FAILSAFE

Prepared for
Redacted

Prepared by
ret2basic.eth
y4y
FailSafe

6 February 2026

Redacted

Smart Contract Audit Report

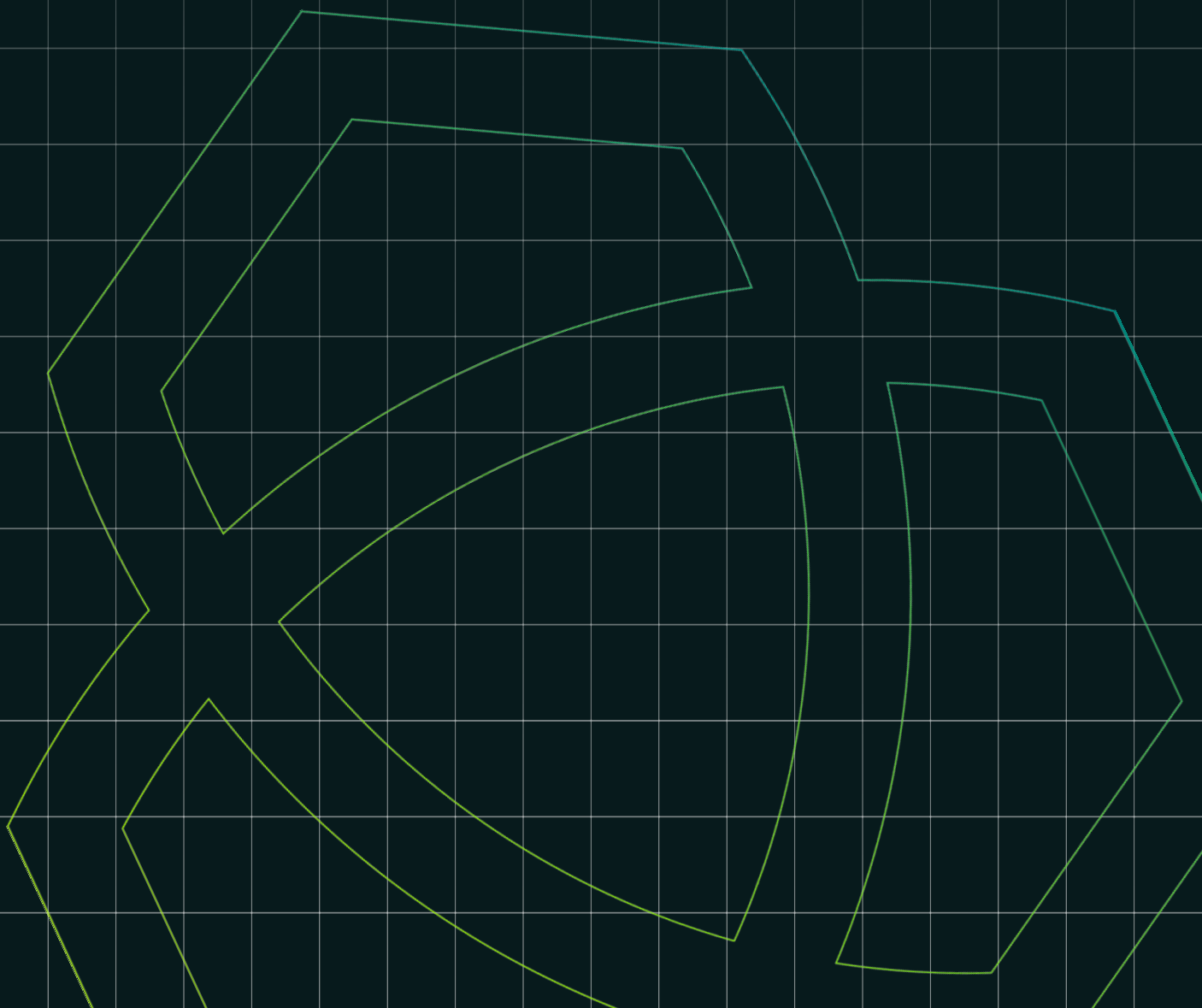

Table of Contents

Executive Summary

Redacted is a privacy-focused protocol built with CosmWasm on Rujira, THORChain's omnichain app layer. Our team approached this audit with meticulous attention to detail, leveraging extensive expertise in blockchain security to provide a thorough analysis of the system's security posture. The goal was to identify potential vulnerabilities and provide actionable recommendations to enhance the security and robustness of the smart contracts in question.

During the audit, several interesting security patterns and vulnerabilities were identified. Notably, a high-severity issue was discovered where fees intended to be applied during batch transactions were bypassed, potentially affecting protocol revenue without directly compromising user funds. Other findings included vulnerabilities that could lead to increased gas costs and potential denial-of-service attacks, such as permissionless deposits causing bloating of victim denom lists and malformed proof inputs leading to transaction panics. Additionally, concerns were raised over potential cross-deployment replay attacks due to a lack of contract-domain separation in zero-knowledge proofs, and the immutability of certain fee configurations which could result in revenue loss if the fee collector's address needs updating.

In conclusion, we commend the development team for their commitment to enhancing the security of their smart contracts. The proactive resolution of the high-severity issue and the team's responsiveness to our findings reflect a strong dedication to maintaining a secure and resilient system. We are confident that with the implementation of our recommendations, the project will significantly bolster its security posture, ensuring a more robust and trustworthy platform for its users and stakeholders.

Project Details

Project	Redacted
Website	https://redacted.gg
Repository	https://github.com/redactedLabs/redacted-contract-main
Blockchain	THORChain
Audit Type	Smart Contract Audit Report
Initial Commit	ad0e19b5c8dbaf807025a2002139c8e2142acac1
Final Commit	81319629397341383daf616bc9a0f867efb04981
Timeline	21 January 2026 - 2 February 2026
	Final Report: 6 February 2026

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- contracts/proxy/src/*
- contracts/sub-wallet/src/*

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
Critical	-	-	-	-	-
High	1	-	-	-	1
Medium	-	-	-	-	-
Low	1	-	-	-	1
Info	7	-	7	-	-
Total	9	0	7	0	2

#	Findings	Severity	Status
1	CallBatch fee not applied	High	Resolved
2	Unauthorized deposits can bloat victim denom list	Low	Resolved
3	Batch calls can bypass fees using non-Bank/Wasm message types	Info	Acknowledged
4	Deposits accept non-hex stealth and can lock funds	Info	Acknowledged
5	Missing field-range validation for stealth can permanently lock funds	Info	Acknowledged
6	Proofs lack contract-domain separation (cross-deployment replay)	Info	Acknowledged
7	Stealth encryption provides no confidentiality or integrity	Info	Acknowledged
8	SubWallet fee configuration is immutable	Info	Acknowledged
9	Verifier can panic on malformed proof input	Info	Acknowledged

Finding 1: CallBatch fee not applied

Severity: High

Status: Resolved

Description:

`CallBatchToSubWallet` computes fee-splitting messages but never executes them. The sub-wallet builds `new_msgs` with fee transfers and adjusted calls, yet returns the original `msgs`, so fees are bypassed whenever batch calls include funds.

The proxy forwards `CallBatchToSubWallet` to the sub-wallet using `SubWalletExecuteMsg::CallBatch { msgs }` (redacted-contract-main/contracts/proxy/src/contract.rs).

In the sub-wallet, `execute_call_batch` iterates over `msgs` and constructs `new_msgs` that include fee transfers and adjusted payouts for both `BankMsg::Send` and `WasmMsg::Execute` with funds (redacted-contract-main/contracts/sub-wallet/src/contract.rs). However, the function returns `Response::default().add_messages(msgs)`, which discards `new_msgs` and executes the original messages without fees (redacted-contract-main/contracts/sub-wallet/src/contract.rs).

Impact:

Fee revenue can be bypassed in any batch call that moves funds, breaking fee invariants and reducing protocol revenue. User funds are not directly stolen.

Source:

redacted-contract-main/contracts/sub-wallet/src/contract.rs, execute_call_batch()

Remediation:

Return new_msgs instead of msgs in execute_call_batch, and add a unit test that verifies fee transfers occur for batch calls with funds.

Discussion:

> *Developer:*
>
> Hey, just letting you know that we've taken care of the issue you pointed out. We made sure to return the properly constructed new_msgs instead of the original msgs. You can check out the changes we made in the latest commit here: https://github.com/redactedLabs/redacted-contract-main/commit/0711023bf4b08eec92a5069b40c8ddc372198845. Thanks for catching that!

> Fix URL: https://github.com/redactedLabs/redacted-contract-main/commit/0711023bf4b08eec92a5069b40c8ddc372198845

> *Auditor:*
>
> Great to hear that you've addressed the issue by updating the return value to use the correctly constructed new_msgs. We'll take a look at the changes in your latest commit to make sure everything's in order. Thanks for the quick turnaround!

Finding 2: Unauthorized deposits can bloat victim denom list

Severity: Low

Status: Resolved

Description:

Deposits are permissionless and only validate stealth length. An attacker can deposit dust in many different denoms to a victim's stealth, bloating the victim's `denoms` list and increasing gas costs for future operations or queries that iterate or serialize this list.

`execute_deposit` accepts any `stealth` with length 64 and any coin denom, then unconditionally inserts the denom into `user.denoms` if missing. There is no ownership or proof check for the `stealth` in deposits. Relevant code:

- `execute_deposit` only checks length and updates `user.denoms`: redacted-contract-main/contracts/proxy/src/contract.rs
- Denom insertion in the deposit path: redacted-contract-main/contracts/proxy/src/contract.rs

Because deposits are permissionless, an attacker can repeatedly send dust in many denoms to the same `stealth`, causing unbounded growth of the `denoms` vector for that user.

Impact:

Griefing: increased gas/serialization costs for the victim's future operations and queries, and potential client/UI slowdowns if the list grows large.

Remediation:

Consider gating deposits to a proof of ownership (or an allow-list of denoms), or store denoms in a bounded/ordered set with limits. Alternatively, avoid persisting a denom list at all and derive denoms by querying balances when needed.

Discussion:

> *Developer:*
>
> Hey, we went ahead and fixed the issue flagged under the FailSafe Admin module. You can check out the changes we made here: https://github.com/redactedLabs/redacted-contract-main/commit/81319629397341383daf616bc9a0f867efb04981. Let us know if everything looks good on your end!
>
> Fix URL: https://github.com/redactedLabs/redacted-contract-main/commit/81319629397341383daf616b

> c9a0f867efb04981

> *Auditor:*
>
> Great to see the fix pushed through for the FailSafe Admin issue. We'll take a closer look at the changes in the commit you shared and get back to you if we spot anything else. Thanks for the quick turnaround!

Finding 3: Batch calls can bypass fees using non-Bank/Wasm message types

Severity: Info

Status: Acknowledged

Description:

`CallBatch` in the sub-wallet only applies fee splitting for `BankMsg::Send` and `WasmMsg::Execute`. All other message types are forwarded unchanged, allowing attackers to move funds without paying protocol fees via non-Wasm/Bank Cosmos messages (e.g., `Stargate` messages).

In `execute_call_batch`, fee logic is applied only for `CosmosMsg::Bank` and `CosmosMsg::Wasm::Execute` branches; all other messages are appended as-is (redacted-contract-main/contracts/sub-wallet/src/contract.rs). This means a caller can use `CosmosMsg::Stargate` (or other supported types) to perform transfers or contract calls that move funds without any fee deduction.

Impact:

Protocol fee collection can be bypassed for batch calls that use non-Bank/Wasm message types. This is a revenue loss and breaks fee invariants.

Source:

redacted-contract-main/contracts/sub-wallet/src/contract.rs, execute_call_batch()

Remediation:

Restrict CallBatch to an explicit allow-list of message types and reject all others, or extend the fee logic to cover any message type that can move funds (including Stargate transfers).

Discussion:

> *Developer:*
>
> So, we took a good look at this finding and it turns out it's not something that can actually happen with our implementation. The executecallbatch function is set up to only allow specific types of messages, specifically CosmosMsg::Bank(BankMsg::Send) and CosmosMsg::Wasm(WasmMsg::Execute). Anything else gets shut down with an error, InvalidCosmosMsg, before it even gets a chance to run. This means that messages like thorchain.MsgDeposit can't sneak through, so the fee-bypass thing you mentioned isn't a concern for us right now.

> *Auditor:*
>
> We've heard what you're saying. It sounds like you've got a solid system in place for filtering out unwanted message types, which should prevent the kind of exploit we were worried about. It's good to know you've got those checks in place to keep everything above board. Let's keep an eye on it though, just in case anything changes down the line.

Finding 4: Deposits accept non-hex stealth and can lock funds

Severity: Info

Status: Acknowledged

Description:

`execute_deposit` only checks length for `stealth` and accepts any 64-character string. Later proof-gated flows derive the plaintext stealth via `decrypt_stealth`, which always returns a canonical lowercase hex string. If a user deposits with non-hex (or non-canonical) `stealth`, their balances are written under an unreachable key and can never be withdrawn.

- Deposits only validate length, then store balances under the provided `stealth` string with no hex or canonicalization checks: redacted-contract-main/contracts/proxy/src/contract.rs.
- Decryption always returns a hex-encoded string of the decrypted bytes (lowercase): redacted-contract-main/contracts/proxy/src/crypto.rs.
- Proof-gated flows use the decrypted stealth as the map key. If the original deposit key was not canonical hex, it will never match the decrypted stealth value, permanently isolating the balance.

Impact:

User funds deposited with a non-hex or non-canonical stealth become irrecoverable. This can occur due to user error or client bugs, and results in a permanent loss of funds for that user.

Remediation:

Validate that stealth is a 32-byte hex string at deposit (e.g., hex::decode success) and store a canonical lowercase hex form. Alternatively, accept raw bytes and normalize to a fixed hex encoding before using it as a storage key.

Discussion:

> *Developer:*
>
> Hey, so about the stealth string issue you pointed out, here's the deal: our contract does accept any 64-character stealth string, but in reality, stealth values never come from users directly. They're always generated by our Rujira system off-chain, which ensures they're always in lowercase hex format. So, yeah, technically someone could throw a weird string directly at the contract, but it wouldn't do anything because those wouldn't link to any valid stealth owner. They just sit there, unable to be withdrawn or interacted with. It's more of a theoretical thing and doesn't really impact our system since we only support deposits through

> Rujira.

> *Auditor:*
>
> We understand your point about the stealth string generation, and it's reassuring that the Rujira system controls this process, keeping values in the correct format. The edge case of malformed deposits is recognized, though it doesn't impact system functionality due to the lack of association with valid owners. It's good to know these wouldn't affect legitimate users or operations. We see this as a theoretical risk rather than an immediate concern, given the controlled deposit flow through Rujira.

Finding 5: Missing field-range validation for stealth can permanently lock funds

Severity: Info

Status: Acknowledged

Description:

Deposits accept any 32-byte `stealth` value,

```
// redacted-contract-main/contracts/proxy/src/contract.rs
// @audit-info 64 digit string => 32 bytes => [0, 2^256 - 1) range
const STEALTH_LEN: usize = 64;
```

but proof verification rejects inputs not in the SNARK scalar field.

```
    // redacted-contract-main/packages/protocol/src/verify.rs
    for (i, item) in input_words.iter().enumerate() {
        if get_uint256_from_vec(item) >= SNARK_SCALAR_FIELD {
            return Err(StdError::generic_err("verifier-gte-snark-scalar-field"));
        }
        vk_x = ecadd(&vk_x, &ecmul(&vk.ic[i + 1], item).unwrap()).unwrap();
    }
```

As a result, deposited stealth values in the range `field_modulus < stealth_value < 2^256 - 1` can never produce a valid proof, permanently locking the user's funds. Considering field modulus is in the 2^254 range, this "limbo" space between field modulus and 2^256 - 1 is actually quite large.

Because `stealth` is used directly as a public input chunk in every proof verification path (e.g., after decryption, `input` starts with the 32-byte `decrypted_stealth`), any `stealth` value ≥ field size will cause proof verification to revert. This is a permanent condition for that user's funds.

Impact:

User deposits can be irrecoverably locked. This is a direct loss-of-funds risk for any user whose stealth is out of field range.

Source:

redacted-contract-main/contracts/proxy/src/contract.rs, execute_deposit()

Remediation:

Enforce field-range validation for stealth at deposit and after decryption (e.g., ensure stealth < SNARK_SCALAR_FIELD), or hash-to-field / reduce into the scalar field before using it as a public input.

Discussion:

Developer:

Hey, so about that finding. It seems there was a bit of a misunderstanding about how our system works. The assumption that users can submit any 32-byte stealth values at deposit time doesn't quite fit with our setup. You see, stealth values aren't something users can just throw in themselves. They're actually generated off-chain by our backend using Poseidon. This means every stealth value we produce is already within the SNARK scalar field limit, so there's no chance of hitting that problematic range where funds could get locked. We made sure our entire system – from the frontend to the backend and all the user interaction paths – never allows for arbitrary stealth input. So, unless we fundamentally change how stealth is generated or somehow let users bypass the backend, this loss-of-funds scenario just can't happen with our current design.

Auditor:

Got it, thanks for clarifying! It sounds like the key point here is that stealth values are tightly controlled and generated by the backend using Poseidon, ensuring they stay within the appropriate range. Since the system doesn't allow users to directly input these stealth values, the risk we identified of funds getting locked due to out-of-range values isn't really applicable here. We appreciate the detailed explanation and will note that the described scenario isn't possible with the current system setup.

Finding 6: Proofs lack contract-domain separation (cross-deployment replay)

Severity: Info

Status: Acknowledged

Description:

ZK proofs are verified without binding the public input to a specific proxy contract instance (or chain). If the same verifying key is deployed to multiple proxy contracts, a valid proof generated for one instance can be replayed on another instance that has the same `stealth` and `nonce` state.

The proof input is built as `decrypted_stealth || sha256(combined_str)` where `combined_str` contains only action parameters and the current `nonce`. The contract address (or chain ID) is not included in any of these hashes. Example locations:

- `execute_call_to_sub_wallet` input construction: redacted-contract-main/contracts/proxy/src/contract.rs
- `execute_withdraw` input construction: redacted-contract-main/contracts/proxy/src/contract.rs

If another proxy instance uses the same verifying key and a user reuses the same `stealth` (or an attacker can observe it), the same proof can be accepted on the other instance as long as the `nonce` matches there.

Impact:

Cross-deployment replay can execute the same action on a different proxy instance. This is a low-severity risk that becomes relevant if multiple deployments share the same verifying key and users reuse a stealth across instances.

Remediation:

Domain-separate the public input by including env.contract.address and (optionally) env.block.chain_id in the hashed value for every proof-verified path. This binds proofs to a specific deployment and prevents cross-contract replay.

Discussion:

> *Developer:*
>
> Okay, so we've taken a look at the items you pointed out, but we don't see them as relevant to our current deployment model or scope. Basically, they don't pose a practical risk in our current system, so we're intentionally not addressing them right now.

> *Auditor:*
>
> Got it, we understand your perspective on not addressing these items because they don't fit into your current model and scope. It's important for us to ensure there's no practical risk, but we're here to help if anything changes or if you need further clarification on those items in the future.

Finding 7: Stealth encryption provides no confidentiality or integrity

Severity: Info

Status: Acknowledged

Description:

The "encrypted stealth" is decryptable by anyone because the key and IV are derived from publicly-visible salts and the user-provided `t` (which is submitted on-chain). This defeats the privacy goal and allows passive tracking.

`decrypt_stealth` derives a key and IV solely from `KEY_SALT`, `IV_SALT`, and `timestamp_str` (`t`). Since `t` is provided in every user call, any observer can reconstruct the key/IV and decrypt the stealth.

- Key/IV derivation and AES-CBC decryption: redacted-contract-main/contracts/proxy/src/crypto.rs
- Usage in proof-gated flows (example withdraw): redacted-contract-main/contracts/proxy/src/contract.rs

Impact:

Privacy assumptions are broken: anyone can recover the stealth identifier and link deposits/withdrawals. If the stealth is meant to be secret, this undermines the protocol's anonymity guarantees and exposes user activity.

Source:

redacted-contract-main/contracts/proxy/src/crypto.rs

Proof of Concept:

1. Observe a transaction that includes `stealth` (ciphertext) and `t` (timestamp).
2. Download the on-chain WASM for the contract's `code_id` and extract embedded strings (e.g., via `wasm2wat`/`wasm-objdump` or a simple `strings` scan) to recover `KEY_SALT` and `IV_SALT` from the data section.
3. Recompute the AES key and IV using the recovered salts and the observed `t`.
4. Decrypt the `stealth` ciphertext to recover the plaintext stealth identifier.
5. Use the recovered identifier to correlate future deposits/withdrawals that use the same stealth.

Remediation:

Replace this scheme with standard public-key encryption or authenticated encryption (e.g., ECIES or X25519 + AEAD), where only the intended recipient can decrypt. Do not derive keys from user-supplied timestamps. If stealth must remain private, remove on-chain decryption altogether and verify proofs using hashed or committed values instead.

Discussion:

> *Developer:*
>
> Hey there! So, we wanted to address the concern about the initialization vector (IV) that was raised. In our setup, the IV is actually treated as a non-public backend value and is kept private. Because of this, the scenario you described wouldn't really be applicable in our production environment. Let us know if you need more details on how we handle IVs.

> *Auditor:*
>
> Thanks for the clarification! We understand that the IV is kept private on your backend, which addresses the concern we had about the attack scenario. It's good to know that you're handling it securely in production. If there are any changes or further considerations, we'd love to hear about them.

Finding 8: SubWallet fee configuration is immutable

Severity: Info

Status: Acknowledged

Description:

SubWallets snapshot `fee_address` and `fee` at instantiation and never update them. If the fee collector key is compromised or needs rotation, existing SubWallets will continue paying the old address indefinitely.

The SubWallet contract stores config only during `instantiate` and exposes no `ExecuteMsg` to update it. `Config` contains `fee_address` and `fee`, but there is no update path in `execute` or any admin/proxy-only entry point to rotate these values. See redacted-contract-main/contracts/sub-wallet/src/contract.rs and redacted-contract-main/contracts/sub-wallet/src/config.rs.

Impact:

If the protocol's fee collector address is compromised or needs rotation, existing SubWallets will continue sending fees to the old address, resulting in permanent revenue loss for those wallets.

Remediation:

Add a proxy-only UpdateConfig execute entry point to SubWallet, or fetch the fee config dynamically from the Proxy during execution (tradeoff: additional gas).

Discussion:

> *Developer:*
>
> Hey, we've taken a good look at the audit findings you mentioned. We understand the concerns, but given our current deployment model and the scope we're working within, we don't see these items as relevant or posing any practical risk to the system right now. That's why we're intentionally not addressing them at this time. Let's keep the conversation going though, in case things change down the road.

> *Auditor:*
>
> We hear you on the current deployment model and scope, and it's clear you've thought about the relevance of these items. From our side, we just want to make sure that even if they don't seem critical now, they're on the radar for any future changes in the deployment or scope. It's all about keeping things secure as things

> evolve, right?

Finding 9: Verifier can panic on malformed proof input

Severity: Info

Status: Acknowledged

Description:

`verify_proof` uses `unwrap()` on hex decoding and indexing without validating input length. Malformed proof or input strings can trigger panics, causing the transaction to abort. This enables gas-griefing/DoS of proof-gated flows.

`verify_proof` calls `hex_to_bytes(...).unwrap()` and indexes `p[0..7]` without length checks (redacted-contract-main/packages/protocol/src/verify.rs). It also `unwrap()`s EC helpers (redacted-contract-main/packages/protocol/src/verify.rs).

If a caller submits a malformed hex string or too-short proof, the contract panics rather than returning a structured error. This reverts the execution and can be used to spam failing transactions.

Impact:

Denial-of-service via gas-griefing on proof-gated methods. No direct fund loss, but reduced availability and wasted gas.

Source:

redacted-contract-main/packages/protocol/src/verify.rs, verify_proof()

Remediation:

Validate hex length and proof size before indexing. Replace unwrap() calls with error returns so invalid proofs fail gracefully.

Discussion:

> *Developer:*
>
> Hey, so about the security concern you mentioned, we wanted to clarify that the proofs are actually generated only by our backend system. Users can't create malformed proofs themselves, so this particular issue shouldn't be something that can happen in production. With the way our architecture is set up right now, this doesn't seem like something that could be exploited.

> *Auditor:*
>
> Okay, thanks for the explanation. It's good to know that the proofs are generated exclusively by the backend, which definitely limits the potential for user-generated errors. We'll take this into account and review how this aligns with the current architecture to ensure everything's secure and working as expected.

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.