BLOCKSEC

Security Audit
Report for Router Contracts

Date: July 21, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Inverter Network
Target	Router Contracts

Version History

Version	Date	Description
1.0	July 21, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-07-27 03:15:42+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 Router Contracts of Inverter Network.

Router Contracts of Inverter Network facilitate minting and redemption of iTRY tokens against a configured collateral token (e.g., USDC) via two distinct settlement pathways. Direct routes allow backend approved executors to submit payer-signed EIP-712 intents, while solver settlement routes enable allowlisted solvers to execute intents via backend-signed authorizations. The project also implements a vault (i.e., InstantRedeemPayoutVault) to release collaterals for instant and solver redeems with rolling bucket and window rate limits.

Note this audit only focuses on the smart contracts in the following directories/files:

- packages/foundry/contracts/periphery

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
iTRY-monorepo	`Version 1`	`86a307ee819c7751f34bf99b664936afde3ee49c`
	`Version 2`	`7e53559071b3437cb7f1cb80b46640b6233b2530`

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.

[1] https://github.com/InverterNetwork/iTRY-monorepo.git

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 have **three** recommendations and **four** notes.

- Recommendation: 3
- Note: 4

ID	Severity	Description	Category	Status
1	-	Remove redundant code	Recommendation	Fixed
2	-	Rename the function `availableWindowAllowance()`	Recommendation	Fixed
3	-	Revise the name of the parameter `minOmInput`	Recommendation	Fixed
4	-	Potential centralization risks	Note	-
5	-	Operational sequencing and reconciliation requirements	Note	-
6	-	The supported chains and tokens	Note	-
7	-	Proper consumption of the direct mint orders	Note	-

The details are provided in the following sections.

2.1 Recommendation

2.1.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is unused or redundant code. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. Redundant non-zero checks for the input `solverRedeemAuthorizationSigner`.

```
105  function setSolverMintAuthorizationSigner(address newSolverMintAuthorizationSigner) external
         override onlyOwner {
106    if (newSolverMintAuthorizationSigner == address(0)) revert InvalidZeroAddress();
107    if (
108      solverRedeemAuthorizationSigner != address(0)
109        && newSolverMintAuthorizationSigner == solverRedeemAuthorizationSigner
110    ) {
111      revert SolverAuthorizationSignersMustDiffer();
112    }
```

Listing 2.1: packages/foundry/contracts/periphery/SolverSettlementRouterBase.sol

2. Redundant invocation of the function `_requireRouterEligible()` in the functions `_validateSolverMintAuthorization()` and `_validateSolverRedeemAuthorization()`.

```
375    _requireRouterEligible();
```

Listing 2.2: packages/foundry/contracts/periphery/SolverSettlementRouterBase.sol

```
396    _requireRouterEligible();
```

Listing 2.3: packages/foundry/contracts/periphery/SolverSettlementRouterBase.sol

Suggestion Remove the redundant code.

Feedback from the project For point 2, the project stated that router eligibility remains enforced at the contract `SolverSettlementRouterBase`'s validation boundary. The duplicate check in the contract `DirectItryRouter`'s solver eligibility hook will be removed.

2.1.2 Rename the function `availableWindowAllowance()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `InstantRedeemPayoutVault`, the implementation of the function `availableWindowAllowance()` returns the releasable allowance, which does not align with its function name. It is recommended to rename the function for better code readability.

```
105 function availableWindowAllowance() external view override returns (uint256) {
106   if (!_releaseCapConfigured()) return 0;
107
108   uint256 bucketCount = releaseWindowBucketCount;
109   uint256 currentBucketId = block.timestamp / releaseBucketLength;
110   uint256 activeReleased = _activeReleasedInWindow(currentBucketId, bucketCount);
111   uint256 windowAvailable = activeReleased >= releaseWindowAllowance ? 0 :
          releaseWindowAllowance - activeReleased;
112
113   ReleaseBucket storage bucket = _releaseBuckets[currentBucketId % bucketCount];
114   uint256 bucketReleased = bucket.id == currentBucketId ? bucket.amount : 0;
115   uint256 bucketAvailable = bucketReleased >= releaseBucketAllowance ? 0 :
          releaseBucketAllowance - bucketReleased;
116
117   return bucketAvailable < windowAvailable ? bucketAvailable : windowAvailable;
118 }
```

Listing 2.4: packages/foundry/contracts/periphery/InstantRedeemPayoutVault.sol

Suggestion Rename the function `availableWindowAllowance()`.

2.1.3 Revise the name of the parameter `minOmInput`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DirectItryRouter`, the function `_acquireRedeemInput()` converts iTRY tokens into DLF tokens and then checks whether the returned DLF amount equals `minOmInput`. However, the parameter `minOmInput` suggests the minimum amount of DLF that the router expects to receive from the issuer's `redeemFor()` call, which would imply a `>=` check rather than a strict equality check. It is recommended to revise the parameter `minOmInput` to reflect its correct semantic.

```
305 function _acquireRedeemInput(uint256 inputAmount, uint256 minOmInput)
306   internal
307   override
308   returns (uint256 omInputAmount)
309 {
310   _resetAndApprove(itry, address(iTryIssuer), inputAmount);
311
312   IERC20 dlfToken = IERC20(address(dlf));
313   uint256 dlfBefore = dlfToken.balanceOf(address(this));
314   bool fromBuffer = iTryIssuer.redeemFor(address(this), inputAmount, minOmInput);
315   itry.safeApprove(address(iTryIssuer), 0);
316   if (!fromBuffer) revert IssuerRedeemNotFromBuffer();
317
318   omInputAmount = dlfToken.balanceOf(address(this)) - dlfBefore;
319   if (omInputAmount != minOmInput) revert SlippageExceeded();
320 }
```

Listing 2.5: packages/foundry/contracts/periphery/DirectItryRouter.sol

Suggestion Revise the anime of the parameter `minOmInput`.

Feedback from the project The project stated that the variable `minOmInput` represents the exact DLF amount expected from the issuer, rather than merely a minimum amount. The backend derives this amount from the issuer's calculation (i.e., the function `previewRedeem()`) when constructing the quote and reads it again during order submission. The amount is deterministic for a fixed issuer state and input amount. If the issuer's NAV, redemption fee, or other relevant state changes between quote creation and submission, the signed intent no longer matches and backend simulation or on-chain validation (i.e., the exact equality check for received DLF amount) rejects the transaction. The project will rename the internal router parameter from `minOmInput` to `expectedOmInput` and clarify the NatSpec without changing execution behavior.

2.2 Note

2.2.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles can conduct sensitive operations. For example, the router owner can update critical configuration through functions such as `setSolverMintAuthorizationSigner()` and `setSolverRedeemAuthorizationSigner()`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol. The project should ensure that privileged accounts are managed with multi-signature wallets and that key rotation procedures are in place.

Feedback from the project The project stated that privileged operations are going to be placed behind a governance setup using multiple Safe wallets and Zodiac modules. They are currently setting up this system. Key rotation and separation of operational responsibilities will be part of the deployment and governance procedures.

2.2.2 Operational sequencing and reconciliation requirements

Introduced by Version 1

Description Several accepted trade-offs rely on off-chain procedures that are not enforced on-chain. If these procedures are not followed, the system may process duplicate payments, lose vault liquidity, or leave funds stuck in the router. The following requirements should be satisfied for correct executions:

1. Solver authorizations should be signed only after the settlement contract has pushed the required tokens to the router. Solver routes use a push-funded model, and the router checks only aggregate balance rather than the source of funds.

2. `INSTANT` redeems should be reconciled before batch settlement runs. The vault pays USDC immediately, while `OrderManager` still records the user as the beneficiary. If batch settlement does not account for the vault payout, the user may be paid twice and the vault may not be reimbursed.

3. Vault cap configuration should be changed only after the affected routes are paused. `setReleaseWindowCap()` clears consumed usage and reopens the full allowance. During an incident, `setReleaseAllowances()` should be used to reduce limits without resetting usage.

4. Solver signer rotation should happen only after outstanding authorizations have expired or been processed. Rotating the signer immediately invalidates authorizations signed by the previous key. If funds were already pushed for such an authorization, they may remain stuck in the router until rescued manually.

Feedback from the project For point 1, the project stated that solver settlement uses a push-before-call model. Specifically, the approved settlement contract transfers the signed input amount to the router and calls the router in the same transaction. The router's balance check is atomic with that call, but intentionally verifies aggregate balance rather than source attribution. The authorization signature does not need to be generated after funding. The operational requirement is that the settlement contract performs the funding transfer first and invokes the router immediately afterward in the same transaction. Separate procedures also cover instant-redeem reconciliation, vault-cap changes, and signer rotation. For point 2, the project stated that INSTANT redemptions are treated as an advance payout by the contract `InstantRedeemPayoutVault` and the corresponding `OrderManager` redeem record still remains. For point 3, the project stated that vault cap changes are subject to an operational sequencing requirement. As for point 4, the project stated that solver authorization signer rotation must be coordinated with outstanding authorizations.

2.2.3 The supported chains and tokens

Introduced by Version 1

Description The project stated that the intended production deployment is on Ethereum Mainnet using USDC as collateral. The protocol does not plan to support USDT on Tron or other non-standard tokens for router contracts. Any future deployment on another chain or with a non-standard token would require a separate compatibility review.

2.2.4 Proper consumption of the direct mint orders

Introduced by Version 1

Description In the contract `DirectItryRouter`, the function `mintItryWithUsdc()` requires the collateral payer to sign the order (i.e., `OrderManager.Order`) and its corresponding intent (i.e., `DirectMintIntent`) for correct executions. The direct intent binds router-specific constraints such as final `iTRY` recipient, settlement vehicle, expiry, and `min_itry_amount`. However, the signed order remains independently executable by the minter role (i.e., `MINTER_ROLE`) of the contract `OrderManager`. If the minter role executes the signed order, the minted DLF tokens remain in the contract `DirectItryRouter` and users fail to mint the corresponding iTRY tokens breaking their intents. The project stated that this behavior is an intentional architectural trade-off. They will ensure that the relevant minter and route-executor identities are restricted to backend-controlled keys. The backend is responsible for ensuring that direct mint orders are executed through the router flow and for reconciling `OrderManager` execution with the corresponding router event.

```
156 function mintItryWithUsdc(
157   IOrderManager.Order calldata orderManagerOrder,
158   IOrderManager.Signature calldata orderManagerSignature,
159   DirectMintIntent calldata intent,
160   Signature calldata intentSignature
161 ) external override whenNotPaused nonReentrant onlyDirectRouteExecutor returns (uint256
      itryAmount) {
162   if (mintPaused) revert MintRoutePaused();
163
164   bytes32 orderKey = keccak256(bytes(orderManagerOrder.order_id));
165   if (executedOrders[orderKey]) revert OrderAlreadyExecuted();
166
167   string memory validationError = verifyDirectMintIntent(orderManagerOrder, intent,
        intentSignature);
168   if (bytes(validationError).length != 0) revert OrderValidationFailed(validationError);
169
170   _requireSubjectEligible(intent.payer);
171   executedOrders[orderKey] = true;
172
173   IERC20 dlfToken = IERC20(address(dlf));
174   uint256 dlfBefore = dlfToken.balanceOf(address(this));
175   orderManager.mint(orderManagerOrder, orderManagerSignature, intent.settlement_contract);
176
177   uint256 dlfAmount = dlfToken.balanceOf(address(this)) - dlfBefore;
178   if (dlfAmount < orderManagerOrder.estimated_token_out_amount) revert OrderManagerMintFailed();
179
180   itryAmount = _deliverMintedAsset(intent.recipient, dlfAmount, intent.min_itry_amount);
181   if (itryAmount < intent.min_itry_amount) revert SlippageExceeded();
```

Listing 2.6: packages/foundry/contracts/periphery/DirectItryRouter.sol


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS