Supporting Secure Coding for Erlang

Melinda Tóth
Faculty of Informatics
ELTE, Eötvös Loránd University
Budapest, Hungary
tothmelinda@elte.hu

István Bozó
Faculty of Informatics
ELTE, Eötvös Loránd University
Budapest, Hungary
bozoistvan@elte.hu

ABSTRACT

Software developer/operator companies have to face growing cyber threats. Secure coding is a development process resulting in software products that resist cyber-attacks. Although several security standards and guidelines exist to help developers create secure software, tool-supported identification of vulnerability issues is also needed. Static analyser tools can help to find security issues in the code. However, those work usually in a language-dependent way. In this paper, we show secure coding issues for the programming language Erlang and demonstrate a tool, RefactorErl, that supports the identification of these vulnerabilities in Erlang systems.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; **Vulnerability scanners**; • **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

secure coding, security checkers, Erlang security, static analysis, RefactorErl

ACM Reference Format:
Melinda Tóth and István Bozó. 2024. Supporting Secure Coding for Erlang. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3605098.3636185

1 INTRODUCTION

Nowadays, one of the most important concepts in software development is source code security. Developing secure code that resists cyber threats is not straightforward. Expert developers can spot security issues in the code. However, using manual code review to locate those vulnerabilities is time-consuming and expensive.

Static source code analyser tools can help to find security issues automatically. For mainstream programming languages several tools and services are already developed [5, 6, 13] based on the available standards.

According to the Application Security Verification Standard [9], the most well-known vulnerabilities come from inappropriate input validation, output encoding, authentication and password management, session management and access control, data protection, system configuration, database security, and file or memory management.

The programming language Erlang [14] was designed to build highly available, reliable, massively concurrent, fault-tolerant distributed systems. The original design was formulated to fit the telecommunication system requirements [1]. However, it proved to be useful in different areas, such as finance, gaming, healthcare, IoT, web services, etc.

One of Erlang's unique properties is the "Let it crash" philosophy that allows unhandled errors in the system and lets a process evaluate the function crash at runtime. In the meantime, Erlang gives a unique approach to identifying and handling process-related errors.

Erlang embraces the concept of Shared Nothing, which means that the processes of the distributed network run isolated with their resources. In addition, immutable data structures, pure functions, modularity and fault tolerance as core language concepts also highly contribute to the hardening of the system against security attacks.

Despite the above mentioned, there are security issues which result in the system crash (by halting the Erlang Virtual Machine). Thus, input validation in some cases can not be avoided. There was also a change in the original concept of running Erlang on protected hosts. In addition, the human factor of the development process (e.g. inexperienced programmers, the lack of knowledge about a specific application) makes it necessary to consider secure coding during the development of Erlang applications.

Security issues in Erlang. Previous research concluded security issues and guidelines for writing secure Erlang systems [2, 4, 8]. Based on the concepts of Erlang, we might group those into five categories [2]:

- Interoperability mechanism-related vulnerabilities - starting unvalidated external executables
- Concurrent programming-related issues - process and data handling resulting in inconsistent/faulty behaviour
- Distributed programming-related issues - unsafe network/communication configuration management
- Injections - executing and loading unvalidated
- Memory overload-related attacks - generating new atoms and overloading the atom table.

Contribution. The main result of the paper is providing a tool for the Erlang community to help identify security vulnerabilities in an early phase of the development process and improve the security level of their Erlang applications. We defined checkers to identify the source code fragments that violate the security rules. The checkers were defined using a set of static source code analyses (e.g. data flow, call-graph information) provided by RefactorErl [3].

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636185

2 VULNERABILITY CHECKING IN REFACTORERL

RefactorErl [3] is a static source code analysis and transformation tool for Erlang. It aims to support everyday software development and maintenance tasks, code comprehension, and software checks.

RefactorErl represents the Erlang source code in a Semantic Program Graph (SPG) [10] representing the lexical, syntactical structure of the source code extended with the result of thorough static semantic analyses (scoping, call graph, data-flow information, etc).

2.1 Detecting vulnerabilities

The general vulnerability detection contains four main phases:

(1) Defining the set of vulnerable functions
(2) Finding the references of the vulnerable functions
(3) Determining the set of possible values that may flow to an argument of a harmful reference (origin values)
(4) Filtering false-positives
   - Filter the unknown sources from the origin values
   - Filter the origin values based on the possibility of user interventions (authorised or non-authorised users can define the value)
   - Filter the known (developer-defined safe sources) from the original values

The set of vulnerable function definitions is based on our former research [2, 4]. The tool stores the identifiers (module name, function name and arity) in its vulnerability database. Identifying the references of these functions can be done efficiently based on the call graph of RefactorErl. The call graph contains both static and dynamic call information. The third step, the identification of the original values, is based on the data-flow analysis [16] of RefactorErl. Using the calculated original values, the tool performs extensive false-positive filtering.

The analysis is conservative, so every unknown source is considered harmful. However, a developer can mark sources/functions as safe. The tool drops those references from the result list that depend only on *safe functions*. Then, it checks whether the original value is from an external source (e.g. an exported function), so it can be defined by a user. If the tool finds no external or unknown source for a reference, it will not report it.

Some of the vulnerability checkers contain additional steps. For example, the ets table traversal checker also checks the existence of a lock on the table, which makes the traversal safe. The unsafe communication checkers are examining the given SSL connection options.

2.2 Using the tool

The security checkers are implemented at the top of the RefactorErl extensive analyser framework. Users can access the result of the security checks through the semantic query language [11] of the tool using either the `mods.funs.unsecure_calls` or the optimised `unsecure_calls` queries. It is also possible to filter the result by the type of vulnerability and by the containing context. For example, listing the unsecured ets table traversals from the module `foo` could be done through `mods[name=foo].funs.unstable_call`.

2.2.1 User interface. Figure 1 demonstrates the usage in the command line interface, and Figure 2 shows the interactively browsable web interface results. Selecting an item from the result list on the web interface navigates to the exact location in the source code and highlights the vulnerability.

```
(refactorerl@localhost)22> ri:q("mods.funs.unsecure_calls").
erlang_v8_vm:handle_info/2
    [[Context]] = ets:match(Table, {'$1', MRef})
erlang_v8_vm:start_port/1
    Port = open_port({spawn_executable, Executable}, Opts)
erlang_v8_vm:os_kill/1
    os:cmd(io_lib:format("kill -9 ~p", [OSPid]))
ok
(refactorerl@localhost)23>
```

Figure 1: Querying unsecured calls through the shell interface

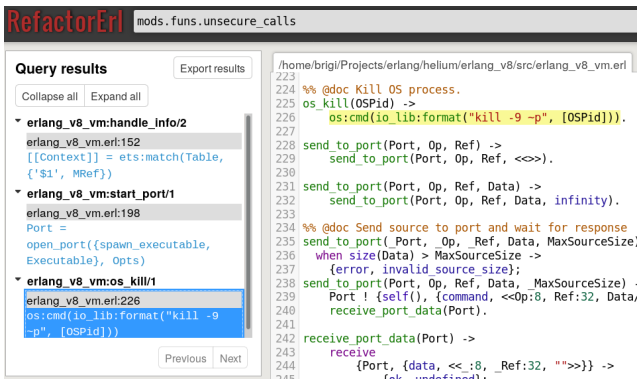

Figure 2: Querying unsecured calls through the web interface

2.2.2 Available Security Checkers. RefactorErl can discover fifteen different vulnerability types. We grouped those into four main categories presented in Table 1. An additional vulnerability group is defined as the *Deprecated* group. Deprecated functions are outdated in the Erlang library and either removed, scheduled to be removed, or just a warning to be removed in an upcoming Erlang release.

2.3 Analysing open-source projects

The vulnerability checker component of the tool was applied to several open-source Erlang projects. We selected some simple projects and others that are heavily used by the community. We are continuously improving the implementation based on the findings. More than 500K Erlang source lines were checked.

The tool found several atom creation-related issues (> 65 2/3). In general, we can say that these are real issues: the tool reported them as vulnerable because the user can provide different input values for them through a call to an exported function. Calling that exported function iteratively more than a million times could crash the Erlang VM. One might say that operators can protect the VM from those calls. Therefore, we assigned *low* severity values to these vulnerabilities. When we find a recursive/iterative execution path in the source code that contains the vulnerable call, we assign *high* severity to the occurrence of the vulnerability.

Table 1: The available security checks in RefactorErl

Injection	Denial-of-Service	Race condition	Man-in-the-Middle	Deprecated
unsafe_os	unsafe_atom	unsafe_prioritisation	unsafe_crypto	deprecated
unsafe_port	unsafe_xml	unsafe_linking	unsafe_network	removed
unsafe_nif	unsafe_file_read	unsafe_ets_calls	unsafe_communication	to_be_removed
unsafe_port_driver				
unsafe_compile_load				
unsafe_file_eval				

A significant amount of injection-like vulnerability was found (< 30%), and a few race conditions and MitM issues. We found no issue related to unsafe linking.

False-positive hits were found due to the missing information about library functions. For example, when a call is marked vulnerable because of the missing information about the function `lists:flatten/2`. Adding built-in knowledge about safe library functions is an ongoing work.

3 RELATED WORK

The early detection of potential vulnerabilities has proved to be important in the software development industry. There are several static analysis tools to help developers identify such issues like SonarQube [5], SpotBugs [13] and CodeChecker [6]. Most of these tools integrate well into popular IDEs and editors such as IntelliJ, Visual Studio Code and Eclipse. They are usually part of a Continuous Integration (CI) pipeline. These do not cover vulnerability checks for Erlang.

Secure coding is becoming a hot topic in the Erlang community. An increasing number of papers and discussions try to raise security awareness [7, 12]. PEST [15] is one of the already available tools to aid developers in identifying security issues. It analyses the source code to detect interoperability-related functions like NIF or port driver creation calls, the usage of OS calls or the decommissioned functions from the `crypto` module. PEST's vulnerability identification is based on syntax tree (abstract code) traversals and does not apply additional filtering on the found candidates. Thus, our RefactorErl-based semantic analysis approach is more accurate and can reduce false-positive hits. RefactorErl can find more types of vulnerabilities.

There is also another prototype [7] based on RefactorErl, which identifies similar vulnerability types. This work focuses on providing a systematic way to identify "trust zones" in Erlang applications at the design phase. The potential benefit of this approach is that security verification can be simplified by using these "trust zones" as the basis of the static analysis.

4 CONCLUSION

Secure coding is an important topic in software development. As the number of user interactions and network traffic increases, we see a significant growth in the number of cyber-attacks, which can not only harm customers but affect the reputation of companies as well. Static analysis can help identify the security issues in the code in the early phase of development and let the developers fix them.

In this paper, we presented some of the Erlang security vulnerabilities and a static analyser tool that can identify fifteen different types of secure coding issues in Erlang source code. We presented the selection algorithm and the false-positive reduction steps and demonstrated how a developer can use the tool to detect security issues.

REFERENCES

[1] Joe L. Armstrong. 2003. Making reliable distributed systems in the presence of software errors. https://api.semanticscholar.org/CorpusID:28795665

[2] Brigitta Baranyai, István Bozó, and Melinda Tóth. 2020. Supporting Secure Coding with RefactorErl. In *Talk at the 19th ACM SIGPLAN International Workshop on Erlang, Virtual Event*, Vol. 23.

[3] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. 2011. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*. Tallin, Estonia, 138–148.

[4] Baranyai Brigitta. 2021. Funkcionális nyelvek és a statikus kódelemzéssel támogatott biztonságos szoftverfejlesztés *(Thesis presented at the National Student Association Conference)*. ELTE, 70.

[5] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action*. Manning Publications.

[6] CodeChecker. [n. d.]. *Documentation*. https://codechecker.readthedocs.io/en/latest/

[7] Viktória Fördős. 2020. Secure Design and Verification of Erlang Systems. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang* (Virtual Event, USA) *(Erlang 2020)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3406085.3409011

[8] Erlang Ecosystem Foundation. [n. d.]. *Secure coding and deployment hardening guidelines*. https://erlef.org/wg/security

[9] The OWASP Foundation. 2019. *Application Security Verification Standard*. https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf vsn 4.0.

[10] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. 2009. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009 (Studia Universitatis Babeş-Bolyai, Series Informatica, Vol. 54(2009) Sp. Issue)*. Cluj-Napoca, Romania, 7–16.

[11] László Lövei, Lilla Hajós, and Melinda Tóth. 2010. Erlang Semantic Query Language. In *Proceeding of 8th International Conference on Applied Informatics, ICAI 2010, ISBN 978-963-98-94-72-3*. Eger, Hungary, 165–172.

[12] Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. 2018. Towards Secure Erlang Systems. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang* (St. Louis, MO, USA) *(Erlang 2018)*. Association for Computing Machinery, New York, NY, USA, 67–70. https://doi.org/10.1145/3239332.3242768

[13] SpotBugs. [n. d.]. *Documentation*. https://spotbugs.readthedocs.io/en/latest/introduction.html

[14] Francesco Cesarini & Simon Thompson. 2009. *Erlang programming*. O'Reilly. http://shop.oreilly.com/product/9780596518189.do

[15] Michael Truog. [n. d.]. *Primitive Erlang Security Tool (PEST)*. https://github.com/okeuday/pest

[16] M. Tóth and I. Bozó. 2012. Static Analysis of Complex Software Systems Implemented in Erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743.