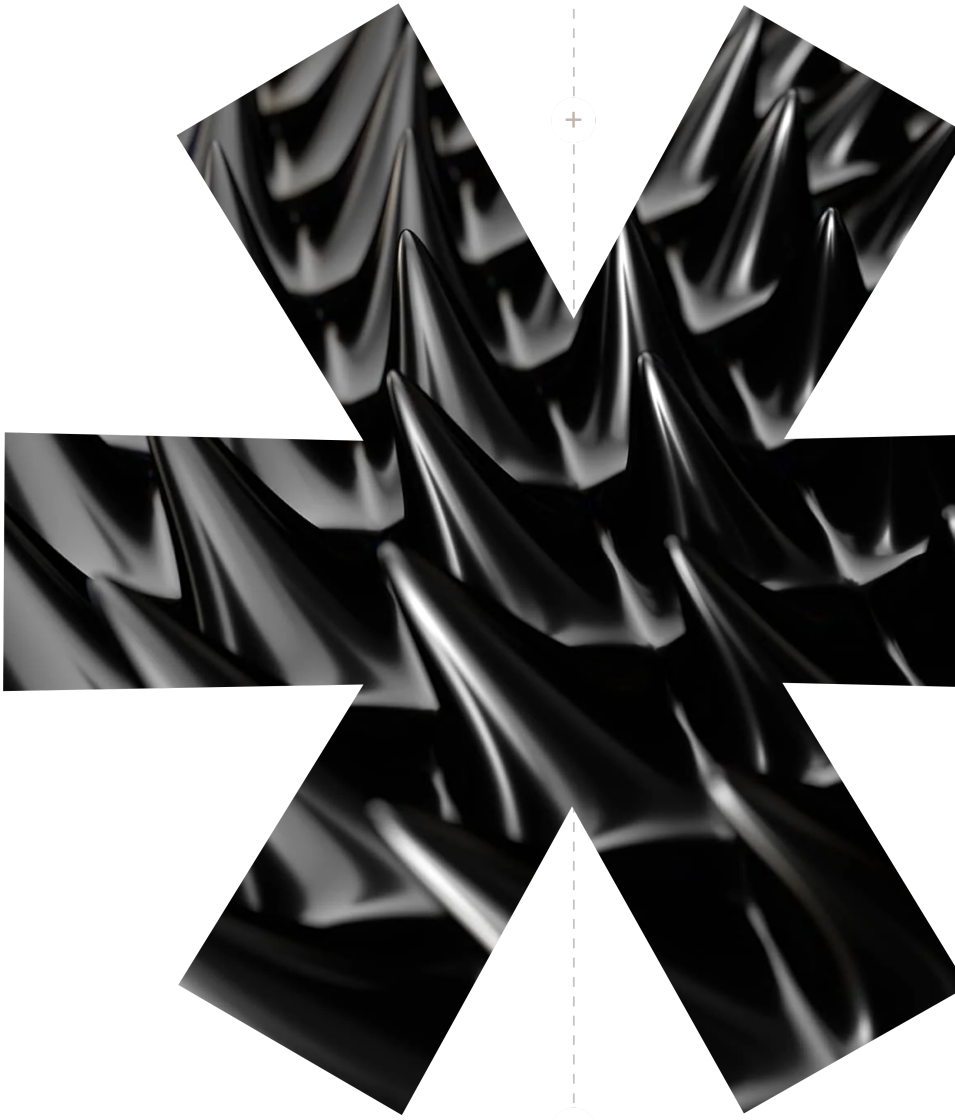
thesisdefense

Security Audit Report

POKT Network

ring-go

Initial Report // June 6, 2024
Final Report // July 5, 2024



Team Members

Jehad Baeth // Senior Security Auditor
PR // Applied Cryptographer and Security Auditor
Bashir Abu-Amr // Head of Delivery

Table of Contents

About Thesis Defense

Thesis Defense serves as the auditing services arm within Thesis, Inc., the venture studio behind tBTC, Fold, Taho, Etcher, and Mezo. Our team of security auditors have carried out hundreds of security audits for decentralized systems across a number of technologies including smart contracts, wallets and browser extensions, bridges, node implementations, cryptographic protocols, and dApps. We offer our services within a variety of ecosystems including Bitcoin, Ethereum + EVMs, Stacks, Cosmos / Cosmos SDK, NEAR and more.

Thesis Defense will employ the Thesis Defense Audit Approach and Audit Process to the in scope service. In the event that certain processes and methodologies are not applicable to the in scope services, we will indicate as such in individual audit or design review SOWs. In addition, Thesis Defense provides clear guidance on successful Security Audit Preparation.

`Section_1.0`

Scope

Technical Scope

- **Repository:** https://github.com/pokt-network/ring-go
- **Audit Commit:** e408436684eeec06ebd39f8453d834a7094ecf10
- **Verification Commit:** 1a8bebebd5223e022c3914341b83ac4928483a9d

Documentation and References:

1. C. Cremers and D. Jackson, “Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman.” 2019. [Online]. Available: https://eprint.iacr.org/2019/526
2. B. Noether Sarang and Goodell, “Triptych: Logarithmic-Sized Linkable Ring Signatures with Applications,” in Data Privacy Management, Cryptocurrencies and Blockchain Technology, G. and H.-J. J. Garcia-Alfaro Joaquin and Navarro-Arribas, Ed., Cham: Springer International Publishing, 2020, pp. 337–354.
3. L. Herskind, P. Katsikouli, and N. Dragoni, “Privacy and Cryptocurrencies—A Systematic Literature Review,” IEEE Access, vol. 8, pp. 54044–54059, 2020, doi: 10.1109/ACCESS.2020.2980950.
4. A. Jivanyan, “Lelantus: A New Design for Anonymous and Confidential Cryptocurrencies.” 2019. [Online]. Available: https://eprint.iacr.org/2019/373
5. J. Yu, M. H. A. Au, and P. Esteves-Verissimo, “Re-thinking untraceability in the CryptoNote-style blockchain.” 2019. [Online]. Available: https://eprint.iacr.org/2019/186
6. C. Cremers, J. Loss, and B. Wagner, “A Holistic Security Analysis of Monero Transactions.” 2023. [Online]. Available: https://eprint.iacr.org/2023/321
7. jonasnick, “Exploiting Low Order Generators in One-Time Ring Signatures,” https://jonasnick.github.io/blog/2017/05/23/exploiting-low-order-generators-in-one-time-ring-signatures/.
8. Ian Miers, “Satoshi Has No Clothes: Failures in On-Chain Privacy,” https://slideslive.com/38911785/satoshi-has-no-clothes-failures-in-onchain-privacy?locale=en.
9. Nathan Borggren, “Probing the Attacks on the Privacy of the Monero 3 Blockchain,” https://monerofund.org/pdf/Borggren-Sept-2023-Probing-the-Attacks-on-the-Privacy-of-the-Monero-Blockchain.pdf.
10. “Blockchain Privacy: Equal Parts Theory and Practice,” https://zfnd.org/blockchain-privacy-equal-parts-theory-and-practice/#decoys.

Section_2.0

Executive Summary

Schedule

This security audit was conducted from June 3, 2024 to June 6, 2024 by 2 security auditors for a total of 7 person-days.

Overview

We performed a security audit of the POKT Network's ring-go implementation. ring-go is a multilayered linkable spontaneous anonymous group signature scheme implemented in Go. It is based off of RingCT supporting ring signatures over both ed25519 and secp256k1 curves.

Threat Model

We analyzed potential threats to the security properties of the Ring Signature/MLSAGs implementation. We considered scenarios where an attacker may attempt to compromise any of the following properties:

- **Anonymity:** An attacker might try to identify the signer by analyzing the signature and its relationship with other signatures in the ring.
- **Unforgeability:** A malicious actor might attempt to forge a signature by impersonating the signer, either by creating a fake public key or by compromising the signing process. They could try to exploit weaknesses in the key generation algorithm, sign multiple messages with the same private key, or use advanced cryptographic attacks to break the underlying signatures.
- **Spontaneity**: An attacker could attempt to disrupt the spontaneity of the signature creation process by introducing coordination or setup requirements which affects the operational integrity of the system. This type of attack can be likened to a denial-of-service (DoS) attack, where the attacker exploits features of the protocol to prevent legitimate usage. Specifically, an attacker could create a situation where the system becomes unavailable or unusable for certain users, thereby disrupting the overall operation of the system.

Review of the Design and Implementation

We conducted an investigation for security issues in the design and implementation of POKT's ring-go.

Investigating the Possibility of Small-order Subgroup Attacks

TL;DR: The linkability of signatures is at risk with ed25519, but not with secp256k1.

The ring-go implementation utilizes two ECC curves: secp256k1 and ed25519. The security audit has identified a potential lead concerning the possible risk of Small-order subgroup attacks, particularly with the use of the ed25519 curve. This attack could potentially compromise the anonymity property of ring signatures if implemented keys belong to small subgroups within the main elliptic curve group.

The secp256k1 curve, which is used in the Ring Signature implementation, has a prime order and thus no small subgroup attacks are possible. This provides a reassuring security measure as it mitigates the risks associated with this type of attack for keys generated using this curve.

However, the ed25519 curve utilized in the implementation has an order of 8q for a prime q. Consequently, there is potential for subgroups of size 8 to exist within this curve group. This could be exploited as the basis for Small-order subgroup attacks. The attack's feasibility hinges on the attacker becoming a member of the ring used for the signature and employing a key that belongs to one of these small subgroups. If successful, this attack could lead to the disclosure of the attacker's identity, breaking the anonymity property of the ring signatures. For reference, the Monero team disclosed a similar bug in CryptoNote based cryptocurrencies as explain in this blogpost.

To mitigate such risks with the use of the ed25519 curve, it is suggested to follow a similar approach as Monero by checking that the key image I is on the prime-order subgroup. This could be done by multiplying it with the curve order and verifying that the result is the identity. This additional check would add an extra layer of security against Small-order subgroup attacks, enhancing the overall reliability and security of the Ring Signature implementation.

Equality Checks and Point Arithmetic Operations are Done in a Time Constant Fashion

TL;DR: The Ring Signature implementation ensures security by using constant-time equality checks and arithmetic operations, preventing timing attacks.

The Ring Signature implementation effectively maintains security by ensuring that equality checks and point arithmetic operations are performed in a time-constant fashion. This is crucial because timing attacks can potentially leak secret information, which could compromise anonymity or forgeability. By using techniques like constant-time comparisons and avoiding conditional branching instructions, the implementation prevents attackers from inferring sensitive information about the signer's private key.

More specifically, for the ed25519 curve, it was noted that the equality checks are time constant as demonstrated by the function `Eq` in the Go code. This method utilizes the `Equal` function from the 'inner' structure of two scalar instances and returns a boolean value based on the result. Additionally, it is worth mentioning the use of `subtle.ConstantTimeCompare` in the edwards25519 implementation, which ensures that comparisons take constant time to prevent timing attacks.

In regards to the secp256k1 curve, the function `Equals` was observed to perform a check using bitwise operations to compare each word of two scalars simultaneously. This technique avoids conditional branching instructions that may vary in computation times based on their values, hence maintaining constant time execution and enhancing security against timing attacks.

These observations suggest that the developers have taken necessary measures to ensure the robustness of the Ring Signature implementation from a security standpoint by incorporating time-constant equality checks and other secure curve arithmetic operations. This is a positive indicator of the implementation's overall resilience and makes it more difficult for potential attackers to exploit vulnerabilities or infer sensitive information about the signer's private key.

Compromised Privacy Feature of a Ring Signature if the Hash to Point Construction Allows for Commutativity

TL;DR: Commutativity attacks were a concern but determined a non-issue upon further analysis.

The security audit has identified a potential observation concerning the privacy feature of the Ring Signature implementation when utilizing hash to point constructions that may allow for commutativity. Specifically, in the provided function, there is an operation that calculates the key image using a hash-to-curve function. This construction could potentially expose the signer's key image to outsiders with just knowledge of their public key.

Upon further analysis, it was determined that the POKT ring-go hash-to-curve functions are based on using the hash output as a candidate for a curve point's X or Y coordinate. This method is resistant to commutativity attacks because it ensures validity by iterating until successful. Thus, an outsider would not be able to calculate the key image of the signer with knowledge only of their public key, providing a reassuring security measure against this particular threat vector.

This finding suggests that while the implementation does have a potential privacy concern, the hash-to-curve functions employed in the library provide an adequate solution to mitigate the risks associated with commutativity attacks.

Use of Dependencies

We investigated the use of dependencies in the POKT ring-go implementation.

Reliance on `DELq`

The Ring Signature implementation's reliance on external libraries, including `go-dleq`, raises concerns about potential vulnerabilities that may compromise anonymity or forgeability. While `go-dleq` provides efficiency benefits through scalar multiplication, its use introduces dependencies that may have underlying issues.

We wanted to highlight that `go-dleq`, based on the Monero research paper, has not undergone extensive scrutiny, which increases the risk of security concerns. The existence of a Rust implementation with identified issues further underscores the need for careful examination.

Although the specific findings from the Serai audit do not necessarily apply to `go-dleq`, it is essential to consider that other potential vulnerabilities might be present in this library. If such issues exist, they could compromise the security of the Ring Signature implementation.

To mitigate these risks, it is recommended that a detailed security review of `go-dleq` be conducted, taking into consideration the specific vulnerabilities identified in the Serai audit. This review should consider the potential impact of these issues on the Ring Signature implementation and propose appropriate mitigation or alternatives. Additionally, considering alternative implementations or libraries with stronger security track records could be beneficial.

Tests

We reviewed the POKT ring-go implementation tests and found them to be sufficient for a prototype, demonstrating good overall coverage. The tests are comprehensive and cover most basic scenarios. However, there are several areas where edge cases and invalid inputs are not adequately addressed.

To enhance the security posture of the ring-go's implementation, **it is recommended that the test suite be expanded to include comprehensive edge case coverage for invalid inputs, including potential malicious data scenarios. Additionally, specific tests for ed25519 signing and serialization failure handling**.

When running the benchmark tests we spotted that performance numbers show an anomalous result for rings of size 32. On inspection, this turns out to be a simple typo in the benchmark code, where the test only uses size 2 instead of 32.

Section_3.0

Key Findings Table

Issues	Severity	Status
ISSUE #1 Lack of Verification of Key Images May Lead to Potential Image Spoofing	Critical	Fixed
ISSUE #2 Lack of Uniqueness Check for Private Keys in Ring Signature Implementation	High	Fixed
ISSUE #3 Unintended Reordering of Public Keys in Ring Signatures Implementation	Low	Fixed
ISSUE #4 Ring Signature Comparison without Length Check	Low	Fixed
ISSUE #5 Error Handling and Validation Improvements	None	Fixed
ISSUE #6 Incorrect Ringsize in Ring Signatures Benchmarking for Size-32 Secp256k1	None	Fixed

Severity definitions can be found in Appendix A

Section_4.0

Findings

We describe the security issues identified during the security audit, along with their potential impact. We also note areas for improvement and optimizations in accordance with best practices. This includes recommendations to mitigate or remediate the issues we identify, in addition to their status before and after the fix verification.

ISSUE#1

Lack of Verification of Key Images May Lead to Potential Image Spoofing

Critical Fixed

Location

ring.go#L240

Description

This vulnerability enables an attacker to perform a brute-force attack on all possible values in the small subgroup of Ed25519-based ring signatures, which use a curve with a cofactor of 8. This allows for subgroups of size 8 to exist. The attacker can exploit this by tricking the `Link` function into accepting different key images corresponding to the same underlying private key, by adding a small-order subgroup point L to the key image I, resulting in I' = I + L. When the challenge scalar c is a multiple of 8, the result of multiplying I' by c is the same as the result of multiplying I by c, as L becomes the identity element. As a result, the signature passes verification for I' != I. This requires a brute-force for c with a 1/8 chance of success i.e. 3 bits of difficulty.

Impact

This vulnerability enables an attacker to defeat the linkability of the ring signatures by creating signatures with different key images despite sharing the secret key.

Recommendation

To mitigate this issue, we recommend implementing a check for key image correctness by multiplying it by the curve order `l`. Verify that the result is the identity element. For each transaction key image, perform the following check and reject the transaction if false:

```
((key image * curve order) == (identity element))
```

This validation ensures that only valid key images are accepted, preventing key image spoofing attacks.

Verification Status

The ring-go development team has fixed this issue by multiplying both key images by the curve cofactor before checking equality in the `Link` function. Multiplying by the cofactor will cause any added small subgroup point to become the identity, essentially removing it, before checking point equality.

ISSUE#2

Lack of Uniqueness Check for Private Keys in Ring Signature Implementation

High Fixed

Location

ring.go#L120-L128

Description

The current implementation of ring signatures does not ensure that the private key used to sign a message is unique within the set of public keys provided by the members of the ring. This lack of check could potentially compromise the anonymity properties of the signature scheme, as it allows for the identification of the signer among the members of the ring with known public keys.

```
// ...
pubkey := r.curve.ScalarBaseMul(privKey)
for i, pk := range r.pubkeys {
    if pk.Equals(pubkey) {
        ourIdx = i
        break
    }
}
// ...
```

In this snippet, the function checks if a public key derived from the provided private key is present in the ring's set of public keys. However, it does not validate that the private key is unique within the ring, allowing for potential non-uniqueness among the members.

Impact

The lack of a uniqueness check for private keys in the current implementation may result in the loss of anonymity guarantees provided by ring signatures. This could potentially expose the true identity of a user among the members of a ring.

Recommendation

We recommend introducing a check within the `Sign` function to ensure that the private key used for signing is unique within the set of public keys provided by the members of the ring. This can be accomplished by iterating over all public keys and checking if the derived public key from any private key matches with the input public key. If a match is found, return an error indicating that the private key is not unique.

We also recommend implementing additional error handling mechanisms to provide clear and actionable feedback when a non-unique private key is encountered during signing or verification processes.

Consider adding a feature flag for disabling the uniqueness check if necessary, with an appropriate warning about potential security implications. This would cater to any obscure cases where a user might intentionally use a non-unique private key but still maintain the overall security and integrity of the system.

Verification Status

The ring-go development team has implemented the recommended suggestion by checking for existing duplicate keys in the ring in functions responsible for creating ring signatures specifically `NewFixedKeyRingFromPublicKeys` and `NewFixedKeyRingFromPublicKeys` functions.

ISSUE#3

Unintended Reordering of Public Keys in Ring Signatures Implementation

Low Fixed

Location

ring.go#L69-L72

ring.go#L106-L110

Description

The `NewKeyRingFromPublicKeys` and `NewKeyRing` functions, which are part of a ring signatures implementation, create a new ring with an inserted public key at a specified index. However, the current implementation uses a circular shift to repopulate the array, leading to the loss of the original order of the public keys. This could potentially cause misuse or confusion for callers as the resulting ring may have a different order of public keys than the one originally supplied by the caller. Since signatures are dependent on the public keys' order in the ring, this discrepancy could theoretically impact the security and functionality of these signatures.

An example illustrates this issue. Given an original array of (1, 2, 3, 4), when a new item '10' is inserted at index 2, the resulting new array is (3, 4, 10, 1, 2). This outcome differs from the expected output and demonstrates that the order of public keys has been altered.

Impact

The issue's impact is mainly informational for users and developers who may have an expectation that the order of public keys in a new ring will be identical to the original ring with the exception of the inserted key. This discrepancy could cause confusion or lead to errors if callers are not aware of this behavior.

Recommendation

To ensure consistency, it is recommended to modify the implementation so that the order of public keys in a new ring remains as similar as possible to the original ring. This could be achieved by shifting elements after the insertion point rather than using a circular shift. A linear shift could maintain the relative positions of public keys before and after the inserted key, minimizing any potential confusion or errors for callers. This change would require minimal modification to the existing codebase and does not significantly impact performance.

Verification Status

The ring-go development team has implemented the recommended suggestion, modifying the `NewKeyRingFromPublicKeys` and `NewKeyRing` functions to maintain the original order of public keys in the Ring Signature implementation.

ISSUE#4

Ring Signature Comparison without Length Check

Low Fixed

Location

ring.go#L23-L32

Description

The `Equals` method in the provided Go code compares two rings by iterating over each public key and checking if they are equal using their respective `Equals` methods. However, there is no check for array length equality before starting the comparison loop. This means that if the two rings have different numbers of public keys, this code will cause a panic due to an out-of-bounds error when accessing `other.pubkeys[i]`. Although this issue was discovered in a testing context only, it is essential to consider fixes to anticipate for future code changes.

Impact

If the function is used with rings having different numbers of public keys, it can cause a panic and terminate the program abruptly. This could lead to unexpected behavior or errors in the system

Recommendation

We recommend adding a check at the beginning of the `Equals` method to ensure that both ring objects have the same number of public keys in their arrays.

Verification Status

The ring-go development team has implemented the recommended suggestion, addressing the issue by adding a length check before comparing the rings.

ISSUE#5

Error Handling and Validation Improvements

None Fixed

Location

ring.go#L59

Description

The provided ring signatures implementation has several areas where error handling and validation could be improved to increase reliability, robustness, and usability. This issue highlights two specific cases that require attention.

Firstly, the `NewKeyRingFromPublicKeys` function currently checks if the input `idx` is greater than `len(pubkeys)` to determine if it's out of bounds and returns an error accordingly. However, this check does not cover negative indices or zero, which could also cause errors when accessing elements in the `newRing` array.

To address this issue, it would be appropriate to check for all invalid index values (negative, zero, and greater than `len(pubkeys)`) and return an error message with more specific information about the valid range of indices. This can be achieved by adding a simple conditional statement:

```
 if idx <= 0 || idx > len(pubkeys) {
    return nil, errors.New("index out of bounds; expected a value between 1 and
len(pubkeys)")
}
```

Secondly, the `Sign` function could benefit from additional error checking and validation. Specifically, it should be ensured that the input private key is non-zero. A zero private key would result in an invalid signature, as the corresponding public key would be the point at infinity.

Impact

The absence of error handling and validation in these two functions could lead to unexpected behavior when interacting with this ring signatures implementation. Specifically in the `NewKeyRingFromPublicKeys` function, failing to check for invalid index values could result in incorrect key ring construction.

Recommendation

We recommend the following fixes:

1. Update the `NewKeyRingFromPublicKeys` function to check for all invalid index values (negative, zero, and greater than `len(pubkeys)`) and return an error message with more specific information about the valid range of indices.
2. Modify the `Sign` function to validate the input private key (ensure it's non-zero).

Verification Status

The ring-go development team has implemented the recommended improvements, adding several checks and validations to handle different possible edge cases in accordance with the suggested recommendations.

ISSUE#6

Incorrect Ringsize in Ring Signatures Benchmarking for Size-32 Secp256k1

None Fixed

Location

bench_test.go#L60-L66

Description

An anomalous result was discovered during benchmark testing for ring signatures implementation with the secp256k1 curve, specifically for rings of size 32. The test was supposed to use a ring size of 32 but instead, it used a size of 2. This discrepancy resulted in incorrect performance measurements which could potentially mislead developers and stakeholders about the actual efficiency and scalability of the implementation.

Impact

This issue may lead to incorrect assumptions about the implementation's efficiency and scalability. Since the issue only affects benchmarking and not the actual implementation, there is no immediate security risk.

Recommendation

We recommend correcting correcting the typo in the benchmark code where the ring size variable was set to 2 instead of 32. This will ensure accurate performance measurements for rings of size 32.

Verification Status

The ring-go development team has corrected the issue by fixing the affected test and updating the corresponding benchmark results in the `benchmarks.md` file.

Section_5.0

Appendix A

Severity Rating Definitions

At Thesis Defense, we utilize the Immunefi Vulnerability Severity Classification System - v2.3.

Severity	Definition

Critical	• Execute arbitrary system commands
	• Retrieve sensitive data/files from a running server, such as:
	  • /etc/shadow
	  • database passwords
	  • blockchain keys (this does not include non-sensitive environment variables, open source code, or usernames)
	  • Taking down the application/website
	• Taking state-modifying authenticated actions (with or without blockchain state interaction) on behalf of other users without any interaction by that user, such as:
	  • Changing registration information
	  • Commenting
	  • Voting
	  • Making trades
	  • Withdrawals, etc.
	• Subdomain takeover with already-connected wallet interaction
	• Direct theft of user funds Malicious interactions with an already-connected wallet, such as:
	  • Modifying transaction arguments or parameters
	  • Substituting contract addresses
	  • Submitting malicious transactions

High	• Injecting/modifying the static content on the target application without JavaScript (persistent), such as:
	  • HTML injection without JavaScript
	  • Replacing existing text with arbitrary text
	  • Arbitrary file uploads, etc.
	• Changing sensitive details of other users (including modifying browser local storage) without already-connected wallet interaction and with up to one click of user interaction, such as:
	• Email or password of the victim, etc.
	• Improperly disclosing confidential user information, such as:
	  • Email address
	  • Phone number
	  • Physical address, etc.
	• PSubdomain takeover without already-connected wallet interaction

Severity	Definition

Medium	• Changing non-sensitive details of other users (including modifying browser local storage) without already-connected wallet interaction and with up to one click of user interaction, such as:
	  • Changing the first/last name of user
	  • Enabling/disabling notifications
	• Injecting/modifying the static content on the target application without JavaScript (reflected), such as:
	  • Reflected HTML injection
	  • Loading external site data
	• Redirecting users to malicious websites (open redirect)

Low	• Changing details of other users (including modifying browser local storage) without already-connected wallet interaction and with significant user interaction, such as:
	  • Iframing leading to modifying the backend/browser state (must demonstrate impact with PoC)
	• Taking over broken or expired outgoing links, such as:
	  • Social media handles, etc
	• Temporarily disabling user to access target site, such as:
	  • Locking up the victim from login
	  • Cookie bombing, etc.

None	• We make note of issues of no severity that reflect best practice recommendations or opportunities for optimization, including, but not limited to, gas optimization, the divergence from standard coding practices, code readability issues, the incorrect use of dependencies, insufficient test coverage, or the absence of documentation or code comments.

Section_6.0

Appendix B

Thesis Defense Disclaimer

Thesis Defense conducts its security audits and other services provided based on agreed-upon and specific scopes of work (SOWs) with our Customers. The analysis provided in our reports is based solely on the information available and the state of the systems at the time of review. While Thesis Defense strives to provide thorough and accurate analysis, our reports do not constitute a guarantee of the project's security and should not be interpreted as assurances of error-free or risk-free project operations. It is imperative to acknowledge that all technological evaluations are inherently subject to risks and uncertainties due to the emergent nature of cryptographic technologies.

Our reports are not intended to be utilized as financial, investment, legal, tax, or regulatory advice, nor should they be perceived as an endorsement of any particular technology or project. No third party should rely on these reports for the purpose of making investment decisions or consider them as a guarantee of project security.

Links to external websites and references to third-party information within our reports are provided solely for the user's convenience. Thesis Defense does not control, endorse, or assume responsibility for the content or privacy practices of any linked external sites. Users should exercise caution and independently verify any information obtained from third-party sources.

The contents of our reports, including methodologies, data analysis, and conclusions, are the proprietary intellectual property of Thesis Defense and are provided exclusively for the specified use of our Customers. Unauthorized disclosure, reproduction, or distribution of this material is strictly prohibited unless explicitly authorized by Thesis Defense. Thesis Defense does not assume any obligation to update the information contained within our reports post-publication, nor do we owe a duty to any third party by virtue of making these analyses available.