our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Kaizen

SECURITY REVIEW

Date: 19 December 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Kaizen

Powering a Lightning-Fast deflation with Electrifying buy and burn.

Volt, built on TitanX, is a hyper-deflationary token with a unique auction system. It features a capped supply with all tokens distributed in the first 10 days, triggering full deflation afterward. Volt utilizes 80% of system value to buy tokens + 8% of value for growing bonded liquidity growth. Volt enters deflation quickly with a massive buy and burn.

Learn more about Volt's concept and the technicalities behind it here.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors

- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days with a total of 160 hours dedicated by 4 researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified two Medium-severity issues, a possible sandwich attack and the possibility of small stuck amounts of tokens by adding liquidity. The report also highlighted other recommendations like missing validation checks and confused token usage.

The Kaizen development team has excelled in protocol development, testing, and their security approach, including conducting multiple security reviews.

5.1 Protocol Summary

Project Name	**Kaizen**
Repository	kaizen-contracts
Type of Project	DeFi, Staking
Audit Timeline	5 days
Review Commit Hash	a618924ff102034566dea66255035c8dfe52b673
Fixes Review Commit Hash	df0855d8439480bf101e0fb30c8a2832c08b4ab2

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/Treasury.sol	28
src/Auction.sol	164
src/const/Constants.sol	23
src/BuyAndBurn.sol	190
src/Kaizen.sol	40
src/VoltBurn.sol	80
src/interfaces/IShogun.sol	4
Total	**525**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **2**
- **Low** issues: **1**
- **Informational** issues: **1**

ID	Title	Severity	Status
[M-01]	`VoltBurn.buyNSendToVoltTreasury()` Function Is Subjected To A Sandwich Attack	Medium	Acknowledged
[M-02]	Small Amounts Of Tokens Will Be Stuck In The Contract When `Auction.addInitialLiquidity()` Is Called	Medium	Acknowledged
[L-01]	Constructor In `VoltBurn.sol` And `BuyAndBurn.sol` Does Not Check For Zero Addres	Low	Fixed
[I-01]	`Auction.addInitialLiquidity()` Uses The `MinVoltAmount` Wording Instead Of `MinShogunAmount`	Informational	Fixed

7. Findings

[M-01] `VoltBurn.buyNSendToVoltTreasury()` Function Is Subjected To A Sandwich Attack

Severity

Medium Risk

Description

The `VoltBurn.buyNSendToVoltTreasury()` swaps Shogun for Volt using UniswapV2 `swapExactTokensForTokensSupportingFeeOnTransferTokens()` function.

The caller sets the minimum volt amount, which can be zero.

This function can be called by anyone if `privateMode` is not enabled.

A user can conduct a sandwich attack by doing a swap in the pool beforehand, calling `buyNSendToVoltTreasury()` with 0 slippage, and swapping back their tokens.

Location of Affected Code

File: src/VoltBurn.sol#L72

```
function buyNSendToVoltTreasury(uint256 _amountVoltMin, uint32 _deadline)
    external
    notExpired(_deadline)
    onlyEOA
    notAmount0(erc20Bal(shogun))
{
    if (privateMode) require(isPermissioned(msg.sender),
       OnlyPermissionedAddressInPrivateMode());

    State storage $ = state;

    require(block.timestamp - $.intervalBetweenBurns >= $.lastBurnTs,
       IntervalWait());
    uint256 balance = erc20Bal(shogun);

    if (balance > $.swapCap) balance = $.swapCap;

    uint256 incentive = wmul(balance, $.incentive);

    balance -= incentive;

    uint256 voltBalanceBefore = volt.balanceOf(address(this));
    _swapShogunForVolt(balance, _amountVoltMin, _deadline);
    uint256 voltAmount = volt.balanceOf(address(this)) -
       voltBalanceBefore;

    emit SentToVoltTreasury(voltAmount);

    volt.transfer(VOLT_TREASURY, voltAmount);
    shogun.transfer(msg.sender, incentive);

    totalVoltSentToTreasury += voltAmount;
    $.lastBurnTs = uint32(block.timestamp);
}
```

Scenario

Let's say the pool has 1000 Shogun and 100 Volt tokens. Originally, if an honest user were to call burn with 1000 Shogun (balance in the contract is 1000), the contract would get back 50 Volt, following the formula x * y = k (2000 Shogun, 50 Volt).

Instead, a malicious user can swap 9000 Shogun directly into the liquidity pool first. He will get back 90 volt tokens. The pool will now be 10000S - 10V tokens.

Then, the malicious user calls burn with 1000 Shogun. The pool will now become 11000S - 9.09V, and the contract gets back 0.91 Volt.

The malicious user then swaps the 90 Volt token back, and now the pool becomes 99.09 Volt * 9991 Shogun. The user successfully extracted 9991 - 9000 = 991 Shogun.

The same goes for `BuyAndBurn.swapShogunForKaizenAndBurn()` The caller can set 0 for slippage.

Impact

The user can extract tokens from the swap due to zero slippage.

Recommendation

If private mode is disabled, ensure that the slippage is still set at an arbitrary amount that the owner can control (eg 90%) to prevent large sandwich attacks.

Team Response

Acknowledged.

[M-02] Small Amounts Of Tokens Will Be Stuck In The Contract When `Auction.addInitialLiquidity()` Is Called

Severity

Medium Risk

Description

In `Auction.addInitialLiquidity()`, the `SHOGUN-KAIZEN` pair is created if not already created, and `_checkPoolValidity()` is called. `_checkPoolValidity()` calls `UniswapV2Pair.skim()` which transfers excess tokens to the contract. Afterwards, `addLiquidity()` is called to add `4B SHOGUN` with `5M KAIZEN`.

File: src/Auction.sol#L178

```
function addInitialLiquidity(uint256 minVoltAmount, uint256
    minKaizenAmount, uint32 _deadline) external onlyOwner {
  require(!hasLP, LiquidityAlreadyAdded());

  require(shogun.balanceOf(address(this)) >= INITIAL_SHOGUN_FOR_LP,
     NotEnougShogunForLiquidity());

  address kaizenShogunPool = _createPairIfNeccessary(address(shogun),
     address(kaizen));

  {
      (uint256 pairBalance, address pairAddress) = _checkPoolValidity(
         kaizenShogunPool);
      if (pairBalance > 0) {
          _fixPool(pairAddress, INITIAL_SHOGUN_FOR_LP,
             INITIAL_KAIZEN_FOR_LP, pairBalance);
      }
  }

  // code
}
```

```
function _checkPoolValidity(address pairAddress) internal returns (
   uint256, address) {
    IUniswapV2Pair pair = IUniswapV2Pair(pairAddress);

@>  pair.skim(address(this));
    (uint112 reserve0, uint112 reserve1,) = pair.getReserves();
    if (reserve0 != 0) return (reserve0, pairAddress);
    if (reserve1 != 0) return (reserve1, pairAddress);
    return (0, pairAddress);
}
```

There are 2 ways a small amount of tokens will be stuck in the contract.

1. If there are excess tokens in the liquidity pool (directly sent to the pool), when `skim()` is called and tokens are transferred to the Auction contract, the tokens will be stuck in the contract
2. When `v2Router.addLiquidity()` is called and not all the 5B Shogun and 4B tokens are used, the excess tokens will be stuck in the contract.

Impact

A small amount of tokens will be stuck in the contract.

Recommendation

Sweep the leftover funds to the owner or a trusted address once `Auction.addInitialLiquidity()` is called.

Team Response

Acknowledged.

[L-01] Constructor In `VoltBurn.sol` And `BuyAndBurn.sol` Does Not Check For Zero Address

Severity

Low Risk

Description

In the constructor of VoltBurn.sol, there is no check for zero address.

```
 constructor(address _shogun, address _volt, address _v2Router, address
    _owner) Ownable(_owner) {
    shogun = ERC20Burnable(_shogun);
    volt = ERC20Burnable(_volt);
    v2Router = _v2Router;

    state.intervalBetweenBurns = 10 minutes;
    state.swapCap = 1_000_000_000e18;
    state.incentive = 0.01e18;
}
```

The same issue is present in BuyAndBurn.sol:

```
constructor(uint32 _startTimestamp, address _shogun, address _kaizen,
    address _v2Router, address _owner)
     Ownable(_owner)
{
     if ((_startTimestamp - 14 hours) % 1 days != 0) revert
        MustStartAt2PMUTC();

     startTimeStamp = _startTimestamp;

     kaizen = Kaizen(_kaizen);
     v2Router = _v2Router;

     shogun = ERC20Burnable(_shogun);

     // code
}
```

The address is checked in `Auction.sol`:

```
constructor(
     uint32 _startTimestamp,
     address _kaizen,
     address _shogun,
     address _v2Router,
     address _v2Factory,
     address _owner
@>   ) notAddress0(_kaizen) notAddress0(_shogun) Ownable(_owner) {
     if ((_startTimestamp - 14 hours) % 1 days != 0) revert
        MustStartAt2PMUTC();

     kaizen = Kaizen(_kaizen);
     shogun = IShogun(_shogun);

     // code
}
```

Impact

Lack of sanity check may result in redeployment and wasted gas.

Recommendation

Recommend adding `notAddress0(x)` in the VoltBurn.sol constructor as well and check for the address of `_shogun`, `_volt` and the Uniswap contracts.

Team Response

Fixed.

[I-01] `Auction.addInitialLiquidity()` Uses The `MinVoltAmount` Wording Instead Of `MinShogunAmount`

Severity

Informational

Description

In `Auction.addInitialLiquidity()`, the wording is used incorrectly. `minVoltAmount` is used when it should be `minShogunAmount` since the protocol is dealing with Shogun and Kaizen tokens.

Location of Affected Code

File: src/Auction.sol#L163

```
@> function addInitialLiquidity(uint256 minVoltAmount, uint256
   minKaizenAmount, uint32 _deadline) external onlyOwner {
    require(!hasLP, LiquidityAlreadyAdded());

    // code

    v2Router.addLiquidity(
        address(shogun),
        address(kaizen),
        INITIAL_SHOGUN_FOR_LP,
        INITIAL_KAIZEN_FOR_LP,
@>      minVoltAmount,
        minKaizenAmount,
        address(this),
        _deadline
    );

    // code
}
```

Recommendation

Change `minVoltAmount` to `minShogunAmount` in the parameter and `addLiquidity()`.

Team Response

Fixed.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!