Zellic

Prepared for
Constantin Guggi
Sigma Assets GmbH

Prepared by
Jisub Kim
Avraham Weinstock
Zellic

February 24, 2025

Swisstronik
Blockchain Security Assessment

Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Sigma Assets GmbH from January 27th to February 17th, 2025. During this engagement, Zellic reviewed Swisstronik's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Could an on-chain attacker bypass the signature check or other transaction checks?
- Could a bad transfer lead to the minting of new coins?
- Is it possible for a transaction not to revert in the case of failure?
- Is the behavior of the EVM module consistent with go-ethereum?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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

1.4. Results

During our assessment on the scoped Swisstronik modules, we discovered nine findings. No critical issues were found. One finding was of high impact, three were of medium impact, four were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Sigma Assets GmbH in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	3
🟩 Low	4
⬜ Informational	1

2. Introduction

2.1. About Swisstronik

Sigma Assets GmbH contributed the following description of Swisstronik:

> Swisstronik is an identity-based chain-agnostic hybrid blockchain ecosystem. It lets Web 3.0 and traditional companies launch KYC, AML and DPR-compliant tokens and applications with enhanced data privacy.

2.2. Methodology

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

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

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

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

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

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

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

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

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

2.3. Scope

The engagement involved a review of the following targets:

Swisstronik Modules

Types	Go, Rust
Platform	Cosmos

Target	Swisstronik
Repository	https://github.com/SigmaGmbH/swisstronik-chain ↗
Version	`96828917a282d15563c74bb0e081eea5a808c867`

Programs	`x/evm/*`
	`x/vesting/*`
	`x/compliance/*`
	`app/*`
	`sgxvm/src/handlers/*`
	`sgxvm/src/backend.rs`
	`sgxvm/src/precompiles/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4.8 person-weeks. The assessment was conducted by two consultants over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jisub Kim
Engineer
jisub@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 27, 2025	Kick-off call
January 27, 2025	Start of primary review period
February 17, 2025	End of primary review period

3. Detailed Findings

3.1. Low gas costs of precompiles lead to denial of service

Target	static-precompiles/src/bn128.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `Bn128Add`, `Bn128Mul`, and `Bn128Pairing` precompiles compute point addition, scalar multiplication, and pairing for the `alt_bn128` elliptic curve, as specified in EIP-196 ↗ and EIP-197 ↗. They have significantly lower gas costs than specified either directly in those EIPs or in the subsequent EIP-1108 ↗ that reduces them.

The `Blake2F` precompile, which computes a specified number of rounds of the `Blake2` compression function, as specified in EIP-152 ↗, has a constant gas cost due to the constant-sized input description, when it should have a gas cost proportional to the number of rounds.

Impact

Having a low gas cost for precompiles that perform time-consuming computation allows smart contracts to perform a denial-of-service attack on nodes.

Recommendations

Increase the gas costs of the BN128 precompiles to match those specified in EIP-1108 ↗. The cost of `Bn128Pairing` cannot be made to exactly match with `LinearCostPrecompile` due to `LinearCostPrecompile` declaring a cost of $\mathtt{WORD} * \left\lfloor \frac{b+31}{32} \right\rfloor + \mathtt{BASE}$, while the specified gas cost is $34000 * \left\lfloor \frac{b}{192} \right\rfloor + 45000$, where b is the number of input bytes; however, it can be closely approximated.

```
impl LinearCostPrecompile for Bn128Add {
    const BASE: u64 = 15;
    const WORD: u64 = 3;
    const BASE: u64 = 150;
    const WORD: u64 = 0;
```

```
impl LinearCostPrecompile for Bn128Mul {
    const BASE: u64 = 15;
```

```
        const WORD: u64 = 3;
        const BASE: u64 = 6000;
        const WORD: u64 = 0;
```

```
impl LinearCostPrecompile for Bn128Pairing {
        const BASE: u64 = 15;
        const WORD: u64 = 3;
        const BASE: u64 = 45000;
        const WORD: u64 = 34000/(192/32);
```

The `Blake2F` precompile's gas cost cannot be approximated with `LinearCostPrecompile` due to having an input-dependent gas cost.

The gas costs of the `Curve25519Add` and `Curve25519ScalarMul` precompiles should likely be increased to match the costs of `Bn128Add` and `Bn128Mul`, respectively.

The gas cost of `Ed25519Verify` should be increased to 2,000 to conform to EIP-665 ↗.

Remediation

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

- 3f9c9d0a ↗
- b0b0998b ↗
- 0ee374e6 ↗
- 3899e260 ↗
- 737d29b8 ↗

3.2. The `ECRecover` precompile computes an incorrect key

Target	static-precompiles/src/ec_recover.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

For the go-ethereum test case `ValidKey`, Swisstronik's implementation of `ECRecover` returns a zero-length output instead of the expected value. This is due to the presence of the check that the signature's `s` value is no more than half the Secp256k1 group order. EIP-2 ↗, which specifies that signatures with an `s` value above half the group order are invalid, also specifies "the ECDSA recover precompiled contract remains unchanged and will keep accepting high s-values; this is useful e.g. if a contract recovers old Bitcoin signatures".

The `ValidKey` test case from go-ethereum is as follows and is run using the test described in section 4.2. ↗:

```
  {
    "Input":
    "18c547e4f7b0f325ad1e56f57e26c745b09a3e503d86e00e5255ff7f715d3d1c00
000000000000000000000000000000000000000000000000000000000000001c73b1693892219d736c
aba55bdb67216e485557ea6b6af75f37096c9aa6a5a75feeb940b1d03b21e36b0e47e79769f095fe
2ab855bd91e3a38756b7d75a9c4549",
    "Expected":
    "000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b",
    "Gas": 3000,
    "Name": "ValidKey",
    "NoBenchmark": false
  },
```

Impact

Contracts that make use of `ECRecover` will fail to accept signatures that they would accept on other EVM implementations.

Recommendations

Remove the check that `s > secp256k1n` from the `ECRecover` precompile:

```
        if signature.s().is_high().into() {
            return (ExitSucceed::Returned.into(), [0u8; 0].to_vec());
        }
```

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit 9e33b5a7 ↗.

3.3. Missing overflow check in `AddTransientGasWanted`

Target	x/feemarket/keeper/keeper.go		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `AddTransientGasWanted` function accumulates the gas into a `uint64` without performing an overflow check. If the sum exceeds the maximum value of `uint64`, it will wrap around to zero, leading to incorrect or unpredictable gas accounting.

```go
// AddTransientGasWanted adds the cumulative gas wanted in the transient store
func (k Keeper) AddTransientGasWanted(ctx sdk.Context, gasWanted uint64)
    (uint64, error) {
!  result := k.GetTransientGasWanted(ctx) + gasWanted
  k.SetTransientBlockGasWanted(ctx, result)
  return result, nil
}
```

Impact

An overflow could reset or distort the block's cumulative gas usage, leading to invalid gas calculations.

Recommendations

Check to ensure the sum does not exceed `math.MaxUint64`.

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit 339e3a92 ↗.

3.4. Lack of validation in `ElasticityMultiplier` causes division by zero

Target	x/feemarket/keeper/eip1559.go		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In `CalculateBaseFee`, the code assumes that `ElasticityMultiplier` cannot be zero due to `validation`, but `validateElasticityMultiplier` only checks for the correct data type (`uint32`) and not whether the value is nonzero.

```
func validateElasticityMultiplier(i interface{}) error {
    _, ok := i.(uint32)
    if !ok {
        return fmt.Errorf("invalid parameter type: %T", i)
    }
    return nil
}
```

If `ElasticityMultiplier` is mistakenly set to zero, division by zero can occur in the base-fee calculation.

```
func (k Keeper) CalculateBaseFee(ctx sdk.Context) *big.Int {
    params := k.GetParams(ctx)
  // [...]

  // CONTRACT: ElasticityMultiplier cannot be 0 as it's checked in the params
  // validation
  parentGasTargetBig := new(big.Int).Div(gasLimit,
    new(big.Int).SetUint64(uint64(params.ElasticityMultiplier)))
  if !parentGasTargetBig.IsUint64() {
    return nil
  }
```

Impact

Division by zero can cause runtime errors or panics, disrupting block processing.

Recommendations

Extend `validateElasticityMultiplier` to ensure `ElasticityMultiplier` is greater than zero — for example:

```
if val, ok := i.(uint32); !ok || val == 0 {
    return fmt.Errorf("elasticity multiplier must be a non-zero uint32")
}
```

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit 40c518a9 ↗.

3.5. Unbounded `originalData` can be provided

Target	x/compliance/keeper/keeper.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `addVerificationDetailsInternal` does not restrict the maximum size of `OriginalData` in the `VerificationDetails` struct. Currently, only a check is performed to ensure that `OriginalData` is nonempty, and no upper bound is enforced. Attackers could potentially supply extremely large data (e.g., close to the maximum transaction size of ~1 MB) to cause elevated memory usage on nodes.

Although the system's default mempool and gas limits (e.g., `max_tx_bytes = 1048576`) do mitigate excessively large transactions, repeated submissions of near–maximal-size transactions can still spam transactions with dummy data.

Impact

Attackers with sufficient resources could push repeated large transactions, potentially making nodes unresponsive or causing them to drop legitimate transactions due to limited block space.

Recommendations

Add a limit for the original data and original chain, such as

```
if len(details.OriginalData) > MaxOriginalDataSize {
    return nil, errors.Wrap(types.ErrInvalidParam, "verification data exceeds
    maximum allowed size")
}
```

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit 5bdea439 ↗.

3.6. The `EvmDenom` can be updated by the EVM module's `MsgUpdateParams`

Target	x/evm/keeper/msg_server.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `EvmDenom` parameter, which determines which Cosmos token is used as the EVM's value, is updatable through the EVM module's `MsgUpdateParams` message.

Impact

If smart contracts (e.g., vaults) keep track of balances denominated in the native EVM value, they have no way to observe the parameter update in order to invalidate the balance, resulting in incorrect behavior (e.g., incorrect values withdrawn from vaults) as balances accrued in the old token are interpreted according to the new token. However, since `MsgUpdateParams` can only be called through the Governance module, this could only be exploited with enough Governance votes to pass proposals to issue `MsgUpdateParams` messages with a modified `EvmDenom`.

Recommendations

Disallow updating the `EvmDenom` parameter.

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit `7bd13cd0` ↗.

3.7. Potential division by zero in `fee_checker`

Target	app/ante/fee_checker.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

A divide-by-zero error may occur if the gas is zero when calculating the fee cap:

```
feeCap := fee.Quo(sdkmath.NewIntFromUint64(gas))
```

Without a check ensuring `gas > 0`, runtime errors could disrupt transaction processing.

Impact

The node could reject or fail to process transactions with zero gas.

Recommendations

Add a check to ensure the gas is nonzero in the validation logic.

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit `1f238cb8` ↗.

3.8. Potential overflow in `fee_checker`

Target	x/feemarket/types/msg.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `ValidateBasic` function currently lacks an upper bound for gas, allowing values that could exceed `math.MaxInt64`:

```
// ValidateBasic does a sanity check of the provided data
func (m *MsgUpdateParams) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(m.Authority); err != nil {
        return errortypes.Wrap(err, "invalid authority address")
    }

    return m.Params.Validate()
}
```

Impact

Extremely large gas values may lead to unexpected behaviors or overflows in later calculations.

Recommendations

Add a check to ensure the gas cannot exceed `math.MaxInt64` in the validation logic.

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit `e727fd5e` ↗.

3.9. No-cliff vesting is not supported

Target	x/vesting/types/vesting_account.go		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Currently, the `Validate()` method imposes `m.GetStartTime() >= m.GetCliffTime()` as an error condition:

```
if m.GetStartTime() >= m.GetCliffTime() {
    return errors.New("vesting start-time cannot be after cliff-time")
}
```

The current validation logic prevents a legitimate vesting use case — no-cliff vesting. In other words, the protocol cannot have an immediate vesting account if the cliff time is strictly required to be greater than the start time.

Impact

The protocol requiring immediate (no-cliff) vesting cannot declare it in a straightforward manner. However, since time units are in seconds, setting `CliffTime = StartTime + 1` would archive nearly the same effect in practice.

Recommendations

Change the condition to `m.GetStartTime() > m.GetCliffTime()`, which would support zero-cliff vesting.

Remediation

This issue has been acknowledged by Sigma Assets GmbH, and a fix was implemented in commit 956b84b0 ↗.

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. Installation steps

The easiest way to compile the chain is to configure BIOS, install Intel SGX SDK, install Intel SGX PSW, update the environment variables, and then build the chain.

Configure BIOS

This first step involves the following.

- Enable software guard extensions (SGX)
- Disable Secure Boot
- Disable hyper-threading
- Disable Turbo Mode
- Enable CPU AES

After adjusting BIOS settings, ensure you save and reboot your machine for the changes to take effect.

Install Intel SGX SDK

Complete the following steps for installation of Intel SGX SDK.

1. **Download the SGX SDK installer.** You can obtain the installer from this repository ↗. Adjust the filename if necessary (e.g., sgx_linux_x64_sdk_${version}.bin).

2. **Make the installer executable.** Run the following command.

```
chmod +x sgx_linux_x64_sdk_${version}.bin
```

3. **Install the SGX SDK.** We recommend installing it to `/opt/intel/`. Use the `--prefix` option for a custom installation directory and follow the on-screen instructions to complete the installation.

```
./sgx_linux_x64_sdk_${version}.bin --prefix /opt/intel
```

Install Intel SGX PSW

Complete the following steps for installation of Intel SGX PSW (platform software).

1. Add Intel's SGX repository, as below.

```
echo "deb https://download.01.org/intel-sgx/sgx_repo/ubuntu $(lsb_release -cs)
    main" | sudo tee /etc/apt/sources.list.d/intel-sgx.list >/dev/null
curl -sSL "https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-
    deb.key" | sudo -E apt-key add -
sudo apt update
```

2. Install core PSW packages.

```
sudo apt install libsgx-launch libsgx-urts libsgx-epid libsgx-quote-ex \
    sgx-aesm-service libsgx-aesm-launch-plugin libsgx-aesm-epid-plugin \
    libsgx-quote-ex libsgx-dcap-ql libsnappy1v5 libsgx-dcap-quote-verify
    libsgx-dcap-default-qpl
```

Update environment variables

This involves the following.

1. Source the SGX SDK environment. This command updates `LD_LIBRARY_PATH` and other necessary variables.

```
source /opt/intel/sgxsdk/environment
```

2. Add the libdcap_quoteprov.so symbolic link (for `quoteprov`) and update the `LD_LIBRARY_PATH` again.

```
sudo ln -s /usr/lib/x86_64-linux-gnu/libdcap_quoteprov.so.1 /usr/lib/x86_64-
    linux-gnu/libdcap_quoteprov.so
sudo ldconfig
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/lib/x86_64-linux-gnu/"
```

Build the chain

With the SGX SDK and PSW properly installed and configured, the project should now be able to be built:

```
make build
```

Ensure this command is run from within the project's directory or wherever the build scripts are located.

4.2. The go-ethereum precompile tests

The following test, to be added to the static-precompiles crate, adapts test cases from go-ethereum's test suite ↗.

```rust
use evm::{
    interpreter::{
        error::ExitSucceed,
        runtime::{Context, RuntimeState, TransactionContext},
    },
    standard::{Config, GasometerState, State},
};
use primitive_types::{H160, U256};
use serde::Deserialize;
use std::rc::Rc;

use crate::{
    blake2f::Blake2F,
    bn128::{Bn128Add, Bn128Mul, Bn128Pairing},
    ec_recover::ECRecover,
    modexp::Modexp,
    Precompile,
};

#[derive(Deserialize)]
#[allow(dead_code, non_snake_case)]
struct GethPrecompileTestCase<'a> {
    Input: &'a [u8],
    Expected: &'a [u8],
    Name: &'a [u8],
    Gas: u64,
    NoBenchmark: bool,
}
```

```
#[derive(Deserialize)]
#[allow(dead_code, non_snake_case)]
struct GethPrecompileTestCaseFail<'a> {
    Input: &'a [u8],
    ExpectedError: &'a [u8],
    Name: &'a [u8],
}

fn test_state<'config>(config: &'config Config) -> State<'config> {
    let caller = H160::zero();
    let address = H160::zero();
    let value = U256::zero();
    let gas_price = U256::one();
    let context = Context {
        caller,
        address,
        apparent_value: value,
    };
    let transaction_context = TransactionContext {
        origin: caller,
        gas_price,
    };
    let runtime_state = RuntimeState {
        context,
        transaction_context: Rc::new(transaction_context),
        retbuf: Vec::new(),
    };
    State {
        runtime: runtime_state,
        gasometer: GasometerState::new(u64::MAX, false, &config),
    }
}

macro_rules! precompile_test {
    ($precompile:ident, $test_name:ident, $success:expr $(, $fail:expr )?) =>
    {
        #[test]
        fn $test_name() {
            let testcases: Vec<GethPrecompileTestCase<'static>> =
                serde_json::from_slice(include_bytes!($success)).unwrap();
            for testcase in testcases {
                let input = hex::decode(&testcase.Input).unwrap();
                let expected = hex::decode(&testcase.Expected).unwrap();
                let config = Config::cancun();
                let mut state = test_state(&config);
                println!("{}", String::from_utf8_lossy(testcase.Name));
                let (result, output) = $precompile::execute(&input,
    &mut state);
                assert_eq!(result, Ok(ExitSucceed::Returned));
                assert_eq!(output, expected);
                if TEST_GAS {
                    assert_eq!(u64::MAX - state.gasometer.gas64(),
    testcase.Gas);
                }
            }
            $(
                let testcases: Vec<GethPrecompileTestCaseFail<'static>> =
                    serde_json::from_slice(include_bytes!($fail)).unwrap();
                for testcase in testcases {
                    let input = hex::decode(&testcase.Input).unwrap();
                    let config = Config::cancun();
                    let mut state = test_state(&config);
                    println!("{}", String::from_utf8_lossy(testcase.Name));
                    let (result, output) = $precompile::execute(&input,
    &mut state);
                    assert!(result.is_err());
                    assert!(output.is_empty());
            }
            )?
        }
    };
}

const TEST_GAS: bool = false;

precompile_test!(
    Blake2F,
    test_blake2f,
    "testdata/blake2F.json",
    "testdata/fail-blake2f.json"
);
precompile_test!(Bn128Add, test_bn256add, "testdata/bn256Add.json");
precompile_test!(Bn128Mul, test_bn256mul, "testdata/bn256ScalarMul.json");
precompile_test!(
    Bn128Pairing,
    test_bn256pairing,
    "testdata/bn256Pairing.json"
);
precompile_test!(ECRecover, test_ecrecover, "testdata/ecRecover.json");
precompile_test!(Modexp, test_modexp, "testdata/modexp_eip2565.json");
```

5. System Design

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

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

5.1. Component: EVM

Description

The EVM module runs Ethereum-compatible transactions inside an SGX enclave. The transactions' calldata may optionally be encrypted towards a particular node. The non-calldata fields of the transactions, including the sender, recipient, and native value, are unencrypted. The native value is represented as Cosmos bank tokens with the `"aswtr"` denomination by default, though `MsgUpdateParams` can change the denomination; see Finding 3.6. ↗.

Invariants

EVM native value corresponds to the Cosmos value of the EvmDenom token.

Test coverage

Cases covered

- The construction of the various kinds of `MsgHandleTx` (`AccessListTx`, `DynamicFeeTx`, and `LegacyTx`) are tested, with both valid and invalid fields.
- The various kinds of signers for `MsgHandleTx` are tested.
- Execution of well-formed, signed `MsgHandleTx`s are integration-tested, including with both valid and invalid gas prices.
- Submitting transactions that transfer value to the SGXVM independently of `MsgHandleTx` are unit-tested, including both success and failure cases.
- Most of the precompiles have unit tests.
- Most of the SGX Connector requests have unit tests.
- Many of the internal functions are unit-tested, though some of these (e.g., x/evm/keeper/params_test.go) only test success cases.

Cases not covered

- Several of the precompiles do not have unit tests. The go-ethereum's unit tests can be used to increase coverage for these precompiles; see section 4.2. ↗.
- The SGX Connector requests that `GetAccountCodeHash`, `GetAccountCodeSize`, `GetAccountStorageCell`, `InsertStorageCell`, and `RemoveStorageCell` do not have unit tests.

Attack surface (Cosmos messages)

Message: `MsgHandleTx`

The `MsgHandleTx` message executes Ethereum transactions.

Its `ValidateBasic` method checks the following properties:

- Its `From` address, if nonempty, is a valid Ethereum address.
- Its `Data` unmarshals as an instance of the `TxData` interface, of which `AccessListTx`, `DynamicFeeTx`, and `LegacyTx` are the only implementations. Each `TxData` instance's `Validate` method checks that the gas prices and fees are nonnegative and do not exceed 2^{256}.
- Its gas requirement is nonzero.
- The transaction's declared hash matches its recomputed hash.

Several AnteHandlers that are part of `NewCosmosAnteHandler` ensure that `MsgHandleTx` cannot occur in a normal Cosmos transaction. This includes `RejectMessagesDecorator`, which ensures that `MsgHandleTx` does not occur directly in a transaction outside of `ExtensionOptionsEthereumTx`. This also includes `RejectNestedMessageDecorator`, which ensures that `MsgHandleTx` does not occur inside an `authz.MsgExec` message. Since Swisstronik does not integrate wasmd, `MsgHandleTx` cannot be indirectly dispatched by CosmWasm contracts.

The top-level AnteHandler uses an Ethereum-specific `NewEthAnteHandler` for transactions whose first extension option is `ExtensionOptionsEthereumTx`. The AnteHandlers in `NewEthAnteHandler` ensure that the only messages in such a transaction are of type `MsgHandleTx`. They check, among other things, that the transactions are signed and the sender has enough balance for the transaction; the details are covered in section 5.2. ↗.

The handler for `MsgHandleTx` runs the provided Ethereum transaction in the SGX enclave, which is provided a connector allowing the enclave to write to a copy of the state, which is committed if the transaction is successful.

Message: `MsgUpdateParams`

The `MsgUpdateParams` message updates the EVM module's parameters, which consist of which Cosmos token denomination is used as the EVM's native token, whether the `CREATE` and `CALL` opcodes are enabled, and various configuration options for the EVM (primarily which block heights various EIP-specified features are enabled by).

Its `ValidateBasic` method ensures that all the fields are well-formed for their types. Its handler ensures that the signer of the message is the `Authority` that the module's keeper was instantiated with, which is currently the Governance module, ensuring that this message is only issued by governance proposals.

Attack surface (EVM precompiles)

Precompile: `ECRecover`

The `ECRecover` precompile computes which public key signed a message hash, given the message hash and (v, r, s) components of an ECDSA signature with a recovery ID. It costs 3,000 gas, which is consistent with go-ethereum's gas cost. However, the implementation does not compute the same key as go-ethereum's test cases; see Finding 3.2. ↗.

Precompile: `Sha256`

The `Sha256` precompile calculates the SHA-256 hash of its input, costing `60 + 12w` gas, where `w` is the number of 32-byte words in the input, consistent with go-ethereum's gas cost.

Precompile: `Ripemd160`

The `Ripemd160` precompile calculates the RIPEMD-160 hash of its input, costing `600 + 120w` gas, where `w` is the number of 32-byte words in the input, consistent with go-ethereum's gas cost.

Precompile: `DataCopy`

The `DataCopy` precompile copies its input to its output, costing `15 + 3w` gas, where `w` is the number of 32-byte words in the input, consistent with go-ethereum's gas cost.

Precompile: `Modexp`

The `Modexp` precompile computes $b^e \bmod m$, where (b, e, m) are variable-sized unsigned integers given as length-prefixed big-endian bytestrings in the input. Its gas cost is calculated according to EIP-2565 ↗ and passes go-ethereum's tests; see section 4.2. ↗.

Precompile: `Bn128Add`

The `Bn128Add` precompile computes point addition for the `alt_bn128` elliptic curve, as specified in EIP-196 ↗. Its gas cost is inconsistent with go-ethereum and EIP-1108 ↗; see Finding 3.1. ↗.

Precompile: `Bn128Mul`

The `Bn128Mul` precompile computes scalar multiplication for the `alt_bn128` elliptic curve, as specified in EIP-196 ↗. Its gas cost is inconsistent with go-ethereum and EIP-1108 ↗; see Finding 3.1. ↗.

Precompile: `Bn128Pairing`

The `Bn128Pairing` precompile computes pairings for the `alt_bn128` elliptic curve, as specified in EIP-197 ↗. Its gas cost is inconsistent with go-ethereum and EIP-1108 ↗; see Finding 3.1. ↗.

Precompile: `Blake2F`

The `Blake2F` precompile computes the `Blake2` compression function with the parameters specified in EIP-152 ↗, costing `15 + 3w` gas, where `w` is the number of 32-byte words in the input, which is always $\left\lfloor \frac{213+31}{32} \right\rfloor = 7$. This cost is inconsistent with the specified cost, which depends on the number of rounds; see Finding 3.1. ↗.

Precompile: `P256Verify`

The `P256Verify` precompile verifies ECDSA signatures over the Secp256r1 elliptic curve, as specified in RIP-7212 ↗. Its gas cost is a constant 3,500, which matches the specification.

Precompile: `Sha3FIPS256`

The `Sha3FIPS256` precompile calculates the SHA3-256 hash of its input, costing `60 + 12w` gas, where `w` is the number of 32-byte words in the input. This gas cost does not conform to a specification but is unlikely to lead to denial of service.

Precompile: `Sha3FIPS512`

The `Sha3FIPS512` precompile calculates the SHA3-512 hash of its input, costing `60 + 12w` gas, where `w` is the number of 32-byte words in the input. This gas cost does not conform to a specification but is unlikely to lead to denial of service.

Precompile: `ComplianceBridge`

The `ComplianceBridge` precompile allows smart contracts to call the `RevocationTreeRoot`, `IssuanceTreeRoot`, `HasVerification`, `AddVerificationDetails`, `AddVerificationDetailsV2`, and `GetVerificationData` functions through the connector object, as described in section 5.1. ↗. It costs `60 + 150w` gas, where `w` is the number of 32-byte words in the input, which is a protobuf-encoded request that is parsed outside the enclave.

Precompile: `Curve25519Add`

The `Curve25519Add` precompile computes point addition for the Curve25519 elliptic curve. While its gas cost of `60 + 12w` is not given by a specification, it should likely be increased to match

`Bn128Add`'s gas cost; see Finding 3.1. ↗.

Precompile: `Curve25519ScalarMul`

The `Curve25519ScalarMul` precompile computes scalar multiplication for the Curve25519 elliptic curve. While its gas cost of `60 + 12w` is not given by a specification, it should likely be increased to match `Bn128Mul`'s gas cost; see Finding 3.1. ↗.

Precompile: `Ed25519Verify`

The `Ed25519Verify` precompile verifies EdDSA signatures over the Curve25519 elliptic curve, as specified in EIP-665 ↗. Its gas cost is inconsistent with the specification; see Finding 3.1. ↗.

Attack surface (SGX Connector)

Request: `GetAccount`

The `GetAccount` request retreives the account associated with the provided Ethereum address, consisting of its balance and current transaction nonce. The balance and nonce are zero if the address does not exist in the store. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `balance` and `nonce` methods.

Request: `ContainsKey`

The `ContainsKey` request checks whether an account exists for the provided Ethereum address. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `exists` method.

Request: `GetAccountCode`

The `GetAccountCode` request returns the bytecode associated with the provided Ethereum address or an empty bytestring if the account does not exist or is not a contract. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `code` method.

Request: `GetAccountCodeHash`

The `GetAccountCodeHash` request returns the Keccack-256 hash of the bytecode associated with the provided Ethereum address, which is the hash of the empty bytestring for noncontract or nonexistent accounts. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `code_hash` method.

Request: `GetAccountCodeSize`

The `GetAccountCodeSize` request returns the length in bytes of the value that would be returned by the `GetAccountCode` request. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `code_size` method.

Request: `GetAccountStorageCell`

The `GetAccountStorageCell` request returns the byte string stored at the specified storage location, determined by an address and index pair. It is called by the EVM implementation through the `RuntimeBaseBackend` trait's `storage` method.

Request: `InsertAccountCode`

The `InsertAccountCode` request sets the code associated with the specified Ethereum address to the provided value. It is called as part of `Backend::apply_changeset` when an EVM transaction's changes should be committed after execution.

Request: `InsertStorageCell`

The `InsertStorageCell` request sets the specified storage location to the provided value. It is called as part of `Backend::apply_changeset` when an EVM transaction's changes should be committed after execution.

Request: `Remove`

The `Remove` request removes the account for the provided Ethereum address. It is called as part of `Backend::apply_changeset` when an EVM transaction's changes should be committed after execution.

Request: `RemoveStorageCell`

The `RemoveStorageCell` request deletes the value at the specified storage location. A handler for it is defined in the host, but the function that encodes the request is not currently called by the EVM implementation in the enclave.

Request: `RemoveStorage`

A `RemoveStorage` request is specified in the protobuf schema used to communicate between the enclave and the host, but no handler for it is defined.

Request: `BlockHash`

The `BlockHash` request returns the hash of the block at the specified height. It is called by the EVM implementation through the `RuntimeEnvironment` trait's `block_hash` method.

Request: `AddVerificationDetails`

The `AddVerificationDetails` request calls the Compliance module's `AddVerificationDetails` method, returning the added verification ID if successful. It is callable by smart contracts with arbitrary parameters through the `ComplianceBridge` precompile.

Request: `HasVerification`

The `HasVerification` request returns whether the specified user has a nonexpired verification of the specified type from a specified set of issuers stored with the Compliance module. It is callable by smart contracts with arbitrary parameters through the `ComplianceBridge` precompile.

Request: `GetVerificationData`

The `GetVerificationData` request returns the verification details for the specified user and issuer stored with the Compliance module. It is callable by smart contracts with arbitrary parameters through the `ComplianceBridge` precompile.

Request: `InsertAccountBalance`

The `InsertAccountBalance` request sets the balance for the account with the provided Ethereum address. It is called as part of `Backend::apply_changeset` when an EVM transaction's changes should be committed after execution.

Request: `InsertAccountNonce`

The `InsertAccountNonce` request sets the nonce for the account with the provided Ethereum address. It is called as part of `Backend::apply_changeset` when an EVM transaction's changes should be committed after execution.

Request: `IssuanceTreeRoot`

The `IssuanceTreeRoot` request returns the root of the Merkle tree of issued credentials stored by the Compliance module, which can be used to verify inclusion proofs returned by the Compliance module's `GetIssuanceProof` query. Credentials that have been issued are present as keys in the issuance tree with a zero value. It is callable by smart contracts through the `ComplianceBridge` precompile.

Request: `RevocationTreeRoot`

The `RevocationTreeRoot` request returns the root of the Merkle tree of revoked credentials stored by the Compliance module, which can be used to verify inclusion proofs returned by the Compliance module's `GetNonRevocationProof` query. Credentials that have been issued but not revoked are present as keys in the revocation tree with a zero value; credentials that have been revoked have their hash as their value in the revocation tree. It is callable by smart contracts through the `ComplianceBridge` precompile.

Request: `AddVerificationDetailsV2`

The `AddVerificationDetailsV2` request calls the Compliance module's `AddVerificationDetailsV2` method, returning the added verification ID if successful. It is callable by smart contracts with arbitrary parameters through the `ComplianceBridge` precompile.

5.2. Component: AnteHandlers

Description

The AnteHandlers perform per-transaction validation, with a view of all the messages in the transaction. The outermost AnteHandler ensures that, if extension options are present in the transaction, the first one is either `ExtensionOptionsEthereumTx` or `ExtensionOptionDynamicFeeTx`. If the former, the `EthAnteHandler` chain of AnteHandlers is used to process the transaction as a set of Ethereum transactions. If the first extension option is `ExtensionOptionDynamicFeeTx`, or if no extension options are present, it uses the `CosmosAnteHandler`.

Invariants

The EVM module's `MsgHandleTx` is only accepted in transactions with the `ExtensionOptionsEthereumTx`.

Test coverage

Cases covered

- The majority of AnteHandlers have test coverage, testing both messages that succeed and messages that fail.

Cases not covered

- `EthEmitEventDecorator` has no test coverage.
- `EthValidateBasicDecorator` has no test coverage; however, `MsgHandleTx`'s `ValidateBasic` is tested separately in the EVM module's tests.
- `EthVestingTransactionDecorator` has no test coverage.
- `RejectMessagesDecorator` has no test coverage; however, `RejectMessagesDecorator` does.

Attack surface (shared)

AnteHandler: `GasWantedDecorator`

The `GasWantedDecorator` is included in both the `EthAnteHandler` and `CosmosAnteHandler` chains of AnteHandlers. It does not check anything for transactions that do not implement the `FeeTx` interface or if the current block height is before the configured height for enabling the London hard fork (though the default height for this is 0 in `DefaultChainConfig`). It checks that gas required by the transaction does not exceed the Cosmos block gas limit, and if the fee market keeper's base fee is enabled, it notifies the fee market keeper to deduct the required amount of gas.

Attack surface (EthAnteHandler)

AnteHandler: `EthSetUpContextDecorator`

The `EthSetUpContextDecorator` rejects transactions that do not implement the `GasTx` interface, initializes the Cosmos SDK context's gas meter, and resets the EVM module's transient gas usage.

AnteHandler: `EthMempoolFeeDecorator`

The `EthMempoolFeeDecorator` performs checks only when in `CheckTx` mode (i.e., when transactions are entering the mempool) and only if the London hard fork is disabled. It rejects transactions containing any messages other than `MsgHandleTx` and ensures that each transaction's fee meets or exceeds the fee computed based on the Cosmos SDK's gas price in the `EVMDenom`.

AnteHandler: `EthMinGasPriceDecorator`

The `EthMinGasPriceDecorator` rejects transactions containing any messages other than `MsgHandleTx`. It ensures that each transaction's fee meets or exceeds the fee computed based on the fee keeper's minimum gas price.

AnteHandler: `EthValidateBasicDecorator`

The `EthValidateBasicDecorator` ensures that the transaction's `ValidateBasic` method succeeds; that it implements the `protoTxProvider` interface; for the transaction returned by `GetProtoTx`, that its optional fields are empty or minimal; and that the sum of the fees and gas limits inside the transaction's `MsgHandleTx` messages match the outer transaction's fee and gas limit.

AnteHandler: `EthSigVerificationDecorator`

The `EthSigVerificationDecorator` rejects transactions containing any messages other than `MsgHandleTx`. If the EVM module's `AllowUnprotectedTxs` is false, it checks that transactions' signatures have EIP-155 replay protection and populates the transactions' `From` fields with the pubkey recovered from the transactions' signatures, in the process verifying the signatures.

AnteHandler: `EthAccountVerificationDecorator`

The `EthAccountVerificationDecorator` performs checks only when in `CheckTx` mode. It rejects transactions containing any messages other than `MsgHandleTx`. It initializes the sender's account with the account keeper if it does not exist, rejects the transaction if the sender has contract code stored with its address in the EVM keeper, and requires that the sender's account has sufficient balance for the transaction's cost.

AnteHandler: `CanTransferDecorator`

The `CanTransferDecorator` rejects transactions containing any messages other than `MsgHandleTx`. It ensures that the signer of the outer Cosmos transaction has sufficient balance of the `EVMDenom` to cover the transaction's value. If the London hard fork is enabled, it also checks that the transaction's `GasFeeCap` exceeds the base fee.

AnteHandler: `EthVestingTransactionDecorator`

The `EthVestingTransactionDecorator` rejects transactions containing any messages other than `MsgHandleTx`. For each message signed by a vesting account, it adds the vested portion of that account's balance and ensures that the sum exceeds the total value spent by that vesting account across the messages in the transaction.

AnteHandler: `EthGasConsumeDecorator`

The `EthGasConsumeDecorator` rejects transactions containing any messages other than `MsgHandleTx`. In `ReCheckTx` mode, it sets the Cosmos SDK context's gas meter's limit to 0 and performs no further checks. Otherwise, it computes the transaction's requested gas, checks that its fee is sufficient to cover the intrinsic gas cost, deducts the gas cost from signer's balance with the EVM keeper, and checks that the total requested gas does not exceed the EVM block gas limit.

AnteHandler: `EthIncrementSenderSequenceDecorator`

The `EthIncrementSenderSequenceDecorator` rejects transactions containing any messages other than `MsgHandleTx`. It checks that each message's sequence number matches the nonce stored in the account keeper, incrementing it. If multiple EVM messages from the same signer are present in the Cosmos transaction, they must be ordered by sequence number within the transaction.

AnteHandler: `EthEmitEventDecorator`

The `EthEmitEventDecorator` rejects transactions containing any messages other than `MsgHandleTx`. It emits a Cosmos event containing the Ethereum transaction hash of each EVM message within the transaction.

Attack surface (CosmosAnteHandler)

AnteHandler: `RejectMessagesDecorator`

The `RejectMessagesDecorator` prevents `MsgHandleTx` occurring directly in a Cosmos transaction from being processed by the CosmosAnteHandler chain.

AnteHandler: `RejectNestedMessageDecorator`

The `RejectNestedMessageDecorator` prevents a set of messages specified by type URL from occuring inside `authz.MsgExec` or `authz.MsgGrant` messages, including recursively inside of `authz.MsgExec` messages, and rejects any messages with more than five layers of nested `authz.MsgExec` messages. The top-level AnteHandler configures it to reject both `MsgHandleTx` as well as `MsgUndelegate` messages.

AnteHandler: `MinGasPriceDecorator`

The `MinGasPriceDecorator` rejects transactions that do not implement the `FeeTx` interface. If the fee keeper's minimum gas price is nonzero, it checks that the transaction's fee is sufficient for the required amount of gas at that price.

AnteHandlers: Default x/ante and ibc-go

The `SetUpContextDecorator`, `ExtensionOptionsDecorator`, `ValidateBasicDecorator`, `TxTimeoutHeightDecorator`, `ValidateMemoDecorator`, `ConsumeGasForTxSizeDecorator`, `DeductFeeDecorator`, `SetPubKeyDecorator`, `ValidateSigCountDecorator`, `SigGasConsumeDecorator`, `SigVerificationDecorator`, and `IncrementSequenceDecorator` AnteHandlers that are part of Cosmos SDK's default AnteHandler chain are in the same order as in Cosmos SDK's `NewAnteHandler`. Additionally, ibc-go's `NewRedundantRelayDecorator` is included.

5.3. Component: Compliance

Description

The Compliance module manages all aspects of identity verification and regulatory compliance within the system. It is responsible for storing and updating details related to issuers, operators, and user addresses and for maintaining records of verifications.

Invariants

The module maintains strong invariants by storing issuer, operator, and verification details under dedicated key prefixes in the Cosmos SDK KVStore. It ensures that only verified issuers may add verification records by cross-checking an issuer's status before accepting new data.

If an issuer is later removed from the system, its associated verification details are ignored in subsequent lookups, thereby preserving the integrity of the compliance data.

Test coverage

Cases covered

- Handling of the messages are unit-tested, with both success and failure cases.
- Insertion into the issuance and revocation Merkle trees are tested.
- Generation of Merkle proofs is tested for present keys.
- Queries are tested with both success and failure cases.

Cases not covered

- Failure to generate Merkle proofs for absent keys is not tested.

Attack surface (Cosmos messages)

Message: `MsgAddOperator`

The `MsgAddOperator` message registers a new operator within the compliance system. Its `ValidateBasic` method checks that both the signer and the new operator addresses are valid Bech32 addresses, ensuring that malformed or unauthorized addresses cannot be inserted.

Message: `MsgRemoveOperator`

The `MsgRemoveOperator` message is used to remove an operator. It validates the signer and the operator addresses and ensures that only regular operators may be removed, preventing accidental deletion of critical operator accounts.

Message: `MsgSetVerificationStatus`

The `MsgSetVerificationStatus` message allows an operator to update an issuer's verification status. Its basic validation ensures the correctness of the signer and issuer addresses, guarding against unauthorized status changes that could enable unverified issuers to operate.

Message: `MsgCreateIssuer`

The `MsgCreateIssuer` message enables the registration of a new issuer with detailed metadata, including name, description, URL, logo, and legal-entity information. The message validates that both the creator's and issuer's addresses are properly formatted, ensuring only authorized entities can register as issuers.

Message: `MsgUpdateIssuerDetails`

The `MsgUpdateIssuerDetails` message permits updating the details of an existing issuer. Its validation routine confirms the format of the signer and issuer addresses, protecting the system from inconsistent or corrupted issuer information.

Message: `MsgRemoveIssuer`

The `MsgRemoveIssuer` message removes an issuer from the system. It validates the signer and issuer addresses before proceeding, but it leaves previously recorded verification data intact for historical reference.

Message: `MsgRevokeVerification`

The `MsgRevokeVerification` message revokes an existing verification record. Its `ValidateBasic` method ensures that the signer's address is valid and that a non-nil verification ID is provided, preventing inadvertent or malicious revocations.

Message: `MsgAttachHolderPublicKey`

The `MsgAttachHolderPublicKey` message attaches a compressed public key to a user's address. Its validation confirms that the signer's address is valid and that the provided public key can be parsed correctly by extracting its x-coordinate, thereby preventing the association of invalid keys.

Message: `MsgConvertCredential`

The `MsgConvertCredential` message converts a verification record into a credential for use in proof systems. Its validation requires that both the signer's address and the verification ID are valid, ensuring that only legitimate records are transformed.

Attack surface (Compliance)

Function: `NewKeeper`

The `NewKeeper` function makes a new keeper by setting the primary and memory store keys along with the parameter. It ensures that the parameter subspace is correctly initialized with a key table. Although its logic is straightforward, any misconfiguration of the key table or store keys could lead to improper parameter handling across the module.

Function: `SetIssuerDetails`

The `SetIssuerDetails` function marshals the provided issuer details into bytes and writes them into the store under a dedicated key prefix. It must successfully serialize protocol buffer messages; any failure in marshaling or misuse of store keys might cause issuer data corruption or denial of service. The integrity of issuer registration depends on this function.

Function: `RemoveIssuer`

The `RemoveIssuer` function deletes an issuer's details and its associated address record from the store. For now, it leaves existing verification data intact to be filtered out later. An attacker triggering removal may cause orphaned verification records; however, the system ensures they are ignored in subsequent lookups.

Function: `GetIssuerDetails`

The `GetIssuerDetails` function retrieves issuer details from the store and unmarshals them. It returns an empty issuer structure if no data is found. Failure in deserialization or manipulation of stored bytes could lead to unexpected behavior.

Function: `IssuerExists`

The `IssuerExists` function checks for an issuer's existence by verifying that the retrieved details contain a nonempty name. Its simplicity minimizes potential errors, yet any flaw in

`GetIssuerDetails` would propagate here, affecting the validation of issuer status for subsequent verification operations.

Function: `GetAddressDetails`

The `GetAddressDetails` function retrieves address details from the store and filters the associated verification records, excluding any linked to issuers that no longer exist. It relies on correct unmarshaling of stored data and the issuer existence check. Improper filtering could lead to outdated or malicious verifications being considered valid.

Function: `SetAddressDetails`

The `SetAddressDetails` function writes address details to the store by serializing protocol buffer messages and writing them under a designated prefix. It must handle serialization errors gracefully to avoid corrupting user data.

Function: `RemoveAddressDetails`

The `RemoveAddressDetails` function removes address details from storage. Its proper use ensures that stale data does not persist, although an attacker inducing repeated removals might disrupt state consistency.

Function: `IsAddressVerified`

The `IsAddressVerified` function checks whether an address is marked as verified by reading the verification flag from the stored address details. Its correct operation is crucial because a false positive or negative in verification status could enable unauthorized actions or block legitimate ones.

Function: `SetAddressVerificationStatus`

The `SetAddressVerificationStatus` function updates an address's verification flag and writes the modified details back to storage. It avoids unnecessary writes if the status is already set, but any miscalculation in status updates might inadvertently change a user's verified state.

Function: `AddVerificationDetailsV2`

The `AddVerificationDetailsV2` function adds verification details with an explicitly provided compressed public key. It first confirms that the issuer is approved, then delegates most of the work to an internal function. After linking the verification to the provided public key and extracting its x-coordinate, it computes a credential hash for integration into the tree. Errors during public-key extraction or hash computation could undermine the verification process, making this function a critical point for cryptographic integrity.

Function: `AddVerificationDetails`

The `AddVerificationDetails` function adds verification details without requiring an explicit public key. If the user has a previously attached key, it links the verification accordingly and computes the credential hash. The reliance on the attached public key and conditional linking introduces subtle risks if key management is not handled correctly.

Function: `addVerificationDetailsInternal`

The `addVerificationDetailsInternal` function handles the core logic for adding verification details. It validates the verification type, issuer timestamp, and ensures nonempty proof data. It computes a unique verification ID using a Keccak-256 hash of the user address, verification type, and serialized details, then prevents duplicate entries by checking for existing IDs. Any weakness in the hash-collision resistance or timestamp validation could allow replay attacks or duplicate submissions, making this function a focal point of the compliance module's security.

Function: `SetVerificationDetails`

The `SetVerificationDetails` function, designed primarily for genesis initialization or testing, writes verification details directly into the store. It associates the verification with a user address and, if available, links it to a public key while computing the credential hash. Its direct store manipulation bypasses some runtime checks, so it must be used cautiously to avoid injecting malformed data.

Function: `MarkVerificationDetailsAsRevoked`

The `MarkVerificationDetailsAsRevoked` function marks a verification record as revoked by updating its state and, if a public key is linked, propagates the revocation into the revocation tree via a credential hash update. It retrieves the holder address and associated public key, so failure in these retrievals could prevent proper revocation. As revocation is central to compliance, any error here may allow revoked credentials to be erroneously accepted.

Function: `GetVerificationDetails`

The `GetVerificationDetails` function retrieves and unmarshals verification details from the store and ensures that the corresponding issuer still exists. If the issuer has been removed, it returns an empty structure. The reliance on issuer existence checks ensures outdated or unauthorized verifications are ignored, but errors in deserialization may lead to state inconsistencies.

Function: `GetVerificationDetailsByIssuer`

The `GetVerificationDetailsByIssuer` function filters a user's stored verifications based on the issuer's address and returns paired lists of verifications and their details. It depends on correct string comparisons and unmarshaling; subtle bugs could allow mismatches between expected and actual issuers.

Function: `GetCredentialHashByVerificationId`

The `GetCredentialHashByVerificationId` function reconstructs the credential from stored verification details and computes its cryptographic hash. It depends critically on the integrity of the underlying cryptographic functions and accurate retrieval of associated public keys. A weakness here would directly impact the security of credential issuance and revocation proofs.

Function: `HasVerificationOfType`

The `HasVerificationOfType` function checks whether a user possesses a valid verification of a specific type, optionally filtering by a set of expected issuers and considering expiration timestamps. Its logic must accurately compare types and timestamps; misinterpretation could lead to false positives or negatives regarding a user's compliance status.

Function: `GetVerificationsOfType`

The `GetVerificationsOfType` function returns detailed verification records that match a specified type and, optionally, expected issuers. It iterates over the user's verifications and applies filtering, relying on correct unmarshaling and comparison logic.

Function: `GetOperatorDetails`

The `GetOperatorDetails` function retrieves operator details from the store by unmarshaling the stored data into a structured format. Its simplicity belies its importance: accurate operator data is essential for access control and administrative actions.

Function: `AddOperator`

The `AddOperator` function validates and adds new operator details into storage, ensuring that only operators with approved types (such as regular operators) are registered. Incorrect validation of operator types could lead to unauthorized access or removal of critical admin roles.

Function: `RemoveRegularOperator`

The `RemoveRegularOperator` function removes an operator, provided that it is classified as a regular operator. It verifies both the existence and type of the operator before deletion. Its strict checks prevent accidental removal of initial or privileged operators, but any lapse here could compromise governance.

Function: `OperatorExists`

The `OperatorExists` function checks for the presence of an operator by verifying the existence of operator details in storage. Its reliability is fundamental to all administrative functions that require operator validation.

Functions: `IterateOperatorDetails`, `IterateVerificationDetails`, `IterateAddressDetails`, `IterateIssuerDetails`, `IterateHolderPublicKeys`, `IterateLinksToHolder`, `IterateLinksToPublicKey`

The Iterate* functions traverse various sections of the store using prefix iterators. They provide essential mechanisms for exporting state and performing internal consistency checks. However, any failure to close iterators correctly, handled by the `closeIteratorOrPanic` function, might lead to resource leaks or state read errors.

Functions: `ExportOperators`, `ExportVerificationDetails`, `ExportAddressDetails`, `ExportIssuerDetails`, `ExportHolderPublicKeys`, `ExportLinksVerificationIdToPublicKey`

The Export* functions export complete sets of stored data into genesis-compatible structures. They are critical for state backup, migration, and debugging. Their correctness depends on the proper functioning of the underlying iterator functions and accurate unmarshaling of stored data.

Function: `GetHolderPublicKey`

The `GetHolderPublicKey` function retrieves the compressed public key associated with a user address from storage.

Function: `SetHolderPublicKey`

The `SetHolderPublicKey` function checks if a public key is already set for a user before setting a new one. It validates the provided key by extracting its x-coordinate to ensure its correctness. This process prevents multiple or malformed keys from being attached to a single user, protecting the integrity of the public key registry.

Function: `SetHolderPublicKeyBytes`

The `SetHolderPublicKeyBytes` function writes the raw x-coordinate bytes of a public key into storage under a specific prefix. Its simplicity demands that the input bytes be correctly prevalidated to avoid corrupting the public-key mapping.

Function: `LinkVerificationToHolder`

The `LinkVerificationToHolder` function creates a one-to-one association between a verification ID and a user address. It checks for existing links and prevents reassigning a verification to a different holder, thereby maintaining the immutability of verification associations.

Function: `getHolderByVerificationId`

The `getHolderByVerificationId` function retrieves the user address linked to a given verification ID.

Function: `LinkVerificationIdToPubKey`

The `LinkVerificationIdToPubKey` function binds a verification ID to a public key. It ensures that a verification cannot be linked to multiple keys by checking existing associations. Any conflict here would raise an error and prevent potential key-substitution attacks.

Function: `GetPubKeyByVerificationId`

The `GetPubKeyByVerificationId` function fetches the public key associated with a verification ID from storage.

Function: `closeIteratorOrPanic`

The `closeIteratorOrPanic` function ensures that iterators used for state traversal are properly closed, panicking if an error occurs. While it guarantees resource cleanup, triggering a panic could lead to denial of service if exploited.

5.4. Component: Feemarket

Description

The Feemarket module calculates EIP-1559 block fees if the fees are enabled and keeps track of the amount of gas requested in each block, which the fee calculation depends on. It calculates the fee for the current block in its `BeginBlock` handler, which is stored and read by the AnteHandlers and the EVM module when needed. Each block's gas consumption is stored for the next block's fee calculation in the Feemarket module's `EndBlock` handler.

Invariants

Each block has a record of the previous block's gas consumption or the specified initial value for the genesis block.

Test coverage

Cases covered

- The fee calculation is unit-tested to have expected values, both when disabled and when enabled with several combinations of increases/decreases in gas consumption.
- Handling of the `MsgUpdateParams` message is tested with both valid and invalid signers.
- Setting of parameters is tested independently of the message handler with valid values.
- Validation of parameters is tested independently of setting parameters.
- The `EndBlock` handler is tested to update the state.

Cases not covered

- The `BeginBlock` handler is not directly tested to update the calculated fee (though this is handled by other integration tests).
- The `MsgUpdateParams` message is not integration-tested with invalid values.

Attack surface

Message: `MsgUpdateParams`

The `MsgUpdateParams` message updates the Feemarket module's parameters, including whether the fee is enabled, at what height to enable it if it should be enabled, a limit on the rate of change of the fee, and an elasticity multiplier that determines the gas limit relative to the gas target, the initial fee, the minimum gas price, and a minimum amount of gas to consider used by the previous block.

Its `ValidateBasic` method ensures that many of the parameters are nonnegative and that parameters that are used as denominators are nonzero (though it does not ensure this for the elasticity multiplier; see Finding 3.4. ↗).

Its `GetSigners` method ensures that the message is signed by the module's `Authority`, which is currently the Governance module.

5.5. Component: Vesting

Description

The Vesting module implements a monthly vesting mechanism that gradually releases tokens to beneficiaries over time, following an initial cliff period. It provides functionality for creating vesting accounts with a defined schedule, computing the amount of tokens vested versus those still locked, and tracking token delegation in accordance with the vesting schedule.

Invariants

The total vesting amount must equal the sum of the amounts distributed over all vesting periods.

Test coverage

Cases covered

- The `MsgCreateMonthlyVestingAccount` message is tested with success and failure cases.
- Vesting-account validation is tested with both valid and invalid parameters.
- Vesting calculations are tested to have expected values with simulated time passage and with combinations of delegation and undelegation events.

Attack surface (Cosmos message)

Message: `MsgCreateMonthlyVestingAccount`

The `MsgCreateMonthlyVestingAccount` message initiates the creation of a new monthly vesting account. Its `ValidateBasic` method verifies that the sender and recipient addresses are valid and nonzero, ensuring that neither is the zero address.

It checks that the cliff days are nonnegative, the number of months is at least one, and that the amount is strictly positive. The `GetSignBytes` and `GetSigners` methods ensure the message is properly signed, which helps prevent unauthorized creation of vesting accounts. These validations are critical to preventing malformed or malicious requests that could disrupt the vesting schedule or lead to unintended fund allocations.

Attack surface (vesting account)

Function: `HandleCreateMonthlyVestingAccount`

The `HandleCreateMonthlyVestingAccount` function processes `MsgCreateMonthlyVestingAccount` messages. It begins by ensuring that the coins to be transferred are allowed by the bank module and that the recipient address is not blocked. It then checks that no account already exists for the recipient, preventing accidental overwrites.

The function creates a new vesting account using the provided parameters, calculating the vesting schedule based on the sender's input for cliff days and months. It transfers the specified coins from the sender to the new vesting account and emits events recording the vesting-account creation details.

Any failure in these validations or in the coin-transfer process can result in the creation of vesting accounts with incorrect parameters or funds being locked improperly, exposing the system to potential financial inconsistencies or denial of service.

Function: `Validate`

The `Validate` function performs critical consistency checks on a vesting account. It ensures that the start time precedes the cliff time and end time and that the sum of all vesting periods exactly matches the original vesting amount. Any deviation may allow tokens to be unlocked prematurely or locked indefinitely, so this function is essential for preserving the integrity of the vesting schedule.

Function: `NewMonthlyVestingAccount`

The `NewMonthlyVestingAccount` function constructs a monthly vesting account by calculating the cliff time from the provided start time and cliff duration, then evenly distributing the total vesting coins across the specified number of months. Accurate arithmetic here is vital to ensure that the vesting schedule correctly reflects the intended distribution without rounding errors.

Function: `GetVestedCoins`

The `GetVestedCoins` function calculates the amount of tokens that have vested up to the current block time by iterating over the vesting periods after the cliff. Its correct operation is crucial because it directly determines the balance available for spending versus the amount that remains locked.

6. Assessment Results

At the time of our assessment, the reviewed code has been in the testnet phase since the summer of 2023. It has reached nearly 10 million blocks, more than 120K wallets, and 10 million transactions.

During our assessment on the scoped Swisstronik modules, we discovered nine findings. No critical issues were found. One finding was of high impact, three were of medium impact, four were of low impact, and the remaining finding was informational in nature.

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.