Zeal P256 Security Audit

Table of Contents

Secure Technical Solutions LLC

Summary

Zeal P256

The Zeal wallet enables users to interact with the blockchain from their web browser. The wallet is compatible with many different EVM-compatible chains and is fully self-custodial. The new feature for the wallet that was reviewed in this security audit was Zeal's recovery feature, leveraging WebAuthn/FIDO2. This feature allows a user to recover their wallet using their WebAuthn-supported device, such as an Apple device or Yubikey. The focus on the audit was the on-chain components of the system, which included a Factory contract that serves as a database of user WebAuthn public keys and a Signer contract that is added as an owner of the Zeal Safe wallet's multisig. Because WebAuthn uses the sec256r1 elliptic curve, instead of the secp256k1 elliptic curve normally used in the Ethereum ecosystem, a specialized library that implements efficient on-chain support for sec256r1 elliptic curve operations is a dependency of this project.

Testing Assumptions

The primary focus of this security review was the custom Zeal smart contracts that support the Safe wallet recovery flow. In the end-to-end WebAuthn flow, several components outside of the core smart contracts scope were still considered. These components included the formal specifications for the WebAuthn protocol, the Zeal wallet monorepo at commit hash 72ba95bee80ed5d100d678b07bfc7f51aac4828f, and the open source Ledger Security Key app.

Because the Zeal Safe wallet and recovery is a new feature, not all aspects of this new feature were implemented yet. For example, it was not possible to use development version 0.3.49 of the Zeal wallet to sign a message when prompted by dApps - instead, a "not implemented" error message was received. The wallet also did not properly detect whether a

security key supported FIDO2 (instead of only FIDO), which allowed a Safe wallet to be created with an unsupported security key but then the recovery process would fail.

Due to the scoping of this security audit, the implementation details of the cryptography libraries were mostly out of scope. This included the signature verification process, which is a key part of the WebAuthn process, but it is implemented by an external library.

The primary focus was on the Zeal contracts in the P256Recovery repository. The repository was reviewed at commit hash 68b8c43d8efb5cdbd3a12796cc944a785ddd7079 which corresponded to extension development version 0.3.49. The Chromium web browser version that was used during this testing was version 112.0.5615.165. The security audit was performed by one security engineer over 7 days.

Testing Methodology

Given the architecture of the Zeal wallet solution, the following list of key security areas were considered:

- Architecture: design of the overall flow for this recovery process, including the initial gnosis safe setup and later recovery
- Cryptography: math involving elliptic curves (public key storage mechanism, signature verification, etc.)
- Smart contracts: weaknesses in any contract implementation details
- Zeal wallet extension: incorrect implementation details in the Zeal wallet WebAuthn implementation

Testing of the frontend (AKA the Zeal wallet extension) was not part of the scope of this audit, because the effort was focused on the on-chain contracts. Some issues that were identified while reviewing WebAuthn code of the frontend were written up as informational out-of-scope findings.

Secure Technical Solutions LLC makes no warranties regarding the security of the code and provides no guarantee that the code is free from defects. The authors of this report do not represent nor imply to third parties that the code has been fully audited nor that the code is free from defects. By using this code, Zeal wallet users agree to use the code at their own risk.

High Findings

1. High - `signerData` mapping value can be overwritten

Summary

The Safe recovery process involves an on-chain lookup of the `signerData` mapping to identify the public key and Signer contract associated with a given hash (AKA user.id or user handle). It is possible to overwrite the value in this mapping, and after a value is overwritten, the original user's public key and Signer will no longer be available on-chain.

Vulnerability Detail

This line of the `Factory._deploy()` implementation does not check if it is overwriting an existing value. If an existing value is overwritten, the original user will silently lose the ability to recover their wallet. An attacker can modify the Zeal wallet extension to send a specific user.id value to their own authenticator device, and the public key that will be stored at this mapping location will use the attacker's public key. Using a new public key will allow the verification check to pass, because the public key corresponds to the private key in the attacker's authenticator device, but will overwrite the existing data of this mapping slot. A new Signer address will also be created and used because the salt used to deploy the Signer contract depends on the public key values.

Recommendation

To enable the on-chain data to remain trustworthy and prevent overwriting, check whether `signerData[_hash]` is zero before writing to this mapping slot and revert if there this mapping slot is non-zero.

However, in the case that the mapping slots become "read only", it's possible this issue could turn into a griefing attack in the case where an attacker can frontrun the process of writing to the storage slot with different data, causing the user's tx to revert because that mapping slot is non-zero. But there would be a cost to sustaining this griefing attack so it is unlikely to be a major issue.

Developer response

Acknowledged, fixed in commit 819196af462c7c3c39a7e9bc0bde13e96b40a5ed. The mapping was removed from the factory and added to a new child contract, FactoryRef, which

will only be deployed on Gnosis chain. Deployment of new signers will now revert if the mapping value already exists.

Medium Findings

1. Medium - Different Factory deployments can contain conflicting recovery data

Summary

The Factory.sol contract is intended to serve as a lookup database for the `_hash` user identifier. This identifier determines which Safe address and which Signer address the Zeal wallet extension should interact with in order to enable a user to regain access to their recovered Zeal Safe address. But there is a different Factory deployment on every chain, and even a single chain can have multiple Factory deployments (due to bug fixes, new features, etc.). If different Factory deployments contain conflicting data for the same `_hash` value, the Zeal recovery system has no way to guide the user on which Safe is the correct one. An attacker can fill other Factory deployments with spam data to make the recovery process more difficult for a user.

Vulnerability Detail

When a user creates a new Zeal Safe wallet, the wallet Signer and the public key is only stored in the Factory.sol SignerData mapping after the first tx on that chain. Because of this, there may only be one Factory.sol deployment on one chain that contains the data related to the user's Zeal Safe wallet. Therefore, the wallet recovery process must query all deployments of the Factory.sol contract on all chains to find non-zero mapping slots that contain data for this wallet stored in any Factory deployment. When there are multiple versions of Factory.sol deployed on the same chain, then these different Factory versions must all be checked, because only one Factory.sol version should contain the user's Safe wallet data.

The issue is due to the fact that an attacker can provide arbitrary data into the SignerData mapping at specific mapping slots. The attacker can insert their own public key data into the same mapping slot in the Factory contract of other chains. The attacker can store multiple conflicting results for the same `_hash` user identifier on the Factory.sol deployment of different chains. This means this attack is possible even after the high issue that involves overwriting `signerData` is resolved, as long as the existing `_hash` values are known to the attacker. By injecting data at the same `_hash` of the mapping into Factory contracts on other chains (or even the same chain if there are different versions of the Factory contract on one

chain), the attacker can make it very difficult for the user to recover their actual account because there will be many “spam” data entries in the recovery databases (each Factory contract on each chain can be thought of as a separate database of wallet public keys). If overwriting data is prevented (fixing the high issue in this report), then the impact of this situation is primarily a griefing attack, where the user will have a difficult time recovering the correct Safe wallet if they have no recollection of which Safe address is theirs.

The issue stems from the fact that the WebAuthn system uses discoverable credentials with the user.id value (AKA user handle) as the unique identifier. But when looking up the public key corresponding to this user.id identifier, there are multiple “databases” or Factory.sol contracts that may contain conflicting data about which Zeal Safe address corresponds to the user.id identifier. The result is that the user will be prompted with many different EVM addresses that they could recover, but if they don’t know which address to “recover” because they forget their wallet address, they will have a trial-and-error process of determining which address the private key in their WebAuthn-supported device actually gives them access to.

Recommendation

Ideally an off-chain solution should be implemented in the Zeal wallet backend to prevent on-chain manipulation of the recovery data. If an off-chain solution is implemented in a way that still relies on on-chain data, it is still possible to manipulate the on-chain data and effectively bypass the off-chain checks.

It is possible to use an on-chain solution to prevent multiple entries for a single user.id (AKA user handle) identifier on a single chain, but there is no obvious cross-chain solution. To resolve this issue on a single chain, each Factory should be versioned. When the newest Factory version has a new entry added to the `signerData` mapping, the factory should call a function in the previous factory version. The function in this older factory version should check if that mapping slot is non-zero in its own `signerData` mapping and then call a function in the previous factory version. The process repeats recursively until the first factory version is checked, confirming that no previous factory version has data stored in this mapping slot.

An alternative to this “linked list” approach to Factory contracts on a single chain is to have one external storage contract that stores the mapping data. The different Factory contract versions can all submit data to this external contract for storage, so the Factory contract does not store the values directly.

However, the problem with this recursive linked list approach or the external storage contract approach is the cross-chain solution. Assume chain A and chain B have Factory versions 1

and 2 deployed. If a user only transactions on chain A with Factory version 1, then an attacker could add data to Factory version 1 or 2 on chain B, because there is no way for contracts to query data on other chains.

Developer response

Acknowledged, fixed in commit 819196af462c7c3c39a7e9bc0bde13e96b40a5ed. The mapping was removed from the factory and added to a new child contract, FactoryRef, which will only be deployed on Gnosis chain. There is only one reference chain now, so the data can't be conflicting.

Low Findings

1. Low - Disable initializers in Signer.sol

Summary

OpenZeppelin's Initializer.sol suggests avoiding uninitialized contracts whenever possible, which is not done in Signer.sol.

Vulnerability Detail

Signer.sol does not call `_disableInitializers()` in its constructor, meaning that the initializer in the implementation contract (not the same as the proxy contract) could be taken over by anyone. There is no obvious impact to someone else initializing this contract, but because there is only one Signer.sol implementation that is used by all Safe wallets, it is best to make sure this contract cannot be initialized by an arbitrary address.

Recommendation

Should.sol should include the code found in the comment of OpenZeppelin's Initializer.sol implementation:

```
constructor() {
    _disableInitializers();
}
```

Developer response

Acknowledged, fixed in commit 8bab8caf4c98bb208bc02584ceec9f87760c679e

2. Low - Cryptographic challenge is not generated securely on the server side

Summary

Section 13.4.3 Cryptographic Challenges of the WebAuthn standard states that cryptographic challenges "MUST be randomly generated by Relying Parties in an environment they trust (e.g., on the server-side)". Because of how this WebAuthn system is implemented, the server-side is the user's browse, which is actually client-side. This means that this requirement is not strictly followed as written in the specification, potentially increasing the risk of replay attacks.

Vulnerability Detail

During the recovery process for a Zeal Safe wallet, the authentication device must sign a randomly generated challenge during the authenticatorGetAssertion operation, which is used to prove that the authentication device has certain private keys. Using a randomly generated challenge prevents the authentication device from using an old signature (replay attack). There are at least two risk vectors with the current design:

1. The random challenge may not be using strong randomness, because generating the challenges on the client-side offers less guarantees about the randomness of challenges.
2. There is no server-side limit to the number of random challenges that can be issued to the authenticator device and signed. However, issuing these challenges to the authenticator device implies some form of digital, or more likely physical, access to the authenticator device and is unlikely to happen by an unauthorized party.

Recommendation

It is hard to give a good recommendation to this issue given the current design of the WebAuthn end-to-end implementation. Ideally there should be some guarantee that the random challenge is sufficiently random.

Developer response

Acknowledged, won't fix. These risks are accepted due to the opinionated manner in which Zeal uses Webauthn for web3 purposes. The Zeal extension makes Webauthn requests in 3 scenarios:

1. Creating a passkey to be associated with a wallet: Since the challenge during registration is unused unless doing attestation, we do not verify the challenge. This is consistent with other wallet implementations that use constant string values as their challenge.
2. Recovering a wallet from a passkey: In this case, the Webauthn request is purely made to retrieve the userHandle of the credential chosen by the user. The challenge is randomly generated using crypto.getRandomValues(), but not verified since it is not an authentication ceremony. The only risk would be the attacker getting access to another

user's userHandle, which is used to predict their smart wallet address. It does not allow them to transact on the user's behalf. The userHandle (recoveryId) is already publicly stored on-chain.

3. Signing a user operation: Unlike Web2, where the challenge is generated securely on the server, and the response is validated against that generated challenge, our use case is solely for signing messages or crypto transactions (user operations). In this case, the challenge is not randomly generated; instead, the user operation hash is used as the challenge. The challenge does not get verified on the browser extension since the P256 on-chain validator contract verifies the signature against the user operation hash.

Gas Saving Findings

1. Gas - Remove duplicate code

Summary

There is duplicate code in SignerProxy.sol for returning the implementation address.

Vulnerability Detail

In SignerProxy.sol, the fallback function and `implementation()` return the implementation address when `implementation()` is called. Removing the separate `implementation()` function keeps the contract more similar to the SafeProxy.sol implementation in the Safe and saves the most gas.

Recommendation

Remove the `implementation()` implementation in SignerProxy.sol and rely only on the fallback function.

Developer response

Acknowledged, this was removed in commit 879b2f446539c22a414e28ec5e88b31ffec3430b

2. Gas - Remove `_implementation` in `_deploy()`

Summary

The `_implementation` value passed to `_deploy()` is immutable, so there is no need for this arg.

Vulnerability Detail

It is more gas efficient, at least for Factory.sol deployment, for `_deploy()` to cache `_implementation` locally instead of receiving it as a function argument.

Recommendation

Modify `_deploy()` as follows. Remember to modify the call to `_deploy()` to remove the unnecessary argument:

```
-    function _deploy(address _implementation, bytes32 _hash, uint256 _x, uint256 _y)
+    function _deploy(bytes32 _hash, uint256 _x, uint256 _y)
        internal
        returns (address signer)
    {
+        address _implementation = IMPLEMENTATION;
```

Developer response

Acknowledged, fixed in commit c92c8c0b226ea41dacdfa5a312d6f8d2d1a1ffe2

3. Gas - Remove unused return value in `_deploy()`

Summary

`_deploy()` is called in only one place, and the return value is ignored in this one place.

Vulnerability Detail

`_deploy()` in Factory.sol returns the signer address, but this return value is ignored in the one place where the function is called.

Recommendation

Remove the return value from `_deploy()` to save a small amount of gas when Factory.sol is initially deployed and also when `deploy()` is called.

```
    function _deploy(address _implementation, bytes32 _hash, uint256 _x, uint256 _y)
        internal
-        returns (address signer)
    {
        bytes32 salt = checkCaller(_implementation, _hash, _x, _y);

-        signer = address(_deploySigner(_implementation, salt));
+        address signer = address(_deploySigner(_implementation, salt));
        Signer(signer).initialize(_x, _y);
```

Developer response

Acknowledge, fixed in commit 17152b532b8f4df6bf24446e7f2a56a80b5b80a4

4. Gas - Remove `isValidSignature(bytes32,bytes)` from Signer.sol

Summary

There are two `isValidSignature()` functions in Signer.sol and one can be removed.

Vulnerability Detail

The Factory.sol contract has hardcoded Safe contract addresses. These addresses are for Safe version 1.3.0, which only calls `isValidSignature(bytes,bytes)` in the Signer.sol contract. The other `isValidSignature(bytes32,bytes)` function was added in PR 581 for the 1.5.0 release and therefore is not needed if hardcoded Safe 1.3.0 contracts are the only interaction.

Recommendation

Remove `isValidSignature(bytes32,bytes)` from Signer.sol.

Developer response

Acknowledged, won't fix

Informational Findings

1. Info - Consider adding foundry fuzz tests

Summary

The foundry.toml configuration file did not have fuzz tests enabled.

Vulnerability Detail

Using even the default foundry fuzz test configuration settings could improve the security assurances of the code. Modifying the tests in Signer.t.sol to accept x and y as parameters will further confirm that the functions in that contract are properly implemented. Using fuzz testing would primarily provide fuzzing of the external FCL WebAuthn library, and it may be even more useful to perform some differential fuzzing of the FCL WebAuthn implementation with another similar implementation> This approach to differential fuzzing is how the cryptofuzz project finds bugs in various cryptographic libraries.

Recommendation

Consider adding foundry fuzz tests to the test suite.

Developer response

Acknowledged, adding additional fuzz tests to the Signer tests will be difficult, as they depend on predefined signatures; the current failing (paths) tests relies on fuzzing, and we will aim to expand our fuzzing tests where possible.

2. Info - Consider fork testing on all relevant chains

Summary

The intention is to allow this set of contracts to be deployed on many different chains. Fork testing should be performed on all supported chains to confirm there are no compatibility issues with certain chains.

Vulnerability Detail

While the foundry tests provide good test coverage, there is no indication that tests are run on a fork of every chain supported by this design. Fork testing is preferred because it will confirm that the hardcoded addresses are working as expected, and that any differences in the virtual machine of that specific chain do not impact the implementation of this design.

Recommendation

Add fork testing on all supported chains to the testing process. This does not have to happen on every commit in CI like in a github action, but should be performed before any new on-chain release.

Developer response

Acknowledged, won't fix. We manually run tests across chains that are relevant to zeal/ where we look to deploy the contracts; we can add this and info regarding chains tested into the README

3. Info - Improve test coverage

Summary

The test coverage of the contracts can be improved.

Vulnerability Detail

While the test coverage from the existing tests are good, there are still some functions that are not tested at all. For example, `Factory.getSafeAddressbyHash()`, `Signer.isValidSignature(bytes32,bytes)`, and `SignerProxy.implementation()` are not tested. Note that `SignerProxy.implementation()` should be removed entirely for gas savings, as noted in a separate finding.

File	% Lines	% Statements	% Branches	% Funcs
src/Factory.sol	92.86% (39/42)	91.94% (57/62)	66.67% (4/6)	91.67% (11/12)
src/IntermediateCaller.sol	100.00% (1/1)	100.00% (2/2)	100.00% (0/0)	100.00% (1/1)
src/Signer.sol	83.33% (10/12)	84.62% (11/13)	100.00% (2/2)	83.33% (5/6)
src/SignerProxy.sol	66.67% (2/3)	0.00% (0/1)	50.00% (1/2)	50.00% (1/2)

Recommendation

Improve test coverage, ideally achieving 100% test coverage.

Developer response

Acknowledged. Current testing covers critical functionality and missing tests generally have equivalent tests (e.g., `Signer.isValidSignature(bytes32,bytes)`) or view functions tested outside the smart contract repo. We will, however, strive to include these tests so that we do not rely on external assumptions about testing.

4. Info - Wallet creation possible with unsupported wallet

Summary

It is possible to use a passkey to create an on-chain Safe with the Zeal wallet using an unsupported FIDO device. The problem is that this device could not later be used for the wallet recovery process, which means that when the feature is most needed, it would be unavailable to the user.

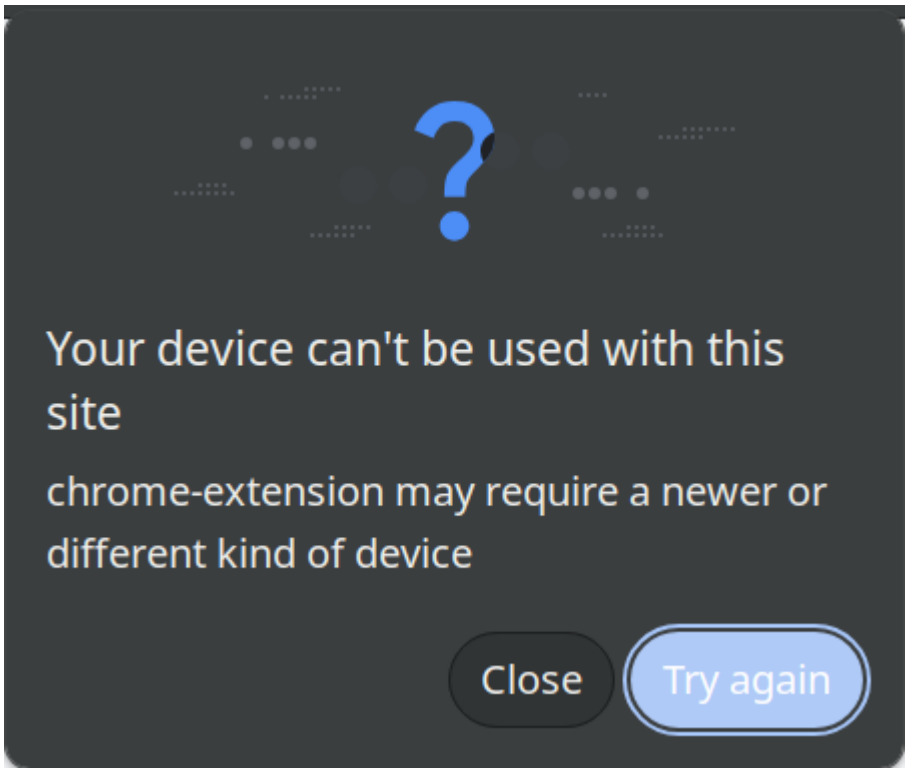
This issue was outside the scope of this audit and is therefore considered only Informational, despite the potential loss of wallet access.

Vulnerability Detail

Using a Ledger Nano X with firmware 2.2.3 and the FIDO U2F app installed from the Ledger Live app catalog, it was possible to create a Safe wallet with Zeal dev wallet version 0.3.49. The problem was that this setup did not support FIDO2 or WebAuthn, so the recovery of this wallet would fail. This scenario would result in a user believing that they would have recovery access to their wallet, but later discovering that they could not recover the wallet.

The same process happens with a Ledger Nano X using the Security Key app if the resident keys feature has not been enabled in the app settings.

The error message that appears for the user when they attempt to recover their wallet with an unsupported device states “Your device can’t be used with this site”.



Recommendation

Unsupported FIDO devices should not be allowed to create a Zeal Safe if they are unable to recover it later. Only devices that can successfully perform the recovery step should be allowed to create a Safe wallet and perform the transaction that triggers the deployment of the Safe code on-chain.

Additionally, add a way for users to test the recovery process of their device to confirm that it will work as expected.

Developer response

Acknowledged, fixed in commit 303ea6318b3534a3769bdadd2f1cdf15d9ccb97a This problem arises from creating non-discoverable credentials, which are not supported during the recovery process. To address this, we have updated the create request to include the following settings:

```
authenticatorSelection: {
    requireResidentKey: true,
```

```
    residentKey: 'required',
}
```

According to the WebAuthn specification, these settings ensure that only discoverable credentials are created, thereby preventing the use of unsupported FIDO devices that do not support FIDO2 or WebAuthn for wallet recovery. This change mitigates the risk of users being unable to recover their wallets due to using unsupported devices, ensuring a more reliable and secure wallet recovery process.

5. Info - Remove unnecessary code

Summary

Some code declares errors and events that are never used.

Vulnerability Detail

Remove:

1. the unused NewFactorySetup event in Factory.sol.
2. the unused ProxyNotDeployed error in Factory.sol, declared in the FactoryErrors library

Recommendation

Remove unused code in Factory.sol to reduce overall complexity.

Developer response

Acknowledged, has been fixed across various commits.

6. Info - Improve variable and function naming

Summary

Some variable and function names could be improved for consistency and clarity.

Vulnerability Detail

Rename:

1. `checkCaller()` to `_checkCaller()` because it's an internal function
2. `isContract()` to `_isContract()` because it's an internal function
3. `IMPLEMENTATION` to `IMPLEMENTATION_SIGNER` to clarify which contract implementation this variable stores

4 `hash` in Factory.sol to `recoveryId`, `userId`, `userHandle`, or a similar name that is more descriptive. This change should only be applied to Factory.sol and not to Signer.sol, because `hash` has a different meaning in Signer.sol

5 `SingerErrors` library in Signer.sol to `SignerErrors` to fix a typo

Recommendation

Rename variables to clarify their purpose.

Developer response

Acknowledged, has been fixed across various commits.

7. Info - No differentiated user flow for device-bound devices

Summary

There is only one WebAuthn signer associated with each Zeal Safe wallet. The goal is to allow the user to recover their wallet using this WebAuthn signer in the case they lose access to their device(s). The current wallet user flow does not differentiate between a device-bound WebAuthn device and a cloud-based WebAuthn (Passkey) device. If a user sets up their recovery signer with a device-bound WebAuthn device, that device is the only way that a user can recover their Zeal wallet.

Vulnerability Detail

Passkey is the terminology used in parts of the Zeal Safe recovery solution, but the term Passkey refers specifically to the implementation of WebAuthn that involves a cloud solution storing the private keys that enable recovery. Devices like yubikey and ledger will work fine with the Zeal wallet recovery feature, but technically they are using WebAuthn, not Passkey. Passkey is backwards compatible with WebAuthn, but unlike the Apple/Google Passkey solutions, losing that specific yubikey or ledger renders the wallet recovery feature useless, because there is no way to extract the private key for a backup in device-bound WebAuthn devices like ledger and yubikey. Yubikey has proposed a solution to this problem, but the proposal is still a draft.

Recommendation

Document the different types of WebAuthn devices (device-bound and cloud-based) so users are aware that their choice of a recovery device will impact the ability to back up the keys needed for wallet recovery. It may be preferred to show an alert during wallet setup if the user is using a device that does not support Passkey (and therefore does not support

backups of key material). It might be possible to use a built-in WebAuthn flag to prompt the user in this situation.

Developer response

Acknowledged, won't fix. While we have developed a specification for detecting the BE (Backup Eligibility) and BS (Backup State) flags from the created credential and displaying an appropriate warning to users, this feature has not been prioritized. We have accepted the associated risk, given that the primary objective is to mitigate the loss of wallet access due to device loss rather than authenticator loss. Even if a user explicitly decides to use a device-bound authenticator like Ledger / Yubikey, in scenarios where they lose access to their wallet, they can still recover it using their chosen authenticator. Additionally, users are encouraged to use mobile devices for passkey creation, which are more likely to support cloud-synced solutions.

8. Info - Current solution incompatible with zkSync Era

Summary

The process of creating a Safe proxy contract uses create2, which allows the address of the proxy to be calculated before deploying on-chain. The address is assumed to be the same on different chains, but this assumption breaks for zkSync Era, because it has a different formula for create2.

Vulnerability Detail

When a Zeal wallet user creates a new Safe wallet, they are immediately provided the wallet address to allow for funds to be sent to this address. The Safe wallet is not actually available on-chain at that address on that specific chain until the first transaction is performed on that chain. Because zkSync Era does not calculate create2 in the same way as other EVM chains, if the Zeal wallet assumes the same wallet address is available on zkSync Era, then a user may send funds to this address on zkSync Era thinking that this is their address on all chains, only to find out they cannot access those funds because create2 doesn't work in the same way on that chain.

Recommendation

Make it clear to users that zkSync Era is not currently supported with the Zeal Safe wallet feature.

Developer response

Acknowledged, this is known and an accepted shortcoming; we won't support chains with non-standard EVM implementations for the time being

9. Info - Recovery option only valid after performing tx

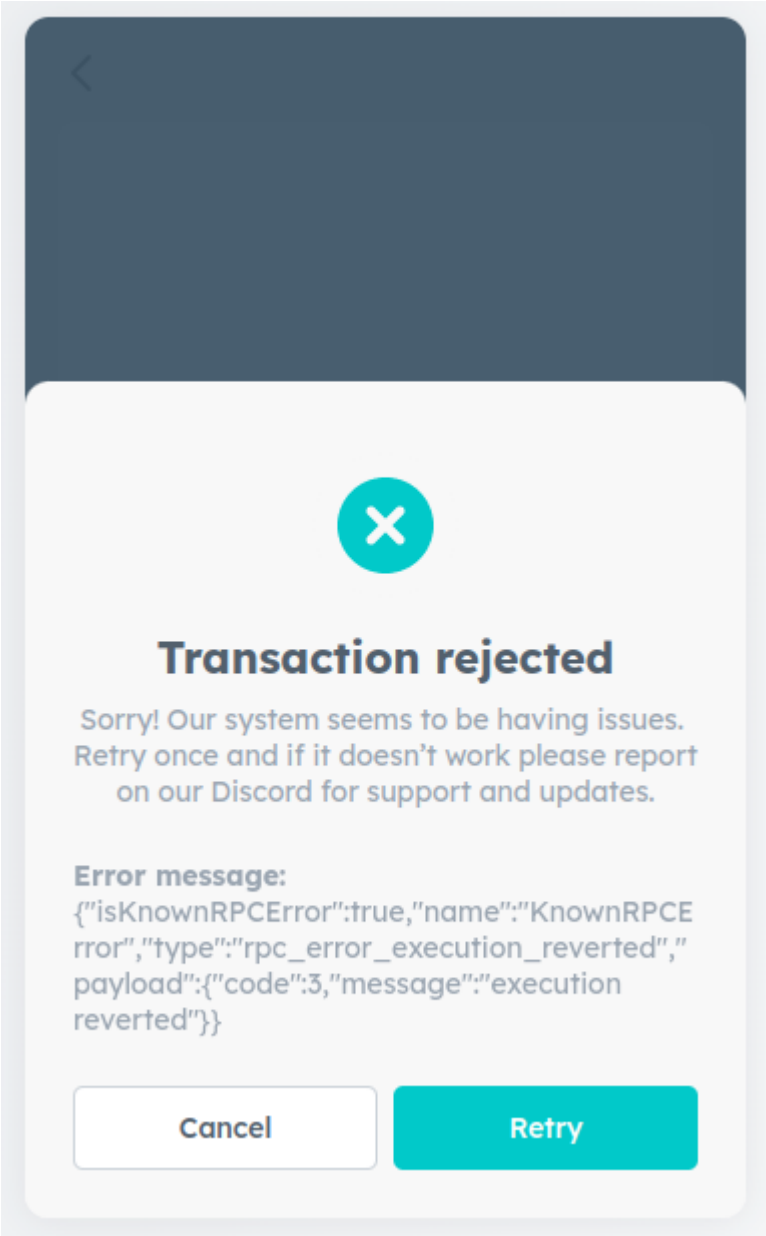
Summary

A Safe wallet using Passkey recovery is not actually recoverable until there is a transaction performed on-chain.

Vulnerability Detail

When a user creates their Zeal Safe wallet, they will receive the address of that wallet immediately. The user can send funds to the address on any compatible chain. But the wallet cannot be recovered until a user executes a tx on one of the chains, which will store their public key in the `signerData` mapping of the factory contract. In the unlikely scenario that a user creates a new Safe wallet, sends tokens to that address, then loses access to their Zeal wallet, they will be unable to recover their wallet using the existing recovery process.

In the event that a user does try to recover their wallet with their passkey before the wallet had an on-chain tx to store the public keys on-chain, the user will be greeted with this error message.



Recommendation

Because of the potential for loss of value in a scenario where the user does not complete one tx before needing to recover their wallet, the frontend should show a warning alert that the passkey recovery setup process is incomplete when a Safe wallet exists but does not yet have an on-chain tx yet.

Developer response

Acknowledged, fixed by deploying reference safe upon creation. The solution has since been updated to deploy a reference Safe account on Gnosis chain when creating a wallet. If this transaction succeeds, the wallet will only be created and usable within Zeal. Gas for this transaction is sponsored by Zeal, so the user is not required to have any native token balance

to create a wallet. Wallet recovery always happens on Gnosis chain, meaning the user no longer needs first to perform a transaction before their wallet is recoverable.

10. Info - Migrate from elliptic dependency to noble-curves

Summary

The noble-curves library has fewer dependencies than elliptic and is audited, which is a better security posture than the elliptic library. The scope of this audit was the smart contracts, so this is only considered informational because it is outside the primary scope.

Vulnerability Detail

The noble-curves library is a more secure alternative to elliptic. The library is involved in the Safe recovery process, handling the public key data.

Recommendation

Replace the elliptic dependency with noble-curves. As a general recommendation, check if there are audited alternatives for all the cryptography libraries used in the project. Paul Miller has written many secure implementations that may not yet have as many downloads as some other libraries, but are recommended for more security sensitive use cases.

Developer response

Acknowledged, the elliptic library is no longer used in the smart wallet flow.

11. Info - Inconsistent WebAuthn ceremony timeouts

Summary

The WebAuthn standard recommends a timeout value in the range of 300000 milliseconds to 600000 milliseconds, but the timeout values in the implementation are outside this range. The scope of this audit was the smart contracts, so this is only considered informational because it is outside the primary scope.

Vulnerability Detail

The timeout values used in the implementation vary from 6000000 to 60000 (1, 2). These timeout values are in milliseconds and none of them are in the recommended range - the timeouts are either larger or smaller than the recommended value.

Recommendation

Follow the recommended default value and use a timeout of 300000 milliseconds or 300 seconds.

Developer response

Acknowledged, fixed in PR #3813

12. Info - No challenge verification in Zeal wallet breaks security assumptions

Summary

When a Zeal Safe is created or restored, a challenge consisting of random bytes is included in the message sent to the authenticator. The response from the authenticator should be checked against this challenge, but that is not done by the Zeal wallet extension. This breaks the security assumptions of the WebAuthn protocol according to the specification. The scope of this audit was the smart contracts, so this is only considered informational because it is outside the primary scope.

Vulnerability Detail

Section 13.4.3 of the WebAuthn specification says:

> the returned challenge value in the client's response MUST match what was generated

This challenge is not an optional value in the create or get process. No such check is found in the Zeal wallet implementation. When `navigator.credentials.create()` is called, the challenge is generated on the fly and is not stored in any variable to be compared with the response from the authenticator. Similarly, when `navigator.credentials.get()` is called, the challenge is also not stored in a variable for later comparison. The example code for SimpleWebAuthn contains an example of a challenge check, and there do not appear to be any calls similar to `verifyAuthenticationResponse` or similar functions from the Simple WebAuthn dependency.

There may be other verification checks missing in the wallet. For example, `verifyAuthenticationResponse` in the SimpleWebAuthn library has a comment stating that its purpose is to "verify that the user has legitimately completed the login process". The checks carried out in `verifyAuthenticationResponse` validate the data provided by the authenticator device. This function is not called by the Zeal wallet when authenticator data is received, but likely should be. There are most likely other similar validation functions that should be called, but this was not investigated deeply as the browser extension was out of scope.

Recommendation

Add a check for this challenge value as required by the specification. Add other checks to validate all data provided by the authenticator device.

Developer response

Acknowledged.

13. Info - Challenge has insufficient bytes

Summary

The challenge included in the create or get actions should have 32 bytes of random data, but the challenge in the current implementation only has 26 bytes of random data.

Vulnerability Detail

The challenge in the `navigator.credentials.get()` and `navigator.credentials.create()` cases are only 26 byte long random values, when the maximum size is a 32 byte value.

Recommendation

Use 32 byte long challenge values instead of 26 bytes for maximum security and replay protection.

Developer response

Acknowledged, fixed in commit 8a5808327abbb273ac991478db1ceaf6b3bc4864

Final Remarks

The end-to-end WebAuthn system has many moving parts and must be implemented properly to comply with a lengthy specification. Errors anywhere in this end-to-end implementation process can create weaknesses in the solution's security, which can result in a worse case scenario of a user gaining unauthorized control over another user's wallet.

Some errors were found in various parts of this implementation process. Some of these errors are likely caused by lack of clarity around which piece of the end-to-end WebAuthn system handles which part of the process. The assumption made during this security audit was:

- the Zeal wallet extension serves as the Relying Party
- the user's Apple device, Android device, yubikey, ledger, or similar device serves as the authenticator
- the Zeal Safe wallet creation process serves as registering a new credential
- the Zeal Safe wallet recovery process serves as verifying an authentication assertion

Using multiple blockchain contracts to store the data used for Safe wallet recovery brings unique challenges, as highlighted by the medium finding in this report. It might be worth considering some architectural changes if the current design introduces added complexity compared to other approaches. If the data stored on the blockchain is moved to a more centralized backend system, it would eliminate the high and medium issues in this report, and would allow wallet recovery even without an on-chain tx (an issue documented in an info finding).

Appendix A: Overview of New Safe Setup Steps with WebAuthn

- When a new public key credential is created, a registration ceremony happens. This is how the user's public key is associated with the user's account
  - https://w3c.github.io/webauthn/#registration-ceremony
- The authenticator creates an attestation signature, which is provided to the Zeal wallet NOTE: Authenticator = WebAuthn-supported security key (Yubikey, Ledger, etc.) or cloud platform (Apple, Google, etc.)
  - https://w3c.github.io/webauthn/#attestation-signature
- The attestation contains many pieces of data. The attestation is signed with the private key of the authenticator device (security key, Apple device, etc.) to prove that the data provided is legitimate
  - https://w3c.github.io/webauthn/#sctn-attestation
- The user.id value chosen by the Zeal wallet, which is the same as the hash value (or recoveryId) in the smart contract, is a random number of 32 bytes length
  - https://github.com/zealwallet/monorepo/blob/c7a86ef460d9d29f7e5a504afbb462de3795f054/frontend/domains/Account/features/CreateNewSafe/helpers/createPasskeyCredential.ts#L30
- The spec states that the user.id, also named user handle in the spec, is supposed to be specified by the Zeal wallet (AKA relying party)
  - https://w3c.github.io/webauthn/#user-handle
- The credential that is created is a discoverable credential. This means that when the user needs to recover their wallet, they don't have to type in a username or wallet address to

select what they want to recover or authenticate - they only need to select the correct account on the authenticator device

- We can confirm that discoverable credentials are used not only because of the lack of username or other information that is entered during the recovery process, but also because the allowCredentials argument is empty in navigator.credentials.get() calls
  - https://w3c.github.io/webauthn/#discoverable-credential
- Furthermore, to the proper requireResidentKey = false and residentKey = preferred are chosen for iOS and Android support
  - https://docs.turnkey.com/passkeys/options#requireresidentkey-and-residentkey
  - https://github.com/zealwallet/monorepo/blob/6cbc431a9418f451b01cc1c0367b2cd991b71873/frontend/domains/Account/features/CreateNewSafe/helpers/createPasskeyCredential.ts#L45-L46
- The attestation is received by the Zeal wallet frontend using navigator.credentials.create(). The response is in PublicKeyCredential.response
  - https://developer.mozilla.org/en-US/docs/Web/API/PublicKeyCredential/response
- The attestation is decoded (not decrypted, it's signed plaintext) to parse the authentication data
- The parsing is done by the `parseAuthenticatorData()` from the WebAuthn library. The goal is to extract the public key value from the authentication data
  - https://github.com/MasterKale/SimpleWebAuthn/blob/5229cebbcc2d087b7eaaaeb9886f53c9e1d93522/packages/server/src/helpers/parseAuthenticatorData.ts#L87
- As this source indicates, "In ECC, a public key is simply a point on the curve."
  - https://research.nccgroup.com/2021/11/18/an-illustrated-guide-to-elliptic-curve-cryptography-validation/#an-illustrated-guide-to-validating-ecc-curve-points
- The public key is decoded by the cbor-x library in an array. Array index -2 is the x coordinate and array index -3 is the y coordinate
  - https://github.com/kriszyp/cbor-x/blob/adae93ba4d695a99d72341cb96826547371ac74c/decode.js#L115
- The encoding format used for the public key is defined in RFC8152, indicating that -2 and -3 are indeed the x and y coordinates
  - https://datatracker.ietf.org/doc/html/rfc8152#appendix-C.7.1
- These docs spell it out even more clearly

- https://github.com/dotnet/designs/blob/main/accepted/2020/cbor/cbor.md#writing-an-elliptic-curve-public-key-as-a-cose_key-encoding

On-chain process for creating a new safe

- The “hash” that is stored on-chain is actually the user.id received from the authenticator
  - https://w3c.github.io/webauthn/#dom-authenticatorassertionresponse-userhandle
- On-chain, the public key x and y coordinates and the “hash” are stored in a mapping. The “hash” is the user.id (AKA user handle) which is random data generated by the Zeal wallet extension.

Appendix B: Overview of Safe Recovery Steps with WebAuthn

- When a user wants to recover their Zeal Safe, they use a WebAuthn authentication ceremony
  - https://w3c.github.io/webauthn/#sctn-verifying-assertion
- A summary of the authentication process is shown at https://webauthn.guide/#authentication navigator.credentials.get() is used to send a challenge to the authentication device The authentication device responds by following the authenticatorGetAssertion process. Because discoverable credentials are used, the user must first select the account (AKA passkey) that they want to use from their authenticator device
  - https://w3c.github.io/webauthn/#authenticatorGetAssertion-prompt-select-credential
- After an account is chosen, the private key for that account is used to sign a message. The data included in that message is described in the spec as the credential id, the authenticator data, signature, and user handle
  - https://w3c.github.io/webauthn/#authenticatorGetAssertion-return-values
- The Zeal wallet does not prompt the user to enter their wallet address. Instead, the user handle (originally generated by the Zeal wallet extension with random bytes) is used to identify the user.
- The spec states that “Discoverable credentials store this identifier and MUST return it as response.userHandle in authentication ceremonies”
  - https://w3c.github.io/webauthn/#user-handle

- As the previous step indicates, the user.id must be in the signed message that the authenticator presents to the wallet extension, because this is how the wallet knows which Safe is recovered.
- After the Zeal wallet receives user.id, the wallet extension uses the smart contract's `getSignerInfo(_hash)` view function to return the public key, the Signer contract address, and the Safe address
  - NOTE: the user.id is DIFFERENT from the credential id, but either one can be used to identify an account. user.id is generated by the Zeal wallet (AKA relying party) while credential ID is generated by the authenticator. The Zeal implementation uses the user.id to identify an account
  - https://w3c.github.io/webauthn/#user-handle

On-chain process for recovery

- When the user is ready to perform a tx from the new Safe, the new local wallet EOA must be added as a signer. Those process steps include:
- User approves the tx with their passkey, which signs some data
- Gelato calls the proper Gnosis safe. Gnosis Safe call flow: GnosisSafe.execTransaction() -> GnosisSafe.checkSignatures() -> GnosisSafe.checkNSignatures() -> Signer.isValidSignature()
  - The reason that Signer is called to check isValidSignature() is called is because the Signer is the currentOwner of the Safe
  - During the validation process, `_validate()` in Signer.sol decodes the signature data. One decoded value is the pair of integers r and s, which is how an ECDSA signature is stored. This is why there is a parameter "rs" fed into `WebAuthnLib.checkSignature()`
  - https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages#ecdsa-sign
  - Note that in the case of contract signatures (but maybe with a different elliptic curve?), the r value is the contract address
  - https://github.com/safe-global/safe-contracts/blob/186a21a74b327f17fc41217a927dea7064f74604/contracts/GnosisSafe.sol#L259-L260
- After all checks have passed, then the Safe will execute GnosisSafe.execute() and add the new wallet EOA to the Safe as another signer