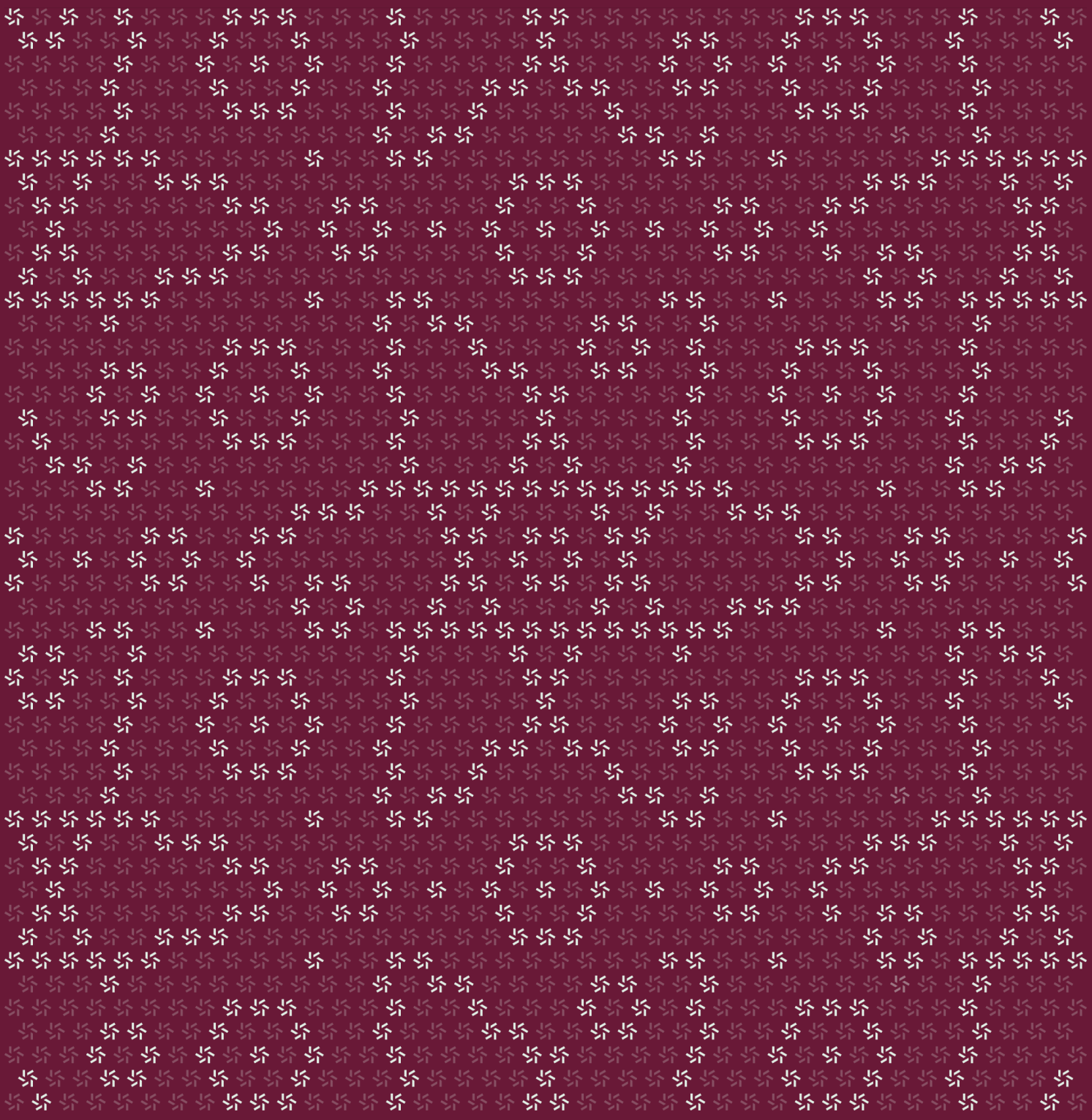
Zellic

Prepared for
Martin Lam
Nibir Pal
Bowen You
Fairblock Inc.

Prepared by
Nan Wang
Avraham Weinstock
Zellic

November 15, 2024

Fairyring
Blockchain Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Fairblock Inc. from October 28th to November 8th, 2024]. During this engagement, Zellic reviewed Fairyring's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are encrypted transactions private prior to the block in which they're decrypted?
- Are encrypted transactions correctly executed?
- Is it possible to halt the chain?
- Does the lane mechanism correctly order transactions from the mempool?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The `fairyringclient`, `fairyport`, and `ShareGenerationClient` components
- Modifications made to Cosmos-SDK and wasmd
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Fairyring modules, we discovered seven findings. Two critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature. The reported issues have been addressed and successfully resolved.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

Fairblock Inc. in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	1
Medium	1
Low	2
Informational	1

2. Introduction

2.1. About Fairyring

Fairblock Inc. contributed the following description of Fairyring:

> Fairblock is a dynamic confidentiality network that orchestrates high performance, low overhead, and custom confidential execution for efficient onchain markets and AI supply chains.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Nondeterminism. Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Complex integration risks. Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational"

finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Fairyring Modules

Type	Go
Platform	Cosmos
Target	fairyring
Repository	https://github.com/Fairblock/fairyring ↗
Version	`6d98dcf2263cda2fcd0408234fa2f0034c5afb77`
Programs	`fairyringd`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nan Wang
Engineer
nan@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 28, 2024	Start of primary review period
October 30, 2024	Kick-off call
November 8, 2024	End of primary review period

3. Detailed Findings

3.1. DOS vulnerability via `MsgRegisterContract` due to unlimited gas execution in `BeginBlock`

Target	fairyring/x/pep/module/module.go		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the `BeginBlock` function of the PEP module, contracts registered for execution at specific block heights are executed without gas restrictions, potentially leading to denial-of-service (DOS) attacks. Specifically, when the application chain reaches a specified height, it executes all contracts registered for that height without any gas limitation. If any of these contracts contain an infinite loop or complex computation, they can consume excessive resources and potentially halt the chain.

Here is a code snippet illustrating the execution of registered contracts:

```
// execute registered contracts
contracts, found := am.keeper.GetContractEntriesByID(ctx,
    strconv.FormatUint(h, 10))
if found && len(contracts.Contracts) != 0 {
    for _, contract := range contracts.Contracts {
        am.keeper.ExecuteContract(
            ctx,
            contract.ContractAddress,
            types.ExecuteContractMsg{
                Identity: strconv.FormatUint(h, 10),
                Pubkey: activePubkey.PublicKey,
                DecryptionKey: key.Data,
            },
        )
    }
}
```

Here is the ExecuteContract definition:

```
func (k Keeper) ExecuteContract(ctx sdk.Context, contractAddr string, msg
    types.ExecuteContractMsg) {
    addr := sdk.MustAccAddressFromBech32(contractAddr)
    msgBytes, err := json.Marshal(msg)
    if err != nil {
```

```
        k.logger.Error("error marshalling msg for contract: %s", contractAddr)
        return
    }

    wasmAddr := authtypes.NewModuleAddress(wasmtypes.ModuleName)
    _, err = k.contractKeeper.Execute(ctx, addr, wasmAddr, msgBytes,
    sdk.Coins{})
    if err != nil {
        k.logger.Error("error executing contract: %s; error: %v",
    contractAddr, err)
    }
}
```

Although the wasmd module's keeper internally tracks gas usage during contract execution, this gas consumption is not applied to any transaction signer in the `BeginBlock` phase, meaning there is no mechanism to halt execution if gas limits are exceeded. Currently, because the module uses an `InfiniteGasMeter` by default, there is no cap on gas usage, allowing infinite gas to be consumed. This configuration enables malicious users to deploy contracts that consume excessive gas through infinite loops or other resource-intensive processes.

Additionally, the wasmd module allows anyone to deploy contracts via `MsgStoreCode`, and the PEP module allows contracts to be registered for automatic execution via `MsgRegisterContract`. This opens an avenue for arbitrary code to be executed in `BeginBlock` without gas limitations, leading to potential DOS attacks.

Impact

This issue creates a DOS vulnerability. If a registered contract contains an infinite loop or resource-intensive computation, it could consume unbounded gas in `BeginBlock`, resulting in the chain becoming unresponsive. This could halt block production and disrupt network operations, as there is no gas limit enforced during execution in `BeginBlock`.

The following commands demonstrate how to deploy and register a contract containing an infinite loop, highlighting the DOS vulnerability:

1. Deploy the loop.wasm contract, which contains an infinite loop.

```
fairyringd tx wasm store loop.wasm -y --home devnet_data/fairyring_devnet
    --from wallet1 --keyring-backend test --chain-id fairyring_devnet --gas
    2000000
```

2. Instantiate the loop.wasm contract on the Fairyring network.

```
fairyringd tx wasm instantiate 1 '{}' -y --home devnet_data/fairyring_devnet
    --from wallet1 --keyring-backend test --chain-id fairyring_devnet --admin
    wallet1 --label foo
```

3. Query and retrieve the contract address for subsequent execution.

```
CONTRACT_ADDR=$(fairyringd q wasm list-contracts-by-code 1 --output=json | jq
    -r '.contracts[0]')
echo $CONTRACT_ADDR
```

4. Set the contract to execute at a future block height. This step uses the latest chain height and adds 30 blocks.

```
TARGET_HEIGHT=$(($(fairyringd q pep latest-height -o json | jq -r '.height')
    + 30))
echo $TARGET_HEIGHT
```

5. Register the contract for execution at the specified target height. The contract will automatically execute at this height.

```
fairyringd tx pep register-contract $CONTRACT_ADDR $TARGET_HEIGHT -y --home
    devnet_data/fairyring_devnet --from wallet1 --keyring-backend
    test --chain-id fairyring_devnet
```

Upon execution, the following log demonstrates the impact:

```
{"level":"info","module":"server","module":"x/pep",
"time":"2024-11-07T10:42:14Z","message":"Found Contract at block 86, Execute
    it!"}
{"level":"info","module":"server","module":"pex","numOutPeers":0,
"numInPeers":0,"numDialing":0,"numToDial":10,
"time":"2024-11-07T10:42:16Z","message":"Ensure peers"}
{"level":"info","module":"server","module":"pex",
"time":"2024-11-07T10:42:16Z","message":"No addresses to dial. Falling back to
    seeds"}
{"level":"info","module":"server","module":"consensus",
"dur":5000,"height":87,"round":0,"step":"RoundStepPropose",
"time":"2024-11-07T10:42:19Z","message":"Timed out"}
{"level":"info","module":"server","module":"p2p","book":
"/root/buildenv/src/app/fairyring/devnet_data/fairyring_devnet/
```

```
config/addrbook.json","size":0,
"time":"2024-11-07T10:42:46Z","message":"Saving AddrBook to file"}
{"level":"info","module":"server","module":"pex","numOutPeers":0,
"numInPeers":0,"numDialing":0,"numToDial":10,
"time":"2024-11-07T10:42:46Z","message":"Ensure peers"}
{"level":"info","module":"server","module":"pex",
"time":"2024-11-07T10:42:46Z","message":"No addresses to dial. Falling back to
    seeds"}
```

This setup demonstrates how an infinite loop contract can be registered and automatically executed at a specified height, illustrating the potential for DOS if gas limitations are not enforced.

Recommendations

Restrict the maximum gas that can be consumed per contract execution within `BeginBlock`. If the gas limit is exceeded, terminate the contract execution.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

3.2. DOS vulnerability from inaccurate gas estimation in `BeginBlock` via `simCheck`

Target	fairyring/x/pep/module/module.go		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the Fairyring project code, `decryptAndExecuteTx` uses `simCheck` to calculate the gas consumed by decrypted transactions and then deducts the corresponding fee:

```
    simCheckGas, _, err := am.simCheck(am.txConfig.TxEncoder(), txDecoderTx)
// We are using SimCheck() to only estimate gas for the underlying transaction
// Since user is supposed to sign the underlying transaction with Pep Nonce,
// is expected that we gets 'account sequence mismatch' error
// however, the underlying tx is not expected to get other errors
// such as insufficient fee, out of gas etc...
if err != nil && !strings.Contains(err.Error(), "account sequence mismatch") {
    am.processFailedEncryptedTx(ctx, eachTx, fmt.Sprintf("error while
    performing check tx: %s", err.Error()), startConsumedGas)
    return err
}
```

Typically, in the Cosmos SDK, transaction gas fees are automatically tracked and deducted during the `DeliverTx` phase. However, in the Fairyring code, decrypted transactions are processed in the `BeginBlock` phase, meaning the SDK's transaction context cannot be utilized to automatically calculate and deduct gas fees. This appears to be the reason for choosing `simCheck` as an alternative.

However, `simCheck` does not accurately reflect the actual gas consumption, as it does not truly execute the transaction; `simCheck` primarily runs `checkTx`, which only performs basic validation (such as `AnteHandler` and `ValidateBasic()` / `Validate()` functions). This is typically adequate for simpler messages with predictable gas consumption but not for more complex operations, especially when executing CosmWasm contracts, where actual gas usage can be significantly higher than estimated.

Impact

This issue could lead to a DOS vulnerability. For example, by crafting a message that calls a smart contract containing an infinite loop or a complex computation, an attacker could consume excessive resources, potentially causing the chain to become unresponsive. Since `simCheck` only estimates the gas, the actual execution cost is not fully captured, allowing the attacker to bypass accurate gas

deductions and exploit the network's resources.

This scenario demonstrates how an attacker could leverage an infinite loop within a CosmWasm contract to cause a DOS attack on the network.

1. Deploy the loop.wasm contract, which contains an infinite loop.

```
fairyringd tx wasm store loop.wasm -y --home devnet_data/fairyring_devnet
    --from wallet1 --keyring-backend test --chain-id fairyring_devnet --gas
    2000000 --gas-prices 1ufairy
```

2. Instantiate the loop.wasm contract on the Fairyring network.

```
fairyringd tx wasm instantiate 1 '{}' -y --home devnet_data/fairyring_devnet
    --from wallet1 --keyring-backend test --chain-id fairyring_devnet --admin
    wallet1 --label foo --gas-prices 1ufairy
```

3. Query and retrieve the contract address for subsequent execution.

```
CONTRACT_ADDR=$(fairyringd q wasm list-contracts-by-code 1 --output=json | jq
    -r '.contracts[0]')
echo $CONTRACT_ADDR
```

4. Create a transaction to execute the contract, which will trigger the infinite loop.

```
fairyringd tx wasm execute $CONTRACT_ADDR '{"identity": "", "pubkey": "",
    "decryption_key": ""}' -y --home devnet_data/fairyring_devnet --from
    wallet1 --keyring-backend test --chain-id fairyring_devnet --generate-only
    --gas-prices 1ufairy > execute_loop_unsigned.json
```

5. Sign the generated transaction to make it ready for submission.

```
fairyringd tx sign execute_loop_unsigned.json --home
    devnet_data/fairyring_devnet --from wallet1 --keyring-backend test
    --chain-id fairyring_devnet > execute_loop.json
```

6. Set the transaction to execute at a future block height. This step uses the latest chain height and adds 30 blocks.

```
TARGET_HEIGHT=$(($(fairyringd q pep latest-height -o json | jq -r '.height')
    + 30))
echo $TARGET_HEIGHT
```

7. Encrypt the signed transaction, which will be decrypted and executed at the specified block height.

```
fairyringd encrypt $TARGET_HEIGHT '' "$(cat execute_loop.json)" --node
    tcp://localhost:26657 --home devnet_data/fairyring_devnet >
    execute_loop.hex
```

8. Submit the encrypted transaction. This transaction will execute at the target height, attempting to run the contract with the infinite loop.

```
fairyringd tx pep submit-encrypted-tx $(cat execute_loop.hex) $TARGET_HEIGHT -y
    --home devnet_data/fairyring_devnet --from wallet1 --keyring-backend
    test --chain-id fairyring_devnet --gas-prices 1ufairy
```

Upon execution, the following log demonstrates the impact:

```
{"level":"info","module":"server","module":"consensus",
"dur":5000,"height":166,"round":0,"step":"RoundStepPropose",
"time":"2024-11-07T08:14:18Z","message":"Timed out"}
{"level":"info","module":"server","module":"pex","numOutPeers":0,
"numInPeers":0,"numDialing":0,"numToDial":10,
"time":"2024-11-07T08:14:26Z","message":"Ensure peers"}
{"level":"info","module":"server","module":"pex",
"time":"2024-11-07T08:14:26Z","message":"No addresses to dial. Falling back to
    seeds"}
```

This log indicates that the chain has become unresponsive due to executing the contract with an infinite loop, causing a time-out in the consensus process and stalling block production.

Recommendations

Replace or supplement `simCheck` with an accurate gas-metering mechanism that can fully execute and track the gas usage of decrypted transactions in `BeginBlock`. This would ensure that actual resource consumption aligns with gas deductions.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

3.3. Gas-price calculation error due to integer division rounding

Target	fairyring/x/pep/module/module.go		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The gas-fee calculation code may lead to inaccuracies due to rounding errors in integer division. Currently, the calculation sequence is as follows:

```
usedGasFee := sdk.NewCoin(
    txFee[0].Denom,
    // Tx Fee Amount Divide Provide Gas => provided gas price
    // Provided Gas Price * Gas Used => Amount to deduct as gas fee
    txFee[0].Amount.Quo(gasProvided).Mul(gasUsedInBig),
)
```

This calculation first divides `txFee[0].Amount` by `gasProvided`, both of which are integers. As a result, the division rounds down, potentially resulting in a loss of precision. For example, if `txFee[0].Amount` is 10,000 and `gasProvided` is 20,000, the result of 10,000 / 20,000 is 0. Multiplying by any `gasUsedInBig` value still results in zero, meaning the calculated gas fee could be significantly smaller than intended.

Impact

In this scenario, a malicious actor could exploit the rounding issue to avoid paying gas fees, potentially submitting numerous transactions at minimal or zero cost. This creates an imbalance by allowing a user to consume chain resources without bearing the intended transaction costs, leading to revenue loss and creating an unfair transaction environment.

The following POC demonstrates this issue in a controlled environment, showing how the chain calculates an incorrect gas fee of zero, even when gas was actually consumed. Using fairyringd, the commands below simulate a transaction with an encrypted payload and set gas price, highlighting the error in gas-fee deduction.

1. Prepare a transaction for encryption to execute a WASM contract **with a specified gas price of 0.5 ufairy**.

```
fairyringd tx wasm execute $CONTRACT_ADDR '{"identity": "", "pubkey": "",
    "decryption_key": ""}' -y --home devnet_data/fairyring_devnet --from
    wallet1 --keyring-backend test --chain-id fairyring_devnet --generate-only
    --gas-prices 0.5ufairy > execute_tx_unsigned.json
```

2. Sign the transaction.

```
fairyringd tx sign execute_tx_unsigned.json --home
    devnet_data/fairyring_devnet --from wallet1 --keyring-backend
    test --chain-id fairyring_devnet > execute_tx.json
```

3. Encrypt the signed transaction for submission.

```
fairyringd encrypt $TARGET_HEIGHT '' "$(cat execute_tx.json)" --node
    tcp://localhost:26657 --home devnet_data/fairyring_devnet > execute_tx.hex
```

4. Submit the encrypted transaction.

```
fairyringd tx pep submit-encrypted-tx $(cat execute_tx.hex) $TARGET_HEIGHT -y
    --home devnet_data/fairyring_devnet --from wallet1 --keyring-backend
    test --chain-id fairyring_devnet --gas-prices 1ufairy
```

Upon execution, the following log demonstrates the impact:

```
{"message":"Underlying tx consumed: 36596, decryption consumed: 95652"}
{"message":"gasUsedInBig: 132248, txFee[0].Amount: 100000, gasProvided:
    200000, usedGasFee: 0"}
{"message":"ChargedGas: 300000"}
{"message":"Deduct fee amount: [<nil>] | Refund amount: 300000"}
```

In this example, despite `gasUsedInBig` showing 132,248 units of gas consumed, the `usedGasFee` is calculated as 0 due to early rounding in integer division. As a result, the fee amount deducted is `[<nil>]`, while a full refund of 300,000 units is issued, effectively leading to no fee being charged for this transaction.

Recommendations

To avoid rounding errors, modify the calculation to multiply before dividing:

```
usedGasFee := sdk.NewCoin(
    txFee[0].Denom,
    txFee[0].Amount.Mul(gasUsedInBig).Quo(gasProvided),
)
```

This sequence maximizes precision and ensures the calculated gas fee accurately reflects the intended charge based on `gasUsedInBig` and `gasProvided`.

Alternatively, consider switching from integer division to safe floating-point division to avoid rounding issues.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

3.4. Insufficient error handling in gas deduction for failed transactions

Target	fairyring/x/pep/module/module.go		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In both `processFailedEncryptedTx` and `decryptAndExecuteTx`, there is insufficient error handling when deducting gas fees from an account with an insufficient balance.

1. When a transaction fails validation, `processFailedEncryptedTx` is triggered to account for gas used in operations like decryption. The function calls `handleGasConsumption` to deduct additional gas used from the account balance. However, if the account lacks sufficient funds, only an error log is created without further action.

```
am.handleGasConsumption(ctx, creatorAddr,
    cosmosmath.NewIntFromUint64(actualGasConsumed), tx.ChargedGas)
...
func (am AppModule) handleGasConsumption(ctx sdk.Context, recipient
    sdk.AccAddress, gasUsed cosmosmath.Int, gasCharged *sdk.Coin) {
    creatorAccount := am.accountKeeper.GetAccount(ctx, recipient)
    ...
    if gasUsed.GT(gasCharged.Amount) {
        deductFeeErr := ante.DeductFees(
            am.bankKeeper,
            ctx,
            creatorAccount,
            sdk.NewCoins(
                sdk.NewCoin(
                    gasCharged.Denom,
                    gasUsed.Sub(gasCharged.Amount)),
            ),
        )
        if deductFeeErr != nil {
            am.keeper.Logger().Error("deduct failed tx fee error")
            am.keeper.Logger().Error(deductFeeErr.Error())
        } else {
            am.keeper.Logger().Info("failed tx fee deducted without error")
        }
    }
```

2. Similarly, `decryptAndExecuteTx` deducts gas fees using `ante.DeductFees`, but it only logs an error if the deduction fails, without additional measures.

```
if refundAmount.IsZero() {
    deductFeeErr := ante.DeductFees(am.bankKeeper, ctx, creatorAccount,
    sdk.NewCoins(usedGasFee))
    if deductFeeErr != nil {
        am.keeper.Logger().Error("Deduct fee Err")
        am.keeper.Logger().Error(deductFeeErr.Error())
    } else {
        am.keeper.Logger().Info("Fee deducted without error")
    }
}
```

In both functions, only logging occurs if an account lacks sufficient funds, allowing potential abuse.

Impact

This insufficient error handling could allow users to consume gas resources without deductions if they lack funds. For example, users could repeatedly submit failing transactions, triggering these functions to consume gas but bypass gas fees. This behavior could waste network resources and allow abuse, as malicious users could exploit this to submit numerous failed transactions without incurring the intended gas charges.

Recommendations

To prevent abuse and ensure proper resource utilization, consider implementing additional handling in both `processFailedEncryptedTx` and `decryptAndExecuteTx` to address failed gas deductions due to insufficient funds.

We suggest charging the user `min(user balance, remaining gas)` instead of only logging the error. This ensures that users are still charged to the extent possible, even if their balance is insufficient to cover the full gas cost.

Additionally, consider implementing a fallback mechanism to halt further processing of transactions if gas deductions cannot be fully covered due to low balance.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in PR 236 ↗ and PR 237 ↗. The fixes were merged in commit 5b830510 ↗.

3.5. Inconsistent keyshare verification and logging due to Epoch switch

Target	fairyring/x/keyshare/keeper/msg_send_keyshare.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In the `SendKeyshare` function, each keyshare transaction triggers two calls to this function. The first call occurs during the simulate stage, where `SendKeyshare` is called to precheck the transaction's validity without affecting the blockchain state. The second call happens during the execution stage when the transaction is actually applied to the blockchain state within a block.

The issue arises if the block height coincides with an Epoch switch. In this case, the Epoch data used during the `checkTx` stage and the execution stage may be inconsistent, causing the `checkTx` stage to use outdated commitments data to verify a new Epoch's keyshare. This discrepancy results in misleading error logs.

Consider two Epochs: Epoch A ends at block 70, and Epoch B starts at block 71. At block 70, fairyring-client submits a keyshare transaction intended for block 71, meaning it should use Epoch B's commitments for verification.

During the `checkTx` stage. At block 70, the `checkTx` stage is still using Epoch A's commitments in `SendKeyshare`. When it verifies this keyshare transaction against Epoch B's keyshare, it fails to match and generates an error log indicating `keyshare verification failed`.

During the executions stage. At block 71, the transaction is included in a block and applied to the blockchain state. By this time, `BeginBlock` has switched the Epoch to Epoch B and updated the keyshare module's data. In this stage, `SendKeyshare` is called again, this time correctly using Epoch B's commitments, so no error log is generated.

Impact

This issue generates misleading error logs during the `checkTx` stage, suggesting keyshare verification failures even for valid transactions, which complicates debugging and consumes resources.

Recommendations

Enhance the logic in the `checkTx` stage to ensure that if the keyshare transaction height belongs to the next Epoch, the queued commitments are used for verification.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

3.6. Slice append issue

Target	fairyring/x/pep/module/module.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `BeginBlock` function in module.go contains an inappropriate usage of make and append when initializing a slice. In the following snippet, the `indexes` slice is preallocated with a specific size, but elements are added using append instead of directly filling each position. This results in default empty strings occupying the `[0...len(encryptedTxs.EncryptedTx)]` range, with the actual data appended after these default values.

```
func (am AppModule) BeginBlock(cctx context.Context) error {
    ...
    indexes := make([]string, len(encryptedTxs.EncryptedTx))
    for _, v := range encryptedTxs.EncryptedTx {
        indexes = append(indexes, strconv.FormatUint(v.Index, 10))
    }
    ctx.EventManager().EmitEvent(
        sdk.NewEvent(types.EncryptedTxDiscardedEventType,
            sdk.NewAttribute(types.EncryptedTxDiscardedEventTxIDs,
    strings.Join(indexes, ",")),
            sdk.NewAttribute(types.EncryptedTxDiscardedEventHeight,
    strconv.FormatUint(h, 10)),
        ),
    )
}
```

Impact

This incorrect population could result in data inconsistencies, especially when `indexes` is used in `EmitEvent`. Consequently, the `EncryptedTxDiscardedEventTxIDs` event attribute may contain unintended default values, potentially causing issues for off-chain services monitoring these events. Such services may misinterpret the event data, leading to inaccurate indexing, processing errors, or incorrect reporting on discarded transactions.

Recommendations

We recommend populating the slice directly instead of using append. For example,

```
for i, v := range encryptedTxs.EncryptedTx {
    indexes[i] = strconv.FormatUint(v.Index, 10)
}
```

Alternatively, initialize an empty slice without a preallocated size:

```
indexes := make([]string, 0)
// or simply
indexes := []string{}
```

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

3.7. Unremoved authorized addresses for unbonded validators in `BeginBlock`

Target	fairyring/x/keyshare/module/module.go		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `BeginBlock` function of the keyshare module, validators who are no longer in the bonded status are removed from the validator set. However, their associated authorized addresses are not removed, which can result in orphaned references to these validators. This is observed in the following code snippet:

```
bondedVal, err := am.stakingKeeper.GetValidator(ctx, sdk.ValAddress(accAddr))
if err != nil {
    am.keeper.RemoveValidatorSet(ctx, eachValidator.Validator)
    continue
}
if !bondedVal.IsBonded() {
    am.keeper.RemoveValidatorSet(ctx, eachValidator.Validator)
}
```

In this case, `RemoveValidatorSet` removes the validator from the set but does not clean up the authorized addresses associated with it. Although this does not currently lead to security issues — since there is a check before each use of authorized addresses — it leaves invalid references in `AllAuthorizedAddress`, which could result in minor inefficiencies or unintended behavior in future updates.

For comparison, the `DeRegisterValidator` function removes both the validator and its associated authorized addresses, as shown here:

```
// DeRegisterValidator removes a validator from the validator set and its
    associated authorized addresses.
func (k msgServer) DeRegisterValidator(goCtx context.Context, msg
    *types.MsgDeRegisterValidator) (*types.MsgDeRegisterValidatorResponse,
    error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    _, found := k.GetValidatorSet(ctx, msg.Creator)
    if !found {
        return nil, types.ErrValidatorNotRegistered.Wrap(msg.Creator)
```

```
    }

    for _, v := range k.GetAllAuthorizedAddress(ctx) {
        if v.AuthorizedBy == msg.Creator {
            k.RemoveAuthorizedAddress(ctx, v.Target)
            k.DecreaseAuthorizedCount(ctx, msg.Creator)
            break
        }
    }

    k.RemoveValidatorSet(ctx, msg.Creator)

    ctx.EventManager().EmitEvent(
        sdk.NewEvent(types.DeRegisteredValidatorEventType,
            sdk.NewAttribute(types.DeRegisteredValidatorEventCreator,
    msg.Creator),
        ),
    )

    return &types.MsgDeRegisterValidatorResponse{
        Creator: msg.Creator,
    }, nil
}
```

In `DeRegisterValidator`, authorized addresses linked to the validator are removed alongside the validator itself. This prevents orphaned references and ensures efficient resource management.

Impact

This issue leads to orphaned authorized addresses for validators no longer bonded, resulting in unused references in `AllAuthorizedAddress`. While this does not pose a direct security risk, it may lead to minor inefficiencies and complicates resource management.

Recommendations

To maintain efficient resource management and consistency, ensure that authorized addresses associated with removed validators are also removed in `BeginBlock`. Implementing this change will align `BeginBlock`'s behavior with `DeRegisterValidator`, which removes both the validator and its associated authorized addresses.

Remediation

This issue has been acknowledged by Fairblock Inc., and a fix was implemented in commit 5b830510 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Risks of executing messages in `BeginBlock`

The PEP module executes arbitrary user-controlled messages, including CosmWasm contract execution, in `BeginBlock`, which is non-standard. Since the usual Cosmos SDK transaction execution mechanism, `runTx`, is not used, the `BeginBlock` handler is responsible for limiting gas, handling panics and errors within individual transactions, and replicating the effects of what would normally be handled by AnteHandlers and PostHandlers, including checking that transactions are signed, deducting fees (or refunding fees, due to prepayment), and checking that transaction nonces are used in the expected order (which is the reason that `runTx` cannot be used in `BeginBlock` with the same AnteHandler stack as the rest of the application as encrypted transactions use an independent stream of nonces from regular transactions). This incurs maintenance overhead, as if additional AnteHandlers or PostHandlers are added, or if new mechanisms are added to `runTx` by upstream Cosmos SDK, the `BeginBlock` handler must be manually kept synchronized with these changes.

Additionally, CosmWasm contracts can execute submessages, which allows users to execute multiple messages atomically and respond to failure of a subset of them, so long as the total gas costs of executing submessages are within the prepaid limits. While this should be fine, since normal transactions allow the atomic execution of multiple messages, and thus message handlers are expected to handle this properly, this may be contrary to expectations for encrypted transactions, as `decryptAndExecuteTx` limits the top-level encrypted transaction to one message.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Keyshare

The keyshare module manages shares of keys for threshold decryption of transactions by a majority of the validator set.

Invariants

Active validators must submit a valid keyshare each block.

Test coverage

The individual messages are tested in Go unit tests.

Attack surface

Messages

`MsgUpdateParams`

The `MsgUpdateParams` message updates the parameters of the keyshare module.

- It must be signed by the keeper's `Authority`, which is initialized to the keyshare module's module address.
- It enforces that the `KeyExpiry`, `MinimumBonded`, and `MaxIdledBlock` fields are of type `uint64`, that the `AvgBlockTime` field is of type `float32`, that `SlashFractionNoKeyshare` and `SlashFractionWrongKeyshare` are decimals in the range $0 < x \leq 1$, and that `TrustedAddresses` is a list of Bech32-encoded addresses.

`MsgRegisterValidator`

The `MsgRegisterValidator` message registers a new validator with the keyshare module.

- It must be signed by the validator being registered.
- It checks that the signer has not already been registered with the keyshare module.

- It checks that the signer is registered with the staking module, and that it exceeds the keyshare module's minimum bonded token threshold.
- It then adds the validator to its validator set and marks it as if it had submitted a keyshare for the current block.

`MsgDeRegisterValidator`

The `MsgDeRegisterValidator` message deregisters an existing validator from the keyshare module.

- It must be signed by the validator being deregistered.
- It checks that the signer is a registered validator with the keyshare module.
- It removes the signer from the list of authorized addresses, decreases the count of addresses authorized by the signer, and removes the signer from its validator set.

`MsgSendKeyshare`

The `MsgSendKeyshare` message sends a keyshare for a validator at a specified height.

- The keyshare must be a hex string containing 96 bytes.
- The signer must be a registered validator or have been authorized by a registered validator.
- If the signer is a validator, they must not have authorized an address to submit keyshares other than the one they are signing with.
- The height specified must be either the current block's height or the next block's height.
- The keyshare must have been committed to at the specified keyshare index.
- If the keyshare is not what was committed to, the validator is slashed by a constant amount (this occurs spuriously in simulation on key-expiry boundaries; see Finding 3.5. ↗).

If the keyshare is valid,

- The keyshare is stored, keyed by the signer and target block height.
- The validator's latest submitted height is set to the current block.
- If the decryption key has not yet been constructed for the next block, and enough keyshares have been submitted to construct it, the decryption key is constructed and stored in both the keyshare module's state and the PEP module's state.

`MsgCreateLatestPubkey`

The `MsgCreateLatestPubkey` message creates a public key for the keysharing protocol for a future epoch.

- The public key must be a hex string containing 48 bytes.
- The message must specify the same number of validators, commitments, and encrypted keyshares, which must not be zero.

- Each commitment must be a hex string containing 48 bytes; each encrypted keyshare must be nonempty Base64-encoded data.
- The signer must be in the keyshare module's list of `TrustedAddresses`.
- A queued public key must not already exist.
- If there is no currently active public key, the new key's expiry height is set to the sum of the current block height and the `KeyExpiry` parameter; otherwise, the new key's expiry height is set to the sum of the current key's expiry height and the `KeyExpiry` parameter.
- The new public key, its encrypted shares, and the commitments are stored in the keyshare module's state as the queued public key and commitments, and the new public key is stored as the PEP module's queued public key.

MsgOverrideLatestPubkey

The `MsgOverrideLatestPubkey` message overrides the public key for the keysharing protocol for the current epoch.

- The public key must be a hex string containing 48 bytes.
- The message must specify the same number of validators, commitments, and encrypted keyshares, which must not be zero.
- Each commitment must be a hex string containing 48 bytes; each encrypted keyshare must be nonempty Base64-encoded data.
- The signer must be in the keyshare module's list of `TrustedAddresses`.
- The new key's expiry height is set to the sum of the current block height and the `KeyExpiry` parameter.
- The active and queued public key are deleted from the state if present.
- Active validators not assigned an encrypted keyshare from the message are removed.
- The new public key, its encrypted shares, and the commitments are stored in the keyshare module's state as the active public key and commitments, and the new public key is stored as the PEP module's active public key.

MsgCreateAuthorizedAddress

The `MsgCreateAuthorizedAddress` message allows a validator to authorize a different address from their validator address to participate in the keysharing protocol.

- The target address must not already have been authorized and must be distinct from the signer.
- The signer must be a validator registered with the keyshare module.
- The validator must not have authorized another address.
- The authorization is stored in the keyshare module's state.

MsgUpdateAuthorizedAddress

The `MsgCreateAuthorizedAddress` message updates whether the target address that a validator designated is authorized to continue to participate in the keysharing protocol.

- The target address must have previously been authorized by the signer.
- The target address must be distinct from the signer.
- The `IsAuthorized` flag is updated in the keyshare module's state based on the message's `IsAuthorized` flag.

MsgDeleteAuthorizedAddress

The `MsgDeleteAuthorizedAddress` message revokes authorization for an address to participate in the keysharing protocol on behalf of a validator.

- The target address must have previously been authorized by the signer.
- The signer must be either the target address or the validator address that provided authorization.
- The authorization is removed from the keyshare module's state.

MsgSubmitGeneralKeyshare

The `MsgSubmitGeneralKeyshare` message submits a keyshare for a general (non–target-height) identity.

- The signer must be a registered validator or have been authorized by a registered validator.
- If the signer is a validator, they must not have authorized another address to submit keyshares than the one they are signing with.
- The message's `IdType` must be one of the `SupportedIdTypes` (currently just `PrivateGovIdentity`).
- For `PrivateGovIdentity` id types, there must be a decryption key request stored in the state under the message's `IdValue` with a non-empty decryption key.
- The keyshare must have been committed to at the specified keyshare index.
- If the keyshare is not what was committed to, the validator is slashed by a constant amount.

If the keyshare is valid —

- The keyshare is stored as a general keyshare, keyed by the signer, ID type, and ID value.
- The validator's latest submitted height is set to the current block.
- If the decryption key has not yet been constructed for the specified identity, and enough keyshares have been submitted to construct it, the decryption key is constructed.

If the identity type for the constructed decryption key is `PrivateGovIdentity` —

- If the decryption key was requested over IBC, a `DecryptionKeyDataPacket` is sent over IBC to the chain that requested it with a 20-second time-out.
- If it was requested by a governance proposal, the decryption key is stored in the governance module's state as part of the proposal.
- Otherwise, the decryption key is stored in the PEP module under the request's identity and added to the PEP module's execution queue.

`MsgSubmitEncryptedKeyshare`

The `MsgSubmitEncryptedKeyshare` message submits an encrypted keyshare for a validator for a private decryption key request.

- The signer must be a registered validator, or have been authorized by a registered validator.
- If the signer is a validator, they must not have authorized an address to submit keyshares other than the one they are signing with.
- A `PrivateDecryptionKeyRequest` must exist for the specified identity.
- The specified keyshare commitment index must be in bounds.
- The encrypted keyshare is stored, and the validator's latest submit height is set to the current block.
- If enough encrypted keyshares are present to reconstruct the decryption key, a `PrivateDecryptionKeyDataPacket` is sent to the chain that initiated the request if the request was initiated over IBC; otherwise, the request is routed to the PEP module.

IBC packets

`RequestDecryptionKeyPacket`

The `RequestDecryptionKeyPacket` packet is used by the PEP module to request a decryption key from the keyshare module for a proposal or identity. The keyshare module acknowledges the packet with a either the active or queued decryption key based on the estimated arrival time of the response based on the average block chain, and it specifies the identity based on whether or not the key is for a proposal. The PEP module updates its state with the key if it did not already have a key for the specified identity.

`GetDecryptionKeyPacket`

The `GetDecryptionKeyPacket` packet is used by the PEP module to get a decryption key from the keyshare module for an identity. The keyshare module acknowledges the packet with either the active or queued pubkey based on the `DecryptionKeyRequest` state associated with the specified identity. The PEP module's packet-acknowledgment handler attempts to decode the response in `OnAcknowledgementGetDecryptionKeyPacket` as if it were a `RequestDecryptionKeyPacketAck`, and it does not make use of the decoded data.

`DecryptionKeyDataPacket`

The `DecryptionKeyDataPacket` packet is used by the keyshare module to send decryption keys to the PEP module. The PEP module stores the key and schedules execution if it has a pending request for the identity associated with the key. The keyshare module attempts retransmission if the PEP module acknowledges the packet with an error.

`PrivateDecryptionKeyDataPacket`

The `PrivateDecryptionKeyDataPacket` packet is used by the keyshare module to send encrypted keyshares to the PEP module. The PEP module stores the encrypted keyshares if it has a pending request for them.

`CurrentKeysPacket`

The `CurrentKeysPacket` packet is used by the PEP module to request the active and queued public keys from the keyshare module. The keyshare module acknowledges the packet with the active and queued public keys, and the PEP module updates its state when processing the packet acknowledgment if their expiries are newer.

`RequestPrivateDecryptionKeyPacket`

The `RequestPrivateDecryptionKeyPacket` is used by the PEP module to request encrypted keyshares from the keyshare module. The keyshare module stores the request in its state on receipt of the packet, and the PEP module stores the pending request on receiving the acknowledgment.

`GetPrivateDecryptionKeyPacket`

The `GetPrivateDecryptionKeyPacket` is used by the PEP module to get encrypted keyshares from the keyshare module. The keyshare module creates a request for the encrypted keyshares if one was not yet present in its state.

5.2. Pre-execution privacy (PEP)

The PEP module manages execution of encrypted transactions. It requests keys for decrypting the transactions from the keyshare module, which may reside on either the same chain or a different chain.

Invariants

Encrypted transactions must have their gas fees prepaid by their creators.

Test coverage

The individual messages are tested in Go unit tests, and flows involving sequences of messages are tested against a local devnet by scripts/tests/pep.sh.

Attack surface

`MsgUpdateParams`

The `MsgUpdateParams` message updates the parameters of the keyshare module.

- It must be signed by the keeper's `Authority`, which is initialized to the PEP module's module address.
- It validates that the `TrustedAddresses` is a list of Bech32-encoded addresses; that the `TrustedCounterParties` each have nonempty channel IDs, connection IDs, and client IDs; that the keyshare channel ID is nonempty; that the `MinGasPrice` and `PrivateDecryptionKeyPrice` are positive; and that the `IsSourceChain` field is of type `bool`.

`MsgSubmitEncryptedTx`

The `MsgSubmitEncryptedTx` message submits an encrypted transaction for execution at a target height.

- The target height must be ahead of the current block height or ahead of the latest source block height if another chain is the source chain.
- The target height must not exceed the expiry of the queued public key, or of the active public key if no queued public key is present.
- The creator's address must be Bech32 encoded.
- The PEP module transfers a gas prepayment from the creator to itself, and the encrypted transaction is stored keyed by the height.

`MsgSubmitGeneralEncryptedTx`

The `MsgSubmitEncryptedTx` message submits an encrypted transaction for a general encrypted identity.

- The identity must be present in the PEP module's state.
- The creator's address must be Bech32 encoded.
- The PEP module transfers a gas prepayment from the creator to itself, and the encrypted transaction is stored in the identity's transaction queue.

MsgSubmitDecryptionKey

The `MsgSubmitDecryptionKey` message submits a decryption key on non-source chains.

- If `IsSourceChain` is set, an error is returned.
- The signer must be in the PEP module's list of `TrustedAddresses`.
- There must be a nonempty active public key in the PEP module's state.
- The provided decryption key must be able to decrypt the string `"test data"` encrypted towards the active public key.
- If the trial decryption succeeds, the decryption key is stored, and the latest height from the source chain is stored.

MsgRequestGeneralIdentity

The `MsgRequestGeneralIdentity` message submits a request for a general identity.

- A request for the specified creator and ID must not already exist.
- If this chain is the source chain, an entry for the requested identity is stored.
- If another chain is the source chain, a `RequestDecryptionKeyPacket` for the requested identity is sent to the source chain.

MsgRequestGeneralDecryptionKey

The `MsgRequestGeneralDecryptionKey` message submits a request for a decryption key for a general identity.

- An entry for the identity must exist, and the entry's creator must be the signer.
- If this chain is the source chain, the decryption key request is stored.
- If another chain is the source chain, a `GetDecryptionKeyPacket` is sent to the source chain.

MsgRequestPrivateIdentity

The `MsgRequestPrivateIdentity` message submits a request for an identity with encrypted keyshares.

- A request for the specified creator and ID must not already exist.
- If this chain is the source chain, an entry for the request is stored.
- If another chain is the source chain, a `RequestPrivateDecryptionKey` packet is sent to the source chain.

MsgRequestPrivateDecryptionKey

The `MsgRequestPrivateDecryptionKey` message submits a request for encrypted keyshares.

- The identity for the request is created if it does not already exist.
- The creator's address must be Bech32 encoded.
- The PEP module transfers the `PrivateDecryptionKeyPrice` to itself.
- If this chain is the source chain, an entry for the request is stored.
- If another chain is the source chain, a `GetPrivateDecryptionKeyPacket` packet is sent to the source chain.

MsgRegisterContract

The `MsgRegisterContract` registers a WASM contract for execution when a specified identity's decryption key is made available.

- The contract's address must be Bech32 encoded, and a contract must have been instantiated at it.
- The signer must be the contract's admin and creator.
- There must not already be a scheduled execution of the contract at the specified identity.
- The contract is scheduled for execution in the specified identity's queue.

MsgUnregisterContract

The `MsgUnregisterContract` unregisters a previously registered WASM contract.

- The contract's address must be Bech32 encoded, and a contract must have been instantiated at it.
- The signer must be the contract's admin and creator.
- The specified identity's list of scheduled contracts is scanned, and if the specified contract is found, it is removed.

5.3. Lanes

The lane mechanism is used to ensure that messages that submit keyshares are executed before other messages to avoid front-running.

Invariants

- The keyshare lane only contains transactions that contain `MsgSubmitDecryptionKey` messages.

- The default lane does not contain any transactions with `MsgSubmitDecryptionKey` messages.

Test coverage

The tests in `abci/checktx/check_tx_test.go` test inserting transactions into a mempool that makes use of the lane mechanism. The tests in `lanes/keyshare/utils_test.go` test that transactions are encoded consistently with block-sdk, which the lane mechanism uses.

Attack surface

`PrepareLaneHandler`

The `PrepareLaneHandler` function is called as part of `PrepareProposal` by the block proposer. It extracts transactions containing `MsgSubmitDecryptionKey` from the mempool and places them in the `KeyshareLane`, validating them with `VerifyTx`, and discarding transactions that exceed `MaxTxBytes`.

`ProcessLaneHandler`

The `ProcessLaneHandler` function is called as part of `ProcessProposal` by all validators. It ensures that the keyshare lane only contains keyshare transactions and that no non-keyshare transactions are in the default lane.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed as a testnet, but not as a mainnet.

During our assessment on the scoped Fairyring modules, we discovered seven findings. Two critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature. The reported issues have been addressed and successfully resolved.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.