Zellic

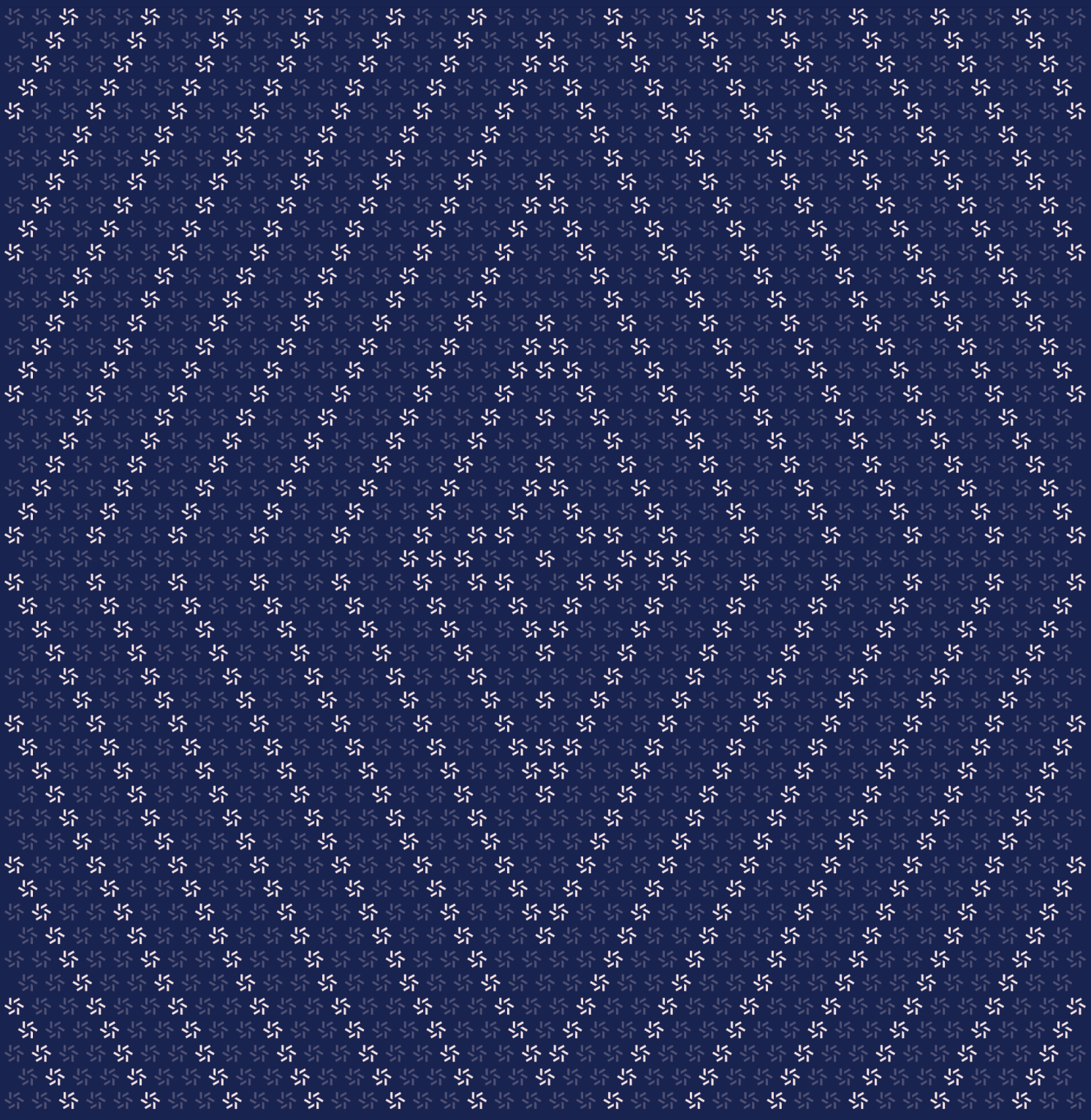
Prepared for
Shama & Genesis
PondFun

Prepared by
Kuilin Li
Sylvain Pelissier
Zellic

September 3, 2024

Memecoin Launcher

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 PondFun from August 2nd to August 4th, 2024. During this engagement, Zellic reviewed the Memecoin Launcher's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Could swap mechanics lead to financial risks for the protocol or its users?
- Could liquidity-pool creations lead to financial risks for its users?
- Could an on-chain attacker drain the contract?

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
- The forum part of the contracts, which is planned to be moved off chain in the next update

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 Memecoin Launcher contracts, we discovered eight findings. No critical issues were found. Two findings were of high impact, one was of medium impact, four were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for PondFun's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	4
Informational	1

2. Introduction

2.1. About PondFun

PondFun contributed the following description of the memecoin launcher:

> PondFun is a memecoin launcher and trading platform designed for easy and seamless token creation and trading on the Linea blockchain. It enables users to effortlessly launch new tokens and engage in buying and selling activities, making the process of entering the memecoin market straightforward and accessible.

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:

Memecoin Launcher Contracts

Type	Solidity
Platform	EVM-compatible
Target	pondfun-contracts-audit
Repository	https://github.com/1vanov/pondfun-contracts-audit ↗
Version	`ff9ebf470a7795b2bdb2dfcf2971cb98249016af`
Programs	`PondERC20` `PondFun`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Kuilin Li
Engineer
kuilin@zellic.io

Sylvain Pelissier
Engineer
sylvain@zellic.io

2.5. Project Timeline

August 26, 2024	Start of primary review period
August 28, 2024	End of primary review period

3. Detailed Findings

3.1. Fee-recipient reentrancy

Target	PondFun		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

After a successful token deployment to a liquidity pool, the functions `quoteToken`, `swapTokenForETH`, and `swapETHForToken` prevent swapping or quoting the token with the contract with the following check:

```
if (token.dexPair != address(0)) revert TokenAlreadyDeployed();
```

During the deployment of the token to a liquidity pool, the fee recipient is paid with the `_safeTransferETH` function before the variable `token.dexPair` is set:

```
function _deployTokenLiquidity(address _address) internal {
        //...

        IERC20(token.token).approve(address(router), tokenAmount);

        (uint256 tokenAddedAmount, uint256 ethAddedAmount, uint256 liquidity)
    = router.addLiquidityETH{ value: quotedEthAmount }(
            token.token,
            false,
            quotedTokenAmount,
            quotedTokenAmount,
            quotedEthAmount,
            address(this),
            block.timestamp + 10
        );

        if (liquidity == 0) revert LiquidityDeployFailed();

        if (tokenAddedAmount < tokenAmount) {
            _safeTransferToken(token.token, $.config.feeRecipient, tokenAmount
    - tokenAddedAmount);
        }
        if (ethAddedAmount < ethAmount) {
            _safeTransferETH($.config.feeRecipient, ethAmount
    - ethAddedAmount);
```

```
        }

        token.dexPair = router.pairFor(address(weth), token.token, false);
        token.bondingCurve = BondingCurve({ vTokenReserves: 0, vETHReserves:
    0, rTokenReserves: 0, rETHReserves: 0 });

        emit TokenLiquidityDeploy(token.token, token.dexPair);
    }
```

Note that this ETH transfer happens before the `dexPair` is set and `bondingCurve` is zeroed, which means that if the fee recipient reenters the contract during the transfer, they will still be able to trade on the invalid bonding curve. If they sell tokens at this time, the tokens will become stuck in the contract, and the ETH sent out will be double-counted, making the contract insolvent.

Impact

Since the fee recipient receives all the fees, it does not cost them anything to attempt this attack.

So, by transactionally deploying a new token and then buying enough of it with ETH such that it immediately is deployed to the DEX, and then doing this reentrancy attack, the fee recipient can freely exfiltrate as much ETH from the native balance of the contract as they like.

Recommendations

The contract should implement the checks-effects-interactions pattern and set the variables `token.dexPair` and `token.bondingCurve` before transferring ETH to the `feeRecipient`.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `b0721f46` ↗.

3.2. Fee recipient can censor forum posts

Target	PondFun		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Forum posts are created by the `createForumPost` function, and some fees are paid to the fee recipient before posting:

```
function createForumPost(address _address, string calldata _text)
    public payable tokenExists(_address) returns (ForumPost memory) {
    PondStorage storage $ = _getStorage();
    Forum storage forum = $.forums[_address];

    if (msg.value < $.config.forumFeeEth) revert NotEnoughETH();
    _safeTransferETH($.config.feeRecipient, msg.value);

    ForumPost memory post = ForumPost({ author: msg.sender, text: _text,
    timestamp: block.timestamp, isHidden: false });

    forum.posts.push(post);

    emit ForumPostCreate(post.author, _address, post.text, post.timestamp);

    return post;
}
```

However, the fee recipient can choose to revert for some fee transfers, which will cause the parent transaction to revert, censoring the corresponding forum post.

Impact

The fee recipient may censor some posts by reverting during the transfer call they receive from `createForumPost`.

Recommendations

A better approach would be to allow the fee recipient to claim their fees independently of the forum post creation.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit ce3e8543 ↗. This fix moves the entire forum part of the contract off-chain.

3.3. Constructor left uninitialized

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

Description

The Memecoin Launcher is upgradable and uses the UUPS Proxy mechanism. Thus, the contract is initialized by the `initialize` function and not the constructor. However, as mentioned in the OpenZeppelin documentation ↗, the implementation contract should not be left uninitialized.

Impact

An attacker may initialize the implementation contract, which allows them to own it. This is a reputational hazard, because the implementation contract looks legitimate, since it was deployed by a legitimate deployer.

Recommendations

To prevent the implementation contract from being used, OpenZeppelin recommends to add a constructor that invokes the `_disableInitializers` function to lock it when it is deployed.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `4e0ee0eb` ↗.

3.4. Identifiable anonymous tokens

Target	PondFun		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

A new token is launched by the `launchToken` function. This function takes as parameter the token data, like its name and symbol, and an argument `isAnon` to decide if the token is anonymously deployed. If `isAnon` is set to true, the token data saves the null address as the deployer:

```
Token memory token = Token({
    token: address(erc20Token),
    deployer: isAnon ? address(0) : msg.sender,
    dexPair: address(0),
    title: data.title,
    symbol: data.symbol,
    description: data.description,
    imageUrl: data.imageUrl,
    links: data.links,
    tradesCount: 0,
    bondingCurve: bondingCurve
});
```

However, the anonymous launch is not really anonymous since the transaction calling `launchToken` would identify the creator.

Impact

The creator address of a token may be identified even if the anonymous launch parameter was set to true.

Recommendations

We recommend removing this feature, because it can potentially mislead users into expecting more anonymity than what is possible on EVM chains. No matter which branch of the `isAnon` ternary above actually executes, the transaction containing the execution of the branch is fully public, so the sender can be identified by re-tracing the execution.

If token creation needs to be completely anonymous, one strategy is to allow the counterfactual deployment of tokens, so that instead of using CREATE to deploy the ERC20 contract, CREATE2 is used with a salt that includes the deploy parameters. This will allow token deployment to happen completely off-chain, and then the first real buyer on the bonding chain, which may or may not be the designer of the token, will cause the actual deployment of the contract.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `b3e7f3fb`↗. PondFun has removed the possibility of creating tokens anonymously.

3.5. Fee amount may be rounded to zero

Target	PondFun		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

To swap an `_amountIn` of ETH or tokens, the user has to pay a swap fee. The fee calculation is done as follows:

```
fee = (_amountIn * $.config.tradeFeeBps) / WEI6;
```

In Solidity, if the amount is less than `WEI6 / $.config.tradeFeeBps`, the fee is rounded down to zero.

Impact

It may be economical for a user to deploy a contract that repeatedly purchases a small amount of tokens, instead of executing a large purchase all at once, to avoid the trade fee. If they do this, the user will pay more gas, but this gas may be cheaper than the fee, depending on the size of the `tradeFeeBps` parameter.

Recommendations

The division by `WEI6` should round up, instead of down. Alternatively, ensure that `tradeFeeBps` is always low enough that the gas needed per trade size to exploit this rounding error is more expensive than just paying the fee.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `07f6d3ea` ↗.

3.6. Edit function does not change ERC-20 symbol and name

Target	PondFun		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The function `editToken` allows the contract owner to edit token details stored in the contract.

```
function editToken(address _address, TokenLaunchData memory data)
    public onlyOwner tokenExists(_address) {
    PondStorage storage $ = _getStorage();
    Token storage token = $.tokens[_address];

    token.title = data.title;
    token.symbol = data.symbol;
    token.description = data.description;
    token.imageUrl = data.imageUrl;
    token.links = data.links;
}
```

However, the function does not edit the ERC-20 token name and symbol stored in the token contract, which is only set when it is deployed.

Impact

Editing the token title or symbol will cause a display discrepancy between the ERC20's fields and the ones used by the frontend. This may confuse users after DEX deployment.

Recommendations

If the goal of the function is to fully edit the token, then the PondERC20 contract should implement such an update mechanism. If the goal is to change details only on the front end, then the function needs to be clearly documented.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `2a039ef1` ↗. PondFun has removed the possibility of editing tokens after creation.

3.7. Native ETH may be sent by mistake in some cases

Target	PondFun		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

This contract has an open and unpermissioned receive handler:

```
receive() external payable {}
```

In normal operation, this is used to receive native ETH from calls to `WETH.withdraw`. However, any user may accidentally send native ETH to this contract, causing it to become stuck.

Another way users may accidentally lose native ETH is through the `swapETHForToken` function. This is a payable function, but if `isWETH` is set to true, it uses WETH instead, never reading the `msg.value`:

```
function swapETHForToken(
    uint256 _amountIn,
    uint256 _amountOutMin,
    address _address,
    bool _isWETH
) public payable tokenExists(_address) {
    // [...]

    if (_isWETH) {
        _safeTransferTokenFrom(address(weth), msg.sender, address(this),
    _amountIn);
        weth.withdraw(_amountIn);
    } else {
        if (msg.value < _amountIn) revert NotEnoughETH();
    }
```

This means that a user may mistakenly send native ETH alongside a call to `swapETHForToken` that actually pays the contract in WETH. This native ETH will become stuck in the contract.

Additionally, in the same code, the `NotEnoughETH` check ensures that `msg.value >= _amountIn` instead of being equal. If a user sends in more native ETH than `_amountIn`, the excess becomes stuck in the contract.

Impact

It is standard to revert upon unexpected or incorrect transfers of native ETH, instead of just accepting the funds, in order to ensure that users do not mistakenly send value that they do not intend to freely donate. However, through the three ways outlined above, native ETH can become stuck in the contract.

Recommendations

We recommend restricting the `receive` function to only WETH, adding a check for `msg.value == 0` in the `_isWETH` branch of `swapETHForToken`, and modifying the check to `msg.value == _amountIn` in the other branch.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit 45ea64e3 ↗. The issue has been fixed by adding a WETH address check and additional `msg.value` checks as recommended.

3.8. TokenTrade event parameters can be incorrect

Target	PondFun		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

During a call to `swapTokenForETH`, the native transfer of ETH to the sender happens before the `TokenTrade` event's parameters are fixed:

```
function swapTokenForETH(
    uint256 _amountIn,
    uint256 _amountOutMin,
    address _address,
    bool _isWETH
) public payable tokenExists(_address) {
    // [...]

    if (_isWETH) {
        weth.deposit{ value: amountOutWithFee }();
        _safeTransferToken(address(weth), msg.sender, amountOutWithFee);
    } else {
        _safeTransferETH(msg.sender, amountOutWithFee);
    }

    // [...]

    emit TokenTrade(msg.sender, token.token, amountOut, _amountIn, false,
    curve.vETHReserves, curve.vTokenReserves, block.timestamp);
}
```

This means that if the sender reenters the contract during the `_safeTransferETH`, they can submit another swap, which modifies the `curve` storage variable.

Impact

In the emitted TokenTrade event, the `curve.vETHReserves` and `curve.vTokenReserves` may not accurately reflect the reserves after the trade that caused the event to be emitted, if a user reenters the contract to do a second trade.

This breaks invariants for off-chain code that monitors those events.

Recommendations

We recommend emitting the event before the reentrancy site, following the standard checks-effects-interactions pattern.

Remediation

This issue has been acknowledged by PondFun, and a fix was implemented in commit `8d50446f` ↗.

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. Token names and symbols may be identical

In the Memecoin Launcher, nothing prevents different users from launching tokens with identical names and symbols. This is also the case for other existing projects where it is up to the user to figure out which token address they really want to buy. However, for the front-end usage, it may be helpful to be able to distinguish between tokens.

4.2. Price gap during deployment

Before the liquidity pool is deployed, the token price is computed with a formula involving virtual ETH and token reserves. When the liquidity pool is deployed, the price that the liquidity-pool contract gets is based on the real reserves, which may be different from the price that is being used immediately before deployment, which includes the constant offset from the virtual assets. Even if for some `dexThreshold` values the gap may be minimal, it should be documented how this value should be chosen to avoid loss during liquidity deployment.

4.3. Incomplete documentation

There are certain areas in the code where a documentation of the mechanisms would help with comprehensibility. The undocumented function `getTokens` return tokens in the reverse order:

```
function getTokens(uint256 _count, uint256 _offset)
    public view returns (Token[] memory) {
    PondStorage storage $ = _getStorage();

    uint256 size = $.tokenAddresses.length;
    if (size < _count + _offset) _count = size - _offset;

    Token[] memory result = new Token[](_count);

    for (uint i = 0; i < _count; i++) {
        result[i] = this.getToken($.tokenAddresses[size - _offset - 1 - i]);
    }
```

```
        return result;
    }
```

For example, asking for offset zero would return the latest created token and then, going backwards, `count` time. It could give unexpected results to a user, depending on how it is used.

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on.

4.4. Centralization risk

There are two types of privileged accounts for the Memecoin Launcher:

- The owner
- The fee recipient

Users can create new tokens but have otherwise no special powers.

The fee recipient has no special power, but it may revert during certain swap operations, preventing some operations to successfully happen. They are also able to censor some forum posts, as described in Finding 3.2. ↗.

The owner can withdraw tokens or ETH with the functions `recoverToken` and `recoverETH` to drain all liquidity from the contract. They can also change the fee amounts, the DEX router, and the fee recipient at any moment with the `setConfig` function. Finally, the owner can upgrade the contract and completely change its behavior, up to and including performing any external call from it.

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. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access and defining the contract as pauseable, to prevent user launching or trading tokens during an emergency, and allowing withdrawal only at that time. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users.

4.5. Variable DEX deploy price creates arbitrage opportunity

When a purchase on the bonding curve causes the real ETH reserves to exceed a preset constant `$.config.dexThreshold`, the purchase will cause the DEX to be deployed immediately afterwards.

This means that the price at which the PondFun contract provides liquidity at is dependent on the size of the last buy that exceeds the `dexThreshold`. Since this price is controllable, and directly affects the amount of real value burned by the protocol (the burnt LP tokens) to add permanent liquidity to the pair, this means that an arbitrageur can take most of that real value themselves.

This arbitrage would look like a user buying up the entire supply on the bonding curve, using an amount of ETH much higher than the `dexThreshold` limit, and then selling the extra tokens back after the protocol deploys the token on the real DEX, earning a profit.

Instead of allowing the last buyer to purchase tokens on the bonding curve above the `dexThreshold`, we recommend either reducing the size of the last buyer's purchase and returning the rest of the ETH with the DEX deploy, or executing the purchase by buying only up to the `dexThreshold` on the bonding curve and then buying the rest on the real DEX after the deploy - with the slippage guarantee still checked on the sum of the purchased tokens. This will ensure that tokens will not be sold at an inappropriate price above the `dexThreshold`.

This issue has been acknowledged by PondFun. PondFun remediated this by executing the remainder of the last buyer's purchase on the real DEX after it is deployed.

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: PondFun.sol

Function: `swapETHForToken()`

The function is used to buy tokens. If the real ETH reserve reaches a certain threshold, the tokens and the ETH reserve are sent in a DEX pool through a router.

Inputs

- `_amountIn`
  - **Validation**: The amount is checked to be bigger than the message value in case ETH amount is used.
  - **Impact**: The amount of tokens to buy.
- `_amountOutMin`
  - **Validation**: The quoted amount is checked to be greater than or equal to `_amountOutMin`; otherwise, it reverts.
  - **Impact**: The minimum amount of tokens to buy.
- `_address`
  - **Validation**: Checked to be a valid token address by the `tokenExists` modifier.
  - **Impact**: The token address to buy.
- `_isWETH`
  - **Validation**: No validation.
  - **Impact**: Decides if ETH or wETH is used to buy the token.

Branches and code coverage (including function calls)

Intended branches

- Swap ETH for tokens with sufficient balance.
  - ☑ Test coverage
- Swap wETH for tokens with sufficient balance.
  - ☑ Test coverage
- Swap ETH for tokens and reach the DEX threshold.
  - ☐ Test coverage

Negative behaviour

- Swap ETH for tokens with insufficient balance.
  - ☐ Test coverage
- Swap wETH for tokens with insufficient balance.
  - ☐ Test coverage
- Swap ETH for tokens already deployed.
  - ☐ Test coverage

Function: `swapTokenForETH()`

The function is used to sell tokens. If the real ETH reserve reaches a certain threshold, the tokens and the ETH reserve are sent in a DEX pool through a router.

Inputs

- `_amountIn`
  - **Validation**: The amount is checked to be bigger than the message value in case ETH amount is used.
  - **Impact**: The amount of tokens to sell.
- `_amountOutMin`
  - **Validation**: The quoted amount is checked to be greater than or equal to `_amountOutMin`; otherwise, it reverts.
  - **Impact**: The minimum amount of ETH to get when tokens are sold.
- `_address`
  - **Validation**: Checked to be a valid token address by the `tokenExists` modifier.
  - **Impact**: The token address to sell.
- `_isWETH`
  - **Validation**: No validation.
  - **Impact**: Decides if ETH or wETH is used to sell the token.

Branches and code coverage (including function calls)

Intended branches

- Swap tokens for ETH with sufficient balance.
  - ☑ Test coverage
- Swap tokens for wETH with sufficient balance.
  - ☑ Test coverage
- Swap tokens for ETH and reach the DEX threshold.
  - ☐ Test coverage

Negative behaviour

- Swap tokens for ETH with insufficient balance.
  - ☐ Test coverage

- Swap tokens for wETH with insufficient balance.
  - ☐ Test coverage
- Swap tokens already deployed for ETH.
  - ☐ Test coverage

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Memecoin Launcher contracts, we discovered eight findings. No critical issues were found. Two findings were of high impact, one was of medium impact, four were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

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

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

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