



Flare
HexWrappedTokens
Smart Contract Audit



Hex Wrapped Tokens Smart Contract Audit

Version: v231211

Prepared for: Flare

December 2023

Security Assessment




1. Executive Summary
2. Summary of Findings
 - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
 - 4.1 Security assumptions
 - 4.2 Decentralization
 - 4.3 Code quality and testing
5. Detailed Findings
 - FLWT-001 - Reviewers are forced to blindly vote for upgrades
 - FLWT-002 - A reviewer can prevent an upgrade from being executed
 - FLWT-003 - Arbitrary upgrades enabled by only 1 colluding reviewer
 - FLWT-004 - Testing coverage is low for several core contracts
 - FLWT-005 - Re-initializing the token in the event of an upgrade could fail

6. Disclaimer

File hashes

1. Executive Summary

In November 2023, Flare engaged Coinspect to review the smart contracts that implement the **Hex Wrapped Tokens**. The objective of the project was to evaluate the security of these upgradable tokens intended to wrap known tokens on the Flare blockchain.

 Solved	 Caution Advised	 Resolution Pending
High 0	High 0	High 0
Medium 3	Medium 0	Medium 0
Low 1	Low 0	Low 0
None 1	None 0	None 0
Total 5	Total 0	Total 0

Coinspect identified 3 medium-risk, 1 low-risk issues and 1 issue that poses no risk.

The first medium-risk issue, FLWR-001, mentions how upgrade reviewers are forced to blindly vote without knowing the new implementation. The next issue FLWR-002 is about the quorum system and how it fails when a vote is revoked. Lastly, FLWR-003 illustrates a collusion scenario that allows performing arbitrary upgrades.

The low-risk issue, FLWR-004 remarks the need to increase the test coverage.

In December 2023, **Coinspect** reviewed the fixes for the issues reported on this document.

2. Summary of Findings

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
FLWT-001	Reviewers are forced to blindly vote for upgrades	Medium
FLWT-002	A reviewer can prevent an upgrade from being executed	Medium
FLWT-003	Arbitrary upgrades enabled by only 1 colluding reviewer	Medium
FLWT-004	Testing coverage is low for several core contracts	Low
FLWT-005	Re-initializing the token in the event of an upgrade could fail	None

3. Scope

The source code review of **Flare's Wrapped Tokens** started on November 21st, 2023 and was conducted on the main branch of the git repository located at <https://github.com/HTMI-Ltd/flare-wrapped-token> as of commit `00e1ed17efa27837d310008732369b61659b114c`.

4. Assessment

The **Hex Wrapped Tokens** are meant to reflect valuable and well known assets from other chains such as **ETH**, **USDT**, and **USDC**. Each wrapped token is deployed using the same contracts, only differing in their metadata and decimals. Every **Wrapped Token** has the following properties:

- Upgradable (controlled by a voting system)
- Mintable and Burnable
- Blacklistable
- Permit
- Role-based Access Control
- Base **ERC20** features

4.1 Security assumptions

Each **Wrapped Token** inherits from modules or abstract contracts that provide the mentioned functionalities and properties. The wrapped token implementation is based on the recently released v5 contracts and libraries from **OpenZeppelin**.

The access control module is a forked implementation of **OpenZeppelin's**, mentioning the following changes on its NatSpec:

- Changes:
 1. Change initializers to remove `initialDelay`
 2. Remove time delay related functions: `_pendingDefaultAdminSchedule`, `_currentDelay`, `_pendingDelay`, `_pendingDelaySchedule`, `defaultAdminDelay`, `pendingDefaultAdminDelay`, `defaultAdminDelayIncreaseWait`, `changeDefaultAdminDelay`, `_changeDefaultAdminDelay`, `changeDefaultAdminDelay`, `_changeDefaultAdminDelay`, `rollbackDefaultAdminDelay`, `_rollbackDefaultAdminDelay`, `_delayChangeWait`, `_setPendingDelay`, `_isScheduleSet`, `_hasSchedulePassed`
 3. Remove `renounceRole()` function
 4. Remove `_pendingDefaultAdminSchedule` from `pendingDefaultAdmin` function
 5. Remove time delay elements from `_beginDefaultAdminTransfer` and `_setPendingDefaultAdmin`
 6. Change `_acceptDefaultAdminTransfer` to remove time delay elements

Wrapped Tokens are upgradable by using the UUPS proxy pattern, each module (abstract contract) fixes a storage slot as a constant variable to prevent future collisions and data corruption. Each slot has its own layout implemented through structs, retrieved every time a read or write operation is made. Initializing the implementation contract is disabled by calling `_disableInitializers` directly through the implementation's constructor.

Each upgrade is triggered according to a voting process where reviewers should vote for an ongoing upgrade request created by the upgrade admin. The upgrade admin sets the required quorum for the request to succeed. Coinspect identified that reviewers vote for an upgrade without knowing its implementation, blindly supporting it, FLWR-001.

Voters (reviewers) are also able to revoke a previously made vote. Coinspect detected that this feature mistakenly updates the quorum status, preventing the upgrade from being executed even if the current amount of votes exceed the minimum required threshold, FLWT-002. Also, Coinspect reported that a voting process can be manipulated with collusion to allow arbitrary upgrades at anytime, FLWT-003.

The token has a blacklist, where a privileged account can deny any other account from receiving or sending tokens at any time. Moreover, other **privileged account roles are able to mint and burn any amount of tokens at any time**, as described in the section below.

4.2 Decentralization

The project relies on an role-based access control system, segregating responsibilities and reducing the impact of attacks if a privileged account gets compromised.

The following roles and privileges are defined by the token:

Minter:

Set as `MINTER_ROLE` constant variable:

- Mint any non-zero amount of tokens to any non blacklisted and non-zero account, at any time.
- Can't be blacklisted.

Burner:

Set as `BURNER_ROLE` constant variable:

- Burn any non-zero amount of tokens from any account, at any time.
- Can't be blacklisted.

Blacklister:

Set as `BLACKLISTER_ROLE` constant variable:

- Add any non-zero account to the transfer blacklist, preventing that account from sending or receiving tokens. No privileged account (any account holding a role) can be blacklisted.
- Remove any non-zero account from the blacklist, reinstating the right to receive and send tokens.
- Can't be blacklisted.

Merchants:

Set as `MERCHANTS_ROLE` constant variable:

- Can't be blacklisted.

Upgrade Reviewers:

Set as `REVIEWERS_ROLE` constant variable:

- Vote for an upgrade request (vote), allowing a contract upgrade to be executed if the quorum is reached.
- Revoke the vote if previously voted for the current upgrade request.
- Can't be blacklisted.

Upgrade Admin:

Set as `UPGRADE_ADMIN_ROLE` constant variable:

- Submit (create) a new upgrade request.

- Authorize an upgrade request (once the quorum has been reached).
- Can't be blacklisted.

Default Admin:

Set as `DEFAULT_ADMIN_ROLE` constant variable:

- Directly grant and revoke any role, but itself.
- Start a two-step ownership transfer.
- Cancel an ongoing ownership transfer.
- Can't be blacklisted.

4.3 Code quality and testing

Overall the code was easy to read and understand. Accompanied with a detailed documentation and NatSpec on each relevant function and variable. In addition, the testing suite is straightforward and easy to work with. However, Coinspect considers that the overall branch coverage should be improved on some contracts `FLWT-002`.

5. Detailed Findings

FLWT-001

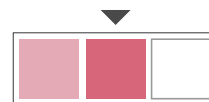
Reviewers are forced to blindly vote for upgrades

Status
Solved



Resolution
Fixed

Risk
Medium



Impact
High
Likelihood
Low

Location

`contracts/QuorumUpgradeable.sol`

Description

Upgrade reviewers have no information about the new implementation that will be used, granting to the `UPGRADE_ADMIN_ROLE` the power to perform an upgrade to an arbitrary implementation.

The users with the `REVIEWERS_ROLE` only vote if they agree to perform an upgrade, but there is no contract enforcement to guarantee the new implementation. This is are the parameters taken into account for an upgrade request:

```
struct Request {
    address requester;
    bool quorumReached;
    uint numConfirmationsRequired;
    address[] confirmers;
}
```

In other words, reviewers blindly trust in the upgrade's new implementation. This system could be made in a more robust way that considers also the new implementation's address prior its voting so the ecosystem and the reviewers are able check it.

The likelihood of this issue is considered low as the only one capable of abusing from this flaw is the user holding the `UPGRADE_ADMIN_ROLE` that is a trusted party.

Proof of Concept

The following case shows how two reviewers vote for an upgrade without knowing its implementation, allowing the `UPGRADE_ADMIN_ROLE` to destroy the token's proxy contract.

To run this script, place it on `test/QuorumTest.ts`.

Output:

```
HexWrappedToken Address: 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
External Code Size - Before Upgrade: 708
External Code Size - After Upgrade: 0
```

```
it("COINSPECT - Allows contract destruction due to blind voting",
  async () => {
    const emptyBytes: Uint8Array = new Uint8Array(0);
    const numConfirmationsRequired = 2;
    await
    contract.connect(authorized).submitUpgradeRequest(numConfirmationsRequired);

    let Factory = await
    ethers.getContractFactory("HexWrappedTestV2");
    const implementation = await Factory.deploy();

    await contract.connect(user).confirmUpgradeRequest();
```

```

    await contract.connect(user2).confirmUpgradeRequest();

    // Deploy or get your contract
    const NewMaliciousImpl = await
ethers.getContractFactory("NewMaliciousImpl");
    const maliciousImpl = await NewMaliciousImpl.deploy();
    await maliciousImpl.deployed();

    // Get the provider
    const provider = ethers.provider;
    // Get the code at the contract address
    let code = await provider.getCode(contract.address);
    // Calculate the code size
    let codeSize = (code.length - 2) / 2; // Subtract 2 for "0x",
divide by 2 because 2 chars represent one byte

    console.log("External Code Size - Before Upgrade:", codeSize);

    // Encode the function call
    const encodedData =
maliciousImpl.interface.encodeFunctionData("initialize");
    await
expect(contract.connect(authorized).upgradeToAndCall(maliciousImpl.addr
ess, encodedData))
        .to.emit(contract, "Upgraded")
        .withArgs(maliciousImpl.address);

    // Get the code at the contract address
    code = await provider.getCode(contract.address);
    // Calculate the code size
    codeSize = (code.length - 2) / 2; // Subtract 2 for "0x", divide
by 2 because 2 chars represent one byte

    console.log("External Code Size - After Upgrade:", codeSize);
});

```

Where the NewMaliciousImpl has the following implementation:

```

//SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract NewMaliciousImpl {
    address internal immutable self;

    constructor() {
        self = address(this);
    }

    function initialize() external {
        if(address(this) != self){
            selfdestruct(payable(address(0)));
        }
    }

    // Required to allow UUPS upgrade
    function proxiableUUID() external pure returns(bytes32){
        // ERC1967Utils.IMPLEMENTATION_SLOT

```

```
    return
    bytes32(0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382
    bbc);
}
```

Recommendation

Include the new implementation's address into the upgrade request. Also, check that this implementation's address is the same as the one used as a parameter when calling `upgradeToAndCall()`.


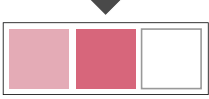
Status

Fixed on commit `f9a915fe038794b1bbf96ca8f76d44c0b05f2ce5`.

The implementation's address is now part of the upgrade request.

FLWT-002

A reviewer can prevent an upgrade from being executed

Status Solved	Risk Medium
	
Resolution Fixed	Impact High Likelihood Medium
Location <code>contracts/QuorumUpgradeable.sol</code>	

Description

The upgrade voting system mistakenly considers that the quorum is lost when a reviewer revokes their vote from an upgrade request, and as a consequence, an additional vote will be required to recover the quorum. This happens even if the total number of votes considerably exceed the request's threshold. In certain scenarios, this means a new upgrade request overriding the current one must be created, wasting the gas fees spent by those reviewers that voted on the previous request. This scenario also could be triggered spontaneously by an unsuspecting reviewer that only wants to revoke a previous vote.

Upgrades require the approval of a specific amount of reviewers, defined when creating the contract upgrade request:


```
function submitUpgradeRequest(uint _numConfirmationsRequired)
```

Then, when the amount of votes exceed the confirmations required for the current upgrade request, the internal quorum flag is set as true:

```
function confirmUpgradeRequest()
  external
  onlyRole(RoleConstant.REVIEWERS_ROLE)
  RequestExists
  notConfirmed(msg.sender)
{
  QuorumStorage storage $ = _getQuorumStorageLocation();
  $_request.confirmers.push(msg.sender);
  emit ConfirmUpgradeRequest(msg.sender);
  if (_checkQuorum()) {
    $_request.quorumReached = true;
    emit QuorumReached();
  }
}
```

However, the `quorumReached` flag is automatically set as `false` when any reviewer revokes its vote, even if the quorum was passed by more than one vote:

```
function revokeUpgradeRequest()
  external
  onlyRole(RoleConstant.REVIEWERS_ROLE)
  RequestExists
  alreadyConfirmed(msg.sender)
{
  QuorumStorage storage $ = _getQuorumStorageLocation();
  deleteArrayElement($_request.confirmers, msg.sender);
  if ($_request.quorumReached) {
    $_request.quorumReached = false;
  }
  emit RevokeUpgradeRequest(msg.sender);
}
```

If all reviewers have already voted, the account that has revoked their vote is the only one with the power to vote again in order to re-set `quorumReached` back to `true`, allowing an upgrade. This scenario forces the creation of a new upgrade request overriding the current one, wasting the gas fees spent by those reviewers that voted on the previous request.

Proof of Concept

The following test scenario shows how an upgrade request requiring 2 votes can be stopped by a reviewer even if the current quorum equals the number of confirmations required.

To run this script, place it on `test/QuorumTest.ts`.

```
it("COINSPECT - Can DoS an upgrade even if the quorum was reached",
  async () => {
    const reviewers = [user.address, user2.address, user3.address];

    const emptyBytes: Uint8Array = new Uint8Array(0);
    const numConfirmationsRequired = 2;
    await
contract.connect(authorized).submitUpgradeRequest(numConfirmationsRequired);

    let Factory = await
ethers.getContractFactory("HexWrappedTestV2");
    const implementation = await Factory.deploy();

    // Three reviewers vote and confirm the upgrade, just a quorum of
    2 was required
    await contract.connect(user).confirmUpgradeRequest();
    await contract.connect(user2).confirmUpgradeRequest();
    await contract.connect(user3).confirmUpgradeRequest();

    // The last one revokes
    await contract.connect(user3).revokeUpgradeRequest();

    // The upgrade reverts, even if there are still 2 votes with a
    quorum of 2.
    await expect(
contract.connect(authorized).upgradeToAndCall(implementation.address,
emptyBytes)
    ).to.be.revertedWithCustomError(contract, "NotSatisfied");
  });
```

Recommendation

When revoking a vote, use the current quorum as the condition for the conditional block.



Status

Fixed on commit `22469baa63b75c4f677b1c5a3a7de7d7b1fb41ea`.

The quorum is now reset by comparing the current amount of voters with its minimum value.

FLWT-003

Arbitrary upgrades enabled by only 1 colluding reviewer

Status Solved	Risk Medium
	
Resolution Fixed	Impact High Likelihood Medium
Location <code>contracts/QuorumUpgradeable.sol</code>	

Description

There are no minimum quorum requirements when submitting a new upgrade request, as a result, the `UPGRADE_ADMIN_ROLE` is able to collude with only one user with the `REVIEWERS_ROLE` to execute an arbitrary upgrade at any time.

When creating an upgrade request, the following checks are made to it's quorum:

```
modifier validRequirement(uint _numConfirmationsRequired) {
    QuorumStorage storage $ = _getQuorumStorageLocation();
    uint _reviewerCount =
    getRoleMemberCount(RoleConstant.REVIEWERS_ROLE);
    if (
        !(_reviewerCount != 0 &&
            _numConfirmationsRequired != 0 &&
```

```

        _numConfirmationsRequired <= _reviewerCount)
    ) revert NotValidRequirements();
    -;
}

```

This means, that as long as there are active reviewers, the number of confirmations is greater than zero and at most equal to the number of active reviewers, the request is valid. This condition enables the creation of requests requiring only one vote regardless the amount of reviewers. In other words, a system with 100 reviewers can have a request with a valid quorum of only 1 vote. This condition enables collusion scenarios that increase with the amount of reviewers.

Proof of Concept

The following case shows how an upgrade is executed after only 1 one even if the system has 3 reviewers.

To run this script, place it on `test/QuorumTest.ts`.

```

it("COINSPECT - Upgrades can be done at anytime with collusion", async
() => {
    const emptyBytes: Uint8Array = new Uint8Array(0);
    const numConfirmationsRequired = 1;
    await
contract.connect(authorized).submitUpgradeRequest(numConfirmationsRequi
red);

    let Factory = await ethers.getContractFactory("HexWrappedTestV2");
    const implementation = await Factory.deploy();

    // There are 3 potential reviewers, but the upgrader colludes with
the user2
    await contract.connect(user2).confirmUpgradeRequest();

    await
expect(contract.connect(authorized).upgradeToAndCall(implementation.add
ress, emptyBytes))
    .to.emit(contract, "Upgraded")
    .withArgs(implementation.address);
    expect(await contract.getVersion()).to.be.equal(2);
});

```

Recommendation

Enforce a minimum quorum calculated based on the current amount of reviewers.

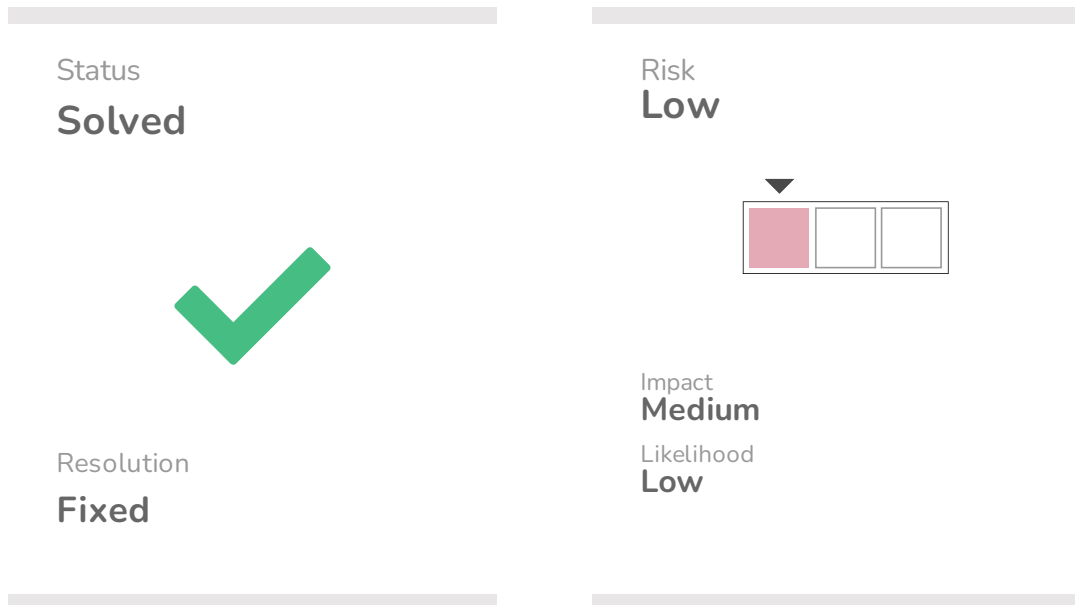
Status

Fixed on commit `05862fae979daf382c1b5e994551d0c9bbb4d770`.

An absolute majority (more than the half) is now enforced when setting the required confirmations upon upgrade request creation.

FLWT-004

Testing coverage is low for several core contracts



Description

The testing coverage for more than one core contract does not reach an acceptable threshold (over 95%), increasing the likelihood of encountering bugs or facing adversarial scenarios in the production phase.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	83.59	60	80	81.07	
(1)	54.29	38.46	52.63	45.65	... 219,232,233
AccessControlUpgradeable.sol	88.89	55.56	75	88.89	61,149,150,153
BlacklistableUpgradeable.sol	100	78.57	100	100	

(2)	100	72.22	100	100	
HexWrappedTest.sol	100	70	100	100	
QuorumUpgradeable.sol	95	55	93.75	94.23	70,71,72

- (1) AccessControlDefaultAdminRulesUpgradeable.sol
- (2) ERC20PermitWithRolesUpgradeable.sol

Recommendation



Increase the overall testing coverage.

Status

Fixed on commit [077167569c186fc4df5225858058ebed9855c625](#).

FLWT-005

Re-initializing the token in the event of an upgrade could fail

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Description

When performing an upgrade, if the target storage slot of the access control module is kept the same, the initialization reverts. This happens because there is a non-zero account previously set as the `_currentDefaultAdmin` for the current slot. However, if for some reason the target slot for the access control module is modified, the contract's initialization will not revert **allowing anyone to take the contracts over by initializing.**

```
//  
keccak256(abi.encode(uint256(keccak256("openzeppelin.storage.AccessControlDefaultAdminRules")) - 1)) & ~bytes32(uint256(0xff))  
    bytes32 private constant  
AccessControlDefaultAdminRulesStorageLocation =  
0xeef3dac4538c82c8ace4063ab0acd2d15cdb5883aa1dff7c2673abb3d8698400;  
  
function _getAccessControlDefaultAdminRulesStorage() private pure  
returns (AccessControlDefaultAdminRulesStorage storage $) {  
    assembly {
```



```

        $.slot := AccessControlDefaultAdminRulesStorageLocation
    }
}

function __HexWrappedToken_init(
    address _owner,
    string memory _name,
    string memory _symbol,
    uint8 _decimals
) internal onlyInitializing nonZA(_owner) {
    __AccessControlDefaultAdminRules_init(_owner);
    __Blacklistable_init();
    __Quorum_init();
    __ERC20PermitWithRoles_init(_name, _symbol, _decimals);
}
}

```

Where the access control's initialization ultimately calls `_grantRole()`:

```

function _grantRole(bytes32 role, address account) internal virtual
override returns (bool) {
    AccessControlDefaultAdminRulesStorage storage $ =
    _getAccessControlDefaultAdminRulesStorage();
    if (role == DEFAULT_ADMIN_ROLE) {
        if (defaultAdmin() != address(0)) {
            revert AccessControlEnforcedDefaultAdminRules();
        }
        $_currentDefaultAdmin = account;
    }
    return super._grantRole(role, account);
}

```

Coinspect observed that across the test suite, no function is being chained to be called when executing `upgradeToAndCall()`. This is because the access control slot is kept the same and the initialization will revert. This informational issue aims to warn that initialization will be required for future upgrades that change the target access control slot.

Recommendation

Consider documenting this scenario.

Status

Fixed on commit `b6d3f4bcfcb2135fc84f0188c858046c2f1c0229`.

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.

File hashes

Located at `./contracts/`:

```
8ac6bc120bbd614cb334d96f50b748226711f4de9adcc413aac0295c451be5e4 ./token/HexWrappedToken.sol
ebf1f515cad341b0e4c3d05920089470e5c3eea8f399bb3b3bbd3f92c54f0a20 ./token/HexWrappedUSDC.sol
2d12e34fcab09a4bc5023d39d280c8aad125379279b24d640a0d2b0779eccc66 ./token/HexWrappedUSDToken.sol
5d781fd5939c64040b27f74358bc471d32220aa63d3c352c964f62f44f52a6b6 ./token/HexWrappedETH.sol
82468ee4a5bdcd2898f4fa1821d98fd96cc199697a5945701858b65ac1b7153d ./BlacklistableUpgradeable.sol
94c374dd2f2b8e2834b9797f3390f73d76c3b9f15c36739658355b1eafa257c9 ./interface/IAccessControl.sol
9bcf3f95dea706a3a22a98b8bb0ac472f8e582d154faf7dfa6eabaa4d156fe1b
./interface/IAccessControlDefaultAdminRules.sol
6b579c84f4c96dfb8dd968eae5f4837512b25edfa4e32e8460df6bcbfc2f4f9 ./ERC20PermitWithRolesUpgradeable.sol
b49f1575435727b6c1afbd45b659878576349a4ece5cf85da97db34988d90d37 ./AccessControlUpgradeable.sol
9f057adde6a1410c90cd36710a495b6b8a95a585046c577d54699e194415fdbb ./utils/Context.sol
f206259f184abfb7b5e6f231bc17d5360150ca5e465b47adaf6b7e3aa5503ff5 ./utils/RoleConstant.sol
2f2bb4829adaad269e23bc842a602f403a185191f80b8cf3477a3707abb1b2cc
./AccessControlDefaultAdminRulesUpgradeable.sol
79298dddec35d1586ad19a723a1c90b8e8c51cb4eae4d09a6f5deb5f46a05fd54 ./QuorumUpgradeable.sol
```