Identifying Security Issues in Elixir Web Applications

Smiljana Knežev
Faculty of Informatics, ELTE, Eötvös Lóránd University, Budapest, Hungary

István Bozó
Faculty of Informatics, ELTE, Eötvös Lóránd University, Budapest, Hungary

Melinda Tóth
Faculty of Informatics, ELTE, Eötvös Lóránd University, Budapest, Hungary

Abstract

The security of software products is extremely important in the era of internet-based applications. Building secure web applications is not straightforward. Several guidelines and tools were developed to support this process. The biggest challenge for those tools is to not overwhelm the developers with false-positive hits. This paper aims to investigate the use of static analysis for accurate vulnerability identification in the case of Elixir web applications.

2012 ACM Subject Classification Security and privacy → Web application security

Keywords and phrases Static analysis, Elixir, security vulnerabilities, XSS

Digital Object Identifier 10.4230/OASIcs.Programming.2025.22

Funding Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

1 Introduction

Elixir [13] became a widely used programming language for developing web-based, fault-tolerant, scalable applications. It inherited all the useful properties of the BEAM virtual machine [19].

Application security become one of the most important characteristics of web-based applications. Developing secure applications for a non-expert programmer is challenging. Therefore, several standards, guidelines, and tools support this process [21, 6, 8]. Static analyzer tools [20, 18, 22] can help to identify security vulnerabilities in an early stage of development [5].

Developing tools to identify security issues is not straightforward. The most challenging part is to provide valuable, useful results. Several issues can be quickly identified with simple text-based searches, but syntactic, tree-matching-based approaches usually provide more accurate results. However, both approaches result in lots of false-positive hits. Providing too many results for developers requires manual post-analysis of the result; it is time-consuming and error-prone. Semantic analysis-based approaches can help reduce false-positive hits. Control- and data-flow analysis-based approaches can help to produce more accurate results.

Security checkers already exist for Elixir [15, 17]. This paper aims to investigate the role of static analysis in the vulnerability identification process and provide a tool for accurate Elixir security analysis. In this paper, Elixir security refers to a subset of recommendations from the Erlang Ecosystem Foundation, focusing on the most common web application vulnerabilities [8]. Some of these vulnerabilities are specific to the Phoenix framework, written in Elixir, while others are part of general recommendations for Elixir.

The rest of the paper is structured as follows. Section 2 gives a short overview of the Elixir security guidelines and the tools used. Section 3 shows an example and demonstrates the need for semantic analysis in vulnerability detection. Section 4 introduced the web-related security bottlenecks in Elixir applications. Sections 5 and 6 define the security checkers we built based on the RefactorErl infrastructure and evaluate them on open-source Elixir projects. Finally, Sections 2.3 and 7 present related works and conclude the paper.

2 Background

Erlang [7] and the BEAM virtual machine [19] were designed for building fault-tolerant distributed applications. Its "Let it crash" concept allows processes to fail instead of handling runtime errors and provides an extensive mechanism to handle process failures. As a new language on the top of BEAM, Elixir [13] also inherited this property. However, security issues can not be handled with process error handling. Therefore in recent years, there has been a huge interest in secure application development on the BEAM. The Security Working Group [9] of the Erlang Ecosystem Foundation (EEF) defined several guidelines for building secure web applications on the BEAM: for Erlang and Elixir.

2.1 Web application security best practices for BEAM languages

EEF's Security Working Group [8] describes a special document for best practices for secure web applications running on BEAM. The recommendations are divided into five big groups: "Common Web Application Vulnerabilities", "Session Management Vulnerabilities", "TLS Vulnerabilities", "Information Leakage", and "Supply Chain Vulnerabilities". These recommendations are mostly related to the use of the Phoenix [14] framework for web applications. In this paper, we will focus on the "Common Web Application Vulnerabilities".

2.2 Analysing Elixir code with RefactorErl

RefactorErl [24] is a static source code analysis and transformation tool designed for Erlang. RefactorErl performs lexical and syntactic analysis on the source code, builds an Abstract Syntax Tree (AST), and calls a set of static semantic analysers to extend the AST with semantic information. The result of these analyses is stored in the Semantic Program Graph (SPG) [11]. The SPG can be considered as an intermediate source code representation for Erlang source code. Different static analysis, code comprehension, or refactoring functionalities can be built on top of the SPG [3]. RefactorErl also provides data- and control-flow analysis [24]. A security analyser framework was built for Erlang applications on top of RefactorErl [22, 23]. Our aim is to extend this framework with Elixir analysis.

Elixir programs are compiled to the same intermediate source-code representation (the so-called Abstract Format[1]) as Erlang programs. RefactorErl can build the SPG from the Abstract Format: it maps the Abstract Format to its AST and performs the semantic analysis on it. The semantics of the Erlang and Elixir generated trees are the same, thus we can run the semantic analysers designed for Erlang on the Elixir AST, and we can define Elixir static analyses within the RefactorErl framework.

[1] `https://www.erlang.org/doc/apps/erts/absform.html`

2.3 Related work

A variety of static analysis tools exist to assist developers in identifying security flaws and code quality issues. Popular tools include SonarQube [5], SpotBugs [18], and CodeChecker [12, 20], among others. These tools are designed to integrate seamlessly into commonly used integrated development environments (IDEs). Other examples of widely adopted static analysis tools include ESLint [10] for JavaScript, Bandit [16] for Python, and Brakeman [4] for Ruby on Rails. These tools all contribute to improved code security and quality by catching potential issues early in the development lifecycle, and are usually incorporated into CI/CD (Continuous Integration and Deployment) pipelines, automatically scanning for vulnerabilities with each build.

Although security checkers for Erlang have been around for a while, they are now gaining increased interest within the Elixir community.

The empirical study looks into vulnerability commits of 25 open-source Elixir projects [2]. It was found that 2% of commits were vulnerability-related. The study also found that in 9 out of 25 projects, one security vulnerability was present. The study suggests further research into the problem is needed and suggests the use of static code analysis tools for more detailed results.

Several static analysis tools support Elixir analysis. Sobelow [15] is the most popular security-focused static analysis tool for Elixir and its web framework Phoenix. It detects the following issues: Insecure configuration, Known-vulnerable Dependencies Cross-Site Scripting, SQL injection, Command injection, Code execution, Denial of Service, Directory traversal, and Unsafe serialization. The tool provides the confidence of each insecurity, dividing it into three color-coded categories: green–low, yellow–medium, and high–red. It also rather over-reports than under-reports, resulting often in false positives. Usually, the results of running a tool need to be triaged (evaluated). EEF recommends including Sobelow in a CI/CD pipeline so it can be run every time code is merged into the main branch [8].

Semgrep [17] is an open-source static analysis tool that supports more than 30 programming languages, including Elixir. Semgrep also allows user to enforce their own rules for code standards in a code-like manner. The tool uses pattern-oriented matching technology.

3 Motivation example

The importance of data-flow analysis when identifying and labelling security vulnerabilities like XSS attacks can be demonstrated in a small example. In Figure 1, we demonstrate an example of a controller from a sample Phoenix application.

We can observe the function `html_resp` taken from the EEF guideline on *Common Web Application Vulnerabilities on Cross-Site Scripting* [8] and its variations `html_resp2`, `html_resp3`, and `html_resp4`. Here, an XSS attack can occur if the second argument in the `html` function comes from an outside untrusted source.

As we can see in Figure 2, Sobelow [15] rightfully flags `html_resp` function as XSS in "html" with High Confidence, and also the `html_resp4` function. Even if we can see that this latter function is private, its parameter comes from function `html_resp3` and could be unsafe. Also, Sobelow flags function `html_resp2` as XSS in "html" – Medium Confidence even if we can see that the `var` parameter is defined in the same function as a constant value: "`<html></html>`". Sobelow does not flag `html_resp3` even though the arg parameter could be unsafe, and passing to an HTML function could lead to an XSS attack.

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller

  def home(conn, _params) do
    render(conn, :home, layout: false)
  end

  def html_resp(conn, %{"i" => i}) do
    html(conn,
     "<html><head>#{i}</head></html>")
  end

  def html_resp2(conn) do
    var = "<html></html>"
    html(conn, var)
  end

  def identity(arg) do
    arg
  end

  def html_resp3(conn, arg)  do
    var = identity(arg)
    html_resp4(conn, var)
  end

  defp html_resp4(conn, obj) do
    html(conn, obj)
  end
end
```

Figure 1 Sample Elixir module.

```
##############################################
#                                            #
#          Running Sobelow - v0.13.0         #
#  Created by Griffin Byatt - @griffinbyatt  #
#     NCC Group - https://nccgroup.trust     #
#                                            #
##############################################

Config.CSP: Missing Content-Security-Policy - High Confidence
File: lib/hello_web/router.ex
Pipeline: browser
Line: 10

------------------------------------------------

Config.HTTPS: HTTPS Not Enabled - High Confidence

------------------------------------------------

XSS.HTML: XSS in `html` - High Confidence
File: lib/hello_web/controllers/page_controller.ex
Line: 36
Function: html_resp4:35
Variable: obj

------------------------------------------------
```

```
------------------------------------------------

XSS.HTML: XSS in `html` - Medium Confidence
File: lib/hello_web/controllers/page_controller.ex
Line: 14
Function: html_resp2:12
Variable: var

------------------------------------------------

XSS.HTML: XSS in `html` - High Confidence
File: lib/hello_web/controllers/page_controller.ex
Line: 9
Function: html_resp:8
Variable: i

------------------------------------------------

...  SCAN COMPLETE ...
```

Figure 2 Sobelow scan results of motivational example.

Semgrep [17], along with the cli tool, offers an online editor for small test samples. In Figure 3, we see that Semgrep, with using *elixir.lang.security.xss-controller-html.xss-controller-html* rule flags only the `html_resp` function. It does not flag the `html_resp4` function because it is private, even though it is reachable through the `html_resp3` function.

Using data-flow analysis and the RefactorErl tool, we can spot the vulnerable functions (Figure 4). Besides `html_resp`, the tool flags `html_resp4` functions. Unlike Sobelow, it does not flag `html_resp3`, which uses interfunctional dataflow.

4 Secure Elixir web applications

Phoenix [14] is the most popular Elixir MVC framework for building web applications. Like Elixir, Phoenix uses the mix tool to create, build, and run applications. This tooling helps developers create safer web applications by including some libraries that prevent security issues.

test code live code NEW metadata docs Pro Turbo

```
1  defmodule HelloWeb.PageController do
2    use HelloWeb, :controller
3
4    def home(conn, _params) do
5      render(conn, :home, layout: false)
6    end
7
8    def html_resp(conn, %{"i" => i}) do
9      html(conn, "<html><head>#{i}</head></html>")
10   end
11
12   def html_resp2(conn) do
13     var = "<html></html>"
14     html(conn, var)
15   end
16
17   def identity(arg) do
18     arg
19   end
20
21   def html_resp3(conn, arg)  do
22     var = identity(arg)
23     html_resp4(conn, var)
24   end
25
26   defp html_resp4(conn, obj) do
27     html(conn, obj)
28   end
29 end
```

Figure 3 Semgrep scan results of motivational example.

```
(hello_csp@kca)8> ri:q("mods.funs.unsecure_xss_call").
Elixir.HelloWeb.PageController:html_resp/2
    'Elixir.Phoenix.Controller':html(_conn@1, <<"<html><head>", case _i@1 of
            _@1 when erlang:is_binary(_@1) -> _@1;
            _@1 -> 'Elixir.String.Chars':to_string(_@1)
        end/binary, "</head></html>">>)
Elixir.HelloWeb.PageController:html_resp4/2
    'Elixir.Phoenix.Controller':html(_conn@1,
        _obj@1)
ok
```

Figure 4 RefactorErl scan results of motivational example.

EEF defines the following "Common Web Application Vulnerabilities": cross-site scripting, content security policy, cross-site request forgery, cross-site WebSocket hijacking, SQL injection, denial of service, and client-side enforcement of server-side security [9].

4.1 Cross-site scripting

Cross-site scripting (XSS) is a common security vulnerability where a web application incorporates user input without performing validation, allowing the user to inject malicious code into the victim's web browser.

Phoenix framework provides protection from XSS attacks. However, weak spots for potential abuse are: `Phoenix.HTML.raw/1` and `Phoenix.Controller.html/2` functions, which render the user's input. Rendering tainted input can also be achieved using a pipeline of functions *send_resp* and *put_resp_content_type* or *put_resp_header* given that the *put_resp_content_type* or *put_resp_header* are accepting *"text/html"* and *send_resp* is accepting tainted value as third argument.

4.1.1 Content Security Policy

Content Security Policy (CSP) is a security feature that helps prevent XSS attacks and data injection. This feature allows developers to create policies to whitelist safe resources for the application to load. CSP is not intended as a first-line defence against content injection vulnerabilities but rather an in-depth defence. Nevertheless, the CSP is a standard de facto security feature.

Using CSP means adding the Content-Security-Policy HTTP Header to a webpage with defined resources application is allowed to load. Phoenix web applications support CSP Level 2 defined by W3 [25]. This document specifies rules and allowed directives that can be used.

EEF recommends using the following Phoenix plugs for CSP: `put_secure_headers`, `plug_content_security` policy, or using nonces for `phoenix_live_dashboard`. If arguments are omitted while using these plugs, default arguments will be applied, but such arguments are generally not considered to be safe. When providing arguments, data-flow analysis can be applied to check if the source of those arguments is safe.

4.2 Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is a vulnerability in web applications that allows an attacker to issue commands on behalf of a victim user.

Phoenix provides default protection against CSRF by combining the `protect_from_forgery plug` with automatically included CSRF tokens in its form helpers when POST requests are used in HTML forms. However, when GET requests are used to change the state, the application is vulnerable to CSRF, which Action Reuse commonly refers to as CSRF.

4.3 Cross-Site WebSocket Hijacking

Cross-Site WebSocket Hijacking (CSWHS) is a vulnerability similar to Cross-Site Request Forgery, but an attacker seeks to establish a WebSocket connection. If successful, the attacker can establish a direct line of communication between themselves and the server, using the victim's session as if they were the victim.

Phoenix's connect_info mechanism passes session information to the WebSocket. When the session is exposed using this mechanism, Phoenix applies CSRF protections to prevent attackers from hijacking the WebSocket connection.

CSRF token verification in Phoenix does not block WebSocket connections but only applies when session information is requested. The connect callback must close the connection if authentication fails. Phoenix also offers the `check_origin` parameter to block connections based on the browser's Origin header, allowing connections only if the origin matches the configured hostname by default. If cross-origin connections are needed, the origin check can be disabled, but authentication should then rely on a token passed via query parameters instead of cookies.

4.4 SQL Injection

SQL injection is a type of attack targeting web applications where malicious input is processed by the underlying database, leading to unauthorized operations being executed.

Phoenix applications usually use Ecto, a database wrapper and query generator. When used properly, Ecto prevents SQL injection. However, if raw SQL queries are being passed to Ecto, it could lead to SQL injection. SQL injection occurs when user input is used to build a query directly.

For example, the `Ecto.Adapters.SQL.query/4` function executes a custom SQL query. If user input is passed here, it could lead to SQL injection.

4.5 Denial of Service

A Denial-of-Service attack occurs when a malicious user attempts to overload a website with requests, slowing it down or making it inaccessible. In the context of Elixir applications, one subtle and dangerous way this can occur is through the uncontrolled creation of atoms based on user input. Since the atoms in Elixir and Erlang are not garbage collected and are stored permanently in memory, an attacker can trigger the creation of a large number of unique atoms that can exhaust the system's atom table, which has a fixed size (by default, 1,048,576 entries). Once this limit is reached, the virtual machine will crash, making this a particularly serious risk for Denial-of-Service vulnerabilities in applications written in Elixir.

4.5.1 Atom exhaustion

Atoms are constant string literals that are stored in the atom table. Since the atom table is not garbage collected, the BEAM limits the number of atoms. When we reach the limit, the BEAM terminates. Therefore, letting the system generate atoms in an uncontrolled way may lead to DoS attacks.

4.5.2 Application layer attacks

If a Phoenix application allows users to do heavy computation tasks, it could lead to a denial of service attack.

Several libraries can be used to rate limit such functions and requests: PlugAttack, Hammer, and Ex_rated. Using these libraries is a recommendation and not a rule against users doing heavy computations; therefore, it is not a security prevention measure.

5 Identifying vulnerabilities in Phoenix

Vulnerabilities in Phoenix are detected by assessing whether the EEF recommendations are being followed. In some cases, like CSRF, this is simply checking whether recommended plugs are being used, while in others, like the XSS attack, a more complex analysis is required. Algorithms 1 and 3 describe these two groups of checkers. We derive the security checkers for the issues listed in Section 4 from these algorithms:

5.1 Cross-site scripting

The algorithm for detecting XSS vulnerabilities, referred to as Algorithm 1, is a modified version of the algorithm defined in [1] for identifying OS injections. The location of function calls posing an XSS threat is found, and then data-flow analysis is run on those functions' parameters. By leveraging RefactorErls' data-flow analysis, we can implement security checkers that reduce false positives. We can trace the origins of arguments and see whether they come from a trusted source or not.

Algorithm 1 Algorithm for detecting XSS vulnerabilities.

```
1: Function get_calls_for_xss
2: Matchers ← [{'Elixir.Phoenix.Controller',[{html, 2]},
3: {'Elixir.Phoenix.HTML',[raw, 1]}]}
4: CandidateFuns ← get_calls_for(Fun,Matchers)
5: FunParamTuples ← new list
6: for C ∈ CandidateFuns do
7:     Param ← get_expr_params_for(C, Matchers, get_all_children(C))
8:     add_to_list({C, Param}, FunParamTuples)
9: end for
10: DataFlowValues ← get_origins(FunParamTuples)
11: FilteredDataFlowValues ← get_unsafe_funs(DataFlowValues)
12: UnknownFuns ← get_funs_no_body(FilteredDataFlowValues)
13: ExportedFuns ← get_exported_funs(FilteredDataFlowValues)
14: return merge(UnknownFuns, ExportedFuns)
```

Helper function get_calls_for_xss searches for function calls defined in Matchers. For a vulnerable candidate found, the parameters are checked with get_expr_params_for. Here we filter out the parameters on which the data-flow analysis is going to be run. The get_origins function runs the data-flow analysis and returns possibly tainted input. Further, the get_unsafe_funs checks against the developer's whitelisted functions. Any function that appears on the whitelist is treated as reliable. In functions get_funs_no_body and get_exported_funs we check if the parameters are functions without bodies or functions that are exported. In both cases, the parameters can introduce security vulnerabilities into the application; as a result, these expressions are automatically classified as unsafe.

Checking if CSP is employed is checking if `put_secure_browser_headers` plug or `plug_content_security_policy` plug are being used. Since these plugs can accept policies, we can check the origin of these arguments by using Algorithm 1.

Algorithm 2 check_xss_pipeline.

```
1: Function check_xss_pipeline
2: AnalyzedFunctions ← analyze_funs(send_resp_call.first_argument)
3: if is_tainted(AnalyzedFunctions) then
4:     if is_tainted(send_resp_call.third_argument) then
5:         return send_resp_call
6:     end if
7: else
8:     put_resp_fn ← find_put_resp(AnalyzedFunctions)
9:     if put_resp_fn exists then
10:        if put_resp_fn.third_argument = "text/html" or
   is_tainted(put_resp_fn.third_argument) then
11:            if is_tainted(send_resp_call.third_argument) then
12:                return send_resp_call
13:            end if
14:        end if
15:    end if
16: end if
```

Algorithm 2 describes finding XSS vulnerability if the content-type of the server response is being set by the user by using a pipeline consisting of *send_resp* and *put_resp* functions. The *analyze_funs* is matching the first ten lines of Algorithm 1 for *send_resp* first parameter. The resulting data-flow value *AnalyzedFunctions* is checked for being tainted by applying lines 11–14 from Algorithm 1. If the first argument of *send_resp* is tainted, we check if the third one is also tainted, and if yes, the function is vulnerable. If the first argument is a *put_resp* call, then we check its third argument for value `"text/html"` or being tainted. If that is the case, then we can look into this argument of *send_resp*, and if it is tainted too, the function is vulnerable.

5.2 Cross Site Request Forgery

Algorithm 3 is a modified version of Algorithm 1 where we are only looking for the existence of a function defined in a plug. Here, we are looking to see if the function is missing and not if it is being used with trusted sources. CSRF detection is described in Algorithm 3.

Algorithm 3 Algorithm for detecting if csrf protection exists.

```
1: Function get_calls_for_csrf
2: Matchers ← [{'Elixir.Phoenix.Controller', [protect_from_forgery, 2]}]
3: Funs ← All_funs_in_module
4: for Fun ∈ Funs do
5:     Result ← get_calls_for(Fun,Matchers)
6: end for
7: if Result is empty list then
8:     return ok
9: else
10:    return "CSRF protection missing"
11: end if
```

5.3 Cross-Site WebSocket Hijacking

In Phoenix, there is not a single built-in function or parameter that explicitly guarantees token-based authentication is passed through query parameters across the framework for CSWHS protection. Instead, developers typically implement token-based authentication using query parameters manually, depending on their specific use case. Therefore, there is not a simple and reliable checker for this issue.

5.4 SQL injection

SQL Injection is checked by applying Algorithm 1 and checking all the Ecto vectors for SQL injection defined by EEF. In Algorithm 1, the second line has to be changed to:

```
Matchers ← [{{'Elixir.Ecto.Adapters.SQL',[
{query,2},{query,3},{query,4},
{'query!', 2}, {'query!', 3}, {'query!', 4},
{query_many, 2}, {query_many, 3}, {query_many, 4},
{'query_many!', 2}, {'query_many!', 3}, {'query_many!', 4},
{stream, 2}, {stream, 3}, {stream, 4}]},
{'Ecto.Repo', [ {query, 2}, {query, 3}, {query, 4},
{'query!', 2}, {'query!', 3}, {'query!', 4},
{query_many, 2}, {query_many, 3}, {query_many, 4}]}]
```

5.5 Denial of service

DoS and atom exhaustion checkers are not just Phoenix-specific, web-related security issues but general Elixir vulnerabilities. Therefore, these have already been implemented in the RefactorErl tool [23]. Application layer attacks are dependent on logic and implementation, making them unsuitable for detection with static analysis tools.

6 Evaluation

The results discussed below were collected by running Refactorl's security checkers and running the Sobelow tool for selected examples and projects. The results were then manually checked and compared. We only compared results for the vulnerabilities described here, even though Sobelow currently supports checkers that are still being developed for Refactoerl. The implementation of the checkers is currently available in a private repository within RefactorErl and is expected to be made available in the future.

While Sobelow provides notably faster analysis, it doesn't offer the semantic evaluation and precision RefactorErl does.

Based on small examples, we observe that for CSP, Sobelow reports *"Missing Content-Security-Policy – High Confidence"* even if `put_secure_browser_headers` is supplied, but without a user-defined policy. If the policy is not provided to `put_secure_browser_headers`, it will use a default one implemented by the library. This policy does not cover all cases but provides protection against the most common attacks. RefactorErl also flags this case as vulnerable, but when the severity levels are included, it would mark it as Low Confidence. Semgrep does not have a rule for checking for CSP yet.

When it comes to checking CSRF action by reuse, where GET and POST requests perform the same changing state, Sobelow reports false positives in some cases. If we have branching, like an if–else statement, and we place these GET and POST requests in different branches so that either will get executed and never both, Sobelow still reports this code as *"CSRF via Action Reuse – High Confidence"*. Since we are analysing compiled Phoenix code with Refactoerl, we are already working with the code that the compiler checked is reachable, so this issue is avoided. Semgrep does not yet have a rule for CSRF.

With SQL injection, both Semgrep with rule *elixir.lang.security.sql-injection.sql-injection* and Sobelow do not recognize all the vectors for SQL injection specified by EEF. Furthermore, Sobelow flags false positives every time `Ecto.Adapters.SQL.query/2` function is used, even if the arguments are safe. Sobelow provides a feature to comment *sobelow_skip ["SQL"]* above the function if the developer finds it safe, and then it would be excluded from the findings from a security scan. Even though it can be useful, this is just a feature to filter

through results more easily and not a security check. RefactorErl SQL injection checker checks against all defined vectors, using the Algorithm 1 and performs data-flow analysis to reduce false positives.

6.1 Scanning open-source projects

We analyzed and compared the results with Sobelow for the following projects. The projects were chosen so that they are actively used by the community and have the vulnerabilities discussed in this paper. In Table 1, we see the size of the projects and their respectable lines of code.

- Hexpm[2] is a repository website that hosts packages used by Hex, Elixir's package manager. Hexpm is widely used in Elixir projects to fetch different libraries.
- Changelog[3] is an open-source podcast software that provides content for developers. These podcasts are listened to by more than tens of thousands of users.
- Plausible Analytics[4] is an open-source alternative to Google Analytics that is more concerned with privacy. It is an independent tool funded by its subscribers. It has 15 thousand paid subscribers, and more than 136 billion tracked page reviews.

Table 1 Projects overview.

Project	Loc
Hexpm	14159
Changelog	11422
Analytics	76094

In Tables 2, 3 and 4, the results obtained from Sobelow and RefactorErl are compared for the common vulnerabilities in web applications described by EEF. The column names are defined as follows: Num is the total number of found vulnerabilities in each category, FP is False Positive, and FN is False Negative. In the RefactorErl section, we differentiate between reachable and unreachable vulnerabilities. Reachable vulnerabilities are those that are accessible to outside users as exported functions, while unreachable vulnerabilities are present in the beam code but are not used in the project and are not accessible through developer-defined functions that are exported. Unreachable vulnerabilities are the result of Elixir macro expansions.

XSS attacks are by far the most common vulnerability in the analysed projects. We distinguish between three cases of XSS attacks, as already described above. The most common one is using *Phoenix.HTML.raw/1* with tainted values.

Since we use beam code for analysis, we can find the function calls that are sometimes hidden in, e.g., ~H sigil that Sobelow does not find. In Phoenix ~H is a sigil used to define HEEx, HTML + Elixir, templates. It is a shorthand for writing HTML-safe templates that can include embedded Elixir code. The embedded Elixir code is not picked up by Sobelow.

Additionally, Table 2 shows that in some cases a vulnerability is detected but flagged as *Traversal.FileModule: Directory Traversal in File.read!*, as illustrated in the Figure 5.

[2] `https://github.com/hexpm/hexpm`
[3] `https://github.com/thechangelog/changelog.com`
[4] `https://github.com/plausible/analytics?tab=readme-ov-file`

Table 2 Hexpm analysis results.

Hexpm							
	Sobelow			RefactorErl			
Vulnerability	Num	FP	FN	Num		FP	FN
				Reachable	Unreachable		
Xss attacks	19	0	1	20	0	0	0
Csp	1	0	0	1	0	0	0
Csrf	1	0	0	1	0	0	0
Sql injection	1	0	0	1	7	0	0
DOS	10	0	0	10	0	0	0

Table 3 Changelog analysis results.

Changelog							
	Sobelow			RefactorErl			
Vulnerability	Num	FP	FN	Num		FP	FN
				Reachable	Unreachable		
Xss attacks	63	3	0	61	1	0	0
Csp	1	0	0	1	0	0	0
Csrf	12	1	0	11	0	0	0
Sql injection	1	1	0	0	4	0	0
DOS	3	0	0	3	0	0	0

Table 4 Plausible analytics analysis results.

Plausible Analytics							
	Sobelow			RefactorErl			
Vulnerability	Num	FP	FN	Num		FP	FN
				Reachable	Unreachable		
Xss attacks	17	1	3	21	0	1	0
Csp	3	0	0	3	0	0	0
Csrf	5	0	0	5	2	0	0
Sql injection	1	0	0	1	17	0	0
DOS	5	2	1	4	2	0	0

```
def compile(path, _name) do
  path
  |> File.read!()
  |> Earmark.as_html!(%Earmark.Options{gfm: true})
  |> header_anchors("h3")
  |> header_anchors("h4")
  |> Phoenix.HTML.raw()
end
```

Figure 5 Hexpm XSS False Negative example.

In all three projects, Sobelow's and RefactoErl's results match for CSP, the difference being in the intent whether to interpret it as high or low severity level as discussed above.

Table 2 confirms consistent results for CSRF. In contrast, in Table 3, we observe one false positive value in Sobelow where one CSRF Action by reuse being vulnerability was falsely flagged. It only had a GET method action without the corresponding POST method. In Table 4 we have two unreachable CSRF calls that are a result of expanding macros not having CSRF protection.

When it comes to SQL injection, many vulnerabilities remain unreachable because of the use of *Ecto.Repo* macro, which can expand to functions using *Elixir.Ecto.Adapters.SQL'* vectors: *{query,3}*, *{'query!', 3}*,*{'query_many', 3}* and *{'query_many!', 3}* depending on the version of Ecto used. We also see one false positive value in Table 3. In this case, the arguments of the query are constant and are not tainted.

Results for Denial Of Services are matching in Table 2 and Table 3, while in Table 4, we have two unreachable positives due to macro expansion in RefactorErl and two false positives and one false negative in Sobelow. Two false positives come from binary interpolation of constant values. One false negative comes from not recognizing that the input of function *String_to_atom* is the constant list (Figure 6). This can be detected with RefactorErl using data-flow analysis.

```
@features [                                  defmacro list_short_names() do
  Plausible.Billing.Feature.Goals,             @features
  Plausible.Billing.Feature.StatsAPI,          |> Enum.map(fn mod ->
  Plausible.Billing.Feature.Props,               Module.split(mod)
  Plausible.Billing.Feature.Funnels,             |> List.last()
  Plausible.Billing.Feature.RevenueGoals,        |> String.to_atom()
  Plausible.Billing.Feature.SiteSegments       end)
]                                            end
```

Figure 6 Plausible Analytics: DOS False Negative example.

7 Conclusions

Application security must be considered in all software products. Developer communities highly desire tools that help identify possible vulnerabilities. We focus our research on Elixir Web applications and want to provide a tool for reliable security checking based on static analysis. In this paper, we presented our approach to Web application vulnerability issues based on EEF security guidelines for the common web application vulnerabilities.

The work on security checkers relies on previous work on security vulnerabilities in Erlang and Elixir, which is already implemented in the RefactorErl tool. By extending the algorithm for os injection and adding two more new ones, we get the three basic algorithms for detecting common Web application vulnerabilities. All algorithms use data flow analysis to detect tainted values with greater precision.

Since Elixir and Erlang run on the same virtual machine, BEAM, we analyze the compiled Elixir Web applications by loading beam files into Refactoerl. When analyzing compiled Elixir code, where all the macros are extended, we come across vulnerabilities that are present because a certain macro was used, but that code is not accessible to the application user, so we devised a new category called Unreachable vulnerabilities. In addition to that, we looked at false positive and false negative hits and compared our results with Sobelow.

We compared our result with the security scanners already available for Elixir and found that the semantic filtering-based approach can significantly reduce the false positive and the false negative ratios. Results were verified on open-source projects.

Further steps. We aim to develop new security checkers specifically for issues described in the EEF guidelines regarding Elixir and Phoenix and refine the already existing ones. Furthermore, we plan to analyze more open-source projects.

Our method marks unknown functions as vulnerable, i.e. functions producing tainted inputs. To improve the accuracy of false positive elimination in vulnerability identification for Elixir, we need to develop an understanding of frequently used library functions, preventing them from being mistakenly marked as vulnerabilities in our analysis tools.

References

1 Brigitta Baranyai, István Bozó, and Melinda Tóth. Supporting secure coding with RefactorErl. In *13th Joint Conference on Mathematics and Informatics – MaCS 2020*, pages 24–25, 2020.

2 Dibyendu Brinto Bose, Kaitlyn Cottrell, and Akond Rahman. Vision for a secure elixir ecosystem: An empirical study of vulnerabilities in elixir programs. In *Proceedings of the 2022 ACM Southeast Conference*, ACM SE '22, pages 215–218, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3476883.3520204.

3 I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. 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*, pages 138–148, Tallin, Estonia, October 2011.

4 Brakeman. Brakeman. https://brakemanscanner.org/. Accessed: 29-04-2025.

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

6 Carnegie Mellon University, Software Engineering Institute. Cert coding standards. https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards. Accessed: 29-04-2025.

7 Francesco Cesarini and Simon Thompson. *Erlang programming*. O'Reilly, June 2009.

8 Erlang Ecosystem Foundation Security Working Group. Security working group. https://erlef.github.io/security-wg/web_app_security_best_practices_beam/index. Accessed: 29-04-2025.

9 Erlang Ecosystem Foundation Security Working Group. Web application security best practices for beam languages. https://erlef.org/wg/security. Accessed: 29-04-2025.

10 ESLint. Eslint. https://eslint.org/. Accessed: 29-04-2025.

11 Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. 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*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.

12 Horváth, Kovács, Szalay, Porkoláb, Orbán, and Krupp. Codechecker. https://codechecker.readthedocs.io/en/latest/. Accessed: 29-04-2025.

13 Saša Jurić. *Elixir in Action*. Manning, 2024. URL: https://www.manning.com/books/phoenix-in-action.

14 Geoffrey Lessel. *Pheonix in Action*. Manning, 2019. URL: https://www.manning.com/books/elixir-in-action-third-edition.

15 NCC Group. Sobelow. https://github.com/nccgroup/sobelow#installation. Accessed: 29-04-2025.

16 PyCQA (Python Code Quality Authority). Bandit. https://bandit.readthedocs.io/en/latest/. Accessed: 29-04-2025.

17 Semgrep, Inc. Semgrep. https://github.com/semgrep/semgrep. Accessed: 29-04-2025.

18 SpotBugs Team. Spotbugs. https://spotbugs.readthedocs.io/en/latest/introduction.html. Accessed: 29-04-2025.

19 Erik Stenman. The BEAM book. `https://blog.stenmans.org/theBeamBook/`. Accessed: 29-04-2025.

20 Richárd Szalay and Ádám Balogh. On the applicability of static analysis for system software using codechecker. In *2024 7th International Conference on Software and System Engineering (ICoSSE)*, April 2024. `doi:10.1109/ICoSSE62619.2024.00011`.

21 The OWASP Foundation. Application security verification standard. `https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf`. Accessed: 29-04-2025.

22 Melinda Tóth and István Bozó. Supporting secure coding for erlang. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, SAC '24, pages 1307–1311, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3605098.3636185`.

23 M. Tóth. Don't let it crash – How we applied our security checks on elixir code. Talk at ELixirConf, Lisbon, 2024, `https://www.elixirconf.eu/talks/dont-let-it-crash-how-we-applied-our-security-checks-on-elixir-code/`.

24 M. Tóth and I. Bozó. 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, 2012.

25 W3. W3. `https://www.w3.org/TR/CSP2/#directives`. Accessed: 29-04-2025.