Axelar

Create3

by Ackee Blockchain

28.2.2023

Contents

1. Document Revisions

1.0	Final report	February 28, 2023

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
Jan Kalivoda	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 token transfers and cross-chain contract calls. The new feature of the protocol, called Create3, allows deploying contracts with predictable addresses without the bytecode affecting the address.

Revision 1.0

Axelar engaged Ackee Blockchain to perform a security review of the Create3 feature in the Axelar protocol with a total time donation of 3 engineering days in a period between February 20 and February 23, 2023, and the lead auditor was Jan Kalivoda.

The audit has been performed on the commit `f1aafa8` and the scope was the following:

- contracts/upgradable/FinalProxy.sol
- contracts/deploy/Create3.sol
- contracts/deploy/Create3Deployer.sol
- contracts/express/ExpressExecutable.sol
- contracts/express/ExpressProxy.sol
- contracts/express/ExpressProxyDeployer.sol
- contracts/express/ExpressRegistry.sol

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. The contracts were tested with Woke testing framework. During the review, we paid special attention to:

- checking the correctness of the Create3 algorithm,

- checking the correctness of Create3 usage,
- ensuring the arithmetic of the system is correct,
- 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 5 findings, ranging from Info to High severity. The most severe one is H1: Final upgrade can not be executed that causes the FinalProxy contract can not be upgraded to the final implementation (the main feature of FinalProxy).

Ackee Blockchain recommends Axelar:

- do not use function shadowing for security-related functionality,
- address all other 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
H1: Final upgrade can not be executed	High	1.0	Reported
M1: Registry deploy front-running	Medium	1.0	Reported
W1: Usage of `solc` optimizer	Warning	1.0	Reported
W2: Potentially dangerous function shadowing	Warning	1.0	Reported
I1: Missing NatSpec documentation	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.

FinalProxy

The contract inherits from Axelar's Proxy contract. It introduces a new feature of a final implementation. The implementation that is considered final (the `finalUpgrade` function is called) can not be changed because it uses [Create3](#) algorithm for deployment.

Create3

The library that contains the deploy logic. It combines the CREATE2 and CREATE1 algorithms to deploy the contract. The input for the deploy function is salt and bytecode. Firstly, it checks if the contract is already deployed and if the bytecode has zero length. Then it deploys a temporary [CreateDeployer](#) contract using CREATE2.

Listing 1. The deployer contract

```
contract CreateDeployer {
    function deploy(bytes memory bytecode) external {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            pop(create(0, add(bytecode, 32), mload(bytecode)))
        }
        selfdestruct(payable(msg.sender));
    }
}
```

The contract is then called and it deploys the passed bytecode using CREATE1 and immediately self-destructs itself. Lastly, the library function checks if the bytecode is deployed with the `deployedAddress` function. An example of deployment can be seen in Appendix C.

Create3Deployer

The contract uses Create3 library to deploy contracts. It has functions for deploy, deploy and call, and a function to obtain the address of the deployed contract.

ExpressExecutable

The abstract contract to inherit for contracts that want to use the express functionality. It comes with express service enabled by default and provides `execute` and `executeWithToken` functions without any validation (unlike AxelarExecutable that needs approval from Axelar gateway).

ExpressProxy

The contract provides functionalities to execute GMP express messages in the final destination (shadows the implementation functions). Functionalities for properly handling messages are extended to the registry contract.

ExpressProxyDeployer

The contract is used to deploy the ExpressProxy contract. It uses Create3 library to deploy the proxy and registry atomically.

ExpresRegistry

The contract is used for registering and processing calls with tokens. The functions are callable only by a proxy address.

Actors

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

Proxy Owner

The role with permission to upgrade the implementation.

5.2. Trust model

The current trust model for the Express Service has issues from the service-user (service has trust assumptions about the user) side. The destination contract is expected to return the tokens, but this is not enforced programmatically and is based on pure trust.

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

H1: Final upgrade can not be executed

High severity issue

Impact:	Medium	Likelihood:	High
Target:	FinalProxy.sol	Type:	DoS

Listing 2. Excerpt from FinalProxy.finalUpgrade

```
        if (implementation() != address(0)) revert AlreadyInitialized();
```

Listing 3. Excerpt from FinalProxy.implementation

```
    function implementation() public view override(BaseProxy, IProxy)
 returns (address implementation_) {
        implementation_ = _finalImplementation();
        if (implementation_ == address(0)) {
            implementation_ = super.implementation();
        }
    }
```

Description

The `finalUpgrade` function can not be executed if the returned value from the `implementation()` function is non-zero (see Listing 2, Listing 3). Since the `implementation()` function has a fallback to `super.implementation()`, it will cause the return value will be a non-zero address. That means all added functionality by FinalProxy to Proxy is useless.

For completeness, the only moment when the `implementation()` function will return zero is when the proxy is created with zero address as implementation and then `finalUpgrade` function can be called once.

Exploit Scenario

The proxy is used for some time and is stable, so Bob wants to choose the final implementation to have more efficient proxying. He calls the `finalUpgrade` function, but it fails because the `implementation()` function returns a non-zero value.

Recommendation

Remove the check, because the library already handles this case. From the contract’s address and hardcoded salt, can not be the final implementation changed twice.

Go back to Findings Summary

M1: Registry deploy front-running

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	ExpressProxy.sol	Type:	Front-running

Listing 4. Excerpt from ExpressProxy.deployRegistry

```
    function deployRegistry(bytes calldata registryCreationCode)
external {
        Create3.deploy(
            REGISTRY_SALT,
            abi.encodePacked(registryCreationCode, abi.encode(
address(gateway), address(this)))
        );
    }
```

Description

The ExpressProxy contract contains the `deployRegistry` function that allows anyone to deploy a new registry (see Listing 4). For deployment, it is using Create3. As a consequence of that, it is possible to call it only once, because the address of the proxy and salt can not be changed. However, if the `deployRegistry` function is not called right after the deployment, it can be front-ran and as a result, the contract can have a registry with an arbitrary code that can not be changed.

Exploit Scenario

Bob runs a badly written deployment script that deploys ExpressProxy contract and after that (in another transaction) calls the `deployRegistry` function. However, Mallory front-runs the transaction and calls the `deployRegistry` function with a different code. As a result, Bob's contract has a registry with Mallory's code and he did not notice that.

Recommendation

Ensure that the `deployRegistry` function is called atomically during the contract creation. For example, put the access control modifier on it and allow only the contract creator to call it.

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: Potentially dangerous function shadowing

Impact:	Warning	Likelihood:	N/A
Target:	ExpressExecutable.sol, ExpressProxy.sol	Type:	Function shadowing

Listing 5. Excerpt from [ExpressExecutable.executeWithToken](#)

```
    /// @notice this function is shadowed by the proxy and can be called only internally
    function executeWithToken(
        bytes32,
        string calldata sourceChain,
        string calldata sourceAddress,
        bytes calldata payload,
        string calldata tokenSymbol,
        uint256 amount
    ) external {
        _executeWithToken(sourceChain, sourceAddress, payload, tokenSymbol, amount);
    }
```

Listing 6. Excerpt from [ExpressProxy.executeWithToken](#)

```
    /// @notice used to handle a normal GMP call when it arrives
    function executeWithToken(
        bytes32 commandId,
        string calldata sourceChain,
        string calldata sourceAddress,
        bytes calldata payload,
        string calldata tokenSymbol,
        uint256 amount
    ) external override {
        registry().processExecuteWithToken(commandId, sourceChain, sourceAddress, payload, tokenSymbol, amount);
    }
```

Description

The ExpressExecutable contract has its execute functions without any data validation (see Listing 5). According to the documentation, it relies on the proxy code (see Listing 6), since it should shadow these functions. However, if the proxy is deployed without these functions, the execute functions can be called directly on the logic contract through this proxy. This can lead to the execution of arbitrary code.

Recommendation

Do not use function shadowing for access controls or any other security-related functionality. Otherwise, ensure that the proxy contract contains these functions.

Go back to Findings Summary

I1: Missing NatSpec documentation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Best practices

Description

The NatSpec documentation is missing. Only a few functions are imperfectly documented. Functions should contain for example an explanation for function parameters and return values. Proper documentation is important for code reviews and further development.

Recommendation

Cover all contracts and functions with the NatSpec documentation.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar: Create3, 28.2.2023.

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.

Appendix C: Woke tests

The following test shows an example of deploying the registry with Create3:

```
def test_create3():
    default_chain.tx_callback = lambda tx: print(tx.console_logs)

    # accounts
    owner = default_chain.accounts[0]
    default_chain.default_tx_account = owner

    # deploys
    Create3.deploy()
    gw = MockGateway.deploy()
    impl = MyExpressMock.deploy(gw)
    ep = ExpressProxy.deploy(impl, owner, bytes(), gw)

    # create3 algorithm
    createDeployerAddr = get_create2_address_from_code(ep,
ep.REGISTRY_SALT(), CreateDeployer.deployment_code())
    create3CalculatedAddr = get_create_address(createDeployerAddr, 1)
    assert ep.registry().address == create3CalculatedAddr

    # get interface
    reg = ExpressRegistry(ep.registry())

    # check if deployed
    with must_revert((Error, eth_abi.exceptions.InsufficientDataBytes)) as
e:
        reg.gateway() # contract does not exist

    # deploy
    registryCreationCode = ExpressRegistry.deployment_code()
    ep.deployRegistry(registryCreationCode)
    reg.gateway() # contract exists

    # final check
    epd = ExpressProxyDeployer.deploy(gw)
    assert epd.isExpressProxy(ep.address) == True
```

ackee

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq