SMART CONTRACTS REVIEW

zokyo

October 3rd 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING ACES

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 2 Critical issues: 2 resolved = 0 points deducted
- 1 High issue: 1 resolved = 0 points deducted
- 7 Medium issues: 4 resolved and 3 acknowledged = - 5 points deducted
- 1 Low issue: 1 acknowledged = - 1 points deducted
- 3 Informational issues: 3 resolved = 0 points deducted

Thus, 100 - 5 - 1 = 94

TECHNICAL SUMMARY

This document outlines the overall security of the ACES smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the ACES smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 2 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the ACES team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the ACES repository:
Repo: https://github.com/pocketacesbot/aces-launchpad-audit

Last commit - 34d2c122a51d4f714faa062f23c7556969435c27

Contracts under the scope:

- AcesFactory.sol
- AcesLaunchpadToken.sol
- AcesToken.sol
- FixedMath.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of ACES smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Aces protocol is aiming to shake up the collectibles industry by creating a liquid transparent market place where collectors, sellers and traders can all participate. This platform is derived from RWAs (Real World Assets) listings with derivatives markers which makes new opportunities available to users such as price discovery, liquidity and easy participation at the entry level. Aces plans to be the only project to monetize RWAs through trading fees by tokenizing collectible assets, and earning revenue by trading and tapping into new markets.

Zokyo was tasked with the security audit of the AcesFactory, AcesLaunchpadToken, AcesToken and FixedMath contracts. Users are able to create launchpad tokens and specify their various attributes such as the type of curve (quadradic/linear), steepness, floor, name, symbol, salt (to avoid duplicate deployments) and the amount at which the tokens are bonded at. Once a new launchpad token is deployed (acesLaunchpadToken), users will be free to purchase it with AcesTokens (buyTokens) and sell in return for AcesTokens (sellTokens). Once a token is bonded, the owner is relinquished and transfers and approvals are able to be performed.

Overall the code is well structured and well documented with details on expected function behaviour and parameters supplied. The vulnerabilities identified ranged from critical down to informational and best security practices. These findings mostly revolved around mathematical and business logic errors allowing malicious users to extract tokens from the contract, loss of tokens, centralisation issues and issues around contract upgradeability. Following the audit, a fix review will take place where the fixes for the findings are carefully reviewed to ensure that further bugs are not introduced and that the aforementioned issues have been addressed and resolved.

The security team at Zokyo recommends reviewing the findings, applying the suggested fixes and running the supplied proof of concept(s) against the fixes (where possible) prior to the fix review to ensure that these attacks are no longer possible. In addition to this, considering the gravity of the issues found and the small time window of the audit which was about 2 business days, it's also highly recommended that the codebase in addition to other contracts and web applications/infrastructure which may be scheduled to deploy into a production environment undergo more extensive and rigorous auditing. The team at Zokyo wishes Aces.fun the best of luck when deploying to the mainnet.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Name of company team and the Name of company team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Rounding Errors In The Way Prices Are Calculated Results In The Theft Of Ace Tokens	Critical	Resolved
2	User pays same ace tokens to buy [i, i + 1) ether launchpad tokens	Critical	Resolved
3	Any user can create tokens that never gets bonded	High	Resolved
4	Token total supply values which are less than 1 ether through burning will result in untradeable tokens	Medium	Resolved
5	No clarity on how token gets listed on Aerodrome DEX	Medium	Resolved
6	AcesFactory Contract Is UUPSUpgradeable But Does Not Have Storage Gaps	Medium	Resolved
7	Owners or Ex Owners Who Still Possess A Seed Phrase For The Contract Owner Can Set Malicious Token Implementations Which Can Cause Unexpected Behaviours	Medium	Acknowledged
8	Deprecated Usage Of transfer In withdrawETH	Medium	Resolved
9	Owner can withdraw all ace tokens anytime	Medium	Acknowledged
10	Sniper bots can buy tokens at cheaper price initially and dump tokens by selling later	Medium	Acknowledged
11	AcesFactory and Aces owner's private key could be compromised and lack of 2 step ownership transfer	Low	Acknowledged
12	sellTokens Is Marked As Payable For Unknown Reasons	Informational	Resolved
13	Unused Storage Variable aerodromeRouterAddress	Informational	Resolved
14	Upgradeability feature for AcesLaunchpadToken is not necessary	Informational	Resolved

Rounding Errors In The Way Prices Are Calculated Results In The Theft Of Ace Tokens

Description:
In the contract AcesFactory.sol, the method createTokens(...) allows anyone to create a launchpad token. Additionally, the buyTokens method allows anyone to purchase any launchpad token that has been created, meaning the token's creator can also buy their own tokens.

The method buyToken() has the parameters `amount` and `aceAmountIn`, which are set by the users. User can pass any amount > `0` and can set aceAmountIn as `0`. There is a case where a user sets `amount < 1 ether` and `aceAmountIn as 0`.

In such a case, the user can mint any amount of the launch pad token since getPrice(...) returns `0` ace token required for buying tokens < 1 ether by repeating the process multiple times.
Once a lot of launchpad tokens (any launchpad token) have been bought for free, the user can sell them for real Ace tokens. Using this, anyone can steal all the Ace tokens from the factory contract.

The root cause stems from the following piece of code:
https://github.com/pocketacesbot/aces-launchpad-audit/blob/main/src/AcesFactory.sol#L315
Ultimately, because we are dividing any value less than 1 ether (ie. `0`.9999....) by 1 ether (`0`.999 * 1e18 / 1 * 1e18), solidity will by default round down to zero. In conjunction with passing zero acesAmountIn will allow users to siphon free launchpad tokens which they are free to trade for ace tokens.

Impact:
Malicious users who exploit this rounding issue are able to get free launchpad tokens. This attack can be run as many times as they desire - furthermore, these tokens can be sold via the sellTokens function to extract ace tokens from the contract. This was rated a critical in severity because this results in the catastrophic economical damage to the protocol through the direct theft of principle (not yield) and any reasonable protocol team would not accept this to be within their risk appetite. In addition to this issue, users who call the sellTokens function and are aiming to sell less than 1 * 1e18 will receive nothing in return due to the same root cause.

PoC:
https://gist.github.com/kuldeep23907/f9006a3aaf026b267827f1bdd2dc421e

Recommendation:
Update the getPrice() calculation logic to calculate the ace token price based on the `amount` of launchpad token provided by the user as opposed to the amount divided by 1e18.

User pays same ace tokens to buy [i, i + 1) ether launchpad tokens

There is another critical issue due to the rounding error in the getPrice() method. The getPrice(...) method returns the same amount of ace tokens for [i, i+1) ether amount of launchpad tokens.

For eg: If a user wants to buy 1 ether launchpad tokens, they need to pay 1e12 ace tokens.
If a user wants to buy 1.5 ether launchpad tokens, they need to pay 1e12 ace tokens.
If a user wants to buy 2 ether launchpad tokens, they need to pay 3e12 ace tokens.
If a user wants to buy 2.5 ether launchpad tokens, they need to pay 3e12 ace tokens.

The root cause stems from the following piece of code:
https://github.com/pocketacesbot/aces-launchpad-audit/blob/f1a4d475c121a136887407e98fdae0c6f906e407/src/AcesFactory.sol#L309 and the line just below.

Ultimately, because we are dividing any value by 1 ether in Solidity, it leads to the truncation of the result. For example: 11/10 will result in 1, not 1.1. So when we divide the amount 1.5 ether by 1 ether, in the supply and amount calculation, it will be 1 ether, not 1.5 ether.
Also, if we remove this normalisation and modify the logic to normalise the value at the end after all calculation, there is a possibility of multiplication overflow. So, using a library (Uniswap fullmath library), which prevents this overflow, is advised.

Impact:
This is a direct loss of funds for the protocol, as users can buy launchpad tokens with smaller amounts than they should actually pay.

PoC:
https://gist.github.com/kuldeep23907/e799593a8118f1e01c4e580b857de9ed

Result:
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/FactoryTest.t.sol:FactoryTest
[PASS] test_UserSellsButGetSameAce() (gas: 382845)

Logs:

```
  buy price for 1 ether 100000000000000000000000000000000
  buy price for > 1 ether 100000000000000000150000000000000
  buy price for 1.5 ether 187500000000000000000000000000000
  buy price for 2 ether 300000000000000000000000000000000
  buy price for 2.5 ether 437500000000000000000000000000000
```

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.69ms (2.23ms CPU time)

Recommendation:

Use the Uniswap math library or the FullMath library.

Any user can create tokens that never gets bonded

Description:
In the contract AcesFactory.sol, the method createTokens(...) allows anyone to create a token and provide the parameter `tokenBondedAt`. This can be any value > 1 ether.

In the AceLaunchPadToken.sol, there is a limitation of max supply for tokens, i.e. 1 billion tokens. Since a token can't be minted for more than 1 billion tokens, if a user creates tokens with `tokenBondedAt` > 1 billion tokens, the token will never get bonded as the following condition will never reach.

```
if (
            !token.tokenBonded && totalSupply + amount >=
token.tokensBondedAt
        ) {
            token.tokenBonded = true;
```

```
            launchPadToken.setTransfersEnabled(true);
            launchPadToken.renounceOwnership();
            emit BondedToken(tokenAddress, totalSupply + amount);
        }
```

Impact:
As this is a launchpad platform, a token becomes tradeable on DEX once it is bonded. Since that won't happen due to tokenBonded set greater than max supply of tokens, the token will not be available on the dex for users to trade.

Recommendation:
Add a check to ensure `tokenBondedAt` is less than max-supply.

Token total supply values which are less than 1 ether through burning will result in untradeable tokens

Description:
In the contract AcesFactory.sol, it is required that a launchpad token has a total supply of at least 1 token for it to work correctly.

There is a case where total supply will become less than 1 ether and that tokens won't be tradeable again on the platform. These are the steps:

- Owner creates a launchpad tokens → 1 ether is minted to owner and Total supply is 1 ether
- The token owner can sell the tokens just after the token is created.
- If the owner sells token <1 ether (for eg: `1 ether - 1`) amount of tokens, the owner will receive `0` ace tokens, but `1 ether - 1` tokens are burned, reducing the total supply.
- Now total supply is 1 wei.

After this, if any buy token attempt by any user will fail since getPrice() will underflow due to the following:

```
        uint256 supply = isBuy ? totalSupply : totalSupply - amount;
//@audit revert when totalSupply < amount
```

Since here `totalSupply` is 1 and amount > totalSupply.

Impact:
The launchpad token becomes untradeable once total supply is less than 1 ether.

Recommendation:
Update the getPrice() calculation logic to mitigate the edge case where total-supply needs to be 1 ether. It is advised to use virtual assets for price calculation.

No clarity on how token gets listed on Aerodrome DEX

Description:
In the contract AcesFactory.sol, the token is marked as bonded with transfer enabled as true in the method buyTokens() once a token reached it's token bonding limit and token becomes untradable on the platform.

After token gets bonded, the doc mentions that the token should get listed on the router along the toal ace amount collected for that token. But there is no code logic performing the same.

If the protocol expects to list the token manually, it can lead to various issues such as front-running, someone else creating the token pair first on dex etc.

Impact
Manually listing token on dex can be sandwiched attack, by someone buying huge amounts of token just before token is going to bond and dumps all of it when token gets listed on dex. The most problematic issue would be if the token never gets listed which will lead all users losing their funds.

Recommendation:
Add logic in the contract where a token pair is created on the dex and funds are transferred.

AcesFactory Contract Is UUPSUpgradeable But Does Not Have Storage Gaps

Description:
The AcesStorage factory inherits the UUPSUpgradeable contract which allows for the contract to upgrade it's implementation should a bug fix or a new feature be added to the contract. Because this contract lacks the storage gaps, upgrading the contract could have unexpected consequences due to collisions..

Impact
Lack of storage gaps could cause storage collisions upon upgrade.

Recommendation:
It's recommended that storage gaps are added to the footer of the contract similarly to the following as an example:

https://github.com/perpetual-protocol/perpetual-protocol/blob/bf5adf651ed948a6e18bd939d0ad4bf2a6cfe779/src/TollPool.sol#L52

Owners or Ex Owners Who Still Possess A Seed Phrase For The Contract Owner Can Set Malicious Token Implementations Which Can Cause Unexpected Behaviours

Description:
The setTokenImplementation allows contract owners to set a new implementation for launchpad tokens. This allows them to keep the token contract updated and to fix future tokens should bugs occur or implement new features; however, the new implementation is set immediately.

Impact
This was rated a medium in severity because this relies on an existing owner or an ex contract owner to behave maliciously. Malicious token implementations may cause unexpected behaviours or even allow such actors to rug the protocol.

Recommendation:
It's recommended that two step, timelocked functions are set for critical functions such as setTokenImplementation, withdrawAces and withdrawETH as users are selling launchpad tokens against Aces and ETH.

Deprecated Usage Of transfer In withdrawETH

Description:
The withdrawETH uses the payable(msg.sender).transfer(owed) method. There exists a flaw in the contracts implementation which can cause problems in the contract's overall gas consumption, potentially leading to unexpected errors and a loss of funds for users.

Impact
This issue was assigned a Medium in severity because it can have consequences for the contract's gas usage and the overall performance of the Ethereum network however, certain edge case conditions must be met for transactions to fail. When `payable()` is called, it sets the stipend for the transfer to the caller's remaining gas, which means the contract has no control over how much gas is used for the transfer. This can lead to the contract running out of gas and failing, or to the contract consuming more gas than necessary and potentially causing congestion on the network. Additionally, the original `transfer()` function uses a fixed stipend of 2300 gas units, which may not be sufficient for some contracts to process the transfer. This can limit the contracts ability to interact with other contracts that require more gas to complete the transaction, potentially hindering its functionality. There is also no check to see if the transaction was successful which may result in a loss of user funds if a transaction fails.

Recommendation:
It's recommended that the contracts use a low level function call when transferring Ether between contracts and EOAs. This can be implemented using the following example:

```solidity
(bool success,) = address(owner()).call{value: address(this).balance}("")
require(success, "Failed to refund Ether!")
```

Owner can withdraw all ace tokens anytime

Description:
In contract AceFactory.sol, the method withdrawACES allows the owner to withdraw total accumulated ace tokens for any launchpad token. The issue here is that the owner can do that anytime, even when trading is going on.

If the owner withdraws all aces when trading is going on, users won't be able to sell their launchpad tokens as there won't be any Ace to get in return.

Impact
Since this is a role-gated method, the likelihood is low but the impact is severe since this will be a direct loss of funds for users.

Recommendation:
Allow the owner to take out Aces only for launchpad tokens that are bonded. It is advised to add an emergency funds withdrawal method which allows the owner to withdraw Ace tokens but only in paused state for buying and selling tokens.

Sniper bots can buy tokens at cheaper price initially and dump tokens by selling later

Sniper bots can buy initial supply of tokens when they are cheap and then when the price goes up of the tokens, they can dump these tokens. This is because the tokens follow a bonding curve and price keeps increasing after each buy.

```
   function buyTokens(
        address tokenAddress,
        uint256 amount,
        uint256 acesAmountIn
    ) public {
        require(amount > 0, "Invalid amount");
  require(tokenAddress != address(0), "Invalid address");
  ...
}
```

This can be unfair with the users who are legitimately trying to buy tokens from the pool.

Impact:
This issue can lead to unfair trading of tokens and price manipulation and hence was rated medium leading to realized losses for late buyers when the sniper bot dumps token immediately afterwards..

Recommendation:
It is advised to add anti-bot features in the contract to disallow Sniper bots from sniping and buying the tokens just after a token launches or create a whitelist to allow only trusted participants to buy. One of the easiest ways to disallow a bot from sniping the pool tokens is to disallow contract addresses from buying these tokens such as by using standard isContract methods from Openzeppelin contracts (but this does introduce a small tradeoff that it does not allow Multisig holders from buying tokens).

AcesFactory and Aces owner's private key could be compromised and lack of 2 step ownership transfer

Description

The AcesFactory contract uses OwnableUpgradeable and the AcesToken uses Ownable contract in order to use access control modifiers for admin functions. But the issue with ownable is that if the private key of the owner is compromised, the complete protocol and treasury can easily get compromised resulting in a critically severe loss of funds.

Also lack of 2 step ownership transfer opens the risk of the admin accidentally transferring the ownership of the contracts to the incorrect addresses, potentially leading to complete loss of ownership of the contracts.

Additionally, if the protocol does not plan on renouncing ownership in future, it is highly recommended that the renounceOwnership function is overridden and disabled from functioning or being called by adding a revert statement. This is because projects can accidentally call this function leading to permanent loss of ownership of the contracts and admin functions.

If not checked it could result in compromise and misuse of the admin function and loss of control of admin via accidental ownership transfer or renouncement

Impact:

This can lead to a critical loss of funds if the owner or admin loses his private key, but the likelihood is low so this issue is rated as low. But it is recommended to fix as this issue can quickly escalate to a highly critical one.

Recommendation:

It is highly recommended that:

1. The project uses multisig wallets such as that of Gnosis for the owners of the contacts with at least 5/9 or 6/11 configuration.
2. `Ownable2step` is used instead of Ownable and `Ownable2StepUpgradeable` instead of OwnableUpgradeable.

INFORMATIONAL-1 (BEST ENGINEERING PRACTICES) | RESOLVED

sellTokens Is Marked As Payable For Unknown Reasons

Description
The sellTokens function has the payable keyword even though there is no reference to msg.value. Users who transfer native tokens to this function will lose funds.

Recommendation:
Remove the payable keyword.

INFORMATIONAL-2 (BEST ENGINEERING PRACTICES) | RESOLVED

Unused Storage Variable aerodromeRouterAddress

Description
There appears to be no reference to the aerodromeRouterAddress in the AcesFactory contract except for modifying the variable.

Recommendation:
Remove the aerodromeRouterAddress variable.

Upgradeability feature for AcesLaunchpadToken is not necessary

The AcesFactory contract is set as the owner of the token when a token is created and launched.

```
   function createToken(
        Curves curve,
        uint256 steepness,
        uint256 floor,
        string memory name,
        string memory symbol,
        string memory salt,
        uint256 tokensBondedAt
    ) public returns (address) {
...

     bytes32 saltPacked = keccak256(abi.encodePacked(salt, msg.sender));
    address tokenAddress = Clones.cloneDeterministic(tokenImplementation,
saltPacked);
        // initialize clone
     AcesLaunchpadToken(tokenAddress).initialize(name, symbol,
msg.sender, tokensBondedAt);
...
}
```

This ownership of the token is renounced when the tokensBondedAt is reached after user buy tokens. During this lifecycle there is no point where the token can be upgraded or need be upgraded.

Impact:
This can lead to unnecessary complications in code understanding and maintenance and potentially lead to confusion and hence was rated informational.

Recommendation:
Thus, it is advised to remove the upgradeability feature of the token if it is not necessary to make the code simpler and to avoid unnecessary complications.

	AcesFactory.sol AcesLaunchpadToken.sol AcesToken.sol FixedMath.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the ACES team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the ACES team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo