Fractionalizer and SellNow

SECURITY ASSESSMENT REPORT

10 April, 2025

Prepared for

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the results of a security assessment conducted by CODESPECT for TokenTable. TokenTable is a token distribution platform that facilitates airdrops, vesting, and other mechanisms for distributing tokens.

This audit focuses on the newly introduced EVM contracts that govern the escrow process for buying and selling token allocations. These allocations can later be fractionalized, allowing multiple users to claim portions of a single token allocation.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of programs, execution testing.

c) Creation of test cases.

CODESPECT found 10 points of attention, one classified as `Critical`, four classified as `High`, one classified as `Medium`, three classified as `Low` and one classified as `Informational`. All of the issues are summarised in Table 2.

Organization of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the system overview.
- **Section 7** presents the issues.
- **Section 8** contains additional notes for the audit.
- **Section 9** discusses the documentation provided by the client for this audit.
- **Section 10** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
Critical	0	1	0
High	0	4	0
Medium	0	1	0
Low	0	2	1
Informational	0	0	1
Total	**0**	**8**	**2**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	TokenTable
Type of Project	Fractionalizer/Escrow
Duration of Engagement	3,5 Days
Duration of Fix Review Phase	2 Days
Draft Report	April 4, 2025
Final Report	April 10, 2025
Repository SellNow	tokentable-sellnow-swap-evm
Repository Fractionalizer	tokentable-unlocker-solana
Commit (SellNow Audit)	101eff63f094aafd9f5ea7bcd1025acba2e272f0
Commit (Fractionalizer Audit)	076d70abad50c2dddf8c5324c98d72f68ec6dc72
Commit (SellNow Final)	8aadc1d34c8e402996c240c993a1b75c465d13ab
Commit (Fractionalizer Final)	3f3a670c0f67983a0dd3acbec5e0eeee11e1806d
Documentation Assessment	Medium
Test Suite Assessment	Medium
Auditors	Talfao, Kalogerone

Table 3: Summary of the Audit

5.1 Scope - Audited Files

5.1.1 SellNow Scope

	File	LoC
1	ISellNow.sol	39
2	SellNow.sol	132
3	BuyerAggregator.sol	106
	Total	**277**

Scope information

The TokenTable team introduced changes to the SellNow code in commit `eafe0d070b48a51e65394fc7b6f4eb3928f9af12`, which were subsequently reviewed by the **CODESPECT** team.

5.1.2 Fractionalizer Scope

	File	LoC
1	ShareFractionalizer.sol	39
	Total	**277**

5.2 Findings Overview

	Finding	Severity	Update
1	`isApprovedForAll` mapping allows arbitrary transfer of any segment	Critical	Fixed
2	Aggregator does not approve token spending	High	Fixed
3	Fees not considered in `expectedAmount` inside `BuyerAggregator`	High	Fixed
4	Native ETH is not supported by `SellNow` contract	High	Fixed
5	`approved` segments don't reset their `approval` after being transferred	High	Fixed
6	DOS in `BuyerAggregator` 's `confirm` function	Medium	Fixed
7	Disallow aggregator withdrawals once the buyer side is confirmed	Low	Fixed
8	Improper handling of cancelled token allocations in Fractionalizer	Low	Acknowledged
9	Use `call(...)` instead of `transfer(...)`	Low	Fixed
10	Missing check for transferability	Info	Acknowledged

6 System Overview

TokenTable introduces a set of new EVM contracts that facilitate an escrow mechanism for buying and selling token allocations, referred to as `Actual`. Each `Actual` allocation is represented by an NFT called `FutureToken`, which can subsequently be fractionalized, allowing multiple users to claim portions of a single token allocation.

6.1 SellNow and BuyerAggregator

The `SellNow` contract functions as the core escrow contract, bridging the interaction between buyers and sellers. Sellers offer their token allocation—represented by a `FutureToken`—while buyers purchase the NFT using a predefined payment token, along with applicable fees payable to TokenTable.

Each buying/selling process is encapsulated within a session, created using the `createSession(...)` function. Only the contract owner is authorized to initiate a session.

```
function createSession(bytes32 sessionId, SwapConfig calldata config) external onlyOwner;
```

The session configuration specifies key participants (buyer and seller), the token allocation for sale, and a set of time-based constraints that govern the flow of the escrow process:

- `pricingTimeStamp`: After this deadline, the protocol owner can update the price of the `FutureToken`.
- `futureTokenDepositDeadline`: The deadline by which the seller must call `confirm(bytes32 sessionId)`. If `requireFutureTokenDeposit` is set to true, the seller must deposit the `FutureToken` into the escrow contract before the finalisation.
- `paymentDepositDeadline`: The deadline by which the buyer must confirm the session using `confirm(bytes32 sessionId)`.

When both parties confirm the session, the escrow is finalized: the `FutureToken` is transferred from the escrow contract to the buyer, and the buyer's payment (including fees) is distributed accordingly.

If the buyer fails to finalize the session before the `paymentDepositDeadline`, a fallback mechanism allows the seller to reclaim their `FutureToken` by calling the `cancel(...)` function—provided they were required to deposit the token beforehand.

The `BuyerAggregator` contract represents the buyer side of the escrow process in cases where multiple buyers are involved. It enables the aggregation of funds from several participants to collectively purchase a single token allocation.

Each `BuyerAggregator` instance is confirmed and approved by the TokenTable protocol. The allocation of allowed buyers and their respective contribution amounts is defined in the `buyerWhitelist` mapping.

Once the total required amount—covering the price of the `FutureToken` and associated protocol fees—has been deposited (in tokens or native ETH), anyone can trigger the `confirm()` function. This function internally calls `confirm(...)` on the `SellNow` contract, finalizing the buyer side of the escrow process.

6.2 ShareFractionalizer

The `ShareFractionaliser` contract enables the final step of fractionalizing a token allocation among multiple users. Once the escrow process is complete, the `FutureToken` is transferred from the `BuyerAggregator` to the `ShareFractionalizer` by TokenTable. Based on each buyer's contribution to the total payment, TokenTable configures segments representing proportional claims to the token allocation. Each segment corresponds to a share of the total allocation and may be updated by the protocol owner if necessary.

Segment owners can claim their portion of tokens by calling the `claim(...)` function. This function verifies whether a segment is eligible for claiming and, if so, internally calls the `claim(...)` function of the unlocker contract to retrieve the underlying tokens. The amount defined by the segment is then transferred to the segment owner.

```
function claim(uint256[] calldata _segmentIDs) external nonReentrant;
```

Segments are fully transferable. Owners can transfer them to other users via `transferFrom(...)` or authorize an operator to manage specific segments using `approve(...)`. However, operators cannot claim tokens on behalf of the user—they can only transfer ownership of the segments. Additionally, owners can grant blanket approval for all their segments to a specific operator using `setApprovalForAll(...)`.

```
function transferFrom(address from, address to, uint256[] calldata segmentIDs) external;

function approve(address operator, uint256[] calldata segmentIDs) external;

function setApprovalForAll(address operator, bool approved) external;
```

7 Issues

7.1 [Critical] `isApprovedForAll` mapping allows arbitrary transfer of any segment

File(s): `ShareFractionalizer.sol`

Description: The `setApprovalForAll(...)` function is intended to allow a trusted operator to manage *all* of a user's segments. While an operator cannot claim a user's segment (i.e., receive token distribution), they are allowed to transfer segments to arbitrary recipients. Since the user explicitly chooses the operator, this is assumed to be a trusted delegation.

However, the `transferFrom(...)` function contains a flawed access control check:

```
function transferFrom(address from, address to, uint256[] calldata segmentIDs) external {
    for (uint256 i = 0; i < segmentIDs.length; i++) {
        // has to be approved to transfer the segment
        require(
            operators[from][segmentIDs[i]] == _msgSender() || isApprovedForAll[from][_msgSender()],
            InvalidAccess() // @audit-issue: no relation to specific segmentID
        );
        // cannot transfer segments that are already claimed
        require(segments[segmentIDs[i]].claimableAmount != 0, SegmentAlreadyClaimed());
        // clear approval
        operators[from][segmentIDs[i]] = address(0);
        segments[segmentIDs[i]].recipient = to;
    }
}
```

The first `require` checks whether `msg.sender` is either:

- Approved for a specific segment via `operators[from][segmentID]`, or;
- Approved for **all** of the user's segments via `isApprovedForAll[from][msg.sender]`;

The issue is with the second condition: `isApprovedForAll[from][msg.sender]` is a global flag and does **not** validate that the segment in question actually belongs to `from`. This means `msg.sender` can transfer **any** segment—even if `from` is not the original owner of that segment—as long as the global approval flag is set.

Impact: An attacker can exploit this logic to transfer *any* active segment (i.e., those with non-zero claimable tokens) from *any* user who has granted them global approval. This effectively allows the theft of all future token distributions tied to those segments.

Recommendation: Modify the access control check to ensure that each `segmentID` being transferred is indeed owned by the `from` address.

Status: Fixed

Update from TokenTable: ecdf862821b03307ef07940a8c7097f8803cbdb7

7.2 [High] Aggregator does not approve token spending

File(s): `BuyerAggregator.sol`

Description: The `BuyerAggregator` contract collects funds from whitelisted buyers (in ETH or an ERC20 token) to facilitate the purchase of future tokens, which are later fractionalized via the `ShareFractionalizer`. The deposit token is defined by the `SellNow` session configuration. Once the required amount is collected, the `confirm()` function is called to finalize the process. This performs two key actions:

1. Locks the aggregator from accepting further deposits;
2. Calls `sellNow.confirm()` to proceed with the buyer-side confirmation;

However, when the `sellNow.confirm()` function is later called from the seller side, it will **revert** because the aggregator does **not approve** the `SellNow` contract to transfer the payment tokens on its behalf.

Since ERC20 transfers initiated by external contracts require explicit approval, the lack of an `approve(...)` call prevents the funds from being moved to the seller, halting the process.

Impact: Denial of service: the selling process cannot be completed because the aggregator has not authorized the `SellNow` contract to transfer the required tokens. As a result, the seller cannot receive their funds, and the finalisation of the session remains stuck.

Recommendation(s): Add an `approve(...)` call for the payment token (including any associated fees) to the `SellNow` before calling `sellNow.confirm()`

Status: Fixed

Update from TokenTable: 1b00e2c2e3daf853d320440668a812e075889ee7

7.3 [High] Fees not considered in `expectedAmount` inside `BuyerAggregator`

File(s): `BuyerAggregator.sol`

Description: The `BuyerAggregator` contract is used to collect funds from whitelisted buyers (in ETH or an ERC20 token) for the purpose of purchasing future tokens. The future tokens are later fractionalized via the `ShareFractionalizer`. The type of token to be deposited depends on the `SellNow` session configuration. Once the required amount is collected, the `confirm(...)` function is called to finalize the process. This action:

1. Locks the aggregator from further deposits;
2. Calls `sellNow.confirm(...)` to proceed with the buyer-side confirmation;

The issue lies in how the aggregator checks whether the required amount of tokens has been collected. In the `confirm(...)` function, the aggregator compares its current balance against `expectedAmount`, which is loaded from `sessionConfig.paymentToken.amount`. However, this value **does not include session fees**.

```
function confirm() external onlyBeforeConfirm {
  confirmed = true;
  address paymentToken = sessionConfig.paymentToken.tokenAddress;
  uint256 expectedAmount = sessionConfig.paymentToken.amount;
  require(
      paymentToken == address(0)
          ? address(this).balance == expectedAmount
          : IERC20(paymentToken).balanceOf(address(this)) == expectedAmount,
      IncorrectPaymentAmount()
  );
  sellNowInstance.confirm(sessionId);
}
```

As a result, when `sellNow.confirm(sessionId)` is called, the aggregator might not have enough funds to cover both the token price and the associated fees. This leads to a revert, even though the aggregator believes the full amount has been collected.

Impact: Denial of service of the selling process: the aggregator will be unable to confirm the session if fees are not accounted for in the `expectedAmount`.

Recommendation(s): Update the aggregator's `confirm(...)` logic to include any applicable session fees in the required balance check.

Status: Fixed

Update from TokenTable: 887d94a2a456bbece39074bf63ced2c33d203646

7.4 [High] Native ETH is not supported by `SellNow` contract

File(s): `SellNow.sol`

Description: The `BuyerAggregator` contract collects funds from whitelisted buyers (in ETH or an ERC20 token) to facilitate the purchase of future tokens, which are later fractionalized via the `ShareFractionalizer`. The type of token accepted is defined in the `SellNow` session configuration. Once the required amount is collected, the `confirm()` function is called to finalize the process. This function:

1. Locks the aggregator from accepting further deposits;
2. Calls `sellNow.confirm()` to proceed with the buyer-side confirmation;

As previously mentioned, the aggregator supports native ETH deposits. However, the `SellNow` contract is **not capable of receiving ETH** during the `confirm(...)` call. Moreover, due to the two-step confirmation process, it is not feasible to use `transferFrom` with native ETH—this method only works for ERC20 tokens.

This results in a fundamental incompatibility between ETH-based sessions and the payment mechanism in `SellNow`.

Furthermore, it is currently **not possible to establish a session using native ETH** as the payment token. In the aggregator, native ETH is represented as `address(0)`, but this configuration is not properly handled during session creation and results in an invalid setup.

Impact: Denial of service: ETH collected by the aggregator cannot be used to pay and finalize the session in the `SellNow` contract. As a result, the confirmation process fails, and the sale cannot be completed.

Recommendation(s): Introduce special handling for ETH payments. Two possible solutions:

- **Wrap ETH into WETH** within the aggregator before confirming, allowing compatibility with ERC20 logic;
- **Define a separate ETH-specific execution path** for `SellNow.confirm()` that accepts native ETH;

Status: Fixed

Update from TokenTable: 423c967dc3596720bb92bd258ad55007263dc888

Update from CODESPECT: The issue has been resolved. The TokenTable team has chosen to wrap the ETH to WETH. In case the seller does not confirm their part of the escrow process, an owner-only function `withdrawDepositOwner(...)` was introduced to allow withdrawal of these tokens.

This function also updates the payment token, which introduces a level of centralisation. However, this trade-off was accepted to prevent scenarios where ETH from deposits could remain stuck in the contract due to wrapping complications.

7.5 [High] `approved` segments don't reset their `approval` after being transferred

File(s): `ShareFractionalizer.sol`

Description: The `approve(...)` function gives permission to a trusted `operator` to be able to `transfer` the selected user's segments. This delegation of authority is considered secure because the user specifically selects who becomes an `operator` and for which of his `segments`. However, if the segments get transferred through `updateShareSegmentsAdmin(...)` by the owner or `transferShareSegments(...)` by the user, the `operator` of those segments remains the same.

The `transferFrom(...)` function doesn't account for this scenario, allowing an `operator` to transfer segments that he was previously approved for by the previous owner:

```
function transferFrom(address from, address to, uint256[] calldata segmentIDs) external {
    for (uint256 i = 0; i < segmentIDs.length; i++) {
        // has to be approved to transfer the segment
        require(
            operators[from][segmentIDs[i]] == _msgSender() || isApprovedForAll[from][_msgSender()], InvalidAccess()
        );
        // cannot transfer segments that are already claimed
        require(segments[segmentIDs[i]].claimableAmount != 0, SegmentAlreadyClaimed());
        // clear approval
        operators[from][segmentIDs[i]] = address(0);
        segments[segmentIDs[i]].recipient = to;
    }
}
```

The issue arises from the fact that the `operator` of a segment doesn't get reset when `updateShareSegmentsAdmin(...)` or `transferShareSegments(...)` is called.

Impact: A malicious user can always set himself as the `operator` of his segments and regain access to them in the case owner calls `updateShareSegmentsAdmin(...)` or possibly sells/has to `transfer` his segments to another address.

Recommendation(s): Set the `operator` to `address(0)` after changing a segment's recipient in `updateShareSegmentsAdmin(...)` and `transferShareSegments(...)`.

Status: Fixed

Update from TokenTable: 3f3a670c0f67983a0dd3acbec5e0eeee11e1806d

7.6 [Medium] DOS in `BuyerAggregator`'s `confirm` function

File(s): `BuyerAggregator.sol`

Description: Users are supposed to call the `confirm()` function in `BuyerAggregator` when they have collected the required amount to buy the NFT:

```
function confirm() external onlyBeforeConfirm {
    confirmed = true;
    address paymentToken = sessionConfig.paymentToken.tokenAddress;
    uint256 expectedAmount = sessionConfig.paymentToken.amount;
    require(
        paymentToken == address(0)
            ? address(this).balance == expectedAmount
            : IERC20(paymentToken).balanceOf(address(this)) == expectedAmount,
        IncorrectPaymentAmount()
    );
    sellNowInstance.confirm(sessionId);
}
```

However, there is a check that the contract's token balance must be equal to the `expectedAmount`. That means that malicious users can always send as little as `1 wei` to the contract to DOS this function call.

Impact: An attacker can DOS the `confirm()` function call by frontrunning the transaction and sending `1 wei` of the token. Even if users `withdrawDeposit(...)` to make the balance correct again, the malicious user can repeat the attack.

Recommendation(s): Make the check confirm that the contract's token balance is >= to the `expectedAmount`.

Status: Fixed

Update from TokenTable: 030d772345c18f08c79dfbab9dd9fffe89772763

7.7 [Low] Disallow aggregator withdrawals once the buyer side is confirmed

File(s): `BuyerAggregator.sol`

Description: The `BuyerAggregator` allows buyers to withdraw their deposits at any time. However, this feature can introduce issues and cause the `SellNow` confirmation process to revert if buyers withdraw their funds **after the aggregator has confirmed the buy side**, but **before the seller has confirmed** their side of the process.

Once the aggregator confirms the buyer side, withdrawals should be disallowed to preserve the integrity of the session. However, if the seller cancels or fails to confirm before a defined deadline, buyers should regain the ability to withdraw their deposits.

Impact: Buyers may withdraw funds after confirmation, potentially breaking the `SellNow` confirmation flow and leading to denial of the confirmation process.

Recommendation(s): Consider restricting buyer withdrawals after buy-side confirmation, while also handling the scenario where the seller fails to confirm within the expected timeframe.

Status: Fixed

Update from TokenTable: 0aa773198e15addcdd9d42637ccc596020e28d46

7.8 [Low] Improper handling of cancelled token allocations in Fractionalizer

File(s): `ShareFractionalizer.sol`

Description: The `actual` in the unlocker represents the token allocation, which is distributed over time. The allocation can be cancelled in two ways: either by wiping out the pending claimable amount to prevent future claims, or by preserving the pending claimable amount for future distribution.

This can potentially cause issues on the `Fractionalizer` claiming side. For example:

- An `actual` is configured to distribute 2000 tokens;
- The fractionalizer splits this into four segments: (500, 500) at time X and (500, 500) at time Y, assigned to User A and User B, respectively;
- Before reaching time X (in the X/2), the actual is cancelled with no wiping, meaning that only 500 tokens could be claimed;
- Due to timing, front-running, or delayed admin actions, the segment shares are not updated by the time X is reached;
- User A calls `claim(...)` and successfully receives their 500-token share;
- User B, however, is unable to claim, as the total distributable amount has been depleted by User A;

This results in an unfair scenario where one user receives their allocation while another does not, even though both were entitled to equal shares under the original distribution.

Impact: Unfair distribution of tokens could arise if such an event occurs, leading to inconsistent or biased claims depending on timing and order of execution.

Recommendation(s): Consider updating the logic to handle this scenario fairly, or require that segments are updated before any further claims can be processed after cancellation.

Status: Acknowledged

Update from TokenTable: very unlikely scenario, resolving it will significantly increase gas cost. Decide to not patch.

7.9 [Low] Use `call(...)` instead of `transfer(...)`

File(s): `BuyerAggregator.sol`

Description: The `BuyerAggregator` allows users to withdraw their deposited funds via the `withdrawDeposit(...)` function. When native ETH is used as the payment token, the contract sends ETH during withdrawal using the low-level `transfer(...)` call. This method has a fixed gas limit, which may cause withdrawals to revert when interacting with smart wallets or contracts that require more gas to accept ETH.

It is generally recommended to use the low-level `call(...)` to avoid such reverting scenarios. However, `call(...)` introduces a reentrancy risk and must be handled with caution. Furthermore, if you introduce the `call(...)`, after execution it returns a boolean value indicating success of the call rather than reverting on failure.

Impact: Withdrawals may revert for users with smart contract wallets.

Recommendation(s): - Consider replacing `transfer(...)` with a low-level `call(...)`, and ensure the return value is checked for success. If switching to `call(...)` is not preferred, consider adding an explicit `recipient` parameter to allow users to withdraw to an EOA, reducing the likelihood of failed transfers.

Status: Fixed

Update from TokenTable: 520ae0070e21837211885a4760636bfc6df53363

7.10 [Info] Missing check for transferability

File(s): `SellNow.sol`

Description: The `SellNow` contract allows establishing a session between a buyer and a seller for an `actual` (i.e., `futureToken`). During session creation, several parameters are validated. However, to ensure consistency and prevent potential issues, the `createSession(...)` function could additionally verify whether the `futureToken` is transferable.

Impact: If a session is created with a non-transferable `futureToken`, the `SellNow` functionality will operate incorrectly.

Recommendation(s): Consider adding a check to ensure the `futureToken` is transferable before allowing session creation.

Status: Acknowledged

Update from TokenTable: will be handled off-chain

8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- Conflicting `paymentDepositDeadline` timestamps in `SellNow.sol` contract (1, 2).
- No validation check that the arrays that are passed as input are of equal size in `ShareFractionalizer.sol`'s `setShareSegments(...)`, `transferShareSegments(...)` and `updateShareSegmentsAdmin(...)` functions.
- No validation check that `SwapConfig`'s `pricingTimeStamp`, `futureTokenDepositDeadline` and `paymentDepositDeadline` parameters are set in the future in `SellNow.sol` contract.

Update from TokenTable:

Update from CODESPECT: The TokenTable team fixed the third item regarding the timestamp checks. (d9acabf69982ff).

9 Evaluation of Provided Documentation

The TokenTable team provided documentation in a single format:

- **Natspec Comments:** The code includes comments for all processes, which explained the purpose of complex functionality in detail and facilitated understanding of individual functions.

The documentation provided by TokenTable while fundamental and basic in nature, was adequate given the limited scope of the audit. It met the necessary requirements and supported the key areas under review. However, the public technical documentation could be further improved to better present the protocol's overall functionality and facilitate the understanding of each component.

Additionally, the TokenTable team was consistently available and responsive, promptly addressing all questions raised by CODESPECT during the evaluation process.

10 Test Suite Evaluation

10.1 Compilation Output

SellNow's compilation output:

```
> forge compile
[] Compiling...
[] Compiling 61 files with Solc 0.8.28
[] Solc 0.8.28 finished in 3.23s
Compiler run successful!
```

Fractionalizer's compilation output:

```
> forge compile
[] Compiling...
[] Compiling 74 files with Solc 0.8.28
[] Solc 0.8.28 finished in 3.13s
Compiler run successful!
```

10.2 Tests Output

SellNow's test output:

```
> forge test
Ran 11 tests for test/SellNow.test.sol:SellNowTest
[PASS] testFuzz_ERC721ReceiverBuyer(uint64,uint256,uint256,uint256,bool) (runs: 256, : 537391, ~: 537336)
[PASS] testFuzz_NonERC721ReceiverBuyer(uint64,uint256,uint256,uint256,bool) (runs: 256, : 537171, ~: 537115)
[PASS] testFuzz_confirm(uint64,uint256,uint256,uint256,bool) (runs: 256, : 439559, ~: 439503)
[PASS] testFuzz_confirm_requireFutureTokenDeposit(uint64,uint256,uint256,uint256,bool) (runs: 256, : 461212, ~: 461158)
[PASS] testFuzz_createSession(uint64,uint256,uint256,uint256) (runs: 256, : 206753, ~: 206909)
[PASS] testFuzz_togglePause(bool) (runs: 256, : 32635, ~: 36722)
[PASS] testFuzz_withdrawFees(uint64,uint256,uint256) (runs: 256, : 472472, ~: 472496)
[PASS] test_confirm_RevertAlreadyConfirmed() (gas: 239516)
[PASS] test_confirm_RevertConfigInvalid() (gas: 22928)
[PASS] test_confirm_RevertWithNonERC721ReceiverBuyerWithoutBypass() (gas: 508751)
[PASS] test_setPaymentTokenAmount() (gas: 218206)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 164.74ms (717.76ms CPU time)

Ran 8 tests for test/BuyerAggregator.test.sol:BuyerAggregatorTest
[PASS] testFuzz_confirm(uint256) (runs: 256, : 591148, ~: 591188)
[PASS] testFuzz_deposit(uint256) (runs: 256, : 650336, ~: 650374)
[PASS] testFuzz_onERC721Received(address,address,uint256,bytes) (runs: 256, : 457432, ~: 458025)
[PASS] testFuzz_setWhitelist(uint256,uint256) (runs: 256, : 1188767, ~: 1078310)
[PASS] testFuzz_withdrawDeposit(uint256,uint256) (runs: 256, : 592417, ~: 592640)
[PASS] testFuzz_withdrawFutureToken(uint256) (runs: 256, : 653905, ~: 654637)
[PASS] testToken() (gas: 2392)
[PASS] test_setConfigOnce(bytes32) (runs: 256, : 442431, ~: 442431)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 177.55ms (658.31ms CPU time)

Ran 2 test suites in 178.35ms (342.29ms CPU time): 19 tests passed, 0 failed, 0 skipped (19 total tests)
```

Fractionalizer's test output:

```
> forge test
Ran 8 tests for test/ShareFractionalizer.t.sol:ShareFractionalizerTest
[PASS] test_approve() (gas: 1987106)
[PASS] test_claim() (gas: 2296407)
[PASS] test_initialize() (gas: 1750557)
[PASS] test_setApprovalForAll() (gas: 1771173)
[PASS] test_setShareSegments() (gas: 2087238)
[PASS] test_transferFrom() (gas: 2024952)
[PASS] test_transferShareSegments() (gas: 1874774)
[PASS] test_updateShareSegmentsAdmin() (gas: 1883221)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 13.49ms (25.73ms CPU time)

Ran 1 test suite in 16.26ms (13.49ms CPU time): 8 tests passed, 0 failed, 0 skipped (8 total tests)
```

10.3 Notes about Test suite

The TokenTable team delivered a test suite that includes coverage of the basic flows and functionalities. It also includes a variety of fuzzing tests. The use of fuzzing tests enabled coverage of numerous edge cases. These tests validate the behaviour of the system under a wide range of inputs, uncovering potential vulnerabilities or inconsistencies that could emerge in unexpected conditions. However, the test suite fails to cover complex and out of the box scenarios, contributing to a critical issue that was later discovered by **CODESPECT**.

CODESPECT also recommends explicitly defining strict invariants that the protocol must uphold. Incorporating tests to validate these invariants would ensure that critical assumptions about the system's behaviour are consistently maintained across all functionalities, further bolstering the protocol's security and stability.