Security Audit Report

Archway 2024-Q1

Authors: Darko Deuric, Marko Juric, Aleksandar Ignjatijevic

Last revised 8 February, 2024

Table of Contents

Title	Archway 2024-Q1
Client	Archway
Team	Darko Deuric Marko Juric Aleksandar Ignjatijevic
Start	08 Jan 2024
End	02 Feb 2024
Domains	Cosmos SDK CosmWasm
Methods	Code review
Languages	Go Rust
Report	

Audit Overview

The Project

In January 2024, the Archway development team engaged Informal Systems to conduct a comprehensive security audit specifically targeting the callback module, the cwfees module, and the refined withdrawal user experience - part of the rewards module.

Scope of this audit

The audit was conducted by the following individuals:

- Darko Deuric
- Marko Juric
- Aleksandar Ignjatijevic

It comprehensively covered three primary components, each with specific areas of focus:

1. **Callback module**:
    - The primary concern was to ensure the system's resilience against potential threats posed by malicious smart contracts. Given that the execution of smart contract code (callback) occurs from the end blocker of the callback module, it was imperative to verify that such executions couldn't crash the blockchain.
    - Another critical aspect was the examination of gas fee calculations within the callback module. The audit aimed to ensure that these calculations were accurate and could not be exploited.
2. **Cwfees module**:
    - The audit focused on the processes of registering and unregistering smart contracts as fee granters. This involved scrutinizing the integrity and security of these processes to prevent any potential vulnerabilities.
    - A significant part of the evaluation was directed towards the custom ante handle mechanism implemented in the rewards module and its interaction with the cwfees module. Unlike the callback module, calls to the contract in this context are made from the msg server context, presenting a different threat model.
3. **Refined withdrawal user experience**:
    - The final aspect of the audit was the automation of the withdrawal rewards process for dApp developers. This included a thorough assessment of the new functionality and its integration into the existing system.
    - Special attention was given to ensure that the introduction of an additional flag in the contract Metadata did not violate existing system protocols or introduce new vulnerabilities.

Conducted work

The audit team conducted an in-depth review of the code for the callback module, cwfees module, and the refined withdrawal user experience part. This review aimed to identify any potential security vulnerabilities, inefficiencies, or deviations from best practices.

To simulate real-world conditions and assess the system's behavior under various scenarios, the team set up a local environment to run the blockchain. Smart contracts were added to this environment, and a range of actions, particularly malicious ones, were executed to observe their effects on the chain. This helped in evaluating the system's resilience and response to potential threats.

Examples of these malicious actions included executing an indefinite loop within a contract in order to reach gas limit, intentionally triggering a panic in a contract, and deliberately omitting the job ID previously registered in a callback. These actions were designed to stress-test the blockchain and gauge its ability to handle and recover from unexpected behaviors, providing valuable insights into the system's robustness and security measures.

The team delved into the unit tests accompanying the system components. This provided valuable insights into the expected functionality of the components and helped in verifying whether they operated as intended.

Throughout the audit process, there were regular synchronization meetings with the Archway team. These meetings were crucial for aligning objectives, sharing findings, and discussing potential improvements.

Conclusions

The audit identified a total of 10 findings, categorized by their severity. Among these, one was classified as Critical. This finding posed a significant threat to the chain's liveness, primarily due to the potential for a panic situation within the end blocker.

The remaining findings were categorized as Medium, Low, or Informational in terms of their severity. Notably, one of the Informational findings highlighted various less significant issues within the code. These were not critical but still important for overall code quality and functionality.

The overall conclusion is that the code for the audited modules is of high quality. It is well-structured and adequately covered with unit tests, ensuring reliability and maintainability.

Each implemented feature in the system is accompanied by an appropriate ADR, providing clarity and rationale behind design choices.

The documentation for the callbacks feature, in particular, was noted for its thoroughness, including a detailed proposal document and specification. This level of documentation exemplifies best practices in software development and aids in future maintenance and scalability.

Despite the critical finding, the Archway system demonstrates a strong foundation in terms of code quality, security considerations, and thorough documentation.

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Cosmos SDK, Rust
- **Artifacts:**
    - callback module, commit
    - cwfees module, commit
    - Refined withdrawal user experience, PR

Engagement Summary

- **Dates**: 08.01.2024 to 02.02.2024
- **Method**: Manual code review, protocol analysis
- **Employees Engaged**: 3

Severity Summary

Finding Severity	#
Critical	1
High	0
Medium	2
Low	4
Informational	3
Total	**10**

System Overview

The following system overview focuses on three Archway components:

- the callback module (ADR-009),
- the cwfees module (ADR-010), and
- the refined withdrawal user experience (ADR-008).

Each part plays a unique role in enhancing the functionality and security of the Archway system.

Callback module

It allows contracts to register or cancel a callback. Contracts provide essential details like ContractAddress, JobId, CallbackHeight, and Fees for requesting a callback. The CallbackHeight is particularly important as it schedules the callback for execution at a specific block height during the endBlocker phase.

During the callback, a smart contract is triggered with a Sudo message. Based on the JobId, the smart contract executes its logic. To prevent malicious activities such as infinite loops, the contract executes in a branched context with a limited gas meter. To be accepted by the chain, the requested callback must have fees equal to or greater than the sum of futureReservationFee, blockReservationFee, and transactionFee (explained in ADR-009).

Cwfees module

Implements a fee-granting mechanism for smart contracts. Smart contracts become fee granters by registering themselves using only their address. Through the rewards' module DeductFeeDecorator, the fee granter can be either the address set in the feegrant module or the granting contract itself. The contract has the option to accept or reject fee grant requests. As adding contracts is permissionless, there is a mechanism which limits the gas that can be spent within the contract, preventing node spamming.

Refined withdrawal user experience

The goal is to simplify the process of withdrawing rewards for dapp developers. It adds the capability for dApp developers to directly withdraw their rewards to a specified wallet address. This approach bypasses the more cumbersome process of creating reward records and using a lazy withdrawal method.

Threat model

Threat: Callback module fees accounting flaw

Maintaining the integrity of the fee accounting process in the callback module is vital for accurate fee distribution.

Assets

- `futureReservationFee` : calculated as `FutureReservationFeeMultiplier * (height_callback - height_current)` .
- `blockReservationFee` : determined by `BlockReservationFeeMultiplier * nmbTotalCallbacksPerHeight` .
- `transactionFee` : defined as `gasLimit * gasPrice` .
- `surplusFees` : calculated as `requestCallbackFees - (futureReservationFee + blockReservationFee + transactionFee)` .

Key definitions

- `gasLimit` : Either the `CallbackGasLimit` (a module parameter set when the callback is created) or `gasUsed` (the amount of gas consumed during callback execution).
- `gasPrice` : The current gas price obtained from the rewards module.
- `FutureReservationFeeMultiplier` and `BlockReservationFeeMultiplier` : Module parameters.
- `height_callback` and `height_current` : Block heights corresponding to the callback height and the current block height, respectively.
- `nmbTotalCallbacksPerHeight` : The total number of registered callbacks at a given block height.
- `requestCallbackFees` : The fee amount sent by the callback creator to the callback module.

Fee distribution flows

1. CancelCallback invocation:
    - The `request.Sender` is refunded `transactionFee + surplusFees` .
    - Remaining funds (`feeCollectorAmount = futureReservationFee + blockReservationFee`) go to the FeeCollector account.
2. Callback Execution (callbackExec):
    - `transactionFeeConsumed` is calculated based on `gasUsed` and the execution-time gas price (`gasPrice2`), which may differ from the registration-time gas price (`gasPrice1`).
    - If `transactionFeeConsumed < transactionFee` , the difference (`refundAmount`) is sent to the callback creator.
    - Remaining funds are sent to the FeeCollector. The `feeCollectorAmount` must equal `futureReservationFee + blockReservationFee + surplusFees + transactionFeeConsumed` .

Invariant

For every callback, the following must hold true:

`requestCallbackFees = refundAmount + feeCollectorAmount`

Example

- **Case 1: CancelCallback**
  - Given: `futureReservationFee=50` , `blockReservationFee=150` , `gasPrice1=10` , `CallbackGasLimit=80` , `requestCallbackFees=1200` .
  - Calculated: `surplusFees=200` , `refundAmount=1000` , `feeCollectorAmount=200` .
  - Invariant check: `1200 = 1000 + 200` .
- **Case 2: Successful callbackExec**
  - Given: `gasPrice2=9` , `gasUsed=70` .
  - Calculated: `transactionFeeConsumed=630` , `refundAmount=170` , `feeCollectorAmount=1030` .
  - Invariant Check: `1200 = 170 + 1030` .
- **Case 3: Failed callbackExec with unchanged** `CallbackGasLimit`
  - Modified: `refundAmount=80` , `feeCollectorAmount=1120` .
  - Invariant check: `1200 = 80 + 1120` .
- **Case 4: Failed callback execution with increased** `CallbackGasLimit`
  - Original `CallbackGasLimit` = 80.
  - Modified `CallbackGasLimit` for failed callback = 90.
  - Unchanged `gasPrice2` = 10.
  - `futureReservationFee` = 50, `blockReservationFee` = 150, `surplusFees` = 200.
  - `requestCallbackFees` = 1200 (initial amount sent to the callback module).

Transaction fee calculation for modified callback

`transactionFee` = 80 * 10 = 800 (calculated at callback registration with original `CallbackGasLimit`)

`transactionFeeConsumed` = 90 * 10 = 900 (calculated at execution with new `CallbackGasLimit`)

Fee collector amount

`feeCollectorAmount` = 50 (`futureReservationFee`) + 150 (`blockReservationFee`) + 200 (`surplusFees`) + 900 (`transactionFeeConsumed`) = 1300.

Refund amount

Since `transactionFeeConsumed` > `transactionFee` , there is no refund to the callback creator.

Result

`feeCollectorAmount` = 1300, which is greater than the `requestCallbackFees` of 1200. This situation creates a deficit in the system and violates the invariant. This issue is briefly analyzed here.

Threat : Panic risk in callback module's endBlocker zone

The following sub-threats all pertain to the potential risk of a system-wide panic within the callback module's endBlocker zone. This is a critical area of concern because a panic in the endBlocker could halt the entire blockchain.

Threat : Panic risk in `RefundFromCallbackModule` function

The threat concerns the `RefundFromCallbackModule` function, which is called within the `endBlocker` of a callback module. This function performs a coin transfer from a callback module account to the callback creator (recipient). The critical risk is a potential chain halt due to a panic if the function encounters an error.

Assets

- **Module account**: The source of funds for refunds.
- **Recipient account**: The destination for the refund transaction.

Attack vector

If the recipient address is black-listed, the `SendCoinsFromModuleToAccount` function will return an error.

Potential impacts

A panic within the `endBlocker` execution could halt the entire blockchain.

An issue is reported here.

Threat : Gas consumption panic in `ExecuteWithGasLimit`

Overview

This threat analysis is centered on the configuration where the end blocker operates with an infinite gas meter. This means that there is no explicit upper limit set on the gas consumption per block. The primary concern in this scenario is the theoretical risk of an integer overflow within this line: `ctx.GasMeter().ConsumeGas(gasUsed, "branch")`.

However, it's important to note that the likelihood of such an event is considerably low.

Attack vector

The sole immediate threat in this setup is the potential for an integer overflow in the gas metering system. This could theoretically occur if the cumulative gas consumption within a block reaches a value exceeding the maximum capacity of the integer data type used for tracking gas. However, this risk is substantially mitigated by operational constraints within the system:

- The callback module inherently limits the number of callbacks that can be executed in a single block. This limitation significantly reduces the risk of reaching a point where an integer overflow could occur in gas consumption.
- Each callback within the system is subject to its own gas limit. This further limits the total gas usage within a block and diminishes the probability of encountering an integer overflow.
- In the current setup, the end blocker operates without being subject to gas limits, which is consistent with the infinite gas meter approach.

Potential impact

Given the system's current design, which includes a limit on callbacks per block and specific gas limits for each callback, the probability of reaching an integer overflow is extremely low.

Threat : Unauthorized access

The following sub-threats all pertain to the potential risk of executing unauthorized actions within callback module which could result with various unwanted consequences.

Threat : Unauthorized callback request creation in `RequestCallback` function

In order to register callback `Sender` needs to specify a `ContractAddress` which will receive a callback at specified block height.

Malicious actors could try to register callback to someone's contract address.

In `RequestCallback` function, there is a `SaveCallback` function which calls `isAuthorizedToModify`, a function which checks what authorizations a `Sender` has in terms of the specified contract.

In order to have authorization for the action, `request.Sender` needs to be either:

- `contractAddress` itself
- `Admin` of the contract (stored inside contract info of contract)
- `OwnerAddress` of the contract, registered in rewards module (`rewards.GetContractMetadata`)

Threat : Unauthorized callback request cancelling in `CancelCallback` function

In order to cancel callback `Sender` needs to specify a `ContractAddress` whose specified callback will be cancelled and as well deleted from callback request storage.

Malicious actors could try to cancel callback to someone's contract address.

In `CancelCallback` function, there is a `DeleteCallback` function which calls `isAuthorizedToModify`, elaborated in previous threat.

Threat : Unauthorized edit of callback module params in `UpdateParams` function

In order to update module `Params` provided in `MsgUpdateParams`, method also checks the other field of `MsgUpdateParams`, which is `Authority`.

This method cannot be called externally, which means there is no way a malicious actor can invoke this action. Currently, it states that unless overwritten, the address which is authorized for this action is x/gov module address.

Threat : Authorization handling in wasm engine with focus on MsgUnregisterAsGranter

The threat model pertains to the authorization mechanism in a wasm engine, particularly focusing on handling the `MsgUnregisterAsGranter` message. The core concern is the potential for a malicious contract to craft and send a `MsgUnregisterAsGranter` message with the `GrantingContract` field set to another contract's address, attempting to unregister it as a granter.

Attack scenario:

- A malicious contract constructs a `MsgUnregisterAsGranter` message.
- The `GrantingContract` field in this message is set to the address of a victim contract, attempting to unregister it as a granter.

Vulnerability analysis:

The perceived vulnerability lies in the ability to manipulate the `MsgUnregisterAsGranter` message to affect other contracts. However, the wasm engine's authorization mechanism mitigates this risk.

- The wasm engine employs a safeguard in the `handleSdkMessage` function:

```
func (h SDKMessageHandler) handleSdkMessage(ctx sdk.Context, contractAddr
sdk.Address, msg sdk.Msg) (*sdk.Result, error) {
    if err := msg.ValidateBasic(); err != nil {
        return nil, err
    }
    for _, acct := range msg.GetSigners() {
        if !acct.Equals(contractAddr) {
            return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, "contract
doesn't have permission")
        }
    }
    ...
}
```

- The `MsgUnregisterAsGranter.GetSigners` method is crucial in this context:

```
func (m *MsgUnregisterAsGranter) GetSigners() []sdk.AccAddress {
    return []sdk.AccAddress{sdk.MustAccAddressFromBech32(m.GrantingContract)}
}
```

- This mechanism ensures that only the contract listed as the `GrantingContract` can initiate the `MsgUnregisterAsGranter`. In the event of a discrepancy between `contractAddr` (the actual sender) and `m.GrantingContract` (the claimed sender), the message is rejected due to unauthorized access.

Conslusion:

The existing authorization checks in the wasm engine ensure that only authorized contracts can execute actions that impact their state, thus maintaining integrity and preventing unauthorized interference.

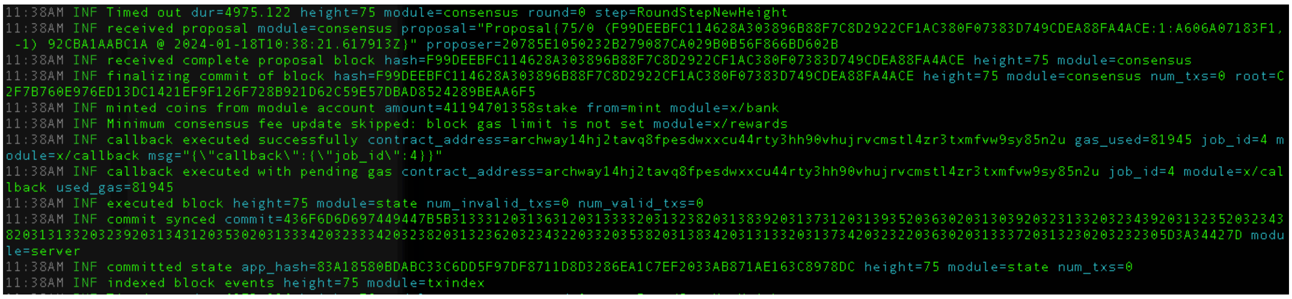
Threat : Malicious activities in smart contract could halt the chain

Is it possible for *panic* in the smart contract to halt the chain?

It appears it is impossible to halt the chain throwing the *panic* within the smart contract. When *panic* in the smart contract is thrown, chain just registers the error (shown on the screenshot below).

Is it possible for different invalid `job_id` to crash callback execution?

Invalid `job_id` still executes callback (shown on the screenshot below).



Malicious infinity loop in callback smart contract?

Malicious infinity loop won't halt the chain. It is handled like an error and thrown `OutOfGas` (shown on the image below).

Threat : DoS attacks

Threat : Overloading the system with excessive callback requests, potentially leading to DoS.

How cheap is for an attacker to populate certain block heights with `MaxBlockReservationLimit` of dummy callbacks?

More on this threat can be found here.

Threat: Spamming the chain with `RequestGrant` calls and large gas limit

A malicious actor deploys a smart contract designed to consume excessive gas. The attacker then initiates a transaction with a very high gas limit (e.g., 100 million gas), **setting this malicious contract as the granter**. The contract is programmed to enter an endless loop, consuming all the provided gas.

Execution context**:**

```
gasLimitToUse := min(sdkCtx.GasMeter().GasRemaining(), RequestGrantGasLimit)
_, err = pkg.ExecuteWithGasLimit(sdkCtx, gasLimitToUse, func(ctx sdk.Context) error {
    _, err = k.wasmdKeeper.Sudo(sdk.UnwrapSDKContext(ctx), grantingContract,
msgBytes)
    return err
})
```

The crucial part is setting the `RequestGrantGasLimit` to a reasonable value to prevent exploitation, because `GasRemaining()` could be quite large.

If the gas limit for executing the smart contract (`RequestGrantGasLimit`) is set too high, it could allow the malicious contract to consume significant chain resources before hitting the limit. This would enable the attacker to spam the chain with transactions that consume excessive computational resources without paying appropriate fees, as the transaction would eventually be reverted because ante handlers would fully revert.

The `RequestGrantGasLimit` must be carefully calibrated to be high enough to allow legitimate transactions but low enough to prevent exploitation through malicious contracts. This limit acts as a crucial control point in the gas consumption strategy.

Current value of `100_000` seems to be reasonable.

You can also consider dynamic adjustments of gas limits based on network conditions, historical data, and observed behaviors of contracts.

Findings

Title	Type	Severity	Status
Callback module fee transfer mechanism can lead to potential chain halt	IMPLEMENTATION	4 CRITICAL	RESOLVED
Case sensitivity issue in authorization check of isAuthorizedToModify function	IMPLEMENTATION	2 MEDIUM	RESOLVED
Potential DoS attack via callback registrations	IMPLEMENTATION	2 MEDIUM	ACKNOWLEDGED
Unnecessary computation when requesting callback in current block	IMPLEMENTATION	1 LOW	RESOLVED
Panicking on inadequate fee denomination provided by callback creator	IMPLEMENTATION	1 LOW	RESOLVED
Optimizing callback existence verification	IMPLEMENTATION	1 LOW	RESOLVED
Optimization of granting contract registration method	IMPLEMENTATION	1 LOW	RESOLVED
Potential panic due to blocked address in RefundFromCallbackModule function	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Various minor issues in the Archway modules	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
A malicious granter contract could intentionally try to spam with wasting the free gas	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED

Callback module fee transfer mechanism can lead to potential chain halt

Title	Callback module fee transfer mechanism can lead to potential chain halt
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/abci.go

Description

We identified a critical issue within the `callbackExec()` function, which is part of the callback module. This function, particularly when handling fee transfers to the feeCollector module, exhibits a design flaw that could lead to chain halt.

Problem Scenarios

One of `callbackExec()` function's tasks is to trigger sending of a calculated fee (`feeCollectorAmount`) from the callback module to the feeCollector module using `SendToFeeCollector` , which in turn calls `SendCoinsFromModuleToModule` .

If the callback module's balance is insufficient to cover the `feeCollectorAmount` , an error is returned. This is especially likely when the `CallbackGasLimit` parameter is modified upwards, increasing the `gasUsed` in the case of failed callback execution. For instance, a change in `CallbackGasLimit` from 80 to 90, coupled with other fixed fees:

`BlockReservationFees` = 50, `FutureReservationFees` = 150, `SurplusFees` = 200,

and `MsgRequestCallback.Fees.Amount` = 1200, `transactionFee` = 10 (`gasPrice`) * 80 (`CallbackGasLimit`) = 800, resulted in a required `feeCollectorAmount` of 50 + 150 + 200 + 10 * 90 = 1300 , exceeding the callback module's balance equal to `MsgRequestCallback.Fees.Amount` of 1200.

Consequently, the error returned by `SendToFeeCollector` triggers a panic within the endBlocker. This reaction is hazardous as it can halt the entire chain.

Recommendation

It is crucial to be mindful of the existing callbacks registered on the chain when considering updates to the `CallbackGasLimit` parameter. The updates to this parameter can significantly influence the handling of fees, particularly in the context of callbacks scheduled for execution in subsequent blocks following the update.

To avoid potential system disruptions, it's better not to change the `CallbackGasLimit` if there are callbacks waiting to be processed.

Case sensitivity issue in authorization check of isAuthorizedToModify function

Title	Case sensitivity issue in authorization check of isAuthorizedToModify function
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/keeper/callback.go

Description

The `isAuthorizedToModify` function is designed to check if a sender is authorized to modify callbacks of a contract. The function determines authorization based on three conditions:

1. If the sender is the contract itself.
2. If the sender is the admin of the contract.
3. If the sender is the owner of the contract, as specified in the contract's metadata.

Problem Scenarios

The core issue arises due to the handling of Cosmos SDK addresses in different case formats (lowercase and uppercase). Since the function compares the sender's address with the contract's address, admin's address, and owner's address as strings, it fails to recognize the equivalence of the same address in different case formats.

For example, `COSMOS14HJ2TAVQ8FPESDWXXCU44RTY3HH90VHUJRVCMSTL4ZR3TXMFVW9S4HMALR` and `cosmos14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9s4hmalr` represent the same address (`AccAddressFromBech32` returns `ADE4A5F5803A439835C636395A8D648DEE57B2FC90D98DC17FA887159B69638B` for both representations), but `isAuthorizedToModify` will treat these as different, leading to incorrect authorization checks.

Let's consider a specific case where a callback is created with the following properties:

```
callback := callbackTypes.Callback{
        ContractAddress:
"cosmos14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9s4hmalr",
        JobId:            123,
        CallbackHeight:   456,
        ReservedBy:
"COSMOS14HJ2TAVQ8FPESDWXXCU44RTY3HH90VHUJRVCMSTL4ZR3TXMFVW9S4HMALR",
        FeeSplit:         &callbackTypes.CallbackFeesFeeSplit{},
    }
```

In this scenario, the `ContractAddress` is provided in lowercase, while the `ReservedBy` address is in uppercase. Even though both addresses represent the same entity, their different case formats lead to a significant problem in the authorization check within `isAuthorizedToModify`.

Consequently, the function incorrectly concludes that the sender (i.e., the entity represented by `ReservedBy`) is not authorized to create the callback.

The issue similarly affects the `CancelCallback` functionality. The authorization check may incorrectly fail if the sender's address case format does not match that of the contract's admin, contract's address or owner address.

Finally, checks if the sender is the admin or the owner of the contract for authorization are also susceptible to the same case sensitivity problem.

Recommendation

Implement a case-insensitive comparison for addresses. This can be achieved by converting both the sender's address and the contract-related addresses (contract itself, admin, owner) to a common case (either lower or upper) before performing the comparison.

Potential DoS attack via callback registrations

Title	Potential DoS attack via callback registrations
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- x/callback/keeper/callback.go

Description

There is a potential for a DoS attack by exploiting the callback registration mechanism. Attackers could create numerous callbacks to fill up blocks, impacting the other users callback 's registration functionality.

Problem Scenarios

Malicious actors create a large number of dummy smart contracts, registering them as callbacks to occupy upcoming block space. This is feasible due to the current `MaxBlockReservationLimit` set to 3, allowing up to three callbacks per block:

```
callbacksForBlock, err := k.GetCallbacksByHeight(ctx, callback.GetCallbackHeight())
if err != nil {
    return err
}
if len(callbacksForBlock) >= int(params.MaxBlockReservationLimit) {
    return types.ErrBlockFilled
}
```

Vulnerability:

1. Attackers could potentially receive most of the `callback.FeeSplit.TransactionFees` back, especially if the dummy contracts execute minimal actions.

2. The `FutureReservationFees` become negligible for callbacks created for the near future, given the `FutureReservationFeeMultiplier` is currently set to 1.
3. The `BlockReservationFees` can be relatively low (up to a maximum of 3) when an attacker fills an entire block, as the `BlockReservationFeeMultiplier` is also set to 1.

Recommendation

1. Appropriately setting the `FutureReservationFeeMultiplier` and `BlockReservationFeeMultiplier` is crucial. These parameters should be balanced to encourage honest use while deterring malicious actors from spamming the network.
2. Implementing a system to limit the number of callbacks an individual address can reserve within a specific block or a set of blocks would significantly enhance security. This limit would prevent a single attacker from monopolizing block space.
3. Consider implementing a dynamic system where `MaxBlockReservationLimit` and associated fees can adjust based on network conditions and historical data of callback registrations. This approach could prevent potential exploitation of static limits.

Unnecessary computation when requesting callback in current block

Title	Unnecessary computation when requesting callback in current block
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/keeper/callback.go
- x/callback/keeper/fees.go
- x/callback/keeper/msg_server.go

Description

In the event of requesting the callback within the current block, there will be an unnecessary usage of time and resources to compute `futureReservationFee`, `blockReservationFee`, and `transactionFee` within `EstimateCallbackFees` function, due to different relational operands used in `if` statements within `EstimateCallbackFees` and `SaveCallback` functions.

Message server firstly calls function `EstimateCallbackFees`, that will check if `request.CallbackHeight < ctx.BlockHeight`. If requested callback is at the same height as blockHeight it will pass through this `if` statement.

Then it calls function `SaveCallback`, that will check if `callback.GetCallbackHeight() <= ctx.BlockHeight()`. If requested callback is at the same height as blockHeight it will report an error that requested callback is not in the future.

Problem Scenarios

When requesting a callback within the current block, the `EstimateCallbackFees` function will execute without error and use time and resources to compute `futureReservationFee`, `blockReservationFee`, and `transactionFee`.

The problematic part of code from `EstimateCallbackFees` function: code snippet from fees.go file.

After finishing the execution of `EstimateCallbackFees` function, `RequestCallback` function will create `callback` and save it using `SaveCallback`.

However, an issue arises in the `SaveCallback` function. It contains an `if` statement that checks if the requested callback is less than or equal to the current block height. This check results in the `ErrCallbackHeightNotInFuture` error, and consequently, the callback is not saved.

The problematic part of code from `SaveCallback` function: code snippet from callback.go file

Recommendation

Recommendation is to change relation operation: less to less than or equal within `EstimateCallbackFees` function. Recommended solution:

```
if blockHeight <= ctx.BlockHeight() {
    return sdk.Coin{}, sdk.Coin{}, sdk.Coin{}, status.Errorf(codes.InvalidArgument,
"block height %d is not in the future", blockHeight)
}
```

Consequently, there will be the need to change error message within that `if` statement to something that would more accurately represent the error.

Current error message returned is `"block height %d is in the past"`. This should be changed to `"block height %d is not in the future"`.

Panicking on inadequate fee denomination provided by callback creator

Title	Panicking on inadequate fee denomination provided by callback creator
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/types/msg.go
- x/callback/keeper/msg_server.go

Description

The `RequestCallback()` function compares the fees from the request (`request.GetFees()`) with the expected fees (`expectedFees`) without verifying that both are in the same denomination. `expectedFees` are calculated as a sum of various fees, all in `DefaultBondDenom` denomination.

However, `request.GetFees()` may contain fees in a different denomination, leading to potential issues in the `.IsLT()` function. It is important to note that while this can cause a panic, its impact is limited as it occurs within the `msgServer` and not in the `endBlocker` . Thus, it cannot halt the entire blockchain.

Problem Scenarios

The absence of a denomination check in the `ValidateBasic()` function of `MsgRequestCallback` is the primary concern. This function currently does not ensure that `MsgRequestCallback.Fees.Denom` is equivalent to `DefaultBondDenom` .

As a result, a `MsgRequestCallback` transaction with an invalid fee denomination can bypass initial checks and reach the `RequestCallback()` function, where the mismatched denominations can cause a panic in the `.IsLT()` comparison.

It's important to note that even if the transaction fails, the gas consumed up to the moment of the error is still taken into account.

Recommendation

To address this issue we recommend to update the `ValidateBasic()` function within the `MsgRequestCallback` structure to include a check that ensures the fee denomination matches `DefaultBondDenom`. This should prevent transactions with incorrect fee denominations from reaching the `RequestCallback()` function, as they would be rejected during the `checkTx` phase.

Optimizing callback existence verification

Title	Optimizing callback existence verification
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/keeper/callback.go

Description

The `CancelCallback` and `DeleteCallback` functions involve operations for handling callback deletion. The `CancelCallback` function serves as the entry point for canceling a callback. It initially **verifies the existence of a callback** using the `GetCallback` method. Once confirmed, it proceeds to call `DeleteCallback` to remove the callback.

Problem Scenarios

Within the `DeleteCallback` method, there is a redundant check for the callback's existence using `ExistsCallback` function. This check is performed despite the previous verification in `CancelCallback`. The `ExistsCallback` function internally uses `gs.Store.Has` which is a gas-intensive operation, as indicated by the `KVGasConfig` function where `HasCost` is set to 1000.

```
func (gs *Store) Has(key []byte) bool {
    gs.gasMeter.ConsumeGas(gs.gasConfig.HasCost, types.GasHasDesc)
    return gs.parent.Has(key)
}

func KVGasConfig() GasConfig {
    return GasConfig{
        HasCost:          1000,
```

```
        ...
    }
}
```

This means each existence check incurs a significant computational cost.

Recommendation

To optimize the process and reduce gas consumption, it is advisable to remove the existence check (`ExistsCallback`) from the `DeleteCallback` method. Since `CancelCallback` already performs this verification, the additional check in `DeleteCallback` is superfluous.

Optimization of granting contract registration method

Title	Optimization of granting contract registration method
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Status	RESOLVED
Issue	

Involved artifacts

- x/cwfees/keeper.go

Description

This finding evaluates the `RegisterAsGranter` and `UnregisterAsGranter` functions in x/cwfees module. Both functions are crucial for managing granting contracts, but they employ different approaches to handle registration and unregistration.

Problem Scenarios

RegisterAsGranter:

Currently, `RegisterAsGranter` checks whether a contract is already registered as a granter before proceeding with registration. This adds an additional step and associated computational/gas cost to the function.

Code snippet:

```
func (k Keeper) RegisterAsGranter(ctx context.Context, granter sdk.AccAddress) error
{
    // we want to assess that the granter is a CW contract.
    if !k.wasmdKeeper.HasContractInfo(sdk.UnwrapSDKContext(ctx), granter) {
        return types.ErrNotAContract
    }
    isGranter, err := k.IsGrantingContract(ctx, granter)
    if err != nil {
        return err
    }
    if isGranter {
```

```
        return types.ErrAlreadyGranter.Wrapf("address %s", granter.String())
    }
    return k.GrantingContracts.Set(ctx, granter)
}
```

The `Set` method in `GrantingContracts` is idempotent and will not result in a different state if called multiple times with the same contract address.

UnregisterAsGranter:

`UnregisterAsGranter` checks if the contract is registered as a granter before attempting to remove it. This check is crucial because the `Remove` method doesn't return an error if the key doesn't exist.

Code snippet:

```
func (k Keeper) UnregisterAsGranter(ctx context.Context, granter sdk.AccAddress)
error {
    isGranter, err := k.IsGrantingContract(ctx, granter)
    if err != nil {
        return err
    }
    if !isGranter {
        return types.ErrNotAGranter.Wrapf("address %s", granter.String())
    }
    return k.GrantingContracts.Remove(ctx, granter)
}
```

Recommendation

1. For `RegisterAsGranter` : Adopt an optimistic approach by removing the initial check for the contract's existence in the granters list. This approach relies on the idempotent nature of the `Set` method, reducing unnecessary steps and saving on computational/gas costs. Revised code:

```
func (k Keeper) RegisterAsGranter(ctx context.Context, granter sdk.AccAddress)
error {
    // we want to assess that the granter is a CW contract.
    if !k.wasmdKeeper.HasContractInfo(sdk.UnwrapSDKContext(ctx), granter) {
        return types.ErrNotAContract
    }
    return k.GrantingContracts.Set(ctx, granter)
}
```

This change assumes that registering a contract multiple times has no adverse effects on the system and that gas cost optimization is a priority.

2. For `UnregisterAsGranter` : Retain the current implementation. The pre-check to verify if the `granter` is registered is essential for maintaining the integrity of the system and providing accurate feedback. Since the `Remove` method acts as a no-op for non-existent keys without returning an error, removing this check would allow silent failures, leading to potential confusion and errors in the system's state management.

Potential panic due to blocked address in RefundFromCallbackModule function

Title	Potential panic due to blocked address in RefundFromCallbackModule function
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	3 HIGH
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- x/callback/keeper/keeper.go

Description

The `RefundFromCallbackModule` function is designed to refund coins from a module account (`x/callback`) to a specified recipient. However, there is a potential issue when the recipient's address is on a blocked address list.

In such cases, the `SendCoinsFromModuleToAccount` function, which performs the actual transfer, returns an error if the recipient address is blocked. Consequently, this leads to a panic in the calling function due to the error not being handled but instead causing a panic.

Problem Scenarios

If `RefundFromCallbackModule` attempts to send coins to a blocked address, it results in an error. Since this error leads to a panic inside endBlocker's domain, it could halt the chain .

However, the usage and management of blocked addresses within the system have been subject to changes and discussions. This includes debates over restricting only module account addresses, dynamically managing the blacklist, and even initiatives to remove the blocked address list entirely.

Perhaps it's worth noting that the Osmosis team employs blocked addresses in their system.

Recommendation

Revise `RefundFromCallbackModule` to handle errors more effectively, particularly in scenarios involving refunds to blocked addresses. Ensure that such errors do not lead to a panic, especially in critical parts of the system like `endBlocker`.

Various minor issues in the Archway modules

Title	Various minor issues in the Archway modules
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- x/callback/keeper/callback.go
- x/callback/keeper/grpc_query.go
- x/callback/keeper/msg_server.go
- x/callback/types/genesis.go
- x/callback/abci.go
- x/cwfees/keeper.go
- x/cwfees/types/msgs.go
- x/rewards/keeper/distribution.go

Description

This is a list of various minor issues that have been noticed in the Archway modules code, but they do not pose a security threat.

- Unused parameter in function:
  - Function isAuthorizedToModify() has unused parameter `height` in the definition.
- Initialized variables used only once:
  - Function EstimateCallbackFees(), function Params() and function Callbacks() all initialize local variable `ctx` by unwrapping SDKContext and then use that variable in functions. It would be better to just call `sdk.UnwrapSDKContext()` function when needed, because that variable is only used once per function, in functions: `keeper.EstimateCallbackFees()` , `keeper.GetParams()` and `GetCallbacksByHeight()` respectively.
  - The same could be said for functions UpdateParams() and RequestGrant().
  - Function DefaultGenesis() initializes local variable `defaultParams` and then returns its value.

  - Function EndBlocker() initializes local varaible `currentHeight` and then uses it only when calling `IterateCallbacksByHeight()`.
- Inconsistency in naming message files. Within callback and rewards module, message files are named *msg.go*, but in cwfees module it is named *msgs.go*. Keep it consistent.
- Unnecessary calling of getter for the same value `CallbackHeight` in function SaveCallback(). `Callback` struct passed to `SaveCallback()` function can not be `nil` because it is created in `RequestCallback()` function. Here is the link to the code snippet. `CallbackHeight` can be acquired just by accessing the `Callback` struct field.
- Inconsistency in implementing `GetSigners()` functions within callback and cwfees modules. Implementation of `GetSigners()` function in callback module is just rewriting implementation of `MustAccAddressFromBech32()` from CosmosSDK (code snippet) and just defining custom error message. That implementation makes code less clear.
- Unnecessary usage of getter functions within RequestCallback() function. Even though `request` can be `nil` it has already been checked at the beginning of the function. Fields could be accessed without getter functions.
- The current implementation sets the `RequestGrantGasLimit` parameter in the cwfees module as a hardcoded constant with the value `100,000`. It is advisable to consider making this value a module parameter, as there might arise a need to adjust it. While this approach offers greater flexibility, it comes with the trade-off of an additional state read each time the cwfees module is utilized.
- `createRewardsRecords` function has undergone an update to enable immediate withdrawals to the specified address. It's important to note that this enhancement allows for the possibility of bypassing the creation of rewards records, in cases where contracts include a flag indicating automatic withdrawal preference. Given this expanded functionality, the current function name might not accurately convey its complete behavior and is advised to be reconsidered for enhanced clarity.

A malicious granter contract could intentionally try to spam with wasting the free gas

Title	A malicious granter contract could intentionally try to spam with wasting the free gas
Project	Archway 2024-Q1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	1 LOW
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- x/cwfees/keeper.go

Description

To facilitate fee grants to the sender, `AnteHandle` undergoes a check to ensure that the specified granting contract is willing to cover the associated fees. To mitigate potential abuse, a constant `RequestGrantGasLimit` is introduced and set to 100,000 (current value), preventing malicious contracts from consuming an indefinite amount of gas. The contract is allocated a fixed amount of gas to process the request and must respond with either a "positive" or "negative" answer.

Problem Scenarios

It's crucial to address potential malicious behavior by contracts on the granting list, such as intentionally reaching the gas limit to cause transaction failure or responding affirmatively without possessing the necessary funds to cover fees. In both scenarios, the transaction will not be included in the mempool, as it fails within the `AnteHandler`.

Additionally, there is a risk that a malicious sender, in collaboration with a malevolent contract, may submit multiple requests leading to transaction failures. Although unsuccessful, this could compel the protocol to execute unnecessary computations, wasting "free gas."

Recommendation

Upon consultation with the Archway team, it was acknowledged that a challenge in trustless fee granting is allowing the contract to consume 100,000 units of free computation.

To address concerns about contracts in the granting list that may not genuinely provide the intended functionality, there could be a monitoring which would detect such contracts and, if necessary, remove them to prevent them from serving as granters. This proactive approach ensures the integrity of the fee-granting process.

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

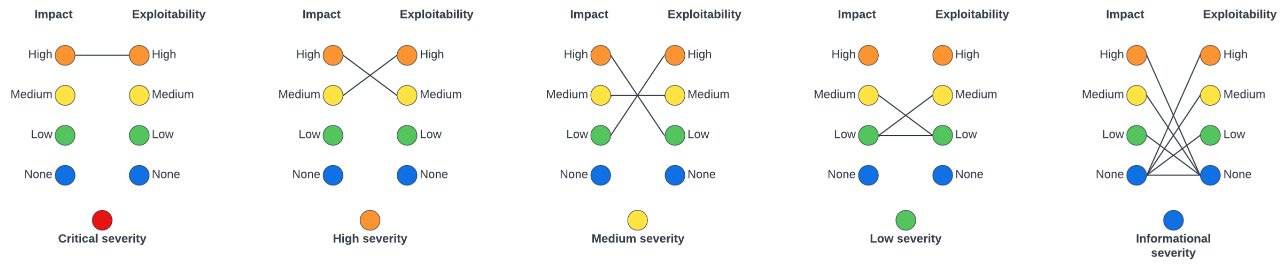
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.



As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	**Examples**
Critical	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/ incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.