Zellic

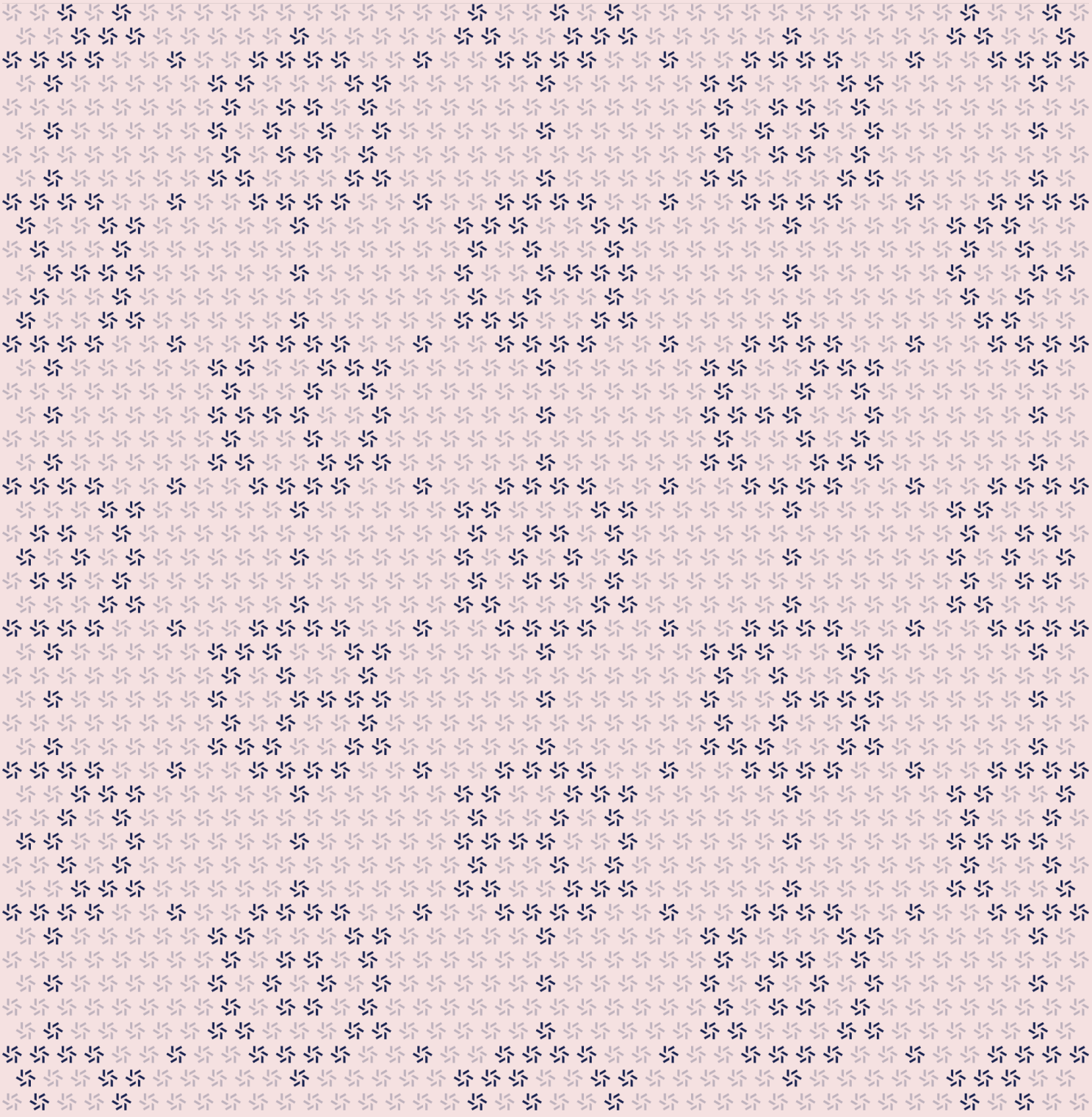
Prepared for
Jayant Krishnamurthy
Pyth Data Association

Prepared by
Ayaz Mammadov
Avraham Weinstock
Zellic

August 4, 2026

Pyth Pro Stellar Contracts

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **5**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 6

2. Introduction **7**

2.1. About Pyth Pro Stellar Contracts 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **11**

3.1. Previous guardian set is able to upgrade the executor contract during the grace period 11

3.2. Governance sub-call authorizations can be front-run 13

3.3. VAA verification can exceed Soroban's instruction limit for large guardian sets 15

3.4. Low-S ECDSA signatures become malleable in a future version of k256 17

4. System Design **22**

4.1. Component: Pyth Lazer 22

4.2. Component: Wormhole Executor 23

4.3. Component: Rust SDK 24

5. Assessment Results 26

5.1. Disclaimer 26

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 Pyth Data Association from July 23rd to July 27th, 2026. During this engagement, Zellic reviewed Pyth Pro Stellar Contracts'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:

- Is the Verified Action Approval (VAA) parsing done by the Wormhole contract consistent with the documented format ↗?
- Can signer set updates and contract bytecode upgrades only be performed by the intended parties?
- Can VAAs be replayed in unintended contexts?
- Are signatures correctly verified against the intended active keys?
- Is Soroban state correctly persisted?
- To what extent is the SDK resistant to misuse by downstream code?
- Will calls in a Pyth Governance Message (PTGM) payload inside a VAA execute as intended?
- Are per-transaction resource limits sufficient to execute the contract calls in the intended configuration?

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

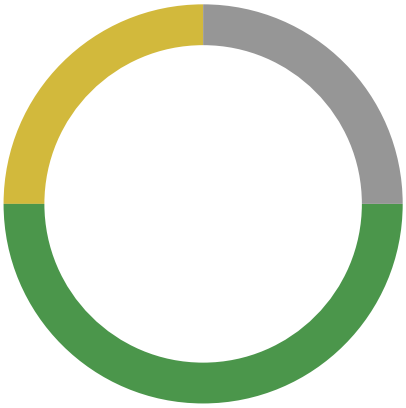
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Pyth Pro Stellar Contracts files, we discovered four findings. No critical issues were found. One finding was of medium impact, two were of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	1



2. Introduction

2.1. About Pyth Pro Stellar Contracts

Pyth Data Association contributed the following description of Pyth Pro Stellar Contracts:

> Pyth Network is a first-party financial oracle network that publishes real-time market data on more than 90 blockchains. Pyth Pro is Pyth's low-latency price channel, distributing publisher-signed price updates that consumers verify on-chain against a rotating set of trusted signers. This engagement covers the Stellar (Soroban) port of the Pyth Pro verifier, which brings price-feed verification to the Stellar ecosystem. The port consists of two Soroban smart contracts — a verifier that authenticates LE-ECDSA signed price updates against a trusted-signer set, and a Wormhole governance executor that verifies Wormhole VAAs and dispatches signer rotations, contract upgrades, and guardian-set upgrades — plus a companion Rust SDK for integrators.

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 files.

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

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

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We

also provide suggestions for possible optimizations, such as gas optimization, upgradability 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:

Pyth Pro Stellar Contracts Files

Type	Rust
Platform	Stellar
Target	pyth-lazer
Repository	https://github.com/pyth-network/pyth-lazer/ ↗
Version	`6220483f02238a4e5b4af5f847727854f2cd121e`

Programs	`contracts/pyth-lazer-stellar`
	`contracts/wormhole-executor-stellar`
	`sdk/rust`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.9 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

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:

Ayaz Mammadov
Engineer
ayaz@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 23, 2026	Start of primary review period
July 27, 2026	End of primary review period

3. Detailed Findings

3.1. Previous guardian set is able to upgrade the executor contract during the grace period

Target	Wormhole Executor		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When the `WormholeExecutor::update_guardian_set` method creates a new guardian set, the previous guardian set can still execute actions through the `WormholeExecutor::execute_governance_action` method until 24 hours (`GUARDIAN_SET_EXPIRY`) have passed. The previous guardian set is specifically disallowed from calling `update_guardian_set` again:

```
// Governance actions require the *current* guardian set. The 24-hour
// grace window in `verify_vaa` applies to non-governance message VAAs
// only; allowing a retired-but-in-window set to authorize a guardian
// set upgrade would let a quorum of the old keys install a malicious
// set N+1 for up to 24 hours after any rotation.
if vaa.guardian_set_index != guardian::get_guardian_set_index(&env)? {
    return Err(ContractError::InvalidGuardianSetIndex);
}
```

However, no such check is present in the branch of `execute_governance_action` that upgrades the executor contract:

```
GovernanceAction::UpgradeExecutor(payload) => {
    let wasm_hash = BytesN::from_array(&env, &payload.wasm_digest);
    env.deployer().update_current_contract_wasm(wasm_hash);
}
```

Impact

Upgrading the executor contract allows the previous guardian set to take control of it. This is a more privileged action than updating the guardian set, because unlike updating the guardian set, an upgrade can lock out the current guardian set, for example by upgrading to a version of the contract that has no grace period and a new set of keys hardcoded.

Recommendations

Disallow guardian sets other than the current one from executing `UpgradeExecutor` actions.

Remediation

Pyth Data Association has acknowledged this issue and provided the following comment:

> The logic here is intentional, as the Wormhole protocol mandates a 24 hour window where old guardian sets are accepted as signers for all messages. This window is required due to asynchrony between when contract upgrades happen and when the signer set rotates. This finding is unlikely to cause a problem in practice, as for it to cause a problem, the old guardian set has to be compromised in the first place.

3.2. Governance sub-call authorizations can be front-run

Target	Wormhole Executor		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `WormholeExecutor::execute_governance_action` method is permissionless and never calls `require_auth` itself. Its `Call` action forwards to whatever target contract and function the payload specifies:

```
GovernanceAction::Call(payload) => {
    let func = decode_function_symbol(&env, &payload.function_name)?;
    let args = Vec::<Val>::from_xdr(&env, &payload.args_xdr)
        .map_err(|_| ContractError::InvalidArgsEncoding)?;
    env.invoke_contract::<()>(&payload.target_contract, &func, args);
}
```

In Soroban, an authorization for a sub-call is only bound to the outer call if the outer call is itself authorized. Because `execute_governance_action` is not authorized, any authorization that the target function needs cannot be nested under it and must instead be sent as its own top-level entry in the transaction.

Such a top-level entry is not tied to the governance call, so anyone who sees the transaction can copy that entry into a transaction of their own.

This only matters if the target function requires authorization from an account or smart wallet. The in-scope target (`PythLazerContract`) requires only the executor's own authorization, which is implicit and cannot be copied, so it is unaffected in the configuration reviewed.

Impact

An attacker could copy the authorization entry and spend its nonce first. When the real governance transaction runs, the inner `require_auth` fails and the whole action reverts, which blocks the governance action.

The copied authorization also runs the inner call outside the governance flow, which can matter for multicall or smart-wallet targets.

Recommendations

The ideal fix in this scenario would be to rearchitect the smart contracts to use `__check_auth`. As a result, `execute_governance_action` would no longer be necessary. Instead, top-level functions that require authorization from the executor would call `executor.require_auth`, with the resulting authorization entries containing the required data (e.g., signatures) that can then be passed to `__check_auth`.

There are two reasonable ways to address this issue in the scope and time limitations of this audit:

- If only a limited set of governance actions is required, restrict the allowed target contracts and functions accordingly. Ideally, only allow calls that do not require additional top-level `require_auth` authorizations from accounts or smart wallets.
- If arbitrary targets must remain supported, document that authorizations required by the target call are not bound to the governance action and can be front-run. Integrators should avoid using `Call` for workflows where this could cause problems, such as multicalls, smart wallets, or other patterns where consuming the authorization independently can affect correctness.

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit `7b4d21aa` ↗.

This was documented in the logic for future developers and users with the following comment:

```
/// This entry point is permissionless and never calls `require_auth` on the
/// submitter, so an authorization that a `Call` target requires cannot be
/// nested under the governance action — it must be submitted as its own
/// top-level authorization entry, which anyone observing the transaction
/// can lift into a transaction of their own and consume independently. The
/// only current target, `PythLazerContract`, requires nothing but this
/// executor's own implicit authorization and is unaffected; integrators
/// wiring up new `Call` targets must avoid workflows where consuming that
/// authorization independently affects correctness, such as multicalls,
/// smart wallets, or any target that authenticates a principal other than
/// this executor.
```

3.3. VAA verification can exceed Soroban's instruction limit for large guardian sets

Target	Wormhole Executor		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `verify_vaa` function loops over every signature in the VAA and, for each one, runs a `secp256k1_recover` and a Keccak hash to recover and check the signer:

```
for i in 0..num_sigs {
    // [...]
    let recovered_pubkey =
        env.crypto()
            .secp256k1_recover(&digest, &sig.signature, sig.recovery_id);
    let recovered_addr = guardian::eth_address_from_pubkey(env,
    &recovered_pubkey);
    // [...]
}
```

Each recovery is expensive. Soroban caps a transaction at 400 million instructions, and a large enough set of signatures pushes the loop past that cap. With a full guardian set of 255 signers and its corresponding quorum of 171 signatures, verification exceeds the limit, so the transaction fails and the VAA can never be verified on-chain.

Wormhole's live guardian set is much smaller than this, so at the time of this assessment it is not a problem in practice. It becomes one only if the guardian set grows substantially.

Impact

If the guardian set is large enough, no VAA from that set can be verified, so `execute_governance_action` and `update_guardian_set` stop working.

Recommendations

Keep the guardian set small enough that verification stays within the 400 million instruction budget.

Remediation

Pyth Data Association has acknowledged this issue and provided the following comment:

> The set of authorized signers is 19 participants, making this scenario unlikely to occur in practice.

3.4. Low-S ECDSA signatures become malleable in a future version of k256

Target	Soroban		
Category	Protocol Risks	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Both the `PythLazerContract` and `WormholeExecutor` contracts use Soroban's `secp256k1_recover` for checking ECDSA signatures. The rs-soroban-env crate uses the k256 crate's implementation of signature verification in recover_key_ecdsa_secp256k1_internal ↗:

```
pub(crate) fn recover_key_ecdsa_secp256k1_internal(
    &self,
    hash: &Hash,
    sig: &k256::ecdsa::Signature,
    rid: k256::ecdsa::RecoveryId,
) -> Result<ScBytes, HostError> {
    let _span = tracy_span!("secp256k1 recover");
    self.charge_budget(ContractCostType::RecoverEcdsaSecp256k1Key, None)?;
    let recovered_key =
        k256::ecdsa::VerifyingKey::recover_from_prehash(hash.as_slice(), &sig,
    rid).map_err(
            |_| {
                self.err(
                    ScErrorType::Crypto,
                    ScErrorCode::InvalidInput,
                    "ECDSA-secp256k1 signature recovery failed",
                    &[],
                )
            },
        )?;
    Ok(ScBytes::from(crate::xdr::BytesM::try_from(
        recovered_key
            .to_encoded_point(/*compress:*/ false)
            .as_bytes(),
    )?))
}
```

At the time of this assessment, Soroban uses k256 version 0.13.3 ↗, which checks in verify_prehashed ↗ that the S component of an ECDSA signature is less than half the curve order:

```
fn verify_prehashed(&self, z: &FieldBytes, sig: &Signature) -> Result<(),
    Error> {
    if sig.s().is_high().into() {
        return Err(Error::new());
    }

    hazmat::verify_prehashed(&self.into(), z, sig)
}
```

However, this check is omitted in k256 version 0.14.0.

Impact

The following proof of concept demonstrates that if (r, s, v) is a valid signature for a message, then for some alternate recovery ID v', $(r, n - s, v')$ is not a valid signature for that message in k256 0.13.3 but is a valid signature for it in k256 0.14.0.

```
/*
[package]
name = "secp256k1-malleability-poc"
version = "0.1.0"
edition = "2024"

[features]

[dependencies]
k256_old = { package = "k256", version = "0.13.3" }
k256_new = { package = "k256", version = "0.14.0" }
*/
use std::ops::Neg;

fn v13() {
    println!("Using k256 v0.13");
    use k256_old::ecdsa::{RecoveryId, Signature, SigningKey, VerifyingKey};
    use k256_old::elliptic_curve::{PrimeField, rand_core::OsRng};

    let mut successes = 0;
    let mut rng = OsRng;
    let privkey = SigningKey::random(&mut rng);
    let msg = b"hello";
    let (signature, recovery_id) = privkey.sign_recoverable(msg).unwrap();
    println!("{:x?}", privkey.verifying_key());
    println!("{:x?}", signature);
    println!("{:?}", recovery_id);
    for new_recovery_id in 0..4 {
```

```
        let new_recovery_id = RecoveryId::from_byte(new_recovery_id).unwrap();
        println!("new_recovery_id {:?}", new_recovery_id);
        let (r, s) = signature.split_scalars();
        let mut r_bytes = r.to_repr();
        let mut s_bytes = s.to_repr();
        use k256_old::{
            Secp256k1,
            elliptic_curve::{Curve, FieldBytesEncoding, bigint::NonZero},
        };
        if recovery_id.is_x_reduced() != new_recovery_id.is_x_reduced() {
            r_bytes = <Secp256k1 as Curve>::Uint::decode_field_bytes(&r_bytes)
                .wrapping_sub(&Secp256k1::ORDER)
                .rem(&NonZero::new(Secp256k1::ORDER).unwrap())
                .encode_field_bytes();
        }
        if recovery_id.is_y_odd() != new_recovery_id.is_y_odd() {
            println!("Negating S");
            s_bytes = s.neg().to_bytes();
        }
        let new_sig = Signature::from_scalars(r_bytes, s_bytes).unwrap();
        println!("new_sig {:x?}", new_sig);
        let new_vk = VerifyingKey::recover_from_msg(msg, &new_sig,
    new_recovery_id);
        println!("new_vk {:x?}", new_vk);
        if let Ok(new_vk) = new_vk {
            if recovery_id != new_recovery_id {
                assert_ne!(signature, new_sig);
            }
            assert_eq!(new_vk, *privkey.verifying_key());
            successes += 1;
        }
    }
    assert_eq!(successes, 1);
    println!("Only one of the four possible recovery IDs was valid.");
}

fn v14() {
    println!("Using k256 v0.14");
    use k256_new::ecdsa::{RecoveryId, Signature, SigningKey, VerifyingKey};
    use k256_new::elliptic_curve::{Generate, PrimeField};
    let mut successes = 0;
    let privkey = SigningKey::try_generate().unwrap();
    let msg = b"hello";
    let (signature, recovery_id) = privkey.sign_recoverable(msg);
    println!("verifying_key: {:x?}", privkey.verifying_key());
    println!("signature: {:x?}", signature);
    println!("recovery_id: {:?}", recovery_id);
```

```
    for new_recovery_id in 0..4 {
        let new_recovery_id = RecoveryId::from_byte(new_recovery_id).unwrap();
        println!("new_recovery_id: {:?}", new_recovery_id);
        let (r, s) = signature.split_scalars();
        let mut r_bytes = r.to_repr();
        let mut s_bytes = s.to_repr();
        use k256_new::{Secp256k1, elliptic_curve::Curve};
        if recovery_id.is_x_reduced() != new_recovery_id.is_x_reduced() {
            r_bytes = <Secp256k1 as Curve>::Uint::from_be_slice(&r_bytes)
                .wrapping_sub(&Secp256k1::ORDER)
                .rem(Secp256k1::ORDER.as_nz_ref())
                .to_be_bytes()
                .into();
        }
        if recovery_id.is_y_odd() != new_recovery_id.is_y_odd() {
            println!("Negating S");
            s_bytes = s.neg().to_bytes();
        }
        let new_sig = Signature::from_scalars(r_bytes, s_bytes).unwrap();
        println!("new_sig: {:x?}", new_sig);
        let new_vk = VerifyingKey::recover_from_msg(msg, &new_sig,
    new_recovery_id);
        println!("new_vk: {:x?}", new_vk);
        if let Ok(new_vk) = new_vk {
            if recovery_id != new_recovery_id {
                assert_ne!(signature, new_sig);
            }
            assert_eq!(new_vk, *privkey.verifying_key());
            successes += 1;
        }
    }
    assert_eq!(successes, 2);
    println!("Two of the four possible recovery IDs are valid.");
}

fn main() {
    v13();
    v14();
}
```

This matters for consumers that deduplicate price updates by signature rather than preventing replays by tracking the highest observed timestamp. Against such a consumer, an attacker could submit an unmodified price update with timestamp t_1 and the original (r, s, v), then submit a second unmodified price update with timestamp t_2, and then replay the t_1 price update with an unmodified payload and the malleated signature $(r, n - s, v')$. The deduplication check would not recognize the replay, so the consumer would accept the stale t_1 price again.

Recommendations

Include a low-S check in the contract, in case Soroban upgrades to k256 0.14.0 without introducing a low-S check of its own.

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit 7b4d21aa ↗.

This was fixed by documenting this edge case.

```
// Parse signature (64 bytes at offset 4). Non-canonical high-`s` signatures
// need no check here: the host rejects them inside secp256k1_recover below.
// It does so by trapping rather than returning a ContractError, unlike the
// recovery_id range that is checked explicitly below.
```

4. 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.

4.1. Component: Pyth Lazer

Description

The Pyth Lazer contract verifies that price updates are signed by a non-expired authorized signer. It is initialized with an executor address (e.g., the address of the Wormhole Executor contract), and that executor can add signers of price updates, remove them, and update their expiry timestamps. The executor can also upgrade the Pyth Lazer contract itself with new WASM bytecode.

Invariants

- Once initialized, `executor` cannot be modified except by WASM bytecode upgrades.
- The signer set is held in persistent storage, and the executor address is held in instance storage.
- Every method except the upgrade method extends the time to live (TTL) of the storage it reads or modifies.

Test coverage

Cases covered

- A valid price update with a non-expired signer public key successfully verifies, even if the current ledger timestamp is one second before the key would expire, and even if other keys are present in the map from signers to expiry times.
- Modified payloads fail to verify, including those with modified header bytes, truncations, modified signature recovery IDs, and modified length fields.
- Verifying an unmodified price update with an incorrect key or an expired correct key fails.
- Modifying the signer set and its expiry times with the correct executor address has the expected results when interleaved with verifications and state queries.
- Modifying the signer set from an address other than the executor fails.
- Upgrading the contract with the correct executor address succeeds, and attempting to upgrade the contract from another address fails.

Cases not covered

- No proptest or fuzzer harness is present for testing that random modifications to the

price update fail.

Attack surface

Every state-modifying method either can be called only once (`__constructor`) or requires authorization from the executor (`update_trusted_signer`, `upgrade`).

The `verify_update` method ensures that the payload to be verified has the correct header, has valid length fields, and is signed by a non-expired authorized signer.

4.2. Component: Wormhole Executor

Description

The Wormhole Executor contract executes governance actions signed on other chains against contracts on Stellar. Governance actions are emitted on the source chain as Wormhole Verifiable Action Approvals (VAAs) and submitted by relayers to the Wormhole Executor contract. VAAs are signed by a $\lfloor \frac{2*\texttt{num_guardians}}{3} \rfloor + 1$ majority of the current guardian set (13-of-19 in the expected deployment). A VAA carries either a guardian set update or a payload holding a Pyth Governance Message (PTGM), which in turn specifies either a contract call or a WASM bytecode upgrade for the executor itself.

When the guardian set is updated, the previous guardian set is given a 24-hour window until expiration, during which it can issue governance actions (including upgrading the executor — see Finding 3.1. ↗) but cannot update the guardian set.

Invariants

- The guardian sets are non-empty.
- The guardian sets, index of the current guardian set, and governance sequence numbers are stored in persistent storage.
- The Wormhole chain IDs and governance/upgrade emitter addresses are stored in instance storage.
- The sequence numbers of executed governance actions are strictly increasing.
- All methods on the contract extend the TTL of the storage they read or modify.

Test coverage

Cases covered

- Various PTGM payloads are constructed and parsed, both for success cases (which are verified to contain the expected data) and failure cases (which are verified to fail with the expected error).

- Various VAA envelopes are constructed and parsed, including a mix of synthetic success cases, mainnet success cases as regression tests, and VAA envelopes that exercise each failure case.
- Various well-formed VAA envelopes are constructed, parsed, and verified, both for success and failure cases. The success cases include 1-of-1, 3-of-3, and 13-of-19 guardian sets. The failure cases include fewer signatures than the quorum thresholds, signatures with the wrong key, signatures from a guardian set other than the current one, duplicate or out-of-bounds indices into the current guardian set, and modified recovery IDs in signatures.
- VAA envelopes containing guardian set updates are tested for success (including consecutive updates) and failure (including incorrect chain IDs and addresses).
- Execution of governance actions is tested for both success and failure in the integration tests. Failure cases include incorrect chain IDs and addresses at both the PTGM and the VAA level, each as a separate test case.

Cases not covered

- No proptest or fuzzer harness is present for testing that random modifications to PTGMs or VAAs fail.

Attack surface

Apart from the initializer (`__constructor`), which sets up the guardian set, every state-modifying method requires signatures from a majority of either the current guardian set (for `update_guardian_set`) or a non-expired guardian set (for `execute_governance_action`). The fields used to check that the emitter is the expected one, along with the payload itself, are covered by the signature.

4.3. Component: Rust SDK

Description

The Rust SDK allows other Soroban contracts to parse price updates and to submit calls to the Pyth Lazer contract to verify that price updates are signed by a current signer. It distinguishes unverified price updates (`Update`) from verified price updates (`VerifiedPayload`). The former is constructed through `parse_payload`, and the latter through `PythLazerClient::verify_update`, which calls the `verify_update` method on the contract address that the `PythLazerClient` was constructed with.

Invariants

- Downstream code cannot construct `VerifiedPayload` without either calling `PythLazerClient::verify_update`, using unsafe code (e.g., `core::mem::transmute`), or exploiting soundness bugs in the Rust compiler.

- Price update parsing returns errors of type `ParseError` instead of panicking when given invalid payload bytes.

Test coverage

Cases covered

- Parsing price updates is tested with success cases that include various combinations of present and absent fields, and with failure cases that include too few or too many bytes, incorrect headers, and invalid fields.
- The integration tests use `PythLazerClient::verify_update` with both success and failure cases.

Cases not covered

- N/A

Attack surface

Downstream code is responsible for passing the correct address of the Pyth Lazer contract when calling `PythLazerClient::new`. If it passes the address of a contract with a mock `verify_update` method instead, the SDK would appear to verify price updates that were never authenticated by the intended signer.

5. Assessment Results

During our assessment on the scoped Pyth Pro Stellar Contracts files, we discovered four findings. No critical issues were found. One finding was of medium impact, two were of low impact, and the remaining finding was informational in nature.

5.1. 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.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

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.