Continuous Verification of Open Source Components in a World of Weak Links

Thomas Hastings and Kristen R. Walcott
Department of Computer Science
University of Colorado Colorado Springs
Colorado, United States
{thasting,kjustice}@uccs.edu

***Abstract*—We are heading for a perfect storm, making open source software poisoning and next-generation supply chain attacks much easier to execute, which could have major implications for organizations. The widespread adoption of open source (99% of today's software utilizes open source), the ease of today's package managers, and the best practice of implementing continuous delivery for software projects provide an unprecedented opportunity for attack. Once an adversary compromises a project, they can deploy malicious code into production under the auspicious of a software patch. Downstream projects will ingest the compromised patch, and now those projects are potentially running the malicious code. The impact could be implementing backdoors, gathering intelligence, delivering malware, or denying a service. According to Sonatype, a leading commercial software security company, these next-generation supply chain attacks have increased 430% in the last year and there is not a good way to vet or monitor an open-source project prior to incorporating the project.**

In this paper, we analyzed two case studies of compromised open source components. We propose six continuous verification controls that enable organizations to make data-driven decisions and mitigate breaches, such as analyzing community metrics and project hygiene using scorecards and monitoring the boundary of the software in production. In one case study, the controls identified high levels of risk immediately even though the package is widely used and has over 7 million downloads a week. In both case studies we found that the controls could have prevented malicious actions despite the project breaches.

***Index Terms*—Secure Software Development, Open Source Components, Code Re-use**

I. Introduction

We are heading towards a perfect storm for insecure and malicious software to make its way into production software stacks. The rise of open-source component utilization, the lack of project vetting techniques, and the overwhelming sense to deliver value faster have left us vulnerable to attack. Open-source software is utilized in 99% of software applications today [1]. Unfortunately, many software engineers rely on limited defensive techniques when vetting software projects, such as looking at recent activity within a project before incorporating the project into their project's software baseline [2]. This method offers very little protection for software projects.

Currently, there is no good way to manage the weak links of open-source packages [3]. To make matters worse, best practices are unwittingly putting organizations at risk. One reason is that open source components allow developers to incorporate new features seamlessly and effortlessly with only minor modifications. This code re-use enables software engineers to deliver value faster to their customer base. This practice is encouraged within organizations and is referenced by the National Institute of Science and Technology (NIST) in their publication for Secure Software Development [4] as a best practice. It is no surprise that modern programming languages have capitalized on the efficiencies of code re-use from open source components and made it easier than ever by providing package managers.

Many package managers use semantic versioning, allowing developers to automatically pull the latest major, minor, or patch versions of components each time they run a build. This convenience is excellent for ensuring the software project is always up to date with all the latest dependency releases, but what happens when the newest release is malicious? As more open-source projects adopt and implement continuous integration (CI) and continuous delivery (CD) pipelines in their builds, it will be easier than ever for an adversary to poison a project and release the malicious code into the wild under the auspiciousness of a simple patch.

There are over 5,000 open-source security advisories on GitHub today [5]. The zero trust model assumes that every project will have some security findings even without targeted supply chain attacks. Therefore, we need better methods of vetting and monitoring open source components throughout the component's life-cycle. We can no longer trust packages straight off the internet, and we need to verify that the packages are doing what they said they would do.

In our work, we develop methods for continuous verification throughout a components life-cycle to provide insight for understanding and monitoring the risk of incorporating open-source components. We take a holistic approach to analyzing components for risk of underlying vulnerabilities. We define six controls that organizations can use to protect themselves from malicious supply chain attacks. Specifically, we utilize the controls to create a repeatable method to understand the risk of incorporating an open-source component throughout it's life-cycle.

This paper makes the following contributions:

- Procedures for organizations to execute to understand open-source component risks before incorporating (Section III)

- An automated systems architecture vetting open-source applications (Section IV)
- Discussion of insights that could lead to more refinement in the way vetting happens for open source software components (Section V)

II. Background and Related Work

We took a multidisciplinary approach to solve our research goals by leveraging lessons learned and research from the fields of cyber security and cloud platform engineering in addition to software engineering for a truly holistic solution leveraging DevSecOps.

A. Development

Software supply chain attacks targeting open-source components have increased 430% in the last year [6]. These exploits continue to grow in frequency and magnitude. Over the previous two years, this topic has become a priority among organizational leaders and researchers. As a result, the software community has identified weak links in packages [3], created standards to highlight best practices for component vetting [7], automated vulnerability look-ups for dependencies [8], and widened the aperture for reporting vulnerabilities and malicious packages [9].

GitHub, the owners of NPM, have double-downed on their commitment to the NPM registry due to the increasing attacks on NPM packages. GitHub is now requiring 2FA for NPM package maintainers [10]. This is a step in the right direction as researchers have identified that hacking 20 high-profile developer accounts could compromise half of the NPM ecosystem [11]. Although it is not a complete solution as researchers have identified six core weak links in the NPM ecosystem. One of which is that maintainers are using expired domains for their email accounts [3]

So, how do we overcome weak links in our supply chain? The Open Source Security Foundation has been doing a lot of research into the topic of software supply chain protection as well. One of the projects we use extensively in our study is their Security Scorecards for Open Source Projects. Although we did not use all the metrics, the scorecards provided and added a couple this research heavily influenced our research. According to the researchers from Google, ”The goal of Scorecards is to auto-generate a ’security score’ for open source projects to help users decide the trust, risk, and security posture for their use case. This data can also be used to augment any decision making in an automated fashion when new open source dependencies are introduced inside projects or at organizations” [7].

Understanding the package ecosystem and the community support around a package is essential, but a more holistic view is required to understand the actual risks before incorporating open source packages. The MITRE Corporation has been a faithful steward of maintaining two critical databases used for static code analysis. The first database is the Common Vulnerabilities and Exposures (CVE) database. This database contains a list of known vulnerabilities in packages [12]. The second database they steward is the Common Weakness Enumeration database. This database contains ”weakness types for software and hardware and is used as a baseline for weakness identification, mitigation, and prevention” [13].

We leverage the CVEs and the CWEs in our methods to identify known vulnerabilities in the open-source component and its dependencies. Then we use the CWEs to check for known code signatures that allow the package to be compromised if measures are not implemented to prevent malicious attacks.

B. Security

The software community is not the only community handling an unprecedented rise in cyber attacks or supply chain exploits. The corporate information technology (IT) security communities have been handling and defending malicious attacks for decades. There is a couple of stand-out models information technology groups use to protect their organizations from attacks. Many IT organizations understand what connects to their networks, have a plan to manage those assets, practice zero trust, and implement defense-in-depth.

NIST describes zero trust as, ”Zero trust (ZT) is the term for an evolving set of cybersecurity paradigms that move defenses from static, network-based perimeters to focus on users, assets, and resources” [14]. NIST defines defense-in-depth as, ”information security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and dimensions of the organization” [15]. Our methods implement elements from zero trust using defense-in-depth concepts by using automated tooling and policy throughout the life cycle of an open-source component.

C. Operations

There is a new way to think about the operations life-cycle, and some have called it the software development life-cycle in the cloud age [16]. The premise is that organizations can manage their IT operations using day 0, day 1, and day 2 nomenclatures. We leverage day 0, day 1, and day 2 in our evaluation. Day 0 is the design phase where an organization considers how or if it will incorporate a new open source component into a software project. Once the decision had been made to use the open-source component, we move into day 1 operations. Day 1 is what goes into actually incorporating the component. This may include adding the component to the package or vendorizing the component to make it available to the organization [17]. Day 2 operations occur after the package is included and running in production, this is when maintenance and monitoring becomes a priority.

III. Methodology

We have taken the lessons learned from the corporate information security community and applied those lessons to help vet open-source software components. We evaluated our methods using 2 case studies, one from NPM’s malicious

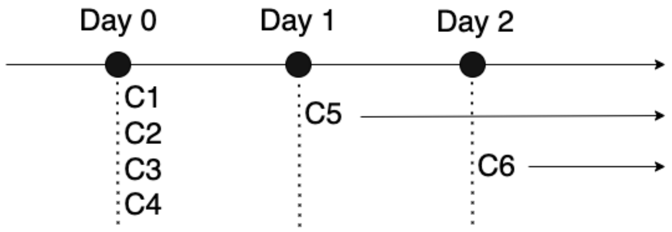

Fig. 1: Controls in the Life-Cycle

modules reports and one from RubyGems and ran a table-top exercise using our methods with the events in each case study. We identified at what stage our controls would have recognized the packages as a potential problem on day 0, day 1, and day 2 as we see in figure 1.

In our research we use 6 controls. Each control can be used independently or in succession to provide better risk mitigation using a defense-in-depth approach. Our controls are grouped into three risk categories based on relative 'knowness': the known knowns, the known unknowns, and the unknown unknowns [18].

A. The Known Knowns

We can learn a lot of information about the current state of a project by looking at the package's dependencies, source code, and community. These four controls are used to understand the current state of a package and the package's community and can inform the decision of whether or not to use the package. These four controls will be used throughout the package's life cycle, beginning at day 0.

C1: checks the package for known vulnerabilities in a package's dependencies and in the package itself. This is accomplished by leveraging a query against the CVE database using a tool like Depdendabot or Snyk.

C2: checks the source code for known weaknesses in the code base using static code analysis, which leverages CWE information.

C3: looks at the package's community to understand the makeup of the project's maintainers. This control looks at the number of companies maintaining a project and recent activity within the last 90 days. Multiple companies supporting a project and recent activity are signs of a healthy project.

C4: looks at the hygiene of the package. This control looks for dangerous workflows, signed-commits, signed-packages, and branch protection.

B. The Known Unknowns

Once a package moves into Day 1 and has been included in our software, we must leverage additional controls to protect against the known unknowns. A core tenant of security tells us to assume everything will be compromised at some point. Whether by a dependency, a code update with a bug, or a malicious maintainer.

C5: is a policy that dictates that no open-source artifact will be included, which is not built by a trusted source. The open-source component's source code will be forked and built in-house, and the artifact will be stored in a central location. When a new package version is released, we will ingest the change, conduct a secure code review [19], and scan the code using C1, C2, and C3. This is a heavy policy that gives the best chance to detect a problem with a dependency, new code that can be exploited, or code added by a malicious maintainer. This is especially true when we can not trust that packages published on sites such as NPM or RubyGems came from their respective communities [10] [20].

C. The Unknown Unknowns

After a package has been built internally and ingested into a software project, we need to protect that project from the unknown unknowns of the package. These unknown unknowns are malicious items we missed using the first five controls and are typically identified during day 2 operations when the package is in a running environment.

C6: is a network perimeter defense around the development and production environment of the software project. This perimeter runs at the network layer of the OSI model and executes packet inspection. This control monitors the activity of the software as it passes data over the network and onto the internet. The control will act on unexpected changes in the software's behavior or when the software attempts to pass or retrieve data from unknown sources.

IV. Results

This section describes the results of our findings across the use cases. We applied the methods to two components that were compromised. The first component, UAParser.js, came from the NPM ecosystem and the second, rest-client, came from the RubyGems ecosystem. These components were prime candidates because they averaged hundreds of downloads a week.

A. Case Study: UAParser.js

UAParser.js is a "JavaScript library to detect Browser, Engine, OS, CPU, and Device type/model from User-Agent data with relatively small footprint" [21]. On October 23, 2021, this NPM package was modified to include malicious code by an outside actor using a maintainers' compromised account. The malicious code injected into the UAParser.js package attempted to install coinminer, and harvest user/credential information [22]. Big tech companies such as Facebook, Slack, IBM, HPE, Dell, Oracle, Mozilla, Shopify, and Reddit used the plugin. Users reported downloading trojans to their local environments after updating the compromised release. The package owner was notified by the community and remediated the problem in a couple of hours, and pushed updated releases soon after that removed the malicious components.

We put the package through our controls using the package's community metrics as they are today. For the static code analysis, we checked out the project's repository and went back in time to a commit that occurred closest to but before the breach happened which was a commit from October 6, 2021 [23].

1) Control Execution:
We executed the 6 controls using the UAParser.js project. Below are our findings.

C1. UAParser.js does not utilize third-party dependencies, so this task was simple and limits the attack surface of the package.

C2. We leveraged eslint, semgrep, and nodejs-scan tools to perform static code analysis. As figure 2 shows, we identified 14 critical and 18 medium vulnerabilities. Although all these findings may not be actionable by a malicious actor, it gives the organization insight into potential attack vectors to which the package opens the organization up. At this point, the organization can accept the risk or triage and confirm the findings.

SCORE	NAME	REASON
10 / 10	Dependencies	0 dependencies found
0 / 10	Static Code Analysis	14 critical and 18 medium vulnerabilities identified

Fig. 2: UAParser.js - C1 and C2 Metrics

C3. We utilized metrics and scoring from the scorecards. As figure 3 shows, this package has maintenance support from 10 different companies. This provides confidence that the package is maintained because it is in the best interests of the companies maintaining it. However, the package does not appear to be well maintained. In the last 90 days, there have only been 2 commits. We also did not detect any activity in project issues by collaborators, members, or owners of the project in the last 90 days, but 3 new issues opened in that time.

SCORE	NAME	REASON
10 / 10	Contributors	10 different companies found
1/10	Maintained	2 commit(s) and 0 issue activity found by maintainers in the last 90 days

Fig. 3: UAParser.js - C3 Metrics

C4. We utilized the branch protection and dangerous workflow metrics from the scorecards in this control. Unfortunately, the scorecards do not provide a way to check for signed commits or packages outside of GitHub. This project publishes packages to NPM. Our findings for C4 are in figure 4. We identified that the project does not include branch protection on the main or release branches. We also identified that the package does not require maintainers to use signed commits, and the packages are not signed.

C5. Building the package did not take much effort. We had already forked the repository from GitHub into GitLab for our analysis. The package does not depend on outside dependencies, so a simple 'node install' was sufficient. We can ingest new changes from the upstream project and validate them using a code review before ingesting and publishing the new package. Figure 5 shows our CI pipeline for implementing this and previous controls' build, test and publish phases.

SCORE	NAME	REASON
0 / 10	Branch-Protection	protection not enabled
10 / 10	Dangerous-Workflow	no dangerous workflows
3 / 10	Signed-Commits	used in some cases
0 / 10	Signed-Packages	false

Fig. 4: UAParser.js - C4 Metrics

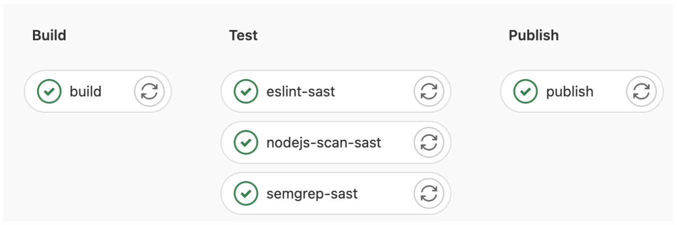

Fig. 5: UAParser.js - C5 pipeline

C6. This control would have identified the package attempting to contact an unknown URL to download the coinminer.

2) Evaluation:
The controls identified many potential problems with the package beginning on day 0, culminating with unexpected behavior identified on day 2.

day 0. While performing controls 1-4, the we identified some indicators that would make it hard to defend the project against weak links. First, there were already critical vulnerabilities in the code base that malicious actors could exploit. The project no longer seemed to be maintained despite many different corporate contributors. The project's community did not protect the main or release branches from maintainers directly committing to those branches and did not enforce signed commits or signed packages. These controls alone may have identified enough risk for an organization to choose not to use the package.

day 1. Executing C5 and bringing the package under internal control was simple because the package resided in one repository and did not have external dependencies. As a result, we were able to build the project and publish the resulting artifact. If an organization had chosen to use this project before October 23, 2021, this control would have required the organization to conduct a code review before ingesting the malicious code. This code review may have identified the malicious code.

day 2. The observability that C6 provides would have provided another chance for the organization to identify the malicious behavior of the package and take actions to protect itself had the code review in C5 missed it. The malicious code went outside of the network to download malicious tooling. C6 would have identified the request, flagged it, and created a notification in the architecture.

Overall the methods identified many risks within the UA-Parser.js package. Ultimately, it is up to the organization to accept the risk. However, our controls would still have provided protection even with an organization accepting the risk of using the UAParser.js before October 23, 2021.

B. Case Study: rest-client Gem

REST Client is a ”simple HTTP and REST client for Ruby, inspired by the Sinatra’s microframework style of specifying actions: get, put, post, delete” [24]. The package was modified on August 14, 2019, to include a malicious backdoor. The malicious actor had a script that would download code from Pastebin.com that was reportedly used to mine cryptocurrency. On August 19, a CVE was generated, RubyGems removed the affected Gems, and the community pulled the malicious code [25]. As with the last case study, we checked out the project’s repository and went back in time to a commit that occurred closest to but before the breach happened, which was a commit from March 28, 2019 [26].

1) Execution:

We executed the 6 controls using the rest-client project. Below are our findings.

C1. Like the UAParser.js, this project does not utilize third-party dependencies, so this task was simple and limits the attack surface of the package.

c2. We leveraged the brakeman tool to perform static code analysis. Unsurprisingly, we did not find any vulnerabilities in the source code as figure 6 shows. The project is mature, and the functionality does not change often.

SCORE	NAME	REASON
10 / 10	Dependencies	0 dependencies found
10 / 10	Static Code Analysis	0 vulnerabilities identified

Fig. 6: rest-client - C1 and C2 Metrics

C3. We utilized metrics and scoring from the scorecards. As figure 7 shows, this package has maintenance support from 26 different companies. There have not been any commits in the last 90 days, and we did not detect any activity in project issues by collaborators, members, or owners of the project in the last 90, and there have not been any issues opened in the last 90 days.

SCORE	NAME	REASON
10 / 10	Contributors	26 different companies found
0/10	Maintained	0 commit(s) and 0 issue activity found by maintainers in the last 90 days

Fig. 7: rest-client - C3 Metrics

C4. We utilized the branch protection and dangerous workflow metrics from the scorecards in this control. Our findings for C4 are in figure 8. We identified that the project does not include branch protection on the main or release branches. We also identified that the package does not require maintainers to use signed commits, and the packages are not signed.

SCORE	NAME	REASON
0 / 10	Branch-Protection	protection not enabled
10 / 10	Dangerous-Workflow	no dangerous workflows
3 / 10	Signed-Commits	used in some cases
0 / 10	Signed-Packages	false

Fig. 8: rest-client - C4 Metrics

C5. Just like the previous case study, we had already forked the repository from GitHub into GitLab for our analysis. As new updates are released, we can conduct a code review and merge the change.

C6. This control would have identified the package attempting to contact an unknown URL at Pastebin.com to download the malicious script.

2) Evaluation:

Our methods identified on day 0 one weak link. Day 1 and 2 controls provided the most protection in this case study.

day 0. While performing controls 1-4, we identified that the package had not been maintained or modified in over 90 days. This is a weak link, but due to the nature of the component and that REST calls do not often change, the perceived risk could be less.

day 1. Executing C5 and bringing the package under internal control saved the organization. The organization could conduct a code review of the malicious code before incorporating the changes. If the organization had overlooked the malicious code, merged, and moved to day 2 operations, the additional controls would have caught it.

day 2. C6 would have identified the call to Pastebin.com and blocked the outgoing request.

Overall the day 1 and day 2 controls would have provided the most protection in this use case.

V. Discussion

According to GitHub, the most popular programming languages today are JavaScript, Python, Java, and Ruby. These languages make it easier to re-use code through package managers [27]. In 2005, David Heinemeier Hansson gave a demo via screencast in which he showed developers how to make a blog in 15 minutes using Ruby on Rails [28]. The demo was impressive, and it got developers excited about using Ruby on Rails. However, the demo relied on a few dependencies and one package manager. Fast forward 15 years and today, a simple ”hello-world” Ruby on Rails application require 966 packages across three separate package managers: Ruby Gems (the default package manager for Ruby), NPM (the default package manager for JavaScript), and Yarn (an additional JavaScript package manager) [29]. Another example of dependency growth is with one of the leading JavaScript frameworks for front-end development, React. Facebook developed and maintained React, which pulls in 1,213 packages. This number does not include the number of dependent packages the dependencies depend on [30]. These packages allow software engineers to start developing faster, enabling the software to be shipped faster, bringing value more quickly to the customer, but at what cost?

1,000+ software packages are a lot of packages to vet before using open source frameworks. Our life-cycle controls excel at vetting single components. Still, it would probably not be the best for an entire framework because of the cost associated with recursively checking all of the dependencies and those dependencies, dependencies. As we noticed in the case studies, the controls on day 1 and day 2 provided the most protection, and these are also the most costly controls.

Organizations have for a long time struggled with the build-or-buy decision [31]. Build the software the organization needs or buy the software from a third-party vendor. That thought process needs to be applied to open source components now because the components might be free, but they are free as in puppy [32]. So it is no longer just a question of the cost to maintain the component. It is also the cost of a breach because of the component. It used to be a patch on Friday to prevent a breach on Monday. It was patched on Friday and breached on Monday because of the prevalence of supply chain attacks on open source components.

The perimeter defense from C6 might act as a catch-all in our table-top exercises and provide protection if an organization cannot spend time checking all of the dependencies. Our controls make a couple of assumptions, with the most significant assumption that the controls are correctly configured, especially for C6. Unfortunately, security misconfigurations happen [33] which makes defense-in-depth so important.

VI. Threats to Validity

Due to the vastness of open source components, we cannot monitor all packages for vulnerabilities or compromise. We used two of the most recent and highly publicized breaches for our use cases. We did not have scores from the Scorecards for C3 or C4 metrics that come from a snapshot of the communities at a given time. We used those metrics as they are today. We do not believe this has jeopardized our results because we are looking at the community as it is today, having gone through a breach. The community is probably more prepared and has more substantial community scores today than it did leading up to the breach.

We relied on third-party analysis of the breaches in our evaluations. We used the analysis from reputable sources such as GitHub, MITRE, and security vendors. We do not believe our analysis would have been any better than the analysis we found during our research.

VII. Future Work and Conclusion

In the future, we would like to scale the controls so that they can better evaluate components and frameworks with many dependencies. We would also like to implement the controls in an organizational setting with other developers to see how it fits in to their workflows and to gauge the efficiency.

In this work, we present a method to manage the life-cycle of open source components that leverages six controls capable of protecting organizations that incorporate open-source components. Our controls are repeatable and capable of identifying risk during day 0, day 1, and day 2 of the open-source component's operational life. In identifying these six controls, we can help organizations avoid risky components and mitigate the fallout from malicious supply chain attacks as we demonstrated in our case studies.

References

[1] S. J. Vaughan-Nichols, "Github: All open-source developers anywhere are welcome," Oct 2019. [Online]. Available: https://www.zdnet.com/article/github-all-open-source-developers-anywhere-are-welcome/

[2] Franklin, "How hackers infiltrate open source projects." [Online]. Available: https://www.darkreading.com/application-security/how-hackers-infiltrate-open-source-projects-/d/d-id/1335072

[3] N. e. a. Zahan, "What are weak links in the npm supply chain?" *arXiv:2112.10165 [cs]*, Feb 2022, arXiv: 2112.10165. [Online]. Available: http://arxiv.org/abs/2112.10165

[4] M. Souppaya, K. Scarfone, and D. Dodson, "Draft nist special publication 800-218 - secure software development 2 3 framework (ssdf) version 1.1: Recommendations for mitigating the risk of software 5 vulnerabilities," Sep 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218-draft.pdf

[5] Mircosoft, "Github advisory database." [Online]. Available: https://github.com/advisories

[6] Sonatype, "State of the 2020 software supply chain - the 6th annual report on global open source software development." [Online]. Available: https://tinyurl.com/4dxtxj3z

[7] K. Lewandowski, "Security scorecards for open source projects," Nov 2020. [Online]. Available: https://openssf.org/blog/2020/11/06/security-scorecards-for-open-source-projects/

[8] "Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months," Jul 2020. [Online]. Available: https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream

[9] MITRE, "cve-website." [Online]. Available: https://www.cve.org/ProgramOrganization/CNAsCNAProgramGrowth

[10] M. Hanley, "Github's commitment to npm ecosystem security," Nov 2021. [Online]. Available: https://github.blog/2021-11-15-githubs-commitment-to-npm-ecosystem-security/

[11] C. Cimpanu, "Hacking 20 high-profile dev accounts could compromise half of the npm ecosystem." [Online]. Available: https://www.zdnet.com/article/hacking-20-high-profile-dev-accounts-could-compromise-half-of-the-npm-ecosystem/

[12] MITRE, "About the cve program." [Online]. Available: https://www.cve.org/About/Overview

[13] ——, "Cwe - about - cwe overview." [Online]. Available: https://cwe.mitre.org/about/index.html

[14] S. a. Rose, *Zero Trust Architecture*, Aug 2020, no. NIST Special Publication (SP) 800-207. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-207/final

[15] C. C. Editor, "defense-in-depth - glossary — csrc." [Online]. Available: https://csrc.nist.gov/glossary/term/defense_in_depth

[16] Nov 2021. [Online]. Available: https://codilime.com/blog/day-0-day-1-day-2-the-software-lifecycle-in-the-cloud-age/

[17] plmrry, "vendorize," Jun 2019. [Online]. Available: https://www.npmjs.com/package/vendorize

[18] A. Sutcliffe and P. Sawyer, "Requirements elicitation: Towards the unknown unknowns," in *2013 21st IEEE International Requirements Engineering Conference (RE)*, Jul 2013, p. 92–104.

[19] MITRE, "Secure code review," Aug 2013. [Online]. Available: https://www.mitre.org/publications/systems-engineering-guide/enterprise-engineering/systems-engineering-for-mission-assurance/secure-code-review

[20] RubyGems, "Build software better, together," May 2022. [Online]. Available: https://github.com/rubygems/rubygems.org/security/advisories/GHSA-hccv-rwq6-vh79

[21] F. Salamn, “ua-parser-js.” [Online]. Available: https://www.npmjs.com/package/ua-parser-js

[22] C. Cimpanu, “Malware found in npm package with millions of weekly downloads,” Oct 2021. [Online]. Available: https://therecord.media/malware-found-in-npm-package-with-millions-of-weekly-downloads/

[23] F. Salman, “Merge pull request 528 from jparismorgan/oculus · faisalman/ua-parser-js@8fe448f,” Oct 2021. [Online]. Available: https://github.com/faisalman/ua-parser-js/commit/8fe448fddfe1b63cb0611b9ec79e69cab5c4442e

[24] Rest-Client, “Rest-client/rest-client: Simple http and rest client for ruby, inspired by microframework syntax for specifying actions.” Jun 2019. [Online]. Available: https://github.com/rest-client/rest-client

[25] J. Koljonen, “[cve-2019-15224] version 1.6.13 published with malicious backdoor. · issue 713 · rest-client/rest-client,” Aug 2019. [Online]. Available: https://github.com/rest-client/rest-client/issues/713

[26] A. Brody, “Update rubocop config for rubocop 0.54. · rest-client/rest-client@d177784,” Mar 2018. [Online]. Available: https://github.com/rest-client/rest-client/commit/d1777841a9b16a5099d848d0d2ed62ef0470c0c0

[27] GitHub. [Online]. Available: https://octoverse.github.com/

[28] D. Heinemeier Hansson, “Ruby on rails demo.” [Online]. Available: https://www.youtube.com/watch?v=Gzj723LkRJY

[29] T. Hastings, “tghastings/freshror.” [Online]. Available: https://github.com/tghastings/freshRoR

[30] ——, “tghastings/freshreactapp.” [Online]. Available: https://github.com/tghastings/freshReactApp

[31] V. Cortellessa, F. Marinelli, and P. Potena, “An optimization framework for “build-or-buy” decisions in software architecture,” *Computers Operations Research*, vol. 35, no. 10, p. 3090–3106, Oct 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054807000238

[32] B. Cotton, “Free as in puppy: The hidden costs of free software — opensource.com,” Feb 2017. [Online]. Available: https://opensource.com/article/17/2/hidden-costs-free-software

[33] “Security misconfigurations and how to prevent them,” vol. 2021.