certora

Security Assessment & Formal Verification Report v1

Safe

June 2024

Prepared for

Safe Ecosystem Foundation

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Passkey Module	https://github.com/safe-global/safe-modules/tree/main/modules/passkey	8a90660	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of **Safe's Passkey Module** using the Certora Prover and manual code review findings. The work was undertaken from **May 15, 2024** to **June 13, 2024**.

The following contract list is included in our scope:

```
contracts/SafeWebAuthnSignerFactory.sol
contracts/SafeWebAuthnSignerProxy.sol
contracts/SafeWebAuthnSignerSingleton.sol
contracts/base/SignatureValidator.sol
contracts/interfaces/IP256Verifier.sol
contracts/interfaces/ISafe.sol
contracts/interfaces/ISafeSignerFactory.sol
contracts/libraries/ERC1271.sol
contracts/libraries/P256.sol
contracts/libraries/WebAuthn.sol
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Please note that a few more formal rules are not included in this report, as they were proven with an unreleased version of the Certora Prover. Once those rules are proven on a released version of the Certora Prover, we will add them to the next version of this document.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0		
High	0		
Medium	0		
Low	0		
Informational	1		
Total	**1**		

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
I-01	EVM Version Shanghai may not work on other chains due to PUSH0	Informational	

Informational Severity Issues

I-01. EVM Version Shanghai may not work on other chains due to PUSH0

Description: This is a general recommendation to bring awareness to a prevalent problem that currently exists in the ecosystem.

The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which includes the new PUSH0 opcode. This opcode may not yet be implemented on all L2s, so deployment on these chains will fail. It's not necessary to specifically use PUSH0 in YUL for it to be included.

For example, this opcode is not supported on Base, which is built upon Optimism Bedrock (see here). See also this relevant issue on the official Solidity github for reference.

Due to this, deployment of any in-scope contract to the Base chain will always fail with error "invalid opcode: PUSH0".

Customer's response: We explicitly target the Paris EVM version which means that the compiler does not emit PUSH0 opcodes. So, while the statement is generally true, it does not affect out contracts given our Solidity compiler configuration:

https://github.com/safe-global/safe-modules/blob/8a906605010520bed5b532c9d2feb04fdf237832/modules/passkey/hardhat.config.ts#L65

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

SafeWebAuthnSignerFactory.sol

Module General Assumptions

- Loop iterations: Any loop was unrolled at most 6 times (iterations)

Contract Properties

P-01. Immutability of Singleton Contract.

Status: Verified	

Rule Name	Status	Description	Link to rule report
singletonNeverChanges	Verified	*This rule verifies that the singleton contract can't be overridden or replaced.*	[Report]()

P-02. getSigner is unique for every x,y, and verifier combination.

Status: Verified	Assumptions required to pass the rule as verified as per Safe's request: 1. Loop iterations: Any loop was unrolled at most 144 times (iterations). 2. Maximum Hashing length bound: 4694 3. Value before cast to address <= max_uint160. 4. Munging required to complete signer data to be constructed from full 32-byte size arrays. 5. mungedEquivalence proof.

Rule Name	Status	Description	Link to rule report
uniqueSigner	Verified	*For any distinct set of parameters (x, y, verifier), getSigner should return a unique address. Conversely, if the parameters are the same, getSigner should return the same address. This property ensures the uniqueness and consistency of signers.*	*Report*

P-03. createSigner and getSigner always return the same address.

Status: Verified	Assumptions: Using a summarization for the getSigner function (Proved in P-02).

Rule Name	Status	Description	Link to rule report
createAndGetSignerEquivalence	Verified	*For any given set of parameters (x, y, verifier), the addresses returned by createSigner and getSigner should be identical. This property ensures consistency between signer creation and retrieval.*	Report

P-04. Deterministic Address Calculation for Signers.

Status: Verified	Assumptions: 1. Loop iterations: Any loop was unrolled at most 144 times (iterations). 2. Maximum Hashing length bound: 4694

Rule Name	Status	Description	Link to rule report
deterministicSigner	Verified	*For any given set of parameters (x, y, verifier), getSigner will always return the same address regardless of the environment. This property ensures the consistency and predictability of the signer addresses.*	Report

P-05. Code Presence Check (_hasNoCode Integrity).

Status: Verified	

Rule Name	Status	Description	Link to rule report
hasNoCodeIntegrity	Verified	*The hasNoCodeIntegrity rule verifies that the specified address does not contain any code. This rule checks that if an address is equal to the proxy, it does have a code associated with it.*	*Report*

P-06. isValidSignatureForSigner consistent.

Status: Verified

Rule Name	Status	Description	Link to rule report
isValidSignatureForSignerConsistency	Verified	*This rule ensures that the function \`isValidSignatureForSigner\` behaves consistently across different environments. Specifically, it verifies that if the function does not revert in either of two calls with the same parameters, it should return the same result (the magic value). Conversely, if one call reverts, the other should also revert.*	*Report*

P-07. getSigner Reverting Conditions

Status: Verified	

Rule Name	Status	Description	Link to rule report
getSignerRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid.*	*Report*

SafeWebAuthnSignerProxy.sol

Module General Assumptions

- Loop iterations: Any loop was unrolled at most 6 times (iterations).

Contract Properties

P-01. Immutability of Configuration Parameters (X, Y, Verifiers, Singleton)	
Status: Verified	

Rule Name	Status	Description	Link to rule report
configParametersImmutability	Verified	*This rule verifies that the immutable fields _SINGLETON, _X, _Y, and _VERIFIERS defined in the proxy are indeed immutable and can never change after any function call.*	Report

P-02. Delegate Call Integrity (Calls Only to Singleton)

Status: Verified

Rule Name	Status	Description	Link to rule report
delegateCallsOnlyToSingleton	Verified	*This rule verifies that the delegate call in the proxy fallback always calls only the Singleton and never any other address.*	*Report*

P-03. Fallback Reverting Conditions

Status: Verified

Rule Name	Status	Description	Link to rule report
fallbackRevertingConditions	Verified	*This rule verifies that the fallback function in the Proxy reverts only when the delegatecall did not succeed (returned 0). In particular, this rule also verifies that the assembly data manipulations done in the fallback does not revert on its own.*	Report

SafeWebAuthnSignerSingleton.sol

Module General Assumptions

- Loop iterations: Any loop was unrolled at most 6 times (iterations).
- WebAuthn function encodeSigningMessage is working properly (Added a summary)
- P256 function verifySignatureAllowMalleability is working properly (Added a summary)

Contract Properties

P-01. Integrity of isValidSignature function.	
Status: Verified	Assumptions: Proved using the call only to isValidSignature(bytes32 message, bytes calldata signature) since we proved both isValidSignature implementations are equal.

Rule Name	Status	Description	Link to rule report
verifySignatureUniqueness	Verified	*This rule verifies that given 2 different messages with the same signature, the output of isValidSignature must be different.*	*Report*
verifySignatureIntegrity	Verified	*This rule verifies that given 2 different messages with the same signature, the output of isValidSignature will be equal if and only if both messages are equal.*	*Report*

P-02. Both isValidSignature behave the same.

Status: Verified	

Rule Name	Status	Description	Link to rule report
verifyIsValidSignatureAreEqual	Verified	*This rule verifies that both implementations of isValidSignature, with bytes and bytes32 messages, are retrieving the same output for the same messages.*	[Report]()

P-03. isValidSignature Reverting Conditions

Status: Verified

Rule Name	Status	Description	Link to rule report
isValidSignatureRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid or the authenticatorData (in signature) length is <= 32.*	[Report]

WebAuthn.sol

Module General Assumptions

- Loop iterations : Any loop was unrolled at most 6 times (iterations).

Contract Properties

P-01. CastSignature Consistent (Once valid always valid, Once failed always failed, includes revert cases and middle call)

Status: Verified

Rule Name	Status	Description	Link to rule report
castSignatureConsistent	Verified	*This rule verifies that if castSignature is valid for a given signature once, it will always be valid for that signature, and if it fails once, it will always fail for that signature. This rule includes cases where the function reverts or is called in different environments, ensuring reliable and consistent behavior across different scenarios.*	Report

P-02. verifySignature implementations equivalence.

Status: Verified	Assumptions: 1. We used a summary of encodeDataJson. 2. We used a summary of verifySignatureAllowMalleability

Rule Name	Status	Description	Link to rule report
verifySignatureEq	Verified	*The verifySignatureEq rule ensures that the two variants of the verifySignature function—one taking the signature as a bytes array and the other as a struct—produce equivalent results. Specifically, it verifies that both versions either revert under the same conditions or return the same result when given the same inputs. This ensures consistency and reliability between the two implementations.*	[Report]()

P-03. CastSignature Deterministic decoding.

Status: Verified

Rule Name	Status	Description	Link to rule report
castSignatureDeterministicDecoding	Verified	*The rule ensures that the castSignature function performs deterministic decoding. Specifically, it verifies that when a WebAuthn.Signature struct is ABI-encoded and then decoded using castSignature, the decoded signature matches the original struct. This guarantees that the decoding process is both canonical and consistent.*	[*Report*]

P-04. CastSignature Length checks validity.

Status: Verified	

Rule Name	Status	Description	Link to rule report
castSignatureLengthCheckValidity	Verified	*The rule ensures that the validity of the castSignature function is influenced by the length of the encoded signature. Specifically, it asserts that if the decoded signature matches the original struct, the validity of the signature decoding (isValid) is true if and only if the length of the encoded signature is less than or equal to the length of the ABI-encoded original struct. This validates that the function's length check is properly enforced.*	*Report*

P-05. verifySignatureConsistent (Always return the same status for the same inputs, not dependent on env or affected by 3rd party calls).

Status: Verified

Rule Name	Status	Description	Link to rule report
verifySignature Consistent	Verified	*The rule ensures the consistency of the `verifySignature` function. It verifies that the function behaves deterministically under the same input conditions. Specifically, it asserts that:* *1. The revert status (whether the function call reverted or not) should be the same across multiple calls with the same parameters.* *2. If neither call reverts, the result of `verifySignature` should be identical for both calls.* *This ensures that `verifySignature` produces consistent and reliable results when called with the same parameters, regardless of the execution environment.*	[Report]

P-06. Reverting Conditions

Status: Verified

Rule Name	Status	Description	Link to rule report
castSignatureRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid.*	[Report]
encodeClientDataJsonRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid.*	[Report]
encodeSigningMessageRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid.*	[Report]
checkAuthenticatorFlagsRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid or the authenticatorData length is <= 32.*	[Report]
verifySignatureRevertingConditions	Verified	*This rule verifies that castSignature reverts iff the function was paid or the authenticatorData (in signature) length is <= 32.*	[Report]

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.