Least Authority
PRIVACY MATTERS

Staking Wallet

Security Audit Report

Blox

Final Report Version: 17 March 2021

Table of Contents

Overview

Background

Blox has requested that Least Authority perform a security audit of the Blox Staking Wallet. Blox is an open-source, fully non-custodial staking platform for Ethereum 2.0. The platform aims to serve as an easy and accessible way to stake Ether and earn rewards on Ethereum 2.0, while ensuring participants retain complete control over their private keys.

Project Dates

Staking Wallet

- **October 21- November 3**: Phase 1 Code review *(Completed)*
- **November 6:** Delivery of Phase 1 Initial Audit Report *(Completed)*
- **November 25:** Delivery of Phase 1 Updated Initial Audit Report *(Completed)*
- **March 15 - 16:** Verification *(Completed)*
- **March 17:** Delivery of Final Audit Report *(Completed)*

Review Team

- Ramakrishnan Muthukrishnan, Security Researcher and Engineer
- Jehad Baeth, Security Researcher and Engineer
- Chris Wood, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Staking Wallet followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Staking Wallet
  - https://github.com/bloxapp/eth2-key-manager
  - https://github.com/bloxapp/key-vault

Specifically, we examined the Git revisions for our initial review:

`44f1a35393affb92d019ddef163b7d32d4f94859` (eth2-key-manager)

`28ba3813d66bebf38e7b15070025af2d88eac487` (key-vault)

For the verification, we examined the Git revision:

693722e9252ee5337dd353e85d45eb70d19095ed (eth2-key-manager)

36a15a3e226eca298c2b28ffde9765bdb81cbbf8 (key-vault)

For the review, these repositories were cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/eth2-key-manager

https://github.com/LeastAuthority/key-vault

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation is available to the review team:

- Staking Wallet
  - README:
    - https://github.com/bloxapp/key-vault/blob/master/README.md
    - https://github.com/bloxapp/eth2-key-manager/blob/master/README.md
  - Blox Documentation: https://www.bloxstaking.com/docs-fundamentals/

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation, including the adherence to the Ethereum 2.0 specification, use of signatures, slashing protection and restore functions;
- Adversarial actions and other attacks on the smart contracts and staking wallet;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Alignment of incentive mechanisms to help prevent unwanted or unexpected behavior;
- Malicious attacks and security exploits that would impact the contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart contracts and wallet code, as well as secure interaction between the related and network components;
- Proper management of encryption and signing keys;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Review Scope

During our security audit of the Blox Staking Wallet, we found the scope to be comprehensive, with sufficient coverage of most security critical components in the code base. However, some dependencies are currently being developed and have not reached stable versions, including `go-eth2-types` and `go-eth2-wallet-encryptor-keystorev4`. The use of dependencies that are in development presents a risk of consequential code changes for the Blox team and creates uncertainty for the reviewers of the code. As a result, changes to dependency code should be carefully and continuously monitored for potential security vulnerabilities that may impact the project at large and follow up reviews of these dependency code changes should be conducted to ensure full coverage.

Use of Dependencies

The Blox Staking Wallet utilizes multiple dependencies to handle security critical tasks, including Hashicorp Vault. Hashicorp Vault requires extensive knowledge of the tool in order to verify the correctness of its implementation and usage. Given that dependency code is out of scope for this review, careful dependency management is critical in the effort to avoid security vulnerabilities. We recommend that the Blox team utilize well known and audited dependencies, regularly update dependencies to the latest release fixing bugs and issues, and pin dependency versions to releases compatible with Blox. (Suggestion 2).

Static analysis tools, including gosec and nancy show some use of indirect dependencies with known security vulnerabilities. We recommend the Blox team assess these vulnerabilities and take appropriate action, such as upgrading, to manage the risk introduced by these particular dependencies.

Code Quality + Documentation

The eth2-key-manager repository is nicely organized into packages. The key-vault repository is a secrets management plugin into HashiCorp Vault. The two projects were easy to build. This allowed the auditors to easily write custom tests to verify the implementation. Code comments were sufficient, outlining the intended purpose and functionality of the implementation, and facilitating an easier comprehension and understanding of the code. However, the current commit messages in the repository are potentially non-descriptive and we recommend that they be improved by adding descriptions that are unique to each commit, thus clearly defining the purpose of the code changes.

The project documentation is helpful and provides an easy to follow overview of the system. The README files in individual sub-directory modules provide explanations for each module. In addition, the code contains both unit and end-to-end testing, however, we recommend increasing test coverage in some areas to maintain a high ratio of end-to-end test coverage (Suggestion 3).

Throughout the code, our team found multiple instances of type assertions that do not check whether the operation succeeded. This makes the code panic at runtime and exit out if the assertions did not succeed and makes the code brittle as a result (Suggestion 4).

For example:

```
err = options.storage.(core.Storage).SaveWallet(wallet)
```

In this example, the value in `options.storage` is interpreted as type `core.Storage` followed by a function that operates on that type is invoked. However, this should be done in two steps, first a type cast and only if that succeeds, the rest should follow. If not, flag an error and exit. We also encountered the inconsistency in logging and error handling. We recommend consistently using the same log-handling and error-handling libraries throughout the codebase to make it easier to track bugs in the codebase. (Suggestion 1).

System Design

The Blox Staking Wallet system depends on Hashicorp's Vault system for management of secrets. Usage of Vault for secrets management in a Blockchain system is unique and Blox itself does not centralize the keys in one place. This design solution for non-custodial staking platforms minimizes risks if the Blox system is compromised as the keys are stored and managed in encrypted format on a docker image running on the user's choice of cloud storage service like Amazon Web Services or Google Cloud. Similarly, the Blox system relies on other free, open and well-established software libraries which is a good practice when building security-critical functionality.

The Blox system implements Ethereum 2.0's slashing protection recommendations, which provide protection against both Proposer slashing and Attester slashing. This is done by keeping a history of proposals and attestation epoch records, in addition to analyzing incoming requests in order to detect and prevent slashable events. As a result, this helps to further protect end users utilizing Blox for staking from being penalized if they engage in slashable activities and thus making the implementation more secure.

Specific Issues & Suggestions

We list the issues found in the code, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Remove KV Root Tokens Following Setup Process in Production Environments	Resolved
Issue B: Save Tokens Securely in a Token Helper	Resolved
Suggestion 1: Improve Consistency in Logging and Error Handling	Resolved
Suggestion 2: Improve Dependency Management	Resolved
Suggestion 3: Maintain High Ratio of End to End Test Coverage	Resolved
Suggestion 4: Correct Unsafe Type Assertions in the Code	Resolved
Suggestion 5: Verify Hashes of Docker Hub Images	Resolved
Suggestion 6: Build Docker Images Reproducibly	Unresolved

Issue A: Remove KV Root Tokens Following Setup Process in Production Environments

Location

key-vault/blob/v0.1.11-rc/config/vault-init.sh

Synopsis

Root tokens should be used only during the initial setup process. Currently, the root tokens are not revoked once the setup process is complete.

Impact

The likelihood of an attack is unknown. However, in the event that the preconditions are met, the impact of the attack would be significant.

Preconditions

An attacker successfully acquires access to one of the root tokens.

Technical Details

If exposed, root tokens allow an attacker to have full control over the vault. This exposes all secrets stored in the vault. HashiCorp Vault documentation also recommends revoking the root token once the vault system has transitioned from development phase to production.

Remediation
As per the HashiCorp Vault documentation, setup up authentication methods and policies necessary to allow administrators and users to acquire more limited tokens, then revoke the root token once the setup process is done.

Status
The Blox team has revoked the root token upon completion of the HashiCorp Vault setup, as suggested.

Verification
Resolved.

Issue B: Save Tokens Securely in a Token Helper

Location
key-vault/blob/v0.1.11-rc/config/vault-init.sh

Synopsis
In the current implementation of Blox, tokens are stored in the file system in an unencrypted state, which leaves them exposed for attacks.

Impact
The likelihood of an attack is unknown. However, in the event that the preconditions are met, the impact of the attack would be significant.

Preconditions
An attacker successfully acquires access to the machine running the vault server file system.

Technical Details
If an attacker gains access to the file system, they can access the unencrypted tokens which may lead to potential exposure of secrets stored in the vault.

Remediation
Implement/utilize a token helper to store, retrieve and manage vault's tokens. HashiCorp Vault documentation provides details on implementing a proper token helper.

Status
The Blox team has implemented the suggested remediation so that the root tokens are no longer stored in the file system. Instead, root tokens are passed as variables to subsequent subprocesses and then removed upon completion of the vault setup.

Verification
Resolved.

Suggestions

Suggestion 1: Improve Consistency in Logging and Error Handling

Synopsis

The Blox Staking Wallet depends upon multiple logging mechanisms (Logrus, Go's `log`, Vault's logging, etc.) from its different package dependencies to generate logs.

For example, in key-vault/blob/master/keymanager/errors.go (L35-45), the Go `log` library is used:

```
func (e *HTTPRequestError) String() string {
	if e == nil {
		return ""
	}

	data, err := json.Marshal(e)
	if err != nil {
		log.Fatal(err)
	}
	return string(data)
}
```

Whereas in key-vault/blob/master/keymanager/keymanager.go (L34-43) the Logrus library is used:

```
type KeyManager struct {
	remoteAddress string
	accessToken   string
	originPubKey  string
	pubKey        [48]byte
	network       string
	httpClient    *http.Client

	log *logrus.Entry
}
```

Similarly, eth2-key-manager/key_vault.go uses the built in Go error mechanisms, whereas other parts of the same package uses the `pkg/errors` library to handle errors.

Mitigation
While these instances may have different policies for logging, we suggest consistent use of the same log-handling and error-handling libraries throughout the codebase. Consistency will aid in enforcing clearer logging and error-handling policies, making it easier to track and fix bugs in the codebase.

Status
The Blox team has implemented more consistent and standardized logging mechanisms across the in-scope repositories. In particular, the Blox team is now making use of Logrus for generating functional logs in the `key-vault` and `eth2-key-manager` repositories referenced above.

Verification
Resolved.

Suggestion 2: Improve Dependency Management

Synopsis
The Blox Staking Wallet depends on several outdated security critical dependencies (e.g. Hashicorp Vault and go-eth2-wallet-encryptor-keystorev4), which may lack security critical updates. For example, the current latest version of Hashicorp Vault is missing tags for the latest `vault/sdk` and `vault/api` dependencies used by Blox.

Mitigation
We suggest the following mitigation strategies:

- Manually assess and regularly monitor and maintain security critical dependencies. Use commit hashes instead of release number tags to point to the latest releases, as needed.
- Update dependencies when security issues and bugs have been detected.
- Pin updated dependency versions to releases compatible with the Blox wallet (in order to avoid breaking the code base upon automatic dependency upgrades).
- Potentially consider vendoring all dependencies into the existing code base(s) in order to help prevent dependency-hijacking attacks.
- Incorporate an automated dependency security check into the CI workflow such as, gosec and nancy.

Status
The Blox team has revised the list of dependencies by utilizing commit hashes instead of version release numbers. In addition, the implementation now utilizes gosec in the Continuous Integration (CI) workflow.

Verification
Resolved.

Suggestion 3: Maintain High Ratio of End-to-End Test Coverage

Location
No test coverage:

- key-vault/keymanager/keymanager_v2.go
- key-vault/keymanager/opts.go
- eth2-key-manager/core/attestation_data.go

Low test coverage:

- key-vault/keymanager
- eth2-key-manager/core/mnemonic.go

Tests do no exercise failure conditions:

- keymanager/keymanager.go

Synopsis

We recommend unit tests for all the exposed functions for a package. While a number of modules have good coverage, many of them would benefit from increased coverage. In particular, projects that handle funds should have a high ratio of test coverage, as it is critical to understand whether all edge cases are sufficiently covered and the system is performing as expected.

Mitigation

Increase and maintain a high ratio of end-to-end test coverage throughout the system.

Status

The Blox team has increased the test coverage ratio in the aforementioned modules in both the `eth2-key-manager` and `key-vault` such that it provides sufficient coverage of the code.

Verification

Resolved.

Suggestion 4: Correct Unsafe Type Assertions in the Code

Location

eth2-key-manager/key_vault.go: L99:

```
err = options.storage.(core.Storage).SaveWallet(wallet)
```

Instances of similar dynamic-type assertions are also used in other areas of the code base.

Synopsis

Some value present in `options.storage` is of type `core.Storage` and then calling a method on that type. If the value is not of that type, the expression would panic at runtime and exit.

Mitigation

The Go specification provides an alternate and more secure approach to type assertions. The above expression can be rewritten as:

```
If v, ok := options.storage.(core.Storage); !ok {
               // gracefully handle the error
      } else {
              // go ahead and call the SaveWallet() method on v
      }
```

Status
The Blox team has implemented an update that adheres to the Golang specification for type assertions.

Verification
Resolved.

Suggestion 5: Verify Hashes of Docker Hub Images

Location
key-vault/blob/v0.1.11-rc/Makefile

Synopsis
The key-vault project is to be run as part of a Docker container as per the instructions in the README.md. An extra step of verification of the downloaded or built image's integrity check would help the user verify the images.

Mitigation
We recommend automating the process of the verification of hashes in the `Makefile` and add a corresponding note in the `README.md`.

Status
The Blox team has added a Github Workflow Action to automate verification of image digests upon building and pulling images from Docker Hub. In addition, they included the latest key-vault image digest to the Github `README.md`, enabling users to manually verify images pulled from Docker Hub.

Verification
Resolved.

Suggestion 6: Build Docker Hub Images Reproducibly

Synopsis
Users of the key-vault Docker image (hosted on Docker Hub by the Bloxstaking organization) depend upon the integrity of this image for security critical functionality. There is no easy way, however, for users to verify that the image and its software performs the operations as described or that it has not otherwise been tampered with.

Mitigation
We recommend taking additional steps to ensure that Docker images can be built reproducibly, such that the hash digest of the Docker image built locally by a user will match that of the image provided by the Bloxstaking team on Docker Hub. This can be achieved by removing indeterminate elements from build inputs (e.g. by pinning the hash of the base of the image and the version of each of the packages installed as part of the Dockerfile, setting/overriding the value of locales and file timestamps to known values, etc.).

Status
The Blox team has acknowledged this suggestion and have indicated their intentions to address this as part of their longer term development roadmap. We recognize the complex and time-consuming nature of the effort to set up a reproducible build process and encourage the Blox team to continue considering how this can be achieved within a reasonable timeframe.

Verification

Unresolved.

Recommendations

We recommend that the unresolved *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

The Blox team has also increased end-to-end integration tests to ensure sufficient coverage of potential edge cases.

We commend the Blox team for maintaining and updating dependencies, ensuring that the most recent versions are utilized and pinned, in order to protect against the potential for bugs and security vulnerabilities. In addition, we suggest that follow up reviews be conducted once the in progress dependencies have been developed to completion.

Finally, the Blox team's adherence to programming best practices is demonstrated through the maintenance of thorough project documentation and prioritization of security in the implementation of this project.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.