Least Authority
PRIVACY MATTERS

Staking Wallet 2nd Review
Security Audit Report

Blox

Final Audit Report: 30 August 2022

Table of Contents

Overview

Background

Blox has requested that Least Authority perform a second review 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

- **July 25 - August 5:** Initial Code Review *(Completed)*
- **August 10:** Delivery of Initial Audit Report *(Completed)*
- **August 26:** Verification Review *(Completed)*
- **August 30:** Delivery of Final Audit Report *(Completed)*

Review Team

- Shareef Maher Dweikat, Security Research and Engineer
- Alejandro Flores, Security Researcher and Engineer
- Giorgi Jvaridze, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer

Coverage

Target Code and Revision

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

The following code repositories are considered in-scope for the review:
- Eth2 Key Manager:
  https://github.com/bloxapp/eth2-key-manager
- Key Vault:
  https://github.com/bloxapp/key-vault/pull/90

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

Eth2 Key Manager: `533315a3802f3c4726325a0fd766a99f63fdd730`

Key Vault: `92389f2023f955b6880136113da16d921391adb1`

For the verification, we examined the Git revisions:

Eth2 Key Manager: `c9c8337629521aa0a37a373b4fab1ec0dce297ff`

Key Vault: `d9ee083b350a9dc1c1b37971ac4ae18aa9b2dc17`

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

Eth2 Key Manager:
https://github.com/LeastAuthority/bloxapp-eth2-key-manager

Key Vault:
https://github.com/LeastAuthority/bloxapp-key-vault

In addition, we referenced the PRs on the following forked repositories:

Eth2 Key Manager:
https://github.com/LeastAuthority/bloxapp-eth2-key-manager/pull/1

Key Vault:
https://github.com/LeastAuthority/bloxapp-key-vault/pull/1

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

In addition, any dependency and third-party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- README:
  https://github.com/bloxapp/key-vault/blob/master/README.md
- README:
  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 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 impact funds, such as the draining or 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 intended use of the contracts or disrupt their execution;
- Vulnerabilities in the smart contracts and wallet code as well as the 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

In a previous audit published on March 17, 2021, our team reviewed the Blox Staking Wallet implementation. Our team identified issues and suggestions in that review. For this review, the scope of the audit was limited to the staking features of the Blox wallet implemented since our last review.

During our second review, we checked that the issues and suggestions from the previous report have been addressed in the current implementation. However, we found that several of the issues and suggestions continue to be unresolved in the current branch of development. We recommend that the Key Vault PR that was used in the verification stage of the previous audit be merged to the main branch of the project repository.

During our investigation, we examined the design and implementation of the changes that were in-scope for this review and compared them to the findings of the previous report. We found that there is an ongoing pattern relating to the insecure use of dependencies as well as inappropriate error handling (Issue A; Issue B).

System Design

We found that security has been taken into consideration in the design of the Blox staking features as demonstrated by the decoupling of the Key Vault from the Eth2 Key Manager, the implementation of appropriate signing and slashing protections, adherence to the EIP-2335 standard regarding control codes removal for correct password normalization, as well as the adherence to the EIP-2334 standard for proper path notation.

We examined the key derivation functions and parameters used for encryption. The Blox staking wallet uses the key derivation algorithm PBKDF2 by default to derive encryption keys from user-selected passwords, which is not a sufficiently secure algorithm for the purpose of encryption key derivation. We recommend that a memory-hard key derivation algorithm be used by default. Furthermore, we suggest that an additional memory-hard key derivation algorithm be made available (Issue C).

API calls to the Key Vault do not require that a valid certificate be used, which could result in connections that are vulnerable to man-in-the-middle attacks (MiTM). We recommend that connections to the API be secured with `InsecureSkipVerify: false` (Issue D).

Code Quality

In our review of the Eth2 Key Manager and Key Vault codebases, we found the code to be well organized and written in accordance with Golang best practices. However, our team identified several areas of improvement that would increase the overall quality and security of the code. We found that errors are not handled appropriately, which could cause the system to behave unexpectedly. We recommend that errors be propagated to the user and provide useful information on the cause of the error (Issue B). We also recommend using `defer unlock` to prevent errors causing unexpected behavior (Suggestion 2).

In the Eth2 Key Manager implementation, we found several instances where the type of variables can be changed to improve performance. We recommend that an alternative type of storage variable be used (Suggestion 1). In addition, as a best practice, we recommend using the logging tool `logrus` consistently throughout the implementation instead of relying on the built-in Go `log` library in some instances (Suggestion 4).

Tests

Although there is sufficient end-to-end and unit test coverage for the Eth2 Key Manager, the test coverage of the Key Vault could be improved. The Eth2 Key Manager implementation also follows the standard of adding the `_test` suffix to test files, while the Key Vault implementation does not follow this standard. Instead, the test files are stored in a `test` folder without using the `_test` suffix. This suffix is used by Golang CLI tools to automatically identify and execute tests, in addition to detecting code coverage.

Documentation

The project documentation provided for this review included comprehensive user documentation and informative READMEs that offered a helpful description of each of the components as well as their functionality and interactions between them.

Code Comments

We found code comments to be sufficient. However, we identified several instances where code comments can be improved. In particular, comments can be more specific when providing the expected results of a function call. In addition, some comments are non-descriptive or provide little to no context to the code. Although we did not find any instances of incorrect or out-of-date comments, there are instances of commented-out code, which can be confusing to reviewers. We recommend that commented-out and unused code be removed from the codebase (Suggestion 3).

Scope

The scope of this review covered only the staking features upgrade, which presented a unique challenge of scope. Our team was required to understand both the previous implementation and the latest developments in the codebase.

Dependencies

Our team examined the dependencies used in the implementation and found dependencies that could make the wallet vulnerable to side-channel attacks (SCA). We recommend that up-to-date, well-maintained dependencies be used (Issue A).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, 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: Usage of Vulnerable Dependencies	Partially Resolved
Issue B: Error Is Not Returned	Resolved
Issue C: PBKDF2 Vulnerable to Dictionary Attacks	Resolved
Issue D: API Connection Certificates Not Checked	Unresolved
Suggestion 1: Change Mutexes to RWMutexes for Read-Only Functions	Resolved
Suggestion 2: Use the Defer Unlock Function	Resolved

Suggestion 3: Delete Commented Code	Resolved
Suggestion 4: Switch to logrus in sign_slot_and_epoch_test.go	Resolved

Issue A: Usage of Vulnerable Dependencies

Location

`/go.mod`

Synopsis

Analyzing the code using the `go list -json -deps | nancy sleuth` command shows three vulnerable dependencies used in the Key Vault.

Our team also identified multiple outdated dependencies by running the command `go list -u -m -json all`.

Impact

Using dependencies or packages with known vulnerabilities exposes the Key Vault to attacks that could result in the exfiltration of sensitive data.

Preconditions

A vulnerable dependency or a new vulnerable update to an already existing dependency gets installed in the project.

Remediation

We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the Key Vault, which includes:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including Automated Dependency auditing reports in the project's CI/CD workflow.

Status

The Blox team has addressed all the vulnerable dependencies except for `go-ethereum`. Running the command `go list -u -m -json all` shows the remaining outdated dependencies.

Verification

Partially Resolved.

Issue B: Error Is Not Returned

Location

`/key-vault/e2ereturn nil, nil/e2e_api_calls.go#L101`

Synopsis

In the function `Sign` on line 99, the following expression is evaluated:

```
req, err := http.NewRequest(http.MethodPost, targetURL, bytes.NewBuffer(body))
```

The variable `err` handles any errors this `NewRequest` could contain. After that, the next line evaluates whether this `err` variable has an error or not:

```
if err != nil
```

If this is `true`, then there is an error. Therefore, it should be returned in the next line. The issue is that the value that is returned is `nil`:

```
return nil, nil
```

This flow is incorrect, as the expected result should return the error value for further error handling.

Impact

This could generate errors for this module of the application, and it might result in a low to medium impact. If this error is not handled, it could cause performance issues or even break the flow of the application.

Preconditions

- Sending a corrupt HTTP Post Body when trying the `Signing` function.
- Having a targetURL that is not available.

Remediation

We recommend following this simple remediation by changing the return statement on line 101 to:

```
return nil, err
```

Status

The Blox team has updated the return statement as suggested.

Verification

Resolved.

Issue C: PBKDF2 Vulnerable to Dictionary Attacks

Location

`key-manager/encryptor/keystorev4/encryptor.go`

Synopsis

Although the application supports both Scrypt and PBKDF2 key derivation functions, PBKDF2 is set as the default. This key derivation function is known to be insufficiently secure, as it is vulnerable to dictionary attacks. As a result, we recommend a more modern and secure password hashing function and suggest that the Blox team consider using Argon2 as a key derivation function. Argon2 is a GPU and memory-hardened password hashing algorithm that is well-supported in Go.

Impact

A PBKDF2 derived encryption key that is compromised would lead to the loss of user funds.

Preconditions

The user uses the default key derivation function to derive an encryption key from a user-selected password.

Mitigation

We recommend removing the default PBKDF2 and adding the Argon2id encryption key derivation algorithm instead.

Status

The Blox team has updated the default to use Scrypt instead of PBKDF2, which we consider sufficiently more secure than PBKDF2.

Verification

Resolved.

Issue D: API Connection Certificates Not Checked

Location

`/e2e/e2e_api_calls.go`

Synopsis

The httpClient variable used to create requests to the Key Vault explicitly allows a TLS connection to be insecure through the use of the flag:

```
TLSClientConfig: &tls.Config{

      InsecureSkipVerify: true,

}
```

Impact

Having an insecure communication channel to the vault could cause attacks such as eavesdropping (MiTM) or even result in a certificate getting replaced on the server-side (vault server) and trusting this certificate by default.

Preconditions

A vault is set up with a self-signed certificate or without a TLS connection at all.

Feasibility

An attacker could examine the connection to the vault and determine easily that an insecure connection can be leveraged for malicious purposes.

Mitigation

A user could easily establish a secure TLS connection with a `Let's Encrypt` certificate in the vault instance and set it to reject insecure or unverified TLS connections.

Remediation

We suggest implementing changes in the code that encourage users to set a secure TLS connection in the vault server by changing the `InsecureSkipVerify`flag to `false`, in the event that it is not the default setting.

Status

The Blox team acknowledged the issue and stated that since they address Key Vault instances by IP, and incoming requests to Key-Vault do not contain sensitive information, this will be postponed to a future post-audit release. As such, the issue remains unresolved at the time of verification.

Verification

Unresolved.

Suggestions

Suggestion 1: Change Mutexes to RWMutexes for Read-Only Functions

Location

/eth2-key-manager/stores/inmemory/slashing_store.go#L19

/eth2-key-manager/stores/inmemory/slashing_store.go#L35

Synopsis

The following functions only read information from the `store`. Therefore, an `RWMutex` could be used in this situation for better performance:

```
// RetrieveHighestAttestation retrieves highest attestation

func (store *InMemStore) RetrieveHighestAttestation(pubKey []byte)
*eth.AttestationData {

      store.highestAttestationLock.Lock()

      val := store.highestAttestation[hex.EncodeToString(pubKey)]

      store.highestAttestationLock.Unlock()

      return val

}

// RetrieveHighestProposal returns highest proposal

func (store *InMemStore) RetrieveHighestProposal(pubKey []byte)
*eth.BeaconBlock {

      store.highestProposalLock.Lock()

      val := store.highestProposal[hex.EncodeToString(pubKey)]

      store.highestProposalLock.Unlock()

      return val

}
```

Mitigation

On the stores/inmemory/store.go file, in the InMemStore struct definition, we suggest changing both highestAttestationLock and highestAttestationLock to RWMutexes to improve performance.

Status

The Blox team has updated highestAttestationLock and highestAttestationLock as suggested.

Verification

Resolved.

Suggestion 2: Use the Defer Unlock Function

Location

/key-vault//backend/path_signs.go#L146

Synopsis

In the function lock:

```
func (b *backend) lock(pubKeyBytes []byte, cb func() error) error {

      b.signMapLock.Lock()

      pubKey := hex.EncodeToString(pubKeyBytes)

      if _, ok := b.signLock[pubKey]; !ok {

            b.signLock[pubKey] = &sync.Mutex{}

      }

      lock := b.signLock[pubKey]

      b.signMapLock.Unlock()

      lock.Lock()

      err := cb()

      lock.Unlock()

      return err

}
```

the signMapLock Mutex is locked on line 147 but the expression if _, ok := b.signLock[pubKey]; on line 149 could fail or result in unexpected behavior.

Mitigation

We recommend implementing the defer UnLock function for cases where this line does not have the expected result so it can safely UnLock the Mutex as a default error handling.

We suggest using the `defer Unlock` function to handle unexpected errors in this part of the code.

Status
The Blox team has implemented the suggested mitigation.

Verification
Resolved.

Suggestion 3: Delete Commented Code

Location
`/ethereumvalidatoraccountsv2/keymanager.go#L41`

Synopsis
The code from line 41 to line 64 is commented out, and is, therefore, not being used.

Mitigation
We recommend deleting this code before merging it with the master code in accordance with the best practice of not having commented-out code in production releases.

Status
The Blox team has removed the commented-out code from the codebase.

Verification
Resolved.

Suggestion 4: Switch to logrus in sign_slot_and_epoch_test.go

Location
`/eth2-key-manager/signer/sign_slot_and_epoch_test.go#L58`

Synopsis
As the rest of the project is using the `logrus` library for logging, it is considered best practice to use it in `sign_slot_and_epoch_test.go` instead of the standard logging library.

Mitigation
We recommend changing the code to use `logrus` for logging.

Status
The Blox team switched from `log.Fatal` to `panic`.

Verification
Resolved.

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 multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, and zero-knowledge protocols. 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. We are an international team that believes 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 include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website 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. As 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 and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and 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 before having 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 and comprehensive 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 Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, 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 a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.