Elixir Security Roadmap: A CTO Level Guide

Michael Lubas, Paraxial.io Founder
michael@paraxial.io

Published November 2023

Paraxial.io

Overview

Motivation

The work outlined in this document is to help you prevent a data breach due to your Elixir web application being hacked. If you are reading this you understand why this is important. Elixir provides an excellent foundation on which to build secure applications, and the language has seen adoption in industries with high security requirements, including finance and healthcare. The amount of work required to secure any web application is substantial, and while Elixir provides a more secure base compared to other languages, proper security controls still need to be implemented.

Security work done in a business context is different compared with other departments. Software development teams can present a demo to the company showing an exciting new feature. Sales can show the number of meetings they book and how many result in closed deals. The goal of a security team is simple and difficult, prevent data breaches. If the Elixir application gets hacked, that is the foothold attackers will use to further compromised the company and leak data.

Equifax is a United States based credit bureau that was hacked in 2017, leading to a high profile data breach and financial settlement of $425 million[1]. The US Senate commissioned a report on the breach, detailing how hackers exploited a known vulnerability in Apache Struts[2]. Elixir is to Phoenix what Java is to Struts, meaning when a company neglects the security of their web applications the risk is massive.

Defending your own software from attack is an achievable goal. A common misconception about corporate security teams is they spend the day behind a hex editor doing magic to repel attackers. The reality is not as glamorous and there really is no secret trick. Companies that successfully stop attacks consistently put in the work: asset management, security checks, monitoring and testing their controls.

This is an opinionated guide on how to best allocate the limited time you have to reduce the risk of your Elixir application getting hacked. Every company has different security requirements, yet there are commonalities. An attacker exploiting remote code execution (RCE) in an internet facing web application is a disaster. The lessons learned from Equifax are applicable to every organization using Elixir.

---

[1] https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement

[2] https://www.hsgac.senate.gov/wp-content/uploads/imo/media/doc/FINAL%20Equifax%20Report.pdf

Scope

Elixir and Phoenix applications can be deployed in many different environments: bare metal servers, cloud servers (AWS, GCP, Azure, etc), or platform as a service (PaaS) providers (Gigalixir, Fly.io, Render). There are appreciable differences in how to secure each option. Generally speaking you will do less security work hosting your Elixir application on a PaaS compared with a cloud provider. For example, with AWS EC2 you must regularly update the underlying web server to apply security patches. Gigalixir handles this work for you.

This document is focused on securing Elixir and Phoenix applications. The web server, cloud environment, and external non-Elixir dependencies will not be covered. This is not to say that they should be out of scope from your own security testing. A secure Elixir application is one link in your company's overall cyber security strategy. This is a guide to ensuring that link is strong.

Audience

The intended audience of this document is a CTO, CISO, VP of Engineering, Manager, or Senior Developer at a company using Elixir in production.

Author

Michael Lubas is the founder of Paraxial.io, a cyber security company focused on Elixir. He created the Paraxial.io security platform, and helps companies securely adopt Elixir through private trainings, strategic consulting, and penetration testing engagements. He is the author of several open source Elixir security projects, including Potion Shop[3] and Exploit Guard[4].

Contact: michael@paraxial.io

[3] https://paraxial.io/blog/potion-shop

[4] https://paraxial.io/blog/exploit-guard

Roadmap

Milestone 1: Asset Management and Updates

How many Elixir applications do you currently have deployed? How many are accessible by anyone on the internet? What version of Phoenix, Ecto, and Plug are they using? These are the basic questions an asset inventory can answer. When the news of a major vulnerability becomes public, the highest priority goal of a security team suddenly becomes:

1. Identify all the software in the organization that is affected by the vulnerability. Internet facing software is the highest priority because attackers use automated scanners to identify and exploit vulnerable servers.
2. Upgrade the software to the safe version.

You do not want to be searching your network and upgrading systems that have not been changed in years the day of a major vulnerability release. That is often the reality for teams that do not prepare for these events. There is no asset inventory standard, and there are many vendor products you can buy to assist with this project. The data you need comes down to:

1. A list of every deployed Elixir application
    i. Elixir version
    ii. Erlang Version
    iii. Dependency Versions (Phoenix, Ecto, Plug, Cowboy, etc.)
    iv. The URL where it is deployed
    v. Does it store customer data or PII?
    vi. Does it belong to the development or production environment?
    vii. Is it publicly accessible?

Note that an Elixir application that anyone on the internet can send HTTP requests to is publicly accessible, even if an attacker cannot create an account or login to a valid session. If the application is behind a VPN then it is not publicly accessible.

When the Log4Shell vulnerability[5] became public, every company using Java had to check the version of Log4j they were using and update each vulnerable system before an attacker could break in. That statement assumes a company has a list of each

[5] https://en.wikipedia.org/wiki/Log4Shell

application using Log4j. When the vulnerability dropped, the number one priority of many security teams suddenly became hunting down legacy Java applications.

When you go to investigate the current state of your company's asset inventory, you may find:

1. The source code of each Elixir application in Github.
2. The AWS servers where it is deployed.

This is better than nothing, but not enough. Consider CVE-2020-15150, remote code execution in the Elixir library Paginator[6]. You check the mix.lock file in Github, see that it is using a patched version, and believe the production application is secure. Later you discover the vulnerable version of Paginator is running in production because someone excluded the lock file from CI/CD to fix an error, resulting in an older version being fetched at deploy time.

The best source of truth for what you have running in production is the actual deployed service. Paraxial.io supports this feature via App Audit[7]. Even if you are not in a position to purchase security software, having an asset inventory is so important that it is worth prioritizing this work. One option is to create a spreadsheet of all your Elixir applications and where they are deployed. The following one line of Elixir code gets a full inventory at runtime:

```
iex> Application.loaded_applications() |> Enum.sort_by(&(elem(&1, 0)))
```

Require every Elixir application at your company to get this inventory at startup and upload it to whatever service you are using to track application metrics. If your service does not support ingesting this type of data just put it in an S3 bucket. The important thing is you will now have a true inventory of what software is deployed.

The reason asset management is so critical for web applications is that they tend to be internet facing. If you have ever peaked at the raw log output of a deployed application, you have probably seen GET requests for */wp-config.php,* even if you are using Elixir. These are bots, constantly scanning every live host on the internet, searching for vulnerable servers to compromise. When Log4Shell became public HTTP traffic logs were full of exploitation attempts.

When it comes to upgrading your Elixir applications it is somewhat like exercise, the more frequently you do it the less painful it will be. Have your development teams make a habit of updating dependencies[8] monthly. There will be errors and code changes will

---

[6] https://www.alphabot.com/security/blog/2020/elixir/Remote-code-execution-vulnerability-in-Elixir-based-Paginator-project.html

[7] https://www.paraxial.io/blog/app-audit

[8] https://paraxial.io/blog/hex-security

have to be made, however it is better to figure out a process for performing this work during normal business hours and not on the weekend of a critical incident.

Project Phases

1. Create a list of every Elixir application running in production.
2. For every application, document:
    i. Elixir version
    ii. Erlang Version
    iii. Dependency Versions (Phoenix, Ecto, Plug, Cowboy, etc.)
    iv. The URL where it is deployed
    v. Does it store customer data or PII?
    vi. Does it belong to the development or production environment?
    vii. Is it publicly accessible?
3. Have each application upload its inventory on deployment.
4. Schedule regular audits to ensure the inventory is correct.
5. Check every Elixir application for outdated dependencies.
6. Automate checks for outdated dependencies in CI/CD.
7. Schedule regular audits to ensure Elixir applications are being updated.

Milestone 2: Developer Education

Consider a company that does not train its developers in secure coding. Security bugs are introduced, deployed into production, and now there is a time window where anyone on the internet could hack the application. The security team notices this flaw during a penetration test, documents the risk, and fixes it.

Now consider a development team that has been trained in common vulnerable code patterns and introduces fewer security bugs. There are major financial and security benefits:

1. The risk of a data breach is reduced.
2. Development schedules do not get disrupted fixing security issues.
3. Preventable security flaws never make it to production.

A common recommendation for Elixir projects is to use Sobelow, the static analysis security tool. This guide will go into detail on Sobelow in the next section. To use Sobelow effectively the developer must have prior knowledge about different types of vulnerabilities (RCE, SQL injection, XSS, etc).

The EEF Security Working Group publishes two guides on Elixir security:

1. Secure Coding and Deployment Hardening Guidelines[9]
2. Web Application Security Best Practices for BEAM languages[10]

Both are recommended reading for all Elixir developers. When it comes to applying the knowledge from these guides, Potion Shop[11] is an open source vulnerable Phoenix application designed for teaching students about common web security problems. The repository includes a tutorial file for developers completely new to security. Developers using Sobelow for the first time have also found Potion Shop to be a useful project which generates true positive findings.

There are two domains of knowledge relevant here:

1. General web security topics: vulnerabilities, access control, cryptography.
2. Elixir specific web security topics: binary_to_term, action reuse CSRF.

---

[9] https://erlef.github.io/security-wg/secure_coding_and_deployment_hardening/

[10] https://erlef.github.io/security-wg/web_app_security_best_practices_beam/

[11] https://github.com/securityelixir/potion_shop

For general web security, “The Web Application Hacker’s Handbook, 2nd edition” is excellent. The author and creator of Burp Suite, Dafydd Stuttard, said in a 2020 interview there will not be a 3rd edition, and instead recommends the PortSwigger Web Security Academy[12]. The labs are free, constantly updated with new vulnerabilities, and are an excellent resource for new students.

For learning more about Elixir security, here are the most relevant resources:

1. Paraxial.io Blog[13]
2. Potion Shop Github[14]
3. Sobelow Github[15]
4. EEF Guidelines[16]
5. Bram Verburg's Blog[17]
6. ElixirConf 2017 - Plugging the Security Holes in Your Phoenix Application[18]
7. Elixir Secure Coding Training Livebooks[19]

Paraxial.io does offer private Elixir security training using Potion Shop for student labs. If the cost of private training is out of your budget a similar training is often available during ElixirConf and CodeBEAM. Check the conference pages for upcoming dates.

Project Phases

1. Survey your development team's security experience.
2. Communicate available resources for Elixir security.
3. Establish a training program to educate developers about security.
4. Ensure new developers are trained in security as part of onboarding.

---

[12] https://portswigger.net/web-security

[13] https://paraxial.io/blog/index

[14] https://github.com/securityelixir/potion_shop

[15] https://github.com/nccgroup/sobelow

[16] https://erlef.github.io/security-wg/

[17] https://blog.voltone.net/

[18] https://www.youtube.com/watch?v=w3lKmFsmlvQ

[19] https://github.com/podium/elixir-secure-coding

Milestone 3: Elixir Code Security Tools

There are several Elixir specific security tools:

1. **Sobelow**, scans your source code for OWASP-style bugs (XSS, RCE)
2. **mix deps.audit - MixAudit**, checks for vulnerable dependencies
3. **mix hex.audit**, checks for retired dependencies which are no longer updated
4. **Exploit Guard**, blocks hacking attempts at runtime

The first three tools can all be added to your CI/CD pipeline, and that is the ideal state most companies using Elixir strive for. A new pull request is opened, Sobelow checks the code for security problems, deps.audit checks for vulnerable dependencies, and hex.audit checks for retired dependencies. If there are no findings then deploy the code.

The first time these commands are run on a project the findings for deps.audit and hex.audit are manageable. Sobelow tends to produce a large number of false positives. Triaging these findings is a significant project. There is also the matter of tracking the true positive Sobelow findings and ensuring they get fixed in a timely manor.

You should absolutely be using Sobelow to secure your Elixir application. The project scope tends to be larger than initially expected, running Sobelow is the easy part. Here is a rough breakdown:

1. Installing and running Sobelow initially, then in CI/CD.
2. Triaging findings, documenting false positives.
3. Documenting and fixing true positive vulnerabilities.
4. Training developers on the correct procedures to handle Sobelow findings.

The above list is from Elixir Security: Real World Sobelow[20], which goes into more detail on how to use Sobelow effectively. The need to train developers in security is closely related to using Sobelow effectively.

Exploit Guard is different from the other tools listed in that it monitors the Elixir application at runtime for exploitation attempts. This means a remote code execution vulnerability exists in the deployed application, which is already a bad situation. When Exploit Guard is triggered, that is an extremely high quality signal that someone is actively trying to hack your application. A developer using a remote iex session will never trigger an alert. See the Exploit Guard launch post[21] for the technical details.

---

[20] https://paraxial.io/blog/real-sobelow

[21] https://paraxial.io/blog/exploit-guard

From a business perspective an upcoming SOC2 audit or HIPPA requirement is often the motivation for using these tools. Records that show scans are being run on a regular basis is an additional requirement. Paraxial.io is the off the shelf solution for managing results, and includes directions for triaging and fixing true positive findings. This guide to Sobelow findings is available publicly on Github[22].

All of these tools are open source, so exporting all this data into your current vulnerability management system is an option. More important than the technical details of your implementation is ensuring that code is being checked before it goes into production, vulnerabilities are being fixed, and the application you deploy is secure.

Project Phases

1. Run Sobelow, deps.audit, hex.audit for the first time.
2. Triaging Sobelow findings, documenting false positives.
3. Documenting and fixing true positive vulnerabilities.
4. Train developers on the correct procedures to handle security findings.
5. Run Sobelow, deps.audit, hex.audit on each new pull request.
6. Monitor for remote code execution at runtime with Exploit Guard.
7. Regularly audit the current state to ensure procedures are being followed.

[22] https://paraxial.io/blog/sobelow-guide

Milestone 4: Penetration Testing

When doing security work an external perspective is helpful. The previous milestones are for the goal of finding security issues with your Elixir applications and fixing them. Now it is time to have an external party test the security of your application and uncover any gaps.

There are debates about terminology here, people generally use the terms penetration test and web application security assessment interchangeably. The goal of the tester is to find security problems with the Elixir application, and show how a real attacker would exploit them. What should you look for in a penetration testing engagement for your Elixir application?

First a word on scope. The engagement described here will not cover someone attempting to break into your corporate network via social engineering, phishing, or sneaking into buildings. We are concerned with the security of software written in Elixir. That being said your Elixir application is likely part of a larger network of services. For example, it may be hosted on AWS, relying on non-Elixir services to process file uploads, and use an external service for storing those uploads. If the database sever for the Elixir application is misconfigured and someone on the open internet can exploit it, you want that finding to be in scope for testing, even though it is not strictly related to Elixir security. You may have split teams at your company, where some people handle cloud and server security, while a different team is responsible for application security. There is no generic guide that is correct for every situation, generally speaking you want to be searching for security issues that could be exploited and lead to a data breach.

The person testing your Elixir application should be familiar with web application security concepts. They should be able to explain what the acronyms RCE, XSS, CSRF, and SSRF mean, with examples of how to exploit each type of vulnerability. Prior experience with Elixir is ideal and Paraxial.io has seen demand for this.

Communicate with your prospective test provider that you are using Elixir, and ask if they have worked with it before. Elixir web applications often sit in a very privileged network position, where compromise by an attacker would be a disaster. For example, consider a bank using Phoenix for their online portal. An attacker able to compromise the web server means client funds will be stolen because that application has the ability to initiate transfers.

There is some debate about testing with source code of the application (white box) compared to only from the perspective of an external attacker (black box). I prefer white box testing for several reasons:

1. **Better Client Value** - In a black box assessment, the tester will have to spend hours brute forcing the application to uncover potential vectors for malicious input. In a white box test, the tester can quickly search for known vulnerable functions in

the code base, a simple yet highly effective technique. This is a better use of the time the client paid for.

2. **Faster Understanding** - Consider an authentication library dependency with a known security problem. With source code access, the tester can read the installed dependencies, find the library, and see if the issue exists. Related to the first point, less time is spent on recon and more focus is put on finding real issues.

3. **Developer Education and Context** - The tester finds a vulnerability. How should it be fixed? A tester with deep understanding of the application source code can provide guidance to the developers about how to best fix the problem and prevent it in the future.

Contact multiple firms and get quotes that match your goals. There is no standard price for a penetration test, higher skilled testers do tend to cost more. A word of caution about the low budget testers, you are unlikely to get any real security value out of the engagement, and will be paying with wasted time and a report that was generated by an automated scanner. If you are putting in the time to do a pentest doing your own research at the start will pay off when the report comes in.

Project Phases

1. Write down a one page project overview document for the pentest.
    i. What are your goals for the test?
    ii. What are your biggest areas of security concern?
    iii. What does your ideal test look like?
2. Reach out to pentest providers, get quotes that match your project plan.
3. Engage the testing company, fix reported vulnerabilities.
4. Evaluate the overall quality of service.
5. Repeat this process annually.

Milestone 5: DDoS and Bot Attacks

All web applications on the public internet today face a similar set of threats. Even if your Elixir application has zero reported findings from Sobelow, deps.audit, and has been tested by an external auditor, there are still relevant threats that need to be mitigated. Consider a SaaS company that makes a project management tool. Anyone from the public internet can create a new account, pay with a credit card, and start using the service. The application owners have followed all the milestones so far, and are deploying code that has been checked for security problems.

There are two broad classes of attack this application is still vulnerable to:

1. **Distributed denial of service (DDoS)** - Where an attacker sends massive amounts of network traffic to the server, causing it to crash and disrupt business operations.
2. **Targeted bot attack** - There are several different variants of this type of attack: carding, card cracking, credential stuffing. Essentially the bad guy writes a bot to abuse some functionality on the website.

DDoS is easier to understand. If the application is experiencing a huge amount of traffic and nobody can access it, that is probably a DDoS attack. If you have ever wondered why access to anti-DDoS services is so important for some websites, without that protection they effectively stop working because DDoS attacks are so common.

For anti-DDoS protection there are several specialized vendors (Cloudflare, Akamai, Fastly). These vendors put their own reverse proxy server in front of your origin server (where the Elixir application is running), and that proxy server is what blocks the attack. There are methods to bypass this protection, for example by leaking the true IP address of your origin server. Your annual pentest should include a test case to bypass anti-DDoS protection.

When it comes to stopping targeted bot attacks, the situation is much more dependent on what type of application you have. Consider the project management SaaS product example. They will inevitably have to deal with:

1. **Credential stuffing attacks** - An attacker obtains lists of email and password pairs from past data breaches, then uses a bot to run thousands of login attempts per hour against the site, compromising users who re-used their password. This is a very common attack that leads to user accounts being taken over by the attacker.
2. **Carding attacks** - An attacker has a list of several thousand stolen credit cards. He writes a bot to create thousands of new accounts, and does a fraudulent

transaction with each account. The business impacts can include the credit card processing vendor banning the company account, lost sales due to downtime, and the incident remediation costs.

3. **Automated email fraud** - The software has a feature where you can invite your co-worker to a project by sending them an email invite through the application. The attacker writes a bot to send out thousands of spam emails using this function, so that each email is from the SaaS company domain. These emails are reported as spam, and result in the company's email provider banning their account. This is a major business disruption, imagine if your Elixir application could no longer send any email.

The risk of bot attack varies depending on what type of application you have. Your pentest provider should be able to perform a bot risk assessment for you. There are several options for stopping these attacks, including:

1. IP based rate limiting.
2. CAPTCHA puzzles (hCaptcha, reCaptcha, etc).
3. Monitoring user behavior and logs.
4. Dedicated vendor solutions.

Each of these solutions has benefits and drawbacks. One common misconception is that putting up reCaptcha is sufficient to stop credential stuffing a credit card fraud attacks. reCaptcha has suffered from its own popularity, it is now so common there are companies dedicated to helping bots bypass it.[23] [24]

Project Phases

1. Setup anti-DDoS protection.
2. Perform a bot risk assessment on the Elixir application.
3. If there is a username/password login, ensure credential stuffing is blocked.
4. If there is a credit card payment option, ensure automated traffic is blocked.
5. Regularly audit the anti-bot configuration by simulating an attack.

---

[23] https://2captcha.com/

[24] https://anti-captcha.com/

Milestone 6: Logging and Monitoring

The previous milestone on DDoS and bot attacks should demonstrate the need for proper logging in your Elixir application. If you normally see 100 login attempts per hour and suddenly that spikes to 100,000 then you are under attack. There are numerous vendors[25] that handle ingesting data and presenting it back in a useful format. For the scope of this document we focus on:

1. HTTP requests
2. GraphQL queries (if applicable)
3. Specific user actions

HTTP requests can help answer questions such as "How many login attempts were done in the past hour", although there are limitations, for example most systems do not log the POST body because it contains password data. With the adoption of GraphQL, sometimes HTTP logs turn into "thousands of POST requests to */graphql* with no visibility into what each one means".

Ideally your monitoring is setup so you have detailed structured logs for specific user actions. For example:

1. User login attempt
2. User performing a credit card payment
3. User performing an expensive API call

When attacks happen logs are critical to understanding what the bad guy is doing. Many logging vendors have built in anomaly detection, meaning they can alert you to unusual patterns on sensitive user action.

Case Study: 23andMe

In October 2023 genetic testing company 23andMe made headlines[26] due to a post on BreachForums advertising stolen data. Based on publicly available reports, it seems only a small number of user accounts were compromised. These accounts were used to scrape data from a feature called DNA relatives. There is some debate online about if the company was hacked, as none of their servers were compromised, but this

---

[25] https://www.honeycomb.io/wp-content/uploads/2023/07/gartner-magic-quadrant-2023-large.png

[26] https://www.wired.com/story/23andme-credential-stuffing-data-stolen/

misses the forrest for the trees. Safeguarding customer data is a business critical task, and all that really matters is an attacker was able to steal this information.

If the above facts are accurate, the DNA relatives feature allowed one user account of 23andMe to access the generic information of thousands of additional accounts. Ideally this feature would not be implemented due to the security risk, but in the real world business requirements often prevail. Rate limiting the number of requests to the API for this feature would have delayed the attacker, yet this situation also requires monitoring accounts making unusually high number of requests to this endpoint.

Without monitoring the attacker will hit the rate limit, scale down the number of requests, and slowly leak out all the data. Even if there is an account ban for hitting the rate limit, the attacker can simply determine the ban threshold and stay under it. Monitoring for accounts making unusually high number of API requests would have helped detect this attack.

Project Phases

1. Start logging HTTP traffic.
2. If applicable, log GraphQL queries.
3. Log specific actions tied to user accounts.
    1. Account authentication
    2. Credit card transactions
    3. Expensive or sensitive API calls
4. Create alerts for anomalous activity.
5. Audit your configuration and alerts regularly via simulated attacks.

Milestone 7: External Vulnerability Reports

Establishing a policy for accepting and addressing vulnerability reports is a good idea. If one of your users tells you about a vulnerability in good faith, it is generally considered bad form today to sue them, rather the standard response is to thank them and fix the security problem. Bug bounty programs have increased in popularity, where companies offer financial incentives to researchers that find high quality flaws.

For most readers of this guide my recommendation is to establish a policy for accepting external vulnerability reports, and prioritize high quality submissions. You will receive many low quality reports, and triaging these takes time. When you start offering financial compensation for vulnerabilities the frequency of reports will go up, and the overhead of dealing with incoming reports[27] is not an efficient use of your team's time. Bug bounty programs are usually the wrong choice for companies with fewer than 100 employees, and ever for larger companies there are usually more pressing security issues that need to be dealt with.

Of all the milestones on this list, accepting external vulnerability reports has the lowest return on investment per engineering hour you put in. That is not to say there is zero value to having a vulnerability disclosure program, some useful security issues may come in. The reality is that most of your time spent on this project will involve triaging low quality reports, an inefficient use of expensive engineer hours.

Project Phases

1. Write a policy for accepting external vulnerability reports and publish it. Do not offer payment for security bugs initially. Limit the scope to Elixir applications because the probability of these crashing is low.
2. Operate the initial program for several months. Determine if it is providing security value.
3. If you have successfully completed all the milestones in this document, and are looking for your next security related project, consider a paid bug bounty program.

---

[27] https://security.stackexchange.com/questions/164811/what-are-some-options-for-a-small-company-on-a-budget-to-maintain-a-bug-bounty-p - A common story among smaller companies doing a paid bug bounty. Haggling over a few hundred dollars per bug is not how you should be spending your limited time.

Conclusion

Implementing all the guidance in this document is a significant project. The intent is not to overwhelm you, that is why the milestones are ranked in order of importance. You should be doing an asset inventory and penetration test before starting a bug bounty program. There will always be more security work to do, you have to prioritize and scope out projects into manageable pieces to make progress.

You do not have to travel this path alone. Paraxial.io[28] is a security platform for Elixir that helps with several of these milestones. The company also offers security consulting[29], penetration testing, and developer security training focused on Elixir. If you have any problems related to Elixir security please reach out to michael@paraxial.io for assistance. Thank you for reading.

[28] https://paraxial.io/

[29] https://paraxial.io/consulting