Zellic

Nibiru

Smart Contract Security Assessment

July 3, 2023

Prepared for:

Unique Divine

Nibiru

Prepared by:

Raj Agarwal and William Bowling

Zellic Inc.

Contents

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 Nibiru from May 15th to June 12th, 2023. During this engagement, Zellic reviewed Nibiru'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:

- Are calculations implemented correctly?
- Could an attacker manipulate the AMM (automated market maker) price?
- Could an attacker extract more funds than they are owed?

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.

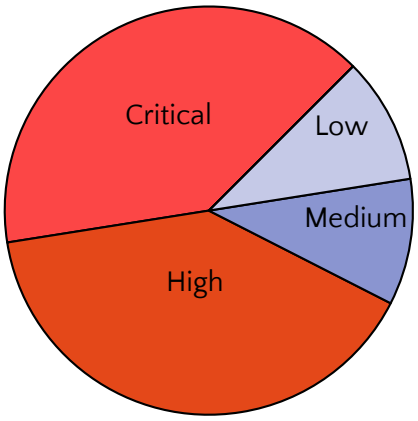
1.3 Results

During our assessment on the scoped Nibiru modules, we discovered 10 findings. Four critical issues were found. Four were of high impact, one was of medium impact, and one was of low impact. Nibiru acknowledged all findings and implemented fixes.

Additionally, Zellic recorded its notes and observations from the assessment for Nibiru's benefit in the Discussion section (6) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	4
Medium	1
Low	1
Informational	0



2 Introduction

2.1 About Nibiru

Nibiru is a sovereign proof-of-stake blockchain, open-source platform, and member of a family of interconnected blockchains that comprise the Cosmos Ecosystem.

Nibiru unifies leveraged derivatives trading, spot trading, staking, and bonded liquidity provision into a seamless user experience, enabling users of over 40 blockchains to trade with leverage using a suite of composable decentralized applications.

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 contracts.

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 the contracts' 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 code base in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. 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.

2.3 Scope

The engagement involved a review of the following targets:

Nibiru Modules

Repository	https://github.com/NibiruChain/nibiru
Version	nibiru: `24b8a7c8137115c4e5d556f38235fdc4ef5f655d`
Programs	• NibiruChain • cw-nibiru
Type	Cosmos
Platforms	Cosmos-SDK, CosmosWasm

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of four 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:

Raj Agarwal, Engineer
raj@zellic.io

William Bowling, Engineer
vakzz@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 16, 2023	Kick-off call
May 16, 2023	Start of primary review period
June 12, 2023	End of primary review period
July 10, 2023	Closing call

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.

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.

Zellic conducted a security assessment for Nibiru from May 15th to June 12th, 2023. During this engagement, Zellic reviewed Nibiru's code for security vulnerabilities, design issues, and general weaknesses in security posture. 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 calculations implemented correctly?
- Could an attacker manipulate the AMM (automated market maker) price?
- Could an attacker extract more funds than they are owed?

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. During our assessment on the scoped Nibiru modules, we discovered 10 findings. Four critical issues were found. Four were of high impact, one was of medium impact, and one was of low impact. Nibiru acknowledged all findings and implemented fixes.

Additionally, Zellic recorded its notes and observations from the assessment for Nibiru's benefit in the Discussion section (6) at the end of the document.

3 Executive Summary

Zellic conducted a security assessment for Nibiru from May 15th to June 12th, 2023. During this engagement, Zellic reviewed Nibiru's code for security vulnerabilities, design issues, and general weaknesses in security posture.

3.1 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:

- Are calculations implemented correctly?
- Could an attacker manipulate the AMM (automated market maker) price?
- Could an attacker extract more funds than they are owed?

3.2 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.

3.3 Results

During our assessment on the scoped Nibiru modules, we discovered 10 findings. Four critical issues were found. Four were of high impact, one was of medium impact, and one was of low impact. Nibiru acknowledged all findings and implemented fixes.

Additionally, Zellic recorded its notes and observations from the assessment for Nibiru's benefit in the Discussion section (6) at the end of the document.

Nibiru is a sovereign proof-of-stake blockchain, open-source platform, and member of a family of interconnected blockchains that comprise the Cosmos Ecosystem.

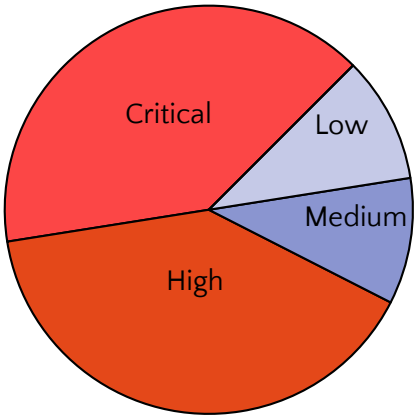
Nibiru unifies leveraged derivatives trading, spot trading, staking, and bonded liquid-

ity provision into a seamless user experience, enabling users of over 40 blockchains to trade with leverage using a suite of composable decentralized applications. Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of four calendar weeks. The key dates of the engagement are detailed below.

May 16, 2023	Kick-off call
May 16, 2023	Start of primary review period
June 12, 2023	End of primary review period
July 10, 2023	Closing call

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	4
Medium	1
Low	1
Informational	0



4 Introduction

4.1 About Nibiru

Nibiru is a sovereign proof-of-stake blockchain, open-source platform, and member of a family of interconnected blockchains that comprise the Cosmos Ecosystem.

Nibiru unifies leveraged derivatives trading, spot trading, staking, and bonded liquidity provision into a seamless user experience, enabling users of over 40 blockchains to trade with leverage using a suite of composable decentralized applications.

4.2 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 contracts.

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 the contracts' 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 code base in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. 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.

4.3 Scope

The engagement involved a review of the following targets:

Nibiru Modules

Repository	https://github.com/NibiruChain/nibiru
Version	nibiru: `24b8a7c8137115c4e5d556f38235fdc4ef5f655d`
Programs	• NibiruChain • cw-nibiru
Type	Cosmos
Platforms	Cosmos-SDK, CosmosWasm

4.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of four 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:

Raj Agarwal, Engineer
raj@zellic.io

William Bowling, Engineer
vakzz@zellic.io

4.5 Project Timeline

The key dates of the engagement are detailed below.

May 16, 2023	Kick-off call
May 16, 2023	Start of primary review period
June 12, 2023	End of primary review period
July 10, 2023	Closing call

5 Detailed Findings

5.1 Margin ratio not checked when removing collateral

- **Target**: x/perp/v2/keeper/margin.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

When removing margin from a position using `RemoveMargin`, there is a check to ensure that there is enough free collateral:

```
func (k Keeper) RemoveMargin(
    ctx sdk.Context, pair asset.Pair, traderAddr sdk.AccAddress,
    marginToRemove sdk.Coin,
) (res *v2types.MsgRemoveMarginResponse, err error) {
    // fetch objects from state
    market, err := k.Markets.Get(ctx, pair)
    if err ≠ nil {
        return nil, fmt.Errorf("%w: %s", types.ErrPairNotFound, pair)
    }

    amm, err := k.AMMs.Get(ctx, pair)
    if err ≠ nil {
        return nil, fmt.Errorf("%w: %s", types.ErrPairNotFound, pair)
    }
    if marginToRemove.Denom ≠ amm.Pair.QuoteDenom() {
        return nil, fmt.Errorf("invalid margin denom: %s",
    marginToRemove.Denom)
    }

    position, err := k.Positions.Get(ctx, collections.Join(pair,
    traderAddr))
    if err ≠ nil {
        return nil, err
    }

    // ensure we have enough free collateral
```

```
    spotNotional, err := PositionNotionalSpot(amm, position)
    if err != nil {
        return nil, err
    }
    twapNotional, err := k.PositionNotionalTWAP(ctx, position,
    market.TwapLookbackWindow)
    if err != nil {
        return nil, err
    }
    minPositionNotional := sdk.MinDec(spotNotional, twapNotional)

    // account for funding payment
    fundingPayment := FundingPayment(position,
    market.LatestCumulativePremiumFraction)
    remainingMargin := position.Margin.Sub(fundingPayment)

    // account for negative PnL
    unrealizedPnl := UnrealizedPnl(position, minPositionNotional)
    if unrealizedPnl.IsNegative() {
        remainingMargin = remainingMargin.Add(unrealizedPnl)
    }

    if remainingMargin.LT(marginToRemove.Amount.ToDec()) {
        return nil, types.ErrFailedRemoveMarginCanCauseBadDebt.Wrapf(
            "not enough free collateral to remove margin; remainingMargin
    %s, marginToRemove %s", remainingMargin, marginToRemove,
        )
    }

    if err = k.Withdraw(ctx, market, traderAddr, marginToRemove.Amount);
    err != nil {
        return nil, err
    }
```

The issue is that there is no check to ensure that the new margin ratio of the position is valid and that it is not underwater.

Impact

This allows someone to open a new position and then immediately remove 99.99% of the margin while effectively allowing them to have infinite leverage.

Recommendations

There should be a check on the margin ratio, similar to `afterPositionUpdate`, to ensure that it is not too low:

```
var preferredPositionNotional sdk.Dec
if positionResp.Position.Size_.IsPositive() {
    preferredPositionNotional = sdk.MaxDec(spotNotional, twapNotional)
} else {
    preferredPositionNotional = sdk.MinDec(spotNotional, twapNotional)
}

marginRatio := MarginRatio(*positionResp.Position,
    preferredPositionNotional, market.LatestCumulativePremiumFraction)
if marginRatio.LT(market.MaintenanceMarginRatio) {
    return v2types.ErrMarginRatioTooLow
}
```

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit ffad80c2.

5.2 AMM price manipulation using `openReversePosition`

- **Target**: x/perp/v2/keeper
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

The Nibiru perp module allows users to open reverse positions to decrease the margin, effectively shrinking the position size. A user can open a buy position and then immediately open a reverse position of the same size. Since `currentPositionNotional` is fractionally larger than `notionalToDecreaseBy`, it is possible to enter the `decreasePosition` flow as follows:

```
if currentPositionNotional.GT(notionalToDecreaseBy) {
    // position reduction
    return k.decreasePosition(
        ctx,
        market,
        amm,
        currentPosition,
        notionalToDecreaseBy,
        baseAmtLimit,
        /* skipFluctuationLimitCheck */ false,
```

This leaves the position with a zero size. Further in `afterPositionUpdate`, the position is not saved due to the following check:

```
func (k Keeper) afterPositionUpdate(
    ctx sdk.Context,
    market v2types.Market,
    amm v2types.AMM,
    traderAddr sdk.AccAddress,
    positionResp v2types.PositionResp,
) (err error) {
    [...]
    if !positionResp.Position.Size_.IsZero() {
        k.Positions.Insert(ctx, collections.Join(market.Pair,
    traderAddr), *positionResp.Position)
    }
```

However, the AMM is still updated in `decreasePosition` as though the position was saved.

```
func (k Keeper) decreasePosition(
    ctx sdk.Context,
    market v2types.Market,
    amm v2types.AMM,
    currentPosition v2types.Position,
    decreasedNotional sdk.Dec,
    baseAmtLimit sdk.Dec,
    skipFluctuationLimitCheck bool,
) (updatedAMM *v2types.AMM, positionResp *v2types.PositionResp, err error)
    {
    [...]
    updatedAMM, baseAssetDeltaAbs, err := k.SwapQuoteAsset(
        ctx,
        market,
        amm,
        dir,
        decreasedNotional,
        baseAmtLimit,
    )
```

Impact

An attacker could repeatedly open and close positions to manipulate the AMM price. They could then liquidate strong positions to make a profit.

Recommendations

It appears that the `afterPositionUpdate` function does not update a position with size zero because it assumes that it has already been deleted — for example in `closePositionEntirely`:

```
positionResp.ExchangedNotionalValue = exchangedNotionalValue
positionResp.Position = &v2types.Position{
    TraderAddress: currentPosition.TraderAddress,
    Pair: currentPosition.Pair,
    Size_: sdk.ZeroDec(),
    Margin: sdk.ZeroDec(),
    OpenNotional: sdk.ZeroDec(),
```

```
    LatestCumulativePremiumFraction:
    market.LatestCumulativePremiumFraction,
    LastUpdatedBlockNumber:  ctx.BlockHeight(),
}

err = k.Positions.Delete(ctx, collections.Join(currentPosition.Pair,
    trader))
```

Instead, a flag could be added to the `PositionResp` type to avoid updating a position after it has been deleted.

Remediation

This issue has been acknowledged by Nibiru, and fixes were implemented in the following commits:

- ffad80c2
- d47861fd

5.3 The sender is not checked for Wasm messages

- **Target**: x/wasm/binding/exec.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The CosmosWasm module has been enabled to allow developers to deploy smart contracts on Nibiru. To allow these contracts to interact with the chain, a custom executor has been written that will intercept and execute the appropriate custom calls:

```
type OpenPosition struct {
    Sender string  `json:"sender"`
    Pair string  `json:"pair"`
    IsLong bool    `json:"is_long"`
    QuoteAmount sdk.Int `json:"quote_amount"`
    Leverage sdk.Dec `json:"leverage"`
    BaseAmountLimit sdk.Int `json:"base_amount_limit"`
}

// DispatchMsg encodes the wasmVM message and dispatches it.
func (messenger *CustomWasmExecutor) DispatchMsg(
    ctx sdk.Context,
    contractAddr sdk.AccAddress,
    contractIBCPortID string,
    wasmMsg wasmvmtypes.CosmosMsg,
) (events []sdk.Event, data [][]byte, err error) {
    // If the "Custom" field is set, we handle a BindingMsg.
    if wasmMsg.Custom != nil {
        var contractExecuteMsg BindingExecuteMsgWrapper
        if err := json.Unmarshal(wasmMsg.Custom, &contractExecuteMsg);
    err != nil {
            return events, data, sdkerrors.Wrapf(err, "wasmMsg: %s",
    wasmMsg.Custom)
        }

        switch {
        // Perp module
        case contractExecuteMsg.ExecuteMsg.OpenPosition != nil:
            cwMsg := contractExecuteMsg.ExecuteMsg.OpenPosition
```

```
            _, err = messenger.Perp.OpenPosition(cwMsg, ctx)
            return events, data, err
...
```

These can then be called from a Cosmos contract:

```
/// NibiruExecuteMsg is an override of CosmosMsg::Custom. Using this msg
/// wrapper for the ExecuteMsg handlers show that their return values are
    valid
/// instances of CosmosMsg::Custom in a type-safe manner. It also shows
    how
/// ExecuteMsg can be extended in the contract.
#[cw_serde]
#[cw_custom]
pub struct NibiruExecuteMsg {
    pub route: NibiruRoute,
    pub msg: ExecuteMsg,
}

pub fn open_position(
    sender: String,
    pair: String,
    is_long: bool,
    quote_amount: Uint128,
    leverage: Decimal,
    base_amount_limit: Uint128,
) -> CosmosMsg<NibiruExecuteMsg> {
    NibiruExecuteMsg {
        route: NibiruRoute::Perp,
        msg: ExecuteMsg::OpenPosition {
            sender,
            pair,
            is_long,
            quote_amount,
            leverage,
            base_amount_limit,
        },
    }
    .into()
}
```

The issue is that there is no validation on the value of `sender`; it can be set to an arbitrary account and end up being sent straight to the message handler:

```
func (exec *ExecutorPerp) OpenPosition(
    cwMsg *cw_struct.OpenPosition, ctx sdk.Context,
) (
    sdkResp *perpv2types.MsgOpenPositionResponse, err error,
) {
    if cwMsg == nil {
        return sdkResp, wasmvmtypes.InvalidRequest{Err: "null open
    position msg"}
    }

    pair, err := asset.TryNewPair(cwMsg.Pair)
    if err ≠ nil {
        return sdkResp, err
    }

    var side perpv2types.Direction
    if cwMsg.IsLong {
        side = perpv2types.Direction_LONG
    } else {
        side = perpv2types.Direction_SHORT
    }

    sdkMsg := &perpv2types.MsgOpenPosition{
        Sender: cwMsg.Sender,
        Pair: pair,
        Side: side,
        QuoteAssetAmount: cwMsg.QuoteAmount,
        Leverage: cwMsg.Leverage,
        BaseAssetAmountLimit: cwMsg.BaseAmountLimit,
    }

    goCtx := sdk.WrapSDKContext(ctx)
    return exec.MsgServer().OpenPosition(goCtx, sdkMsg)
}
```

Impact

This allows a CosmosWasm contract to execute the `OpenPosition`, `ClosePosition`, `AddMargin`, and `RemoveMargin` operations on behalf of any user.

Recommendations

The `sender` should not be able to be arbitrarily set; it should be the address of the contract that is executing the message. If the `sender` needs to be configurable, only a whitelisted or trusted contract should be able to do it and that contract should have the appropriate checks to ensure the sender is set to the correct value.

Remediation

This issue has been acknowledged by Nibiru, and fixes were implemented in the following commits:

- bb898ae9
- 75041c3d

5.4 Wasm bindings do not validate messages

- **Target**: x/wasm/binding/exec.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

It was found that the Wasm bindings use messages directly after they are unmarshalled without calling `ValidateBasic`. The messages are directly passed to the handlers and crucial checks are skipped.

```
func (messenger *CustomWasmExecutor) DispatchMsg(
    ctx sdk.Context,
    contractAddr sdk.AccAddress,
    contractIBCPortID string,
    wasmMsg wasmvmtypes.CosmosMsg,
) (events []sdk.Event, data [][]byte, err error) {
    // If the "Custom" field is set, we handle a BindingMsg.
    if wasmMsg.Custom != nil {
        var contractExecuteMsg BindingExecuteMsgWrapper
        if err := json.Unmarshal(wasmMsg.Custom, &contractExecuteMsg);
    err != nil {
            return events, data, sdkerrors.Wrapf(err, "wasmMsg: %s",
    wasmMsg.Custom)
        }

        switch {
        // Perp module
        case contractExecuteMsg.ExecuteMsg.OpenPosition != nil:
            cwMsg := contractExecuteMsg.ExecuteMsg.OpenPosition
            _, err = messenger.Perp.OpenPosition(cwMsg, ctx)
```

Any checks that the handlers rely on `ValidateBasic` for are skipped and can be exploited if the respective checks are not present in the handlers.

Impact

The following are the examples of messages that can be exploited:

- For one, `ExecuteMsg.AddMargin` does not check if the margin denom is the same as the pair denom. This could allow incorrect collateral to be used.
- Another is that `ExecuteMsg.RemoveMargin` does not check that the amount to remove is positive, allowing the margin of a position to be increased without transferring any funds from the user. The inflated marging could then be withrawn to drain the `VaultModuleAccount` and `PerpEFModuleAccount` pools.

Recommendations

After creating each `sdkMsg`, the `ValidateBasic()` should be called on each before they are passed to the `MsgServer` in the executor.

Remediation

This issue has been acknowledged by Nibiru, and fixes were implemented in the following commits:

- ba58517e
- da51fdf0

5.5 Incorrect TWAP calculation

- **Target**: x/oracle/keeper/keeper.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The oracle module uses the `calcTwap` to compute the TWAP (time-weighted average price). Here, the maximum of `snapshots[0].TimestampMs` and `ctx.BlockTime().UnixMilli() - twapLookBack` is used as `firstTimeStamp`.

```
func (k Keeper) calcTwap(ctx sdk.Context, snapshots []types.PriceSnapshot)
    (price sdk.Dec, err error) {
    [ ... ]
    firstTimeStamp := ctx.BlockTime().UnixMilli() - twapLookBack
    cumulativePrice := sdk.ZeroDec()

    firstTimeStamp = math.MaxInt64(snapshots[0].TimestampMs,
    firstTimeStamp)
    [ ... ]
            nextTimestampMs = snapshots[i+1].TimestampMs
        }

        price := s.Price.MulInt64(nextTimestampMs - timestampStart)
```

This is not sound as it is possible for the price to be negative if `timestampStart` is greater than `nextTimestampMs`.

Impact

If `timestampStart` is greater than `nextTimestampMs`, the resulting TWAP data will be incorrect. However, this is not an issue currently since the caller for `calcTwap` only includes snapshots starting from `ctx.BlockTime().UnixMilli() - twapLookBack`.

Recommendations

Ideally, `firstTimeStamp` should always just be equal to the timestamp of the first snapshot.

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit 53487734.

5.6 Panic in `EndBlock` hooks will halt the chain

- **Target**: x/inflation, x/oracle
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: High
- **Impact**: High

Description

When executing a transaction, Cosmos automatically handles any panics that may occur with the default recovery middleware (see runtx_middleware), but this is not the case for anything that runs within an `EndBlock` or `BeginBlock` hook. In these cases it is vital that there are no panics and that all errors are handled correctly; otherwise, it will result in a chain halt as all the validators will panic and crash.

The following locations are all reachable from an `EndBlock` or `BeginBlock` (`AfterEpochEnd` is called from a `BeginBlock`):

- x/inflation/keeper/hooks.go#L64-L64
- x/oracle/keeper/slash.go#L52-L52
- x/oracle/keeper/update_exchange_rates.go#L80-L80
- x/oracle/keeper/reward.go#L71-L71
- x/oracle/keeper/reward.go#L60-L60
- x/oracle/keeper/ballot.go#L69-L69
- x/oracle/types/ballot.go#L111-L111

Impact

If any of these error conditions are met, there will be a chain halt as all the validators will crash.

Recommendations

The panics should be replaced with the appropriate error handling for each case and either log the error or fail gracefully.

Remediation

This issue has been acknowledged by Nibiru, and fixes were implemented in the following commits:

- 73d9bfd4
- 85859f2b

5.7 The `ReserveSnapshots` are never updated

- **Target**: x/perp/v2/module/abci.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The perp module has an `EndBlocker`, which is designed to create a snapshot of the AMM in order to calculate the TWAP prices:

```
// EndBlocker Called every block to store a snapshot of the perpamm.
func EndBlocker(ctx sdk.Context, k keeper.Keeper) []abci.ValidatorUpdate {
    for _, amm := range k.AMMs.Iterate(ctx,
    collections.Range[asset.Pair]{}).Values() {
        snapshot := types.ReserveSnapshot{
            Amm:  amm,
            TimestampMs:  ctx.BlockTime().UnixMilli(),
        }
        k.ReserveSnapshots.Insert(ctx, collections.Join(amm.Pair,
    ctx.BlockTime()), snapshot)
    }
    return []abci.ValidatorUpdate{}
}
```

The issue is that the `EndBlocker` is not hooked up and is never called.

Impact

The `ReserveSnapshots` are never updated and so anything relying on it (such as `CalcTwap`) will be using whatever values were set during genesis.

Recommendations

The `EndBlocker` should be called from the perp module's `EndBlock`:

```
func (am AppModule) EndBlock(ctx sdk.Context, _ abci.RequestEndBlock)
    []abci.ValidatorUpdate {
    EndBlocker(ctx, am.keeper)
    return []abci.ValidatorUpdate{}
}
```

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit 7144cc96.

5.8 Distributing zero coins causes chain halt

- **Target**: x/oracle/keeper/hooks.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The oracle module uses an `AfterEpochEnd` hook, which allocates rewards for validators. This hook is inside the `BeginBlocker`.

```
func (h Hooks) AfterEpochEnd(ctx sdk.Context, epochIdentifier string, _
    uint64) {
        [...]
        balances := h.bankKeeper.GetAllBalances(ctx,
    account.GetAddress())
        for _, balance := range balances {
            validatorFees
    := balance.Amount.ToDec().Mul(params.ValidatorFeeRatio).TruncateInt()
            rest := balance.Amount.Sub(validatorFees)
            totalValidatorFees = append(totalValidatorFees,
    sdk.NewCoin(balance.Denom, validatorFees))
            totalRest = append(totalRest, sdk.NewCoin(balance.Denom,
    rest))
        }

        [...]

        err = h.k.AllocateRewards(
            ctx,
            perptypes.FeePoolModuleAccount,
            totalValidatorFees,
            1,
        )
        if err ≠ nil {
            panic(err)
        }
```

The issue here is that `validatorFees` could be zero for very small positions. This means `AllocateRewards` could be called with one or more coins with a zero amount.

Impact

The `AllocateRewards` function in turn calls `bankKeeper.SendCoinsFromModuleToModule`, which will fail if any of the coins have a nonpositive amount.

```
func (coins Coins) Validate() error {
        [...]
        if err := ValidateDenom(coins[0].Denom); err ≠ nil {
            return err
        }
        if !coins[0].IsPositive() {
            return fmt.Errorf("coin %s amount is not positive", coins[0])
        }
```

Since the `AfterEpochEnd` hook is inside the `BeginBlocker`, this will cause the chain to halt.

Recommendations

If the final value of `totalValidatorFees` is not greater than zero then the call to `h.k.AllocateRewards` should not be made.

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit c430556a.

5.9 Large `rewardSpread` due to miscalculation

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

Description

The oracle module uses the `rewardSpread` to check if the price data from the validator is within an acceptable range from the chosen price.

```
func Tally(ballots types.ExchangeRateBallots, rewardBand sdk.Dec,
    validatorPerformances types.ValidatorPerformances) sdk.Dec {
    sort.Sort(ballots)

    weightedMedian := ballots.WeightedMedianWithAssertion()
    standardDeviation := ballots.StandardDeviation(weightedMedian)
    rewardSpread := weightedMedian.Mul(rewardBand.QuoInt64(2))

    if standardDeviation.GT(rewardSpread) {
        rewardSpread = standardDeviation
```

```
sum := sdk.ZeroDec()
for _, v := range pb {
    deviation := v.ExchangeRate.Sub(median)
    sum = sum.Add(deviation.Mul(deviation))
}
```

The standard deviation for the ballots is used directly as the `rewardSpread` if it is greater than the calculated `rewardSpread`.

```
if standardDeviation.GT(rewardSpread) {
    rewardSpread = standardDeviation
```

The `StandardDeviation` function, however, does not ignore negative votes. This could allow a malicious validator to submit abstaining votes with very large negative values and increase the `rewardSpread`.

Impact

Two malicious validators could collude to repeatedly submit prices outside the acceptable price band. They can do this without being slashed due to `rewardSpread` having a very high value. If eventually the attacker succeeds in publishing an invalid price, they could profit by liquidating strong postions through the perp module.

Recommendations

Abstained votes should be ignored when calculating the standard deviation for the ballots.

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit 908571f0.

5.10 Iterating over maps is nondeterministic

- **Target**: x/oracle/keeper
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Low

Description

It is vitally important that any calculations done by the chain are deterministic and can be reproduced by every validator so that the state of the network can be agreed upon. One source of nondeterminism in Go is iterating over maps: the specification states, "The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next" (https://go.dev/ref/spec#RangeClause).

The `removeInvalidBallots` and `countVotesAndUpdateExchangeRates` functions are both called from an `EndBlocker` and iterate over a map:

```
func (k Keeper) removeInvalidBallots(
    ctx sdk.Context,
    pairBallotsMap map[asset.Pair]types.ExchangeRateBallots,
) (map[asset.Pair]types.ExchangeRateBallots, set.Set[asset.Pair]) {
    whitelistedPairs := set.New(k.GetWhitelistedPairs(ctx)...)

    totalBondedPower
    := sdk.TokensToConsensusPower(k.StakingKeeper.TotalBondedTokens(ctx),
    k.StakingKeeper.PowerReduction(ctx))
    thresholdVotingPower
    := k.VoteThreshold(ctx).MulInt64(totalBondedPower).RoundInt()
    minVoters := k.MinVoters(ctx)

    for pair, ballots := range pairBallotsMap {
        // If pair is not whitelisted, or the ballot for it has failed,
    then skip
        // and remove it from pairBallotsMap for iteration efficiency
        if _, exists := whitelistedPairs[pair]; !exists {
            delete(pairBallotsMap, pair)
            continue
        }

        // If the ballot is not passed, remove it from the
    whitelistedPairs set
        // to prevent slashing validators who did valid vote.
```

```
        if !isPassingVoteThreshold(ballots, thresholdVotingPower,
    minVoters) {
            delete(whitelistedPairs, pair)
            delete(pairBallotsMap, pair)
            continue
        }
    }

    return pairBallotsMap, whitelistedPairs
}

func (k Keeper) countVotesAndUpdateExchangeRates(
    ctx sdk.Context,
    pairBallotsMap map[asset.Pair]types.ExchangeRateBallots,
    validatorPerformances types.ValidatorPerformances,
) {
    rewardBand := k.RewardBand(ctx)

    for pair, ballots := range pairBallotsMap {
        exchangeRate := Tally(ballots, rewardBand, validatorPerformances)

        k.SetPrice(ctx, pair, exchangeRate)

        ctx.EventManager().EmitEvent(
            sdk.NewEvent(types.EventTypeExchangeRateUpdate,
                sdk.NewAttribute(types.AttributeKeyPair, pair.String()),
                sdk.NewAttribute(types.AttributeKeyExchangeRate,
    exchangeRate.String()),
            ),
        )
    }
}
```

Impact

Currently, the operations that are performed in `k.SetPrice` end up being the same regardless of the order it is called, so it is unlikely to cause a chain halt. However, a simple unrelated change to a keeper in the future could cause this issue to occur as the order that the pairs are processed can be different.

Recommendations

Instead of iterating over the map, the function should iterate over a sorted list of keys.

```
// countVotesAndUpdateExchangeRates processes the votes and updates the
    ExchangeRates based on the results.
func (k Keeper) countVotesAndUpdateExchangeRates(
    ctx sdk.Context,
    pairBallotsMap map[asset.Pair]types.ExchangeRateBallots,
    validatorPerformances types.ValidatorPerformances,
) {
    rewardBand := k.RewardBand(ctx)

    keys := make([]string, 0, len(pairBallotsMap))
    for key := range pairBallotsMap {
        keys = append(keys, key.String())
    }
    sort.Strings(keys)

    for _, key := range keys {
        pair := asset.MustNewPair(key)
        ballots := pairBallotsMap[pair]
        exchangeRate := Tally(ballots, rewardBand, validatorPerformances)

        k.SetPrice(ctx, pair, exchangeRate)

        ctx.EventManager().EmitEvent(
            sdk.NewEvent(types.EventTypeExchangeRateUpdate,
                sdk.NewAttribute(types.AttributeKeyPair, pair.String()),
                sdk.NewAttribute(types.AttributeKeyExchangeRate,
    exchangeRate.String()),
            ),
        )
    }
}
```

Remediation

This issue has been acknowledged by Nibiru, and a fix was implemented in commit 3482a22d.

6 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

6.1 Logic of `change_admin` is inconsistent with `instantiate`

When instantiating the CosmosWasm contracts, the `admin` is set up and also added as a member:

```
#[entry_point]
pub fn instantiate( eps: DepsMut, _env: Env, _info: MessageInfo,
    msg: InitMsg) -> StdResult<Response> {
    let whitelist = Whitelist {
        members: vec![msg.admin.clone()].into_iter().collect(),
        admin: msg.admin,
    };
    WHITELIST.save(deps.storage, &whitelist)?;
    Ok(Response::default())
}
```

But when `change_admin` is called, the new `admin` is removed from the member list (and the old admin is kept as a member):

```
ExecuteMsg::ChangeAdmin { address } => {
    check_admin(check)?;
    let api = deps.api;
    let addr = api.addr_validate(address.as_str()).unwrap();
    whitelist.admin = addr.clone().into_string();
    whitelist.members.remove(addr.as_str());
    WHITELIST.save(deps.storage, &whitelist)?;
```

Consider changing the logic so that `change_admin` does not affect the member list (which can be done in a separate call to `add_member` or `remove_member`) or to make it match the `instantiate` method so that the `admin` becomes a member.

7 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts 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.

7.1 Module: x/oracle

Message: `AggregateExchangeRatePrevote`

The `AggregateExchangeRatePrevote` handler is responsible for submitting an aggregate exchange rate prevote, which is part of the commit-reveal scheme used by the oracle module to reduce the risk of centralization and of free riders.

The parameters that are controllable by the user are

- `Hash` — This is the hash of the exchange rates that will be revealed in the aggregate exchange rate vote, which is calculated as the hex string of `SHA256("{salt}:({pair},{exchange_rate})|...|({pair},{exchange_rate}):{voter}")`.
- `Feeder` — This is the address of the price feeder that is submitting the aggregate exchange rate prevote. This address must have signed the message.
- `Validator` — This is the address of the validator that is submitting the aggregate exchange rate prevote. This may be the same as the `Feeder` address if there is no delegation.

Before recording the prevote, the `Feeder` is checked to ensure that it is the same as the `Validator` or that the `Validator` has delegated to the `Feeder`. The `Validator` is also checked to ensure that it is a valid bonded validator.

Message: `MsgAggregateExchangeRateVote`

The `MsgAggregateExchangeRateVote` handler is responsible for submitting an aggregate exchange rate vote, which must match a previously submitted aggregate exchange rate prevote.

The parameters that are controllable by the user are

- `Salt` — This is the salt used to generate the aggregate vote hash; it must be a

string of length 1 to 4.

- `ExchangeRates` — These are the exchange rates to vote on; they must be a string of exchange rate tuples in the format of `({pair},{exchange_rate})|...|({pair},{exchange_rate})`, for example `(nibi:usd,100)|(btc:usd,200000)`.
- `Feeder` — This is the address of the feeder that is submitting the vote. This must be a valid address and the signer of the message.
- `Validator` — This is the validator address to submit the vote for. This must be a valid address.

Before recording the vote, the `Feeder` is checked to ensure that it is the same as the `Validator` or that the `Validator` has delegated to the `Feeder`. The `Validator` is also checked to ensure that it is a valid bonded validator. Then the prevote is checked to ensure that it exists and that the recorded hash matches the hash generated from the parameters. Finally, the exchange rates are parsed and checked to ensure that they are valid.

Message: `MsgDelegateFeedConsent`

The `MsgDelegateFeedConsent` handler is responsible for delegating oracle voting rights to another address.

The parameters that are controllable by the user are

- `Operator` — This is the validator address to delegate oracle voting rights from.
- `Delegate` — This is the address to delegate oracle voting rights to.

The `Operator` must have signed the message and also be a validator. The `Delegate` must be a valid address. Once called, the `Delegate` will be able to vote on oracle votes on behalf of the `Operator`.

Hook: `SlashAndResetMissCounters`

The `SlashAndResetMissCounters` hook is called every `SlashWindow` blocks. It iterates through all validators that were recorded as missing votes in the previous `SlashWindow` blocks and checks if they missed more than `MinValidPerWindow` of the votes. If so, it slashes the validator and resets their miss counter to zero.

Hook: `UpdateExchangeRates`

The `UpdateExchangeRates` hook is called every `VotePeriod` blocks and is responsible for updating the exchange rates based on all the votes that have been cast from the validators. The basic flow is

1. Reset the current exchange rates by removing them all from the store.

2. Split the votes into ballots by pair, removing any votes from nonbonded or jailed validators, any non-whitelisted pairs, pairs with less than the minimum number of votes, or pairs that have less than the voting power threshold.
3. For each valid pair, tally up the votes and find the weighted median exchange rate and record the validators that voted for an exchange rate that was within the reward band, and then update the exchange rate.
4. Count up the validators who missed the vote and increase the appropriate miss counters.
5. Distribute rewards to the validators that voted for an exchange rate that was within the reward band.
6. Clear previous prevotes and all votes from the store.

7.2 Module: x/perp/v2

Message: `AddMargin`

The `AddMargin` handler is responsible for adding collateral to an existing position to increase the margin ratio.

The parameters that are controllable by the user are

- `pair` — This is the pair of the open position to add collateral to; it is checked to ensure that it is valid, that there is an open position, and there is a market and an AMM for it.
- `marginToAdd` — This is the amount of coins to add to the position; it is checked to ensure that the amount is positive and that the denom is the same as the specified pair.

When adding margin, any outstanding funding payment is realized and the position `LatestCumulativePremiumFraction` is updated to match the market's value.

Message: `MsgClosePosition`

The `MsgClosePosition` handler is responsible for closing an existing position and returning any remaining margin back to the user, or it realizes any bad debt if there is any.

The only parameter that is controllable by the user is the `pair`, which is checked to ensure that it is valid and that there is an open position, a market, and an AMM for the pair.

Message: `MsgMultiLiquidate`

The `MsgMultiLiquidate` handler is responsible for liquidating underwater positions. It can be called by anyone with a list of trader and pair mappings and will attempt to partially or fully liquidate each position.

The parameter that is controllable by the user is

- `liquidationRequests` — This is an array consisting of groups of a trader address and a pair. Each address and pair is validated to ensure they are well-formed.

When liquidating multiple positions, the transaction will fail if all the liquidations are unsuccessful, but it will return a success if there is at least one valid liquidation.

For each position, the margin ratio is checked, and if it is greater than the market's specified `MaintenanceMarginRatio`, then the liquidation fails with `LiquidationFailedEvent_POSITION_HEALTHY`. Otherwise, if the spot margin ratio is greater or equal to the market's `LiquidationFeeRatio` (there is enough margin to pay the liquidation fee), a partial liquidation is performed by reducing the size to the market's `PartialLiquidationRatio`. If there is not enough margin to pay the liquidation fee, then a full liquidation is performed and the position is closed out.

Message: `MsgOpenPosition`

The `MsgOpenPosition` handler is responsible for creating a new long or short position on a specified pair as well as increasing or decreasing an existing position. The parameters that a user can control are

- `pair` — This is the pair to open or modify the position on. It is checked to ensure that the format is valid, that an enabled market exists for it, and that there is an AMM for it.
- `dir` — This is the direction the user is taking and must be either `Direction_SHORT` or `Direction_LONG`.
- `quoteAssetAmt` — This is the amount of quote asset to open a position with; it must be greater than zero.
- `leverage` — This is the leverage to open a position with; it must be positive and not greater than the market's `MaxLeverage` setting.
- `baseAmtLimit` — This is the minimum base asset amount to open a position with; it must not be negative.

Message: `MsgRemoveMargin`

The `MsgRemoveMargin` handler is responsible for removing collateral from an existing position and decreasing the margin ratio.

The parameters that are controllable by the user are

- `pair` — This is the pair of the open position to remove collateral. It is checked to ensure that it is valid and that there is an open position, a market, and an AMM for it.
- `margin` — This is the amount of coins to remove from the position. It is checked to ensure that the amount is positive and that the denom is the same as the specified pair.

When removing margin, any outstanding funding payment is realized and the position's `LatestCumulativePremiumFraction` is updated to match the market's value. The method does not check the margin ratio, which allows you to remove the majority of the collateral (see 5.1).

7.3 Module: x/wasm

Message: `ExecuteMsg.AddMargin`

This message can be sent via a CosmosWasm contract and allows for a collateral to be added to an existing position. The parameters that a user can control are

- `Sender` — This is the account holding the position. There are no checks (see 5.3).
- `Pair` — This is the pair of the position to be closed. It is checked to ensure that it is valid and that there is an open position, a market, and an AMM for the pair.
- `Margin` — This is the amount of collateral to add to the position. There are no checks on the amount or the denom to ensure it matches the pair (see 5.4).

The flow is then the same as the regular `AddMargin` transaction; see 7.2.

Message: `ExecuteMsg.ClosePosition`

This message can be sent via a CosmosWasm contract and allows for a position to be closed. The parameters that a user can control are

- `Sender` — This is the account holding the position; there are no checks (see 5.3).
- `Pair` — This is the pair of the position to be closed. It is checked to ensure that it is valid and that there is an open position, a market, and an AMM for the pair.

The flow is then the same as the regular `ClosePosition` transaction; see 7.2.

Message: `ExecuteMsg.CreateMarket`

This message can be sent via a CosmosWasm contract and allows a new pool to be created for a specific pair.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `Pair` — This is the pair of the market to be created. It is checked to ensure that it is valid and that there is no existing market for it.
- `PegMult` — This is the peg multiplier to set for the market; it must be greater than zero.
- `SqrtDepth` — This is the square root of the depth multiplier to set for the market; it must be greater than zero.
- `MarketParams` —
  - `PriceFluctuationLimitRatio`: This is the percentage that a single open or close position can alter the reserve amounts; it must be between 0 and 1.
  - `MaintenanceMarginRatio`: This is the minimum margin ratio that a user must maintain on this market,; it must be between 0 and 1.
  - `MaxLeverage`: This is the maximum leverage a user is able to be taken on this market; it must be greater than zero.
  - `LatestCumulativePremiumFraction`: This is the latest cumulative premium fraction for a given pair.
  - `ExchangeFeeRatio`: This is the percentage of the notional given to the exchange when trading; it must be between 0 and 1.
  - `EcosystemFundFeeRatio`: This is the percentage of the notional transferred to the ecosystem fund when trading; it must be between 0 and 1.
  - `LiquidationFeeRatio`: This is the percentage of liquidated position that will be given as a reward. Half of the liquidation fee is given to the liquidator, and the other half is given to the ecosystem fund, it must be between 0 and 1.
  - `PartialLiquidationRatio`: This is the portion of the position size we try to liquidate if the available margin is higher than the liquidation fee; it must be between 0 and 1.
  - `FundingRateEpochId`: This specifies the interval on which the funding rate is updated.
  - `TwapLookbackWindow`: This is the amount of time to look back for TWAP calculations.

Message: `ExecuteMsg.DepthShift`

This message can be sent via a CosmosWasm contract and allows for the swap invariant of an AMM pool to be updated, after making sure there is enough money in the PerpEFModule fund to pay for it. These funds get send to the vault to pay for a

trader's new net margin.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `Pair` — This is the pair of the AMM to update the peg multiplier for. It is checked to ensure that it is valid and that there is an AMM for it.
- `DepthMult` — This is the new depth multiplier to set for the AMM; it must be greater than zero.

Message: `ExecuteMsg.EditOracleParams`

This message can be sent via a CosmosWasm contract and allows for the oracle parameters to be edited.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `VotePeriod` — This is the number of blocks during which voting takes place.
- `VoteThreshold` — This is the minimum proportion of votes that must be received for a ballot to pass.
- `RewardBand` — This is a maxium divergence that a price vote can have from the weighted median in the ballot. If a vote lies within the valid range defined by `μ := weightedMedian, validRange := μ ± (μ * rewardBand / 2)`, then rewards are added to the validator performance.
- `Whitelist` — This is the set of whitelisted markets, or asset pairs, for the module — for example, `["unibi:uusd","ubtc:uusd"]`.
- `SlashFraction` — This is the proportion of an oracle's stake that gets slashed for failing a voting period.
- `SlashWindow` — This is the number of voting periods that specify a "slash window".
- `MinValidPerWindow` — This is the minimum number of valid votes per window that a validator must submit to avoid being slashed.
- `TwapLookbackWindow` — This is the amount of time to look back for TWAP calculations.
- `MinVoters` — This is the minimum number of voters (i.e., oracle validators) per pair for it to be considered a passing ballot.
- `ValidatorFeeRatio` — This is the validator fee ratio that is given to validators every epoch.

The oracle `Params` has a `Validate` method, but it does not seem to be called. Ideally, `SetOracleParams` should check this after merging the new params with the existing ones.

Message: `ExecuteMsg.InsuranceFundWithdraw`

This message can be sent via a CosmosWasm contract and allows for the insurance fund to be withdrawn from.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `Amount` — This is the amount of nUSD to withdraw from the insurance fund; it must be greater than zero.
- `To` — This is the address to send the withdrawn nUSD to; it must be a valid address.

Message: `ExecuteMsg.OpenPosition`

This message can be sent via a CosmosWasm contract and allows for a new position to be opened. The parameters that a user can control are

- `Sender` — This is the account to open a position on. There are no checks (see 5.3).
- `Pair` — This is the pair to open or modify the position on. It is checked to ensure that the format is valid, that an enabled market exists for it, and that there is an AMM for it.
- `IsLong` — This is a boolean indicating the direction the user is taking.
- `QuoteAmount` — This is the amount of quote asset to open a position with; it must not be zero.
- `Leverage` — This is the leverage to open a position with; it must not be zero and not greater than the market's `MaxLeverage` setting.
- `BaseAmountLimit` — This is the minimum base asset amount to open a position with; it must not be negative.

After the `sdkMsg` is created, there is no call to `ValidateBasic` (which automatically happens for regular Cosmos transactions), so a lot of vital checks (such as negative numbers) are missed. See the related finding at 5.4.

The flow is then the same as the regular `OpenPosition` transaction; see 7.2.

Message: `ExecuteMsg.PegShift`

This message can be sent via a CosmosWasm contract and allows for the peg multiplier of an AMM pool to be updated after making sure there is enough money in the PerpEFModule fund to pay for it. These funds get sent to the vault to pay for a trader's new net margin.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `Pair` — This is the pair of the AMM to update the peg multiplier for. It is checked to ensure that it is valid and that there is an AMM for it.
- `PegMult` — This is the new peg multiplier to set for the AMM; it must be greater than zero.

Message: `ExecuteMsg.RemoveMargin`

This message can be sent via a CosmosWasm contract and allows for a collateral to be removed from an existing position. The parameters that a user can control are

- `Sender` — This is the account holding the position; there are no checks (see 5.3).
- `Pair` — This is the pair of the position to remove margin from. It is checked to ensure that it is valid and that there is an open position, a market, and an AMM for the pair.
- `Margin` — This is the amount of collateral to remove from the position; there are no checks on the amount.

After the `sdkMsg` is created, there is no call to `ValidateBasic` (which automatically happens for regular Cosmos transactions), so a lot of vital checks (such as negative numbers) are missed. In this case, a negative margin can be specified when it ends up being added to the position margin without transferring any funds from the user; see the related finding at 5.4.

The flow is then the same as the regular `AddMargin` transaction; see 7.2.

Message: `ExecuteMsg.SetMarketEnabled`

This message can be sent via a CosmosWasm contract and allows for a market to be enabled or disabled.

The `contractAddr` address is checked to ensure that it is contained within the set of sudo contracts defined in the x/sudo module.

The parameters that a user can control are

- `Pair` — This is the pair of the market to enable or disable. It is checked to ensure that it is valid and that there is a market for it.
- `Enabled` — This is the new enabled state to set for the market.

7.4 Module: bindings-perp

This CosmosWasm module is a simple wrapper to allow other contracts (or using `nibid tx wasm execute`) to execute messages against the Nibiru perp module. There are no checks on any of the messages; they are simply passed through and handled by the custom Wasm executor (see 7.3 for more details).

The entry points are

- `ExecuteMsg::OpenPosition` — with controllable parameters `sender`, `pair`, `is_long`, `quote_amount`, `leverage`, and `base_amount_limit`
- `ExecuteMsg::ClosePosition` — with controllable parameters `sender` and `pair`
- `ExecuteMsg::AddMargin` — with controllable parameters `sender`, `pair`, and `margin`
- `ExecuteMsg::RemoveMargin` — with controllable parameters `sender`, `pair`, and `margin`
- `ExecuteMsg::MultiLiquidate` — with controllable parameters `pair` and `liquidations`
- `ExecuteMsg::DonateToInsuranceFund` — with controllable parameters `sender` and `donation`
- `ExecuteMsg::NoOp` — with no parameters

7.5 Module: controller

This CosmosWasm module is used to execute privileged messages in the Nibiru perp module (see 7.3 for more details), which can only be executed if the contract has been added to the sudo module. This module also maintains a whitelist of addresses that are allowed to execute the messages in this contract as well as the address of the current admin.

The `InsuranceFundWithdraw`, `SetMarketEnabled`, and `EditOracleParams` messages are simple wrappers that first check if the sender is a member of the whitelist, and if so, they forward the message.

The `AddMember`, `RemoveMember`, and `ChangeAdmin` messages are used to update the whitelist and admin of this module and can only be performed by the current admin. For each message the new address is first validated, then the members list is updated or the admin is changed. When the admin is changed, the new admin is removed from the

list of members.

The entry points are

- `ExecuteMsg::InsuranceFundWithdraw` — with controllable parameters `amount` and `to`
- `ExecuteMsg::SetMarketEnabled` — with controllable parameters `pair` and `enabled`
- `ExecuteMsg::EditOracleParams` — with controllable parameters `vote_period`, `vote_threshold`, `reward_band`, `whitelist`, `slash_fraction`, `slash_window`, `min_valid_per_window`, `twap_lookback_window`, `min_voters`, and `validator_fee_ratio`
- `ExecuteMsg::AddMember` — with controllable parameter `address`
- `ExecuteMsg::RemoveMember` — with controllable parameter `address`
- `ExecuteMsg::ChangeAdmin` — with controllable parameter `address`

7.6 Module: shifter

This CosmosWasm module is used to execute privileged messages in the Nibiru perp module (see 7.3 for more details), which can only be executed if the contract has been added to the sudo module. This module also maintains a whitelist of addresses that are allowed to execute the messages in this contract as well as the address of the current admin.

The `DepthShift` and `PegShift` messages are simple wrappers that first check if the sender is a member of the whitelist, and if so, they forward the message.

The `AddMember`, `RemoveMember`, and `ChangeAdmin` messages are used to update the whitelist and admin of this module and can only be performed by the current admin. For each message the new address is first validated, then the members list is updated or the admin is changed. When the admin is changed, the new admin is removed from the list of members.

The entry points are

- `ExecuteMsg::DepthShift` — with controllable parameters `pair` and `depth_mult`
- `ExecuteMsg::PegShift` — with controllable parameters `pair` and `peg_mult`
- `ExecuteMsg::AddMember` — with controllable parameter `address`
- `ExecuteMsg::RemoveMember` — with controllable parameter `address`
- `ExecuteMsg::ChangeAdmin` — with controllable parameter `address`

8 Audit Results

At the time of our audit, the audited code was not deployed to mainnet.

During our assessment on the scoped Nibiru modules, we discovered 10 findings. Four critical issues were found. Four were of high impact, one was of medium impact, and one was of low impact. Nibiru acknowledged all findings and implemented fixes. Nibiru acknowledged all findings and implemented fixes.

8.1 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 audit version of 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.

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.