apriorit

EUR-L

Security audit report

Prepared for Lugh

March 3, 2022

Document management

Revision history

Version	Date	Version details
1.0	February 25, 2022	Initial version
1.1	March 3, 2022	Review after fixes

Table of contents

Project summary

Name	**Eurl Solidity**
Source	**Repository**: *https://gitlab.com/sceme/eurl-solidity* **Revision**: de1233cb948eca55099fe9e4788f8d13c642dc07 40c47b69fc6f1687951fdf3ad3d74f6b609d2b6b
Methods	Code review Unit test coverage analysis Static/symbolic analysis Behavioral analysis Manual penetration testing Gas usage analysis

Coverage and scope of work

The audit focused on an in-depth analysis of the EUR-L token implementation as well as its interaction with Forwarder smart contract. The main targets of the analysis were the following smart contracts:

- Blacksitable.sol
- Forwarder.sol
- Token.sol

Out of Scope:

- testToken.sol
- testTokenV2.sol
- whitepaper document

We conducted the audit in accordance with the following criteria:

- Consistency between the behavior of the contract and its description
- Behavioral analysis of smart contract source code
- Checks against our database of vulnerabilities and manual attacks against the contract
- Symbolic analysis of potentially vulnerable areas
- Static analysis
- Manual code review and code quality evaluation
- Unit test coverage analysis

- GAS usage analysis

The audit was performed using manual code analysis. Once potential vulnerabilities were discovered, manual attacks were performed to check if they could be easily exploited.

Smart contract overview

EURL Solidity smart contracts have been developed using Solidity language for the Ethereum network. They offer:

- ERC20 compatible functionality

- gasless transactions

- pausing of token mintage and burn

- blacklisting

- roles-based access control

Gasless transfer was built in such a way that the user signs the data off-chain, but the transaction is broadcasted by a paymaster. This includes the following steps:

- userA signs off-chain permit operation where a spender is a Forwarder smart-contract;
- paymaster calls permit on EURL token with result signature and broadcasts it;
- userA signs off-chain request for an usual ERC20 token transfer;
- paymaster broadcasts signed request by calling Forwarder smart-contract;
- after a sequence of verification checks paymaster receives some predefined fee from sender for gasless transfer operation;
- userB receives tokens.

There are several roles represented in the contract:

OWNER/DEFAULT_ADMIN_ROLE - the highest role in hierarchy that can update administrator, master minter, itself and also upgrade the EURL contract.

ADMIN/PAUSER_ROLE - responsible for fees management operations, blacklisting of users, pausing mintage, forcing transfers, settings of a trusted forwarder and a fees faucet.

MASTER_MINTER - responsible for adding/removing minters, updating of minting allowances, mint/burn.

MINTER_ROLE - responsible for mint/burn, minting management

RESERVE - role declared in a description but never mentioned in the actual implementation as an actual role. In reality it is a fee faucet that receives some fee from a sender based on manual transaction rate.

Executive overview

Apriorit conducted a security assessment of Eurl Solidity in February-March 2022 to evaluate its current state and risk posture, evaluate exposure to known security vulnerabilities, determine potential attack vectors, and check if any can be exploited maliciously.

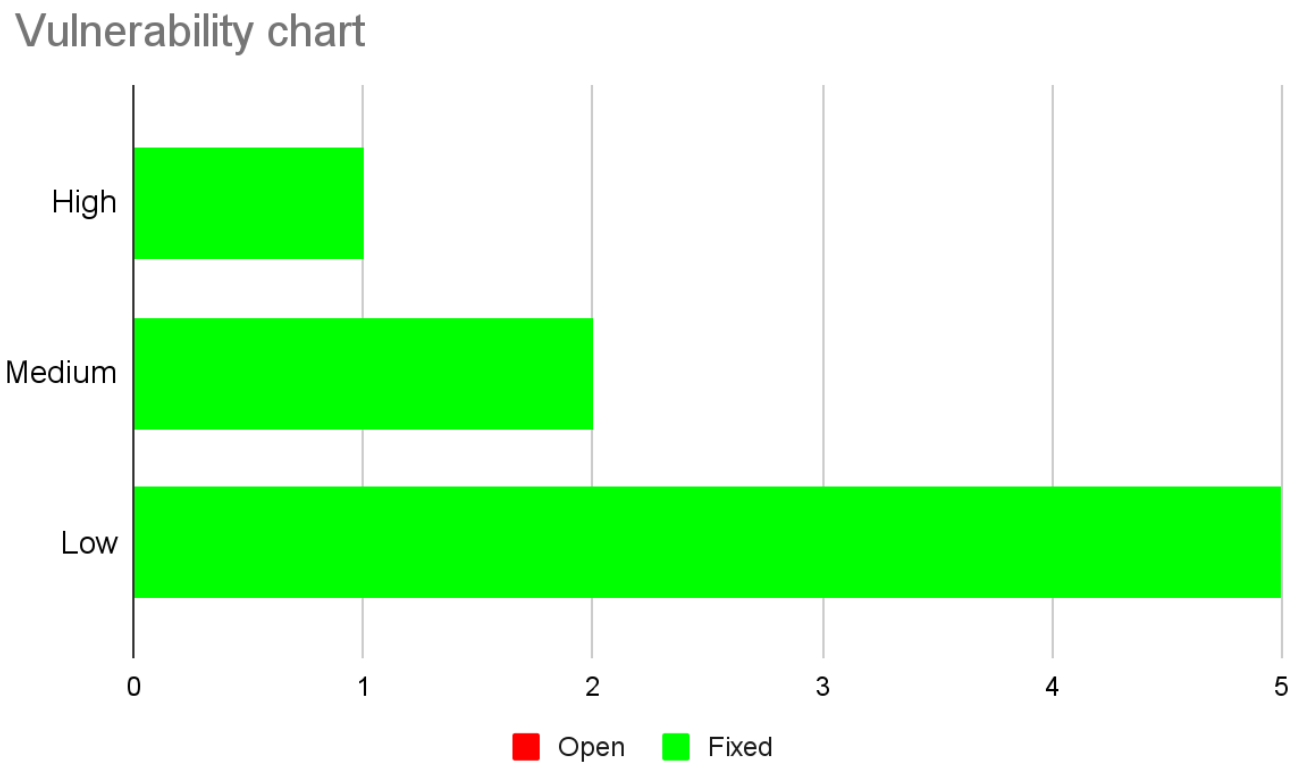
Summary of strengths

Building upon the strengths of the available implementation can help better secure it by continuing these good practices. In this case, a number of positive security aspects were readily apparent during the assessment:

- the code and project files are well structured, which makes them easy to read and maintain. The code is self-explanatory.
- the contracts perform only the declared functionality
- the main contract uses role-based access control
- verification errors have a custom explanation
- new compiler version was used

Summary of discovered vulnerabilities

During the assessment, two high-risk vulnerabilities were discovered, but during the manual penetration testing one of them was considered as a false positive. Overall, one high risk, two medium-risk, five low-risk vulnerabilities were discovered. All vulnerabilities were fixed after the initial audit. The chart below shows the distribution of findings.


Vulnerability chart
High
Medium
Low
0
1
2
3
4
5
Open
Fixed

Summary of high-risk vulnerabilities and recommendations

The recommendations below should be implemented as soon as possible since they provide for the effective and efficient mitigation of many of the high-risk issues identified. For more detailed information on all of the findings, refer to Appendix A: Detailed Findings) of the report.

Table 1: High-risk vulnerabilities

Risk rating	Finding name	Recommendation	Status
High	Local variables shadowing	Rename the local variables that shadow another component	FIXED

Summary of medium-risk vulnerabilities and recommendations

Table 2: Medium-risk vulnerabilities

Risk rating	Finding name	Recommendation	Status
Medium	Unprotected Function	The function can be called by anyone, and can mutate the state of the Forwarder smart contract. Consider access restriction only for trusted origin.	FIXED
Medium	DoS with Block Gas Limit	The function is gas sensitive because of the for loop. Consider setting a restriction on length of input data.	FIXED

Summary of low-risk vulnerabilities and recommendations

For more detailed information on all of the findings, refer to Appendix A: Detailed Findings.

Table 3: Low-risk vulnerabilities

Risk rating	Finding name	Recommendation	Status
Low	Missing events	Emit events when critical parameters change so it is possible to track off-chain changes in fee scheme.	FIXED
Low	Absence of zero address check	Check that the address is not zero before assigning variables in EURLToken smart contract.	FIXED
Low	Absence of zero address check	Check that the address is not zero before assigning variables in Forwarder smart contract.	FIXED
Low	Large contract size	EURLToken smart contract may not be deployable on mainnet. Consider enabling the optimizer (with a low "runs" value!), turning off revert strings, or using libraries.	FIXED
Low	Dependence on a predictable environment variable	It is recommended to use small deadline values.	FIXED

Security rating

Apriorit reviewed Lugh security posture in regards to the EURLToken and Forwarder smart contracts, and Apriorit consultants identified security strengths as well as vulnerabilities that create high, medium and low levels of risk. Taken together, the combination of asset criticality, threat likelihood, and vulnerability severity have been used to assign a grade for the overall security of the application. An explanation of the grading scale is included in the second table below.

In conclusion, Apriorit recommends that Lugh continue to follow existing good security practices and further improve their security posture by addressing all of the described findings.

	High	Medium	Low	Security	Grade
EUR-L token	0	0	0	Highly secure	A

Security grading criteria

Grade	Security	Criteria description
A	Highly secure	Exceptional attention to security. No high- or medium-risk vulnerabilities and few minor low-risk vulnerabilities.
B	Moderately secure	Good attention to security. No high-risk vulnerabilities and only a few medium- or several low-risk vulnerabilities.
C	Marginally secure	Some attention to security, but security requires improvement. A few high-risk vulnerabilities that can be exploited.
D	Insecure	Significant security gaps exist. A large number of high-risk vulnerabilities.

Code review and recommendations

Based on years of software development experience, Apriorit formed a list of best practices to write clear and understandable code. Following these best practices makes maintenance easier.

During the assessment, contract code was compared against our list of best practices. As a result of the code review, we formed the following recommendations.

1. FIXED Delete dead lines of code

Remove unused function calls because “dead code” is not used in the test flow, and makes the code's review and code maintenance more difficult.

Affected code:

- test/UpgradedToken.js L:17, L:21, L:31, L:43, L:64, L:66, L:68, L:70, L:84, L:87, L:90, L:93

2. FIXED Remove unused imports

It is detected importing unused libraries in the tests.

Affected code:

- test/TokenMetaTx.js L:4, L:7
- test/UpgradedToken.js L:3
- Contracts/Token.sol L:32

3. FIXED Use external modifier where it’s possible

Consider using the external visibility modifier for functions that are intended to be called only outside a smart contract and won't be called from within a declared contract. Potentially, it can reduce gas usage.

Example:

- Contracts/Token.sol, **payGaslessBasefee(address payer, address paymaster)** can be restricted to external

4. FIXED Use visibility modifiers for state variables

Even though **internal** is the default visibility for state variables it is a good practice to be explicit about visibility modifiers.

Affected code:

- Contracts/Token.sol L:53, L:54
- Contracts/Forwarder L:51, L:52

5. FIXED Use constants

Consider using constants instead of magical numbers.

Affected code:

- Contracts/Token.sol, decimals(), L:77
- Contracts/Token.sol, updateTxFeeRate(uint256 newRate), L:136
- Contracts/Token.sol, calculateTxFee(), L:152
- Contracts/Token.sol, _msgSender(), L:325
- Contracts/Token.sol, _msgData(), L:334

6. PARTIALLY FIXED Use mixed case for state variables

It is good practice to use respective naming conventions for variables: https://docs.soliditylang.org/en/v0.8.10/style-guide.html#local-and-state-variable-names

Affected code:

- Contracts/Blacklistable.sol L:68, L:76, L:85, L:90
- Contracts/Token.sol L:53, L:54, L:84, L:98, L:109, L:242, L:272

Secondary check: Contracts/Blacklistable.sol, L:90, _newBlacklister is not in the mixed case.

7. DECLINED Perform check on contract interface

Consider checking on contract interface in initialization logic of Forwarder smart-contract. Because Forwarder smart-contract depends on EURL token it would be convenient to check if an instance hidden by the eurlToken address implements the required interface to eliminate initialization with untrusted smart contracts.

See @openzeppelin/contracts-upgradeable/utils/introspection/IERC165Upgradeable.sol for details

Affected code:

- Contracts/Forwarder.sol, initialize(address eurlToken), L:73

Rejection reason: Since Lugh will originate the forwarder, there are no huge risks to implement such logic

8. FIXED zero address checks

Add checking on zero address before assigning state variables.

Affected code:

- Contracts/Forwarder.sol:L72
- Contracts/EURLToken.sol:L127
- Contracts/EURLToken.sol:L311

9. FIXED Add restriction on forward request registration

The Forwarder allows users to register a new request type and change the state of the smart contract. The call is gas sensitive because it contains a loop through the ‘typeName’. If a client doesn’t account for input size and forms an input with a long input string as a ‘typeName’, the transaction will fail due to an “out-of-gas” exception. This will result in wasted Ether. Consider restriction of access to ‘registerRequestType’ method only to an allowed origin.

Affected code:

- Contracts/Forwarder.sol: L120

Test coverage analysis

Unit tests are an essential part of smart contract development. They help to find problems in the code that are missed by the compiler before deploying the contract to the blockchain.

During the audit, the percentage of unit test coverage for each of the contracts was evaluated. The results are presented in the table below.

Contract	Initial coverage	Final coverage
Blacklistable.sol	100 %	100 %
Forwarder.sol	72 %	96 %
Token.sol	76 %	91 %

After the secondary review it was discovered that all sensitive exceptional scenarios were tested.

EURLToken contract uncovered test cases

Function	Description	Status
removeMinter, L:221-L:222	resetting of minter's allowance, revoking minter's role	CLOSED
updateMintingAllowance, L:233	changing of minting allowance for minter	CLOSED
updateTxFeeRate, L:136	setting of incorrect transaction fee	CLOSED
payGaslessBasefee, L:191, L:193	negative cases during forward execution flow: - checking on trusted forwarder - checking on ability to pay gasless base fee	CLOSED

Forwarder contract uncovered test cases

Function	Description	Status
verify, L:84, L:85	request verification flow: negative and positive cases	CLOSED
registerRequestType, L:122-L-128	new request type registration	CLOSED

Symbolic analysis

The results of the symbolic test execution detected one possible issue. Dependence on a predictable environment variable in EURLToken smart contract. See Appendix A for details.

Except for this issue the smart contracts are symbolic clean.

Static analysis

Static analysis of the contract was conducted using well known analysis tools.

Static analysis showed several possible errors such as:

1) delegated external call/multiple sends in a single transaction

2) unprotected initialization function

3) local variables shadowing

4) missed events

5) absence of zero address check

6) large contract size

Some of the vulnerabilities turned out to be false positives after a detailed analysis. See Appendix A for details.

Analysis of GAS use

The approximate GAS costs for each function of the `EURLToken` contract are:

Function	Estimated GAS use
constructor	153922
setMasterMinter	54684-90684
addMinter	30292-120104
removeMinter	29670 - 29850
updateMintingAllowance	26533 - 45733
mint	46279 - 76279
burn	43864
transfer	55964 - 75164
transferFrom	50978 - 65978
permit	43504 - 77684
setAdministrator	89238 - 165438
forceTransfer	43094 - 58094
setTrustedForwarder	28211 - 47423
updateGaslessBasefee	26828 - 46040
payGaslessBasefee	48601
setOwner	78685
transferOwnership	32796
setFeeFaucet	27981- 47181
updateTxFeeRate	26942 - 46142
unBlacklist	16448 - 27248
blacklist	28376 - 47576
pause	49335
unpause	18974
upgradeTo	51416

The approximate GAS costs for each function of the `Forwarder` contract are:

Function	Estimated GAS use
constructor	1144293
registerRequestType	Depends on input data. Usual request with “ForwardRequest” as input will cost 40293 - 59493
execute	118285 - 133281

Function that can be optimized:

- verify(Forwarder.ForwardRequest,bytes32,bytes32,bytes,bytes)
- execute(Forwarder.ForwardRequest,bytes32,bytes32,bytes,bytes)
- payGaslessBasefee(address,address)
- forceTransfer(address,address,uint256)

Consider using the external attribute for functions never called from the contract.

After the initial audit payGaslessBasefee method was marked as external which can help to optimize gas usage and reduce gas costs for a paymaster.

Appendixes

Appendix A. Detailed findings

Risk rating

Our risk ratings are based on the same principles as the Common Vulnerability Scoring System. The rating takes into account two parameters: exploitability and impact. Each of these parameters can be rated as high, medium, or low.

Exploitability — What knowledge the attacker needs to exploit the system and what preconditions are necessary for the exploit to work:

- **High** — Tools for the exploit are readily available and the exploit requires no specialized system knowledge.
- **Medium** — Tools for the exploit are available but have to be modified. The exploit requires specialized knowledge about the system.
- **Low** — Custom tools must be created for the exploit. In-depth knowledge of the system is required to successfully perform the exploit.

Impact — What effect will the vulnerability have on the system if exploited:

- **High** — Administrator-level access and arbitrary code execution or disclosure of sensitive information (private keys, personal information)
- **Medium** — User-level access with no disclosure of sensitive information.
- **Low** — No disclosure of sensitive information. Failure to follow recommended best practices does not result in an immediately visible exploit.

Based on the combination of parameters, an overall risk rating is assigned to a vulnerability.

Vulnerabilities discovered in the smart contract

Closed issues

High risk

Local variables shadowing

Description: The owner performs transfer of ownership and shadows the state variable _owner of OwnableUpgradeable smart contract.

Affected code:

Token.sol L:109, setOwner() shadows _owner OwnableUpgradeable.sol L:22

Recommendation:

Rename the local variable _owner that shadow another component

Details:

https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing

Medium risk

Unprotected Function

Description: The function registerRequestType unprotected and allows to mutate a state of smart contract by anyone.

Affected code:

Forwarder.sol, registerRequestType() L:120

Recommendation:

Consider access restriction only for trusted origin.

Medium risk

DoS with Block Gas Limit

Description: it is possible to perform DoS attacks on the gas sensitive function. The function is gas sensitive because of the for loop.

Affected code:

Forwarder.sol, registerRequestType() L:120

Recommendation:

Consider setting of restriction on parameter bytes(typeName).length

Low risk

Missing events

Description: Detected missing events for critical arithmetic parameters such as transaction txfee_rate and gasless_basefee.

Affected code:

Token.sol L:137, updateTxFeeRate()

Token.sol L:174, updateGaslessBasefee()

Recommendation:

Emit events when critical parameters change so it is possible to track off-chain changes in fee scheme.

Low risk

Absence of zero address check

Description:

Detected missing zero address validation.

Affected code:

Forwarder.sol L:72, initialize()

Recommendation:

Check that the address is not zero before assigning variables. Example:

```
require(addr != address(0))
```

Low risk

Absence of zero address check

Description:

Detected missing zero address validation.

Affected code:

Token.sol L:127, setFeeFaucet()

Token.sol L:311, setTrustedForwarder()

Recommendation:

Check that the address is not zero before assigning variables. Example:

```
require(addr != address(0))
```

Low risk

Large contract size

Description:

EURLToken contract code size is 31665 bytes and exceeds 24576 bytes

This contract may not be deployable on mainnet.

Recommendation:

Consider enabling the optimizer (with a low "runs" value!), turning off revert strings, or using libraries.

Low risk

Dependence on a predictable environment variable

Description:

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Affected code:

node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/draft-ERC20PermitUpgradeable.sol:57

require(block.timestamp <= deadline, "ERC20Permit: expired deadline")

Recommendation:

The issue can be considered as a false positive because of the difficulty of reproduction and minimal severity of such manipulation but it is recommended to use small deadline values.

Appendix B. Description of methodologies

Smart contract security checks

Apriorit uses a comprehensive and methodical approach to assess the security of blockchain smart contracts. We take the following steps to find vulnerabilities, expose weaknesses, and identify deviations from accepted best practices in assessed applications. Notes and results from these testing steps are included in the corresponding section of the report.

Our security audit includes the following stages:

1. **Discovery**. The first step is to perform reconnaissance and information gathering to decide how resources can be used in the most secure way. It is important to obtain a thorough understanding of the smart economics, the logic of smart contracts, and the environment they operate within so tests can be targeted appropriately. Within this stage, Apiorit completes the following tasks:
    a. Identifies technologies
    b. Analyzes the specification, whitepaper, and smart contract source base
    c. Creates a map of relations among smart contracts
    d. Researches the structure of smart contract storage
    e. Researches and analyzes standard implementations for functionality
2. **Configuration Management.** The configuration of the smart contracts is analyzed.
3. **User management and user permissions.** The majority of smart contracts have to manage individual users and their permissions. Most smart contracts split permissions between the contract owner, administrator, etc. Within this stage, Apiorit does the following:

a. Determines whether all functions can be called only by the expected role

b. Reviews user management functions and role assignment

c. Reviews permissions for each role

4. **Data validation**. Inputs to a smart contract from users or other smart contracts are its operational life-blood but are also the source of most high-risk attacks. These steps ensure that data provided to the application is treated and checked. All cases of invalid or unexpected data should be handled appropriately.

5. **Efficiency check**. Each function uses some amount of GAS during the call. In the case of a GAS shortage or overlimit, the smart contract function call will fail. Chipper smart contracts will be more interesting for users because no one wants to waste money, so all functions should be optimized in terms of GAS use.

6. **High-quality software development standards**. The standard requires teams to follow the best practices of coding standards. This will help Lugh avoid or mitigate the most common mistakes during development that lead to smart contract security vulnerabilities. It will also help with traceability and root cause analysis. This stage includes:

   a. Manual code review and evaluation of code quality

   b. Unit test coverage analysis