ackee | blockchain security

Monerium

V2

by Ackee Blockchain

23.5.2024

ackee | blockchain security

Contents

1. Document Revisions

0.1	Draft report	9.5.2024
1.0	Final report	16.5.2024
1.1	Fix review	23.5.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Monerium is a financial technology company that provides ERC-20 token equivalents for fiat money. The audit was performed for the second version of these tokens.

Revision 1.0

Monerium engaged Ackee Blockchain to perform a security review of the Monerium protocol with a total time donation of 5 engineering days in a period between May 2 and May 9, 2024, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `b5e2473` [1] and the scope was the whole `src` folder except for the `tokenfrontend.sol` file.

We began our review using static analysis tools, including Wake. This discovered two true positives (see I1 and I2 issues). We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. During the review, we paid special attention to:

- checking the correctness of the upgradeability pattern,
- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 11 findings, ranging from Info to High severity. The most severe one is the possibility to bypass validator rules (see H1 issue).

The contracts are centralized and need the trust of users. The system owners can potentially mint unlimited amounts of tokens and steal the existing tokens from users via M1 issue or H2 issue.

Ackee Blockchain recommends Monerium:

- abandon the TokenFrontend compatibility with the ControllerToken contract since it increases complexity and attack surface,
- create a more comprehensive test suite (including fuzz tests),
- address all reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The fix review was done on the given commit: `2705e9a` [2]. The scope of the review was the fixes for the issues found in the previous revision.

All issues were addressed and also W3 issue was remediated by fixing L1 issue.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

[1] full commit hash: `b5e2473bae9339e1349839962eebd10251553a46`

[2] full commit hash: `2705e9aa99eebd632b43532937bd8b8033fc24b1`

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
H1: Anyone can bypass validator rules	High	1.0	Fixed
H2: Token frontend can impersonate anyone	High	1.0	Fixed partially
M1: Insufficient signature validation	Medium	1.0	Fixed
L1: Insufficient data validation	Low	1.0	Fixed
W1: Missing constructor allows to call `initialize` on implementation	Warning	1.0	Fixed
W2: Renounce ownership	Warning	1.0	Fixed

	Severity	Reported	Status
W3: Validator can cause DoS	Warning	1.0	Acknowledged
W4: Double entrypoint for granting roles with different privileges	Warning	1.0	Fixed partially
I1: Unused using for statement	Info	1.0	Fixed
I2: Unused modifiers	Info	1.0	Fixed
I3: Two exposed initialize functions	Info	1.0	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

Token

The contract inherits from SystemRoleUpgradeable, MintAllowanceUpgradeable, and Openzeppelin's upgradable contracts. It implements ERC20 permit and transfers must be allowed by Validator. Minting is possible by System Accounts. By System Accounts, it is also possible to burn or recover tokens. There is no maximum limit for minting.

SystemRoleUpgradeable

The contract inherits from Openzeppelin's `Ownable2StepUpgradeable` and `AccessControlUpgradeable`. It is used to manage system roles and permissions (System Account, Admin Account and Owner).

MintAllowanceUpgradeable

The contract is used to limit mint allowance for System Accounts that are supposed to mint Token.

ControllerToken

The contract is for backward compatibility with the V1 contracts architecture

and new V2 contracts (for already deployed instances).

BlacklistValidatorUpgradeable

The contract represents an implementation of Validator. It is used to blacklist addresses for transfers of Token.

Actors

This part describes actors of the system, their roles, and permissions.

Owner

The owner can upgrade the Token contract, set the validator and maximal mint allowance. Also, the owner is initially `DEFAULT_ADMIN_ROLE` which allows him to set other roles. There can be only one owner account.

System Account

The system accounts can mint, burn, and recover tokens. There can be multiple system accounts.

Admin Account

The admin accounts can set a mint allowance for specified addresses (system accounts). There can be multiple admin accounts.

Token Frontend

The address that can call public functions on the ControllerToken.

Validator

The address that can blacklist addresses for transfers of Token.

5.2. Trust Model

Users have to trust Owner to set correctly all the elevated privileges and

protocol parameters. Malicious Owner, Admin Account, or System Account can lead to a total disaster and loss of funds. Malicious Validator can cause DoS (see W3: Validator can cause DoS).

H1: Anyone can bypass validator rules

High severity issue

Impact:	Medium	Likelihood:	High
Target:	ControllerToken.sol	Type:	Logic error

Description

For the networks where are already deployed V1 contracts, there is going to be deployed ControllerToken instead of Token as an implementation for the proxy. This contract is backward compatible with the legacy interface called Token Frontend. For this purpose, there are public entrypoints for the Token functions that forward the caller of the token frontend contract to the token contract. Such as for the `transfer` function there is `transfer_withCaller` function.

Listing 1. ControllerToken.transfer_withCaller

```
function transfer_withCaller(
    address caller,
    address to,
    uint256 amount
) external onlyFrontend returns (bool) {
    _transfer(caller, to, amount);
    return true;
}
```

However, this function does not contain validation from Validator.

Listing 2. Token.transfer

```
function transfer(
    address to,
    uint256 amount
) public override returns (bool) {
```

```
    require(
        validator.validate(_msgSender(), to, amount),
        "Transfer not validated"
    );
    return super.transfer(to, amount);
}
```

As a result, the banned addresses can bypass the validator rules by calling the `transfer_withCaller` function to transfer their tokens.

Exploit scenario

Bob is banned by the validator and uses the `transfer_withCaller` function to transfer his tokens to Alice.

Recommendation

Add the validation from the Token contract to all of the entrypoints in the ControllerToken contract to match the intended behavior.

Fix 1.1

The validation in the ControllerToken contract is now the same as in the Token contract.

Go back to Findings Summary

H2: Token frontend can impersonate anyone

High severity issue

Impact:	High	Likelihood:	Medium
Target:	ControllerToken.sol	Type:	Trust model

Description

The ControllerToken contract has special functions that forward the caller of the Token Frontend contract (out-of-scope) to the token contract. The most vulnerable are `transfer_withCaller` and `mintTo_withCaller` functions.

```
function transfer_withCaller(
    address caller,
    address to,
    uint256 amount
) external onlyFrontend returns (bool) {
    _transfer(caller, to, amount);
    return true;
}
```

```
function mintTo_withCaller(
    address caller,
    address to,
    uint256 amount
) external onlyFrontend onlySystemAccount(caller) returns (bool) {
    _useMintAllowance(caller, amount);
    _mint(to, amount);

    return true;
}
```

The Token Frontend address can be set by the contract owner.

```
function setFrontend(address _address) public onlyOwner {
```

```
    _getControllerStorage().frontend = _address;
}
```

So potentially, the Token Frontend contract can impersonate any address. That means stealing tokens from someone by calling the transfer with `caller` parameter as the address of the victim. Or impersonating System Account to mint tokens. If some of the system accounts have a bigger or unlimited allowance, then the attacker can mint a huge amount of tokens and destroy the whole token economy.

Exploit scenario

The frontend address is set to Alice. The system account A has an unlimited mint allowance. Alice calls `mintTo_withCaller(system_account_a, alice, 10**60)`.

Recommendation

Any flaw in the Token Frontend contract can lead to a serious security issue for the whole system because of this design decision. Ideally, stick to Token implementation and do not use Token Frontend as it increases the attack surface. If the Token Frontend is necessary, reconsider the design and make sure that the Token Frontend contract can not impersonate anyone.

Fix 1.1

The team is planning to deprecate the ControllerToken contract in the future, however, for a better user experience they prefer to use the legacy token frontend address during the transition. This address is already deployed so to prevent having an incorrect token frontend address, addresses of token frontends are hardcoded per ticker (these contracts are not upgradable) and per chain.

Listing 3. EthereumControllerToken.sol

```
function getFrontend() public view returns (address) {
    bytes3 t = ticker();
    if (t == 0x455552) {
        // EUR
        return 0x3231Cb76718CDeF2155FC47b5286d82e6eDA273f;
    } else if (t == 0x474250) {
        // GBP
        return 0x7ba92741Bf2A568abC6f1D3413c58c6e0244F8fD;
    } else if (t == 0x555344) {
        // USD
        return 0xBc5142e0CC5eB16b47c63B0f033d4c2480853a52;
    } else if (t == 0x49534b) {
        // ISK
        return 0xC642549743A93674cf38D6431f75d6443F88E3E2;
    } else {
        revert("Unsupported ticker");
    }
}
```

Go back to Findings Summary

M1: Insufficient signature validation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Token.sol	Type:	Logic error

Description

In the Token contract, signatures are used for permit allowance (implemented by Openzeppelin) and for `burn` and `recover` functions. For these functions, the signature passed as a parameter is checked against the hash that is also passed by the parameter. In the function body, there is only called `isValidSignatureNow` function with those parameters.

```
function burn(
    address from,
    uint256 amount,
    bytes32 h,
    bytes memory signature
) public onlySystemAccounts {
    require(
        from.isValidSignatureNow(h, signature),
        "signature/hash does not match"
    );
    _burn(from, amount);
}
```

That means System Account can use an arbitrary hash with a corresponding signature from someone to burn his tokens. Moreover, in the `recover` function, it is possible to burn his tokens and mint them for someone else.

```
function recover(
    address from,
    address to,
```

```
    bytes32 h,
    uint8 v,
    bytes32 r,
    bytes32 s
) external onlySystemAccounts returns (uint256) {
    bytes memory signature;
    if (r != bytes32(0) || s != bytes32(0)) {
        signature = abi.encodePacked(r, s, v);
    }
    require(
        from.isValidSignatureNow(h, signature),
        "signature/hash does not match"
    );
    uint256 amount = balanceOf(from);
    _burn(from, amount);
    _mint(to, amount);
    emit Recovered(from, to, amount);
    return amount;
}
```

Also, it is possible to replay this signature as many times as needed. As a result, if anyone exposed anywhere and anytime in the past some hash (or a preimage) with a corresponding signature then he is susceptible to being exploited by this issue.

Exploit scenario

Bob has 1000 EURe and sends a signed permit digest to Charlie.

```
permit_digest = m.getPermitDigest(bob, charlie, 10 * (10**decimals), nonce,
deadline)
signature = bob.sign_hash(permit_digest)
```

Charlie has a signature and calls `permit` function with the arguments from the permit digest what they agreed on. Malicious System Account sees the data on-chain and decides to steal Bob's tokens. The system account takes the data from the `permit` function to calculate the permit digest and signature.

With those parameters, the system account calls the `recover` function to steal Bob's tokens (for example all of them by front-running Charlie).

Recommendation

After a discussion with the team, the purpose of this signature validation is to increase trust between the user and the system. When users undergo KYC, they sign a specific message and the system stores the signature. If they lose access to their wallet, the system can recover funds for them thanks to KYC. However, users who are without KYC can be exploited by this issue.

The simple solution to mitigate possible exploitation is to add a hash digest. During the KYC signing, there will be signed a digest + message hash, where the digest will be for example `keccak256("Monerium KYC")`. Then if the system account needs to recover funds, he can't use any hash but only hashes with this digest. See the following test for a better explanation.

```
# simulate KYC registration
david = chain.accounts[7]
mint_erc20(m, david, 1000)
kyc_digest = keccak256(abi.encode_packed("Monerium KYC"))
kyc_message = keccak256(abi.encode_packed("some specific metadata"))
kyc_result = keccak256(abi.encode_packed(kyc_digest, kyc_message))
kyc_sig = david.sign_hash(kyc_result)
r = kyc_sig[:32]
s = kyc_sig[32:64]
v = int.from_bytes(kyc_sig[64:], 'big')

# >>> david lost access to his account <<<

# david asks off-chain for recover to his new account
new_david = chain.accounts[8]
# system account have stored kyc_message without digest and signature
m.recover(david, new_david, kyc_message, v, r, s, from_=system_acc)
assert m.balanceOf(new_david) == 1000
assert m.balanceOf(david) == 0
```

So the `recover` function validation could look the followingly.

```
bytes32 digest = keccak256(abi.encodePacked("Monerium KYC"));
bytes32 message = h; ①
bytes32 result = keccak256(abi.encodePacked(digest, message));
require(
    from.isValidSignatureNow(result, signature),
    "signature/hash does not match"
);
```

① The `h` is the input parameter and is saved to the local variable for a better explanation.

For the `burn` function, the validation should be also more restricted (potentially similarly) based on the functional requirements.

Fix 1.2

The issue was fixed by requiring an exact hash (in the both entrypoints).

```
require(

from.isValidSignatureNow(0x7eb17b1cb295cfefd09fe4c8604c0699503f9e35dd36bb15a4fed852a828f503, signature),
    "signature/hash does not match"
);
```

Go back to Findings Summary

L1: Insufficient data validation

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Token.sol	Type:	Logic error

Description

The contract is missing a zero-address check in the initialize function for the `validator` argument. Moreover, the check is also missing in the setter for this argument.

```
function setValidator(address _validator) public onlyOwner {
    validator = IValidator(_validator);
}
```

Exploit scenario

By accident, an incorrect value is passed to the setter. Instead of reverting, the call succeeds.

Recommendation

Add zero-address check to the setter and initialize function. Ideally, (since the validator address is a crucial parameter) utilize contract IDs.

The contract IDs help to correctly link contracts within the protocol. In this case, the Validator contract is assigned in the `initialize` and `setValidator` functions (in the Token contract).

For this purpose:

1. Define an ID for the Validator contract, eg: `bytes32 public constant`

`CONTRACT_ID = keccak256("Monerium Validator")`.

2. When setting the contract address, check that the contract ID matches:

```
require(
    IBase(_validator).CONTRACT_ID() == keccak256("Monerium Validator"),
    "Not Monerium Validator contract"
);
```

So, for example, the `setValidator` function could look followingly.

```
function setValidator(address _validator) public onlyOwner {
    require(
        IValidator(_validator).CONTRACT_ID() == keccak256("Monerium
Validator"),
        "Not Monerium Validator contract"
    );
    validator = IValidator(_validator);
}
```

This approach of validation will make the contract more resilient against incorrect values.

Fix 1.1

The contract ID based validation is now implemented.

Go back to Findings Summary

W1: Missing constructor allows to call `initialize` on implementation

Impact:	Warning	Likelihood:	N/A
Target:	BlacklistValidatorUpgradeable.sol	Type:	Logic error

Description

The contract is missing a constructor with the `_disableInitializers()` call. This allows unrestricted call of the `initialize` function on the logic contract (implementation). Any caller can become owner of the implementation and get privileged access to the logic contract. However, this privileged access does not affect maliciously the proxy.

Recommendation

Add a constructor with the `_disableInitializers()` call to prevent possible issues with future development.

Fix 1.1

The constructor is now implemented.

Go back to Findings Summary

W2: Renounce ownership

Impact:	Warning	Likelihood:	N/A
Target:	SystemRoleUpgradeable.sol	Type:	Logic error

Description

Owner role can be renounced. This presents a risk in the current scope since the owner is responsible for setting the validator and other roles. If the owner is no longer needed then the contract can be upgraded to a new implementation without upgrade capability and other privileged functions. Roles can be also renounced but since there can be multiple addresses with roles, the renounce capability might be useful and less risky.

Recommendation

Override `renounceOwnership` function to disable the renounce ownership capability.

Fix 1.1

The function is now overridden to disable the renounce ownership capability.

Go back to Findings Summary

W3: Validator can cause DoS

Impact:	Warning	Likelihood:	N/A
Target:	Token.sol	Type:	Trust model

Description

The transfer functions are calling the Validator contract before acting as usual. The address of the validator can be changed at any time (potentially involving front-running). The validator itself can be upgraded to arbitrary implementation and does not enforce having the same owner and roles as the Token contract. All these aspects can cause temporary DoS or even permanent when the issue is combined for example with W2: Renounce ownership.

```
// Override transfer function to invoke validator
function transfer(
    address to,
    uint256 amount
) public override returns (bool) {
    require(
        validator.validate(_msgSender(), to, amount),
        "Transfer not validated"
    );
    return super.transfer(to, amount);
}

// Override transferFrom function to invoke validator
function transferFrom(
    address from,
    address to,
    uint256 amount
) public override returns (bool) {
    require(validator.validate(from, to, amount), "Transfer not
validated");
    return super.transferFrom(from, to, amount);
```

```
}
```

Recommendation

Pay extra attention to the management of this address. The validator address could be checked with contract IDs (see L1: Insufficient data validation).

Go back to Findings Summary

W4: Double entrypoint for granting roles with different privileges

Impact:	Warning	Likelihood:	N/A
Target:	SystemRoleUpgradeable.sol, BlacklistValidatorUpgradeable.sol	Type:	Logic error

Description

In the SystemRoleUpgradeable contract there are two options to set the `ADMIN_ROLE` role. One of them is to call the `addAdminAccount` function.

```
function addAdminAccount(address account) public virtual onlyOwner {
    grantRole(ADMIN_ROLE, account);
    emit AdminAccountAdded(account);
}
```

The second option is to call the `grantRole` function directly.

```
function grantRole(bytes32 role, address account) public virtual
onlyRole(getRoleAdmin(role)) {
    _grantRole(role, account);
}
```

In the first case, there is a need to be Owner and also role admin of the `ADMIN_ROLE` role (for example the `DEFAULT_ADMIN_ROLE` role). For the second option, there is only a need to be role admin of the `ADMIN_ROLE` role. Access controls then don't make sense, because one function is more restrictive than the other and they are doing the same thing (except emiting the additional event).

Moreover, two entrypoints for granting roles are also in the

BlacklistValidatorUpgradeable contract. The `grantRole` function and `addAdminAccount` are as in previous examples. However, the `onlyOwner` modifier is missing in the `addAdminAccount` function.

```
function addAdminAccount(address account) public virtual  {
    grantRole(ADMIN_ROLE, account);
    emit AdminAccountAdded(account);
}
```

So the BlacklistValidatorUpgradeable contract needs only role admin of the `ADMIN_ROLE` role to add an admin account.

As a result, someone can potentially expect that only the owner can add admin accounts but there can be multiple other accounts that are eligible to add them. For example old owner (initial contract deployer).

Recommendation

Decide which access controls are needed and adjust the code to use only one type of privilege (if desired).

Fix 1.1

The `grantRole` function is now not used in the SystemRoleUpgradeable contract. The setter functions now use the internal `_grantRole` function. As a result, only the owner can add admin accounts. However, the BlacklistValidatorUpgradeable contract remained unchanged.

Also, it is important to note that the `grantRole` function is still available, however, default admin is not set in the current context. Therefore, the `grantRole` function is not usable in the current context.

Go back to Findings Summary

I1: Unused using for statement

Impact:	Info	Likelihood:	N/A
Target:	ControllerToken.sol	Type:	Dead code

Description

The ControllerToken contract has unused using for statement:

```
using SignatureChecker for address;
```

Unused code should be removed to make the contract more readable and maintainable.

Recommendation

Remove the unused using for statement.

Fix 1.1

The unused using for statement is now removed.

Go back to Findings Summary

I2: Unused modifiers

Impact:	Info	Likelihood:	N/A
Target:	MintAllowanceUpgradeable.sol, SystemRoleUpgradeable.sol	Type:	Dead code

Description

In the MintAllowanceUpgradeable contract the `onlyAllowedMinter` modifier is unused and in the SystemRoleUpgradeable contract the `onlyAdminAccount` modifier is unused. Unused code should be removed to make the contract more readable and maintainable.

Recommendation

Remove the unused modifiers.

Fix 1.1

The unused modifiers are now removed.

Go back to Findings Summary

I3: Two exposed initialize functions

Impact:	Info	Likelihood:	N/A
Target:	Token.sol	Type:	Code maturity

Description

In the Token.sol file is the Token contract with the initialize function that has a configurable name and symbol. Then the file contains contracts such as EURe.sol, GBPe.sol that looks followingly.

```
contract EURe is Token {
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(address _validator) public initializer {
        super.initialize("EUR", "EURe", _validator);
    }
}
```

As a result, there are two initialized functions with different arguments and it is possible to initialize the EURe contract with a different name and symbol than is intended. This does not pose any risk but if it is possible to initialize the EURe contract with a different name and symbol than is intended then the contract does not need to exist at all.

Recommendation

Override the initialize function to disallow the initialization with a different name and symbol.

Fix 1.1

The `EURe` and other contracts in the `Token.sol` file were removed.

Go back to Findings Summary

6. Report revision 1.1

There were added new contracts to the codebase.

- `EthereumControllerToken`
- `GnosisControllerToken`
- `PolygonControllerToken`

These contracts contain hardcoded addresses of the token frontends on the respective chains. They were added to the codebase as a remediation of the possible H2 issue (the flexible option, ControllerToken contract is still present).

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Monerium: V2, 23.5.2024.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

ackee | blockchain security

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://twitter.com/AckeeBlockchain