Vision for a Secure Elixir Ecosystem: An Empirical Study of Vulnerabilities in Elixir Programs

Dibyendu Brinto Bose
Reve System
Dhaka, Bangladesh

Kaitlyn Cottrell
Tennessee Tech University
USA

Akond Rahman
Tennessee Tech University
USA

ABSTRACT

Since its inception in 2011, Elixir has emerged as a popular programming language. Currently, Elixir is used in a diverse set of domains, such as instant messaging, smart farming, and e-commerce. Usage of Elixir in above-mentioned domains necessitates gaining an understanding of the state of vulnerabilities that are reported for Elixir programs. An empirical analysis of vulnerability-related commits, i.e., commits that indicate action taken to mitigate vulnerabilities, can help us understand how frequently vulnerabilities appear in Elixir programs. Such understanding can also be a starting point to integrate secure software development practices into the Elixir ecosystem. We conduct an empirical study where we mine 4,446 commits from 25 open source Elixir repositories from GitHub. Our findings show that (i) 2.0% of the 4,446 commits are vulnerability-related, (ii) 18.0% of the 1,769 Elixir programs in our dataset are modified in vulnerability-related commits, and (iii) the proportion of vulnerability-related commits is highest in 2020. Despite Elixir being perceived as a 'safe' language, our empirical study shows programs written in Elixir to contain vulnerabilities. Based on our findings, we recommend researchers to investigate the root causes of introducing vulnerabilities in Elixir programs.

CCS CONCEPTS

• **Software and its engineering** → *Frameworks*; • **Security and privacy** → *Software security engineering*.

KEYWORDS

dataset, empirical study, elixir, vulnerability

ACM Reference Format:
Dibyendu Brinto Bose, Kaitlyn Cottrell, and Akond Rahman. 2022. *Vision for a Secure Elixir Ecosystem*: An Empirical Study of Vulnerabilities in Elixir Programs. In *2022 ACM Southeast Conference (ACMSE 2022), April 18–20, 2022, Oxford, AL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3409334.3YYYYYY

1 INTRODUCTION

Elixir is a programming language that provides fault tolerance without sacrificing the simplicity of scripting languages, such as Ruby. Since its inception in 2011, Elixir has gained in popularity in recent years. Information technology (IT) organizations, such as Discord, Pinterest, and WeChat have reported benefits in using Elixir. For example, Discord used Elixir to build their service to support 5 million concurrent users [13]. As another example, Pinterest reported Elixir to help them *"speed up the performance of their notification system delivering 14,000 notifications per second, and cut down the number of servers by half—from 30 to just 15—compared to when the service relied on Java"* [13].

Practitioners perceive Elixir to be a 'safe' language, as the language allows practitioners to write fast software programs without introducing vulnerabilities, unlike other languages, such as C [16]. Positive perception of practitioners about the safety and security Elixir programs is subject to empirical validation. Practitioner perceptions are formed through personal experience, and not based on empirical evidence [9]. A systematic empirical investigation that quantifies reported security vulnerabilities can shed light on the state of security of Elixir programs. Such empirical investigation can also yield recommendations for practitioners and researchers on how to securely develop Elixir programs. As Elixir's use becomes more and more prevalent in large-scale systems with millions of users, such as for WeChat with 300 million daily active users [10], insecure Elixir programs with latent vulnerabilities can result in serious consequences.

We answer the following research question in our empirical study: ***RQ*: How Frequent are Vulnerabilities Reported for Elixir Programs?**

We conduct an empirical study where we mine 4,446 commits from 25 OSS Elixir repositories from GitHub to quantify vulnerability-related commits, i.e., commits that indicate action taken to mitigate vulnerabilities. We apply a qualitative analysis technique called closed coding [18], where we use raters to inspect commit messages in order to determine if commits are reflective of taking an action related to vulnerability mitigation.

Contribution: We list our contribution as follows:

- An empirical analysis of how frequently vulnerabilities are reported for Elixir programs; and
- A curated dataset where vulnerability-related commits are identified.

2 BACKGROUND AND RELATED WORK

We provide background and related work related to Elixir in this section.

2.1 Background

Released in 2011, Elixir is meant to incorporate all of Erlang's powerful tools for parallel and concurrent computation while having Ruby-like syntax. Erlang is a programming language founded on

ACMSE 2022, April 18–20, 2022, Oxford, AL, USA

ACM ISBN 978-1-4503-8697-5/22/04…$15.00
https://doi.org/10.1145/3409334.3YYYYYY

three primary principles: concurrency, fault tolerance, and distribution. These principles were defined based on the original use case of telecommunications. As time went on, Erlang remained relevant, but still had problems surrounding syntax and lack of modern tooling. This was addressed by creating Elixir. All of Elixir's code ends up as byte code which is used by the Erlang virtual machine called BEAM [15].

Since Elixir is built on top of BEAM, it shares the same abstractions as Erlang that makes it well suited for concurrent applications. Elixir makes use of lightweight, isolated processes that run across all CPUs, and immutable data to make concurrency easier. These same isolated processes also allow for simple scalability, both in adding more machines, and in using a current machine's resources more efficiently. The processes also help to acheive reliability because they can be restarted quickly by the supervisory system if a problem occurs. Apart from BEAM, Elixir also inherits the Open Telecom Platform (OTP) from Erlang, which is a standard resource in Erlang's library for use in fault-tolerance for telecom systems, though is now useful for any concurrent applications [11].

We provide an example Elixir program in Listing 1. The `File.open()` function in the program takes two parameters: the path of the file, and the mode to be opened in. The program creates a new file called 'HelloFile', and opens that file for writing data in it. The `File.open` function returns a tuple where the first element of the tuple represents status (`:ok`), and the second element of the tuple is the process ID of the process that will handle the file. Next, the `IO.binwrite()` function is used to write to the file opened with `File.Open()`. Next, the `File.close` function closes the file. Finally, with the `File.read()` function 'HelloFile' is read. The `File.read()` returns a tuple where the first element of the tuple represents status (`:ok`), and the second element of the tuple is the file content, i.e., 'world'.

```
{:ok, file} = File.open("HelloFile", [:write])
IO.binwrite(file, "world")
File.close(file)
File.read("hello")
```

Listing 1: An example Elixir program to demonstrate file read write operations.

2.2 Related Work

Our paper is closely related to existing research that has investigated Elixir programs and their usage. Avila et al. [3] explores how Elixir may potentially help to alleviate the issues that can come with coding for embedded devices. Fedrecheski et al. [12] investigates how useful Elixir could be in creating IoT software programs. Chhabra et al. [7] presents a machine learning tool set, Tensorflex, which allows Elixir programmers to utilize Tensorflow, a machine learning framework to create machine learning applications. Cassola et al. [6] proposes a type system that makes it possible to perform static type-checking for Elixir programs.

The above-mentioned discussion shows a lack of research related to vulnerabilities in Elixir programs, which we address in our paper.

3 METHODOLOGY

We provide the methodology to answer: ***RQ*: How Frequent are Vulnerabilities Reported for Elixir Programs?** in this section. Our methodology involves two steps: *first*, mining of OSS Elixir repositories, and *second*, use of closed coding to identify vulnerability-related commits that are related to vulnerabilities. An overview of our methodology is presented in Figure 1.

3.1 Elixir Repository Mining

We use OSS Elixir repositories hosted on GitHub to conduct our empirical study. *First*, we use GitHub's search utility to filter repositories whose main programming language is Elixir. This step yielded 31,624 repositories. Next, we use the count of stars to identify the top 25 repositories that are popular amongst the open source Elixir community. We use star count as stars are reflective of popularity for repositories hosted on GitHub [4]. From the 25 Elixir repositories, we identify 4,446 commits that are used to modify Elixir programs. Attributes of the used repositories are available in Table 1.

Table 1: Repository Attributes

Attribute	Count
Repositories	25
Elixir Files	1,769
Elixir-related Commits	4,446
Duration	03/2012-12/2020

3.2 Qualitative analysis of commits

We use commits to quantify vulnerabilities as commits express actions related to software development [1]. We use all 4,446 commits collected from Section 3.1. We apply a qualitative analysis technique called closed coding, where a rater inspects text excerpts and maps the set of text excerpts to a pre-defined category [18]. We apply closed coding with two raters who are well-versed in software security to conduct the closed coding process. The first rater is the first author of the paper with two years of academic experience in cybersecurity, and one year of professional experience in software engineering. The second rater is not an author of the paper and participated voluntarily. The second rater is a graduate student in the department with one year of professional experience in cybersecurity. Both raters individually determine if each of the collected commits to be security-related by performing the following activities: *Activity-1*: The rater observes if any of the following keywords appear in each of the collected commit messages: 'race', 'racy', 'buffer', 'overflow', 'stack', 'integer', 'signedness', 'widthness', 'underflow', 'improper', 'unauthenticated', 'gain access', 'permission', 'cross site', 'css', 'xss', 'htmlspecialchar', 'denial service', 'dos', 'crash', 'deadlock', 'sql', 'sqli', 'injection', 'format', 'string', 'printf', 'scanf', 'request forgery', 'csrf', 'xsrf', 'forged', 'security', 'vulnerability', 'vulnerable', 'hole', 'exploit', 'attack', 'bypass', 'backdoor', 'threat', 'expose', 'breach', 'violate', 'fatal', 'blacklist', 'overrun', and 'insecure'. We collect these keywords from prior work [5]. *Activity-2*: The rater determines a commit to be a security-related defect if the message indicates that an action was taken to address a security concern for the software of interest. The rater determines a commit message to be related to a vulnerability if any of the following security objects are violated: confidentiality, integrity, or availability. We apply this step because only relying on keyword search could

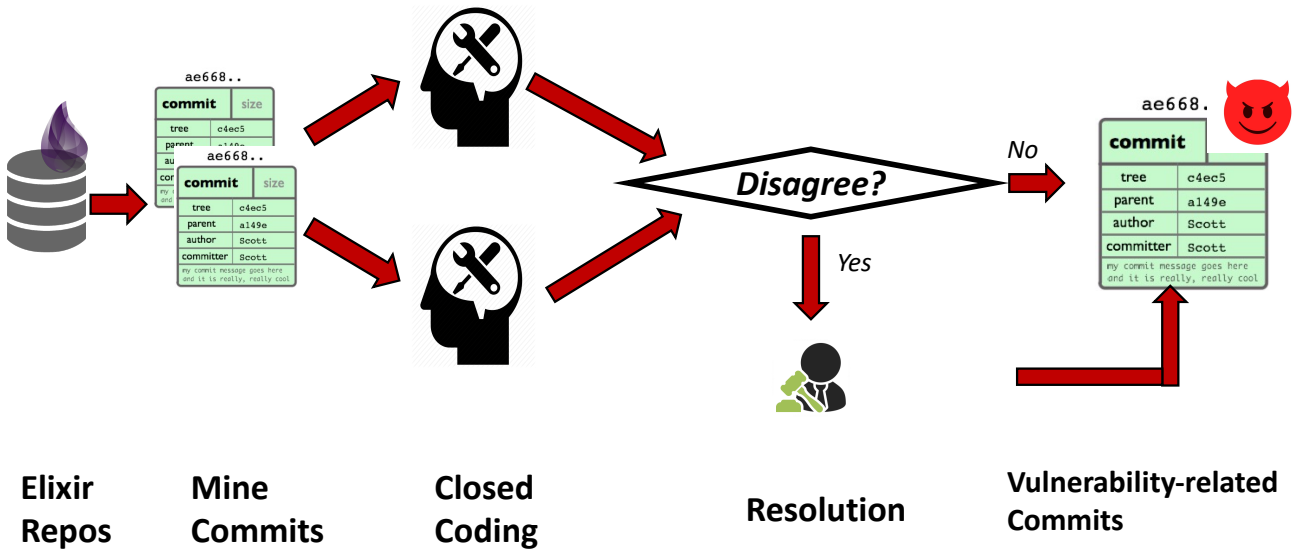

Figure 1: An overview of our methodology.

generate false positives. *Activity-3*: We calculate rater agreement using Cohen's Kappa [8].

After completing the above-mentioned activities, we obtain a dataset with each commit to be vulnerability-related or not. If a commit is related to a vulnerability, it is labeled as 'VULNERABLE', otherwise the commit is labeled as 'NEUTRAL'. Our research question is answered by reporting the count and proportion of commits that are labeled 'VULNERABLE'. We also report the proportion of commits that are labeled as 'VULNERABLE' for each year using Equation 1.

$$\text{PER\_YEAR\_PROPORTION}(y) = \frac{\text{\# of commits in month } y \text{ mapped as 'VULNERABLE'}}{\text{count of commits in year } y} \quad (1)$$

3.3 Limitations

Our methodology is susceptible to the following limitations:

- The derived dataset is restricted by rater bias, which we mitigate using two raters.
- Our dataset is collected from the OSS GitHub repositories, which is limiting and may not generalize to proprietary datasets.
- Our identification process of vulnerabilities using keywords can generate false positives, which we mitigate through manual inspection by two raters.

4 RESULTS

In this section, we provide results with relevant discussion.

4.1 Results

The closed coding process took 121 and 186 hours respectively, for the first and second authors. The Cohen's Kappa between the first and second author is 0.88, which is 'almost perfect', according to Landis and Koch [14]. Upon completion of the closed coding process, we identify disagreements for 101 commit messages. The last author of the paper with 10 years of experience in software engineering acted as a resolver. The last author's decision is final while resolving disagreements.

Answer to RQ: How Frequent are Vulnerabilities Reported for Elixir Programs? Upon resolving the disagreements we altogether identify 90 commits to be labeled as 'VULNERABLE', whereas 4,356 commits are labeled as 'NEUTRAL'. The proportion of vulnerability-related commits is 2.0%. We observe 319 Elixir programs to be modified in the 90 vulnerability-related commits. We also observe 9 of the 25 Elixir OSS repositories to include one vulnerability-related commit. Attributes of our vulnerability dataset are available in Table 2. The labeled dataset is available online as a CSV file [2], which can be imported using existing APIs, such as Pandas [17] and R [19].

Table 2: Attributes of Vulnerability-related Commits and Elixir Programs

Attribute	Count
Vulnerability-related Commits	90
Elixir Programs with >= 1 Vulnerability-related Commits	319
Repositories with >= 1 Vulnerability-related Commits	9

We report the 'PER_YEAR_PROPORTION' values in Table 3. We observe the highest PER_YEAR_PROPORTION for 2020, whereas the second highest PER_YEAR_PROPORTION is observed for 2019. For 2015, we observe the third highest PER_YEAR_PROPORTION values. One possible explanation is that the observed trends are limited to the mined 25 repositories, i.e., how vulnerability-related commits appear in these repositories determines the trends. Our findings reported in Table 3 provides groundwork for further analysis related to vulnerability-related trends for OSS Elixir programs.

Table 3: PER_YEAR_PROPORTION Values. We observe an increase in PER_YEAR_PROPORTION after 2019.

Type	PER_YEAR_PROPORTION (%)
2012	0.94
2013	0.48
2014	1.74
2015	3.38
2016	1.92
2017	1.22
2018	1.05
2019	3.45
2020	3.85

4.2 Discussion

We discuss our findings in the following subsections:

4.2.1 A Disaster Waiting to Happen?: Implications for Researchers. Table 2 shows that 18.0% of Elixir programs to be modified in at least one vulnerability-related commit. As commit messages contain developer-reported observations and actions [1], we conjecture that our identified vulnerability-related commits can be under-reported and there might exist latent vulnerabilities that developers are not aware of. Our conjecture is subject to empirical validation, which researchers can investigate. One low-hanging fruit can be conducting empirical studies that quantify vulnerabilities in Elixir programs by applying static analysis tools. Furthermore, researchers can investigate why developers introduce security vulnerabilities. Such investigation can lead to the identification of reasons, such as lack of tools, lack of security awareness, and pressure to develop projects. Based on identified reasons researchers can work together with the Elixir community to derive best practices to mitigate vulnerabilities in Elixir programs. Without derivation and integration of secure software development practices for Elixir, we predict an impending disaster waiting to happen for Elixir-based projects, as Elixir is used in a wide-range of domains, such as smart farming [1].

4.2.2 Implications for Practitioners. From Table 2 we observe 18.0% of the Elixir programs in our dataset to be modified in a commit that maps to a vulnerability. We advocate for a 'shift left' approach where practitioners integrate secure software development practices so that developers repair vulnerable code changes before code is pushed to production. One possible approach could be using security-focused code review, where developers with security expertise review code changes for potential vulnerabilities. Practitioners can also use Elixir-focused static analysis tools, such as Sobelow [2].

[1] https://elixir-lang.org/blog/2020/08/20/embedded-elixir-at-farmbot/
[2] https://github.com/nccgroup/sobelow

5 CONCLUSION

The widespread use of Elixir in a wide range of domains, such as instant communication, smart farming, and social media necessitates Elixir programs to be free of vulnerabilities so that malicious users cannot exploit developed Elixir programs. An empirical study of vulnerabilities in Elixir programs can provide an understanding of how frequent vulnerabilities are introduced in Elixir programs. By applying closed coding with 4,446 commits mined from 25 Elixir OSS repositories we quantify the frequency of vulnerability-related commits for Elixir programs. We observe 2.0% of the 4,446 commits to be vulnerability-related. Based on our findings, we (i) recommend practitioners to adopt a shift left policy by applying static analysis on Elixir programs, and (ii) recommend researchers to systematically investigate the reasons why vulnerabilities are introduced in Elixir programs. Our paper lays the groundwork for a vision of a Elixir ecosystem, where developers will pro-actively mitigate vulnerabilities with secure software development practices.

ACKNOWLEDGMENTS

We thank the PASER group at Tennessee Tech University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) through Award # 1852126, # 2043324, and # 2026869.

REFERENCES

[1] A. Alali, H. Kagdi, and J. I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *2008 16th IEEE International Conference on Program Comprehension*. 182–191. https://doi.org/10.1109/ICPC.2008.24
[2] Anonymous. 2021. Dataset for Paper. https://figshare.com/s/3d9ef789dcdffbdcd0d8
[3] Humberto Rodriguez Avila, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2017. An Elixir library for programming concurrent and distributed embedded systems. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*. 1–1.
[4] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Predicting the popularity of GitHub repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. 1–10.
[5] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/2635868.2635880
[6] Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. 2020. A gradual type system for elixir. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*. 17–24.
[7] Anshuman Chhabra and José Valim. 2018. Tensorflex: Tensorflow bindings for the Elixir programming language. (2018).
[8] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:http://dx.doi.org/10.1177/001316446002000104
[9] Prem Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief and Evidence in Empirical Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 108–119. https://doi.org/10.1145/2884781.2884812
[10] Gints Dreimanis. 2020. 8 Companies That Use Elixir in Production. https://serokell.io/blog/elixir-companies
[11] Gints Dreimanis. 2020. Serokell Software Development Company. https://serokell.io/blog/introduction-to-elixir
[12] Geovane Fedrecheski, Laisa CP Costa, and Marcelo K Zuffo. 2016. Elixir programming language evaluation for IoT. In *2016 IEEE International Symposium on Consumer Electronics (ISCE)*. IEEE, 105–106.
[13] Karolina Kurcwald. 2020. Eight Famous Companies Using Elixir—And Why They Made the Switch. https://www.monterail.com/blog/famous-companies-using-elixir
[14] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http:

//www.jstor.org/stable/2529310
[15] Wolfgang Loder. 2015. Erlang and Elixir for Imperative Programmers. *Chapter 16: Code Structuring Concepts, section title Actor Model* (2015).
[16] Nathan Long. 2021. Elixir is Safe. https://dockyard.com/blog/2021/03/30/elixir-is-safe
[17] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14, 9 (2011).
[18] Johnny Saldaña. 2015. *The coding manual for qualitative researchers.* Sage.
[19] Sylvia Tippmann. 2015. Programming tools: Adventures with R. *Nature News* 517, 7532 (2015), 109.