ackee blockchain

Axelar

Ethereum Bridge

14 May, 2022

by Ackee Blockchain

Contents

1. Document Revisions

0.9	Draft report	May 14, 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 and 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 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 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
Miroslav Škrabal	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

Between 10. May and 13. May, Axelar engaged Ackee Blockchain to conduct a follow-up security review of the Solidity CGP Gateway project.

Initially, we were allocated three engineering days to audit new changes made to the protocol between the versions v3.1.1 and v3.2.2, particularly the gas optimizations and the new `GasReceiver` feature. However, in the middle of the audit, the scope was changed by Axelar to validate the new feature AxelarDepositService. After we provided quick feedback on the new feature, Axelar sent us a new commit addressing our findings. Therefore this audit was done on three different commits over three days, and as a result, we present only a draft report of our findings. With this approach, we tried to maximalize our support for Axelar during a turbulent market period.

The commits we worked on are the following:

1. protocol v3.2.2: `6c895ff`,
2. `GasReceiver` feature before feedback: `5d95c55`,
3. `GasReceiver` feature after feedback: `6a8bdd5`.

Our review resulted in 7 findings, ranging from Informational to Medium severity.

4. 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.

4.1. 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 border-line 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.

5. Findings

This section contains the list of discovered findings. Unless overriden 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 solve the underlying issue better (albeit possibly only with architectural changes) than others.

Because we performed this audit on three different commits, we assigned each target a number to link it with a corresponding commit:

1. protocol v3.2.2: `6c895ff` → 1,
2. `GasReceiver` feature before feedback: `5d95c55` → 2,
3. `GasReceiver` feature after feedback: `6a8bdd5` → 3.

Summary of Findings

Id		Type	Impact	Likelihood	Status
M1	M1: Upgradeability	Upgradeability	High	Low	Reported
M2	M2: External calls lack existance checks	Data validation	High	Low	Reported

Id		Type	Impact	Likelihood	Status
M3	M3: Token symbol and address decoupling	Data validation	High	Low	Reported
M4	M4: Token symbol length validation	Data validation	High	Low	Reported
W1	W1: Usage of `solc` optimizer	Usage of optimizer	Warning	N/A	Reported
W2	W2: Misleading error	Reporting	Warning	N/A	Reported
W3	W3: Event data validation	Data validation	Warning	N/A	Reported

Table 1. Table of Findings

M1: Upgradeability

Impact:	High	Likelihood:	Low
Target:	AxelarGasReceiverProxy.sol, Proxy.sol (1, 2, 3)	Type:	Data validation

Description

There are several issues with the current upgradeability mechanism:

1. `AxelarGasReceiverProxy.constructor` **and** `Proxy.constructor` **lack data validation for the implementation address.**

Recommendation

Add a getter to the implementations that returns a hash unique to the (project, contract) tuple, and check it on proxy construction. This will ensure the maximum possible data validation of the logic.

2. `AxelarGasReceiverProxy.fallback` **and** `Proxy.fallback` **lack an existence check for** `implementation`**.**

Recommendation

Add an existence check using `implementation.code.length`. This will ensure early error detection in case the implementation ceases to exist.

3. `AxelarGasReceiver.upgrade` **and** `Upgradable.upgrade` **have insufficient data validation of** `newImplementation`

Recommendation

Add a getter to the implementations that returns a hash unique to the (project, contract) tuple, and check it in the `upgrade` function. This will ensure the maximum possible data validation of the logic.

4. It may be possible to call functions other than `setup` and `_setup`, before `setup` and `_setup` are called.

Recommendation

Ensure that calling all state-changing public-entry points before `setup` results in no state changes that wouldn't be available after it is called. Alternatively, add a require to every state-changing public entry point that it cannot be called before `setup` is called.

5. Function shadowing is currently used for authorization

Description

Currently, the `setup` functions only have access control to ensure it is not called on the logic, but lack access control to ensure it is not called by an attacker on the proxy. This is currently done by shadowing the `setup` function in the proxy. This pattern is error-prone, and should be avoided.

Recommendation

Use a traditional way of access control, by require the setup function to be called only once on a single proxy instance.

Go back to Findings Summary

M2: External calls lack existance checks

Impact:	High	Likelihood:	Low
Target:	AxelarGasReceiver.sol (1)	Type:	Data validation

Description

External calls in `_safeTransfer` and `_safeTransferFrom` lack existence checks. While Solidity performs existence checks on all high-level calls, using a low-level call bypasses that check. On the EVM, calling accounts that don't contain code with arbitrary calldata results in a successful call. Calling a contract is usually intended to bring about a side-effect, and reverting early in that case will mean the undesired behavior is not propagated to the system.

Exploit scenario

A contract that is expected to be at that particular address self-destructs. The call to it returns success, which can lead to unintended consequences further down the line.

Recommendation

Short term, add existence checks using `account.code.length`. This will ensure that undesired behavior is not propagated to the system.

Long-term, do existence checks in all cases when using low-level calls, or document why the contract must exist, or a void call is expected behavior. This will ensure there are no surprises for the stakeholders of the system.

Go back to Findings Summary

M3: Token symbol and address decoupling

Impact:	High	Likelihood:	Low
Target:	AxelarDepositService.sol (2)	Type:	Data validation

Description

The function `handleTokenSend` receives `tokenSymbol` and `tokenAddress` as arguments. By using the `tokenAddress`, funds are transferred from the `DepositHandler` to the `DepositService` contract. `DepositService` approves those funds to be spent by the `AxelarGateway`. `sendToken` is called on the `AxelarGateway`, and this function receives only the `tokenSymbol`, not `tokenAddress`.

Therefore, the gateway manipulates the funds based on an address retrieved using the symbol. Such an address can differ from the one supplied in `handleTokenSend`. If the gateway has an allowance to spend the tokens, it can emit an event concerning funds that the originator of the transaction did not own.

Exploit scenario

An attacker supplies an address of a token that does not correspond to the supplied symbol. At the same time, he manipulates the allowances, e.g., after the logic of some function after an upgrade changes. Consequently, he could cause the gateway to emit an event concerning funds that he does not own.

Recommendation

Do not supply the `tokenAddress` to the `handleTokenSend` function. Instead, retrieve the address directly from the gateway using the provided `tokenSymbol`.

Go back to Findings Summary

M4: Token symbol length validation

Impact:	High	Likelihood:	Low
Target:	AxelarDepositService.sol (2)	Type:	Data validation

Description

The `_setup` function lacks data validation of the supplied `symbol`.

Additionally, the function `wrappedSymbol` makes assumptions about the length of the `symbol`:

```
// recovering string length as the last 2 bytes of the data
uint256 length = 0xff & uint256(symbolData);
// restoring the string with the correct length
assembly {
    symbol := mload(0x40)
    // new "memory end" including padding (the string isn't larger than 32
bytes)
    mstore(0x40, add(symbol, 0x40))
    // store length in memory
    mstore(symbol, length)
    // write actual data
    mstore(add(symbol, 0x20), symbolData)
}
```

Exploit scenario

A wrong symbol is passed to `setup`. Consequently, the `wrappedSymbol` can produce unexpected results.

Recommendation

Add better data validation to the `setup` function. For example:

```
bytes memory sBytes = bytes(symbol);
require(sBytes.length > 0 && sBytes.length < 31);
```

Go back to Findings Summary

W1: Usage of `solc` optimizer

Impact:	High	Likelihood:	Low
Target:	/* (1, 2, 3)	Type:	Compiler configuration

Description

The project uses the `solc` optimizer. Enabling the `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

W2: Misleading error

Impact:	Warning	Likelihood:	N/A
Target:	AxelarDepositService.sol (3)	Type:	Reporting

Description

In `sendToken` and in `sendNative` approvals are executed through the DepositReceiver, DepositReceiver:

```
(success, returnData) = depositReceiver.execute(
            wrappedTokenAddress,
            0,
            abi.encodeWithSelector(IERC20.approve.selector, gatewayAddress,
amount)
        );
```

And the return data is validated using:

```
if (!success || (returnData.length != uint256(0) && !abi.decode(returnData,
(bool)))) revert TransferFailed();
```

The `TransferFailed` error is raised if the validation condition is met. That might be misleading because the validation is concerned with `approval`, not `transfer`.

Exploit scenario

User calls `sendToken` or `sendNative`, and the error `TransferFailed()` is raised during the approval phase. As a result, debugging is more complicated.

Recommendation

Add a new `error` that would be more suitable for the given use case.

Go back to Findings Summary

W3: Event data validation

Impact:	Warning	Likelihood:	N/A
Target:	AxelarGasReceiver.sol (1)	Type:	Data validation

Description

The functions used for paying gas do not perform any data validation on data passed to the corresponding events. An attacker can pass carefully crafted malicious data to the functions, for example:

1. `msg.sender` differs from `sender`
2. `sender` or `refundAddr` addresses are address(0)
3. `payload` can contain malicious data
4. non-existent `destinationChain`
5. `amount` and `gasFeeAmount` can be in a wrong proportion

Exploit scenario

An event with malicious data is emitted in one of the functions used for paying gas. The observers do not perform careful validation and misinterpret the data. As a result, they perform undesirable actions.

Recommendation

Perform data validation on the parameters to be passed to the events.

Go back to Findings Summary

6. Appendix A

6.1. How to cite

Please cite this document as:

Ackee Blockchain, ”Report template”, May 14, 2022.

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

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

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

7. Appendix B

7.1. 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 1)
2. Add a `initializer` storage slot that gets set to `true` on initialization (see Listing 2). 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 3).

Listing 1. To be added to the logic contract

```
bool public initialized;

constructor() initializer {}
```

Listing 2. To be added to `initialize` on the logic contract

```
initialized = true;
```

Listing 3. 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

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq