// Public Smart Contract Security Assessment

04.24.2024 - 05.28.2024

Staking
Altcoinist

HALBORN

Staking - Altcoinist

Prepared by: HALBORN

Last Updated 09/10/2024

Date of Engagement by: April 24th, 2024 - May 28th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
23	1	6	5	2	9

TABLE OF CONTENTS

Altcoinist engaged Halborn to conduct a security assessment on their smart contracts beginning on April 24th, 2024, and ending on May 28th, 2024. Additionally, Halborn conducted a security review on the provided updated code-base, from August 19th, 2024 to August 22nd, 2024. The security assessments were scoped to the smart contracts provided in the altcoinist-com/contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

Halborn was provided 34 (thirty-four) days for the initial engagement, and 3 (three) days for the additional engagement, and assigned 2 (two) full-time security engineers to review the security of the smart contracts in scope. The engineers are blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security issues, that were mostly addressed by the Altcoinist team. The main security issues were:

- Inconsistencies in the calculation of user shares, particularly in the StakingVault contract.
- Implement safeguards against reward manipulation, particularly in scenarios involving multiple deposits or withdrawals.
- Implement consistent input validation across all functions to prevent unexpected behaviors.
- Rewards are distributed to stakers as intended.
- Users can withdraw their principal stake amount at any time.
- Penalties are properly applied to inactive subscribers.

`Halborn` performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

Captures whether the attack requires compromising a specific account.

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to

smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Measures the impact to the deposits made to the contract by either users or owners.

Measures the impact to the yield generated by the contract for either users or owners.

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	
High	
Medium	

SEVERITY	SCORE VALUE RANGE
Low	
Informational	

FILES AND REPOSITORY

(a) Repository: contracts

(b) Assessed Commit ID: 40fbb1f

(c) Items in scope:

- src/ALTT.sol
- src/AuthorTokenFactory.sol
- src/PaymentCollector.sol
- src/StakingFactory.sol
- src/StakingVault.sol
- src/SubscribeRegistry.sol
- src/XPRedeem.sol
- src/Notifier.sol
- altcoinist-com/contracts/commit/d4abfeca2ec63b1147e06866b5e2ffe99b81eae9

Out-of-Scope:

FILES AND REPOSITORY

(a) Repository: contracts

(b) Assessed Commit ID: 37dbffd

(c) Items in scope:

- TWAP.sol

Out-of-Scope:

FILES AND REPOSITORY

(a) Repository: contracts

(b) Assessed Commit ID: db67d35

(c) Items in scope:

- f8fec96ddf1ba4b9e51cfadf4f2bcde7c76bf532
- 3f33ecd7e5cddf2f18131667f08ade1ebc37e9cb

Out-of-Scope:

REMEDIATION COMMIT ID:

- 1366351
- 3a82adc
- 7a37c64
- aeeccfb
- 97bc3a7

- 35945a9
- e74789a
- d812e2e
- d383fc2
- f33a5b2
- b1ca0f6
- 3ba0391
- e3ed074

Out-of-Scope: New features/implementations after the remediation commit IDs.

CRITICAL HIGH MEDIUM LOW INFORMATIONAL

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
	CRITICAL	SOLVED - 08/13/2024
	HIGH	SOLVED - 08/23/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
	HIGH	SOLVED - 08/13/2024
	HIGH	SOLVED - 06/10/2024
	HIGH	SOLVED - 06/10/2024
	HIGH	SOLVED - 06/14/2024
	HIGH	SOLVED - 08/23/2024
	MEDIUM	RISK ACCEPTED
	MEDIUM	SOLVED - 06/11/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
	MEDIUM	SOLVED - 06/10/2024
	MEDIUM	SOLVED - 08/23/2024
	MEDIUM	SOLVED - 06/10/2024
	LOW	SOLVED - 08/23/2024
	LOW	SOLVED - 06/10/2024
	INFORMATIONAL	SOLVED - 06/10/2024
	INFORMATIONAL	SOLVED - 06/10/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
	INFORMATIONAL	ACKNOWLEDGED
	INFORMATIONAL	ACKNOWLEDGED
	INFORMATIONAL	SOLVED - 08/13/2024
	INFORMATIONAL	SOLVED - 08/13/2024
	INFORMATIONAL	SOLVED - 08/23/2024
	INFORMATIONAL	SOLVED - 08/23/2024
	INFORMATIONAL	SOLVED - 08/23/2024

// CRITICAL

Description

The exponential decay logic in StakingVault aims to reduce the rewards earned by late stakers, but it unintentionally reduces the principal amount of stakers as well. This occurs because the logic reduces the shares minted to users, thereby diminishing their redeeming capacity. The current implementation is as follows:

```
for (uint256 i = 0; i < daysSinceStart; i++) {

    shares = (shares * 9965) / 10000;

}
```

This reduces the shares each day, effectively slashing the principal amount of the stakers.
Example:

- **Alice deposits 10,000e18 ALTT on Day 1**
  - Shares minted: 9965e18
- **Bob deposits 10,000e18 ALTT on Day 251**
  - Shares minted: 4147e18

Bob instantly loses approximately 50% of his principal stake amount.
Additionally, if Bob tries to stake an extra 1000e18 ALTT, he will be unable to do so because the system will attempt to redeem his 10,000 ALTT first. This requires approximately, 7052 shares, leading to an `ERC20InsufficientBalance` error due to insufficient shares.
In summary, the logic's intention to reduce rewards for late stakers is overshadowed by its unintended consequence of reducing stakers' principal amounts and causing potential errors in additional staking attempts.

Proof of Concept

Following is the test function which can be added to provided PoC file:

```
function test_deposit_async_POC() public postTGE {
    console.log("========  Day 1  ======== ");
    console.log("[+] Alice buys 1 month Sub and stakes 10,000 ALTT");
    _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);
    console.log("Alice's shares: ", IStakingVault(authorVault).balanceOf(alice));

    skip(250 days);

    console.log("========  Day 251  ======== ");
    console.log("[+] Bob buys 1 month Sub and stakes 10,000 ALTT");
    _subscribe(bob, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);
    console.log("Bob's shares: ", IStakingVault(authorVault).balanceOf(bob));

    skip(30 days);

    vm.prank(owner);
    altt.transfer(bob, 1000e18);

    vm.prank(bob);
    altt.approve(authorVault, 1000e18);

    vm.prank(bob);
    IERC4626(authorVault).deposit(1000e18, bob);
    console.log("Bob's ALTT Bal:", IERC20(altt).balanceOf(bob));
    console.log("Carol Vault's ALTT Bal:", IERC20(altt).balanceOf(authorVault));
}
```

BVSS

AO:A/AC:L/AX:L/C:H/I:C/A:C/D:C/Y:H/R:N/S:U (10.0)

Recommendation

Modify the logic to prevent slashing of the principal stake and should only affect the staker's rewards.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/1366351db0b7267a4523b3239ff1d9169646b384

References

altcoinist-com/contracts/src/StakingVault.sol#L103C1-L106C35

// HIGH

Description

In the `StakingVault` contract, there's a variable shadowing issue in the `_deposit` function that leads to incorrect calculation of shares. The `daysSinceLastDeposit` variable is declared twice, with the inner declaration shadowing the outer one. This results in the outer variable being used for calculations instead of the intended inner variable, potentially leading to incorrect share allocations.

- src/StakingVault.sol

```
uint256 daysSinceLastDeposit = 0;
if (lastDeposit[receiver].timestamp > 0) {
    uint256 daysSinceLastDeposit = Math.min(1 + (block.timestamp - lastDeposit[receiver].timestamp)/(1
}

// Later in the function
for(; i<daysSinceLastDeposit; i++) {
    r = (r*9965)/10000;
}
```

In this code, the `daysSinceLastDeposit` used in the for loop is always 0, regardless of the calculation inside the if statement. This is because the inner declaration creates a new variable that shadows the outer one, and this inner variable is not accessible outside the if block.

Proof of Concept

The following test case demonstrates the impact of the incorrect share calculation due to the shadowed `daysSinceLastDepost` variable:

```
  function test_last_deposit_POC() public {
        console.log("========  Day 1  ======== ");
        console.log("[+] Bob buys 1 month Sub and stakes 10,000 ALTT");
        _subscribe(bob, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);
        console.log("Bob's shares: ", IStakingVault(authorVault).balanceOf(bob));
```

```
        skip(250 days);

        console.log("========  Day 251  ======== ");
        console.log("[+] Bob buys 1 month Sub and stakes 10,000 ALTT");
        _subscribe(bob, carol, SubscribeRegistry.packages.MONTHLY, 2, 10000e18, address(0), true);
        console.log("Bob's shares: ", IStakingVault(authorVault).balanceOf(bob));
    }
```

```
Ran 1 test for test/POC.t.sol:POC
[PASS] test_last_deposit_POC() (gas: 1307971)
Logs:
  0
  1000000000
  ========  Day 1  ========
  [+] Bob buys 1 month Sub and stakes 10,000 ALTT
  days since last 0
  Bob's shares:  996500000000000000000
  ========  Day 251  ========
  [+] Bob buys 1 month Sub and stakes 10,000 ALTT
  PENALTY ISSUED
  days since last 0
  Bob's shares:  5343459176889392716968

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 20.50s (2.89s CPU time)
```

Notice that the `days since last` value is `0` instead of `250`.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:L (8.1)

Recommendation

Remove the `uint256` type declaration from the inner assignment to fix the shadowing issue:

```
uint256 daysSinceLastDeposit = 0;
```

```
if (lastDeposit[receiver].timestamp > 0) {
    daysSinceLastDeposit = Math.min(1 + (block.timestamp - lastDeposit[receiver].timestamp)/(1 days), 5
}
```

This change ensures that the `daysSinceLastDeposit` variable is correctly updated and used in subsequent calculations.

Remediation Progress

SOLVED: The suggested mitigation was implemented by the **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// HIGH

Description

A vulnerability exists in the `SubscribeRegistry:subscribe()` function, allowing subscribers to effectively pay 32% less than the subscription fees to the author by manipulating the referral address. Specifically, a subscriber can set the referral address (`ref`) to an address they control, which has subscribed to the author either in the present or past. This enables them to exploit the referral system and gain a significant discount.
In this code, the function checks if the `ref` address is not `address(0)`, and if so, it verifies that the `ref` has previously subscribed to the author.

```
if (ref != address(0)) {
    require(subDir[author][ref] != 0, "RNS");
    toAuthor = (basePrice * 4800) / 10000; // 48%
    toPool = (basePrice * 1200) / 10000; // 12%
    toReferer = (basePrice * 3200) / 10000; // 32%
    toTeam = basePrice - toAuthor - toPool - toReferer; // 8%
    weth.safeTransferFrom(sender, ref, toReferer);
} else {
    toAuthor = (basePrice * 8000) / 10000; // 80%
    toPool = (basePrice * 1200) / 10000; // 12%
    toTeam = basePrice - toAuthor - toPool;
}
```

The issue lies in this check:

```
require(subDir[author][ref] != 0, "RNS");
```

This requirement only ensures that the referral address has subscribed at some point in the past, without verifying if the subscription is currently active.

Consequently, a subscriber can:

1. Subscribe with `address1` for a month.
2. After the subscription expires, subscribe again using `address2`, setting `address1` as the referral address.
3. Continue to receive a 32% "cashback" by exploiting the referral system, as the check passes due to the historical subscription of `address1`.

This manipulation reduces the author's share from 80% to 48%, effectively allowing the attacker to pay only 68% of the base price (100% - 32%).

Proof of Concept

Add the following test to the provided PoC file

```
    function test_referrer_discount() public {
        address addr1 = makeAddr("addr1");
        address addr2 = makeAddr("addr2");

        console.log("[+] Alice buys 1 month Sub with addr1");
        _subscribe(addr1, carol, SubscribeRegistry.packages.MONTHLY, 1, 0, address(0), true);

        skip(35 days);

        console.log("Addr1 balance: ", IERC20(WETH).balanceOf(addr1));
        console.log("[+] Alice buys 1 month Sub with addr2 setting referrer as addr1");
        _subscribe(addr2, carol, SubscribeRegistry.packages.MONTHLY, 1, 0, addr1, true);
        console.log("Addr1 balance: ", IERC20(WETH).balanceOf(addr1));
    }
```

Run `forge test --mt "test_referrer_discount" -vvv`

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:H/Y:N/R:N/S:U (8.8)

Recommendation

To fix this issue, the check should enforce that the referral address has an active subscription, not just a historical one. For example:

```
require(subDir[author][ref] > block.timestamp, "RNS");
```

This ensures that the referral address must have a current, valid subscription for the referral discount to apply.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/1366351db0b7267a4523b3239ff1d9169646b384

References

altcoinist-com/contracts/src/SubscribeRegistry.sol#L127C1-L133C59

// HIGH

Description

In `SubscribeRegistry`, after the TGE, WETH is swapped for ALTT before being sent as a "reward" to the author pool.
Below is the relevant code snippet from `SubscribeRegistry::_distribute()`:

```
if (altt.totalSupply() > 0) { // Rewards are only sent after TGE
    if (StakingVault(vault).totalSupply() > 0) {
        uint256 alttReceived = swapWethForAltt(toPool);
        TransferHelper.safeApprove(address(altt), vault, alttReceived);
        TransferHelper.safeTransfer(address(altt), vault, alttReceived);
    }
} else {
    uint256 sent = StakingVault(vault).depositWeth(vault, toPool);
    require(sent == toPool);
}
```

After TGE, the first condition checks if the total supply of ALTT is greater than 0, ensuring rewards are only distributed post-TGE. Then, it verifies that the total supply of shares in the author vault is not zero.
Consequently, the fees from the first subscriber after TGE will not be sent to the pool and will remain as WETH in `SubscribeRegistry` as no shares have been minted yet.

Proof of Concept

Add the following test to the provided PoC test file:

```
function test_fees_loss_POC() public postTGE {
    console.log("[+] Alice buys 1 month Sub and stakes 10,000 ALTT");
    _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);
```

```
        //console.log("Pool's ALTT Balance: ", altt.balanceOf(authorVault));
        //MONTHLY FEES = 1e18 WETH (~100e18 ALTT)
        //Pool's portion = 12% of MONTHLY = 12e18
        //Pool balance should be ~10,012 (10,000 stake amount + 12 ALTT as fees)
        assertApproxEqAbs(altt.balanceOf(authorVault), 10000e18 + 12e18, 0.2e18);
    }
```

Run with `forge test --mt "test_fees_loss_POC" -vvv`

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:H/R:N/S:U (8.8)

Recommendation

Remove the following condition in the `_distribute()` function: `if (StakingVault(vault).totalSupply() > 0 )`

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/7a37c64e6e4c618d962041e1b9f8753a0cb7552d

References

altcoinist-com/contracts/src/SubscribeRegistry.sol#L143

// HIGH

Description

To stake, a subscriber must have an active subscription to the author.
However, in the `StakingVault::_deposit` function, the following checks are used to restrict staking:

```
require(author != address(0), "UI");
require(authorTokenFactory.balanceOf(receiver, uint256(uint160(author))) > 0, "PD");
```

First, the function ensures the author's address is non-zero, and then checks if the subscriber holds an ERC1155 token corresponding to the author. This token is minted when a user subscribes to the author for the first time. However, this mechanism does not guarantee that the subscriber currently has an active subscription.
Due to this flaw, a past subscriber can stake in the author's pool and accrue rewards without an active subscription. These rewards cannot be redeemed until the subscriber renews the subscription. Once renewed, the subscriber becomes eligible to claim the rewards earned during the inactive subscription period.

Proof of Concept

Add the following function to the test PoC file:

```
function test_stake_noSub() public postTGE {
        console.log("========  Day 1  ======== ");
        console.log("[+] Alice buys 1 month Sub and stakes 10,000 ALTT");
        _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);

        skip(30 days);

        console.log("========  Day 31 ======== ");
        deal(address(altt), alice, 500e18);
        console.log("[+] Alice stakes 500 ALTT more even if her subscription is expired");
        vm.startPrank(alice);
```

```
        altt.approve(address(authorVault), 500e18);
        IERC4626(authorVault).deposit(500e18, alice);
}
```

Run `forge test --mt "test_stake_noSub" -vvv`

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:H/R:N/S:U (8.8)

Recommendation

Add the following check to `StakingVault::_deposit()`:

```
require(registry.getSubDetails(author, owner) > block.timestamp);
```

This ensures that the staker has an active subscription before they stake.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/aeeccfb46664a9c9ed8803d1b9eff2c1c2a05801

References

altcoinist-com/contracts/src/StakingVault.sol#L94

// HIGH

Description

While the user is not subscribed to an author, they must not be able to earn yield in the staking pool.
This is enforced using a penalty system. The penalty is supposed to slash off the rewards earned during the "not subscribed" period, but instead, it merely delays those rewards.
For example, let's assume there is only one staker:

- Alice subscribes for 1 month and stakes 1000 ALTT.
- After 30 days, the yield generated in the vault from fees is 100 ALTT, which fully belongs to Alice.

Now Alice's subscription has ended.

- After 180 days, the yield generated is 1000 ALTT. Alice should not be able to claim this yield since she does not have an active subscription. The penalty system enforces this restriction.

Alice subscribes for 7 months now.

- For the next 7 months, the yield generated is 1200 ALTT.
- She can now claim the rewards from the 30 days and the 7 months (100 ALTT + 1200 ALTT), whereas she should only be able to claim the rewards for the 30 days and the 7 months (100 ALTT + 1200 ALTT), not the 180 days (1000 ALTT).

Proof of Concept

Add the following test to the provided PoC file

```
    function test_yield_generation_POC() public postTGE {
        console.log("========  Day 1  ======== ");
        console.log("[+] Alice buys 1 month Sub and stakes 10,000 ALTT");
        _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), true);

        skip(20 days);
        console.log("========  Day 21  ======== ");
```

```
        console.log("[*] Simulating total fees accrual in author pool");
        _addReward(100e18);

        skip(180 days);
        console.log("========  Day 201  ======== ");
        console.log("[*] Simulating total fees accrual in author pool");
        _addReward(1000e18);

        console.log("[+] Alice's Unlocked Rewards:", IStakingVault(authorVault).unlockedRewards(alice));

        //To claim all the rewards, alice now subscribes again for 7 months, offset the penalty
        console.log("[+] Alice Resubscribes for 7 months");
        _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 7, 0, address(0), true);

        skip(190 days);
        console.log("========  Day 391  ======== ");
        console.log("[*] Simulating total fees accrual in author pool");
        _addReward(1000e18);

        //The penalty that alice faces is delay in claiming the rewards INSTEAD of her not recieving those rewards in
the first place
        console.log("[+] Alice's Unlocked Rewards:", IStakingVault(authorVault).unlockedRewards(alice));
    }
```

Run `forge test --mt "test_yield_generation_POC" -vvv`

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:H/R:N/S:U (8.1)

Recommendation

Modify the penalty logic to slash off the rewards permanently instead of delaying the rewards.

Remediation Progress

SOLVED: A new logic was implemented which correctly slashes the user's rewards during their inactive time.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/97bc3a7a41be0f8e91a6d46bde5ad17101333845

References

altcoinist-com/contracts/src/StakingVault.sol#L268C1-L273C41

// HIGH

Description

The `depositWeth` function in the `StakingVault` contract contains a vulnerability that allows malicious actors to reset the reward accrual timestamp for any user. This can be exploited to prevent legitimate users from accruing their rewards, effectively performing a Denial of Service attack on the reward mechanism.

- `src/StakingVault.sol`

```
function depositWeth(address receiver, uint256 amount) public nonReentrant returns (uint256) {
    // ... (other checks)
    else {
            // after TGE+LP
            uint256 amountOut = swapWethForAltt(amount);
            _deposit(self, receiver, amountOut, amountOut);
            notifier.notifyWethDeposit(creator, receiver, amount, amountOut);
            return amountOut;
        }
    // ...
}
```

The vulnerability stems from the fact that anyone can call `depositWeth` on behalf of any other user, even with a minimal amount (1 wei). This call updates the `lastDepost[owner].timestamp` to the current block timestamp, which is used to calculate the unlocked rewards.
An attacker can repeatedly call this function with minimal amounts, constantly resetting the timestamp and preventing the target user from accruing significant rewards. This attack can be sustained until the user's subscription expires, at which point they may have lost all their potential rewards.

Proof of Concept

The provided test case demonstrates how Bob can reset Alice's rewards to zero by depositing just 1 wei of WETH into Alice's staked amount:

```
    function test_DoS_Rewards_POC() public {
        deal(WETH, bob, 10e18);
        vm.startPrank(bob);
        IERC20(WETH).approve(address(authorVault), 10e18);
        vm.stopPrank();

        console.log("========  Day 1  ======== ");
        console.log("[+] Alice buys 9 month Sub and stakes 10,000 ALTT");
        _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 9, 10000e18, address(0), true);

        skip(20 days);
        console.log("========  Day 21  ======== ");
        console.log("[+] Total Fees Accrued: 100e18");
        _addReward(100e18);

        console.log("[+] Alice's Unlocked Rewards:", IStakingVault(authorVault).unlockedRewards(alice))

        console.log("[+] Bob deposits 1 wei of WETH (converted to ALTT) into Alice's staked amount");
        vm.prank(bob);
        IStakingVault(authorVault).depositWeth(alice, 1);

        console.log("[+] Alice's Unlocked Rewards:", IStakingVault(authorVault).unlockedRewards(alice))
    }
```

Output:

```
Ran 1 test for test/POC.t.sol:POC
[PASS] test_DoS_Rewards_POC() (gas: 1151347)
Logs:
  0
  1000000000
  ========  Day 1  ========
  [+] Alice buys 9 month Sub and stakes 10,000 ALTT
  days since last 0
  ========  Day 21  ========
```

```
  [+] Total Fees Accrued: 100e18
  [+] Alice's Unlocked Rewards: 142558433868844052357680
  [+] Bob deposits 1 wei of WETH (converted to ALTT) into Alice's staked amount
  days since last 0
  [+] Alice's Unlocked Rewards: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 38.19s (8.59s CPU time)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (7.5)

Recommendation

To address this vulnerability, consider implementing one or more of the following measures:

- Restrict `depositWeth` to only allow deposits by the receiver themselves:

```
require(msg.sender == receiver, "Only the receiver can deposit");
```

- Implement a minimum deposit amount that is significant enough to discourage frequent small deposits:

```
require(amount >= MINIMUM_DEPOSIT, "Deposit amount too low");
```

Remediation Progress

SOLVED: A check was added which prevents a user from depositing for anyone else.

```
if (receiver != _msgSender() && _msgSender() != address(registry) && receiver != self) {
          revert("cannot deposit for else");
}
```

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// MEDIUM

Description

The code lines mentioned in the `References` section indicates that a minimum amount of 0 output tokens from the swap is acceptable, opening up the protocol to a loss of funds via MEV bot sandwich attacks.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:L/D:M/Y:N/R:N/S:U (6.9)

Recommendation

Allow the caller to specify a maximum slippage or minimum amount of output tokens to be received from the swap, such that the swap will revert if it wouldn't return the caller-specified minimum amount of output tokens.
Also provide a sensible default if the caller doesn't specify a value, but user-specified slippage parameters must always override defaults.

Remediation Progress

RISK ACCEPTED: The **Altcoinist team** accepted the risk of this issue.

References

altcoinist-com/contracts/src/ALTT.sol#L92-L93
altcoinist-com/contracts/src/StakingVault.sol#L216-L217
altcoinist-com/contracts/src/SubscribeRegistry.sol#L225-L226

// MEDIUM

Description

The `SubscribeRegistry` contract contains a vulnerability in its `subscribe` function The function is designed to allow users to subscribe to a service and stake tokens in a single transaction. However, due to the order of operations, the subscription status is updated after the staking operation, leading to a race condition.

```
function subscribe(address author, address subber, packages package, uint256 qty, uint256 stakeAmount,
    // ... (earlier parts of the function)

    if (stakeAmount > 0) {
        if(altt.totalSupply() > 0) {
            TransferHelper.safeTransferFrom(address(altt), _msgSender(), self, stakeAmount);
            TransferHelper.safeApprove(address(altt), vault, stakeAmount);
            StakingVault(vault).deposit(stakeAmount, subber);
        } else {
            weth.safeTransferFrom(_msgSender(), self, stakeAmount);
            uint256 sent = StakingVault(vault).depositWeth(subber, stakeAmount);
            require(sent == stakeAmount, "failed to stake all tokens");
        }
    }

    subDir[author][subber] = expiry;  // Subscription status updated after staking

    // ... (remaining parts of the function)
}
```

The `StakingVault::deposit` function, which is called during the staking process, checks for an active subscription. As the subscription status is only updated after this call, it causes the transaction to revert, preventing users from subscribing and staking in one call.

Proof of Concept

Following is the test function which can be added to provided PoC file:

```
function test_weth_stake_fail_POC() public {
     console.log("[+] Alice buys 1 month Sub and stakes 10,000 ALTT");

     _subscribe(alice, carol, SubscribeRegistry.packages.MONTHLY, 1, 10000e18, address(0), false);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:L/D:M/Y:N/R:N/S:U (6.9)

Recommendation

To resolve this issue, the order of operations in the `subscribe` function should be modified. The subscription status should be updated before the staking operation is performed.

```
function subscribe(address author, address subber, packages package, uint256 qty, uint256 stakeAmount,
    // ... (earlier parts of the function)

    subDir[author][subber] = expiry;  // Update subscription status before staking

    if (stakeAmount > 0) {
        if(altt.totalSupply() > 0) {
            TransferHelper.safeTransferFrom(address(altt), _msgSender(), self, stakeAmount);
            TransferHelper.safeApprove(address(altt), vault, stakeAmount);
            StakingVault(vault).deposit(stakeAmount, subber);
        } else {
            weth.safeTransferFrom(_msgSender(), self, stakeAmount);
            uint256 sent = StakingVault(vault).depositWeth(subber, stakeAmount);
            require(sent == stakeAmount, "failed to stake all tokens");
        }
```

```
    }

    // ... (remaining parts of the function)
}
```

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/35945a9b40cac93087783d6cc131e752428efd9b

// MEDIUM

Description

The `start` is being set when the StakingVault implementation contract is deployed, as it's being assigned in the constructor.

```
    constructor(IERC20 _altt, SubscribeRegistry _registry)
        ERC20("", "")
        ERC4626(_altt) // duplicate due to IR compilation
    {
        wethDepositSum = 0;
        conversionDone = false;
        weth = IERC20(0x4200000000000000000000000000000000000006);
        swapRouter = IV3SwapRouter(0x2626664c2603336E57B271c5C0b26F421741e481);
        lockSwap = false;
        registry = _registry;
        start = block.timestamp;
    }
```

This will lead to all the author pools having the start time as when the staking vault implementation was deployed, instead of setting the start time when the specific author pool was created.

Proof of Concept

Add the following test to provided PoC file:

```
    function test_clone_invalidStartDate() public {
        vm.startPrank(address(registry));

        //Deploying Alice's Pool
        address aliceVault = factory.createPool(alice);
        IStakingVault(aliceVault).init(aliceVault, alice, "Test", address(factory), address(tokenFactory));
```

```
        uint256 aliceStart = IStakingVault(aliceVault).start();

        vm.warp(block.timestamp + 200 days);

        //Deploying Bob's Pool
        address bobVault = factory.createPool(bob);
        IStakingVault(bobVault).init(aliceVault, alice, "Test", address(factory), address(tokenFactory));
        uint256 bobStart = IStakingVault(bobVault).start();

        //Both start dates are same even if bob's pool is deployed after 200 days
        assertEq(aliceStart, bobStart);
    }
```

Run `forge test --mt "test_clone_invalidStartDate" -vvv`

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:U (6.3)

Recommendation

Assign the `start` value in the `init` function instead of the `constructor`.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/e74789ad535f523b3e4ed55a86765b0ff879fddd

References

altcoinist-com/contracts/src/StakingVault.sol#L52

// MEDIUM

Description

In the `penalizeUser` function of the StakingVault contract, there's a potential mismatch between how the penalty is calculated and how it's applied. The penalty is calculated in terms of assets, but it's burned in terms of shares, which could lead to inconsistencies.

- src/StakingVault.sol

```
if (deposits[owner] < balanceInAssets) {
    penalty = Math.min(
        balanceInAssets - deposits[owner],
        (inactivityRatio * balanceOf(owner)) / 1e18
    );
    _burn(owner, penalty);
}
```

The `penalty` is calculated as the minimum of two values:
1. The difference between the balance in assets and the deposits (in assets)
2. A proportion of the user's balance (in shares)
However, the `_burn` function is then called with this `penalty` value, which burns shares, not assets.
This mismatch could lead to incorrect penalty applications:
1. When the share price is greater than 1, users might be penalised more heavily than intended.
2. In extreme cases, the contract might attempt to burn more shares than the user has, potentially causing a revert
This inconsistency could lead to unfair penalties.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:M/Y:N (5.6)

Recommendation

To address the issue, the penalty calculation should be consistently done in either assets or shares, and then converted if necessary before burning. Here's a suggested fix:

```
if (deposits[owner] < balanceInAssets) {
    uint256 penaltyInAssets = Math.min(
        balanceInAssets - deposits[owner],
        (inactivityRatio * balanceInAssets) / 1e18
    );
    uint256 penaltyInShares = _convertToShares(penaltyInAssets, Math.Rounding.Up);
    _burn(owner, penaltyInShares);
}
```

Additionally, consider adding a check to ensure the calculated penalty in shares doesn't exceed the user's balance:

```
penaltyInShares = Math.min(penaltyInShares, balanceOf(owner));
```

Remediation Progress

SOLVED: The suggested mitigation was implemented by the **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// MEDIUM

Description

Anyone can call `StakingVault::initWethConversion()` after the Token Generation Event (TGE) to convert all accumulated WETH into ALTT by initiating a swap.
The function includes the following check:

```
require(wethBalance >= wethDepositSum && wethDepositSum > 0);
```

This ensures two conditions:
1. The actual WETH balance of the staking vault must be greater than or equal to the total WETH staked before the TGE.
2. The total WETH staked must be greater than 0.
However, if all users withdraw their WETH before this function is called, the remaining WETH in the contract will be stuck. This means it cannot be converted into ALTT to be distributed as rewards, effectively locking those funds in the contract.

Proof of Concept

Add the following test to provided PoC file:

```
    function test_no_WETH_conversion() public {
        console.log("[+] Alice buys Lifetime Sub and stakes 10,000 WETH before TGE");
        _subscribe(alice, carol, SubscribeRegistry.packages.LIFETIME, 1, 1000e18, address(0), true);

        test_init_TGE();
        console.log("[+] All the staked WETH is withdrawn after TGE");
        vm.prank(alice);
        IStakingVault(authorVault).withdrawWeth(10000e18);

        vm.expectRevert();
        IStakingVault(authorVault).initWethConversion();
```

```
    }
```

Run `forge test --mt "test_no_WETH_conversion" -vvvv`

BVSS

AO:A/AC:L/AX:M/C:N/I:M/A:L/D:N/Y:M/R:N/S:U (4.6)

Recommendation

Remove the `require` statement as it is redundant. By removing this check, we ensure that WETH can be converted into ALTT and distributed as rewards even if all users have withdrawn their WETH before this function is called.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/d812e2eb63c4bacfbee39dbd35c3058f73800e2c

References

altcoinist-com/contracts/src/StakingVault.sol#L197

// LOW

Description

In the `penalizeUser` function of the StakingVault contract, there's an inconsistency in the event emission logic. The `notifyPenalizedBoost` event is emitted regardless of whether a penalty is actually applied.

`- src/StakingVault.sol`

```
// we only penalize rewards
if (deposits[owner] < balanceInAssets) {
    penalty = Math.min(
        balanceInAssets - deposits[owner],
        (inactivityRatio*balanceOf(owner))/1e18
    );
    _burn(owner, penalty);
}
notifier.notifyPenalizedBoost(creator, owner, value);
```

The event is emitted outside the `if` block that determines whether a penalty is applied. This means the event will be emitted even in cases where no penalty is actually enforced (i.e., when `deposits[owner] >= balanceInAssets`).

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Move the event emission inside the `if` block to ensure it only occurs when a penalty is actually applied:

```
// we only penalize rewards
if (deposits[owner] < balanceInAssets) {
```

```
    penalty = Math.min(
        balanceInAssets - deposits[owner],
        (inactivityRatio*balanceOf(owner))/1e18
    );
    _burn(owner, penalty);
    notifier.notifyPenalizedBoost(creator, owner, value);
}
```

Remediation Progress

SOLVED: The suggested mitigation was implemented by the **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// LOW

Description

The following contracts implements the Ownable pattern:

- `AuthorTokenFactory`
- `PaymentCollector`
- `SubscribeRegistry`
- `ALTT`

However, the assessment revealed that the solution does not support the two-step-ownership-transfer pattern. The ownership transfer might be accidentally set to an inactive EOA account. In the case of account hijacking, all functionalities get under permanent control of the attacker.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:H/D:N/Y:N/R:N/S:C (2.2)

Recommendation

It is recommended to implement a two-step process (Openzeppelin's `Ownable2Step.sol`) where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

Remediation Progress

SOLVED: The suggested mitigation was applied by the **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/d383fc2f734a1c809db0ee59e9cc53e9c8229cb2

// INFORMATIONAL

Description

The following function lacks address(0) validation for certain parameters:

- *PaymentCollector::constructor* - `_owner`, `_altt`, `_registry`
- *XPRedeem:constructor* - `_altt` , `_signer`
- *ALTT::constructor* - `initialOwner`
- *StakingVault::init* - `_self`

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:H/D:N/Y:N/R:P/S:C (1.0)

Recommendation

Every input address should be checked not to be zero, especially the ones that could lead to rendering the contract unusable, lock tokens, etc. This is considered a best practice.

Remediation Progress

SOLVED: The suggested mitigation was applied by **Altcoinist team**.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/f33a5b23b3d316befd7fff768c71ef387bbac5cf

// INFORMATIONAL

Description

The function `AuthorTokenFactory::setMintWhitelist()` sets the mapping `whitelist` instead of `mintWhitelist`

```
    function setMintWhitelist(address addr, bool val) public onlyOwner {
        require(addr != address(0) && addr != registry);
        whitelist[addr] = val;
    }
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to modify the function to set `mintWhitelist` mapping or change the function name to `setWhitelist` for clarity.

Remediation Progress

SOLVED: The function name was changed to `setMintWhitelist.`

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/b1ca0f6ad9f2e51feefe46793802e1ce4abf66aa

// INFORMATIONAL

Description

In the `VaultNotifier` contract, there is a function `notifyWethDeposit` that declares a parameter `wethAmount` which is never used within the function body.

```
function notifyWethDeposit(address vault, address addr, uint256 amount, uint256 wethAmount) public only
    emit WETHDeposit(vault, addr, amount);
}
```

Score

Impact:

Likelihood:

Recommendation

To address this issue, consider one of the following options:

- Remove the unused parameter:

```
function notifyWethDeposit(address vault, address addr, uint256 amount) public onlyVault {
    emit WETHDeposit(vault, addr, amount);
}
```

- If the `wethAmount` is intended to be used but was overlooked, update the function to properly utilize this parameter:

```
event WETHDeposit(address indexed vault, address indexed addr, uint256 amount, uint256 wethAmount);

function notifyWethDeposit(address vault, address addr, uint256 amount, uint256 wethAmount) public only
```

```
    emit WETHDeposit(vault, addr, amount, wethAmount);
}
```

Remediation Progress

ACKNOWLEDGED : The **AltcoinIst team** acknowledged the issue.

// INFORMATIONAL

Description

The following contracts currently imports `forge-std/console.sol`, which is a debugging tool typically used during development and testing:

- ALTT.sol
- StakingFactory.sol
- StakingVault.sol
- SubscribeRegistry.sol
- XPRedeem.sol

This import is redundant in a production environment.

Score

Impact:

Likelihood:

Recommendation

It is recommended to remove the following import statement from the listed contract:

```
import "forge-std/console.sol";
```

Remediation Progress

ACKNOWLEDGED : The **AltcoinIst team** acknowledged the issue.

// INFORMATIONAL

Description

The constructor of the contract lacks zero address checks.

Score

Impact:

Likelihood:

Recommendation

Implement zero address checks for the ***altt and*** factory parameters in the constructor.

Remediation Progress

SOLVED: The issue has been resolved by adding validation on the parameters.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3ba03911edc759b5377885b3c7aa816940b1c2ed

// INFORMATIONAL

Description

The **iterateTWAP()** function iterates over an unbounded array shares. This iteration occurs at the end of the function:

```
for (uint256 i = 0; i < shares.length; i++) {
    uint256 ratio = (shares[i].wethAmount * 1e24) / wethSum;
    uint256 out = Math.min(altt.balanceOf(self), (amountOut * ratio) / 1e24);
    IERC20(altt).safeTransfer(shares[i].pool, out);
}
```

If the shares array grows too large, the gas required to execute this loop could exceed the block gas limit. This would make the `iterateTWAP()` function impossible to call, effectively breaking this core functionality of the contract.

Score

Impact:

Likelihood:

Recommendation

- Batch Processing: Instead of processing all shares in one transaction, implement a batching mechanism. This would allow processing a fixed number of shares per transaction, spreading the work across multiple blocks if necessary.
- Pagination: Implement a pagination mechanism that processes a subset of shares in each call. This could be done by adding start and end parameters to the function, allowing external callers to iterate through the shares in manageable chunks.

Remediation Progress

SOLVED: The issue has been resolved by adding function parameters.

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/e3ed0749d939086d773bd2b38857c3528695e73f

// INFORMATIONAL

Description

The constructor of the StakingVault contract does not perform null address checks for certain parameters. Specifically, the `_altt` and `_registry` addresses are not validated to ensure they are not the zero address (0x0).

`- src/StakingVault.sol`

```
constructor(IERC20 _altt, SubscribeRegistry _registry)
    ERC20("", "")
    ERC4626(_altt) // duplicate due to IR compilation
{
    // ... other initializations ...
    registry = _registry;
}
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add address validation checks in the constructor to ensure that neither `_altt` nor `_registry` is the zero address:

```
constructor(IERC20 _altt, SubscribeRegistry _registry)
    ERC20("", "")
    ERC4626(_altt)
{
    require(address(_altt) != address(0), "ALTT address cannot be zero");
    require(address(_registry) != address(0), "Registry address cannot be zero");
```

```
    // ... other initializations ...
    registry = _registry;
}
```

Remediation Progress

SOLVED: The suggested mitigation was implemented by the **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// INFORMATIONAL

Description

The `topUpWeth` function in the `StakingVault` contract does not include an `afterLP` check. While this function is restricted to the creator, adding this check would align it with best practices seen in other parts of the contract.

`- src/StakingVault.sol`

```
function topUpWeth(uint256 amount) public returns (uint256 amountOut) {
    require(_msgSender() == creator, "PD");
    weth.safeTransferFrom(_msgSender(), self, amount);
    amountOut = swapWethForAltt(amount);
    notifier.notifyTopupWeth(creator, amount);
}
```

The `afterLP` check is used elsewhere in the contract to ensure operations only occur after the Liquidity Pool is established. Adding this check to `topUpWeth` would maintain consistency across the contract's functions.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider adding the `afterLP` modifier to the `topUpWeth` function for consistency.

Remediation Progress

SOLVED: The suggested mitigation was implemented by the **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

// INFORMATIONAL

Description

`CreatorTokenFactory.whitelist` is never used in CreatorTokenFactory.
`TWAP.dailySum` is never used in TWAP.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider removing unused state variables.

Remediation Progress

SOLVED: The suggested mitigation was implemented by **Altcoinist team.**

Remediation Hash

https://github.com/altcoinist-com/contracts/commit/3a82adceeb2934e25b2996f2382b1294fc20afce

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity `Information` and `Optimization` are not included in the below results for the sake of report readability.

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Reentrancy in ALTT.initTGE() (src/ALTT.sol#41-47):
        External calls:
        - addLiquidity() (src/ALTT.sol#45)
                - (success,data) = token.call(abi.encodeWithSelector(IERC20.transfer.selector,to,value)) (lib/v3-periphery/contracts/libraries/TransferHelper.sol#34)
                - pool = NFTManager(nonfungiblePositionManager).createAndInitializePoolIfNecessary(WETH,address(this),100,792281625142643375935439503360) (src/ALTT.s
ol#52-58)
                - (success,data) = token.call(abi.encodeWithSelector(IERC20.approve.selector,to,value)) (lib/v3-periphery/contracts/libraries/TransferHelper.sol#48)
                - TransferHelper.safeApprove(WETH,address(nonfungiblePositionManager),amount0ToMint) (src/ALTT.sol#64)
                - TransferHelper.safeApprove(address(this),address(nonfungiblePositionManager),amount1ToMint) (src/ALTT.sol#65)
                - (tokenId,liquidity,amount0,amount1) = NFTManager(nonfungiblePositionManager).mint(params) (src/ALTT.sol#81)
                - TransferHelper.safeApprove(address(WETH),address(nonfungiblePositionManager),0) (src/ALTT.sol#84)
                - TransferHelper.safeTransfer(address(WETH),msg.sender,refund0) (src/ALTT.sol#86)
                - TransferHelper.safeApprove(address(this),address(nonfungiblePositionManager),0) (src/ALTT.sol#90)
                - TransferHelper.safeTransfer(address(this),msg.sender,refund1) (src/ALTT.sol#92)
        State variables written after the call(s):
        - renounceOwnership() (src/ALTT.sol#46)
                - _owner = newOwner (lib/openzeppelin-contracts/contracts/access/Ownable.sol#97)
        Ownable._owner (lib/openzeppelin-contracts/contracts/access/Ownable.sol#21) can be used in cross function reentrancies:
        - Ownable._transferOwnership(address) (lib/openzeppelin-contracts/contracts/access/Ownable.sol#95-99)
        - Ownable._transferOwnership(address) (lib/openzeppelin-contracts/contracts/access/Ownable.sol#95-99)
        - Ownable.owner() (lib/openzeppelin-contracts/contracts/access/Ownable.sol#56-58)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
```

The reentrancy is a false positive.

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation oper
ator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
```

```
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
AuthorTokenFactory.constructor(address,address)._registry (src/AuthorTokenFactory.sol#13) lacks a zero-check on :
                - registry = _registry (src/AuthorTokenFactory.sol#16)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

Only one valid issue regarding zero address check, which was included in the report.

```
INFO:Detectors:
Reentrancy in PaymentCollector._pull(address,address) (src/PaymentCollector.sol#27-48):
        External calls:
        - CSM(subber).execute(address(weth),0,cd) (src/PaymentCollector.sol#36)
        - registry.subscribe(author,subber,SubscribeRegistry.packages.MONTHLY,1,0,address(0)) (src/PaymentCollector.sol#39-46)
        State variables written after the call(s):
        - pullLocked = false (src/PaymentCollector.sol#47)
        PaymentCollector.pullLocked (src/PaymentCollector.sol#16) can be used in cross function reentrancies:
        - PaymentCollector._pull(address,address) (src/PaymentCollector.sol#27-48)
        - PaymentCollector.constructor(address,address,address) (src/PaymentCollector.sol#19-25)
```

The reentrancy issue was checked to ensure that it cannot be used to exploit the system as there are proper checks in place.

```
INFO:Detectors:
AuthorTokenFactory.constructor(address,address)._registry (src/AuthorTokenFactory.sol#13) lacks a zero-check on :
                - registry = _registry (src/AuthorTokenFactory.sol#16)
StakingFactory.constructor(address,address)._registry (src/StakingFactory.sol#16) lacks a zero-check on :
                - registry = _registry (src/StakingFactory.sol#18)
StakingFactory.constructor(address,address)._impl (src/StakingFactory.sol#16) lacks a zero-check on :
                - impl = _impl (src/StakingFactory.sol#19)
StakingVault.init(address,address,string,address,address)._self (src/StakingVault.sol#60) lacks a zero-check on :
                - self = _self (src/StakingVault.sol#73)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

Only one valid issue regarding zero address check, which was included in the report.

```
INFO:Detectors:
Reentrancy in StakingVault._deposit(address,address,uint256,uint256) (src/StakingVault.sol#87-113):
        External calls:
        - super._withdraw(caller,receiver,caller,refund,_convertToShares(refund,Math.Rounding.Ceil)) (src/StakingVault.sol#98)
                - returndata = address(token).functionCall(data) (lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol#96)
                - SafeERC20.safeTransfer(_asset,receiver,assets) (lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol#278)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#87)
        External calls sending eth:
        - super._withdraw(caller,receiver,caller,refund,_convertToShares(refund,Math.Rounding.Ceil)) (src/StakingVault.sol#98)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#87)
        State variables written after the call(s):
        - deposits[receiver] = 0 (src/StakingVault.sol#100)
```

```
        StakingVault.deposits (src/StakingVault.sol#31) can be used in cross function reentrancies:
        - StakingVault._deposit(address,address,uint256,uint256) (src/StakingVault.sol#87-113)
        - StakingVault._withdraw(address,address,address,uint256,uint256) (src/StakingVault.sol#115-127)
        - StakingVault.deposits (src/StakingVault.sol#31)
        - StakingVault.getDeposit(address) (src/StakingVault.sol#267-269)
        - StakingVault.getRewards(address) (src/StakingVault.sol#214-220)
        - StakingVault.maxWithdraw(address) (src/StakingVault.sol#129-132)
        - StakingVault.unlockedRewards(address) (src/StakingVault.sol#240-257)
        - StakingVault.unlockedShares(address) (src/StakingVault.sol#222-238)
Reentrancy in StakingVault._deposit(address,address,uint256,uint256) (src/StakingVault.sol#87-113):
        External calls:
        - super._withdraw(caller,receiver,caller,refund,_convertToShares(refund,Math.Rounding.Ceil)) (src/StakingVault.sol#98)
                - returndata = address(token).functionCall(data) (lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol#96)
                - SafeERC20.safeTransfer(_asset,receiver,assets) (lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol#278)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#87)
        - SafeERC20.safeTransferFrom(IERC20(asset()),caller,self,assets) (src/StakingVault.sol#108)
        External calls sending eth:
        - super._withdraw(caller,receiver,caller,refund,_convertToShares(refund,Math.Rounding.Ceil)) (src/StakingVault.sol#98)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#87)
        State variables written after the call(s):
        - _mint(receiver,shares) (src/StakingVault.sol#109)
                - _balances[from] = fromBalance - value (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#199)
                - _balances[to] += value (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#211)
        ERC20._balances (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#35) can be used in cross function reentrancies:
        - ERC20._update(address,address,uint256) (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#188-216)
        - ERC20.balanceOf(address) (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#97-99)
        - _mint(receiver,shares) (src/StakingVault.sol#109)
                - _totalSupply += value (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#191)
                - _totalSupply -= value (lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#206)
```

The above reentrancies are a false positive.

```
INFO:Detectors:
XPRedeem.redeemXP(uint256,uint256,bytes) (src/XPRedeem.sol#36-59) uses a dangerous strict equality:
        - require(bool,string)(redeemTimes[msg.sender] == 0,already redeemed) (src/XPRedeem.sol#39)
XPRedeem.startRedeem() (src/XPRedeem.sol#29-33) uses a dangerous strict equality:
        - require(bool)(msg.sender == offchainSigner && start == 0) (src/XPRedeem.sol#30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
```

The above issue mentioned in XPRedeem is a false positive.

ID	PROPERTY	RUNS	RESULT

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the

codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.