BLOCKSEC

Security Audit
Report for Normal Account Contracts

Date: June 10, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	MegaETH
Target	Normal Account Contracts

Version History

Version	Date	Description
1.0	June 10, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-06-11 03:16:41+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Normal Account Contracts of MegaETH.

The Normal Account Contracts of MegaETH is a smart account infrastructure for EOAs leveraging EIP-7702 delegation. Its core objective is to upgrade standard EOAs into fully programmable smart accounts without deploying individual proxy contracts per user. Through EIP-7702's code delegation mechanism, an EOA can dynamically bind to predefined account logic while retaining its original address. The system supports multi-key authorization (secp256k1, P-256, WebAuthn, external signers), fine-grained spending limits, customizable execution guards to constrain transaction behavior, a new gas-payment system for flexible fee sponsorship, and cross-chain settlement capabilities that let a single account securely and efficiently interact across multiple chains

Note this audit focuses on the smart contracts located in the src directory, excluding the following directories/files:

- src/vendor/layerzero/mocks/EndpointV2Mock.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
normal-account	`Version 1`	`1790533a1d9d8a316cf46f85fc56707b4f7f2782`
	`Version 2`	`50d9bbf56229c9e007c5104b4863e3bae646b395`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on,

[1] https://github.com/megaeth-labs/normal-account.git

the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency

- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **ten** potential security issues. Besides, we have **seven** recommendations and **three** notes.

- High Risk: 3
- Medium Risk: 3
- Low Risk: 4
- Recommendation: 7
- Note: 3

ID	Severity	Description	Category	Status
1	High	Lack of validations for the address `i.funder`	Security Issue	Fixed
2	High	Potential lock of funds due to the use of non-ERC20 standard tokens	Security Issue	Fixed
3	High	The improper construction of `replaySafeDigest`	Security Issue	Fixed
4	Medium	Pre-execution side effects persist when business call fails	Security Issue	Confirmed
5	Medium	Lack of the fee validation in the function `executeSend()`	Security Issue	Fixed
6	Medium	Potential replay attacks due to the incorrect multichain flow	Security Issue	Fixed
7	Low	Potential loss of gas fee due to malicious intents	Security Issue	Fixed
8	Low	Lack of signature validity checks before invoking the function `ISettler.send()`	Security Issue	Fixed
9	Low	The improper design of the function `isValidSignatureWithKeyHash()`	Security Issue	Fixed
10	Low	The execution design of the function `initConfig()`	Security Issue	Fixed
11	-	Remove redundant code	Recommendation	Fixed
12	-	Revise the error name `VerificationError`	Recommendation	Fixed
13	-	Add non-zero checks in the contract `IthacaAccount`'s constructor	Recommendation	Fixed
14	-	Add the non-reentrant guard to the `withdrawTokens()` and `executePreCalls()` functions	Recommendation	Fixed
15	-	Revise the inconsistent approval amount	Recommendation	Fixed
16	-	Override the function `setPeer()` in the contract `LayerZeroSettler`	Recommendation	Fixed

17	-	Add non-zero checks in the contracts `SimpleSettler` and `LayerZeroSettler`	Recommendation	Fixed
18	-	Ensure proper selection of relayers	Note	-
19	-	Proper intent construction	Note	-
20	-	Ensure proper LayerZero configurations	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of validations for the address `i.funder`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Orchestrator`, the address `i.funder` extracted from the user' intent is responsible for sending funds to bootstrap the user's account (i.e., `i.eoa`) to successfully perform self-calls. However, the address `i.funder` is relayer-controllable and the function `_callFund()` lacks reassigning non-zero value (i.e., similar to the function `_callHandlePreCalls()`) to the variable `err` in its catch block. Malicious replayers can execute users' valid intents with a malicious funder, which always reverts with a zero error. As a result, the self-calls of users' valid intent will fail and pay compensation to the malicious relayer.

```
643   function _callFund(
644       address eoa,
645       address funder,
646       bytes32 digest,
647       bytes[] calldata encodedFundTransfers,
648       bytes calldata funderSignature
649   ) internal returns (bytes4 err) {
650       if (encodedFundTransfers.length == 0) return 0;
651
652       try this.selfCallFund537021665(eoa, funder, digest, encodedFundTransfers, funderSignature)
              {
653           return 0;
654       } catch (bytes memory returndata) {
655           if (returndata.length >= 4) {
656               assembly ("memory-safe") {
657                   err := mload(add(returndata, 0x20))
658               }
659           }
660       }
661   }
```

Listing 2.1: src/Orchestrator.sol

```
664   function _callHandlePreCalls(
665       address parentEOA,
666       address payer,
667       uint256 flags,
668       bytes[] calldata encodedPreCalls
669   ) internal returns (bytes4 err) {
670       try this.selfCallHandlePreCalls537021665(parentEOA, payer, flags, encodedPreCalls) {
671           return 0;
672       } catch (bytes memory returndata) {
673           if (returndata.length >= 4) {
674               assembly ("memory-safe") {
675                   err := mload(add(returndata, 0x20))
676               }
677           }
678           if (err == 0) err = PreCallError.selector;
679       }
680   }
```

Listing 2.2: src/Orchestrator.sol

Impact The self-calls of users' valid intent will fail and pay compensation to the malicious relayer.

Suggestion Add proper error handling in the function `_callFund()`.

2.1.2 Potential lock of funds due to the use of non-ERC20 standard tokens

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Escrow`, the function `escrow()` creates escrows for users by transferring tokens from the depositor to the contract `Escrow`. However, for non-ERC20 standard tokens (e.g., fee-on-transfer tokens, rebasing tokens, deflationary tokens, or tokens with interests), the actual received token amount may not align with the recorded transfer amount (i.e., `escrowAmount`). As a result, the potential token settlements (i.e., the function `settle()`) and refunds (e.g., the function `refund()`) may revert or fail to transfer the extra-gained portion (e.g., the accumulated interests of the token stETH), leading to potential lock of funds.

```
 91   function escrow(Escrow[] memory _escrows) public payable {
 92       uint256 totalNativeEscrowAmount;
 93
 94       for (uint256 i = 0; i < _escrows.length; i++) {
 95           if (_escrows[i].refundAmount > _escrows[i].escrowAmount) {
 96               revert InvalidEscrow();
 97           }
 98
 99           bytes32 escrowId = keccak256(abi.encode(_escrows[i]));
100
101           // Check if the escrow already exists
102           if (statuses[escrowId] != EscrowStatus.NULL) {
```

```
103             revert InvalidStatus();
104         }
105
106         statuses[escrowId] = EscrowStatus.CREATED;
107         escrows[escrowId] = _escrows[i];
108
109         if (_escrows[i].escrowAmount > 0) {
110             if (_escrows[i].token == address(0)) {
111                 totalNativeEscrowAmount += _escrows[i].escrowAmount;
112             } else {
113                 SafeTransferLib.safeTransferFrom(
114                     _escrows[i].token, msg.sender, address(this), _escrows[i].escrowAmount
115                 );
116             }
117         }
118
119         emit EscrowCreated(escrowId);
120     }
121
122     if (msg.value != totalNativeEscrowAmount) {
123         revert InvalidEscrow();
124     }
125 }
```

Listing 2.3: src/Escrow.sol

```
227 function _settle(bytes32 escrowId) internal {
228     Escrow storage _escrow = escrows[escrowId];
229
230     // Status has to be CREATED.
231     if (statuses[escrowId] != EscrowStatus.CREATED) {
232         revert InvalidStatus();
233     }
234
235     statuses[escrowId] = EscrowStatus.FINALIZED;
236
237     // Check with the settler if the message has been sent from the correct sender and chainId.
238     bool isSettled = ISettler(_escrow.settler)
239         .read(_escrow.settlementId, _escrow.sender, _escrow.senderChainId);
240
241     if (!isSettled) {
242         revert SettlementInvalid();
243     }
244
245     if (_escrow.escrowAmount > 0) {
246         TokenTransferLib.safeTransfer(_escrow.token, _escrow.recipient, _escrow.escrowAmount);
247     }
248
249     emit EscrowSettled(escrowId);
250 }
```

Listing 2.4: src/Escrow.sol

Impact Potential lock of funds due to the adoption of non-ERC20 standard tokens.

Suggestion Revise the token accounting logic accordingly.

Clarification from BlockSec The project must ensure that only standard ERC20 tokens are supported in the protocol to avoid potential lock of funds.

2.1.3 The improper construction of `replaySafeDigest`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `IthacaAccount`, the function `isValidSignature()` reconstructs the digest (i.e., `relaySafeDigest`) with domain separators to prevent replay risks. However, the function `_hashTypedDataOnlyVerifyingContract()` omits `chainid`, making the same signature valid across different chains. As a result, the improper construction of the variable `relaySafeDigest` is vulnerable to replay attacks, leading to potential loss of funds.

```
265   function isValidSignature(bytes32 digest, bytes calldata signature)
266       public
267       view
268       virtual
269       returns (bytes4)
270   {
271       // To sign an app digest (e.g. Permit2), you would need to perform a ‘hashTypedData‘ on the
              app’s 712,
272       // along with the app’s domain, then ‘signTypedData‘ with the account’s 712 and account
              domain.
273       // The account domain is added as a layer to prevent replay attacks since some apps do not
              include the
274       // account address as a field in their 712 data.
275       bytes32 replaySafeDigest = EfficientHashLib.hash(SIGN_TYPEHASH, digest);
276       digest = _hashTypedDataOnlyVerifyingContract(replaySafeDigest);
277
278       (bool isValid, bytes32 keyHash) = unwrapAndValidateSignature(digest, signature);
279       if (LibBit.and(keyHash != 0, isValid)) {
280           isValid = _isSuperAdmin(keyHash)
281               || _getKeyExtraStorage(keyHash).checkers.contains(msg.sender);
282       }
283       // ‘bytes4(keccak256("isValidSignature(bytes32,bytes)")) = 0x1626ba7e‘.
284       // We use ‘0xffffffff‘ for invalid, in convention with the reference implementation.
285       return bytes4(isValid ? 0x1626ba7e : 0xffffffff);
286   }
```

Listing 2.5: src/IthacaAccount.sol

Impact The improper construction of the variable `relaySafeDigest` is vulnerable to replay attacks, leading to potential loss of funds.

Suggestion Revise the construction of the variable `relaySafeDigest`.

2.1.4 Pre-execution side effects persist when business call fails

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract `Orchestrator`, the intent execution process (i.e., the function `execute()`) can be split into two main phases: Preparation (i.e., the function `_prepareExecution()`) and Execution (i.e., the function `_callSelfExecute()`). To preserve relayer compensation, the Preparation phase currently does not revert the entire transaction on certain failures (e.g., invalid signatures or expired intents). However, this design is improper. Specifically, if the Execution phase fails, the actions (i.e., asset funding and pre-calls via the function `_callHandlePreCalls()`) from the Preparation phase can still take effect, causing unexpected state changes.

```
438       // Funder/pre-call steps happen before main signature verification so they can bootstrap
439       // the account into a state where the final intent becomes valid.
440       updatedErr = _callFund(i.eoa, i.funder, digest, i.encodedFundTransfers, i.funderSignature);
441       if (updatedErr != 0) {
442           if (flags == _SIMULATION_MODE_FLAG) {
443               assembly ("memory-safe") {
444                   mstore(0x00, updatedErr)
445                   revert(0x00, 0x20)
446               }
447           }
448           return (keyHash, paymentCollected, nonceIncremented, updatedErr);
449       }
450
451       if (i.encodedPreCalls.length != 0) {
452           updatedErr = _callHandlePreCalls(i.eoa, i.payer, flags, i.encodedPreCalls);
453           if (updatedErr != 0) {
454               if (flags == _SIMULATION_MODE_FLAG) {
455                   assembly ("memory-safe") {
456                       mstore(0x00, updatedErr)
457                       revert(0x00, 0x20)
458                   }
459               }
460               return (keyHash, paymentCollected, nonceIncremented, updatedErr);
461           }
462       }
```

Listing 2.6: src/Orchestrator.sol

Impact If the Execution phase fails, the actions (i.e., asset funding and pre-calls via the function `_callHandlePreCalls()`) from the Preparation phase can still take effect, causing unexpected state changes.

Suggestion Revise the logic accordingly.

Feedback from the project The project stated that most pre-calls are independent and are allowed to succeed even if the execution phase fails. Moreover, the relayer will perform proper validation before submitting the transaction on chain.

2.1.5 Lack of the fee validation in the function `executeSend()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `LayerZeroSettler`, the function `executeSend()` accepts `msg.value` to cover cross-chain fees. Specifically, it iterates all messages across different endpoints and calls `_lzSend()` to send messages with the quoted fee. However, it does not validate the provided fee (i.e., `msg.value`) against the total quoted fees. As a result, the extra native value remains in the contract, allowing other users to freely send cross-chain messages.

```
61  function executeSend(
62      address sender,
63      bytes32 settlementId,
64      bytes calldata settlerContext,
65      bytes calldata signature
66  ) external payable {
67      bytes32 key = keccak256(abi.encode(sender, settlementId, settlerContext));
68
69      // Check that send() was called first
70      if (!validSend[key]) {
71          revert InvalidSettlementId();
72      }
73
74      // Verify LOSettlerSigner signature
75      bytes32 digest = computeExecuteSendDigest(sender, settlementId, settlerContext);
76
77      if (!SignatureCheckerLib.isValidSignatureNow(lOSettlerSigner, digest, signature)) {
78          revert InvalidLOSettlerSignature();
79      }
80
81      // Clear the authorization to prevent replay
82      validSend[key] = false;
83
84      // Decode settlerContext as an array of LayerZero endpoint IDs
85      uint32[] memory endpointIds = abi.decode(settlerContext, (uint32[]));
86
87      bytes memory payload = abi.encode(settlementId, sender, block.chainid);
88
89      // Type 3 options with minimal executor configuration for self-execution
90      bytes memory options = hex"0003";
91
92      // If the fee sent as msg.value is incorrect, then one of these _lzSends will revert.
93      for (uint256 i = 0; i < endpointIds.length; i++) {
94          uint32 dstEid = endpointIds[i];
95          if (dstEid == 0) revert InvalidEndpointId();
96
97          // Quote individual fee for this destination
98          MessagingFee memory fee = _quote(dstEid, payload, options, false);
99
100         // Send with exact fee, refund to msg.sender
```

```
101             _lzSend(dstEid, payload, options, MessagingFee(fee.nativeFee, 0), payable(msg.sender));
102         }
103     }
```

Listing 2.7: src/LayerZeroSettler.sol

```
200     function _payNative(uint256 _nativeFee) internal pure override returns (uint256 nativeFee) {
201         // Return the fee amount; the base contract will handle the actual payment
202         return _nativeFee;
203     }
```

Listing 2.8: src/LayerZeroSettler.sol

Impact The lack of fee validations in the contract `LayerZeroSettler` allows other users to send cross-chain messages with remaining native values.

Suggestion Validate that `msg.value` against the total quoted fees in the function `executeSend()`.

2.1.6 Potential replay attacks due to the incorrect multichain flow

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract Orchestrator, the function `_prepareExecution()` allows multichain execution (i.e., `i.isMultichain` = `true`) for chain-agnostic intents (i.e., when `p.nonce » 240` is equal to `MULTICHAIN_NONCE_PREFIX`). This design is vulnerable to replay attacks because the signed Merkle leaf can be valid across multiple chains. As a result, a chain-agnostic intent could be replayed on other chains, leading to potential loss.

```
394     function _prepareExecution(
395         bytes calldata encodedIntent,
396         Intent calldata i,
397         uint256 flags,
398         bytes4 err
399     )
400         internal
401         virtual
402         returns (bytes32 keyHash, bool paymentCollected, bool nonceIncremented, bytes4 updatedErr)
403     {
404         updatedErr = err;
405
406         if (i.supportedAccountImplementation != address(0)) {
407             if (accountImplementationOf(i.eoa) != i.supportedAccountImplementation) {
408                 updatedErr = UnsupportedAccountImplementation.selector;
409                 if (flags == _SIMULATION_MODE_FLAG) {
410                     revert UnsupportedAccountImplementation();
411                 }
412             }
413         }
414
415         // Porto's original semantics keep multichain execution mode separate from
```

```
416         // chain-agnostic signing. `i.isMultichain` selects the runtime path, while the
417         // nonce prefix controls whether the digest omits the chain ID.
418         bool usesMultichainRuntime = i.isMultichain;
419
420         if (!usesMultichainRuntime && i.paymentMaxAmount != 0 && updatedErr == 0) {
421             address maxPayer = Math.coalesce(i.payer, i.eoa);
422             if (TokenTransferLib.balanceOf(i.paymentToken, maxPayer) < i.paymentMaxAmount) {
423                 updatedErr = PaymentError.selector;
424                 if (flags == _SIMULATION_MODE_FLAG) {
425                     revert PaymentError();
426                 }
427             }
428         }
429
430         if (updatedErr != 0) {
431             return (keyHash, paymentCollected, nonceIncremented, updatedErr);
432         }
433
434         bytes32 digest = _computeDigest(i);
435         bytes32 paymentDigest = _computePaymentDigest(i);
436         bool isValid;
```

Listing 2.9: src/Orchestrator.sol

```
947     function _computeDigest(SignedCall calldata p) internal view virtual returns (bytes32) {
948         bool useChainAgnosticDigest = p.nonce >> 240 == MULTICHAIN_NONCE_PREFIX;
949         bytes32[] memory f = EfficientHashLib.malloc(5);
950         f.set(0, SIGNED_CALL_TYPEHASH);
951         f.set(1, LibBit.toUint(useChainAgnosticDigest));
952         f.set(2, uint160(p.eoa));
953         f.set(3, _executionDataHash(p.executionData));
954         f.set(4, p.nonce);
955
956         return
957             useChainAgnosticDigest ? _hashTypedDataSansChainId(f.hash()) : _hashTypedData(f.hash())
                ;
958     }
```

Listing 2.10: src/Orchestrator.sol

Impact The chain-agnostic intents could be replayed on multiple chains.

Suggestion Revise the code accordingly.

2.1.7 Potential loss of gas fee due to malicious intents

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `Orchestrator` is prefunded via the function `_collectPayment()` and temporarily holds `paymentMaxAmount` of tokens before executing self-calls. However, the function `withdrawTokens()` lacks a non-reentrant guard (i.e., the modifier `nonReentrant`). Self-calls

can invoke the function `withdrawTokens()` (line 174) to sweep assets precollected to the contract `Orchestrator` during execution. As a result, the function `_settlePayment()` may fail, causing the relayer to lose gas fee.

```
172   /// @dev Allows sweeping tokens accidentally held by the orchestrator.
173   /// If `token` is `address(0)`, withdraws the native gas token.
174   function withdrawTokens(address token, address recipient, uint256 amount) public virtual {
175       TokenTransferLib.safeTransfer(token, recipient, amount);
176   }
```

Listing 2.11: src/Orchestrator.sol

```
334       // Run the business call in a separate self-call so inner failures can be caught.
335       (, bytes4 selfCallErr) = _callSelfExecute(maxGas, encodedIntent, flags, keyHash, err);
336       if (err == 0) err = selfCallErr;
337
338       // Measured from the start of the outer frame so settlement can charge for all overhead.
339       gUsed = Math.rawSub(gStart, gasleft());
340
341       // Settlement happens after execution so the relayer is compensated on business-path
              failure.
342       if (paymentCollected) {
343           _settlePayment(i, gStart, gUsed, txEntryGas, modeOverhead);
344       }
345
346       emit IntentExecuted(i.eoa, i.nonce, nonceIncremented, err);
347   }
```

Listing 2.12: src/Orchestrator.sol

Impact The function `_settlePayment()` may fail, causing the relayer to lose gas fee.

Suggestion Add a `nonReentrant` modifier to the function `withdrawTokens()`.

Clarification from BlockSec The project must prevent users or other entities from directly transferring assets to the contract `Orchestrator`, since the function `withdrawTokens()` has been removed.

2.1.8 Lack of signature validity checks before invoking the function `ISettler.send()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Orchestrator`, the function `_prepareExecution()` invokes the function `ISettler(i.settler).send()` for multichain intents without validating the corresponding signatures beforehand. Moreover, invalid signatures do not revert the transaction and execution proceeds. As a result, multichain intents with invalid signatures may still trigger the function `ISettler(i.settler).send()`, causing unexpected results.

```
464       if (usesMultichainRuntime) {
```

```
465             (isValid, keyHash) = _verifyMerkleSig(digest, i.eoa, i.signature);
466             if (i.encodedFundTransfers.length > 0) {
467                 ISettler(i.settler).send(digest, i.settlerContext);
468             }
469         } else {
470             (isValid, keyHash) = _verify(digest, i.eoa, i.signature);
471         }
472
473         if (flags == _SIMULATION_MODE_FLAG) {
474             isValid = true;
475         }
476
477         if (!isValid) {
478             updatedErr = VerificationError.selector;
479             if (flags == _SIMULATION_MODE_FLAG) {
480                 revert VerificationError();
481             }
482             return (keyHash, paymentCollected, nonceIncremented, updatedErr);
483         }
```

Listing 2.13: src/Orchestrator.sol

Impact Multichain intents with invalid signatures may still trigger the `ISettler(i.settler).send()` function, causing unexpected results.

Suggestion Add signature validity checks before invoking the `ISettler(i.settler).send()` function.

2.1.9 The improper design of the function `isValidSignatureWithKeyHash()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MultiSigSigner`, the function `isValidSignatureWithKeyHash()`, is used to perform a threshold-based signature validation for configured `ownerKeyHash`es. Specifically, when a signature is validated, the corresponding slot in the value of `config.ownerKeyHashes[j]` is marked as used by setting it to `bytes32(0)`. However, in the contract `IthacaAccount`, the function `unwrapAndValidateSignature()`, validates EOA signatures (64 or 65 bytes) and returns a zero `keyHash` (i.e., `bytes32(0)`), which creates potential collisions in the multi-sig design of the contract `MultiSigSigner`. As a result, this design potentially creates duplicate counts due to the use of the in-memory marker `bytes32(0)`.

```
179   function isValidSignatureWithKeyHash(bytes32 digest, bytes32 keyHash, bytes memory signature)
180       public
181       view
182       returns (bytes4 magicValue)
183   {
184       bytes[] memory signatures = abi.decode(signature, (bytes[]));
185       Config memory config = _configs[msg.sender][keyHash];
186
```

```
187         uint256 validKeyNum;
188
189         // Iterate over signatures, until threshold is met.
190         for (uint256 i; i < signatures.length; ++i) {
191             // Unwrap and validate the signature.
192             (bool isValid, bytes32 ownerKeyHash) =
193                 IIthacaAccount(msg.sender).unwrapAndValidateSignature(digest, signatures[i]);
194
195             if (!isValid) {
196                 continue;
197             }
198
199             uint256 j;
200             while (j < config.ownerKeyHashes.length) {
201                 if (config.ownerKeyHashes[j] == ownerKeyHash) {
202                     // Incrementing validKeyNum
203                     validKeyNum++;
204                     // Mark the ownerKeyHash as used.
205                     config.ownerKeyHashes[j] = bytes32(0);
206
207                     // If threshold is met, return success.
208                     if (validKeyNum == config.threshold) {
209                         return _MAGIC_VALUE;
210                     }
211
212                     break;
213                 }
```

Listing 2.14: src/MultiSigSigner.sol

```
494     function unwrapAndValidateSignature(bytes32 digest, bytes calldata signature)
495         public
496         view
497         virtual
498         returns (bool isValid, bytes32 keyHash)
499     {
500         // Early return if unable to unwrap the signature.
501         if (signature.length < 0x21) return (false, 0);
502
503         // If the signature's length is 64 or 65, treat it like an secp256k1 signature.
504         if (LibBit.or(signature.length == 64, signature.length == 65)) {
505             return (ECDSA.recoverCalldata(digest, signature) == address(this), 0);
506         }
```

Listing 2.15: src/IthacaAccount.sol

Impact The design potentially creates duplicate counts due to the use of the in-memory marker `bytes32(0)`.

Suggestion Revise the code accordingly.

2.1.10 The execution design of the function `initConfig()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MultiSigSigner`, the function `initConfig()` does not invoke the function `_checkKeyHash()` to perform a check for the context key hash. This design allows any session key with the execution permission of the function `initConfig()` (especially via the wildcard `ANY_FN_SEL`) to initialize arbitrary key hashes in the variable `_configs`, violating the intended design.

```
75  function initConfig(bytes32 keyHash, uint256 threshold, bytes32[] memory ownerKeyHashes)
76      public
77  {
78      // Threshold can't be zero or greater than the number of owners
79      if (threshold == 0 || threshold > ownerKeyHashes.length) revert InvalidThreshold();
80
81      Config storage config = _configs[msg.sender][keyHash];
82
83      if (config.threshold > 0) {
84          revert ConfigAlreadySet();
85      }
86
87      _configs[msg.sender][keyHash] =
88          Config({threshold: threshold, ownerKeyHashes: ownerKeyHashes});
89  }
```

Listing 2.16: src/MultiSigSigner.sol

```
67  function _checkKeyHash(bytes32 expectedKeyHash) internal view {
68      bytes32 keyHash = IIthacaAccount(msg.sender).getContextKeyHash();
69      if (keyHash != expectedKeyHash) revert InvalidKeyHash();
70  }
```

Listing 2.17: src/MultiSigSigner.sol

```
436 function getContextKeyHash() public view virtual returns (bytes32) {
437     LibTStack.TStack memory t = LibTStack.tStack(_KEYHASH_STACK_TRANSIENT_SLOT);
438     if (LibTStack.size(t) == 0) {
439         return bytes32(0);
440     }
441
442     return LibTStack.top(t);
443 }
```

Listing 2.18: src/IthacaAccount.sol

Impact The design allows any session key with the execution permission of the function `initConfig()` to initialize arbitrary key hashes in the variable `_configs`.

Suggestion Revise the code accordingly.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to remove the following unreachable and unused code for better code readability.

```
479          if (flags == _SIMULATION_MODE_FLAG) {
480              revert VerificationError();
481          }
```

Listing 2.19: src/Orchestrator.sol

```
129    error KeyTypeCannotBeSuperAdmin();
```

Listing 2.20: src/IthacaAccount.sol

Suggestion Remove the redundant code.

2.2.2 Revise the error name `VerificationError`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Orchestrator`, the function `_prepareExecution()` invokes the function `_callFund()` before the function `_verify()`. When `_callFund` succeeds but `_verify` subsequently fails, the contract `SimpleFunder` has already transferred funds to the EOA and marked the mapping `usedDigests[digest]` as `true`. In this scenario, the returned error `VerificationError` carries no indication that the fund has already executed. A relayer may retry with the original digest without clearing the field `encodedFundTransfers`, which causes a retry failure. It is recommended to return a distinct error when the function `_verify()` fails after the function `_callFund()` has already succeeded, and document that the relayer must clear the field `encodedFundTransfers` when retrying after receiving this error.

```
438        // Funder/pre-call steps happen before main signature verification so they can bootstrap
439        // the account into a state where the final intent becomes valid.
440        updatedErr = _callFund(i.eoa, i.funder, digest, i.encodedFundTransfers, i.funderSignature);
```

Listing 2.21: src/Orchestrator.sol

```
476
477        if (!isValid) {
478            updatedErr = VerificationError.selector;
479            if (flags == _SIMULATION_MODE_FLAG) {
480                revert VerificationError();
481            }
482            return (keyHash, paymentCollected, nonceIncremented, updatedErr);
```

Listing 2.22: src/Orchestrator.sol

```
156         if (usedDigests[digest]) {
157             revert DigestUsed();
158         }
159         usedDigests[digest] = true;
```

Listing 2.23: src/SimpleFunder.sol

Suggestion Revise the error name `VerificationError` accordingly.

2.2.3 Add non-zero checks in the contract `IthacaAccount`'s constructor

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `IthacaAccount`'s constructor initializes the variable `ORCHESTRATOR` based on the input `orchestrator`. It is recommended to add a non-zero check for the input `orchestrator` to prevent potential mis-configuration.

```
228     constructor(address orchestrator) payable {
229         ORCHESTRATOR = orchestrator;
230     }
```

Listing 2.24: src/IthacaAccount.sol

Suggestion Add a non-zero check for the input `orchestrator`.

2.2.4 Add the non-reentrant guard to the `withdrawTokens()` and `executePreCalls()` functions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to add the non-reentrant guard (i.e., the modifier `nonReentrant`) to the function `withdrawTokens()` and `executePreCalls()`.

```
174     function withdrawTokens(address token, address recipient, uint256 amount) public virtual {
```

Listing 2.25: src/Orchestrator.sol

```
179     function executePreCalls(address parentEOA, SignedCall[] calldata preCalls) public virtual {
```

Listing 2.26: src/Orchestrator.sol

Suggestion Add non-reentrant guard to mentioned functions.

2.2.5 Revise the inconsistent approval amount

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SimpleFunder`, the function `fund()` incorrectly uses a `type(uint160).max` as an unlimited approval amount. It is recommended to revise the approval amount to align with the annotation `type(uint256).max`.

```
198                 mstore(add(m, 0x40), 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) // type(
                        uint256).max
```

Listing 2.27: src/SimpleFunder.sol

Suggestion Revise the inconsistent approval amount.

2.2.6 Override the function `setPeer()` in the contract `LayerZeroSettler`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `LayerZeroSettler` overrides the function `peers()` to always return `address(this)` for all endpoints, which indicates that cross-chain messages are always addressed to the same contract address. However, the contract `LayerZeroSettler` inherits from the contract `OAppCore`, where the function `setPeer()` remains callable.

Since function `peers()` is overridden to ignore the underlying `_peers` mapping, any writes performed via `setPeer()` have no runtime effect. Thus, it is recommended to explicitly override `setPeer()` to revert, improving code clarity and avoiding potential discrepancy.

```
223   function peers(uint32 _eid) public view virtual override returns (bytes32 peer) {
224       // Always return this contract's address for all endpoints
225       // This enables self-execution model where the same contract address is used across all
              chains
226       _eid; // Silence unused parameter warning
227       return bytes32(uint256(uint160(address(this))));
228   }
```

Listing 2.28: src/LayerZeroSettler.sol

```
53   function setPeer(uint32 _eid, bytes32 _peer) public virtual onlyOwner {
54       _setPeer(_eid, _peer);
55   }
```

Listing 2.29: src/vendor/layerzero/oapp/OAppCore.sol

Suggestion Override the function `setPeer()` in the contract `LayerZeroSettler`.

2.2.7 Add non-zero checks in the contracts `SimpleSettler` and `LayerZeroSettler`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to add non-zero checks in the following constructors and setter functions of the contract `SimpleSettler` and `LayerZeroSettler` to avoid potential misconfigurations.

```
17   constructor(address _owner) {
18       _initializeOwner(_owner);
19   }
```

Listing 2.30: src/SimpleSettler.sol

```
33    constructor(address _owner, address _l0SettlerSigner) OApp(_owner) Ownable(_owner) {
34        l0SettlerSigner = _l0SettlerSigner;
35    }
```

Listing 2.31: src/LayerZeroSettler.sol

```
177   function setL0SettlerSigner(address newL0SettlerSigner) external onlyOwner {
178       l0SettlerSigner = newL0SettlerSigner;
179   }
```

Listing 2.32: src/LayerZeroSettler.sol

Suggestion Add non-zero checks in the contracts `SimpleSettler` and `LayerZeroSettler`.

2.3 Note

2.3.1 Ensure proper selection of relayers

Introduced by `Version 1`

Description In the contract `Orchestrator`, the function `execute()` is fully permissionless and anyone can act as a relayer after obtaining users' intents and signatures based on the code. While the user's signature covers core `Intent` fields (execution data, nonce, payer, payment amounts, and etc.), several fields are excluded from the signed digest and can be freely modified by the relayer. Therefore, the project must select trusted and reliable relayers to avoid potential risks.

2.3.2 Proper intent construction

Introduced by `Version 1`

Description In the contract `Orchestrator`, the fields `txEntryGas`, `gasFloor`, and `paymentMaxAmount` are included in the signed payload. It is noted that the off-chain infrastructure is responsible for correctly computing and validating these fields before presenting the intent to the user for signing. The fields `txEntryGas` and `gasFloor` must be computed from the same calldata. The field `paymentMaxAmount` must be sufficient to cover outer execution overhead and leave a non-zero gas budget for the business self-call.

2.3.3 Ensure proper LayerZero configurations

Introduced by `Version 1`

Description The contract `LayerZeroSettler` is implemented to handle cross-chain transactions via the LayerZero protocol. The project must properly configure explicit LayerZero peers and source chain IDs to ensure proper functionality.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS