

# Audit Report

---

## EUROe Stablecoin

Delivered: 2022-12-23

Prepared for Membrane Finance by Runtime Verification, Inc.



[Summary](#)

[Disclaimer](#)

[EUROe: Contract Description and Properties](#)

[Overview](#)

[Properties](#)

[Findings](#)

[AO1: A blocked address can unblock itself](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

[Informative findings](#)

[Bo1: Missing check that permit spender has the BURNER\\_ROLE](#)

[Recommendation](#)

[Status](#)

[Bo2: No restrictions on which addresses can be blocked or unblocked](#)

[Recommendation](#)

[Bo3: BLOCKED addresses can still perform critical operations](#)

[Recommendation](#)

[Bo4: A BLOCKED address can still perform some operations](#)

[Recommendation](#)

# Summary

---

[Runtime Verification, Inc.](#) has audited the smart contract source code for Membrane Finance EUROe Stablecoin. The review was conducted from 2022-12-12 to 2022-12-16.

Membrane Finance engaged Runtime Verification in checking the security of their EUROe Stablecoin project. EUROe is a euro stablecoin always redeemable for 1 EUR, it is fiat-backed with liquid Euro denominated reserves held at European banks and financial institutions. Furthermore, it is regulated as an e-money institution in Europe by the Finnish Financial Supervisory Authority.

The issues which have been identified can be found in section [Findings](#). Additional suggestions can be found in section [Informative findings](#). We also created a Foundry test-suite, which can be found [here](#).

## Scope

The audited smart contract is:

- `EUROe.sol`

The audit has focused on the above smart contract, and has assumed correctness of the libraries and external contracts they make use of. The libraries are widely used and assumed secure and functionally correct.


The review focused mainly on the `membranefi/euroe-stablecoin` public code repository. The code was frozen for review at commit [1409f41dd52dc45459502eda2e58aa7d8bc455d2](#). After addressing a critical finding, the code was reviewed at commit [48681778aebfb11844c5c2f3ff2fa32df4f4c398](#).

## Assumptions

The audit is based on the following assumptions and trust model.

1. All addresses that have been assigned a role need to be trusted for as long as they hold that role. This roles include: `BLOCKLISTER_ROLE`, `PAUSER_ROLE`, `UNPAUSER_ROLE`, `MINTER_ROLE`, `RESCUER_ROLE`, `BURNER_ROLE` and `DEFAULT_ADMIN_ROLE`.
2. The contracts are upgradeable. Thus, `PROXYOWNER_ROLE`, which has the power to upgrade the contract, must be trusted, as they can significantly change the behavior of the protocol.

These assumptions are documented in [EUROe Developer Portal](#). Note they roughly assume honesty and competence. However, we will rely less on competence, and point



out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

## Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in [Disclaimer](#), we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to [known security issues and attack vectors](#). Thirdly, we developed a Foundry test-suite with fuzzing tests to ensure the desired properties are holding. Finally, we participated in meetings with the Membrane Finance team providing our feedback and suggested development practices as well as design improvements.

This report describes the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.



# Disclaimer

---

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

# EUROe: Contract Description and Properties

This section describes the EUROe project at a high-level, and which invariants we expect it to always respect at the end of a contract interaction.

## Overview

EUROe Stablecoin is a euro-backed stablecoin project. It is mostly a standard ERC20 token with few extra properties. In fact, the token follows the [ERC20 token standard](#) and implements all of its functionality through the [OpenZeppelin ERC20 implementation](#). The EUROe Stablecoin defines multiple roles with differing levels of access and to restrict access to certain functionality makes use of the [OpenZeppelin role-based access control](#) library. Below are the roles and which functions they can interact with (for more detail, see [here](#) the various roles and their function on the contract).

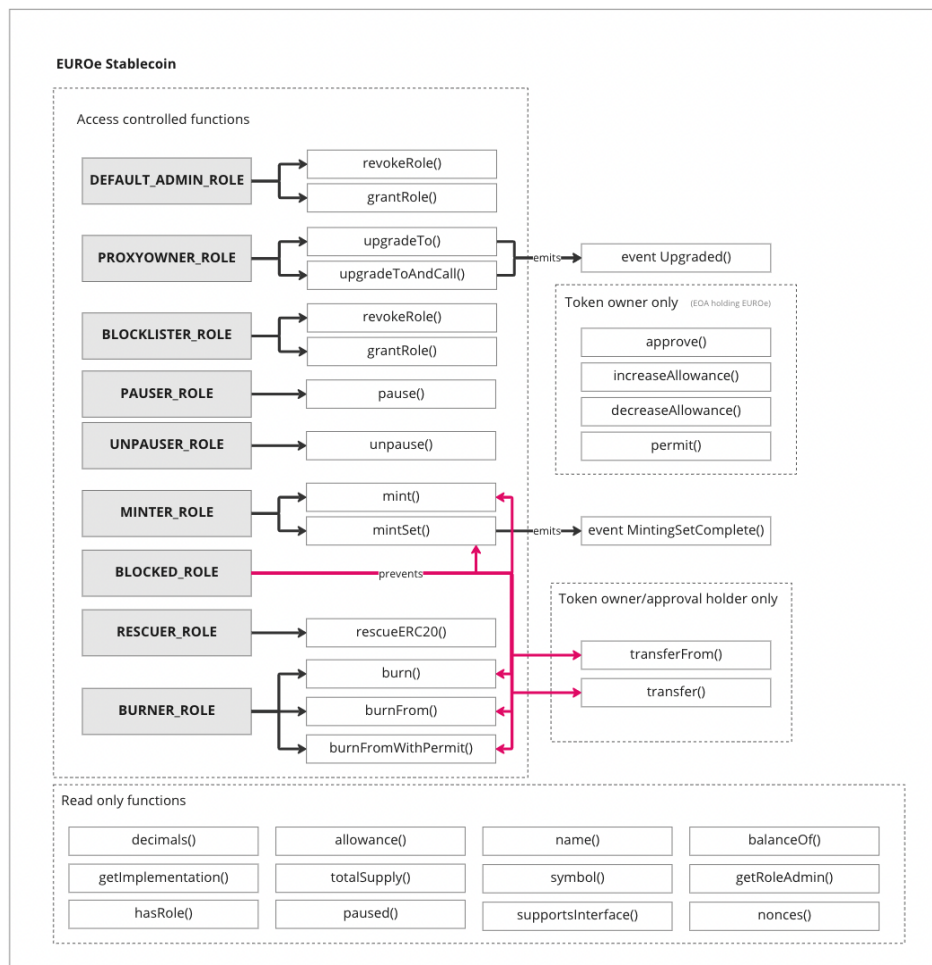


Fig 1. Architecture Diagram from [EUROe Developer Portal](#)

The contract is upgradable to allow for future changes to the code, including, for example, new features. Therefore, the contract is implemented through an [EIP-1967](#) UUPS upgradable proxy. This proxy is not by itself upgradeable, it is the role of the implementation to include all the code necessary to update the implementation's address that is stored at a specific slot in the proxy's storage. To achieve this EUROe makes use of the [OpenZeppelin UUPS pattern](#) library.

EUROe also supports [EIP-2612 Permit Extension for EIP-20 Signed Approvals](#). So it is possible for any user to give allowance to their tokens without a transaction by signing an [OpenZeppelin permit](#) message.

As mentioned above, the EUROe Stablecoin defines multiple roles and some functionalities are only accessible to a certain role. However, Fig 1 also shows the `BLOCKED_ROLE` which does not intend to give access to some functionality, but instead to block some functionalities to addresses assigned to that role. An address that has been assigned the `BLOCKED_ROLE` may not: mint new EUROe, burn existing EUROe, receive EUROe or send EUROe.

## Properties

---

There are several important properties that should be satisfied by the contract at all times. Here will only be listed properties related to the contract under the audit - `EUROe.sol`. Properties related to the use of external libraries are assumed to hold.

In the following, we list several properties that the contract should satisfy. These are not the only ones, but they are fundamental for the correctness of the protocol and as such deserve special attention.

- P1** It should not be possible to `initialize` the contract twice
- P2** The `upgradeTo` and `upgradeToandCall` function should only be accessible to an address with `PROXYOWNER_ROLE`
- P3** The `pause` function should only be accessible to an address with `PAUSER_ROLE`
- P4** The `pause` function should prevent tokens to be minted, burned and token transfers
- P5** The `unpause` function should only be accessible to an address with `UNPAUSER_ROLE`
- P6** If the contract is unpause then all the functionalities should be available

- P7** The `burn`, `burnFrom` and `burnFromWithPermit` functions should only be accessible to an address with `BURNER_ROLE`
- P8** The `burnFromWithPermit` functions should consume a permit and burn tokens based on the permit
- P9** The `mint` and `mintSet` functions should only be accessible to an address with `MINTER_ROLE`
- P10** The `rescueERC20` function should only be accessible to an address with `RESCUER_ROLE`
- P11** Only the `DEFAULT_ADMIN_ROLE` can `grantRole` or `revokeRole` the following roles to a given address: `MINTER_ROLE`, `BURNER_ROLE`, `RESCUER_ROLE`, `PROXYOWNER_ROLE`, `BLOCKLISTER_ROLE`, `PAUSER_ROLE` and `UNPAUSER_ROLE`
- P12** Only the `BLOCKLISTER_ROLE` can `grantRole` or `revokeRole` the `BLOCKED_ROLE` to a given address
- P13** An address with the `BLOCKED_ROLE` should not be able to send or receive tokens, neither to mint or burn tokens.



# Findings

---

## AO1: A blocked address can unblock itself

---

[ Severity: High | Difficulty: Low | Category: Security ]

The BLOCKLISTER\_ROLE can assign the BLOCKED\_ROLE to some address in order to prevent it from sending or receiving tokens, to be minted new tokens or to burn tokens. This is achieved through the modifier whenNotBlocked on the \_beforeTokenTransfer function. However, a blocked address can call the renounceRole function and get unblocked again. This would mean that an address would be able to send and receive tokens again, or to be minted or burn tokens again. This would violate **P13**.

```
function renounceRole(bytes32 role, address account) public virtual override {
    require(account == _msgSender(), "AccessControl: can only renounce roles
for self");

    _revokeRole(role, account);
}
```

## Scenario

1. BLOCKLISTER\_ROLE can assign the BLOCKED\_ROLE to Alice
2. Alice calls renounceRole() and has no longer the BLOCKED\_ROLE assigned to herself
3. Alice can send and receive tokens again, or to be minted new tokens or burn tokens again.

## Recommendation

Prevent an address assigned to BLOCKED\_ROLE to call the renounceRole function.

## Status

The client addressed the issue in commit 48681778aebfb11844c5c2f3ff2fa32df4f4c398 by making the renounceRole function always revert. Now a BLOCKED address can only be removed from BLOCKED\_ROLE if the BLOCKLISTER calls the revokeRole for that address. Note that addresses (controlled by Membrane) assigned to any other roles can still be removed to the role, but only the ADMIN can do that.

# Informative findings

## BO1: Missing check that permit spender has the BURNER\_ROLE

[ Severity: - | Difficulty: - | Category: Input Validation]

When a permit signature is valid, the owner approves the spender to spend value amount of tokens on its behalf. According to the documentation the function `burnFromWithPermit` is supposed to consume a received permit signature and burn tokens based on the permit. However, there is no check that the spender of the permit has the `BURNER_ROLE`, so it is possible to consume a different permit and still be able to burn tokens (because prior allowance was given).

This is not a security vulnerability but rather a mismatch between implementation and specification and makes it harder for Membrane to keep track of the trail of approvals.

```
function burnFromWithPermit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public onlyRole(BURNER_ROLE) {
    super.permit(owner, spender, value, deadline, v, r, s);
    super.burnFrom(owner, value);
}
```

### Scenario

1. Alice signs a valid permit P giving allowance to address Bob to spend value x
2. Alice approves BURNER value x
3. BURNER calls `burnFromwithPermit` with Alice as owner, Bob as spender and permit P. The `super.permit` invocation does not revert because the permit is valid. Consuming the permit will give Bob approval to spend x tokens from Alice. The `super.burnFrom` function will burn x amount of tokens from Alice because approval was also given to BURNER.

## Recommendation

Add a check that ensures that `spender` has the `BURNER_ROLE`:

```
require(hasRole(BURNER_ROLE, spender));
```

## Status

The client addressed the issue. However instead of following the suggestion they require that the `spender` is the `msg.sender`. This is a stronger requirement, because if there are two addresses with the `BURNER_ROLE`, each one can only submit permits in which they are the spender.

## BO2: No restrictions on which addresses can be blocked or unblocked

---

[ Severity: - | Difficulty: - | Category: Input Validation]

BLOCKLISTER can block or unblock any address. Notice that BLOCKLISTER is owned by Membrane Finance and according to the documentation:

*“Addresses are not blocked arbitrarily; an address may only be blocked pursuant to the Access Denial Policy, available at <https://euroe.com/legal/access-denial-policy>”*

Nevertheless, it is possible for BLOCKLISTER to block address o, and in that case it would make the `mint`, `mintSet`, `burn`, `burnFrom` and `burnFromWithPermit` unavailable, since before all transfers it is checked if both addresses involved in the transfer are not blocked. It is also possible for BLOCKLISTER to unblock the contract address which is not desirable.

### Recommendation

Either add a sanity check or document properly.

## Bo3: BLOCKED addresses can still perform critical operations

[ Severity: - | Difficulty: - | Category: Documentation]

The following table presents which operations can privileged addresses still perform even if they are blocked.

| BLOCKED address with: | Can perform:                        |
|-----------------------|-------------------------------------|
| DEFAULT_ADMIN_ROLE    | grantRole() and revokeRole()        |
| PROXY_OWNER_ROLE      | upgradeTo() and UpgradeToAndCall()  |
| PAUSER_ROLE           | pause()                             |
| UNPAUSER_ROLE         | unpause()                           |
| MINTER                | mint() and mintSet()                |
| BURNER                | burnFrom() and burnFromwithPermit() |
| RESCUER               | rescueERC20()                       |

This is not a security concern because these addresses are owned by the Membrane team, and in case they get compromised for some reason, Membrane should revoke the role to the compromised address, to remove their privileged access, instead of blocking it.

### Recommendation

Document prominently.

## Bo4: A BLOCKED address can still perform some operations

---

[ Severity: - | Difficulty: - | Category: Documentation]

A BLOCKED address can still perform some operations that change the state of the contract, such as: `approve`, `permit` and `transferFrom` between two addresses that are not its own. However, this is not a security concern because the BLOCKED address still gets its funds locked.

### Recommendation

Document prominently.