



SMART CONTRACT SECURITY AUDIT

# Stork

Fuel Oracle Contracts

CLIENT

Stork

PLATFORM

Fuel

LANGUAGE

Sway

REVIEW

July 2025

## DOCUMENT

## Document information

|            |                                     |
|------------|-------------------------------------|
| Project    | Stork, Fuel Oracle Contracts        |
| Client     | Stork                               |
| Platform   | Fuel                                |
| Language   | Sway                                |
| Repository | Not recorded in the original report |
| Commit     | Not recorded in the original report |
| Methods    | Not recorded in the original report |
| Review     | 26 June to 4 July 2025              |

## Document history

| VERSION | DATE         | CHANGE                                                                |
|---------|--------------|-----------------------------------------------------------------------|
| 1.0     | 26 July 2025 | Original report issued <a href="#">View original report</a>           |
| 1.1     | October 2026 | Reformatted into the Safe Edges standard template. Findings unchanged |

This report was reformatted into the Safe Edges standard template in October 2026. Findings, severities, recommendations and code references are unchanged from the original report, which is preserved unchanged in the [Safe Edges Audit Portfolio on GitHub](#).

## CONTENTS

# Contents

|    |                                                                                                                   |    |
|----|-------------------------------------------------------------------------------------------------------------------|----|
| 01 | Executive summary                                                                                                 | 3  |
| 02 | Scope                                                                                                             | 4  |
| 03 | Methodology                                                                                                       | 5  |
| 04 | Severity and status                                                                                               | 6  |
| 05 | Findings summary                                                                                                  | 7  |
| 06 | Detailed findings                                                                                                 | 8  |
|    | H-01 Tokens Sent to the Contract Are Not Checked, Allowing the utilization of Worthless Tokens to Pay for Updates | 8  |
|    | M-01 Fees are never refunded if unspent                                                                           | 9  |
|    | L-01 No Way to Update or Revoke Contract Ownership                                                                | 10 |
|    | L-02 Incomplete Handling of match Statements                                                                      | 11 |
|    | L-03 Signature Malleability and Replay Risk                                                                       | 12 |
|    | I-01 Front-running initialization                                                                                 | 13 |
|    | Disclaimer                                                                                                        | 14 |

# Executive summary

## About Stork

An oracle protocol that enables ultra-low latency connections between data providers and both on- and off-chain applications.

## Summary of the review

Safe Edges reviewed Stork's Sway oracle contracts on Fuel between 26 June and 4 July 2025. Six issues were identified: one High, one Medium, three Low and one Informational. The High issue let subscribers pay update fees with any token, including worthless ones they create themselves. Stork fixed the High and all three Low issues, and acknowledged the Medium and Informational issues, with its comments recorded in this report.

## Findings by severity



## Findings by status

| STATUS       | ISSUES |
|--------------|--------|
| Resolved     | 4      |
| Acknowledged | 2      |

## Scope

---

|            |                                     |
|------------|-------------------------------------|
| Repository | Not recorded in the original report |
| Commit     | Not recorded in the original report |
| Platform   | Fuel                                |
| Language   | Sway                                |

Only the code at the commit above was reviewed. Changes made after this commit are not covered by this report.

## Methodology

---

Methods recorded for this engagement: Not recorded in the original report.

### Manual review

Auditors read the in-scope code line by line, tracing control flow, state changes and trust boundaries against the intended behaviour described by the client.

### Threat modelling

We map privileged roles, external dependencies and value flows, then test the assumptions the code makes about each of them.

### Functional and automated testing

Where tooling exists for the language and chain, we run static analysis and targeted tests to confirm suspected issues and check edge cases.

### Fix review

After the client addresses the findings, we review the updated code and record the final status of each issue.

## Severity and status

### Severity levels

| SEVERITY      | DEFINITION                                                                                                                                            |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Critical      | Direct loss or permanent lock of funds, or full compromise of the protocol, with few or no preconditions. Must be fixed before deployment.            |
| High          | Significant loss of funds, broken core functionality or manipulation of critical state under realistic conditions. Should be fixed before deployment. |
| Medium        | Limited loss or degraded functionality, or issues that need specific conditions or privileged access to exploit. Should be fixed.                     |
| Low           | Minor impact, low likelihood, or deviations from best practice. Fix when practical.                                                                   |
| Informational | Code quality, documentation, gas and style observations with no direct security impact.                                                               |

### Status definitions

| STATUS             | DEFINITION                                                             |
|--------------------|------------------------------------------------------------------------|
| Resolved           | The issue was fixed and the fix was verified.                          |
| Partially resolved | The risk was reduced but not fully removed.                            |
| Acknowledged       | The client accepted the issue without changing the code.               |
| Open               | The issue was not addressed at the time of the report.                 |
| Not recorded       | The original report did not record a resolution status for this issue. |

## Findings summary

| ID   | TITLE                                                                                                        | SEVERITY      | STATUS                                                   |
|------|--------------------------------------------------------------------------------------------------------------|---------------|----------------------------------------------------------|
| H-01 | Tokens Sent to the Contract Are Not Checked, Allowing the utilization of Worthless Tokens to Pay for Updates | High          | ● Resolved <a href="#">fix</a><br><a href="#">commit</a> |
| M-01 | Fees are never refunded if unspent                                                                           | Medium        | ● Acknowledged                                           |
| L-01 | No Way to Update or Revoke Contract Ownership                                                                | Low           | ● Resolved <a href="#">fix</a><br><a href="#">commit</a> |
| L-02 | Incomplete Handling of match Statements                                                                      | Low           | ● Resolved <a href="#">fix</a><br><a href="#">commit</a> |
| L-03 | Signature Malleability and Replay Risk                                                                       | Low           | ● Resolved <a href="#">fix</a><br><a href="#">commit</a> |
| I-01 | Front-running initialization                                                                                 | Informational | ● Acknowledged                                           |

## Detailed findings

### H-01 Tokens Sent to the Contract Are Not Checked, Allowing the utilization of Worthless Tokens to Pay for Updates

Severity **High** Status ● Resolved [fix commit](#)

#### DESCRIPTION

Subscribers receive updates from aggregators, and fees are required for every price update. However, on the Fuel ecosystem, every token is a native asset, allowing subscribers to bypass fees by creating their own custom but worthless tokens and sending the required amount to the `update_temporal_numeric_values_v1()` function to avoid paying with the intended token.

Here is the relevant function:

```
#[storage(read, write), payable]
fn update_temporal_numeric_values_v1(update_data: Vec<TemporalNumericValueInput>)
{
    // ... code snippet to update the tvns
    let required_fee = get_total_fee(num_updates);
    // @audit the msg asset id isn't checked, allowing any msg asset to be accepted
    if std::context::msg_amount() < required_fee {
        log(StorkError::InsufficientFee);
        revert(0);
    }
}
```

While `msg_amount()` is checked, `msg_asset_id()` is never verified, allowing any token to satisfy the fee requirement. In practice, this could allow users to pay with valueless tokens rather than the protocol's designated token, undermining the economic model of the system and potentially preventing the protocol from funding aggregators or capturing revenue.

#### RECOMMENDATION

Strictly validate the accepted token's asset ID to ensure only the designated token is accepted for payment.

Below is an example of the corrected implementation:

```
#[storage(read, write)]
#[payable]
fn update_temporal_numeric_values_v1(update_data: Vec<TemporalNumericValueInput>)
{
    // ... code to update tvns
    let required_fee = get_total_fee(num_updates);
    // Expected asset ID (e.g., BASE_ASSET_ID or your token's asset ID)
    let expected_asset_id = BASE_ASSET_ID; // or set to your token's ID
    // Check the correct asset and sufficient amount
    if std::context::msg_asset_id() != expected_asset_id {
```

```
        log(StorkError::InvalidPaymentAsset);
        revert(0);
    }
    if std::context::msg_amount() < required_fee {
        log(StorkError::InsufficientFee);
        revert(0);
    }
}
```

This ensures that only the intended token can be used to pay fees, preserving protocol integrity and economic value.

## M-01 Fees are never refunded if unspent

Severity **Medium** Status ● Acknowledged

### DESCRIPTION

The `update_temporal_numeric_valuess_v1()` function is a core function of the stork contract, where on each call the caller specifies the feeds and forwards the required amount of fees needed for the given updates, however given the scenario where some of the feeds to be updated are already updated (where no fee is charged), the excess being sent on the contract call remains in the contract without no refund to the caller even though no updates were made.

You can see the `required_fee` is the product of the stork fee and the number of updates made in that context.

```
if (num_updates == 0) {
    log(StorkError::NoFreshUpdate);
    revert(0);
}
let required_fee = get_total_fee(num_updates);
```

In a scenario where updates are already made, the user funds sent will be stuck in the contract with no refunds to the user, leading to unintended loss for the callers.

```
#[storage(read, write), payable]
fn update_temporal_numeric_values_v1(update_data: Vec<TemporalNumericValueInput>) {
    let mut num_updates = 0;
    let mut i = 0;
    while i < update_data.len() {
        let x = update_data.get(i).unwrap();
        let verified = verify_stork_signature(
            _stork_public_key(),
            x.id,
            x.temporal_numeric_value
                .timestamp_ns,
            x.temporal_numeric_value
                .quantized_value,
            x.publisher_merkle_root,
            x.value_compute_alg_hash,
            x.r,
```

```
        x.s,  
        x.v,  
    );  
    // code snippet ...  
    let required_fee = get_total_fee(num_updates);  
    if (std::context::msg_amount() < required_fee) {  
        log(StorkError::InsufficientFee);  
        revert(0);  
    } //no refund if fee is too high.  
}
```

#### RECOMMENDATION

Refund unspent fees after processing the update.

#### CLIENT COMMENT

Most pushers will be running Stork's open source chain pusher, which should mitigate this edge case. Stork would rather not introduce two way payment capabilities at this time. Stork reserves the capability to address this further in a future upgrade.

## L-01 No Way to Update or Revoke Contract Ownership

Severity **Low** Status ● Resolved [fix commit](#)

#### DESCRIPTION

There is no way to update or revoke ownership, even though the contract implements the src5 standard. In scenarios where ownership must be transferred or revoked (e.g., team changes, compromised keys, governance upgrades), this limitation prevents any update to the owner address, making recovery or delegation impossible.

#### RECOMMENDATION

Implement a secure ownership transfer mechanism using a two-step process (propose and accept) to avoid accidental or malicious changes.

For example:

1. Propose transfer:

```
#[storage(read, write)]  
fn propose_owner(new_owner: Address) {  
    onlyOwner();  
    state.pending_owner = new_owner;  
}
```

2. Accept transfer:

```
#[storage(read, write)]  
fn accept_ownership() {  
    if std::context::msg_sender() != state.pending_owner {  
        revert(0);  
    }  
}
```

```
}
state.owner = state.pending_owner;
state.pending_owner = ZERO_ADDRESS;
}
```

This approach ensures ownership can be updated safely without requiring a contract redeployment.

#### CLIENT COMMENT

Two step ownership transfer has been implemented, however Stork would rather not introduce ownership revocation at this time. Stork reserves the capability to implement ownership revocation in a future upgrade.

## L-02 Incomplete Handling of match Statements

Severity **Low** Status ● Resolved [fix commit](#)

#### DESCRIPTION

In the `only_owner()` call, not all match cases are handled safely, which can lead to unintended access to functions that are meant to be owner-restricted.

For example, if ownership is revoked to signal permissionlessness by the protocol, anyone will be able to call functions protected by `only_owner()`.

Below is the relevant code snippet:

```
#[storage(read)]
fn only_owner() {
    match storage.owner.read() {
        standards::src5::State::Uninitialized => {},
        standards::src5::State::Initialized(owner) => {
            require(msg_sender().unwrap() == owner, "Only Owner");
        },
        standards::src5::State::Revoked => {} // no logic for revoked state
    }
}
```

In this logic, the Revoked state simply does nothing, effectively allowing any caller through.

#### RECOMMENDATION

Ensure exhaustive and secure handling of all match cases.

For example, explicitly revert in the Revoked state to prevent unintended access:

```
standards::src5::State::Uninitialized => {},
// .. snippet...
standards::src5::State::Revoked => {
    revert(0);
}
```

## L-03 Signature Malleability and Replay Risk

Severity **Low** Status ● Resolved [fix commit](#)

## DESCRIPTION

When obtaining a signature via  $r$ ,  $s$ , and  $v$ , the `try_get_rsv_signature_from_parts()` function only checks whether  $v$  is 27 or 28. However, it does not protect against signature malleability.

In Ethereum, all signatures are inherently malleable because every  $s$  value has a symmetrically corresponding  $s$  on the other side of the secp256k1 curve. Without restricting the allowed  $s$  range, signatures can be manipulated.

Additionally, without including a nonce in each signature, the same signature can be replayed multiple times.

Although this is not directly exploitable in this contract, since updates only occur if the current timestamp exceeds the previous one, it is considered best practice to enforce non-malleable signatures and include a nonce whenever using Ethereum-signed messages.

## IMPACT

Low, as this issue is not directly exploitable in the current implementation.

## RECOMMENDATION

1. Restrict  $s$  to the lower half of the curve (EIP-2 constraints): Require  $s$  to be less than or equal to  $\text{secp256k1n} / 2$ .
2. Introduce a nonce per signature to prevent replay attacks.

Below is an example of enhanced validation logic:

```
fn try_get_rsv_signature_from_parts(r: b256, s: b256, v: u8) -> Option<Secp256k1>
{
    // Reject malleable signatures by requiring s in lower range
    require(
        s <= b256::from(0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0),
        "InvalidSignature"
    );
    // EIP-2 still allows signature malleability for ecrecover().
    // Remove this possibility and make the signature unique.
    // Appendix F in the Ethereum Yellow Paper
    (https://ethereum.github.io/yellowpaper/paper.pdf):
    // Valid s: 0 < s < secp256k1n / 2 + 1
    // Valid v: {27, 28}
    match v {
        27 => {
            // Clear the highest bit
            let mask = b256::max()
                & !
                (b256::from(0x8000000000000000000000000000000000000000000000000000000000000000));
            let s_cleared = s & mask;
            Some(Secp256k1::from((r, s_cleared)))
        }
        28 => {
```

```
// Set the highest bit
let mask =
b256::from(0x8000000000000000000000000000000000000000000000000000000000000000);
let s_set = s | mask;
Some(Sevp256k1::from((r, s_set)))
}
_ => None,
}
}
```

#### ADDITIONAL RECOMMENDATION

Include a nonce in each signature and store it in contract state to ensure each signed message can only be used once.

### I-01 Front-running initialization

Severity **Informational** Status **● Acknowledged**

#### DESCRIPTION

The call to `initialize()` sets core states required by the protocol to function, the issue stems from the ability for the function to be initialized before the real owner, either leading to grieving where the owner has to redeploy the contract where gas is wasted, or the owner is unaware leading to more issues in the future.

```
#[storage(read, write)]
fn initialize(
    initial_owner: Identity,
    stork_public_key: EvmAddress,
    single_update_fee_in_wei: u64,
) {
    require(!storage.initialized.read(), "Already initialized");
    require(!storage.initializing.read(), "Already initializing");
    storage.initializing.write(true);
    storage
        .owner
        .write(standards::src5::State::Initialized(initial_owner));
    set_single_update_fee_in_wei(single_update_fee_in_wei);
    set_stork_public_key(stork_public_key);
    storage.initialized.write(true);
}
```

#### RECOMMENDATION

Set a constant identity that is the only one permitted to call this function after deployment, or ensure a script is called to initialize it after contract creation

# Disclaimer

---

This report is based on the code at the commit listed in the Scope section and on information the client provided during the review. It reflects Safe Edges' assessment at the time of the review and is not a guarantee that the code is free of vulnerabilities.

Changes made after the reviewed commit, deployment configuration, off-chain components and third-party dependencies are outside the scope of this report unless stated otherwise.

Security is ongoing. We recommend continued testing, monitoring and bug bounty coverage alongside any audit.

This report is not investment advice and is not an endorsement of the project, its team or any token.



Safe Edges is a Web3 security firm and official security partner for the Fuel, ENI and XI EcoChain ecosystems. We work across smart contract audits, Rust and WASM reviews, ZK circuit review, AI and MCP security, and penetration testing.

200+

PROJECTS AUDITED

\$4B+

TVL SECURED

0

POST-AUDIT HACKS