Security Audit Report

Mezo:

Orangekit Smart Contracts

Initial Audit Report: April 5, 2024

thesis defense
defense@thesis.co

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 Embody. Our team of senior security and cryptography auditors has extensive security experience in the decentralized technology space. In addition, the Thesis Defense team has a demonstrated track record in a variety of languages and technologies, including, but not limited to, smart contracts, cryptographic protocols including zk-cryptography, dApps including wallets and browser extensions, and bridges. Thesis Defense has extensive experience conducting security audits across a number of ecosystems, including, but not limited to, Ethereum, Zcash, Stacks, Mina, Polygon, Filecoin, and Bitcoin.

Thesis Defense will employ the Thesis Defense Audit Approach and Audit Process to the above in-scope service. In the event that certain processes and methodologies are not applicable to the aforementioned 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.

Scope

Overview

Thesis Defense conducted a manual code review of Mezo Orangekit smart contract implementation.

Project Team

- **Bernd Artmüller,** Security Auditor
- **Yuma Buchrieser,** Security Auditor
- **Bashir Abu-Amr,** Senior Technical Writer

Schedule

- **Code Review:** April 1 - 5, 2024
- **Audit Report Delivery:** April 5, 2024

Code

- **Repository:** https://github.com/thesis/orangekit/tree/main/solidity
- **Hash:** `44355ad8dbac7df34d069fca2720c9e3f96b0ff9`

Project Documentation

- **Technical Documentation & Architectural Diagram:** RFC: OrangeKit Bitcoin Account Metaprotocol
- The instructions for setting up the repository and running tests are available in the README file

Bibliography/Reference

- Message signing - Bitcoin Wiki. (n.d.). https://en.bitcoin.it/wiki/Message_signing

---

Findings

The Mezo Orangekit protocol allows users to interact with the Ethereum EVM ecosystem using their Bitcoin private key. This is facilitated by using an ERC-4337 compatible Gnosis Safe and a custom implementation of a Safe owner. This Safe owner smart contract can verify different Bitcoin signatures and allows the user to control the actions of the Gnosis Safe by signing messages and sending them to the Safe to execute.

Threat Model

For this review, our team considered a threat model whereby external components to the smart contracts are untrusted but function as intended. These components include any user interface that enables interaction with the protocol, any off-chain components that are an integral part of the system, and any third-party dependencies or services that are necessary for the protocol to function as intended. Furthermore, we considered the governance of the protocol to be not malicious.

Due to the usage of the battle-tested Gnosis Safe, the attack surface of the Orangekit contracts is very limited. Nevertheless, multiple threats were considered in the audit:

- Forgeability of signatures
- Replaying of signatures
- Unauthorized access to the safe / deployer / factory
- Unpredictability of addresses

The main attackers considered are attackers trying to pass invalid or already used signatures to get the Gnosis Safe to act maliciously.

Security by Design

The Orangekit system design is robust, and the protocol's security has been considered and prioritized. The usage of the Gnosis Safe contract instead of a custom implementation also supports this fact. Throughout the code, authorization is correctly implemented, and many potential attack vectors are already mitigated through various security measures. For instance, replaying transactions is prevented by the Gnosis Safe contract by incorporating the chain id as well as an increasing nonce in the message digest. The additional possibility of emergency upgrades through a trusted entity allows the governance (Multisig) to take remediative action in case of a compromised implementation of the Safe owner smart contract.

Secure Implementation

We found the code to be well-organized, properly documented, and adhering to best practices. We investigated the security of the implementation of the two most sensitive areas: signature verification and contract deployment to predictable addresses. As a result of our review, we did not identify any critical security vulnerabilities.

Use of Dependencies

We ran the `pnpm audit` dependency analysis tool and did not identify any issues in the use of Solidity dependencies.

Tests

The Orangekit repository contains unit and integration tests for the contracts in the scope of this review, with 96.79% line and 93.52% branch coverage, in accordance with best practices. We recommend adding tests for the `OrangekitDeployer` contract that try to redeploy the individual contracts, which results in an error because they use the same salt and bytecode.

Project Documentation

The Orangekit smart contracts are well documented in RFC: OrangeKit Bitcoin Account Metaprotocol, provided by the client. This documentation encompasses detailed technical descriptions of the smart contracts' essential functions and is also supplemented by an architectural diagram. Moreover, the code itself is well-commented.

In addition, we recommend making the available documentation publicly available to make it easier for developers to inform themselves about the workings of the protocol.

Issues and Suggestions

Issues	Status
Issue A: The BitcoinSafeOwner and OrangeKitSafeFactory Contracts can be Upgraded with a Non-Contract Address Resulting in a Non-Functional Proxy	Reported
Issue B: Two-Step Ownership Transfer is Recommended	Reported

Suggestions	Status
Suggestion 1: Rename the EmergencyUpgradesDisabled Error	Reported
Suggestion 2: Private and Internal Functions Do Not Adhere to the Solidity Style Guide	Reported
Suggestion 3: Comment Includes Spelling Issue	Reported
Suggestion 4: Missing NatSpec Return Value Documentation	Reported
Suggestion 5: Add Indexing to Events With Multiple Parameters	Reported
Suggestion 6: Missing Derived Address Prefix Allows Users to Pass Compressed P2PKH Signatures as P2WPKH Signatures	Reported
Suggestion 7: Emit an Additional Event During Emergency Upgrades of the BitcoinSafeOwner Contract	Reported
Suggestion 8: Avoid Using the Latest Solidity Compiler Version To Reduce Compiler Bug Risks	Reported
Suggestion 9: No Protection Against Outdated Version Upgrade	Reported

Issues

Issue A: The BitcoinSafeOwner and OrangeKitSafeFactory Contracts can be Upgraded with a Non-Contract Address Resulting in a Non-Functional Proxy

Location

BitcoinSafeOwner.sol#L157-L162
OrangeKitSafeFactory.sol#L171-L176

Description

The `BitcoinSafeOwner` and `OrangeKitSafeFactory` contracts can be upgraded via the `upgradeSingleton` function by changing the proxy's implementation address, stored in `singleton`. The address of the new implementation contract is validated to prevent the zero address, the proxy's address, or the address of the current implementation contract from being set as the new implementation.

However, if the address provided does not refer to a contract, the upgrade will succeed due to the proxy's low-level `delegatecall` not reverting if the target is not a contract. As a result, the proxy will be non-functioning and cannot be upgraded again as the upgrade functionality is not available.

Impact

The upgraded smart contracts are non-functional and cannot be upgraded again.

Remediation

We recommend using the `isContract` function to check if the new implementation address is a contract before proceeding with the upgrade.

Issue B: Two-Step Ownership Transfer is Recommended

Location

OrangeKitSafeFactory.sol#L192-L199

Description

The `OrangeKitSafeFactory` implements an ownership mechanism that is used to allow the `EmergencyUpgrader` to upgrade the singletons in case of a potential compromise. The

functionality currently only allows for a single-step ownership transfer, which could lead to issues if the ownership is accidentally transferred to the wrong address.

Impact

If the ownership of the `OrangeKitSafeFactory` is transferred to an unintended address by accident it will make it impossible for the `EmergencyUpgrader` to upgrade the singletons in the case of the used one becoming vulnerable. This will result in new safes being deployed with a vulnerable version without a way of preventing it.

Remediation

We recommend implementing a two-step ownership transfer. This can be done by using an already implemented library implementation like OpenZeppelin. To still grant the ownership to the `EmergencyUpgrader` when calling `initialize`, the function can be adapted so that an owner parameter can be passed that is set when initializing.

Suggestions

Suggestion 1: Rename the EmergencyUpgradesDisabled Error

Location

EmergencyGovernance.sol#L28

Description

The OrangeKit protocol implements an emergency governance feature that allows a MultiSig controlled by the protocol itself to upgrade the `BitocinSafeOwner`. To inform the deployed `BitcoinSafeOwner` smart contracts what address currently holds the `EmergencyUpgrader` role, the `EmergencyGovernance` smart contract is used. This smart contract can also be disabled by its owner, which will make emergency upgrades impossible afterward. After the `EmergencyGovernance` was disabled, the `EmergencyUpgradesDisabled` error is thrown at each call to its 3 functionalities:

1. Retrieving the current emergency upgrader
2. Disabling the `EmergencyGovernance`
3. Setting a new emergency upgrader

As the error's name suggests, it is intended to be returned when the emergency governor tries to upgrade a safe owner, but the `EmergencyGovernance` is already disabled. In the other two cases, the error might lead to confusion.

Remediation

We recommend splitting the error into 3 separate errors (which would increase overhead) or renaming it to a more generic version that fits all 3 cases of reverting. One recommendation is to rename the error to `EmergencyGovernanceDisabled`.

Suggestion 2: Private and Internal Functions Do Not Adhere to the Solidity Style Guide

Location

OrangeKitSafeFactory.sol#L208, L270, L322, L339, L359, L396
OrangeKitDeployer.sol#L137,
BitcoinSafeOwner.sol#L278, L371, L398, L425, L456, L480, L492, L516, L539

Description

To ensure good code readability and prevent future issues, it is highly recommended that Solidity code follow the Solidity Style Guide. The style guide states that non-external functions should be prefixed with an underline. This currently needs to be implemented for many functions used in the protocol.

Remediation

We recommend prefixing the private and internal functions accordingly.

Suggestion 3: Comment Includes Spelling Issue

Location

solidity/contracts/BitcoinSafeOwner.sol#L317

Description

One of the comments inside the `BitcoinSafeOwner` includes a typo in the word “varint” which should correctly be “variant.”

Remediation

We recommend fixing the spelling issue.

Suggestion 4: Missing NatSpec Return Value Documentation

Location

BitcoinSafeOwner.sol#L127
EmergencyGovernance.sol#L44
LegacyERC1271.sol#L34
OrangeKitSafeFactory.sol#L99, L122, L208, L270

Description

It is recommended to use NatSpec documentation to improve code readability. Throughout the codebase, the `@return` tag is used most of the time correctly to document the returned variables. Still, some functions are fully missing the comment or have a comment describing the returned value but with an incorrect tag.

Remediation

We recommend documenting the return value in NatSpec for each of the functions.

Suggestion 5: Add Indexing to Events With Multiple Parameters

Location

OrangeKitSafeFactory.sol#L39-L40
BitcoinSafeOwner.sol#L58
EmergencyGovernance.sol#L19

Description

The Orangekit protocol correctly emits events on state changes. To make it easier to monitor these events it is recommended to index them if they have more than one argument. This is correctly implemented for one event but not for all.

Remediation

We recommend indexing the mentioned events to improve monitoring abilities.

Suggestion 6: Missing Derived Address Prefix Allows Users to Pass Compressed P2PKH Signatures as P2WPKH Signatures

Location

BitcoinSafeOwner.sol#L456

Description

The OrangeKit protocol implements the `BitcoinSafeOwner` contract which can be used to verify messages signed by a Bitcoin address. These messages can be verified in four ways depending on the type of address that has encoded them. For the address types compressedP2PKH and P2WPKH the signatures are verified the same way. The only way the contract distinguishes between them is by checking the v value and then decreasing it by 8 if P2WPKH is detected.

```
uint8 prefix = uint8(uint256(y) & 1) + uint8(2);
bytes20 publicKeyHash = hash160(abi.encodePacked(prefix, x));
```

```
return
      ecrecover(signedMessage, v, r, s) ==
      publicKeyToEthereumAddress(x, y) &&
      truncatedBitcoinAddress == publicKeyHash;
```

Consequently, any compressed P2PKH signature can also be passed as a P2WPKH signature by increasing the value of v by 8.

Fortunately, this currently does not lead to issues as the Gnosis Safe protects against message replays by using nonces inside the messages. Nevertheless, this could lead to issues if the `BitcoinSafeOwner` is ever used with a different multi-sig contract, that, for example, protects against replay attacks by creating a digest of the message + signature. In that case, the message could be used once as compressed P2PKH and once as P2WPKH.

Remediation

We recommend adding an enum for the type of address used when creating the Safe and only allowing signature verification of that type.

Suggestion 7: Emit an Additional Event During Emergency Upgrades of the BitcoinSafeOwner Contract

Location

BitcoinSafeOwner.sol#L215

Description

In case of emergency, the `BitcoinSafeOwner` contract can be upgraded by the governance upgrader via the `emergencyUpgradeSingleton` function. As a result, the event `SingletonUpgraded(address oldSingleton, address newSingleton)` is emitted, similarly to a regular upgrade by the contract owner via the `upgradeSingleton` function. Emitting an additional event in case of an emergency upgrade would allow better differentiation between governance and owner upgrades for more effective off-chain monitoring.

Remediation

We recommend emitting an additional event, for example, `SingletonUpgradedEmergency`, in the `emergencyUpgradeSingleton` function.

Suggestion 8: Avoid Using the Latest Solidity Compiler Version To Reduce Compiler Bug Risks

Location

BitcoinSafeOwner.sol#L2
ERC1271.sol#L2
EmergencyGovernance.sol#L2
LegacyERC1271.sol#L2
OrangeKitDeployer.sol#L2
OrangeKitSafeFactory.sol#L2
Proxy.sol#L2

Description

All smart contracts in the scope of this review use the `pragma solidity 0.8.25` statement. This sets the version of the Solidity compiler, solc, to 0.8.25, which is the latest version at the time of this review. However, new compiler versions can occasionally introduce bugs and unknown vulnerabilities, and therefore, using the latest compiler version may pose a risk.

Remediation

We recommend not using the latest Solidity compiler version, especially if none of the latest compiler features are used. This reduces the risk of potentially introducing unknown compiler bugs. For example, consider using solc version 0.8.24.

Suggestion 9: No Protection Against Outdated Version Upgrade

Location

BitcoinSafeOwner.sol#L152

Description

The `BitcoinSafeOwner` smart contract allows upgrades to include new functionalities or remove vulnerabilities. To protect the upgrading process against signature replay attacks, i.e., redeploying previous contract versions, each message includes the domain separator that includes the version number. As long as the version number increases at each upgrade, the signature can never be replayed.

Unfortunately, besides code comments pointing out the necessity to strictly increase the version, there are no explicit checks in the code that enforce this. The code only checks that the new initializer function is called with a function value, but no check is done to verify that the version number has increased.

Remediation

There are 2 ways how this issue can be mitigated:

1. **Hash mapping:** The more gas-intensive way would be to implement an additional mapping of `keccak256(versionString) => bool`. At the end of the `setup` function, it must be checked that the hash of the new version is not already contained in that mapping, reverting otherwise. If the version is indeed new, it is added to the mapping, and the upgrade succeeds. This way, the version formatted as a string can stay, but this comes at the cost of increased gas costs.
2. **Numerical version number:** The less gas-intensive way of securing users against accidentally upgrading to an old version is to use numerical version numbers. This can be done by replacing the version number string with an `uint256`. In this case, the function can easily check if the new version number is higher than the old one and revert otherwise.