Josh Hatfield

INFO-I 430 (Lecture Section 8835)

Professor Kapadia

4 December 2022

Open-Source Dependencies: RCE Exploitation, Detection, and Mitigation

Section I: Introduction

Although large enterprises utilizing private registries for proprietary code distribution can mitigate many external threats outside of dependency confusion, the majority of open-source proliferation attacks, notably through instances like Log4j or environments like Node.js, have occurred due to applications' increasing dependency on packages and libraries so that they can incorporate functionalities without the need to replicate this source code. It should be noted that libraries contain a collection of packages, which entail a collection of modules, and because most of these dependencies require replicability in addition to scalability for integration, threat attackers through reconnaissance methods can identify a potentially larger attack surface to exploit dependencies to orchestrate, for example, remote code execution (RCE). Due to the evolving combination of coding languages, server functionalities, and dependence of open-source code for multi-industry commerce, this will remain an interesting issue since vulnerability scanners and active mitigation tools will likewise need to evolve to prevent escalation of these expanding threats.

It is an equally important requisite to observe this issue through RCE and determine existing solutions pertaining to dependency vulnerabilities since RCE tends to be the most common attack mechanism in place for exploiting these vulnerabilities; by supplementing more precise vulnerability scanners with current mitigation methods, developers can, in a sense, standardize coding practices across various industries since softwares have become an evermore ubiquitous component with multilateral entities facilitating important clientele, financial transactions, Software as a Service (SaaS) models, etc. with these integrated dependencies. As

progressively more entities managing millions to billions of dollars navigate and implement open-source dependencies to streamline their applications, it is imperative that these industries can remain more equipped for minimizing threats by collectively scanning for vulnerabilities and mitigating them before they become exploitable across these entire industries or even the Internet itself.

Section II: Approach

In the subsequent section, which accentuates the existing techniques employed for scanning open-source dependency vulnerabilities in addition to mitigation methods, the first facet better defines RCE in terms of library dependencies and demonstrates one viable vulnerability detection method utilizing static taint analysis, where I expound upon Shcherbakov et al.'s (2022) methodology using Node.js in relation to prototype pollution and injection sinks. The second facet thereafter highlights hybrid program analysis using machine learning as another less common yet growing vulnerability prediction model based on exposed frontend systems such as web applications. Lastly, for Section III, the latter facet addresses tangible mitigation strategies through, for example, code sanitization and potential string mutability on frontends where libraries' interaction with the applications require user inputs.

In Section IV, I finally synthesize the existing techniques from Section III and any other (perhaps, less refined) findings to provide suggestions for future work that can improve upon the current situation. These suggestions may either consider new avenues for vulnerability scanning and mitigation strategies that have not been researched enough or coalesce with the existing techniques like machine learning to make them more effective at preventing RCE in library dependencies.

Section III: Survey of Existing Techniques / Solutions

RCE Definition and Taint Analysis

While RCE is often conceptualized as a security vulnerability that attackers exploit for running malicious code on a victim's host machine, RCE can be pervasively applied and

executed across many devices depending on how this code interacts with a server or web application maintaining the library dependencies, for instance. Moreover, RCE encapsulates various open-source attack vectors relating to vulnerabilities such as server access and payload deployment, deserialization from stored database information or other configuration files, and most commonly, frontend user inputs that execute with the runtime environment containing these library dependencies (“What Is”). Therefore, existing techniques for vulnerability scanners and mitigation methods attempt to address the open-source environments containing the library dependencies, such as Node.js, since they are utilized more ubiquitously across web applications/servers and therefore do not need to address the attack vectors as independent vulnerabilities.

Node.js acts as an open-source JavaScript runtime environment and library containing other systems such as NPM, which they themselves can granularly host millions of packages and has become popular for scalable network applications on the backend of the clients’ browsers; in turn, Shcherbakov et al. (2022) exacerbate a growing yet challenging technique known as static taint analysis (through Node.js, due to its popularity), where they manually set the conditions of their taint algorithm to determine injection sinks whenever prototype pollution is possible based on the parameters attackers pass as default arguments into the library functions, thereby changing the root prototype for all objects pointing to the same prototype. In essence, as noted by Shcherbakov et al. (2022), prototype pollution occurs whenever the attacker-controlled input modifies the code to assign a new root prototype based on object orientation and define any value to a property relating to the root prototype, including malicious RCE that can deploy across a server whenever the client makes a request and receives a payload based on the updated code. As a result, although arguments do not guarantee the exploitation of injection sinks, Shcherbakov et al.’s (2022) taint analysis determines how many entry points reach the injection sinks using a call graph, and more notably, whether they constitute inputs for the attack sinks where execution on runtime could systematically propagate

the RCE onto either the host server of the application or its clients depending on the taint analysis's call flow from the entry point to the attack sink. After the call graph is generated, Shcherbakov et al. (2022) continue taint analysis by combining dynamic and static analysis to detect potential gadgets for common code execution points and finally conduct manual end-to-end exploitation to determine validity of the true positive rate generated from the initial algorithm.

Even though Shcherbakov et al. (2022) research a more granular flow with taint analysis in relation to Node.js, which is one of many open-source libraries containing dependencies, they demonstrate that RCE based on any code dependency is subject to very systemic patterns due to prototype pollution and near-universal gadgets that exist across some of the most popular open-source applications. Therefore, taint analysis is commonly employed for mitigating RCE since it identifies the entry points and injection sinks for potential vulnerabilities, ensures that open-source libraries can be uniquely analyzed for prototype pollution so that they remain secure as dependent libraries within software stacks, and offers a more universal solution for vulnerability predictors given the systemic patterns identified across wide-ranging applications dependent on these open-source libraries. Most importantly, taint analysis can be applied multilaterally across the application as well as any API calls since, as described in the aforementioned sections, it the very nature of library dependencies to minimize the amount of subsidiary code and therefore allow the cybersecurity community to detect and mitigate vulnerabilities from the source rather than each application independently.

Machine Learning Analysis

In comparison to more common taint analysis methods, such as prototype pollution predictors and manual reviews of the call flows, RCE can still be difficult to predict when obfuscated through encoding with user inputs and has begun to incentivize more recent research on machine learning for determining other vulnerability predictors. Moreover, library dependencies generate complex flows between multiple flows from multiple entry points, which

can be delimited through machine learning models so that the RCE vulnerabilities are predicted immediately at the injection sink sites given these encoding methods as examined by Shar et al. (2015). However, because machine learning is in relative infancy regarding code analysis, many of the existing techniques may still retain lower precision yet high recall rates, but the following examples will ultimately illustrate how machine learning provides a strong foundation for conducting vulnerability predictors with RCE and can improve in the subsequent section for suggestions.

Within their journal article, Shar et al. (2015) iterate upon their previous *PhpMiner* tool to perform backward slices where sinks exist within applications and enumerate a set of potential vulnerability attributes for user inputs on each frontend to determine whether a vulnerability exists based on the type of vulnerability (Structured Query Language injection, Cross Site Scripting, RCE, etc.). Given the extensive vulnerabilities examined due to the vulnerability attributes, however, the precision for RCE remained lower than the other vulnerabilities yet caught fairly consistent trends across the two test subjects analyzed. Such attributes, for example, included “Propagate,” “String-delimiter,” “DotDotDash,” “Encode,” “Session,” and “Un-taint,” all of which represent attributes pervasive with RCE that may still require manual review in infancy but can be trained over time to improve the efficacy of machine learning models specific to RCE across the open-source libraries provided (Shar et al., 2015).

Differentiating slightly from Shar et al. (2015), Chen, Xu, Guan, Zhang, and Fu (2022) limit their machine learning model to command and Structured Query Language (SQL) injections for zero-day exploits, where they compare with intrusion prevention system signatures and indicate based on real instances that the machine learning model can detect new exploits with high confidence that the IPS signatures themselves had overlooked. Furthermore, according to their studies, “From ~1 million benign and ~2.2 million malicious samples containing web searches and possible command injections, [the] command injection model achieve[d] a 0.011% false positive rate and a 92% true positive rate”; this demonstrates that

machine learning tools with large neural chains and training sets can achieve the level of efficacy for corresponding injection sinks to the inputs themselves since all code and inputs are subject to systemic patterns that attempt to exploit vulnerabilities through RCE (Chen et al., 2022).

Overall, while methods for determining prototype pollution can act as vulnerability predictors for detecting one form of RCE in code, RCE can also be executed and obfuscated through other less “controlled” channels such as configuration files or the frontend user inputs where encoding can bypass callback functions that sanitize any string values. Thus, machine learning offers a mechanism for identifying vulnerabilities with new libraries and packages existent within large open-source source code such as Node.js, and if the efficacy through precision and recall remain high given the trained input sets, machine learning can remain a more viable alternative to manual review or algorithms through taint analysis when attempting to coordinate the complex combination of relationships across the various dependencies differentiating with each software stack. To reiterate, since code follows specific patterns and syntax such as object-oriented programming, loops, and information flows, RCE itself requires its own construction of exploitable syntax with user inputs on command lines or frontend applications, and we can see a few of many machine learning tools progressively being employed to determine these patterns.

String Mutability and Sanitization

On frontends, where user inputs are generally transferred to SQL $_POST variables to be stored within databases or computed for other means, Hypertext Processor (PHP) already offers several important functions to prevent code injection within these inputs like *mysqli_real_escape_string* or *htmlspecialchars*. These are the most standard mitigation methods that protect most frontend inputs and will likely remain applicable as existing solutions given the intended limited nature of user inputs for applications as well as their ubiquity with most existing libraries. However, encoded RCE may occasionally bypass these functions with

both PHP and other programming languages by altering, for example, spaces to special characters that are standard with any ASCII encoding or deserialization process given the bytestream generated from the input (“Unicode Encoding”). As a result, research–and more specifically Zürcher’s (2022) research, to highlight a particular example–has attempted to consider string mutability with programming languages outside PHP and therefore a larger collection of library dependencies, where strings are analyzed before initialization of the runtime to prevent RCE.

Zürcher (2022) focuses on OpenJDK VM for built-in Java classes, where the *BString* tool implements standard sanitization prior to initialization for preventing code injection, yet it can more importantly identify string derivations and set conditions depending on how the operations are realized by the tool. Lastly, the tool augments a value history tree for each string variable so that any derivations or even concatenations with new string variables are connected as child and parent nodes, similar to how Git repositories work when comparing version histories of the same files within the repository. As such, Zürcher’s (2022) functionality remains critical for RCE mitigation since it minimizes the need for taint analysis across multiple libraries, separates security sanitization from the remaining application code on the software stack, and provides real-time version updates whenever string variables are derived when they should not be, even if an attacker can bypass the sanitization checks prior to runtime initialization.

Similar to Shcherbakov et al. (2022), while Zürcher (2022) provides an unconventional method to RCE detection and prevention due to the lack of standardization for frontend inputs, this research exacerbates one of many ways in which applications are analyzing these frontend inputs, especially in relation to strings, prior to runtime so as to prevent RCE at entry points. Strings offer more opportunity for derivation and need to be limited to reflect the limited nature of user inputs, which existing techniques attempt to accomplish yet still encounter some shortcomings due to potential exploits through encoding in addition to how they interact with each other as archetypal variables.

Section IV: Suggestions

As demonstrated in the previous section, existing techniques have laid a strong foundation for identifying potential vulnerabilities and some methods to mitigate them, yet we can directly see how they require extensive manual intervention to either tweak the conditions to identify the attack sectors or actually prevent the RCE after a vulnerability has been exploited. Moreover, both static and dynamic analysis tend to considerably increase performance overhead, meaning that vulnerability detection may benefit less from taint algorithms that need to encompass all possible attack vectors like intrusion prevention systems. For this reason, I believe that more research needs to be undertaken with two types of machine learning tools as an alternative to manual taint analysis, utilizing a combination of supervised and semi-supervised learning, where one addresses vulnerability detection and prevention on the frontend of applications while another addresses these components on backend library dependencies.

For the first machine learning tool, research would expand derivation to other data types such as character arrays, boolean values, integers, and any combination with the immutable strings previously researched to better understand exploitations with encoding in addition to deserialization once a bytestream is accepted through the user input. Due to the nomenclature with encoding, the machine learning tool would require training data based on previous vulnerable inputs, ASCII characters, and data flow reflecting other application parameters such as session management or database population to test RCE in multiple scenarios. After enough training data has been tested to improve the neural chain of the machine learning tool across programming languages, which would require some backend data to be included in the training, research could then utilize a combination of predetermined inputs with inputs then encoded to further train the model to prevent exploitation of encoding. Due to the prevalence and arbitrary nature of user inputs based only on the input types, this machine learning tool would be

applicable to any frontend with user inputs, regardless of the potential sinks or library dependencies, since the tool would conduct analysis and prevention at the initial entry point.

As for the second machine learning tool, this would work similarly to the first tool yet would require more oversight to determine potential prototype pollution on the backend across multiple programming languages where the library dependencies interact. Therefore, the tool would need some neural chains to determine possible injection sinks, which would use semi-supervised learning so that the tool can potentially create labels based on the object to determine whether any changes occur between the entry point and the injection sink as with Zürcher's (2022) string derivations. Finally, rather than testing with a single open-source library like Node.js, further training would be accomplished through existing applications, given their unique software stacks between the library dependencies and proprietary code, to increase the tool's ubiquity beyond one programming language. Overall, training could take years until both machine learning tools are efficacious enough to prevent zero-day exploits, yet any machine learning system can learn over time and may become more effective than predetermined algorithms requiring manual intervention.

Section V: Conclusions

Overall, although the presented findings exhibit many technicalities, they generally intend to prevent RCE since attackers can destabilize entire systems' and servers' infrastructures through malicious commands that only need to exploit a single library dependency. Within Section III, Shcherbakov et al. (2022) expand upon taint analysis as one form of vulnerability prediction, where attackers can pollute prototypes within libraries to then propagate the RCE on an attack sink, thereby affecting entire applications or servers. Next, both Shar et al. (2015) and Chen et al. (2022) highlight machine learning tools for vulnerability prediction, which offer other alternatives compared to taint analysis using algorithms. Lastly, Zürcher (2022) focuses on the nature of frontend inputs and string mutability to prevent potential RCE prior to runtime initialization. While these solutions pertaining to research literature

encapsulate very specific examples, taint analysis, machine learning, and frontend input checks have all had considerable research due to the importance of preventing RCE, which we have seen with paramount examples like the Log4j vulnerability.

Even though RCE has become a larger priority to mitigate given how applications have become large revenue channels for entities across many industries such as finance, logistics, and obviously commerce, library dependencies seem to be overlooked since we intuitively assume that open-source libraries integrated within these applications are resistant to vulnerabilities. Likewise, attackers can utilize many attack vectors on both the frontend and backend to exploit these library dependencies, and because existing techniques do not view these vectors more holistically, they do not provide solutions that can be applied more accessibly to applications maintaining complex interactions across these dependencies. Thus, by combining taint analysis or vulnerability prevention with machine learning models that can be trained over time rather than manually tweaked, models can expand their proficiency with analyzing new libraries and countering zero-day exploits as they occur. Conclusively, this can help standardize coding to better emphasize security practices in relation to RCE as well as assure that libraries can be shared without the risk of vulnerabilities in the future.

References

Chen, J., Fu, Y., Guan, A., Xu, L., & Zhang, Z. (2022, September 16). *Zero-Day Exploit Detection Using Machine Learning*. Unit 42. Retrieved December 4, 2022, from https://unit42.paloaltonetworks.com/injection-detection-machine-learning/

Shar, L. K., Briand, L. C., & Tan, H. B. (2015). Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning. *IEEE Transactions on Dependable and Secure Computing*, *12*(6), 688–707. https://doi.org/10.1109/tdsc.2014.2373377

Shcherbakov, M., Balliu, M., & Staicu, C.-A. (2022). Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js, 1–23. https://doi.org/10.48550/arXiv.2207.11171

Unicode Encoding. OWASP Foundation. (n.d.). Retrieved December 4, 2022, from https://owasp.org/www-community/attacks/Unicode_Encoding

What Is RCE Vulnerability? Remote Code Execution Meaning. Wallarm. (n.d.). Retrieved December 3, 2022, from https://www.wallarm.com/what/the-concept-of-rce-remote-code-execution-attack

Zürcher, C. (2022). *BString: A String-based Framework to Improve Application Security* [Master's thesis, University of Bern]. Software Composition Group Institute for Computer Science, Bern. Retrieved October 1, 2022, from https://scg.unibe.ch/archive/masters/Zuer22a-BString.pdf.