PUBLIC

Code Assessment

of the Unstoppable Wallet Smart Contracts

February 7, 2025

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Unstoppable Team,

Thank you for trusting us to help Unstoppable with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of the Unstoppable Wallet according to Scope to support you in forming an opinion on their security risks.

Unstoppable implements a smart contract wallet that supports authentication via WebAuthn, and enables ownership transfers via an inheritance, and a social recovery mechanism.

The most critical subjects covered in our audit are the safety of the funds, the security of the ownership recovery mechanisms and their resistance against malicious actors, the signature validation, and the correct configuration of the wallet. Our most important findings concerned the different recovery mechanisms of the wallet, as described in the issues Ownership transfer race conditions and Social recovery with less than minConfirmations possible. All the issues have been addressed by Unstoppable and security regarding the afore mentioned areas is high.

The general subjects covered are interactions with other addresses, access control, and gas efficiency. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase could provide a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	4
• Code Corrected	3
• Specification Changed	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Unstoppable Wallet repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	10 Dec 2024	8769d48adcf45a04e1cbb3f6ae7dc342a21ca7d9	Initial Version
2	14 Jan 2025	6c57943cd6809a5b03a45d547cf7a745e47ec74c	Fixes
3	7 Feb 2025	c7dfcee29686a85c878230c5bbc10a9e85b30289	Final version

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The following contracts are in the scope of the review:

```
src/
    GlobalConfig.sol
    UpgradeableBeacon.sol
    Wallet.sol
    WalletFactory.sol
    interfaces/
        IGlobalConfig.sol
        IUpgradeableBeacon.sol
        IWallet.sol
        IWalletFactory.sol
```

2.1.1 Excluded from scope

The contracts in scope make use of the OpenZeppelin library. The library is assumed to function correctly. The contracts are intended to be deployed, initially on Arbitrum, where the execution environment is assumed to be equivalent to Ethereum. Wallet owners can use WebAuthn for authentication. We assume that the server creates valid messages for the users to sign through the authenticator. Any attacks on the front-end of the application (e.g., DNS hijacks) are considered out of scope. The authentication scheme was reviewed only for replayability and malleability resistance on the smart contract level. For signature verification, the verifier deployed at `0xc2b78104907F722DABAc4C69f826a522B2754De4` is used and assumed to function properly.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Unstoppable offers the implementation of the Unstoppable wallet, a smart contract wallet that supports user authentication via WebAuthn, wallet inheritance after a specified period of inactivity and a social recovery mechanism.

An Unstoppable Wallet defines an owner and an operator. The owner is identified by an Ethereum address, while the operator is defined by secp256r1 public key material used for verifying WebAuthn-compliant signatures. Both owner and operator can make the wallet interact with other addresses by making calls or sending native assets. Upon initialization, only the operator is defined. As long as there's no owner (`s_owner == address(0)`), the operator is the most privileged role. This means that in addition to making external calls, the operator can change various configurations of the wallet. If an owner is set, then the owner is the most privileged role. This hierarchy is enforced by the `onlyOwnerOtherwiseOperator` modifier. As the operator cannot sign messages, all their actions are initiated by `executeCalls` which verifies the operator's signature. Therefore, calling an admin function is achieved by having the wallet call itself. This means that an action initiated by the operator is checked by asserting that `msg.sender == address(this)`. On the other hand, calls to the wallet from the wallet are not allowed for the owner. Note that this check is not very strict as in theory an owner can simply craft a callpath that will allow them to initiate an operator's action and vice versa.

2.2.1 Call execution interface

- The owner of the contract can execute a batch of calls via `executeCalls()`. The whole batch reverts if any call fails. A call is considered failed if it reverts during its runtime or if it targets a non-contract address with some non-empty calldata. The exception to this rule is when calling the wallet itself during its construction which is still allowed. The wallet keeps track of the timestamp of the latest executed call.
- Any user can call `executeCalls()` on behalf of the operator as long as they provide a valid operator signature. If `s_verifyContract` is set, these calls are verified by a verification contract. For P256 signature verification, the wallet uses the `P256Verifier` contract at `0xc2b78104907F722DABAc4C69f826a522B2754De4`. Upon receiving signed data via `executeCalls()`, it is first verified that the signed message contains the correct data; this includes the input parameters, the wallet's current nonce, the wallet address, and the chain ID. The Base64 URL-encoded hash of this data is compared against the challenge data found at a specific offset within the signed message (`_clientDataJSON`). It is then verified that the signature was produced by the operator's private key via the `P256Verifier` contract. If this verification fails, the call reverts. For every successful execution of calls signed by the operator, the wallet's nonce is incremented by one. Apart from the described signature verification, external calls signed by the operator behave identically to those initiated by the owner.

2.2.2 2-step Ownership Transfer

The wallet implements a 2-step ownership transfer where the owner or the operator can propose a new owner by calling `proposeOwner()` and in the second step the new owner can accept the ownership by calling `acceptOwnership()`. Moreover, the owner can call `renounceOwnership()` which sets the owner of the wallet to the zero address.

2.2.3 Ownership Inheritance

- The owner or the operator can set up the inheritance mechanism by calling `updateInheritanceSetup()` where they specify the hash of the heir's address and the inheritance delay.

- If the wallet does not execute any call for the time period of the inheritance delay, the heir can call `inheritOwnership()` and become the new owner. After that the heir and the inheritance delay are reset.

Note that the inheritance mechanism is optional.

2.2.4 Social Recovery Mechanism

Ownership of the Unstoppable Wallet may also be transferred via the social recovery mechanism. While not in recovery mode, the owner or the operator can define a set of guardian address hashes (`[add|remove]Guardian()`) which form a quorum to propose a new owner. The lifecycle of a proposal is as follows:

- `initiateRecovery()`: at any point in time, a guardian can propose a hashed address of a new owner and set the wallet to recovery mode. There can only be exactly one proposal for a specific new owner during a recovery period. A guardian can make a new proposal every `s_recoveryRecreationDelay` seconds. When making a new proposal their previous proposal is deleted.
- `supportRecovery()`: other guardians can support an already existing proposal. If a minimum number (`s_recoveryMinConfirmations`) of guardians agrees on the same owner, the new owner can claim the ownership after `s_recoveryMinConfirmationsDelay` seconds. Any additional vote for a specific proposal further decreases the required waiting time in a linear fashion. If all guardians agree on the same owner then the ownership becomes claimable after `s_recoveryFullConfirmationsDelay` seconds. The wallet tracks the guardian's votes by populating the `s_recoveryProposalConfirmations` mapping.
- `recoverOwnership()`: after the required delay time has elapsed, the new owner can call to claim the ownership. The wallet exits the recovery state and all proposals and votes are deleted. Moreover, any pending proposed owner is deleted.
- `cancelRecovery()`: can be executed by the owner or the operator to switch off the recovery mode and delete all pending proposals.
- `updateRecoverySetup()`: while the wallet is not in recovery mode the owner or the operator can specify the minimum required confirmations for a social recovery, the delay when these confirmations are given and the delay if all guardians agree on a proposed owner.

Note that the social recovery mechanism is optional. The default setting of social recovery requires only 2 guardians to be set. Moreover, by default there's no delay if all guardians agree on a new owner (instant ownership transfer possible). Moreover, there are no constraints for the minimum and the full confirmation delay.

2.2.5 Supported Standards

The wallet supports the following standards:

- EIP-1271 (`isValidSignature()`)
- EIP-721 (`onERC721Received()`)
- EIP-1155 (`onERC1155[Batch]Received()`)

2.2.6 Trust Model and Assumptions

Besides the owner and the operator who are in full control of the funds, the wallet defines the guardian role which is granted by the owner/operator. The guardians are not necessarily fully trusted. The owner/operator can optionally define a delay (`s_recoveryFullConfirmationsDelay`) to allow them to cancel a proposal even if all guardians collude to propose a new owner. The configuration of the social recovery directly correlates with the trust of the owner in the guardians.

2.2.7 Changes in Version 2

In Version 2 the following changes are introduced:

- All owner/operator actions, all actions from the new owners e.g., `acceptOwnership`, and all ownership transfers of the wallet update the last action timestamp, resetting the countdown for inheritance.
- The social recovery countdown for a specific proposal starts as soon as a proposal is created, as opposed to Version 1 of the code where it would start as soon as the minimum number of confirmations were collected.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Social Recovery With Less Than minConfirmations Possible Code Corrected

Medium-Severity Findings	2

- Inconsistent Computation of claimableAt Timestamp Code Corrected
- Ownership Transfer Race Conditions Code Corrected

Low-Severity Findings	4

- Active Recovery Proposals Transferred to New Owner Code Corrected
- Resetting s_verifyContract Code Corrected
- Setting a Single Guardian Code Corrected
- Updating Operator's Public Key Specification Changed

Informational Findings	5

- Event Indexing Code Corrected
- Failing Early Code Corrected
- Floating Pragma Code Corrected
- Gas Inefficiencies Code Corrected
- staticcall Succeeds on EOAs Code Corrected

6.1 Social Recovery With Less Than minConfirmations Possible

Correctness High Version 1 Code Corrected

CS-UNSTW-009

The `supportRecovery()` function allows guardians to support a social recovery proposal, thereby increasing the proposal's number of confirmations. The social recovery mechanism is specified to work as follows:

- Ownership of the wallet can be recovered if the respective proposal has at least `s_recoveryMinConfirmations` confirmations from guardians.
- Transferring the ownership of a wallet via social recovery enforces a delay. The ownership is only claimable through `recoverOwnership()` after the delay has passed.
- If a proposal has exactly the minimum number of confirmations, a delay of `s_recoveryMinConfirmationsDelay` should be incurred.

- If all guardians confirm a proposal, a delay of `s_recoveryFullConfirmationsDelay` should be incurred (note: the full confirmations delay must be less or equal to the minimum confirmations delay)
- For any number of confirmations between `s_recoveryMinConfirmations` and the total number of guardians, the delay should be linearly interpolated between `s_recoveryMinConfirmationsDelay` and `s_recoveryFullConfirmationsDelay`.

The current implementation of the `supportRecovery()` function enables social recovery, even if less than `s_recoveryMinConfirmations` guardians confirm. Consider the number of registered guardians to be 10 and the minimum confirmations required to be 6. The first guardian calling the `supportRecovery()` function on a given proposal will increase the `confirmationsCount` to 2 which should not be enough to execute social recovery. The following code snippet shows how the timestamp at which ownership becomes claimable is set. The second confirmation will trigger the else branch which falsely sets the incurred delay to less than `s_recoveryMinConfirmationsDelay`. Essentially, with less than the minimum number of confirmations, the delay is smaller than with the threshold number of confirmations. This is a violation of the intended behavior of the social recovery mechanism.

```
if (s_recoveryProposal.confirmationsCount == s_recoveryGuardians.length()) {
    s_recoveryProposal.claimableAt = Math.min(s_recoveryProposal.claimableAt,
                                              block.timestamp + s_recoveryFullConfirmationsDelay);

} else if (s_recoveryProposal.confirmationsCount == s_recoveryMinConfirmations) {
    s_recoveryProposal.claimableAt = block.timestamp + s_recoveryMinConfirmationsDelay;

} else {
    // What percentage of the guardians confirmed?
    uint percConfirmed = s_recoveryProposal.confirmationsCount * 100 / s_recoveryGuardians.length();
    // Calculate the delay based on the above percentage.
    // The more confirmations, the less delay
    uint recoveryDelay = (s_recoveryMinConfirmationsDelay - s_recoveryFullConfirmationsDelay)
                         * (100 - percConfirmed) / 100;
    s_recoveryProposal.claimableAt = Math.min(s_recoveryProposal.claimableAt, block.timestamp + recoveryDelay);
}
```

Code corrected:

The case where:

```
s_recoveryProposal.confirmationsCount < s_recoveryMinConfirmations
```

is handled separately and does not set `claimableAt`. Therefore, the recovery countdown does not start in this case. Note that in the new version, when the number of confirmations is above the minimum required, the delay depends on the following term:

```
uint delayToSubtract = totalTimeBetweenMinimumAndFull / numberOfGuardiansAboveMinimum * numberOfGuardiansAboveMinimumThatSupported;
```

This term introduces a rounding error due to division before multiplication. However, the error is negligible in practice.

6.2 Inconsistent Computation of `claimableAt` Timestamp

Correctness **Medium** **Version 1** **Code Corrected**

CS-UNSTW-003

In the case where fewer than all guardians confirm a proposal, the time at which the ownership of a wallet is claimable through social recovery is computed as follows:

```
uint recoveryDelay = (s_recoveryMinConfirmationsDelay - s_recoveryFullConfirmationsDelay)
                        * (100 - percConfirmed) / 100;
s_recoveryProposal.claimableAt = Math.min(s_recoveryProposal.claimableAt,
                                          block.timestamp + recoveryDelay);
```

The computation has the following issues:

- It is missing the offset of `s_recoveryFullConfirmationsDelay`. As such, depending on the value of `recoveryDelay`, the delay incurred by not all guardians supporting the proposal might be less than when all guardians confirm the proposal.
- In case `s_recoveryMinConfirmationsDelay = s_recoveryFullConfirmationsDelay`, the incurred delay will be 0.

Code corrected:

`supportRecovery` correctly updates `claimableAt` when a guardian supports a proposal that already has at least the minimum number of confirmations required for social recovery.

6.3 Ownership Transfer Race Conditions

Design | Medium | Version 1 | Code Corrected

CS-UNSTW-010

The `s_lastActionTimestamp` is set whenever `_executeCall()` successfully executes. The timestamp of the last action is used to determine when the heir can claim the wallet. Ownership inheritance is allowed as soon as `s_inheritanceDelay` seconds after the last action timestamp has been set has passed.

Currently, it is possible that the heir of the wallet inherits the ownership just after a transfer of ownership has happened through another mechanism. The race conditions between the different ownership transfer mechanisms is due to the `s_lastActionTimestamp` not being reset when ownership is transferred through the social recovery or the propose-accept mechanism.

Code corrected:

Whenever an action is executed by the owner/operator or ownership of the wallet is transferred, `_updateLastActionTimestamp()` is called which updates the `s_lastActionTimestamp`. This eliminates the race conditions described.

6.4 Active Recovery Proposals Transferred to New Owner

Design | Low | Version 1 | Code Corrected

CS-UNSTW-008

The active recovery proposals kept in the wallet's storage are not reset upon ownership transfer. Despite the new owner being able to call `cancelRecovery()` in order to remove the ongoing social recovery processes, this puts an extra burden on the new owner. Imagine the following scenario:

- Bob inherits Alice's wallet with an active recovery proposal that is close to being claimable.
- Bob is not aware of the ongoing recovery process and does not call `cancelRecovery()`.
- Bob may lose access to the wallet right after inheriting it due to the ongoing recovery process.

Code Corrected:

`acceptOwnership()`, `inheritOwnership()` and `recoverOwnership()` correctly reset the social recovery mechanism by setting the recovery state to `INACTIVE` and removing all active proposals.

6.5 Resetting `s_verifyContract`

Design | Low | Version 1 | Code Corrected

CS-UNSTW-012

The owner of the contract can specify additional logic to limit certain calls made by the operator. The additional logic is stored in `s_verifyContract`. However, the operator is not able to modify this address. In case the owner renounces the ownership, a bug in `s_verifyContract` could lead to the wallet being stuck.

Code corrected:

`renounceOwnership()` now reverts if `s_verifyContract` is set.

6.6 Setting a Single Guardian

Design | Low | Version 1 | Code Corrected

CS-UNSTW-013

An operator or an owner can specify the minimum number of required confirmations to initiate the social recovery by calling `updateRecoverySetup()`. Note that the function doesn't enforce a minimum number for this value. Therefore a user could set this to 0 or 1 and specify a single guardian. In this case, the recovery process can be initiated by this guardian by calling `proposeOwner()`. However, such proposal would never be claimable. Therefore, the ownership of the wallet is not recoverable through social recovery with 0 or 1 guardians.

Code corrected:

The `updateRecoverySetup()` function has been updated so that `_minConfirmations` and consequently `s_recoveryMinConfirmations` must be at least `MIN_CONFIRMATIONS_LOWERBOUND`. Currently, `MIN_CONFIRMATIONS_LOWERBOUND` is set to 2, ensuring the social recovery mechanism is not inherently blocked.

6.7 Updating Operator's Public Key

Security | Low | Version 1 | Specification Changed

CS-UNSTW-002

Transferring the ownership of the wallet happens in two steps. The owner needs to first `proposeOwner()` and then the new owner should `acceptOwnership()`. However, `updateOperatorPublicKey` is performed in one step. Note that the update can happen without an owner being set. Therefore a wrong update could lead to loss of funds.

Specification changed:

The modifier of `updateOperatorPublicKey()` has changed to `onlyOwner`. This prevents an operator from incorrectly updating the operator public key and therefore losing access to the wallet, i.e. only the owner can update it and thus in case of an incorrect update the owner can simply update it again.

6.8 Event Indexing

Informational | Version 1 | Code Corrected

CS-UNSTW-006

The `OwnerUpdated` event emitted upon ownership transfer of the wallet does not index the addresses of the old and the new owner.

Indexing the above-mentioned fields may be useful for filtering events.

Code corrected:

Indexing was added accordingly.

6.9 Failing Early

Informational | Version 1 | Code Corrected

CS-UNSTW-004

When executing calls, `_executeCalls()` checks whether the target address of a call contains code and reverts if it doesn't. Note that the revert doesn't depend on the result of the call. As a matter of fact, a call to a target without code will always succeed. Therefore, `_executeCalls()` could fail earlier saving some gas.

Code correct:

The code has been restructured so failure happens early.

6.10 Floating Pragma

Informational | Version 1 | Code Corrected

CS-UNSTW-005

The contracts use a floating solidity pragma: ^0.8.22. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Note that for interfaces a less restrictive version could be used so that third parties can easily reuse the interfaces.

Code correct:

`pragma solidity` is fixed to version 0.8.28.

6.11 Gas Inefficiencies

Informational | Version 1 | Code Corrected

CS-UNSTW-011

Some operations consume more gas than necessary. Below is a non-exhaustive list of gas inefficiencies:

1. `Wallet.updateInheritanceSetup()` and `Wallet.updateRecoverySetup()` perform multiple storage loads of the same slots that could be avoided.
2. `s_recoveryRecreationDelay` could be an immutable given it's only set during construction.

Code corrected:

1. `updateInheritanceSetup()` and `updateRecoverySetup()` cache storage values.
2. `RECOVERY_PROPOSAL_RECREATION_DELAY` constant set to 1 day has replaced the state variable.

6.12 `staticcall` Succeeds on EOAs

Informational | Version 1 | Code Corrected

CS-UNSTW-007

When called by the operator, `executeCalls()` executes some arbitrary logic stored in `s_verifyContract` address. If the address doesn't contain code then the static call will succeed. In case the owner has set the address wrongly, the operator could never realize it giving a false sense of security.

Code corrected:

A check is added to ensure that `s_verifyContract` is not an EOA.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Instant Inheritance

Informational **Version 1** **Acknowledged**

CS-UNSTW-001

`updateInheritanceSetup()` doesn't enforce a minimum inheritance delay. Setting the delay to 0 therefore makes inheritance equivalent to `proposeOwner()`.

Acknowledged:

Unstoppable responded:

> Technically correct but no sensible minimum can be defined that covers correctly all potential use cases. Therefore up to the user to decide.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Challenges When Deploying a Contract to the Same Address

Note Version 1

Users might wish to have the same address for their wallet on all EVM-compatible chains.

However, certain networks and deployment strategies may pose challenges to this requirement:

1. Optimism-stack-based chains: when bridging funds to an OP chain, the nonce of the `from` account is incremented. See https://specs.optimism.io/protocol/deposits.html#nonce-handling. This should be taken into account in cases where the deployer of the deployer contract (`WalletFactory`) has bridged funds. This would increase their nonce and thus affect the address of the factory contract as the address depends on the nonce of the deployer. Consequently, the address of the token contract will be also affected as it depends on the address of the CREATE2 caller (which is the factory contract). Affected chains: Optimism, Base, etc.
2. Since the same EOA needs to be used for deploying the deployer on all the networks, the private key of the EOA must be used to sign multiple transactions. Factors such as bad setup, insider threat, malicious code or infrastructure, bad key generation, etc. may lead to the private key leak. It is important that the deployer of deployer contract does not have any special permissions and does not have access to funds.
3. `zkSync Era`: CREATE_PREFIX used in zkSync is different from the mainnet one. In addition, to use CREATE2 in a deployer contract, zkSync requires that the deployed contract bytecode is already known to the compiler as states here.

8.2 Maximum Number of Guardians

Note Version 1

There is no limit to the maximum number of guardians that can be defined. Each guardian could initiate a new social recovery proposal. All proposals are deleted in case one of the proposals passes the wallet, is inherited, or the owner explicitly cancels social recovery. However, if the number of proposals is too big, the gas cost for deletion could exceed the block gas limit, essentially blocking the wallet. Unstoppable mentioned that there's an upper bound enforced off-chain (12-16 guardians). In practice, the scenario described above is highly unlikely.

8.3 State After Ownership Transfer

Note Version 1

Users should be aware that upon ownership transfer, most of the wallet's state is not reset. More specifically, the following remain unchanged:

- The verification contract for external calls

- The operator's public key
- The heir (if ownership is transferred via social recovery or a 2-step transfer)
- Active social recovery proposals (if ownership is transferred via a 2-step transfer)
- Registered recovery guardians
- The operator and owner storage

This can lead to unexpected behavior if the new owner makes assumptions on the wallet’s existing state.