// Private Smart Contract Security Assessment 02.19.2024 - 03.08.2024

Tenderize - Tenderswap SCA

Tenderize

HALBORN

Tenderize - Tenderswap SCA - Tenderize

Prepared by: HALBORN

Last Updated 04/10/2024

Date of Engagement by: February 19th, 2024 - March 8th, 2024

Summary

88% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	3	1	4

TABLE OF CONTENTS

1. Introduction

`Tenderize` engaged Halborn to conduct a security assessment on their smart contracts beginning on **02/19/2024** and ending on **03/08/2024**. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided **two weeks** for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were partially addressed by the `Tenderize team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and 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.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Brownie`, `Anvil`, `Foundry`)

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/within the **remediation commit IDs**.

4. RISK METHODOLOGY

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.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

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.

ATTACK COMPLEXITY (AX):

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.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

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.

INTEGRITY (I):

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.

AVAILABILITY (A):

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.

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

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	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: tenderswap

(b) Assessed Commit ID: 92d6693

(c) Items in scope:

- src/LPToken.sol
- src/Swap.sol
- src/UnlockQueue.sol
- src/util/ERC721Receiver.sol
- src/util/Multicall.sol
- src/util/SelfPermit.sol

Out-of-Scope: Third-party libraries and dependencies., Economic attacks.

REMEDIATION COMMIT ID:

- 9e1cef99e1cef9

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	1	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
USERS CAN INSTANT DEPOSIT/WITHDRAW TO GET BETTER SWAPS	Medium	FUTURE RELEASE

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF SLIPPAGE CONTROL IN DEPOSIT/WITHDRAW FUNCTIONS	Medium	SOLVED - 04/07/2024
QUEUE MECHANISM DOES NOT CHECK FOR OVERWRITES IN PUSH	Medium	SOLVED - 04/07/2024
LIABILITIES COULD BE INFLATED TO FRONT-RUN USERS	Low	RISK ACCEPTED
UNUSED PARAMETER	Informational	SOLVED - 04/07/2024
OPEN TODOS	Informational	SOLVED - 04/07/2024
FLOATING PRAGMA	Informational	SOLVED - 04/07/2024
CONTRACT PAUSE FEATURE MISSING	Informational	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 USERS CAN INSTANT DEPOSIT/WITHDRAW TO GET BETTER SWAPS

// MEDIUM

Description

The protocol allows users and smart contract to perform deposits and withdraws in order to provide liquidity to the pool and receive incentives based on all fees collected in swaps, also allowed by the protocol.
When a swap is executed, fees and returned amount of tokens are calculated by a formula defined in `_quote` function:

```
function _quote(address asset, uint256 amount, SwapParams memory p) internal view
returns (uint256 out, uint256 fee) {
    Data storage $ = _loadStorageSlot();

    SD59x18 x = sd(int256(amount));
    SD59x18 L = sd(int256($.liabilities));
    SD59x18 nom;
    SD59x18 denom;

    {
        SD59x18 sumA = p.u.add(x);
        sumA = sumA.mul(K).sub(p.U).add(p.u);
        sumA = sumA.mul(p.U.add(x).div(L).pow(K));

        SD59x18 sumB = p.U.sub(p.u).sub(K.mul(p.u)).mul(p.U.div(L).pow(K));

        nom = sumA.add(sumB).mul(p.S.add(p.U));

        denom = K.mul(UNIT.add(K)).mul(p.s.add(p.u));
    }
    SD59x18 baseFee = BASE_FEE.mul(x);
    fee = uint256(baseFee.add(nom.div(denom)).unwrap());

    fee = fee >= amount ? amount : fee;
    unchecked {
        out = amount - fee;
    }
}
```

This formula, as it was expected, uses the `liabilities` which are increased/decreased by deposits/withdraws and collected fees, in order to return the amount of tokens and fees in a swap. Then,

tokens are transferred to the user and the received tokens are `unlocked` in the protocol.
However, it has been identified that in the process an user can perform instant deposits prior to a swap in order to receive a better swap, increasing the amount in return and decreasing the fees to pay without receiving any penalty for manipulating the liquidity of the pool in a single transaction by depositing and withdrawing underlying tokens.

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function test_swapFlashLoan() public {
    uint256 liquidity = 100 ether;
    underlying.mint(address(this), liquidity);
    underlying.approve(address(swap), liquidity);
    swap.deposit(liquidity);

    uint256 amount = 10 ether;
    uint256 tokenId = _encodeTokenId(address(tToken0), 0);

    vm.mockCall(address(tToken0), abi.encodeWithSelector(Tenderizer.unlock.selector,
amount), abi.encode(0));
    vm.mockCall(address(tToken0),
abi.encodeWithSelector(Tenderizer.unlockMaturity.selector, 0),
abi.encode(block.number + 100));

    uint256 initialState = vm.snapshot();

    console.log("[-] Balance before:", underlying.balanceOf(address(this)));
    console.log("[-] Liquidity before:", swap.liquidity());

    tToken0.mint(address(this), 10_000 ether);
    tToken0.approve(address(swap), 150 ether);
    (uint256 out, uint256 fee) = swap.swap(address(tToken0), 10 ether, 5 ether);

    uint256 regularOut = out;

    console.log("[-] Out balance:", out);
    console.log("[-] Fee:", fee);
    console.log("[-] Balance after:", underlying.balanceOf(address(this)));
    console.log("[-] Liquidity after:", swap.liquidity());
    console.log("");

    vm.revertTo(initialState);

    uint256 loops = 2;
    for (uint256 i; i < loops; ++i) {
        console.log("[+] Balance before:", underlying.balanceOf(address(this)));
```

```
        console.log("[+] Liquidity before:", swap.liquidity());

        // Flash loan
        uint256 loanAmount = 100_000 ether;
        underlying.mint(address(this), loanAmount);
        underlying.approve(address(swap), loanAmount);
        swap.deposit(loanAmount);

        tToken0.mint(address(this), 10_000 ether);
        tToken0.approve(address(swap), 150 ether);
        (out, fee) = swap.swap(address(tToken0), 10 ether, 5 ether);

        console.log("[*] Out balance:", out);
        console.log("[*] Fee:", fee);
        console.log("[+] Balance after:", underlying.balanceOf(address(this)));

        swap.withdraw(loanAmount);
        underlying.burn(address(this), loanAmount);

        console.log("[+] Liquidity after:", swap.liquidity());
        console.log("[!] Profit:", out - regularOut);
        console.log("");
    }
}
```

Below can be seen the described behavior. The first output block shows the value resulting from a simple swap, and successive outputs show results from swaps with a prior deposit as it's described in the test above.

```
[PASS] test_swapFlashLoan() (gas: 1050451)
Logs:
  [-] Balance before: 0
  [-] Liquidity before: 10000000000000000000
  [-] Out balance: 9992500000000000000
  [-] Fee: 7500000000000000
  [-] Balance after: 9992500000000000000
  [-] Liquidity after: 9000000000000000000

  [+] Balance before: 0
  [+] Liquidity before: 10000000000000000000
  [*] Out balance: 9994999999997507488
  [*] Fee: 5000000002492512
  [+] Balance after: 9994999999997507488
  [+] Liquidity after: 9000000000000000000
  [!] Profit: 2499999997507488

  [+] Balance before: 9994999999997507488
  [+] Liquidity before: 9000000000000000000
  [*] Out balance: 9994999999962612278
  [*] Fee: 5000000037387722
  [+] Balance after: 19989999999960119766
  [+] Liquidity after: 8000000000000000000
  [!] Profit: 2499999962612278
```

BVSS

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

Recommendation

It is recommended to add any kind of cold-down/penalty mechanism in order to prevent from instant deposits and withdraws that could arbitrarily manipulate the formula involved in swaps.

Remediation Plan

PENDING: The **Tenderize team** accepted the risk of this finding and states that it will be considered in future releases.

7.2 LACK OF SLIPPAGE CONTROL IN DEPOSIT/WITHDRAW FUNCTIONS

// MEDIUM

Description

The protocol allows users and smart contract to add liquidity to the pool in exchange for some shares (`LPTokens`), `deposit` and `withdraw` functions provides this functionality to the smart contract.
These functions do not implement a way to limit the slippage when deposits/withdraws are performed.
This condition affects specially to `EOA` since they don't have a way to verify the amount of tokens received and revert the transaction in case they are too few compared to what was expected to be received.
Applying this security consideration would help to `EOA` to avoid being front-run and losing tokens in transactions towards these smart contracts.
src/Swap.sol#L173

```
function deposit(uint256 amount) external returns (uint256 lpShares) {
    Data storage $ = _loadStorageSlot();

    // Transfer tokens to the pool
    underlying.safeTransferFrom(msg.sender, address(this), amount);

    // Calculate LP tokens to mint
    lpShares = _calculateLpShares(amount);

    // Update liabilities
    $.liabilities += amount;

    // Mint LP tokens to the caller
    lpToken.mint(msg.sender, lpShares);

    emit Deposit(msg.sender, amount, lpShares);
}
```

src/Swap.sol#L198

```
function withdraw(uint256 amount) external {
    Data storage $ = _loadStorageSlot();

    uint256 available = liquidity();

    if (amount > available) revert InsufficientAssets(amount, available);

    // Calculate LP tokens to burn
    uint256 lpShares = _calculateLpShares(amount);
```

```
    // Update liabilities
    $.liabilities -= amount;

    // Burn LP tokens from the caller
    lpToken.burn(msg.sender, lpShares);

    // Transfer tokens to caller
    underlying.safeTransfer(msg.sender, amount);

    emit Withdraw(msg.sender, amount, lpShares);
}
```

BVSS

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

Recommendation

It is recommended to add another parameter to the aforementioned functions which allows users to specify a minimal amount of tokens to receive in exchange for each deposit or withdraw and revert in case this value is not reached.

Remediation Plan

SOLVED: The **Tenderize team** solved this finding by adding and extra parameter to `deposit`/`withdraw` function in order to control the slippage when there is an interaction with those functions, as it was recommended.

Remediation Hash

https://github.com/Tenderize/tenderswap/commit/9e1cef97e6bb40570a8104e7df1fc2b159084ad5

7.3 QUEUE MECHANISM DOES NOT CHECK FOR OVERWRITES IN PUSH

// MEDIUM

Description

The protocol implements a `queue` structure in order to store different data related to swap mechanism and liquidity replenishment in the pool. The contract in charge of this task is `UnlockQueue`, which stores all nodes as part of the queue in an `index => Node` mapping containing all Node struct pointing to its data and next/previous nodes in the queue.
The function used to add new data into the queue is named `push`, and it allows specifying the queue struct where the operation will be executed and the item to store. However, since the mapping where the nodes are stored by using indexes, the push function could overwrite values in the queue by specifying already in use indexes in the struct.
src/UnlockQueue.sol#L111-L114

```
function push(UnlockQueue.Data storage q, Item memory unlock) internal {
    uint256 tail = q._tail;
    uint256 newTail = unlock.id;

    q.nodes[newTail].data = unlock;
    q.nodes[newTail].prev = tail;

    if (tail == 0) {
        q._head = newTail;
    } else {
        q.nodes[tail].next = newTail;
    }

    q._tail = newTail;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

It should be verified whether the `id` specified as an argument is already in use by any other node in the storage.

Remediation Plan

SOLVED: The **Tenderize team** solved this finding by validating if a node already exists in the queue, as it was recommended.

Remediation Hash

https://github.com/Tenderize/tenderswap/commit/9e1cef97e6bb40570a8104e7df1fc2b159084ad5

7.4 LIABILITIES COULD BE INFLATED TO FRONT-RUN USERS

// LOW

Description

Since the protocol allows to deposit/withdraw liquidity, LPs must receive some shares in exchange for each deposit, in this case, by minting new shares when a deposit is executed and burning involved shares when a withdraw is triggered. The amount of shares to mint/burn in each case is calculated by the following function:

src/Swap.sol#L537-L548

```
function _calculateLpShares(uint256 amount) internal view returns
(uint256) {
    Data storage $ = _loadStorageSlot();

    uint256 supply = lpToken.totalSupply();

    if (supply == 0) {
        return amount;
    }

    return amount * supply / $.liabilities;
}
```

In addition to this basic formula, when a user deposits liquidity the liabilities increase at the same time as the LPToken supply increases, this also happens during withdraws but decreasing both values instead. Therefore, the deposit/withdraw process does not allow inflating liabilities nor shares. However, there is a path which can be used to inflate `liabilities` since `buyUnlock` and `redeemUnlock` functions add fees directly to the pool as liabilities. Nonetheless, this situation is very unlikely to happen and it would require the usage of huge amounts of swaps to inflate enough the liabilities through fees.

src/Swap.sol#L354

```
$.liabilities += unlock.fee - reward;
```

src/Swap.sol#L449

```
$.liabilities += fee;
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:L (3.8)

Recommendation

It is important to implement slippage mechanism in deposit/withdraws to prevent front-runs as well as check to prevent from minting 0 shares during the deposit process.
Also, pool deployers can protect against this type of attack by making an initial deposit of a non-trivial amount of the asset, such that price manipulation becomes infeasible.

Remediation Plan

RISK ACCEPTED: The **Tenderize team** accepted the risk of this finding.

7.5 UNUSED PARAMETER

// INFORMATIONAL

Description

During the execution of swap and quote functions, there is an internal function (`_quote`) which is being called to calculate quotes based on several arguments such as an amount and different parameters from the pool's current state.
However, this function specifies an extra argument name `address asset` which is not used in any part of the function.
src/Swap.sol#L454-L480

```
function _quote(address asset, uint256 amount, SwapParams memory p) internal view
returns (uint256 out, uint256 fee) {
    Data storage $ = _loadStorageSlot();

    SD59x18 x = sd(int256(amount));
    SD59x18 L = sd(int256($.liabilities));
    SD59x18 nom;
    SD59x18 denom;

    {
        SD59x18 sumA = p.u.add(x);
        sumA = sumA.mul(K).sub(p.U).add(p.u);
        sumA = sumA.mul(p.U.add(x).div(L).pow(K));

        SD59x18 sumB = p.U.sub(p.u).sub(K.mul(p.u)).mul(p.U.div(L).pow(K));

        nom = sumA.add(sumB).mul(p.S.add(p.U));

        denom = K.mul(UNIT.add(K)).mul(p.s.add(p.u));
    }
    SD59x18 baseFee = BASE_FEE.mul(x);
    fee = uint256(baseFee.add(nom.div(denom)).unwrap());

    fee = fee >= amount ? amount : fee;
    unchecked {
        out = amount - fee;
    }
}
```

Score

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

Recommendation

It is recommended to remove `address asset` argument from `_quote` function.

Remediation Plan

SOLVED: The **Tenderize team** solved this finding by removing the unused parameter as it was recommended.

Remediation Hash

https://github.com/Tenderize/tenderswap/commit/9e1cef97e6bb40570a8104e7df1fc2b159084ad5

7.6 OPEN TODOS

// INFORMATIONAL

Description

Open To-dos can point to architecture or programming issues that still need to be resolved. Often these kinds of comments indicate areas of complexity or confusion for developers. This provides value and insight to an attacker who aims to cause damage to the protocol.
src/Swap.sol#L31

```
pragma solidity >=0.8.19;

// TODO: UUPS upgradeable

SD59x18 constant BASE_FEE = SD59x18.wrap(0.0005e18);
UD60x18 constant RELAYER_CUT = UD60x18.wrap(0.1e18);
UD60x18 constant MIN_LP_CUT = UD60x18.wrap(0.1e18);
SD59x18 constant K = SD59x18.wrap(3e18);
```

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 resolving the To-dos before deploying code to a production context. Use an independent issue tracker or other project management software to track development tasks.

Remediation Plan

SOLVED: The **Tenderize team** solved this finding by removing TO-DO comments from contracts.

Remediation Hash

https://github.com/Tenderize/tenderswap/commit/9e1cef97e6bb40570a8104e7df1fc2b159084ad5

7.7 FLOATING PRAGMA

// INFORMATIONAL

Description

Smart contracts in scoped folder use the floating pragma `>=0.8.19`. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a `pragma` version too new which has not been extensively tested.

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 locking the pragma version with known bugs for the compiler version by removing the `>=` symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan

SOLVED: The **Tenderize team** solved this finding by setting a fixed pragma version.

Remediation Hash

https://github.com/Tenderize/tenderswap/commit/9e1cef97e6bb40570a8104e7df1fc2b159084ad5

7.8 CONTRACT PAUSE FEATURE MISSING

// INFORMATIONAL

Description

It was identified that no high-privileged user can pause any of the scoped contracts. In the event of a security incident, the owner would not be able to stop any plausible malicious actions. Pausing the contract can also lead to more considered decisions.

Score

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

Recommendation

It is recommended including the pause feature in the contracts.

Remediation Plan

ACKNOWLEDGED: The **Tenderize team** acknowledged this finding.

8. AUTOMATED TESTING

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.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.
