certora

Security Assessment

Final Report



Safe Smart Account v1.3.0

July 2026

Prepared for Safe

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 3
- Protocol Overview 5

Assessment Methodology **6**

Threat and Security Overview **7**
- System Model 7
- Invariants and Trust Boundaries 7
- Attack Surface 8
- Review and Test Coverage 9
- Known Issues 9
- Findings Summary 13
- Severity Matrix 13

Detailed Findings **14**
- Low Severity Issues 15
  - L-01: Frontrunning of delegatecall actions to non-payable targets can cause a denial of service 15
- Informational Severity Issues 17
  - I-01: MultiSend does not validate transaction-entry boundaries 17
  - I-02: Native payment in setup call is silently waived in zero-gas-price transactions 18

Disclaimer **20**

About Certora **20**

certora

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Safe Smart Account v1.3.0	https://github.com/safe-fndn/safe-smart-account/tree/v1.3.0-libs.0	767ef36	Solidity

Project Overview

This document describes the security review of **Safe Smart Account v1.3.0**. The work was undertaken from **July 20th, 2026** to **July 29th, 2026**.

The following contract list is included in our scope:

```
contracts/GnosisSafe.sol
contracts/GnosisSafeL2.sol
contracts/accessors/SimulateTxAccessor.sol
contracts/base/Executor.sol
contracts/base/FallbackManager.sol
contracts/base/GuardManager.sol
contracts/base/ModuleManager.sol
contracts/base/OwnerManager.sol
contracts/common/Enum.sol
contracts/common/EtherPaymentFallback.sol
contracts/common/SecuredTokenTransfer.sol
contracts/common/SelfAuthorized.sol
contracts/common/SignatureDecoder.sol
contracts/common/Singleton.sol
contracts/common/StorageAccessible.sol
```

contracts/external/GnosisSafeMath.sol
contracts/handler/CompatibilityFallbackHandler.sol
contracts/handler/DefaultCallbackHandler.sol
contracts/handler/HandlerContext.sol
contracts/interfaces/ERC1155TokenReceiver.sol
contracts/interfaces/ERC721TokenReceiver.sol
contracts/interfaces/ERC777TokensRecipient.sol
contracts/interfaces/IERC165.sol
contracts/interfaces/ISignatureValidator.sol
contracts/interfaces/ViewStorageAccessible.sol
contracts/libraries/CreateCall.sol
contracts/libraries/GnosisSafeStorage.sol
contracts/libraries/MultiSend.sol
contracts/libraries/MultiSendCallOnly.sol
contracts/libraries/SignMessageLib.sol
contracts/proxies/GnosisSafeProxy.sol
contracts/proxies/GnosisSafeProxyFactory.sol
contracts/proxies/IProxyCreationCallback.sol

The team performed a manual audit of all the files in scope. During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

The safe-smart-account project implements Gnosis Safe v1.3.0, a modular smart-account system for EVM-compatible networks. Each Safe is deployed as a lightweight proxy that delegates execution to a shared singleton implementation. During initialization, the Safe configures an owner set and confirmation threshold. Subsequent transactions must satisfy the configured authorization policy before the Safe can perform calls or delegatecalls. Authorization supports EOA signatures, EIP-1271 contract signatures, and owner-approved transaction hashes, with nonce-based and chain-aware replay protection.

The system includes several extension mechanisms. Enabled modules may execute transactions through the Safe without collecting signatures for each operation; guards can perform checks before and after owner-authorized transactions; and fallback handlers can expose additional functionality such as contract-signature validation and token-receiver interfaces. Supporting contracts provide batched execution through MultiSend, contract creation, signed-message registration, transaction simulation, deterministic proxy deployment, and optional deployment callbacks. GnosisSafeL2 preserves the core execution model while emitting additional transaction information intended for indexing on EVM-compatible networks.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

System Model

Safe v1.3.0 implements a threshold-controlled smart account using a singleton-and-proxy architecture. Each GnosisSafeProxy delegates execution to a shared implementation while retaining its own storage and assets. During one-time initialization, setup configures the owner set, confirmation threshold, optional setup-time delegatecall target and payload, fallback handler, and setup payment. Owners authorize transactions through a threshold of accepted confirmations over an EIP-712 Safe transaction digest. Confirmations may be direct ECDSA signatures, prefixed eth_sign signatures, on-chain hash approvals, or contract-owner signatures. Transactions are bound to the Safe address, chain ID, nonce, target, calldata, operation, value, and reimbursement parameters.

The Safe holds ETH and arbitrary tokens without maintaining an internal balance ledger. Value may leave through owner-authorized execution, module execution, initialization payments, or gas reimbursement. External calls may interact with arbitrary protocols and token implementations, making target contracts, token behavior, callbacks, and return-data handling part of the security boundary.

Invariants and Trust Boundaries

The principal invariants reviewed were that the threshold remains between one and the current owner count; owner and module linked lists remain unique and sentinel-terminated; signatures represent distinct current owners; signature pointers remain within the supplied signature bytes; and each execTransaction() invocation whose outer call completes increments the Safe nonce exactly once, while an outer revert rolls back the increment. When safeTxGas or gasPrice is nonzero, failure of the inner call does not necessarily revert execTransaction(). In that case, the nonce may be consumed, reimbursement paid, and ExecutionFailure emitted even though the outer Ethereum transaction succeeds. Integrations must not treat the outer receipt status as proof that the requested action succeeded. Administrative functions must only be reachable through calls from the Safe itself; absent storage corruption, successful setup must make

subsequent setup calls unavailable; and proxy and delegatecall-library storage layouts must remain compatible.

The owner quorum is not the system's complete authority model. An enabled module can execute arbitrary calls and delegatecalls without owner signatures or guard checks. They also do not consume ordinary Safe transaction nonce. It can also cause the Safe to call its own administrative functions, giving the module effective control over owners, modules, the threshold, guard, fallback handler, and implementation. Modules must therefore be treated as fully trusted controllers.

Guards and fallback handlers introduce separate trust assumptions. A guard can veto owner-authorized transactions, while an incompatible or reverting guard can make them unavailable. Fallback handlers control unknown calls, token callbacks, simulation, and EIP-1271 behavior. Fallback handlers are invoked using CALL, not DELEGATECALL, and therefore execute in their own storage context. The handler sees the Safe as msg.sender, while the original caller is appended to calldata. Handler authorization must account for this forwarding model.

CREATE2 deployment binds the predicted address to the factory, singleton-bearing creation code, initializer hash, and salt nonce. This prevents substitution of a different initializer at the same address, but does not guarantee that the initializer calls setup() or leaves the Safe securely initialized. Initialization is especially security-sensitive because setupModules delegatecalls an initializer with unrestricted access to Safe storage. A malicious initializer can corrupt configuration or introduce hidden authority that is not reachable through ordinary owner or module enumeration. Safe deployment must therefore use a trusted initializer, and the resulting state should be verified before assets are deposited.

Transaction executors are untrusted. They control the outer gas limit, transaction gas price, signature submission, caller identity, and msg.value, subject to the owner-signed reimbursement parameters. Although signed transaction parameters cannot normally be changed, the outer msg.value is not signed and remains visible during delegatecall execution.

Attack Surface

The audit considered malicious or compromised owners, modules, guards, fallback handlers, relayers, contract-signature validators, initializers, callbacks, execution targets, and tokens. Review areas included signature parsing and ordering, stale approvals, linked-list corruption, hidden owners and modules, delegatecall storage modification, guard and handler denial of

service, reentrancy, calls to code-less addresses, proxy and CREATE2 deployment behavior, module pagination, migration safety, and evolving SELFDESTRUCT semantics.

Execution and economic analysis covered EIP-150 gas forwarding, unchecked Solidity 0.7 arithmetic, failed-execution handling, nonce consumption, ETH and ERC-20 reimbursements, return-data expansion, unsigned msg.value, and zero-gas-price networks. MultiSend analysis covered malformed headers, oversized data lengths, zero-extended memory, nested delegatecalls, and differences between on-chain execution and off-chain parsing. Cross-protocol review included EIP-712, EIP-1271, ERC-165, ERC-721, ERC-1155, ERC-777, non-standard ERC-20 behavior, and arbitrary external protocol calls.

Review and Test Coverage

All in-scope contracts were manually reviewed, including the core Safe and L2 implementation, owner and module management, execution, guards, fallback handling, proxies and factories, signature decoding, storage access, payment logic, MultiSend, contract creation, signed-message support, simulation, migrations, and token-transfer helpers. Relevant Solidity 0.7.6 compiler advisories and core bytecode were also examined.

The Hardhat suite completed with 264 passing tests and 4 pending tests. Covered scenarios include initialization, owner and threshold changes, module authorization and execution, mixed signature types, signature bounds, replay protection, calls and delegatecalls, successful and failed execution, reimbursements, guards, fallback forwarding, token callbacks, proxy deployment, CREATE2, batching, simulation, storage layout, and migrations.

No new stateful fuzzing or audit-specific formal verification was performed. Historical formal-verification and symbolic-execution reports apply to earlier versions or commits.

Known Issues

The client confirmed the issues below as known as of the review cutoff date, July 28, 2026. A known issue is a security-relevant behavior affecting the scoped Safe v1.3 contracts that had already been documented publicly or disclosed by the client. Known issues are excluded from the novel-finding total.

Some of these issues were independently identified during the review before comparison with the client-confirmed list. These are marked as rediscovered below. This designation records

review coverage; it does not claim novelty or duplicate credit. A match requires the same underlying behavior and impact—not merely a related technical concept.

Audit references use the following identifiers: A14 — Ackee v1.4.0, C15 — Certora v1.5.0, and A15 — Ackee v1.5.0.

ID	Client-confirmed known issue	Prior disclosure	Rediscovered
K-01	Callback proxy deployment can be front-run so the callback is omitted	A15-M1	Yes
K-02	A reverting or incompatible guard can deny Safe transactions	A14-M1; GH-309	Yes
K-03	Low-level calls to code-less addresses can appear successful, including during module setup	A14-M2; GH-483	Yes
K-04	ERC-777 reception requires explicit ERC-1820 registration	C15-M-01; GH-655	No
K-05	The proxy applies a non-standard full-word check to `masterCopy()` calldata	A15-L2	No
K-06	The proxy factory can accept a singleton without deployed code	A14-L1; GH-462	Yes
K-07	Setup and later delegatecalls can corrupt Safe state and introduce hidden owners, modules, or approvals	G0 v1.1.1 Issue 1; OpenZeppelin 2020; A14-W1	Yes
K-08	The Safe can be configured as its own fallback handler	A14-W2; PR-534	Yes

ID	Client-confirmed known issue	Prior disclosure	Rediscovered
K-09	Removed owners' approved hashes remain in storage	A14-W3	Yes
K-10	Proxy upgrade compatibility depends on reserving storage slot zero	A14-W4	No
K-11	A malicious module can act before a pending removal transaction	A14-W5	No
K-12	An impractically high threshold can make execution impossible	A14-W6	No
K-13	`SafeSetup` may not represent the final state after the setup delegatecall	A15-W4	No
K-14	Tokens sent directly to the callback handler can become locked	A15-W6	No
K-15	The module-pagination cursor can cause an enabled module to be omitted	Safe bounty; GH-461	Yes
K-16	Mixed EOA and contract signatures need not authorize the same message	GH-497	No
K-17	Signature padding can increase account-paid ERC-4337 verification costs	Safe bounty; GH-754	No
K-18	Native-token refunds can fail for contract receivers because `send()` is used	GH-601	No

ID	Client-confirmed known issue	Prior disclosure	Rediscovered
K-19	Contract-signature validation propagates attacker-influenced revert data	PR-1115	No
K-20	`CompatibilityFallbackHandler` can validate its own contract-owner signature when configured as an owner	PR-866	Yes
K-21	Proxy factory initialization failures discard the underlying revert data	A15-I4	Yes

In total, the client-confirmed set contains **20 known issues**, of which **9 were independently rediscovered** during the review.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	1	1	0
Informational	2	2	0
Total	3	3	0

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
L-01	Frontrunning of delegatecall actions to non-payable targets can cause a denial of service	Low	Acknowledged
I-01	MultiSend does not validate transaction-entry boundaries	Informational	Acknowledged
I-02	Native payment in setup call is silently waived in zero-gas-price transactions	Informational	Acknowledged

Low Severity Issues

L-01: Frontrunning of delegatecall actions to non-payable targets can cause a denial of service

Severity: Low	Impact: Low	Likelihood: Low
Files: contracts/base/Executor.sol	Status: Acknowledged	

Description:

`GnosisSafe.execTransaction` is payable, but the EIP-712 Safe transaction hash does not commit to the call's `msg.value`. The signed `value` argument is included in the hash, but it has different semantics: `Executor.execute` forwards it as native value for a normal `CALL`, whereas the EVM ignores it for a `DELEGATECALL`.

During a delegatecall, the target executes in the Safe's context and observes the `msg.value` of the outer `execTransaction` call. The caller can submit valid owner signatures with an arbitrary `msg.value`, since this is not part of the transaction hash. This provides a concrete denial-of-service path against non-payable delegatecall targets such as `SignMessageLib` and the `CreateCall` entry points. An attacker can copy a signed transaction from the mempool and front-run it with a nonzero `msg.value`. Since the target functions are non-payable, they reject the delegatecall because the call value is nonzero, so `execute` returns `false`. If the owners signed a nonzero `safeTxGas` or `gasPrice`, the Safe permits the inner failure under:

```
require(success || safeTxGas != 0 || gasPrice != 0, "GS013")
```

The nonce is consumed, and the Safe emits an `ExecutionFailure` event. When `gasPrice` is nonzero, the Safe may also reimburse the signed refund receiver. If that receiver defaults to `tx.origin`, the attacker receives the reimbursement.

The intended operation can no longer execute with the same signatures because its nonce has been consumed.

The attack requires a non-payable delegatecall target, a nonzero `safeTxGas` or `gasPrice` so the inner failure does not revert the outer Safe transaction, and a signature bundle that remains valid when submitted by another executor. It is not applicable where authorization relies on a caller-dependent prevalidated signature or where a guard restricts the executor or rejects failed execution. The attacker must also win the transaction-ordering race. If both `safeTxGas` and `gasPrice` are zero, `GS013` reverts the outer transaction and restores the nonce.

The attack is particularly relevant to time-sensitive operations. Safe previously documented in issue #227 that delegatecalls preserve the outer `msg.value` and addressed that compatibility problem for `MultiSend`. The current finding does not claim novelty for that EVM behavior. It concerns the remaining non-payable production libraries and the previously undocumented permissionless nonce-consumption and reimbursement attack enabled by unsigned `msg.value`.

Recommendations:

Bind the outer `msg.value`, at the owner-authorized entry point rather than inside the generic `Executor.execute()` function. For delegatecalls, require it to equal the signed `value` field:

```
function execTransaction(
    // ...
) public payable virtual returns (bool success) {
    if (operation == Enum.Operation.DelegateCall) {
        require(msg.value == value, "GSxxx");
    }
    // Existing transaction validation and execution logic
}
```

This ensures that an executor cannot alter the delegatecall's value context without invalidating the transaction. Applying the check in `execTransaction()` avoids imposing owner-signature semantics on setup and module execution paths that also use `Executor.execute()`.

Customer Response:

Acknowledged. Fix will be evaluated for future version.

Informational Severity Issues

I-01: MultiSend does not validate transaction-entry boundaries

Files:

contracts/libraries/MultiSend.sol
contracts/libraries/MultiSendCallOnly.sol

Description:

`MultiSend.multiSend` and `MultiSendCallOnly.multiSend` parse each packed transaction using assembly, but only check `lt(i, length)` before reading its 85-byte header. They do not verify that the complete header and the declared `dataLength` fit within the `transactions` byte array.

As a consequence, a truncated final entry can cause the assembly code to read beyond the end of the `transactions` byte array. A `dataLength` greater than the remaining payload can also cause the final `call` or `delegatecall` to receive zero-extended memory that was not explicitly included in the byte array. For example, an entry may declare 36 bytes of data while providing only a 4-byte selector, causing the target to receive the selector followed by 32 zero bytes.

In the normal Safe execution path, the out-of-bounds bytes are deterministic zeros, not attacker-controlled unsigned data. The transaction target, value, operation, declared length, and available data are also covered by the Safe transaction signature. Therefore, this behavior does not directly bypass Safe authorization, but it permits non-canonical encodings and can create discrepancies between on-chain execution and off-chain parsers, guards, or user interfaces. Furthermore, future changes that cause either function to allocate attacker-controlled memory immediately after the `transactions` byte array could allow an attacker to execute arbitrary actions that were not signed by the owners.

Recommendations:

Before executing an entry, verify that its full 85-byte header and entire `dataLength`-byte payload are present in the `transactions` byte array. Revert if either check fails, and require the cursor to end exactly at the end of the byte array.

Customer Response:

Acknowledged.

I-02: Native payment in setup call is silently waived in zero-gas-price transactions

Files:

contracts/GnosisSafe.sol

Description:

`GnosisSafe.setup` supports an optional exact payment to the account that initializes the Safe. To avoid adding separate payment logic, it reuses `handlePayment` as follows:

```
// baseGas = 0, gasPrice = 1 and gas = payment => amount = (payment + 0) * 1 = payment
handlePayment(payment, 0, 1, paymentToken, paymentReceiver);
```

The comment assumes that the resulting amount always equals `payment`. That is true for ERC-20 payments, but the native-token branch of `handlePayment` caps the supplied gas price at the transaction's effective gas price:

```
payment = gasUsed.add(baseGas).mul(gasPrice < tx.gasprice ? gasPrice : tx.gasprice);
```

For the setup call, this evaluates to `payment * min(1, tx.gasprice)`. If the network accepts transactions with `tx.gasprice == 0`, the calculated native payment is zero. Sending zero native tokens to an EOA succeeds, so setup completes, emits `SafeSetup`, and initializes the Safe even though the configured recipient does not receive the intended payment. The unpaid funds remain in the Safe.

This is not equivalent to the zero reimbursement produced by `execTransaction` on a zero-fee network. In `execTransaction`, the signed `gasPrice` is a maximum reimbursement rate and capping it at the executor's actual gas price is intentional. But in `setup`, the arguments to `handlePayment` are supposed to encode an absolute payment, so in that case the gas-price should not be capped.

The ERC-20 path is not affected because it calculates `(gasUsed + baseGas) * gasPrice` without applying the `tx.gasprice` cap.

On EVM networks that accept transactions with an effective gas price of zero, a Safe can be successfully initialized without making its configured native-token setup payment.

Recommendations:

Do not route exact setup payments through gas-reimbursement pricing logic. Implement a dedicated setup-payment path that transfers exactly `payment` for both native tokens and ERC-20 tokens while preserving the existing default-recipient behavior.

Customer Response:

Intended outcome as the costs should be capped to what was actually paid. The difference between native payment and erc20 payments is part of the code documentation: https://github.com/safe-fndn/safe-smart-account/blob/77901a5a1ad835b74ad3b72f73a8412cfe491c57/contracts/Safe.sol#L251

Fix Review:

We recommend documenting that distinction between native and ERC20 payments on deployment explicitly.

Customer Response:

Acknowledged.

Disclaimer

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 and is widely used by developers and security researchers during audits and bug bounties.

Certora also provides architectural design reviews, where researchers analyze system design and trust boundaries early to identify structural risks and reduce complexity before implementation.

Certora conducts off-chain audits covering backend services, APIs, frontend, and mobile applications, focusing on vulnerabilities in authentication, data handling, key management, and wallet-related workflows.

Certora also offers Penetration Testing and Red Teaming to simulate real-world attacks and uncover exploitable weaknesses across infrastructure, applications, and business logic.

In addition, Certora provides operational security (OpSec) assessments, reviewing key management, access controls, and operational processes to strengthen overall security posture.