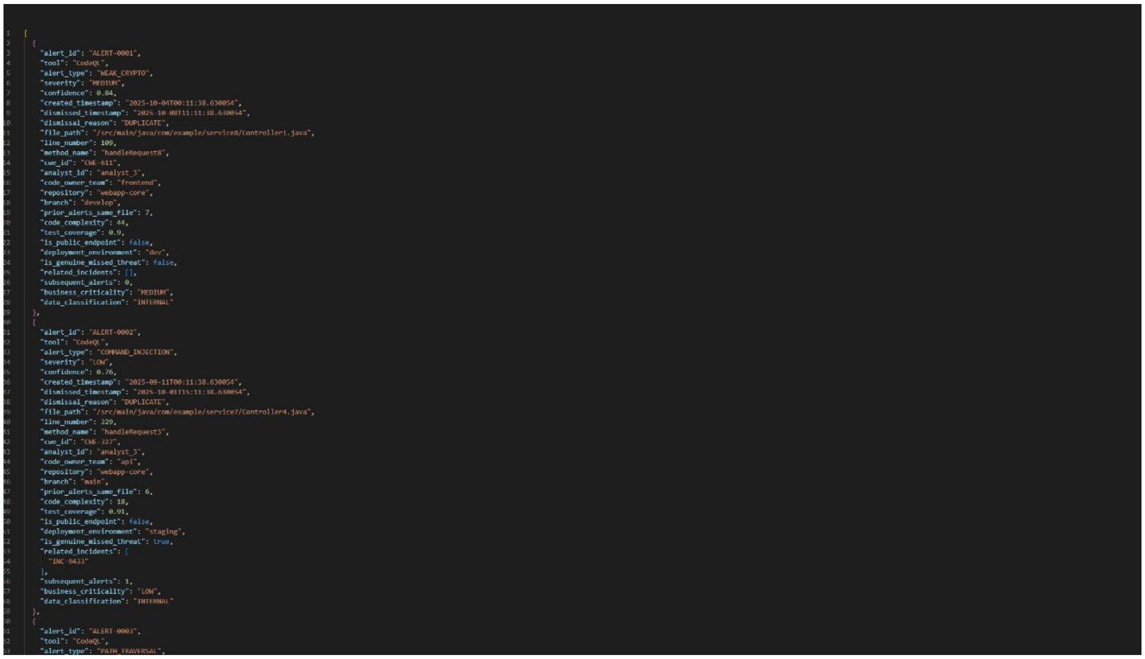


REST API endpoints for code scanning

Use the REST API to retrieve and update code scanning alerts from a repository.

- REST API endpoints for code scanning
  - List code scanning alerts for an organization
  - List code scanning alerts for a repository
  - Get a code scanning alert
  - Update a code scanning alert
  - Get the status of an autofix for a code scanning alert
  - Create an autofix for a code scanning alert
  - Commit an autofix for a code scanning alert
  - List instances of a code scanning alert
  - List code scanning analyses for a repository
  - Get a code scanning analysis for a repository
  - Delete a code scanning analysis from a repository
  - List CodeQL databases for a repository
  - Get a CodeQL database for a repository
  - Delete a CodeQL database
  - Create a CodeQL variant analysis
  - Get the summary of a CodeQL variant analysis
  - Get the analysis status of a repository in a CodeQL variant analysis
  - Get a code scanning default setup configuration
  - Update a code scanning default setup configuration
  - Upload an analysis as SARIF data
  - Get information about a SARIF upload

Test JSON



```json
[
  {
    "alert_id": "ALERT-0001",
    "tool": "CodeQL",
    "alert_type": "WEAK_CRYPTO",
    "severity": "MEDIUM",
    "confidence": 0.84,
    "created_timestamp": "2025-10-04T00:11:38.630054",
    "dismissed_timestamp": "2025-10-08T11:11:38.630054",
    "dismissal_reason": "DUPLICATE",
    "file_path": "/src/main/java/com/example/service8/Controller1.java",
    "line_number": 109,
    "method_name": "handleRequest8",
    "cwe_id": "CWE-611",
    "analyst_id": "analyst_3",
    "code_owner_team": "frontend",
    "repository": "webapp-core",
    "branch": "develop",
    "prior_alerts_same_file": 7,
    "code_complexity": 44,
    "test_coverage": 0.9,
    "is_public_endpoint": false,
    "deployment_environment": "dev",
    "is_genuine_missed_threat": false,
    "related_incidents": [],
    "subsequent_alerts": 0,
    "business_criticality": "MEDIUM",
    "data_classification": "INTERNAL"
  },
  {
    "alert_id": "ALERT-0002",
    "tool": "CodeQL",
    "alert_type": "COMMAND_INJECTION",
    "severity": "LOW",
    "confidence": 0.76,
    "created_timestamp": "2025-09-11T00:11:38.630054",
    "dismissed_timestamp": "2025-10-01T15:11:38.630054",
    "dismissal_reason": "DUPLICATE",
    "file_path": "/src/main/java/com/example/service7/Controller4.java",
    "line_number": 329,
    "method_name": "handleRequest3",
    "cwe_id": "CWE-327",
    "analyst_id": "analyst_3",
    "code_owner_team": "api",
    "repository": "webapp-core",
    "branch": "main",
    "prior_alerts_same_file": 6,
    "code_complexity": 18,
    "test_coverage": 0.91,
    "is_public_endpoint": false,
    "deployment_environment": "staging",
    "is_genuine_missed_threat": true,
    "related_incidents": [
      "INC-9433"
    ],
    "subsequent_alerts": 1,
    "business_criticality": "LOW",
    "data_classification": "INTERNAL"
  },
  {
    "alert_id": "ALERT-0003",
    "tool": "CodeQL",
    "alert_type": "PATH_TRAVERSAL",
```

Github Rest API Documentation relevant to the study

https://docs.github.com/en/rest/code-scanning/code-scanning?apiVersion=2022-11-28#list-code-scanning-alerts-for-a-repository

https://docs.github.com/en/rest/code-scanning/code-scanning?apiVersion=2022-11-28#get-a-code-scanning-analysis-for-a-repository

https://docs.github.com/en/rest/secret-scanning/secret-scanning?apiVersion=2022-11-28#list-secret-scanning-alerts-for-a-repository

https://docs.github.com/en/rest/secret-scanning/secret-scanning?apiVersion=2022-11-28#get-secret-scanning-scan-history-for-a-repository

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/security-log-events