ackee blockchain

Axelar

Forecall and Auth modules

by Ackee Blockchain

19.12.2022

Contents

1. Document Revisions

0.1	Draft report	December 15, 2022
1.0	Final report	December 19, 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, 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 Woke 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. 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	Medium	-
	Low	Medium	Medium	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
Miroslav Škrabal	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

Axelar is a cross-chain communication protocol. It allows for token transfers and cross-chain contract calls.

Revision 1.0

Axelar engaged Ackee Blockchain to perform a security review of the Forecall and Auth modules with a total time donation of 5 engineering days in a period between December 5 and December 16, 2022 and the lead auditor was Miroslav Škrabal.

The audit has been performed on the following three pull requests:

1. Forecall Service, 32a6172
2. Auth gas optimizations, 7e07cad
3. Forecallable, c1fcd7f

The first introduced the Axelar Forecall service (and corresponding changes in the Gas Service), the second optimizations to the Auth module and the third added the Forecallable implementation for client applications.

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

- reuse of admin signatures,
- signature malleability,
- detection of reentrancies,
- proper data validation,
- adherence to solidity best practices,

- upgradeability process,
- compatibility with non-standard tokens,
- different call sequences of execute and forecall in Forecall Service,
- borrow/return mechanism in Forecall Service,
- detection of possible slot collision in Forecallable,
- the correctness of access controls.

Our review resulted in 14 findings, ranging from Info to Medium severity.

Ackee Blockchain recommends Axelar:

- pay attention to data validation in constructors,
- address all the reported issues.

See Revision 1.0 for the system overview of the codebase.

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 *Solution*.

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
M1: Dangerous ownership transfer	Medium	1.0	Reported
M2: Forecallable constructor data validation	Medium	1.0	Reported
M3: ForecallService constructor data validation	Medium	1.0	Reported
M4: Tx can revert due to a slot collision	Medium	1.0	Reported
M5: Returning borrowed tokens	Medium	1.0	Reported
W1: Usage of `solc` optimizer	Warning	1.0	Reported

	Severity	Reported	Status
W2: Lack of amount validation in token transfer functions	Warning	1.0	Reported
W3: Integration of tokens with fees on transfer	Warning	1.0	Reported
W4: Integration of tokens with blacklisting	Warning	1.0	Reported
W5: Lack of events	Warning	1.0	Reported
I1: Make reused code into a library	Info	1.0	Reported
I2: Typo in contractId	Info	1.0	Reported
I3: Unused code	Info	1.0	Reported
I4: Validation of token address	Info	1.0	Reported

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.

AxelarForecallService.sol

AxelarForecallService is used to make cross-chain calls faster. It allows both cross-chain calls and cross-chain asset transfers. In case of transfers the tokens are lent to the destination contract and are returned after the classical GMP call is finalized. It is a contract designed to be used by trusted partners and it is maintained by an Axelar's microservice.

AxelarForecallable.sol

AxelarForecallable is a base contract that provides the functionality and an interface for the forecall functionality. It provides functions to receive (or borrow) and return (or repay) tokens.

Upgradeable.sol

Upgradeable is an abstract contract that provides the functionality to upgrade the logic contract. It is used by AxelarForecallService.

Actors

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

Owner

The AxelarForecallService contract is owned by the owner. The owner has permission to upgrade the logic contract. The ownership can also be transferred to another address.

forecallOperator

The AxelarForecallService can only be operated by the forecallOperator. The operator can trigger forecalls and withdraw tokens from the contract. The operator can also be changed to another address via an upgrade.

5.2. Trust model

The current trust model for the AxelarForecallService has issues from the service-user (service has trust assumptions about the user) side, which were described in an individual issue Returning borrowed tokens.

From the user-service (user has trust assumptions about the service) side, the service can provide arbitrary data in the forecalls and the user applications have to trust that the data is valid. This could become problematic if the service is compromised.

M1: Dangerous ownership transfer

Medium severity issue

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

Description

`Upgradeable.sol` allows the owner to transfer the ownership to another address.

```
function transferOwnership(address newOwner) external virtual onlyOwner {
        if (newOwner == address(0)) revert InvalidOwner();

        emit OwnershipTransferred(newOwner);
        // solhint-disable-next-line no-inline-assembly
        assembly {
            sstore(_OWNER_SLOT, newOwner)
        }
    }
```

However, the transfer function does not have a robust verification mechanism for the proposed owner address. If a wrong owner address is accidentally passed to it, the error cannot be recovered. Thus passing a wrong address can lead to irrecoverable mistakes.

Exploit scenario

The current owner Alice wants to transfer the ownership to Bob. Alice calls the `transferOwnership` function but supplies the wrong address by mistake. As a result, the ownership will be passed to the wrong and possibly dead address.

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

The current owner Alice wants to transfer the ownership to Bob. The two-step process would have the following steps:

- Alice proposes a new owner, namely Bob. This proposal is saved to a variable candidate.
- The candidate Bob calls the function for accepting the ownership, which verifies that the caller is the new proposed candidate.
- If the verification passes, the function sets the caller as the new owner.

If Alice proposes the wrong candidate, she can change it. However, it can happen, though with an extremely low probability, that the wrong candidate is malicious (most often, it would be a dead address) and is able to accept the ownership in the meantime.

An authentication mechanism can also be employed to prevent the malicious candidate from accepting the ownership, though such a mechanism is almost never used in practice.

Go back to Findings Summary

M2: Forecallable constructor data validation

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	AxelarForecallable.sol	Type:	Data validation

Description

AxelarForecallable lacks data validation in its constructor.

```
constructor(address gateway_, address forecallService_) {
        if (gateway_ == address(0) || forecallService_ == address(0))
revert InvalidAddress();

        gateway = IAxelarGateway(gateway_);
        forecallService = forecallService_;
    }
```

Exploit scenario

An incorrect value of `gateway` or `forecallService` is passed to the constructor. Instead of reverting, the call succeeds. If such a mistake is not discovered quickly and the contracts are not redeployed, the protocol can behave in an undefined way.

Recommendation

The address corresponding to the `foreCallService` can be very easily validated by checking the `contractId` of the service.

The gateway currently doesn't have a `contractId` but it could be added. Optionally, the gateway could be queried for some constant value that is known not to change through upgrades.

Long term, Axelar could deploy a library or an additional service that would validate whether the provided address corresponds to a valid Axelar component.

Go back to Findings Summary

M3: ForecallService constructor data validation

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	AxelarForecallService.sol	Type:	Data validation

Description

AxelarForecallable lacks data validation in its constructor.

```
constructor(address gateway_, address forecallOperator_) {
        gateway = IAxelarGateway(gateway_);
        forecallOperator = forecallOperator_;
    }
```

Exploit scenario

An incorrect value of `gateway` or `forecallOperator` is passed to the constructor. Instead of reverting, the call succeeds. If such a mistake is not discovered quickly and the contracts are not redeployed, the protocol can behave in an undefined way.

Recommendation

The same recommendation from Forecallable constructor data validation applies here. Validating an EOA is harder, at least this might include a zero address check.

Go back to Findings Summary

M4: Tx can revert due to a slot collision

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AxelarForecallable.sol	Type:	Code logic

Description

AxelarForecallable stores the digest of the forecall data using a keccak hash and sstore:

```
forecallSlot = keccak256(abi.encode(PREFIX_FORECALL, sourceChain,
sourceAddress, payload));
```

When a forecall is called, the function checks that the digest isn't currently used, ie that the message wasn't already forecalled:

```
function forecall(
        string calldata sourceChain,
        string calldata sourceAddress,
        bytes calldata payload
    ) external {
        _authForecall(sourceChain, sourceAddress, payload, msg.sender);

        (bytes32 forecallSlot, address forecaller) =
_getForecallData(sourceChain, sourceAddress, payload);

        if (forecaller != address(0)) revert AlreadyForecalled();
        _setForecaller(forecallSlot, msg.sender);
```

However, the digest is made only from PREFIX_FORECALL, sourceChain, sourceAddress, payload, which arent't guaranteed to be unique across different commands. And thus the transaction can incorrectly revert.

Exploit scenario

Multiple transactions with the same forecall data are initiated on the same source chain from the same source address. If those are sent to the AxelarForecallService contract, the first one will succeed, but the remaining revert due to a slot collision.

Recommendation

It is recommended to supply a nonce for each forecall, and use it to generate the digest.

Go back to Findings Summary

M5: Returning borrowed tokens

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	AxelarForecallService.sol	Type:	Trust model

Description

AxelarForecallService allows for faster message passing and token transfers. In the case of token transfers it borrows the tokens to the destination contract (because the forecall is faster than the classical GMP).

The destination contract is expected to return the tokens, but this is not properly enforced programmatically and is based on pure trust.

There are multiple issues that can arise when returning the tokens:

1. The destination contract doesn't contain the return logic in the first place.
2. The destination contract contains the return logic, but is upgradeable and thus allows for changing the return logic in the future.
3. The destination contract can overwrite the forecaller slot with `address(0)`. If this happens the execute function won't execute the return.
4. The destination contract might not have enough balance to return the tokens, eg because tokens with transfer fees are used.

The Axelar's development team suggested possible mitigations to this issue. Firstly, moving the foreCallWithToken and executeWithToken functions to the proxy instead of the logic contract. And additionally, moving the forecaller slot manipulation to the AxelarForecallService.

Though these are certainly steps in the right direction, they are still

problematic. The logic contract can be upgraded to contain self-destruct. The self-destruct can be triggered through proxy. If the proxy was deployed as a metamorphic contract then it can be redeployed with the same address and different bytecode. Additionally, this fix requires the contract to be deployed with the correct proxy in the first place, which requires additional checks and effort after deployment.

Exploit scenario

AxelarForecallService borrows tokens to the destination contract. The destination contract isn't properly audited to contain the return logic (or is upgradeable). The destination contract overwrites the forecaller slot with `address(0)` or upgrades the logic of the execute functions. As a result, the tokens are not returned.

Recommendation

Moving the logic to the proxy contract and maintaining the forecaller slot in the AxelarForecallService contract is a step in the right direction, but it doesn't solve the issue completely.

To have full confidence that the tokens will be returned the client contracts must be audited and the deployed bytecode has to match the audited source code. If the contract is upgradeable the service should contain a whitelist mechanism of approved logic contracts. If the contract upgrades, the service should notice this and prevent transferring tokens before the audit of the new logic contract is completed.

If the client contract contains a self-destruct or a delegate call it can metamorph, i.e. it can change its bytecode and maintain the address. Such contracts should not be allowed to be white-listed in the first place.

Go back to Findings Summary

W1: 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

W2: Lack of amount validation in token transfer functions

Impact:	Warning	Likelihood:	N/A
Target:	AxelarForecallService.sol	Type:	Data validation

Description

AxelarForecallService does not contain any data validation on the amount fields. The service is expected to borrow tokens up to a certain amount, but currently, the amount is validated only off-chain.

```
if (commandId == bytes32(0)) {
            _safeTransfer(gateway.tokenAddresses(tokenSymbol),
contractAddress, amount);

IAxelarForecallable(contractAddress).forecallWithToken(sourceChain,
sourceAddress, payload, tokenSymbol, amount);
        } else {
            if (gateway.isCommandExecuted(commandId)) {

IAxelarForecallable(contractAddress).executeWithToken(commandId,
sourceChain, sourceAddress, payload, tokenSymbol, amount);
            } else {
                _safeTransfer(gateway.tokenAddresses(tokenSymbol),
contractAddress, amount);

IAxelarForecallable(contractAddress).forecallWithToken(sourceChain,
sourceAddress, payload, tokenSymbol, amount);
            }
        }
```

Exploit Scenario

By a mistake, the operator of the service sends a higher amount of tokens to be borrowed than intended. The service does not perform any validation on

the amount and the tokens are transferred to the destination contract.

Recommendation

Mitigation of this issue is to perform data validation on the amount field. Ideally, this would include querying an oracle for the current price of the token and validating that the value is not higher than the maximum value that can be borrowed.

Go back to Findings Summary

W3: Integration of tokens with fees on transfer

Impact:	Warning	Likelihood:	N/A
Target:	AxelarForecallService.sol, AxelarForecallable.sol	Type:	Denial of service

Description

The AxelarForecallService supports arbitrary tokens. The transfers are performed on the addresses passed by the operator. Some tokens have fees on transfer (e.g. STA). If such a token is passed to the service, the tokens might be not returned at all or returned in a lower amount.

Exploit scenario

The operator of the service passes a token with fees on transfer to the service. When those tokens are to be returned the following scenarios can happen:

1. If the gateway mints the tokens, the tokens are returned in a lower amount (because of the fee on the transfer from Forecallable to the service).
2. If the gateway transfers the tokens, the tokens are not returned at all because the Forecallable contract will not have enough tokens to transfer and the transfer will revert.

Recommendation

Either the service should not support tokens with fees on transfer or the service should require the destination contract to compensate the fees on transfer.

Go back to Findings Summary

W4: Integration of tokens with blacklisting

Impact:	Warning	Likelihood:	N/A
Target:	AxelarForecallService.sol, AxelarForecallable.sol	Type:	Denial of service

Description

The AxelarForecallService supports arbitrary tokens. The transfers are performed on the addresses passed by the operator. Some tokens support the blacklisting of addresses. If the destination address gets blacklisted, the funds will not be returned.

Exploit scenario

The service sends tokens to the destination contract. In the meantime, the destination contract gets blacklisted. As a result, the funds are not returned to the service.

Recommendation

Consider the possibility of blacklisting the destination address. If such a possibility is not acceptable, don't support tokens that support blacklisting.

Go back to Findings Summary

W5: Lack of events

Impact:	Warning	Likelihood:	N/A
Target:	AxelarForecallService.sol, AxelarForecallable.sol	Type:	Logging

Description

The AxelarForecallable.sol contract doesn't emit events on any of the important actions. This makes it hard to monitor that the tokens are borrowed and returned correctly.

Exploit scenario

The destination forecallable contract doesn't return borrowed tokens. Due to a lack of logging, it takes a longer time to detect this issue.

Recommendation

Add events to the functions that relate to borrowing and returning tokens. Logging will help to monitor anomalies and detect issues faster.

Go back to Findings Summary

I1: Make reused code into a library

Impact:	Info	Likelihood:	N/A
Target:	AxelarForecallService.sol, AxelarForecallable.sol	Type:	Code copying

Description

AxelarForecallService and AxelarForecallable (and other Axelar components) contain the same functions `_safeTransfer` and `_safeTransferFrom`. Those functions are copied each time they are needed.

Recommendation

To avoid code duplication, the functions should be moved to a library. This will have the following benefits:

- the code will be easier to maintain, as it will be in a single place,
- the code will be easier to audit, as it will be in a single place,
- no copy-paste error will be possible,
- the code will be more readable and elegant.

Go back to Findings Summary

I2: Typo in contractId

Impact:	Info	Likelihood:	N/A
Target:	AxelarForecallService.sol	Type:	Quality assurance

Description

AxelarForecallService has the following `contractId`:

```
function contractId() external pure returns (bytes32) {
        return keccak256('axelar-forecaller-service');
}
```

Such an ID is inconsistent with the other contracts, which use the following format: function contractId() external pure returns (bytes32) {

```
return keccak256('axelar-deposit-service');
    }
```

Recommendation

Change the `contractId` to `axelar-forecall-service` to maintain consistency and readability.

Go back to Findings Summary

I3: Unused code

Impact:	Info	Likelihood:	N/A
Target:	IAxelarForecallService.sol	Type:	Unused code

Description

IAxelarForecallService declares the following error:

```
error AlreadyExecuted();
```

However, this error is never used in the contract.

Recommendation

If the error is not planned to be used anywhere, it should be removed.

Go back to Findings Summary

I4: Validation of token address

Impact:	Info	Likelihood:	N/A
Target:	AxelarForecallService.sol	Type:	Data validation

Description

The AxelarForecallService supports the callWithToken method which allows forecalling with the supplied token.

The service fetches the token address from the gateway and calls `_safeTransfer`.

```
  _safeTransfer(gateway.tokenAddresses(tokenSymbol), contractAddress,
amount);

IAxelarForecallable(contractAddress).forecallWithToken(sourceChain,
sourceAddress, payload, tokenSymbol, amount);
```

If the given token is not supported by the gateway, `address(0)` will be returned and passed to `_safeTransfer`. There the function will revert with an ambiguous error message.

Recommendation

Check that the token is supported by the gateway before forecalling, i.e. check that the returned token address is not `address(0)`.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar: Forecall and Auth modules, 19.12.2022.

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

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq