BLOCKSEC

Security Audit
Report for EMUSD Contracts

Date: August 5, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Inverter Network
Target	EMUSD Contracts

Version History

Version	Date	Description
1.0	August 5, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-10 03:30:11+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 EMUSD Contracts of Inverter Network.

EMUSD Contracts of Inverter Network introduces the token EmAssetToken, which serves as the issuance token for their RWA tokenization and settlement system. Specifically, the token EmAssetToken can be minted and burned by the contract OrderManager when handling buy and redeem orders. Additionally, some minor changes have been made to the order manager module and related proxy contracts to support the corresponding adaption.

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

- packages/foundry/audit/ordermanager/new/OrderManager.sol
- packages/foundry/audit/proxies/new/InverterBeacon_v1.sol
- packages/foundry/audit/proxies/new/InverterBeaconProxy_v1.sol
- packages/foundry/audit/proxies/new/InverterReverter_v1.sol
- packages/foundry/audit/token/new/EmAssetToken_v1.sol
- packages/foundry/audit/token/new/SingleAdminAccessControlUpgradeable.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
emusd-monorepo	`Version 1`	`e2223e88bb36d2422dd1b77000a2f14d322d5caf`
	`Version 2`	`5f46ace1133d5b030db1cb729d1d2de814447999`

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/InverterNetwork/emusd-monorepo

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 **one** potential security issue. Besides, we have **one** recommendation and **one** note.

- Low Risk: 1
- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential allowance restoration	Security Issue	Fixed
2	-	Revise the error `DLF_NOT_INITIALIZED`	Recommendation	Fixed
3	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential allowance restoration

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `OrderManager`, the function `_handlePermitExecution()` skips the permit call (i.e., the function `_executePermit()`) when the existing allowance is sufficient. However, the valid permit remains usable, allowing anyone to perform the approval. As a result, users' allowance can be restored via unused permits after consuming their existing allowance.

```
872 function _handlePermitExecution(
873   address token,
874   address benefactor,
875   uint128 requiredAmount,
876   PermitParams calldata permitParams
877 ) internal returns (string memory) {
878   // Validate permit parameters
879   string memory failureReason = _validatePermitParamsSafe(requiredAmount, permitParams);
880
881   // Execute the permit if validation passed. A sufficient allowance may
882   // already exist - e.g. the same permit signature was relayed to the token
883   // by a third party (front-run), which consumes the permit nonce and makes
884   // our own permit() call revert. The order needs the allowance, not the
885   // permit call itself, so a permit failure is only fatal when the allowance
886   // still does not cover the order.
887   if (bytes(failureReason).length == 0) {
888     bool covered =
889       IERC20(token).allowance(benefactor, address(this)) >= requiredAmount;
890     if (!covered) {
891       _executePermit(token, benefactor, permitParams);
892       covered =
```

```
893         IERC20(token).allowance(benefactor, address(this)) >= requiredAmount;
894       }
895       if (!covered) {
896         return OrderManagerErrors.PERMIT_EXECUTION_FAILED;
897       }
898     }
```

Listing 2.1: packages/foundry/audit/ordermanager/new/OrderManager.sol

Impact Unused permits can restore the allowance.

Suggestion Check the allowance coverage after executing valid permits.

2.2 Recommendation

2.2.1 Revise the error `DLF_NOT_INITIALIZED`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `OrderManager`, the error `DLF_NOT_INITIALIZED` is still used after refactoring. It is recommended to revise the error `DLF_NOT_INITIALIZED` for better readability.

```
31   string internal constant DLF_NOT_INITIALIZED = "DLFNotInitialized";
```

Listing 2.2: packages/foundry/contracts/dlf/OrderManagerErrors.sol

```
655 function _validateMintOrder(
656   Order calldata order,
657   Signature calldata signature,
658   address settlementContract
659 ) private view returns (string memory failureReason) {
660
661   // Check issuance token is initialized before processing mint orders
662   if (address(issuanceToken) == address(0)) {
663     return OrderManagerErrors.DLF_NOT_INITIALIZED;
664   }
```

Listing 2.3: packages/foundry/audit/ordermanager/new/OrderManager.sol

```
683 function _validateRedeemOrder(
684   Order calldata order,
685   Signature calldata signature,
686   address settlementContract
687 ) private view returns (string memory failureReason) {
688
689   // Check issuance token is initialized before processing redeem orders
690   if (address(issuanceToken) == address(0)) {
691     return OrderManagerErrors.DLF_NOT_INITIALIZED;
692   }
```

Listing 2.4: packages/foundry/audit/ordermanager/new/OrderManager.sol

Suggestion Revise the error `DLF_NOT_INITIALIZED`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, the admin role can grant the minter role to any contracts, allowing that contract to mint tokens. 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