BLOCKSEC

Security Audit

Report for oripa-contract

Date: November 13, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About the Audit Target . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 2
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 Security Issue . . . 5
2.1.1 Misattribution of CCTP deposits in the function `relay()` . . . 5
2.1.2 Potential DoS due to the improper rounding direction in the function `_syncDebt()` 7
2.1.3 Inconsistent remainder accounting after rate changes . . . 8
2.1.4 Potential unfair reward distribution . . . 10
2.1.5 Potential DoS due to malicious session keys . . . 12
2.1.6 Lack of USDC recovery mechanisms for messages with invalid hook data . 14
2.2 Recommendation . . . 15
2.2.1 Avoid the reuse of retired session keys . . . 15
2.2.2 Add zero-address checks in the contract `CctpDepositReceiver` . . . 16
2.2.3 Remove the redundant parameter of the event `Minted` . . . 16
2.3 Note . . . 17
2.3.1 Ensure correct off-chain calculations . . . 17
2.3.2 Avoid the use of 18-decimal reward tokens . . . 17
2.3.3 Ensure correct runtime configurations . . . 17
2.3.4 Timely updates of the variable `minHouseReserve` . . . 17
2.3.5 Potential centralization risks . . . 19

Report Manifest

Item	Description
Client	Oripa
Target	oripa-contract

Version History

Version	Date	Description
1.0	November 13, 2026	First release

Signature



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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-11-13 07:26:31+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 oripa-contract of Oripa.

Oripa-contract of Oripa is an on-chain trading-card gacha protocol that lets users deposit USDC and rip packs from pools committed before sale. Rip outcomes are computed off-chain, while the contracts enforce signed debits, pool pricing, sequence numbering, balances, and withdrawals. Users authorize a session key once, and a relayer batches their signed intents. Settled losses mint $ORIPA, and stakers receive pro-rata USDC rewards. Received cards can be shipped after a signed fee or melted back for USDC from house funds. The platform cannot debit without a user signature, and withdrawals remain available while the protocol is paused.

Note this audit only focuses on the codes located in the following directories/files.

- src/OripaExchange.sol
- src/OripaStaking.sol
- src/OripaToken.sol
- src/OripaPools.sol
- src/SessionKeyManager.sol
- src/CctpDepositReceiver.sol
- src/interfaces/IOripa.sol
- src/interfaces/ICctp.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
oripa-contracts-audit-2026-11	`Version 1`	`3afd420fa1bbdb55e0b97bc5850766747c4f3103`
	`Version 2`	`f412b98ecc175d9488ed61bfec2eea136fa8cb1e`

[1] https://github.com/tyleryue-79/oripa-contracts-audit-2026-11

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, 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:

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

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

- **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.

Chapter 2 Findings

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

- Medium Risk: 1
- Low Risk: 5
- Recommendation: 3
- Note: 5

ID	Severity	Description	Category	Status
1	Medium	Misattribution of CCTP deposits in the function `relay()`	Security Issue	Fixed
2	Low	Potential DoS due to the improper rounding direction in the function `_syncDebt()`	Security Issue	Fixed
3	Low	Inconsistent remainder accounting after rate changes	Security Issue	Fixed
4	Low	Potential unfair reward distribution	Security Issue	Confirmed
5	Low	Potential DoS due to malicious session keys	Security Issue	Fixed
6	Low	Lack of USDC recovery mechanisms for messages with invalid hook data	Security Issue	Fixed
7	-	Avoid the reuse of retired session keys	Recommendation	Confirmed
8	-	Add zero-address checks in the contract `CctpDepositReceiver`	Recommendation	Fixed
9	-	Remove the redundant parameter of the event `Minted`	Recommendation	Confirmed
10	-	Ensure correct off-chain calculations	Note	-
11	-	Avoid the use of 18-decimal reward tokens	Note	-
12	-	Ensure correct runtime configurations	Note	-
13	-	Timely updates of the variable `minHouseReserve`	Note	-
14	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Misattribution of CCTP deposits in the function `relay()`

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description In the contract CctpDepositReceiver, the function relay() requires destinationCaller to be the contract CctpDepositReceiver and therefore rejects messages with an empty destinationCaller, consistent with the file README.md known issue 15. However, this check cannot prevent attackers from exploiting messages with an empty destinationCaller by following steps below:

1. The attacker relays an authenticated generic message whose destinationCaller is the contract CctpDepositReceiver and header recipient is an attacker-controlled contract.

2. The attacker-controlled contract submits the victim deposit message, which has an empty destinationCaller, via the contract TokenMessengerV2. This submission mints USDCs to the contract CctpDepositReceiver.

3. The contract CctpDepositReceiver credits the victim USDCs to the attacker through the malicious hookData.

As a result, the victims' cross-chain deposits can be credited to the attacker.

```
80  function relay(bytes calldata message, bytes calldata attestation)
81      external
82      nonReentrant
83      returns (address user, uint256 amount, bool credited)
84  {
85      if (message.length < BODY + HOOK_DATA) revert MessageTooShort();
86      // Burns with an empty destinationCaller could be received by anyone at the transmitter,
            minting here without a
87      // deposit; the frontend always sets it to this contract, and only those are accepted
88      if (bytes32(message[DESTINATION_CALLER:DESTINATION_CALLER + 32]) != _toBytes32(address(this))
            ) {
89          revert WrongDestinationCaller();
90      }
91      bytes32 nonce = bytes32(message[NONCE:NONCE + 32]);
92      uint32 sourceDomain = uint32(bytes4(message[SOURCE_DOMAIN:SOURCE_DOMAIN + 4]));
93      uint256 fee = uint256(bytes32(message[BODY + FEE_EXECUTED:BODY + FEE_EXECUTED + 32]));
94      bytes32 recipient = bytes32(message[BODY + MINT_RECIPIENT:BODY + MINT_RECIPIENT + 32]);
95
96      if (recipient != _toBytes32(address(this))) {
97          transmitter.receiveMessage(message, attestation); // CCTP mints straight to the named
                recipient
98          user = address(uint160(uint256(recipient)));
99          amount = uint256(bytes32(message[BODY + AMOUNT:BODY + AMOUNT + 32])) - fee;
100         emit CrossChainDeposit(nonce, sourceDomain, user, amount, fee, false);
101         return (user, amount, false);
102     }
103     user = _userOf(message[BODY + HOOK_DATA:], bytes32(message[BODY + MESSAGE_SENDER:BODY +
            MESSAGE_SENDER + 32]));
104
105     uint256 before = usdc.balanceOf(address(this));
106     transmitter.receiveMessage(message, attestation); // reverts on a bad attestation or a used
            nonce
107     amount = usdc.balanceOf(address(this)) - before;
108     if (amount == 0) revert NothingMinted();
109
```

```
110     usdc.forceApprove(address(exchange), amount);
111     try exchange.depositFor(user, amount) {
112         credited = true;
113     } catch {
114         usdc.forceApprove(address(exchange), 0);
115         usdc.safeTransfer(user, amount);
116     }
117     emit CrossChainDeposit(nonce, sourceDomain, user, amount, fee, credited);
118 }
```

Listing 2.1: src/CctpDepositReceiver.sol

Impact The victims' cross-chain deposits can be credited to the attacker.

Suggestion Add checks to ensure the header recipient of users' messages is the contract `TokenMessengerV2`.

2.1.2 Potential DoS due to the improper rounding direction in the function `_syncDebt()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `OripaStaking`, the function `_syncDebt()` calculates users' reward debts to reflect their pending rewards. However, the function rounds the reward debt down, which can allow users to accrue additional pending rewards and make total accrued rewards exceed the available reward balance. As a result, the last user to claim rewards may encounter a DoS issue due to the improper rounding direction.

```
186 function _accrue(address user) internal {
187     uint256 s = stakeOf[user];
188     if (s > 0) accrued[user] += (s * accRewardPerShare) / ACC - rewardDebt[user];
189 }
190
191 function _syncDebt(address user) internal {
192     rewardDebt[user] = (stakeOf[user] * accRewardPerShare) / ACC;
193 }
```

Listing 2.2: src/OripaStaking.sol

```
216 function _claim(address user) internal returns (uint256 amount) {
217     _accrue(user);
218     _syncDebt(user);
219     amount = accrued[user];
220     if (amount == 0) return 0;
221     accrued[user] = 0;
222     claimedOf[user] += amount;
223     exchange.creditReward(user, amount);
```

Listing 2.3: src/OripaStaking.sol

```
609 function creditReward(address user, uint256 amount) external {
610     if (msg.sender != address(staking)) revert OnlyStaking();
611     if (amount == 0) return;
612     if (rewardBalance < amount) revert InsufficientBalance();
613     rewardBalance -= amount;
614     balanceOf[user] += amount;
615     totalUserBalance += amount;
616     emit RewardCredited(user, amount);
```

Listing 2.4: src/OripaExchange.sol

Impact The last claim may suffer a DoS issue due to the improper rounding direction in the function `_syncDebt()`.

Suggestion Rounding up the reward debt to avoid the potential DoS issues.

2.1.3 Inconsistent remainder accounting after rate changes

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `OripaExchange`, the function `processSettlement()` applies the remainder reserve rate (i.e., `remainderReserveBps`) to preserve assets for potential melts and the function `processShipSelect()` also uses the same variable `remainderReserveBps` to calculate the gap amount (i.e., `charged`). However, if the governance changes the rate between both operations, the calculation of the gap amount will use the updated rate. As a result, the remainder accounting may become inconsistent due to the update of the remainder rate.

```
454 function processSettlement(Settlement calldata s) external {
455     if (msg.sender != address(this)) revert OnlySelf();
456     DrawRecord storage rec = draws[s.drawRef];
457     if (rec.user == address(0)) revert UnknownDraw();
458     if (rec.settled) revert AlreadySettled();
459     if (s.loss > rec.cost) revert LossAboveCost();
460     // Σ max(0, price -P_c)  cost -ΣP_c: a lower locked value has to come with a higher loss (more
            mint to the user)
461     if (s.loss + s.lockedValue < rec.cost) revert InconsistentSettlement();
462     rec.settled = true;
463     rec.lockedValue = SafeCast.toUint128(s.lockedValue);
464     uint256 minted = s.loss == 0 ? 0 : token.mintForLoss(rec.user, _toUsd18(s.loss));
465     emit DrawSettled(s.drawRef, rec.user, s.loss, minted);
466     uint16 rbps = remainderReserveBps;
467     if (rbps != 0) {
468         uint256 fee = address(staking) == address(0) ? 0 : (uint256(rec.cost) * feeBps) / 10_000;
469         int256 r = SafeCast.toInt256(uint256(rec.cost) - fee) - SafeCast.toInt256((s.lockedValue *
                rbps) / 10_000);
470         remainderBalance += r;
471         emit DrawRemainder(s.drawRef, s.lockedValue, r);
472     }
```

Listing 2.5: src/OripaExchange.sol

```
508 function processShipSelect(ShipSelect calldata s) external {
509     if (msg.sender != address(this)) revert OnlySelf();
510     uint16 rbps = remainderReserveBps;
511     if (rbps == 0) revert RemainderOff();
512     if (s.lockedValue == 0) revert ZeroAmount();
513     if (s.restockBps < rbps || s.restockBps > restockCapBps) revert InvalidRestock();
514     if (shipSelections[s.shipRef].lockedValue != 0) revert DuplicateRef();
515     DrawRecord storage d = draws[s.drawRef];
516     if (!d.settled) revert NotSettled();
517     // the cards shipped from one rip cannot add up to more than the locked value reported when
          it was settled
518     if (uint256(d.shippedValue) + s.lockedValue > d.lockedValue) revert ShippedAboveLocked();
519     uint256 charged = (s.lockedValue * (s.restockBps - rbps)) / 10_000;
520     d.shippedValue += SafeCast.toUint128(s.lockedValue);
521     shipSelections[s.shipRef] = ShipRecord({
522         drawRef: s.drawRef,
523         lockedValue: SafeCast.toUint128(s.lockedValue),
524         charged: SafeCast.toUint128(charged),
525         restockBps: s.restockBps,
526         reversed: false
527     });
528     remainderBalance -= SafeCast.toInt256(charged);
529     restockTotal += (s.lockedValue * s.restockBps) / 10_000;
530     emit ShipSelected(s.shipRef, d.user, s.drawRef, s.lockedValue, s.restockBps, charged);
```

Listing 2.6: src/OripaExchange.sol

```
681 /// @notice Reserve share of the locked price in the ticket remainder (v7), which is the melt
        rate (mainnet 8500); the
682 ///         platform quotes melts at this rate (v8). -MIN_MELT_RATE_BPS10000. It starts at 0 (
        remainder accounting off,
683 ///         local runs); once set it cannot be turned back to 0 (v9), so the floor cannot be
        sidestepped by switching the
684 ///         on-chain rate off. Admin (timelock)
685 function setRemainderReserveBps(uint16 bps) external onlyRole(DEFAULT_ADMIN_ROLE) {
686     if (bps == 0) {
687         if (remainderReserveBps != 0) revert InvalidFee();
688     } else if (bps < MIN_MELT_RATE_BPS || bps > 10_000) {
689         revert InvalidFee();
690     }
691     remainderReserveBps = bps;
692     emit RemainderReserveBpsUpdated(bps);
693 }
694
695 /// @notice The melt rate the platform pays (bps of a card's price), the same number as
        remainderReserveBps; 0 = never set
696 ///         (remainder accounting off)
697 function meltRateBps() external view returns (uint16) {
698     return remainderReserveBps;
699 }
700
701 /// @notice Buffer over minHouseReserve kept before the remainder is paid to stakers (v8; mainnet
         1000 = 10%). Admin (timelock)
```

```
function setRemainderBufferBps(uint16 bps) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (bps > 10_000) revert InvalidFee();
    remainderBufferBps = bps;
    emit RemainderBufferBpsUpdated(bps);
}

/// @notice Ceiling of c_c for ship selections (v11): between MIN_MELT_RATE_BPS and MAX_RESTOCK_CAP_BPS. The public text:
///       a shipped card's restock deduction is capped by its locked price, and the cap is this parameter
function setRestockCapBps(uint16 bps) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (bps < MIN_MELT_RATE_BPS || bps > MAX_RESTOCK_CAP_BPS) revert InvalidRestock();
    restockCapBps = bps;
    emit RestockCapBpsUpdated(bps);
```

Listing 2.7: src/OripaExchange.sol

Impact The remainder accounting may become inconsistent due to the update of the remainder rate.

Suggestion Store the applicable rate on each draw and use it for shipment accounting. Or, remove the on-chain calculation of the gap amount (i.e., `charged`) and calculate it off-chain by adding a variable (e.g., `charged`) in the struct `ShipSelect`.

2.1.4 Potential unfair reward distribution

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `OripaStaking`, users can instantly stake and unstake their ORIPA tokens without a timelock. Rewards are calculated from each user's staked amount and the variable `accRewardPerShare`, which is updated via the function `notifyReward()`. However, this design is potentially vulnerable to the MEV attack, leading to unfair reward distributions. Specifically, a malicious user can front-run the function `notifyReward()` by staking a large amount of ORIPA tokens bought from the potential market (e.g., DEX pools). After the reward distribution is recorded, the user can immediately claim rewards and unstake. As a result, a malicious user can capture most of the rewards issued via the function `notifyReward()`.

```
function notifyReward(uint256 amount) external {
    if (msg.sender != address(exchange)) revert OnlyExchange();
    uint256 total = amount + unallocated;
    if (totalStaked == 0) {
        unallocated = total;
        return;
    }
    unallocated = 0;
    accRewardPerShare += (total * ACC) / totalStaked;
    totalDistributed += total;
    emit RewardNotified(total, accRewardPerShare);
```

Listing 2.8: src/OripaStaking.sol

```
195 function _stake(address user, uint256 amount) internal {
196     if (amount == 0) revert ZeroAmount();
197     _accrue(user);
198     IOripaToken(address(token)).moveToStaking(user, amount);
199     stakeOf[user] += amount;
200     totalStaked += amount;
201     _syncDebt(user);
202     emit Staked(user, amount);
203 }
204
205 function _unstake(address user, uint256 amount) internal {
206     if (amount == 0) revert ZeroAmount();
207     if (stakeOf[user] < amount) revert InsufficientStake();
208     _accrue(user);
209     stakeOf[user] -= amount;
210     totalStaked -= amount;
211     _syncDebt(user);
212     token.transfer(user, amount);
213     emit Unstaked(user, amount);
```

Listing 2.9: src/OripaStaking.sol

```
177 function moveToStaking(address from, uint256 amount) external {
178     if (msg.sender != staking) revert OnlyStaking();
179     _transfer(from, staking, amount);
180 }
181
182 /// @dev Moves into staking are always allowed (locked tokens can be staked) except from the
        vault; moves out of
183 ///     staking are always allowed. Any other transfer needs transfers open and, once the
        schedule is set, fits in
184 ///     `from`'s transferable balance. Both ends get their  M recorded first: before OPEN tokens
        only move between a
185 ///     wallet and staking, so until an address's first transfer at or after OPEN its wallet
        balance plus its stake,
186 ///     less what it minted since OPEN, is exactly what it minted before OPEN
187 function _update(address from, address to, uint256 value) internal override {
188     if (from != address(0) && to != address(0)) {
189         if (to == staking) {
190             if (from == vault) revert VaultCannotStake();
191         } else if (from != staking) {
192             if (!_transfersOpen()) revert TransferRestricted();
193             if (transfersOpenAt != 0) {
194                 _recordLockBase(from);
195                 _recordLockBase(to);
196                 uint256 free = transferableOf(from);
197                 if (value > free) revert TransferLocked(free);
198             }
199         }
```

Listing 2.10: src/OripaToken.sol

Impact A malicious user can leverage the MEV attack to capture most of the rewards issued via the function `notifyReward()`.

Suggestion Add the timelock mechanism for the unstake operation.

Feedback from the project The project will not change the contract. Rewards are notified in small, frequent increments, and the token cannot be traded before transfers open on 2027-01-13, so just-in-time staking has no exit before then. The remaining lump-sum notifications are controlled operationally: the function `fundRewardsFromHouse()` is not called by the backend and will only be used in small, irregularly timed amounts, and the treasurer keeps a margin above `minHouseReserve` so that any held-back remainder stays small. The staking contract remains upgradeable through the 48-hour timelock if these controls prove insufficient after transfers open.

2.1.5 Potential DoS due to malicious session keys

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SessionKeyManager`, the function `registerSession()` allows users to register arbitrary session keys (e.g., ERC-1271 signer contracts) for verifications in the functions `debitDraws()` and `chargeFees()` of the contract `OripaExchange`. However, a malicious session key can cause a DoS issue in those functions. Specifically, a user can implement the function `isValidSignature()` in their malicious signer contracts to consume all gas provided to the invocation of the functions `debitDraws()` and `chargeFees()`, which normally process a batch of items. As a result, the functions `debitDraws()` and `chargeFees()` may fail until the malicious item is separated from the batch.

```
 99 function _register(address user, address key, uint8 scopes, uint64 expiry) internal {
100     if (scopes == 0 || (scopes & ~Scopes.ALL) != 0) revert InvalidScopes();
101     if (key == address(0) || expiry <= block.timestamp || expiry > block.timestamp +
            MAX_SESSION_DURATION) {
102         revert InvalidExpiry();
103     }
104     sessions[user][key] = Session({expiry: expiry, scopes: scopes});
105     emit SessionRegistered(user, key, scopes, expiry);
```

Listing 2.11: src/SessionKeyManager.sol

```
352 /// @notice Batch debit. Items are independent: a bad one (bad signature, low balance, duplicate)
          only emits
353 ///         DrawRejected and does not affect the rest of the batch
354 function debitDraws(DrawDebit[] calldata items) external nonReentrant onlyRole(RELAYER_ROLE)
        whenNotPaused {
355     for (uint256 i; i < items.length; ++i) {
356         try this.processDraw(items[i]) {}
```

```
357         catch (bytes memory reason) {
358             _bubbleOutOfGas(reason);
359             emit DrawRejected(items[i].intent.drawRef, items[i].intent.user, reason);
360         }
361     }
```

Listing 2.12: src/OripaExchange.sol

```
364 /// @dev Self-call target used by debitDraws' try/catch only
365 function processDraw(DrawDebit calldata d) external {
366     if (msg.sender != address(this)) revert OnlySelf();
367     DrawIntent calldata it = d.intent;
368     if (block.timestamp > it.deadline) revert SignatureExpired();
369     if (d.cost == 0) revert ZeroAmount();
370     if (d.cost > it.maxCost) revert CostAboveMax();
371     if (draws[it.drawRef].user != address(0)) revert DuplicateRef();
372     if (!sessionKeys.isAuthorized(it.user, d.sessionKey, Scopes.DRAW)) revert
            SessionNotAuthorized();
373     if (!SignatureChecker.isValidSignatureNow(d.sessionKey, drawDigest(it), d.signature)) {
374         revert InvalidSignature();
375     }
```

Listing 2.13: src/OripaExchange.sol

```
406 /// @notice Batch fee charge. Items are independent; a bad one only emits FeeRejected
407 function chargeFees(FeeCharge[] calldata items) external nonReentrant onlyRole(RELAYER_ROLE)
        whenNotPaused {
408     for (uint256 i; i < items.length; ++i) {
409         try this.processFee(items[i]) {}
410         catch (bytes memory reason) {
411             _bubbleOutOfGas(reason);
412             emit FeeRejected(items[i].feeRef, items[i].user, reason);
413         }
414     }
415 }
416
417 function processFee(FeeCharge calldata f) external {
418     if (msg.sender != address(this)) revert OnlySelf();
419     if (block.timestamp > f.deadline) revert SignatureExpired();
420     if (f.amount == 0) revert ZeroAmount();
421     if (feeCharged[f.feeRef]) revert DuplicateRef();
422     if (!sessionKeys.isAuthorized(f.user, f.sessionKey, Scopes.MELT)) revert SessionNotAuthorized
            ();
423     if (!SignatureChecker.isValidSignatureNow(
424             f.sessionKey, feeDigest(f.user, f.feeRef, f.amount, f.deadline), f.signature
425         )) {
426         revert InvalidSignature();
427     }
```

Listing 2.14: src/OripaExchange.sol

```
735 /// @dev Classic try/catch batching pitfall: when the inner call runs out of gas the revert data
        is empty. If we caught it,
```

```
736 ///     the outer tx would succeed and gas estimation would settle on an amount where every item
        fails silently.
737 ///     Empty revert data always reverts the whole tx so that estimation keeps increasing gas.
738 ///     All of our own errors are custom errors (with a selector), so they are never empty.
739 function _bubbleOutOfGas(bytes memory reason) internal pure {
740     if (reason.length == 0) revert OutOfGas();
741 }
```

Listing 2.15: src/OripaExchange.sol

Impact The functions `debitDraws()` and `chargeFees()` may fail until the item with the malicious session key is separated from the batch.

Suggestion Revise the code accordingly.

2.1.6 Lack of USDC recovery mechanisms for messages with invalid hook data

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `CctpDepositReceiver`, the function `relay()` only accepts CCTP messages with valid hook data. Specifically, if users provide a non-EVM address (e.g., Solana address) in the hook data, the function `_userOf()` reverts with the error `BadHookData`. However, the contract `CctpDepositReceiver` lacks implementing a recovery mechanism for such messages, while users' USDCs have been already burnt on the source chain. As a result, users' USDCs may be locked due to the lack of recovery mechanisms.

```
128 /// @dev hookData = abi.encode(address user): one 32-byte word with the upper 12 bytes zero.
        Otherwise the burn's
129 ///     messageSender, if it is an EVM address (EVM source chains). A Solana sender with bad
        hookData cannot be
130 ///     mapped to an address here: revert, and the owner settles it by hand.
131 function _userOf(bytes calldata hookData, bytes32 messageSender) internal pure returns (address)
        {
132     if (hookData.length == 32) {
133         uint256 word = uint256(bytes32(hookData));
134         if (word >> 160 == 0 && word != 0) return address(uint160(word));
135     }
136     uint256 sender = uint256(messageSender);
137     if (sender >> 160 == 0 && sender != 0) return address(uint160(sender));
138     revert BadHookData();
139 }
140
141 function _toBytes32(address a) internal pure returns (bytes32) {
142     return bytes32(uint256(uint160(a)));
143 }
```

Listing 2.16: src/CctpDepositReceiver.sol

Impact Users' USDCs may be locked due to the lack of recovery mechanisms.

Suggestion Add a recovery mechanism for messages with invalid hook data.

2.2 Recommendation

2.2.1 Avoid the reuse of retired session keys

Status Confirmed

Introduced by Version 1

Description In the contract `SessionKeyManager`, the function `revokeSession()` allows users to revoke their session keys by deleting them from the map `sessions`. However, this design does not prevent the reuse of retired session keys. It is recommended to add a map to record all retired keys and reject the corresponding registrations.

```
62 /// @notice Called by the user directly (pays gas)
63 function registerSession(address key, uint8 scopes, uint64 expiry) external {
64     _register(msg.sender, key, scopes, expiry);
65 }
66
67 /// @notice Relayed: the user signs RegisterSession with the wallet in the frontend and the
       platform pays gas
68 function registerSessionBySig(
69     address user,
70     address key,
71     uint8 scopes,
72     uint64 expiry,
73     uint256 deadline,
74     bytes calldata signature
75 ) external {
76     if (block.timestamp > deadline) revert SignatureExpired();
77     bytes32 digest = _hashTypedDataV4(
78         keccak256(abi.encode(REGISTER_TYPEHASH, user, key, scopes, expiry, _useNonce(user),
                deadline))
79     );
80     if (!SignatureChecker.isValidSignatureNow(user, digest, signature)) revert InvalidSignature();
81     _register(user, key, scopes, expiry);
82 }
83
84 /// @notice Only the user can revoke (costs a little gas, same as Papertrade)
85 function revokeSession(address key) external {
86     delete sessions[msg.sender][key];
87     emit SessionRevoked(msg.sender, key);
88 }
```

Listing 2.17: src/SessionKeyManager.sol

```
 99 function _register(address user, address key, uint8 scopes, uint64 expiry) internal {
100     if (scopes == 0 || (scopes & ~Scopes.ALL) != 0) revert InvalidScopes();
101     if (key == address(0) || expiry <= block.timestamp || expiry > block.timestamp +
            MAX_SESSION_DURATION) {
102         revert InvalidExpiry();
103     }
104     sessions[user][key] = Session({expiry: expiry, scopes: scopes});
105     emit SessionRegistered(user, key, scopes, expiry);
```

Listing 2.18: src/SessionKeyManager.sol

Suggestion Add a map to record all retired keys and reject the corresponding registrations.

2.2.2 Add zero-address checks in the contract `CctpDepositReceiver`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `CctpDepositReceiver`, the constructor lacks zero-address checks for input parameters (i.e., `transmitter_`, `exchange_`, and `usdc_`). It is recommended to add zero-address checks for these parameters.

```
60  constructor(IMessageTransmitterV2 transmitter_, IOripaExchange exchange_, IERC20 usdc_, address
        owner_)
61      Ownable(owner_)
62  {
63      transmitter = transmitter_;
64      exchange = exchange_;
65      usdc = usdc_;
66  }
```

Listing 2.19: src/CctpDepositReceiver.sol

Suggestion Add zero-address checks in the contract `CctpDepositReceiver`.

2.2.3 Remove the redundant parameter of the event `Minted`

Status Confirmed

Introduced by `Version 1`

Description In the contract `OripaToken`, the function `mintForLoss()` emits the event `Minted`, which allocates the same value to the parameters `lossUsd18` and `mintedLossUsd18`. Based on the file `README.md`, since the per-wallet mint cap was removed in v7, the parameter `mintedLossUsd18` becomes redundant. It is recommended to remove the parameter `mintedLossUsd18` from the event `Minted`.

```
128  function mintForLoss(address to, uint256 lossUsd18) external onlyRole(MINTER_ROLE) returns (
         uint256 minted) {
129      minted = quoteMint(cumulativeLossUsd, lossUsd18);
130      cumulativeLossUsd += lossUsd18;
131      if (minted > 0) {
132          _mint(to, minted);
133          // v12: mints at or after OPEN stay out of  M (only needed until  M is recorded)
134          if (_afterOpen() && !_lockBase[to].recorded) mintedSinceOpen[to] += minted;
135      }
136      emit Minted(to, lossUsd18, lossUsd18, minted);
```

Listing 2.20: src/OripaToken.sol

```
15  | `OripaToken` (`$ORIPA`) | Minted only from rip losses. Curve `flatRate × (S/(S+H))²`, minted
        with the closed-form integral so splitting a loss mints the same as one large loss. No per-
        wallet cap (v7). At launch tokens can only move between the wallet and the staking contract.
        Optional vault co-mint (`setVault`, a share of each user mint, set once before the first mint
        , off by default). v12 unlock schedule (`setUnlockSchedule`, once): from OPEN (2027-01-13 UTC
        ) 50% of what each address minted before OPEN, then 1/26 every 14 days; the vault locked
        until VCLIFF (2027-03-02 UTC), then 1/13 every 14 days, and never staked (docs/audit/README.
        md §12a). |
```

Listing 2.21: README.md in `Version 2`

Suggestion Remove the redundant parameter `mintedLossUsd18` from the event `Minted`.

2.3 Note

2.3.1 Ensure correct off-chain calculations

Introduced by `Version 1`

Description The project must ensure that all inputs (e.g., `s.lockedValue` and `m.amount`) are correctly computed off-chain. Specifically, the settlement loss, locked value, melt amount, and shipment restock data must match the committed card data.

2.3.2 Avoid the use of 18-decimal reward tokens

Introduced by `Version 1`

Description The project stated that rewards use 6-decimal USDC in the doc. An 18-decimal reward token is unsuitable under the current formula of the reward calculation because it can cause multiplication overflow during reward accounting. The implementation must not be reused with an 18-decimal reward token without redesigning its precision and overflow bounds.

2.3.3 Ensure correct runtime configurations

Introduced by `Version 1`

Description The project must ensure correct runtime configuration. Specifically, all fields of the parameter struct `Params` must be configured correctly. The variable `staking` must be set before draws are enabled. The parameters `restockCapBps` and `remainderReserveBps` must also be maintained within reasonable ranges.

2.3.4 Timely updates of the variable `minHouseReserve`

Introduced by `Version 1`

Description In the contract `OripaExchange`, the functions `fundRewardsFromHouse()` and `withdrawHouse()` allow the roles operator and treasurer to transfer assets based on the difference between the variables `houseBalance` and `minHouseReserve`. The project must ensure the timely updates of the variable `minHouseReserve` after card settlements.

```
592 function fundRewardsFromHouse(uint256 amount) external nonReentrant onlyRole(OPERATOR_ROLE) {
593     if (operatorCreditsFrozen) revert OperatorFrozen();
594     if (amount == 0) revert ZeroAmount();
595     if (address(staking) == address(0)) revert OnlyStaking();
596     uint256 day = block.timestamp / 1 days;
597     uint256 cap = params.dailyRewardFundCap;
598     if (cap != 0 && rewardFundedOnDay[day] + amount > cap) revert CapExceeded();
599     if (houseBalance < amount) revert InsufficientBalance();
600     // v8: like a treasury withdrawal, never below the reserve held for outstanding cards
601     if (houseBalance - amount < params.minHouseReserve) revert HouseReserve();
602     rewardFundedOnDay[day] += amount;
603     houseBalance -= amount;
604     rewardBalance += amount;
605     staking.notifyReward(amount);
606     emit RewardFunded(amount, amount);
607 }
```

Listing 2.22: src/OripaExchange.sol

```
639 function withdrawHouse(address to, uint256 amount) external nonReentrant onlyRole(TREASURER_ROLE)
        {
640     if (amount == 0) revert ZeroAmount();
641     if (houseBalance < amount || houseBalance - amount < params.minHouseReserve) revert
            HouseReserve();
642     houseBalance -= amount;
643     asset.safeTransfer(to, amount);
644     emit HouseWithdrawn(to, amount);
645 }
```

Listing 2.23: src/OripaExchange.sol

```
621 /// @notice Daily write of the reserve held against outstanding cards (0.85 × current index,
        computed off-chain).
622 ///         Only moves the floor for withdrawHouse; no funds move. May exceed houseBalance (
        coverage below 1).
623 ///         The admin can still set it through setParams
624 function setMinHouseReserve(uint256 amount) external onlyRole(RESERVE_ROLE) {
625     emit MinHouseReserveUpdated(params.minHouseReserve, amount);
626     params.minHouseReserve = amount;
627 }
628
629 /// @notice v7 (public tokenomics: the daily melt cap follows reserve coverage): written by the
        same daily job as the
630 ///         reserve. Coverage  1 →0 (no cap); below 1 →what the house bucket can pay that day;
        melts above it queue
631 ///         off-chain in order. The admin can still set it through setParams
632 function setReserveAndMeltCap(uint256 reserve, uint256 dailyCreditCap) external onlyRole(
        RESERVE_ROLE) {
633     emit MinHouseReserveUpdated(params.minHouseReserve, reserve);
634     emit DailyCreditCapUpdated(params.dailyCreditCap, dailyCreditCap);
635     params.minHouseReserve = reserve;
636     params.dailyCreditCap = dailyCreditCap;
637 }
```

Listing 2.24: src/OripaExchange.sol

2.3.5 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE`) can conduct sensitive operations, which introduce potential centralization risks. For example, the `DEFAULT_ADMIN_ROLE` can update `staking` via the function `setStaking()` to divert staking rewards and strand users' stake records based on the protocol. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS