ackee | blockchain security

Axelar

CGP & GMP SDK refactor

by Ackee Blockchain

15.02.2024

ackee | blockchain security

Contents

1. Document Revisions . . . 4
2. Overview . . . 5
   2.1. Ackee Blockchain . . . 5
   2.2. Audit Methodology . . . 5
   2.3. Finding classification . . . 6
   2.4. Review team . . . 8
   2.5. Disclaimer . . . 8
3. Executive Summary . . . 9
   Revision 1.0 . . . 9
   Revision 2.0 . . . 10
4. Summary of Findings . . . 12
5. Report revision 1.0 . . . 14
   5.1. System Overview . . . 14
   L1: Double events on `transferOperatorship` . . . 15
   W1: Final upgrade logic can be bypassed . . . 17
   W2: Different revert logic in AxelarGasService . . . 18
6. Report revision 2.0 . . . 20
   6.1. System Overview . . . 20
   6.2. Actors . . . 21
   6.3. Trust Model . . . 22
   M1: Nonce does not protect against replaying failed batches . . . 23
   L2: `msg.data` used in `keccak256` . . . 25
   W3: NatSpec comments not used . . . 27
   W4: Incorrect NatSpec parameters . . . 28
   W5: The reuse of the same nonce and signatures on another chain may confuse off-chain components . . . 30

W6: The documentation does not mention that signers and signatures should be sorted . . . 31
W7: The check of the minimal length of the proof is incorrect . . . 32
W8: The `chainName` input argument to the constructor can be an empty string . . . 33
W9: In `executeCalls`, nonces may be reused between different calls . . . 34
W10: The `withdraw` function is marked `payable` . . . 35
Appendix A: How to cite . . . 36
Appendix B: Glossary of terms . . . 37

1. Document Revisions

1.0	Final report	8.9.2023
2.0	Re-audit final report	15.2.2024

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 Wake 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 fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz 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	Low	-
	Low	Medium	Low	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
Andrey Babushkin	Lead Auditor
Jan Kalivoda	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.

Revision 1.0

Axelar engaged Ackee Blockchain to perform a security review of the changes between specific versions in Axelar cross-chain gateway and Axelar GMP SDK with a total time donation of 3 engineering days in a period between September 4 and September 7, 2023, and the lead auditor was Andrey Babushkin.

The audit has been performed on the changes between:

- v5.0.0 and v6.1.0 on Axelar cross-chain gateway [1]
- v4.0.3 and v5.3.0 on Axelar GMP SDK [2]

The changes were mostly adding inline documentation, reordering events, moving some contracts and a few refactorings.

We began with a manual review of all the changes and then moved to tests with Wake testing framework. During the review, we paid special attention to:

- ensuring refactoring does not affect the logic,
- ensuring the event reorder does not cause incorrect event emitting,
- detecting possible new issues that could be introduced by the changes.

Our review resulted in 13 findings, ranging from Warning to Medium severity.

Ackee Blockchain recommends Axelar:

- address the reported issues.

Revision 2.0

Axelar engaged Ackee Blockchain to perform a security review of the CGP & GMP SDK refactor with a total time donation of 5 engineering days in a period between January 29 and February 14, 2024, and the lead auditor was Andrey Babushkin.

The review was done on multiple commits from different pull requests:

- 8d2dfc2 from PR#125,
- 44fba61 from PR#128,
- c79e7f2 from PR#129.

The scope was changed multiple times during the time slot of the audit. The initial scope consisted of only one pull request #125 on the commit `d1dfbe7919ac570edc79dfedf3baeef116745a38`, however, the code could not be compiled. The new version of the commit reviewed in this revision was fixed five days after the faulty version. Later, Axelar engaged to extend the scope onto two additional pull requests, #128 and #129. We do not consider this a good approach and utilization of the audit slot. We recommend implementing better planning and always delivering only tested and compilable source code for the audit review in one commit.

We began our review by using static analysis tools, namely Wake. During our review, we paid special attention to:

- verifying the correct functionality of the interchain multisig contracts,
- checking if there are possibilities of replay attacks,
- validating if the weight mechanism does not malfunction and the protocol cannot enter an incorrect state,
- evaluating the overall code quality,

- and checking for common issues like reentrancy, access controls and data validations.

Our review resulted in 10 findings, ranging from Informational to Medium severity. Most of the auditing time was spent on fuzz testing and manual code review. This approach resulted in finding the M1 issue.

Ackee Blockchain recommends Axelar:

- implement the protection mechanism against replay attacks for failed batches,
- check if all conditions for validating input data are in place and correct,
- validate the correctness of the NatSpec documentation,
- improve the internal testing and development cycle,
- address the other reported issues.

[1] Commits range: `9c7a260c848011f27d6e7ecb1cba88de79206ccc` - `134d264144b5adc32c42b1389618e86c91fbdc75`

[2] Commits range: `928371eb4410c2bb2295a7889f1827a0703b72fc` - `c20941bbed15dd2c571fc5578b98f8a39a653dbd`

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 *Fix*.

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
L1: Double events on `transferOperatorship`	Low	1.0	Reported
W1: Final upgrade logic can be bypassed	Warning	1.0	Reported
W2: Different revert logic in AxelarGasService	Warning	1.0	Reported
M1: Nonce does not protect against replaying failed batches	Medium	2.0	Reported
L2: `msg.data` used in `keccak256`	Low	2.0	Reported
W3: NatSpec comments not used	Warning	2.0	Reported

	Severity	Reported	Status
W4: Incorrect NatSpec parameters	Warning	2.0	Reported
W5: The reuse of the same nonce and signatures on another chain may confuse off-chain components	Warning	2.0	Reported
W6: The documentation does not mention that signers and signatures should be sorted	Warning	2.0	Reported
W7: The check of the minimal length of the proof is incorrect	Warning	2.0	Reported
W8: The `chainName` input argument to the constructor can be an empty string	Warning	2.0	Reported
W9: In `executeCalls`, nonces may be reused between different calls	Warning	2.0	Reported
W10: The `withdraw` function is marked `payable`	Warning	2.0	Reported

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

For the system overview look into our past reports where is described each component.

L1: Double events on `transferOperatorship`

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	AxelarGateway	Type:	Events

Listing 1. Excerpt from AxelarGateway.transferOperatorship

```
708     function transferOperatorship(bytes calldata newOperatorsData, bytes32)
    external onlySelf {
709         emit OperatorshipTransferred(newOperatorsData);
710
711         IAxelarAuth(authModule).transferOperatorship(newOperatorsData);
712     }
```

Listing 2. Excerpt from AxelarAuthWeighted. transferOperatorship

```
87         emit OperatorshipTransferred(newOperators, newWeights, newThreshold);
88     }
```

Description

The codebase contains only the `AxelarAuthWeighted` contract that matches the `IAxelarAuth` interface. Based on these conditions, the contract always emits a similar event two times.

Exploit scenario

The `transferOperatorship` function is called and it emits two similar events.

Recommendation

Decide which event to keep and remove the other one.

Go back to Findings Summary

W1: Final upgrade logic can be bypassed

Impact:	Warning	Likelihood:	N/A
Target:	FinalProxy	Type:	Uncontrolled delegatecall

Description

The FinalProxy should serve as a feature for a final update of the implementation. However, this feature can be bypassed if the final implementation address has no code. This can be done via a phenomenon called metamorphic contracts. If the implementation contains `SELFDESTRUCT` or `DELEGATECALL` then there is a possibility of destroying the contract and then calling the `finalUpgrade` function again.

Exploit scenario

Bob calls the `finalUpgrade` function for implementation called MyFinalImplementation. He calls a function containing `SELFDESTRUCT` on the MyFinalImplementation. Then he can call the `finalUpgrade` function again on a new implementation[1].

Recommendation

Take into consideration if it is not an issue.

Go back to Findings Summary

W2: Different revert logic in AxelarGasService

Impact:	Warning	Likelihood:	N/A
Target:	AxelarGasService	Type:	Revert behavior

Description

The reverting behavior changed for the `payNativeGasForContractCall` and `payNativeGasForContractCall` functions. This can be possibly an issue if some of the integrators rely on the revert for `msg.value == 0`.

Listing 3. Old behavior

```
function payNativeGasForContractCall(
    address sender,
    string calldata destinationChain,
    string calldata destinationAddress,
    bytes calldata payload,
    address refundAddress
) external payable override {
    if (msg.value == 0) revert NothingReceived();
    emit NativeGasPaidForContractCall(sender, destinationChain,
destinationAddress, keccak256(payload), msg.value, refundAddress);
}
```

Listing 4. New behavior

```
    function payNativeGasForContractCall(
        address sender,
        string calldata destinationChain,
        string calldata destinationAddress,
        bytes calldata payload,
        address refundAddress
    ) external payable override {
        emit NativeGasPaidForContractCall(sender, destinationChain,
destinationAddress, keccak256(payload), msg.value, refundAddress);
    }
```

The same changes apply to the `payNativeGasForContractCallWithToken`

function.

Recommendation

Take this into consideration and appropriately inform the integrators of the change. Update the documentation.

Go back to Findings Summary

[1] e.g. called MyFinalFinalImplementation for completness

6. Report revision 2.0

6.1. System Overview

The scope consists of multiple interconnected contracts. These contracts contain the logic for multi-signature execution of arbitrary calls. The multisig mechanism is weighted, i.e. all signers have specified weights. For a call batch to be executed, the sum of weights corresponding to each signer, for whom the signature is acquired, should be greater or equal to a certain threshold. The threshold is specified together with the weights and signers' addresses.

ECDSA

The library implements secure signature verification and is mostly inspired by the OpenZeppelin library with the same name. Furthermore, it has the `toEthSignedMessageHash` function that builds a non-transactional Ethereum Signed Message from any `bytes32` hash.

BaseWeightedMultisig

The base contract with the core logic of the weighted multisig solution. It keeps track of all signers' epochs and the retention period set in the constructor. The retention period specified how many previous epochs of sighers may be used for verification. Except for signature verification using the `validateProof` function, the contract has the internal `_rotateSigners` function that creates a new epoch of a new set of signers, weights and a threshold.

The contract is declared abstract. Its implementation is based on the similar implementation from the Axelar CGP project.

InterchainMultisig

This contract extends the BaseWeightedMultisig contract to allow for batch execution of arbitrary calls by any party having a list of valid signatures by approved signers. This batch execution is made by the `executeCalls` function, which is the only entry point to the contract—all other functions can only be called as targets in the multi-call batch. The retention period is set to 0, which means that only the last signers can verify calls.

The contract supports receiving and withdrawing of Ether, however, the `withdraw` function is protected to only be called from `executeCalls`.

AxelarServiceGovernance

This contract, which is a part of the Axelar Governance system, was extended to include the newly implemented multisig solution that replaced the usage of `BaseMultisig`. The corresponding interface `IAxelarServiceGovernance` was also modified.

This contract is a part of the extended scope from pull request #128.

AxelarGatewayWeightedAuth

The newly implemented contract implements the BaseWeightedMultisig contract with a retention period of 15 epochs. This contract is used on the Axelar Gateway to verify signed commands. The contract is `Ownable`, the owner can add new signer sets by calling the `transferOperatorship` function. The corresponding `IAxelarGatewayWeightedAuth` interface is also declared.

This contract is a part of the extended scope from pull request #129.

6.2. Actors

Valid Signer

A party that can sign messages with a call batches hash. The address of the party is a part of a set of signers within the current retention period.

Invalid Signer

A party whose signatures cannot be used for verification. Valid Signer becomes Invalid Signer if it is either in a signer set outside of the current retention period or in case it is not in any signer set.

User

This is an address that can call the `executeCalls` function. If the address has a list of valid signatures from approved signers, the call batch will be executed. Otherwise, the function will revert, and no state changes will happen.

AxelarGatewayWeightedAuth's Owner

The deployer of the AxelarGatewayWeightedAuth contract. It is enabled to add new signer sets for the call verification on the Axelar Gateway.

6.3. Trust Model

The multisig solution makes the trust more distributed and decentralized. However, AxelarGatewayWeightedAuth has a simple ownership mechanism, which defeats the purpose of multisig. Users should trust the owner that they will not add malicious signers.

M1: Nonce does not protect against replaying failed batches

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	InterchainMultisig.sol	Type:	Logic

Description

In InterchainMultisig, call batches are implemented to fail if one of the calls fails. The attacker may cause the batch to fail by making signers sign a transaction with a withdrawal to a malicious contract. If the batch fails, nothing will happen. However, the nonce and the call batch will not be marked as executed. Therefore, the attacker may switch the malicious contract to a non-failing mode and perform the replay attack using the same set of signatures and nonce.

Exploit Scenario

1. Eve makes a valid transaction to send a token from a source chain to a destination chain. The destination address is set to a contract that implements the `IInterchainExecutable` interface and reverts on every call.
2. The transaction is signed by signers, and, with a corresponding nonce, is sent to the gateway.
3. The destination chain call reverts as expected, and Eve refunds the sent tokens on the source chain.
4. Eve switches her contract into a receiving mode.
5. Eve executes the same call batch with the known nonce and signatures.
6. Eve receives the asset twice: on the source chain and the destination chain.

Recommendation

Keep track of used nonces or implement the expiry date for signatures.

Go back to Findings Summary

L2: `msg.data` used in `keccak256`

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	BaseMultisig.sol	Type:	Logic

Description

The contract `BaseMultisig` computes the `keccak256` value from `msg.data` to record a vote for a multisig function call.

Listing 5. Excerpt from BaseMultisig._isFinalSignerVote

```
150         bytes32 topic = keccak256(msg.data);
151         uint256 epoch = signerEpoch;
152         Voting storage voting = votingPerTopic[epoch][topic];
```

However, a single function call with given arguments may be represented by multiple `msg.data` payloads. This is for the following reasons:

- types shorter than 256 bits encoded into 256-bit words may contain dirty higher-order bits,
- this is only possible with ABI encoder v1, which is not currently used in the file,
- function call payload may contain additional data after properly encoded arguments.

This may lead to an issue when some untypical function calls would be treated as a vote for a different topic.

Exploit Scenario

Due to a bug in a client or a library, a function call with extra data is sent to

the contract. The extra data is not relevant to the function call, but it is included in the `msg.data` and is used to compute the `keccak256` value. As a consequence, the user with the misbehaving client or library may unintentionally vote for a different topic.

Recommendation

Make sure the risk of using `msg.data` to compute the `keccak256` value is acceptable. If not, consider using a different approach to record votes.

Go back to Findings Summary

W3: NatSpec comments not used

Impact:	Warning	Likelihood:	N/A
Target:	BaseWeightedMultisig, SafeNativeTransfer	Type:	Code quality

Description

The project uses normal comments instead of NatSpec comments in `BaseWeightedMultisig.sol` and `SafeNativeTransfer.sol` even though the comments follow the syntax of NatSpec comments.

Recommendation

Change `//` to `///` and `/ to /*` to convert normal comments to NatSpec comments.

Go back to Findings Summary

W4: Incorrect NatSpec parameters

Impact:	Warning	Likelihood:	N/A
Target:	BaseWeightedMultisig	Type:	Code quality

Description

Some functions in the `BaseWeightedMultisig` contract contain incorrect parameter names in the attached comments. When converted to NatSpec comments, the compiler raises compilation errors for the following functions.

Listing 6. Excerpt from BaseWeightedMultisig.validateProof

```
64     /*
65      * @notice This function takes messageHash and proof data and reverts if
   proof is invalid
66      * @param messageHash The hash of the message that was signed
67      * @param weightedSigners The weighted signers data
68      * @param signatures The signatures data
69      * @return isLatestSigners True if provided signers are the current ones
70      */
71     function validateProof(bytes32 messageHash, bytes calldata proof) public
   view returns (bool isLatestSigners) {
```

Listing 7. Excerpt from BaseWeightedMultisig._rotateSigners

```
100     /*
101      * @notice This function rotates the current signers with a new set of
    signers
102      * @param newWeightedSigners The new weighted signers data
103      */
104     function _rotateSigners(WeightedSigners memory newSigners) internal {
```

Listing 8. Excerpt from BaseWeightedMultisig._validateSignatures

```
141     /*
142      * @notice This function takes messageHash and proof data and reverts if
    proof is invalid
143      * @param messageHash The hash of the message that was signed
```

```
144      * @param weighted The weighted signers data
145      * @param signatures The sorted signatures data
146      */
147     function _validateSignatures(
148         bytes32 messageHash,
149         WeightedSigners memory weightedSigners,
150         bytes[] memory signatures
151     ) internal pure {
```

Listing 9. Excerpt from BaseWeightedMultisig._baseWeightedStorage

```
206     /*
207      * @notice Gets the storage slot for the WeightedMultisigStorage struct
208      * @return the storage slot
209      */
210     function _baseWeightedStorage() private pure returns
    (WeightedMultisigStorage storage slot) {
```

Recommendation

Fix the parameter names in the functions mentioned above.

Go back to Findings Summary

W5: The reuse of the same nonce and signatures on another chain may confuse off-chain components

Impact:	Warning	Likelihood:	N/A
Target:	InterchainMultisig.sol	Type:	Logic

Description

Signed messages for InterchainMultisig, `executeCalls`, do not include `chainId`. The validation, if calls are supposed to be executed on a called chain, is performed for every call in a batch using the comparison of `Call.chainName` with the `chainNameHash` variable. This check prevents replay attacks when, in case two chains have the same set of signers, the signed message may be reused with the same nonce to perform the same action on another chain. However, the current implementation still emits the `BatchExecuted` event on every chain where the pair `(calls, nonce)` is reused. While this may not pose a security risk, this behavior may be confusing for off-chain components.

Recommendation

Consider emitting events only after the successful execution of at least one `Call`.

Go back to Findings Summary

W6: The documentation does not mention that signers and signatures should be sorted

Impact:	Warning	Likelihood:	N/A
Target:	BaseWeightedMultisig.sol	Type:	Documentation

Description

In `_validateSignatures` of BaseWeightedMultisig, signatures are validated against the list of signers without a complete loop over all signers. This technique saves gas. However, that means that this function will fail cases when pairs `(signer, signature)` are not sorted in ascending order by `signer`. Moreover, while `BaseWeightedMultisig::_rotateSigners` validates that the signer list is stored in the storage sorted, there are no checks in the contract that verify if the signatures are sorted accordingly, and neither it is mentioned in the documentation. That means that even for a valid set of signatures, the `InterchainMultisig::executeCalls` function will revert with the `MalformedSignatures()` error.

Example

Consider three signers with addresses `0x111…111`, `0x222…222`, and `0x333…333` stored in the same order as defined here. Consider they produce signatures `S1`, `S2`, and `S3`. If we call `executeCalls` with `signatures=[S1, S3, S2]` (notice the change in the order), the function will revert. Since there are no mentions of this behavior anywhere, this may lead to confusion.

Recommendation

Consider making changes to the documentation and mention the order of signatures.

Go back to Findings Summary

W7: The check of the minimal length of the proof is incorrect

Impact:	Warning	Likelihood:	N/A
Target:	BaseWeightedMultisig.sol	Type:	Logic

Description

In `validateProof` of BaseWeightedMultisig, the first operation checks if the length of the proof is not less than `32 * 4` bytes. However, `proof` is decoded as `address[], uint256[], uint256, bytes[]`, so the minimum length should rather be `32 * 7` bytes since even empty lists are encoded with two words: the position in memory or calldata and the length. Non-empty arrays additionally occupy further words to store actual data. That means the `proof` variable will occupy at least two words for each of the three included lists and one word for `uint256`. In total, it gives a minimum of seven words.

Recommendation

Change the check of the proof length to be compared with the correct value.

Go back to Findings Summary

W8: The `chainName` input argument to the constructor can be an empty string

Impact:	Warning	Likelihood:	N/A
Target:	InterchainMultisig.sol	Type:	Data Validation

Description

In InterchainMultisig, the constructor accepts the `chainName` string. However, there are no checks on the length of the input argument, and the contract may be mistakenly deployed with the incorrect chain name.

Recommendation

Consider adding a check for the input argument.

Go back to Findings Summary

W9: In `executeCalls`, nonces may be reused between different calls

Impact:	Warning	Likelihood:	N/A
Target:	InterchainMultisig.sol	Type:	Logic

Description

In `executeCalls` of InterchainMultisig, the `nonce` for a call batch may be reused for another `calls` argument. This is not a problem, since the purpose of the nonce is to prevent the caller from executing the same call batch more than one time. However, `nonce` usually stands for a monotonically increasing number, while the given implementation of nonce rather behaves like a salt, since the same nonce may be used for different call batches.

Recommendation

Consider changing the logic of nonce generation, or storing used nonces in the storage.

Go back to Findings Summary

W10: The `withdraw` function is marked `payable`

Impact:	Warning	Likelihood:	N/A
Target:	InterchainMultisig.sol	Type:	Logic

Description

In the `withdraw()` function of InterchainMultisig is marked `payable`. However, transferring Ether from the contract and receiving it are two opposite operations, so having a function that does both may be confusing. Consider removing the `payable` keyword.

Recommendation

Consider removing the `payable` keyword.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain, Axelar: CGP & GMP SDK refactor, 15.02.2024.

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 | blockchain security

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://twitter.com/AckeeBlockchain