Zellic

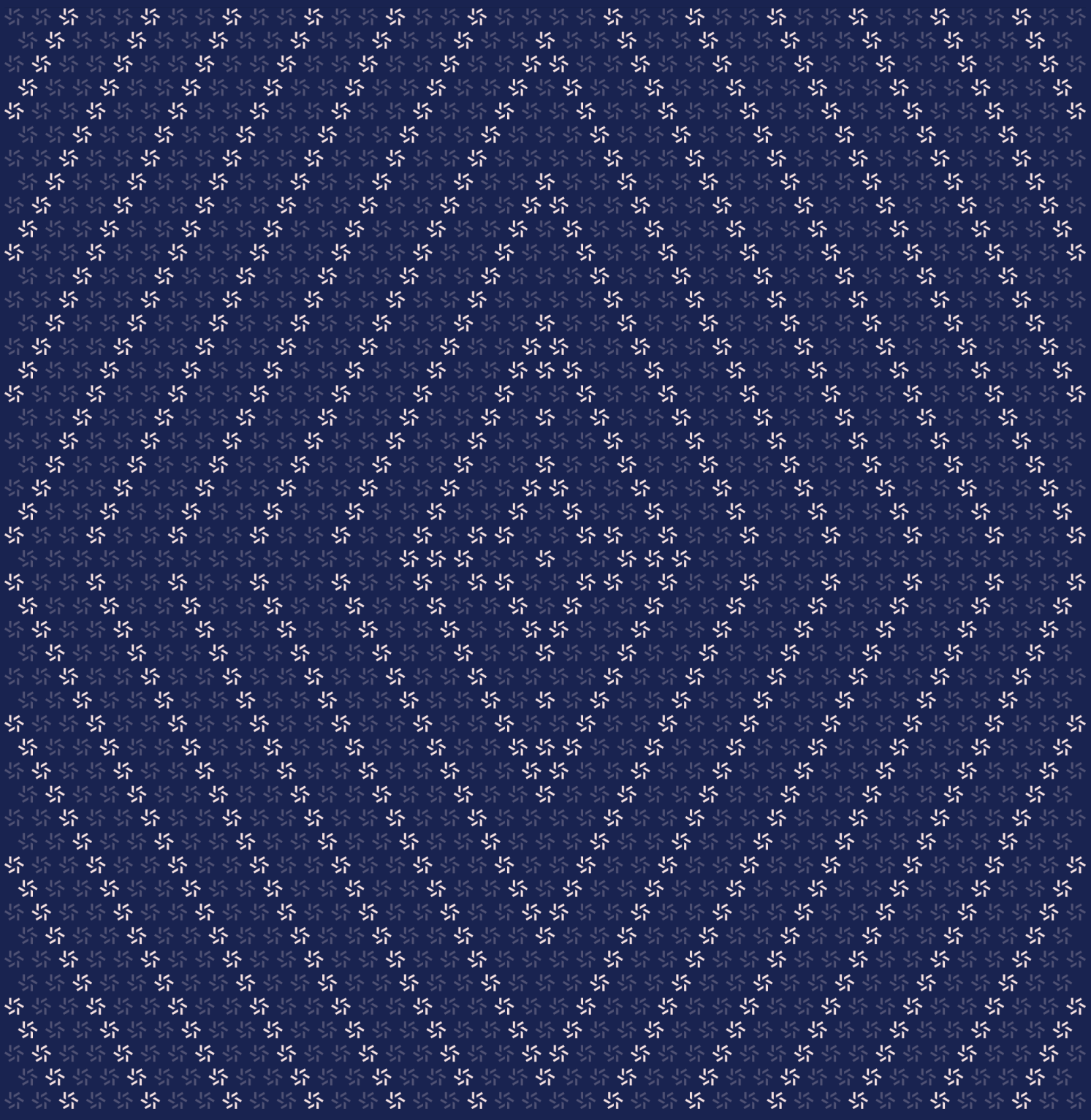
Prepared for
Sam (SAX)
SAX

Prepared by
Katerina Belotskaia
Malte Leip
Zellic

February 22, 2024

SAX

Smart Contract Security Assessment



Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for SAX from February 20th to February 22nd, 2024. During this engagement, Zellic reviewed SAX'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:

- Are there edge cases of a bonding curve when the contract runs out of liquidity as a result of an incorrectly calculated price?
- Is there a way for users to bypass access controls and create new hashtags when they should not be able to?
- Could a malicious user claim a share of the token twice?

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 SAX contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for SAX's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

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

2. Introduction

2.1. About SAX

SAX contributed the following description of SAX:

> Sax is a SocialFi project. Users are able to trade trending Twitter hashtags. As a hashtag starts trending, prices of that hashtag's token increase. The trending factor is what we call virality score, which is related to how many tweets, comments, and likes a hashtag has.

2.2. Methodology

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

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

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

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

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review 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.

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

2.3. Scope

The engagement involved a review of the following targets:

SAX Contracts

Repository	https://github.com/SAX-github/SAX-contracts ↗
Version	SAX-contracts: `4180c98c2a083caf96734d26c5c492bd3d108e34`
Programs	• BondingViralityController • TrendingERC20 • MerkleTokenShare
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 0.6 person-weeks. The assessment was conducted over the course of three calendar days.

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:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Malte Leip
Engineer
malte@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 20, 2024	Kick-off call
February 20, 2024	Start of primary review period
February 22, 2024	End of primary review period

3. Detailed Findings

3.1. Pricing model does not match claimed functionality

Target	Project		
Category	Business Logic	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

The BondingViralityController contract allows users to buy and sell tokens associated to a hashtag. The description of the project suggests that the price of the token increases when the hashtag's virality score increases. However, when the virality score increases, only the *buy price* (the price at which users can buy the token from the contract) increases accordingly, while the *sell price* (the price at which users can sell their tokens to the contract) decouples from the virality score and depends instead only on the value locked in the contract allocated to this token (the liquidity) and the token supply. In particular, when buying after a rise in virality, the buyer's position will be at an immediate loss, as sell price will be below buy price. Further increases in the virality score will not change the sell price; hence, such a buyer will be entirely dependent on investments by further buyers (facing the same conditions) in order to break even or make a profit.

Terminology and assumptions

We will now explain this in more detail. While every buy/sell costs a 5% fee, we will ignore this in the following explanation (that is, pretend that the fee is 0%), as the fee only makes it more difficult for users to break even or make a profit. Another thing we will ignore is rounding errors. We only consider a single hashtag token, so when we refer to, for example, the contract's liquidity, this will always implicitly mean the contract's liquidity for the token under consideration.

A token's supply is how many tokens have been minted and are in circulation. The initial supply is 10,000 tokens. Users can buy and sell tokens from and to the contract, paying and receiving payment in a payment token P such as USDB. The contract's balance of P held with regards to the hashtag token (i.e., the amount the contract made from users buying tokens, minus the amount the contract paid out when users sell tokens back to the contract) is called the liquidity.

Base price

Buy and sell prices are calculated by the contract's `getBuyPrice` and `getSellPrice` functions, respectively. Both use the `getPrice` function internally. We will call the three prices the *buy price*, *sell price*, and *base price*, respectively. The `getPrice` function is defined as follows:

```
function getPrice(uint256 newSupply, uint16 score)
    public view returns (uint256) {
    uint256 bondingCurvePrice = initialPrice * (newSupply ** exponent)
    / (initialSupply ** exponent);
    return bondingCurvePrice * score;
}
```

Here, `initialPrice`, `initialSupply`, and `exponent` are constants, with the latter having a value of 3. The base price is thus of the form $C \cdot score \cdot supply^3$, where C is some constant.

Buy price

The buy price is then defined by the following function:

```
function getBuyPrice(address token, uint256 amount)
    public view returns (uint256) {
    HashtagData memory data = registeredTokens[token];

    uint256 supplyCeiling = data.supply + amount;
    uint256 i = data.supply;
    uint256 totalPrice;

    while (i < supplyCeiling) {
        unchecked { i++; }

        totalPrice += getPrice(i, data.viralityScore);
    }

    return totalPrice;
}
```

Thus, if the current supply is S, then buying one token, which would bring the supply to $S + 1$, will cost exactly `getPrice(S+1, viralityScore)` — so $C \cdot viralityScore \cdot (S + 1)^3$. Buying multiple tokens at once costs the same as buying them one at a time (recall that we are assuming transaction fees are zero here). We can summarize this as buying a token costing the token's base price, and this price is proportional to the virality score.

Sell price

Let us now consider the sell price:

```
function getSellPrice(address token, uint256 amount)
    public view returns (uint256) {
```

```
    HashtagData memory data = registeredTokens[token];
    if (data.supply - amount < initialSupply) revert NotAllowed();

    uint256 supplyFloor = data.supply - amount;

    uint256 i = initialSupply;
    uint256 totalPrice;

    uint256[] memory prices = new uint256[](amount);

    uint256 j;
    while (i < data.supply) {
        uint256 price = getPrice(i+1, data.viralityScore);

        if (i >= supplyFloor && i < data.supply) {
            prices[j] = price;
            unchecked { j++; }
        }

        if (i < data.supply) {
            totalPrice += price;
        }

        unchecked { i++; }
    }

    // account for contract liquidity
    uint256 finalTotalPrice;
    uint256 k = 0;
    while (k < j) {
        uint256 pctAllocation = prices[k] * PCT_PRECISION / totalPrice;
        uint256 allocationPrice = data.liquidity * pctAllocation
    / PCT_PRECISION;

        finalTotalPrice += prices[k] < allocationPrice ? prices[k] :
    allocationPrice;
        unchecked { k++; }
    }

    return finalTotalPrice;
}
```

Let us denote the initial supply by S_0, the supply after selling S_1, and the current supply S_2. Further-

more, we denote by v the current virality score and by L, liquidity. Then we will have

$$\begin{aligned}
totalPrice &= \sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \\
prices[k] &= C \cdot v \cdot (S_1 + 1 + k)^3 \\
allocationPrice[k] &= L \cdot prices[k] / totalPrice \\
&= L \cdot C \cdot v \cdot (S_1 + 1 + k)^3 \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \right)^{-1} \\
finalTotalPrice &= \sum_{k=0}^{S_2 - S_1 - 1} \min\left(prices[k], allocationPrice[k]\right) \\
&= \sum_{k=0}^{S_2 - S_1 - 1} \min\Bigg(C \cdot v \cdot (S_1 + 1 + k)^3, \\
&\qquad L \cdot C \cdot v \cdot (S_1 + 1 + k)^3 \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \right)^{-1} \Bigg) \\
&= \sum_{k=0}^{S_2 - S_1 - 1} C \cdot v \cdot (S_1 + 1 + k)^3 \cdot \min\left(1, L \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \right)^{-1} \right) \\
&= \min\left(1, L \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \right)^{-1} \right) \cdot \left(\sum_{k=0}^{S_2 - S_1 - 1} C \cdot v \cdot (S_1 + 1 + k)^3 \right) \\
&= \min\left(1, L \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot v \cdot S^3 \right)^{-1} \right) \cdot \left(\sum_{S=S_1+1}^{S_2} C \cdot v \cdot S^3 \right)
\end{aligned}$$

Let us define $v_L = L \cdot \left(\sum_{S=S_0+1}^{S_2} C \cdot S^3 \right)^{-1}$. Note that if so far, there were no sells for this token, and all tokens have been bought when the virality score was v', then we would have $v_L = v'$. With this definition of v_L, we obtain the following, continuing from the calculations above.

$$finalTotalPrice = \min\left(1, v_L \cdot v^{-1}\right) \cdot \left(\sum_{S=S_1+1}^{S_2} C \cdot v \cdot S^3 \right)$$

This leaves us two cases. If $v_L \geq v$, we obtain

$$finalTotalPrice = \sum_{S=S_1+1}^{S_2} C \cdot v \cdot S^3$$

Thus, if $v_L \geq v$, the sell price for a token is the same as the buy price.

If instead $v_L < v$, then we obtain

$$finalTotalPrice = v_L \cdot v^{-1} \cdot \left(\sum_{S=S_1+1}^{S_2} C \cdot v \cdot S^3 \right)$$
$$= \sum_{S=S_1+1}^{S_2} C \cdot v_L \cdot S^3$$

Thus, if $v_L < v$, then the sell price will otherwise be independent of v. Instead, the sell price will be what the buy price would be at the lower virality score v_L instead of at the actual virality score v. This implies that when the virality score v is rising, as soon as it rises above v_L, the sell price will decouple from buy price and remain constant (supply staying equal), while buy price continues to rise proportionally with v.

A concrete example

Let us consider a concrete scenario. Let us assume that virality was 100 so far, and 10,000 tokens have been bought by users in total, so that (including the initial supply of 10,000) the supply is now 20,000. Note that we can set the constant C to a positive value of our choosing without loss of generality (different values amount only to different scalings of the denomination of prices in P). We choose $C = 10^{-14}$ here to get numbers that are easy to state.

If user A bought the last token (the one that made the supply go from 19,999 to 20,000), then they paid 8 for it. Assume that now virality increases to 1,000. Thus now $v = 1000$, but we still have $v_L = 100$, so the sell price will not change, while the buy price will be 10 times as high as before. Selling now, user A would thus only be paid 8.00, making no profit. To make a profit, user A is dependent on v_L increasing. As $v > v_L$, this will happen if the total supply increases — so if other users buy at this higher virality v.

Let us now consider a possible user B that buys a token in this situation. Buying a token now will cost 80.01. If user B were to sell that token again immediately after buying it, they would still only get paid 8.02 for it (the ratio between buy and sell price is not quite 10 anymore, as user B buying at a virality score higher than v_L increased v_L a little). User B's position will thus start out at a loss of 89.98%.

Under what conditions would user B be able to break even when selling again? At the current liquidity, this will be impossible no matter the virality score; even if the virality score would rise further, the sell price would be stuck in the second case where it is independent of the virality score. If the virality score were to fall enough for the sell price to be calculated according to the first case, then this would only imply a sell price even lower than 8.02. To break even or to make a profit, user B is thus dependent on liquidity increasing. Concretely, assuming that virality stays at 1,000, the current liquidity of 37,583.51 would have to rise to 465,452.93 for user B to not sell at a loss. The multiplicative increase in liquidity required is thus around 12.38. It would not help user B much if the virality score would increase further. Assuming the virality score increases tenfold another time to 10,000, then liquidity would have to rise to 385,245.39 for user B to not sell at a loss. The multiplicative increase in liquidity required here is around 10.25. In fact, if $v > v_L$, then buys will never be able to bring v_L to $v_L \geq v$, so sell price in the considered scenario where virality increases further after user B's buy will always be calculated according to the second case. Let r be the ratio of user B's buy price to user

B's sell price if immediately selling again, which is about $r \approx 80.01/8.02 \approx 9.98$. Then we will show in the next subsection that user B will only be able to break even if liquidity increased by a factor of at least r.

To calculate the concrete values mentioned above, we wrote a script reproduced in the last two subsections of this section.

Bounding liquidity required to break even

We assume that the initial supply was S_0, the supply is now S_1, and user B bought the token that raised the supply to S_1. The virality score is v, liquidity at this point is L_1, and v_{L_1} is as defined before. We consider the situation where $v > v_{L_1}$. Let r be the ratio between user B's buy price and the price at which they could immediately sell again. Concretely, r is the following.

$$\begin{aligned} r &= \left(C \cdot v \cdot S_1^3\right) \cdot \left(C \cdot v_{L_1} \cdot S_1^3\right)^{-1} \\ &= v \cdot v_{L_1}^{-1} \end{aligned}$$

Now suppose that further buys happened, bringing supply to S_2 and liquidity to L_2, with corresponding new v_{L_2}. We do not assume that the virality score stayed the same. If B sells their token now, their sell price will satisfy the following.

$$sellPrice \leq C \cdot v_{L_2} \cdot S_2^3$$

User B's buy price was instead

$$buyPrice_B = C \cdot v \cdot S_1^3$$

For user B to break even, we must have $sellPrice \geq buyPrice_B$. We are interested in the minimum liquidity L_2 at which this is possible. We obtain

$$\begin{aligned} & sellPrice \geq buyPrice_B \\ \implies & C \cdot v_{L_2} \cdot S_2^3 \geq C \cdot v \cdot S_1^3 \\ \iff & v_{L_2} \cdot S_2^3 \geq v \cdot S_1^3 \\ \iff & v_{L_2} \cdot S_2^3 \geq r \cdot v_{L_1} \cdot S_1^3 \end{aligned}$$

Note that S_2 must be strictly bigger than S_1, as $S_2 = S_1$ would imply no buys happened, and hence $v_{L_2} = v_{L_1}$, and then the above inequality would contradict $r > 1$. We continue from above:

$$\begin{aligned} & \iff \frac{L_2}{C \cdot \sum_{S=S_0+1}^{S_2} S^3} \cdot S_2^3 \geq r \cdot \frac{L_1}{C \cdot \sum_{S=S_0+1}^{S_1} S^3} \cdot S_1^3 \\ & \iff \frac{L_2}{L_1} \geq r \cdot \frac{S_1^3}{S_2^3} \cdot \frac{\sum_{S=S_0+1}^{S_2} S^3}{\sum_{S=S_0+1}^{S_1} S^3} \end{aligned}$$

What we want to bound below is exactly $\frac{L_2}{L_1}$. We thus obtain the following chain of inequalities. We use that $\sum_{S=0}^{T} S^3 = \left(\frac{T\cdot(T+1)}{2}\right)^2$, and that $a < b \leq c$ implies $0 < b-a \leq c-a$ and hence $\frac{c-a}{b-a} \geq 1$.

$$\begin{aligned}
\frac{L_2}{L_1} &\geq r \cdot \frac{S_1^3}{S_2^3} \cdot \frac{\sum_{S=S_0+1}^{S_2} S^3}{\sum_{S=S_0+1}^{S_1} S^3} \\
&\geq r \cdot \frac{S_1^3}{S_2^3} \cdot \frac{(S_2\cdot(S_2+1))^2 - (S_0\cdot(S_0+1))^2}{(S_1\cdot(S_1+1))^2 - (S_0\cdot(S_0+1))^2} \\
&\geq r \cdot \frac{S_1^3}{S_2^3} \cdot \frac{(S_2\cdot(S_2+1))^2}{(S_1\cdot(S_1+1))^2} \\
&\geq r \cdot \frac{S_1^3}{S_2^3} \cdot \frac{S_2^4 + 2\cdot S_2^3 + S_2^2}{S_1^4 + 2\cdot S_1^3 + S_1^2} \\
&\geq r \cdot \frac{S_2 + 2 + S_2^{-1}}{S_1 + 2 + S_1^{-1}}
\end{aligned}$$

Now as $S_2 > S_1$ as we remarked above, and both are natural numbers, we must have $S_2 \geq S_1 + 1$, and hence we can conclude $S_2 + 2 + S_2^{-1} \geq S_1 + 2 + S_1^{-1}$. Thus, we can conclude

$$\frac{L_2}{L_1} \geq r$$

So if user B bought the token in a situation where the buy price was $r > 1$ times the sell price, then user B will only be able to break even on selling this token when the liquidity multiplied by a factor of at least r. We emphasize that this holds no matter how the virality score changes after user B bought their token.

Script to calculate example values

The following SageMath script was used to calculate the values for the concrete example given above.

```
#!/usr/bin/env sage

import random

INITIAL_SUPPLY = 10_000
INITIAL_VIRALITY = 100
SUPPLY = 20_000
VIRALITY = 1000
VIRALITY_LAST = 10000
PRICE_CONSTANT = 10**(-14)
```

```
def sum_of_cubes(x):
  "1^3 + ... + x^3"
  return ((x*(x+1))/2)^2

def test_sum_of_cubes():
  x = random.randrange(100, 1000)
  correct = sum([i**3 for i in range(x+1)])
  assert sum_of_cubes(x) == correct

test_sum_of_cubes()

def getPrice(supply, score):
  return float(PRICE_CONSTANT * score * (supply**3))

def sellPrice(supply, score, liquidity):
  price_from_buy = getPrice(supply, score)
  price_from_liquidity = (liquidity * (supply**3)) / (sum_of_cubes(supply)
    - sum_of_cubes(INITIAL_SUPPLY))
  return min([price_from_buy, price_from_liquidity])

def liquidity_after_buys(supply_before, supply_after, score, liquidity):
  supply = supply_before
  while supply < supply_after:
    supply += 1
    liquidity += getPrice(supply, score)
  return liquidity

def solve_supply(sell_price, current_supply, current_liquidity, score):
  S = var('S')
  # We want
  # (liquidity * (S**3)) / (sum_of_cubes(S) - sum_of_cubes(INITIAL_SUPPLY)) >=
    sell_price
  # So look for S with
  # (liquidity * (S**3)) = sell_price * (sum_of_cubes(S) -
    sum_of_cubes(INITIAL_SUPPLY))
  # Note liquidity also depends on S.
  liquidity = current_liquidity + PRICE_CONSTANT * score * (sum_of_cubes(S)
    - sum_of_cubes(current_supply))
  lhs = liquidity * (S**3)
  rhs = sell_price * (sum_of_cubes(S) - sum_of_cubes(INITIAL_SUPPLY))
  S_by_solving = (lhs == rhs).find_root(0, 10**6)
  S_by_solving = ceil(S_by_solving)

  #S = current_supply + 1
  #while sellPrice(S, score, liquidity_after_buys(current_supply, S, score,
    current_liquidity)) < sell_price:
  # S += 1
```

```
  #S_by_trying = S
  #assert S_by_solving == S_by_trying

  return S_by_solving


liquidity = liquidity_after_buys(INITIAL_SUPPLY, SUPPLY, INITIAL_VIRALITY, 0)
print(f'Virality so far was {INITIAL_VIRALITY}')
print(f'Supply is {SUPPLY:,}, and liquidity {liquidity:,.2f}')
user_A_buy_price = getPrice(SUPPLY, INITIAL_VIRALITY)
print(f'User A bought the last token, for {user_A_buy_price:.2f}')

print(f'\nNow virality is {VIRALITY:,}.')
print(f'If user A were to sell now, they would get {sellPrice(SUPPLY, VIRALITY,
    liquidity):.2f}. Thus they would not make a profit yet, even though
    virality multiplied by 10.')
print(f'To obtain a sale price n times the price they bought at, user A would
    have to wait until liquidity increased by a factor of this:')
for n in range(2, 11):
  S = solve_supply(n*user_A_buy_price, SUPPLY, liquidity, VIRALITY)
  L = liquidity_after_buys(SUPPLY, S, VIRALITY, liquidity)
  print(f'{n}: {L / liquidity:.2f}')

user_B_buy_price = getPrice(SUPPLY+1, VIRALITY)
liquidity += user_B_buy_price
print(f'\nIf user B buys a token now, it costs them {user_B_buy_price:.2f}')
user_B_sell_price = sellPrice(SUPPLY+1, VIRALITY, liquidity)
print(f'If they were to immediately sell again, they would get
    {user_B_sell_price:.2f}. Immediately after buying, their position is thus
    at a loss of {(1 - user_B_sell_price / user_B_buy_price)*100:.2f}%.')
user_B_breakeven_supply = solve_supply(user_B_buy_price, SUPPLY + 1,
    liquidity, VIRALITY)
user_B_breakeven_liquidity = liquidity_after_buys(SUPPLY + 1,
    user_B_breakeven_supply, VIRALITY, liquidity)
#print(sellPrice(user_B_breakeven_supply, VIRALITY,
    user_B_breakeven_liquidity))
print(f'Liquidity would have to rise from {liquidity:,.2f} to
    {user_B_breakeven_liquidity:,.2f} for user B to not sell at a loss. The
    multiplicative increase in liquidity required is thus
    {user_B_breakeven_liquidity / liquidity:.2f}.')

print(f"\nIt would not help user B's position much if virality would increase
    further.")
print(f'Assume virality would increase to {VIRALITY_LAST:,}.')
user_B_sell_price2 = sellPrice(SUPPLY+1, VIRALITY_LAST, liquidity)
print(f'If user B were to sell now, they would again only get
    {user_B_sell_price2:.2f}. Their position is thus still at a loss of
```

```
    {(1 - user_B_sell_price2 / user_B_buy_price)*100:.2f}%.')
user_B_breakeven_supply2 = solve_supply(user_B_buy_price, SUPPLY + 1,
    liquidity, VIRALITY_LAST)
user_B_breakeven_liquidity2 = liquidity_after_buys(SUPPLY + 1,
    user_B_breakeven_supply2, VIRALITY_LAST, liquidity)
print(f'Liquidity would have to rise from {liquidity:,.2f} to
    {user_B_breakeven_liquidity2:,.2f} for user B to not sell at a loss. The
    multiplicative increase in liquidity required is thus
    {user_B_breakeven_liquidity2 / liquidity:.2f}.')
```

Output of the script

The output of the SageMath script reproduced in the preceeding section is as follows.

```
Virality so far was 100
Supply is 20,000, and liquidity 37,503.50
User A bought the last token, for 8.00

Now virality is 1,000.
If user A were to sell now, they would get 8.00. Thus they would not make a
    profit yet, even though virality multiplied by 10.
To obtain a sale price n times the price they bought at, user A would have to
    wait until liquidity increased by a factor of this:
2: 2.06
3: 3.18
4: 4.35
5: 5.58
6: 6.85
7: 8.18
8: 9.54
9: 10.95
10: 12.41

If user B buys a token now, it costs them 80.01
If they were to immediately sell again, they would get 8.02. Immediately after
    buying, their position is thus at a loss of 89.98%.
Liquidity would have to rise from 37,583.51 to 465,452.93 for user B to not sell
    at a loss. The multiplicative increase in liquidity required is thus 12.38.

It would not help user B's position much if virality would increase further.
Assume virality would increase to 10,000.
If user B were to sell now, they would again only get 8.02. Their position is
    thus still at a loss of 89.98%.
Liquidity would have to rise from 37,583.51 to 385,245.39 for user B to not sell
    at a loss. The multiplicative increase in liquidity required is thus 10.25.
```

Recommendations

Consider using a different pricing model.

Remediation

This issue has been acknowledged by SAX.

3.2. Erroneous token transfer direction in the `UpdateTokenShares` function

Target	BondingViralityController		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `updateTokenShares` function is intended to be called by the contract owner to update the Merkle root representing users' percentage claims to a token, list tokens available for claim, and claimable amounts. However, due to a mistake, this function tries to transfer tokens from the contract to the owner, instead of allowing the owner to send tokens to the contract for users to claim.

```
function updateTokenShares(bytes32 merkleRoot, address[] calldata tokens,
    uint256[] calldata amounts)
    external
    onlyOwner
{
    if (merkleRoot.length == 0 || tokens.length == 0 || tokens.length !=
    amounts.length) revert InvalidInput();

    uint256 length = tokens.length;
    uint256 i;
    while (i < length) {
        IERC20(tokens[i]).safeTransfer(msg.sender, amounts[i]);

        unchecked { i++; }
    }

    _setTokenShares(merkleRoot, tokens, amounts);
}
```

Impact

This functionality does not work because, as a result of an error, the owner cannot send tokens to the contract so that users can withdraw them later.

Recommendations

Modify the `updateTokenShares` function to transfer tokens from `msg.sender` to the contract address as shown below:

```
function updateTokenShares(bytes32 merkleRoot, address[] calldata tokens,
    uint256[] calldata amounts)
        external
        onlyOwner
    {
        ...
            IERC20(tokens[i]).safeTransferFrom(msg.sender, address(this),
    amounts[i]);
        ...
```

Remediation

This issue has been acknowledged by SAX, and a fix was implemented in commit b5651803 ↗.

3.3. The `_toLower` function incorrectly handles the Unicode characters

Target	BondingViralityController		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The internal `_toLower` function converts the `hashtag` string from uppercase to lowercase. This function accurately processes only ASCII characters in the range of A to Z. However, the input string can contain Unicode symbols, where the character set is significantly broader and includes not only the Latin alphabet. So in the case where an input string contains characters outside the specified range, the function will not process them as expected and the characters will remain unchanged.

```
function _toLower(string memory str) internal pure returns (string memory) {
    bytes memory bStr = bytes(str);
    bytes memory bLower = new bytes(bStr.length);
    for (uint i = 0; i < bStr.length; i++) {
        // Uppercase character...
        if ((uint8(bStr[i]) >= 65) && (uint8(bStr[i]) <= 90)) {
            // So we add 32 to make it lowercase
            bLower[i] = bytes1(uint8(bStr[i]) + 32);
        } else {
            bLower[i] = bStr[i];
        }
    }
    return string(bLower);
}
```

Impact

If the input `hashtag` string contains non-Latin characters, they will not be reduced to lowercase and will remain unchanged after processing with the `_toLower` function. Therefore, other functionality of this contract can be violated by the incorrect string handling.

Recommendations

Consider limiting the input character set to only the required ASCII characters if the project does not assume that users can use the extended Unicode character set.

Remediation

This issue has been acknowledged by SAX, and a fix was implemented in commit dfb4e3f1 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unused and redundant fields of structures

The `HashtagData.lastPrice` is not used in the contract BondingViralityController and can be deleted. Also, the `BatchClaimTokensInput.index` from the MerkleTokenShare contract is used to calculate the hash when verifying the proof, but in fact it is redundant and can be deleted without consequences to the functionality and security of the contract.

The `HashtagData.lastPrice` was removed in commit 7690a193 ↗.

4.2. Checks-effects-interactions pattern

We recommend following the checks-effects-interactions pattern in the `buyTokens()` and `sellTokens()` functions by changing the state of the contract before calling the external contract. Although we did not identify any reentrancy attacks, it is a best practice to prioritize security and prevent potential future attacks.

```
function buyTokens(address token, uint256 amount)
    public whenNotPaused onlyRegistered(token) {
        ...
        IERC20(paymentToken).safeTransferFrom(msg.sender, address(this),
    totalPrice);

        TrendingERC20(token).mint(msg.sender, amount);

        registeredTokens[token].supply += amount;
        feesEarned[owner()] += protocolFee;
        registeredTokens[token].liquidity += price;
        ...
    }

    function sellTokens(address token, uint256 amount)
    public whenNotPaused onlyRegistered(token) {
        ...
        TrendingERC20(token).burn(msg.sender, amount);
        IERC20(paymentToken).safeTransfer(msg.sender, totalPrice);

        registeredTokens[token].supply -= amount;
        feesEarned[owner()] += protocolFee;
        registeredTokens[token].liquidity -= price;
```

```
        ...
    }
```

The recommended changes were implemented in commit f17601a0 ↗.

4.3. Redundant if conditions in the `getSellPrice` function

The while condition guarantees that the `i` value will be less than `data.supply`, so the other internal `i < data.supply` checks are redundant.

```
function getSellPrice(address token, uint256 amount)
    public view returns (uint256) {
    ...
    while (i < data.supply) {
        uint256 price = getPrice(i+1, data.viralityScore);

        if (i >= supplyFloor && i < data.supply) {
            prices[j] = price;
            unchecked { j++; }
        }

        if (i < data.supply) {
            totalPrice += price;
        }

        unchecked { i++; }
    }
...
}
```

The second redundant check was removed in commit 16811766 ↗.

4.4. Centralization risk

There are three types of privileged accounts for the BondingViralityController contract:

- The owner
- The viralityScoreUpdater
- The users on the allowlist

The users on the allowlist can create new tokens but have otherwise no special powers.

The viralityScoreUpdater is able to call `updateViralityScores` to update the virality score for any registered token. An attacker gaining access to the viralityScoreUpdater could use this to drain all

liquidity in the contract; by setting virality to zero, buying a lot of tokens at zero cost, setting virality to a high value, and then selling their tokens again, they can extract (part of) the liquidity held by the contract for each token. By repeating this, all liquidity can be drained.

The owner has the same powers as the viralityScoreUpdater but can additionally claim and withdraw different types of fees/yield and set the viralityScoreUpdater and various other settings. For example, the owner can use the `togglePaused` function to pause and unpause contract state to enable or disable trading. This can lead to the blocking of user funds if it is impossible to sell tokens.

The owner could also use `updateTokenShares` to drain all liquidity from the contract by submitting the payment token as the token and a Merkle tree that includes `shareBbps` adding up to more than `BPS_MAX`, all made out to the owner. See also 4.8. ↗ for background on this.

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system. We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

4.5. Initial mint sellable

When new tokens are created, an initial supply of 10,000 tokens is minted. When selling tokens, it is not possible to sell tokens when supply would fall below this initial amount:

```
function getSellPrice(address token, uint256 amount)
    public view returns (uint256) {
    HashtagData memory data = registeredTokens[token];
    if (data.supply - amount < initialSupply) revert NotAllowed();
    // ...
}
```

However, the initial mint of 10,000 is allocated to the creator of the token and can be sold as any other:

```
function createToken(string calldata hashtag, address recipient,
    uint16 viralityScore) external {
    // ...
    // create the token + mint the inital supply; this contract can mint/burn
    TrendingERC20 token = new TrendingERC20(address(this), hashtag, hashtag);
    token.mint(recipient, initialSupply);
    // ...
}
```

This means that it is possible for users to buy tokens while later not being able to sell them. If, for

example, after the initial creation another 10,000 tokens get sold by the contract to users, and then the creator of the token sells their 10,000 tokens to the contract, then none of the users will be able to sell their token. To avoid this, consider not allocating the initial supply to any user.

4.6. Token creator can set initial virality score

When creating a new token, the caller can set the initial virality score:

```
function createToken(string calldata hashtag, address recipient,
    uint16 viralityScore) external {
    // ...
    registeredTokens[address(token)] = HashtagData({
        supply: initialSupply,
        hashtag: hashtag,
        viralityScore: viralityScore,
        lastUpdatedAt: block.timestamp,
        lastPrice: initialPrice,
        liquidity: 0
    });
    // ...
}
```

Note that after `allowCreateTokens` is set, anyone can call this function and create a new token. If the caller sets `viralityScore` to zero, they will be able to buy an arbitrary amount of tokens at no cost. Damage to other users would not materialize if they do not interact with this token, recognizing that the token creator used this to allocate a large amount of tokens for themselves. This still amounts to a squatting issue in that case, as only one token can be created per lowercased hashtag.

In commit `0135f3b5` ↗, `createToken` was changed to set `viralityScore` of a newly created token to a fixed value rather than a caller-supplied value.

4.7. Incomplete test coverage

Certain aspects of the codebase, particularly those related to shares functionality, remain untested. This presents some uncertainties about the reliability and performance of these specific features. To enhance confidence in the system, it would be beneficial to conduct comprehensive testing for these unverified components.

4.8. Claimed token shares might not sum up to total amount

When claiming tokens, the amount transferred could have been rounded down:

```
function claimTokens(
    bytes32[] calldata proof,
    uint256 epoch,
    uint256 index,
    uint256 tokenIdx,
    uint16 shareBbps
) public {
    // ...
    uint256 amount = share.amounts[tokenIdx] * shareBbps / BPS_MAX; //
    Rounding down can happen at this division
    IERC20(token).safeTransfer(recipient, amount);
    // ...
}
```

If this happens, then some amount of the token will be left behind in the contract after all shares have been claimed, even if the `shareBbps` for all shares add up to `BPS_MAX`.

There is also no check that the `shareBbps` for all shares add up to `BPS_MAX`, so this is something the owner has to ensure when constructing a call to `updateViralityScores`. Note that this could be used to recover stuck tokens due to rounding as mentioned above, by adding an extra share amounting to the stuck tokens. If the sum of `shareBbps` for all shares add up to more than `BPS_MAX` and the contract has an insufficient balance to cover all claims, the call to `claimTokens` will revert when the remaining balance is insufficient for the claim.

In commit e3890a6d ↗, `claimTokens` was converted to operate with absolute share amounts rather than ratios in basis points, thereby avoiding the rounding issue.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: BondingViralityController.sol

Function: `buyTokens(address token, uint256 amount)`

The function allows the caller to buy tokens associated with a specific hashtag. The current buy price depends on the current `viralityScore` and `supply` of the token, which may change over time, and the unchangeable `initialPrice`, `exponent`, and `initialSupply` values. The function is available only `whenNotPaused`.

Inputs

- `token`
  - **Validation**: `onlyRegistered(token)`.
  - **Impact**: The hashtag-token address that the caller wants to buy.
- `amount`
  - **Validation**: The `amount` of tokens is not directly validated, but the user must have enough funds to buy this amount of tokens.
  - **Impact**: The amount of token to buy.

Branches and code coverage (including function calls)

Intended branches

- The caller received the expected amount of tokens in exchange for the expected amount of `paymentToken` tokens.
  - ☑ Test coverage

Negative behavior

- The caller does not own enough `paymentToken` tokens to buy the `amount` of hashtag tokens.
  - ☑ Test coverage
- The `token` is not registered.
  - ☐ Test coverage
- The contract is paused.
  - ☐ Test coverage

Function call analysis

- `getBuyPrice(token, amount)`
  - **External/Internal?** Internal.
  - **Argument control?** `token` and `amount`.
  - **Impact**: Calculates the current price of the the requested `amount` of tokens.
- `IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), totalPrice)`
  - **External/Internal?** External.
  - **Argument control?** N/A.
  - **Impact**: Transfers of payment for the requested `amount` of tokens.
- `TrendingERC20(token).mint(msg.sender, amount)`
  - **External/Internal?** External.
  - **Argument control?** `amount`.
  - **Impact**: Mints the requested amount of hashtag tokens for the caller of the function.

Function: `createToken(string hashtag, address recipient, uint16 viralityScore)`

The function allows to deploy the TrendingERC20 contract related to some `hashtag` string. Tokens should only be created for unique hashtags.

Inputs

- `hashtag`
  - **Validation**: `registeredHashtags[hashtagLowercase]` — the hashtag should not be already registered.
  - **Impact**: The hashtag for which the new TrendingERC20 contract will be deployed.
- `recipient`
  - **Validation**: No validation.
  - **Impact**: The recipient of the initial supply of tokens.
- `viralityScore`
  - **Validation**: No validation.
  - **Impact**: The initial virality score.

Branches and code coverage (including function calls)

Intended branches

- The token related to the hashtag was successfully deployed.
  - ☑ Test coverage

Negative behavior

- The `hashtag` is already registered.
  - ☑ Test coverage
- The `hashtag` is already registered and in uppercase.
  - ☐ Test coverage
- The caller is not an owner and allowed caller, and `allowCreateTokens` is false.
  - ☐ Test coverage
- The caller is not an owner and allowed caller, and `allowCreateTokens` is true but user does not own enough TrendingERC20 to burn.
  - ☑ Test coverage

Function call analysis

- `_toLower(hashtag)`
  - **External/Internal?** Internal.
  - **Argument control?** `hashtag`.
  - **Impact**: The function lowercases the string to ensure the uniqueness of the token.
- `_verifyPermissions(_msgSender())`
  - **External/Internal?** Internal.
  - **Argument control?** N/A.
  - **Impact**: The function validates that the caller is the owner of the contract, or an allowed address from the `_allowlist`; or if the function is available to everyone, that the caller has burned the required number of other TrendingERC20 tokens.
- `new TrendingERC20(address(this), hashtag, hashtag)`
  - **External/Internal?** External.
  - **Argument control?** `hashtag`.
  - **Impact**: Creates new token contract related to the `hashtag`.
- `token.mint(recipient, initialSupply)`
  - **External/Internal?** External.
  - **Argument control?** `recipient`.
  - **Impact**: Mints the initial supply amount of tokens.

Function: `sellTokens(address token, uint256 amount)`

The function allows the caller to sell tokens they own associated with a specific hashtag. The current sell price is calculated differently than the buy price, and it is the minimum of two prices: the first one depends on the current liquidity and full price of all tokens, and the second depends on `viralityScore` and `supply`. The function is available only `whenNotPaused`.

Inputs

- `token`
  - **Validation**: `onlyRegistered(token)`.
  - **Impact**: The hashtag token address that the caller wants to sell.
- `amount`
  - **Validation**: If the balance of the caller `IERC20(token).balanceOf(msg.sender)` is less than amount, the function will revert.
  - **Impact**: The amount of token to buy.

Branches and code coverage (including function calls)

Intended branches

- The caller received the expected amount of `paymentToken`.
  - ☑ Test coverage

Negative behavior

- The caller does not own enough hashtag tokens to sell.
  - ☐ Test coverage
- The `token` is not registered.
  - ☐ Test coverage
- The contract is paused.
  - ☐ Test coverage

Function call analysis

- `getSellPrice(token, amount)`
  - **External/Internal?** Internal.
  - **Argument control?** `token` and `amount`.
  - **Impact**: Calculates the current price for selling the specified amount of tokens.
- `IERC20(token).balanceOf(msg.sender)`
  - **External/Internal?** External.
  - **Argument control?** `token` and `amount`.
  - **Impact**: Returns the current hashtag token balance for the caller account to be sure that the caller owns enough tokens.
- `TrendingERC20(token).burn(msg.sender, amount)`
  - **External/Internal?** External.
  - **Argument control?** `token` and `amount`.
  - **Impact**: Burn the hashtag tokens from the caller account.
- `IERC20(paymentToken).safeTransfer(msg.sender, totalPrice);`
  - **External/Internal?** External.
  - **Argument control?** N/A.

- **Impact**: Transfers the amount of `paymentToken` for the token sale to the caller of the function.

5.2. Module: MerkleTokenShare.sol

Function: `batchClaimTokens(BatchClaimTokensInput[] calldata input)`

Calls the `claimTokens` for data from the `input` array in turn. For more information, see the description of the `claimTokens` function.

Function: `function claimTokens(bytes32[] proof, uint256 epoch, uint256 index, uint256 tokenIdx, uint16 shareBbps)`

The function allows the user to claim their share if provided a valid Merkle proof. The `merkleRoot` data for verification can be only provided by the owner of the contract using the `updateTokenShares` function. The caller of the function cannot control recipient address, so they can only claim the funds assigned to them. At the same time, the caller has control of which of the epochs they want to claim funds.

Inputs

- `proof`
  - **Validation**: The `MerkleProofLib.verify` function returns true if the proof is valid.
  - **Impact**: Merkle proof containing sibling hashes on the branch from the leaf to the root of the Merkle tree.
- `epoch`
  - **Validation**: If `tokenClaims[epoch][recipient]` is true, the caller already claimed for this `epoch`.
  - **Impact**: Epoch for which the user wants to claim tokens.
- `index`
  - **Validation**: Used to calculate the hash of the leaf; if `MerkleProofLib.verify` returns true, the `index` is valid.
  - **Impact**: It is used to calculate the hash, but in fact it is redundant.
- `tokenIdx`
  - **Validation**: Used to calculate the hash of the leaf; if `MerkleProofLib.verify` returns true, the `tokenIdx` is valid.
  - **Impact**: The index of the token that will be claimed.
- `shareBbps`
  - **Validation**: Used to calculate the hash of the leaf; if `MerkleProofLib.verify` returns true, the `shareBbps` is valid.
  - **Impact**: The portion of `share.amounts[tokenIdx]` transfererred to the caller, in basis points.

Branches and code coverage (including function calls)

Intended branches

- The caller claimed funds properly.
  - ☐ Test coverage

Negative behavior

- The invalid `proof`.
  - ☐ Test coverage
- The `tokenShares` does not exist for `epoch`.
  - ☐ Test coverage
- The caller already claimed funds for this `epoch`.
  - ☐ Test coverage
- The invalid `tokenIdx`.
  - ☐ Test coverage
- The invalid `shareBbps`.
  - ☐ Test coverage

Function call analysis

- `MerkleProofLib.verify(proof, share.merkleRoot, keccak256(abi.encodePacked(index, recipient, tokenIdx, shareBbps)`
  - **External/Internal?** Internal.
  - **Argument control?** `proof`, `index`, `tokenIdx`, and `shareBbps`.
  - **Impact**: Returns `true` if `leaf` exists in the Merkle tree with `root` and given `proof`.
- `IERC20(token).safeTransfer(recipient, amount)`
  - **External/Internal?** External.
  - **Argument control?** N/A.
  - **Impact**: Transfers the amount of token calculated based on `shareBbps` to the caller of the function.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Blast Testnet.

During our assessment on the scoped SAX contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature. SAX acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.