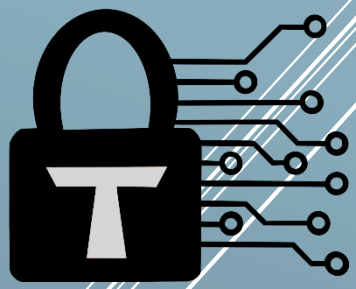
Trust Security



Smart Contract Audit

Degen Express

15/05/2024

Executive summary

Category	Launch Platforms
Audited file count	14
Lines of Code	519
Auditor	MiloTruck
Time period	05/05 - 08/05

Findings

Severity	Total	Open	Fixed	Acknowledged
High	1	0	1	0
Medium	5	0	4	1
Low	1	0	1	0

Centralization score

Signature

Digitally signed by Trust
Date: 2024.05.15 21:45:15 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	08/05/24	Client report
0.2	13/05/24	Mitigation review
0.3	15/05/24	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/Token.sol
- contracts/diamond/Diamondable.sol
- contracts/diamond/Ownable.sol
- contracts/diamond/upgradeInitializers/DegenInit.sol
- contracts/diamond/libraries/LibFakePools.sol
- contracts/diamond/libraries/LibUtils.sol
- contracts/diamond/libraries/LibDegen.sol
- contracts/diamond/libraries/LibTokens.sol
- contracts/diamond/structs/ChainlinkOracle.sol
- contracts/diamond/facets/OwnershipFacet.sol
- contracts/diamond/facets/degen/DegenTokens.sol
- contracts/diamond/facets/degen/ExpressSwap.sol
- contracts/diamond/facets/degen/DegenAdmin.sol
- contracts/diamond/facets/degen/FakePools.sol

Repository details

- **MD5 hash of ZIP file:** 72e5ed73a226c17e5709f9e3f963d4d8
- **Mitigation review MD5 hash:** 52d1ef509a95c53f88240aca5eca6d5b
- **Mitigation review #2 MD5 hash:** 8711d9a976820bfdbdbb2db27cbfbd3f

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

MiloTruck is a blockchain security researcher who specializes in smart contract security. Since March 2022, he has competed in over 25 auditing contests on Code4rena and won several of them against the best auditors in the field. He has also found multiple critical bugs in live protocols on Immunefi and is an active judge on Code4rena.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	Project is not complex, but some code could have been simplified.
Documentation	Mediocre	Project currently has no documentation.
Best practices	Good	Project consistently adheres to industry standards.
Centralization risks	Moderate	Project has some centralization risks.

Findings

High severity findings

TRST-H-1: *DegenTokens.launch()* can be permanently DOSed for individual tokens

- **Category:** Logical flaws
- **Source:** DegenTokens.sol
- **Status:** Fixed

Description

Whenever the owner calls *launch()* to launch a token, *addLiquidityETH()* from the Solidly router is called to create a new token pair and move ETH/token liquidity from the fake pool into the pair:

```
(,, uint256 lp) = d.router.addLiquidityETH{ value: eth }(
    token,
    false,
    tokens,
    tokens - 1,
    eth - 1,
    teamAlloc.beneficiary == 0 ? address(0x000000000000000000000000000000000000dEaD) :
address(this),
    block.timestamp
);
```

However, *addLiquidityETH()* successfully adds liquidity into a pair only if:

1. The pair has not been created.
2. The pair has both **tokenA** and **tokenB** reserves.

If the pair does not fulfill either of the two conditions above, *addLiquidityETH()* will revert. Therefore, an attacker can always force *addLiquidityETH()* to revert by doing the following:

1. Call *createPair()* in the Solidly factory to create the token/WETH pair.
2. Transfer some amount of WETH into the pair.
3. Call *sync()* in the pair, which updates the pair's **tokenB** reserve amount to its current WETH balance.

Now, when *launch()* is called, the pair has **tokenB** reserves but no **tokenA** reserves, which causes *addLiquidityETH()* to revert when *quoteLiquidity()* is called:

```
function quoteLiquidity(uint amountA, uint reserveA, uint reserveB) internal pure
returns (uint amountB) {
    require(amountA > 0, 'BaseV1Router: INSUFFICIENT_AMOUNT');
    require(reserveA > 0 && reserveB > 0, 'BaseV1Router: INSUFFICIENT_LIQUIDITY');
```

This will permanently DOS *launch()* for that token, causing all ETH and token liquidity to be stuck in the fake pool forever.

Recommended mitigation

Instead of using *addLiquidityETH()* to create a new token pair and add liquidity, consider calling the factory's *createPair()* function and the pair's *mint()* function directly:

```
// Create pair if it doesn't exist
address pair = d.router.pairFor(token, d.router.wETH(), false);
if (pair == address(0)) {
    pair = IBaseV1Factory(factory).createPair(token, d.router.wETH(), false);
}

// Transfer tokens and ETH to pair
token.transfer(pair, tokens);
d.router.wETH().call{ value: eth }("");
Token(d.router.wETH()).transfer(pair, eth);

// Mint LP tokens
uint256 lp = IBaseV1Pair(pair).mint(address(this));
```

Calling *mint()* directly without any form of slippage checks is safe since **token** is non-transferable before *launch()* is called. As such, it is not possible for the pair to have any liquidity beforehand – when *mint()* is called in *launch()*, it is always guaranteed to be the first mint for that pair.

Team response

Recommended mitigation adopted.

Mitigation review

Verified, the recommended mitigation was implemented. Note that LP tokens are only minted to **address(this)** when the token has a beneficiary, otherwise, the LP tokens are minted to a dead address.

Medium severity findings

TRST-M-1: Using tx.origin in *buy()* and *sell()* allows funds to be stolen from smart accounts

- **Category:** Logical flaws
- **Source:** DegenTokens.sol
- **Status:** Fixed

Description

When tokens are bought with ETH in *DegenTokens.buy()*, the tokens are transferred to **tx.origin** instead of **msg.sender**:

```
Token(token).transfer(tx.origin, tokensOut); // Transfer tokens to buyer
emit Bought(tx.origin, token, msg.value, tokensOut, newPrice);
```

Similarly, when tokens are sold for ETH in *DegenTokens.sell()*, tokens are taken from **tx.origin**, and ETH is sent to **tx.origin**:

```
Token(token).transferFrom(tx.origin, address(this), amount);
(bool sent,) = tx.origin.call{ value: ethOut }(""); require(sent);
emit Sold(tx.origin, token, ethOut, amount, newPrice);
```

However, using **tx.origin** is incompatible with smart wallets, such as Gnosis Safe. A typical smart wallet will store some functions to call with their calldata, verify the data's authenticity via signatures, and perform the requested external call. As long as the calldata is authorized via signatures, anyone can perform the call as **tx.origin**.

This allows an attacker to steal funds when a smart wallet buys tokens:

1. A Gnosis Safe user authorizes a call to *buy()* with 10 ETH.
2. An attacker executes the call to *buy()* from the EOA, using the Safe's *execTransaction()* function.
3. When *buy()* is called, 10 ETH is transferred from the victim's Safe to the contract.
4. However, bought tokens are transferred to **tx.origin** instead of the victim's Safe.

Since the tokens were not transferred to **msg.sender**, the attacker is able to steal 10 ETH worth of tokens from the victim.

Recommended mitigation

In *buy()* and *sell()*, use **msg.sender** instead of **tx.origin**:

```
- Token(token).transfer(tx.origin, tokensOut); // Transfer tokens to buyer
- emit Bought(tx.origin, token, msg.value, tokensOut, newPrice);
+ Token(token).transfer(msg.sender, tokensOut); // Transfer tokens to buyer
+ emit Bought(msg.sender, token, msg.value, tokensOut, newPrice);
```

```
- Token(token).transferFrom(tx.origin, address(this), amount);
- (bool sent,) = tx.origin.call{ value: ethOut }(""); require(sent);
- emit Sold(tx.origin, token, ethOut, amount, newPrice);
+ Token(token).transferFrom(msg.sender, address(this), amount);
+ (bool sent,) = msg.sender.call{ value: ethOut }(""); require(sent);
+ emit Sold(msg.sender, token, ethOut, amount, newPrice);
```

To accommodate this change, refactor *DegenTokens.create()* to call *FakePools._buy_FakePool()* directly for the initial buy:

```
  eth -= initialBuy;
- DegenTokens(this).buy{ value: initialBuy }(tokenAddress, t, 0);
+ (uint256 tokensOut, uint256 newPrice) = (0, 0);
+ if (t == TokenType.FakePool) {
+   (tokensOut, newPrice) = FakePools(address(this))._buy_FakePool{ value:
initialBuy }(tokenAddress);
+ }
+ Token(token).transfer(msg.sender, tokensOut); // Transfer tokens to buyer
+ emit Bought(msg.sender, origin, token, msg.value, tokensOut, newPrice);
```

Team response

Recommended mitigation adopted.

Mitigation review

The issue has been appropriately fixed in *buy()* and *sell()* by using **msg.sender** in place of **tx.origin**. However, the *DegenTokens(this).buy()* call in *create()* was not removed when it should be.

Team response

Removed the call to *DegenTokens(this).buy()* call in *create()*.

Mitigation review #2

Verified, *create()* now follows the recommended mitigation for handling initial buys.

TRST-M-2: *LibUtils.getOraclePrice()* doesn't check if the L2 sequencer is active

- **Category:** Validation issues
- **Source:** LibUtils.sol
- **Status:** Fixed

Description

LibUtils.getOraclePrice() uses Chainlink's *latestRoundData()* to fetch the price from the given price feed:

```
(
        uint80 roundID,
        int signedPrice, // answer
        /*uint startedAt*/,
        uint timeStamp, // updatedAt
        uint80 answeredInRound
) = IAggregatorV3(oracle.priceFeed).latestRoundData();
```

However, the function does not check if the L2 sequencer is active before using the price returned by the feed. If the L2 sequencer is down, the returned price will be incorrect as Chainlink will not be able to update the price feed.

If the returned price is lower than it should be, users will be able to create tokens for less than the creation price.

Recommended mitigation

Consider adding a L2 sequencer uptime check in *getOraclePrice()*, as recommended by Chainlink's documentation:

```
// Check if the sequencer is up
(, int256 answer, uint256 startedAt, , ) = sequencerUptimeFeed.latestRoundData();
require(
    answer == 0 && block.timestamp - startedAt >= GRACE_PERIOD_TIME,
    "PriceFeed: Sequencer is down"
);
```

Team response

Recommended mitigation adopted.

Mitigation review

Verified, the L2 sequencer uptime check was added to *getOraclePrice()* as recommended.

TRST-M-3: *buy()* and *sell()* could be executed on the wrong token in the event of a chain re-org

- **Category:** Re-org attacks
- **Source:** DegenTokens.sol
- **Status:** Fixed

Description

The *create()* function uses Solady's *LibClone.clone()* to deploy new tokens:

```
address tokenAddress = LibClone.clone(d.blueprintToken);
```

LibClone.clone() uses the CREATE opcode to deploy minimal proxy clones of **d.blueprintToken**. This means that **tokenAddress** is fully dependent on the **DegenToken** contract's address and nonce.

However, allowing different users to deploy tokens to the same address is dangerous as *buy()* and *sell()* identify tokens by their address:

```
function buy(address token, TokenType t, uint256 min) external payable {
```

```
function sell(address token, TokenType t, uint256 amount, uint256 min) external {
```

In the event of a chain re-org, a user calling *buy()* or *sell()* could end up executing a swap on the wrong token. For example, assume the following unfinalized blocks exist:

- Block 1: Alice calls *create()*, which deploys token A at address 0x1234…
- Block 2: Alice calls *buy()* with **tokenAddress** as 0x1234… to buy some of token A.
- Block 3: Bob calls *create()*, which deploys token B at address 0xABCD...

A chain re-org occurs, placing block 3 before block 1. The new order of execution is:

- Block 3 is executed, which deploys token B at address 0x1234…
- Block 1 is executed, which deploys token A at address 0xABCD…
- Block 2 is executed, which calls *buy()* with **tokenAddress** as 0x1234… and ends up buying some of token B.

Due to a chain re-org, Alice has mistakenly bought some of token B, although she meant to buy token A.

Recommended mitigation

Consider using *LibClone.cloneDeterministic()* with **msg.sender** in the salt used, which prevents different users from being able to deploy tokens to the same address.

For example, the salt could be the keccak256 hash of **msg.sender**, **name**, **symbol** and **description**:

```
address tokenAddress = LibClone.cloneDeterministic(
    d.blueprintToken,
    keccak256(abi.encode(msg.sender, name, symbol, description))
);
```

Team response

Recommended mitigation adopted.

Mitigation review

Verified, *cloneDeterministic()* is now used to deploy tokens. **block.timestamp** was also added as part of the salt.

TRST-M-4: *DegenTokens.launch()* can be called on tokens that haven't been locked

- **Category:** Logical flaws
- **Source:** FakePools.sol
- **Status:** Fixed

Description

Whenever tokens are bought or sold through fake pools, *checkMarketCapThreshold()* is called, which checks if the token's marketcap has exceeded 75k USD and sets **pool.locked** to true if so:

```
if (amountUsd >= d.fakePoolMCapThreshold) {
    pool.locked = true;
    emit FakePoolMCapReached(pool.token);
}
```

Once a pool is locked, the token can no longer be traded through fake pools:

```
function _buy_FakePool(address token) external onlyDiamond payable returns (uint256,
uint256) {
    LibFakePools.FakePool storage pool = LibFakePools.store().poolMap[token];
    require(pool.token != address(0) && !pool.locked);
```

```
function _sell_FakePool(address token, uint256 amount) external onlyDiamond returns
(uint256, uint256) {
    LibFakePools.FakePool storage pool = LibFakePools.store().poolMap[token];
    require(pool.token != address(0) && !pool.locked);
```

The token will have to be launched using *DegenTokens.launch()* for it to resume trading.

However, the *_launchstats_FakePool()* function, which is called by *launch()*, does not check if the token pool is locked. This means that *launch()* can be called on pools which have not yet reached 75K USD and are still tradeable through fake pools.

Whenever a token is launched, the ETH reserves in the fake pool are transferred out of the protocol and into the newly created Solidly pair as liquidity.

If the owner launches a token that hasn't been locked, an attacker could buy tokens from the Solidly pair and sell them for ETH in the protocol's fake pool. This results in insolvency as the ETH transferred out to the user belong to fake pools corresponding to other tokens.

Recommended mitigation

In *_launchstats_FakePool()*, consider checking if **pool.locked** is true:

```
  function _launchstats_FakePool(address token) external view returns (uint256,
uint256) {
      LibFakePools.FakePool storage pool = LibFakePools.store().poolMap[token];
+     require(pool.token != address(0) && pool.locked);
      return (pool.ethReserve - pool.fakeEth, pool.tokenReserve);
  }
```

This ensures that *DegenTokens.launch()* can only be called on tokens that are already locked.

Team response

Fixed by setting **pool.locked** to true in *DegenTokens.launch()*.

Mitigation review

Verified, this ensures that a token cannot be traded through fake pools once it is launched. However, note that this still allows tokens to be launched by the owner before they reach the required market cap.

TRST-M-5: Sell penalty can be reduced by selling tokens in multiple smaller amounts

- **Category:** Logical flaws
- **Source:** FakePools.sol
- **Status:** Acknowledged

Description

Whenever tokens are sold to a fake pool through *sell()*, a sell penalty is deducted from the ETH amount received by the user. This ETH amount is then added back the pool's ETH reserve:

```
function deductSellPenalty(LibFakePools.FakePool storage pool, uint256 eth) internal
returns (uint256) {
    uint256 fee = calculateSellPenalty(pool, eth);
    if (fee == 0) {
        return eth;
    } else {
        // redistribute the penaltyFee back into the ether reserve
        pool.ethReserve += fee;
        return eth - fee;
    }
}
```

However, this allows the sell penalty to be bypassed by splitting a large amount of tokens to sell into multiple smaller amounts. By calling *sell()* multiple times with smaller amounts instead of once with a huge amount, users will end up paying less of the sell penalty.

In *_create_FakePool()*, the maximum sell penalty is currently 70%:

```
require(sellPenalty <= 700);
```

The following test demonstrates that with a sell penalty of 70%, calling *sell()* twice with 0.5 ETH worth of tokens instead of once with 1 ETH worth of tokens results in an extra 0.025 ETH received:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.23;

import "forge-std/Test.sol";

contract SellPenaltyTest is Test {
    uint256 tokenReserve = 1e27; // 1 billion tokens
    uint256 ethReserve = 1.56 ether;
    uint256 sellPenalty = 700; // 70%

    function testSwap() public {
        uint256 tokens = _simulateBuy(1 ether);

        uint256 snapshot = vm.snapshot();
        uint256 ethBefore = _simulateSell(tokens);

        vm.revertTo(snapshot);
        uint256 ethAfter = _simulateSell(tokens / 2);
        ethAfter += _simulateSell(tokens / 2);

        console2.log("ETH after single sell: %e", ethBefore);
        console2.log("ETH after multiple sells: %e", ethAfter);
    }

    function _simulateBuy(uint256 eth) internal returns (uint256 tokens) {
        tokens = (eth * tokenReserve) / (eth + ethReserve);
        tokenReserve -= tokens;
        ethReserve += eth;
    }

    function _simulateSell(uint256 tokens) internal returns (uint256 eth) {
        eth = (tokens * ethReserve) / (tokens + tokenReserve);
        tokenReserve += tokens;
        ethReserve -= eth;

        uint256 fee = eth * sellPenalty / 1000;
        ethReserve += fee;
        eth -= fee;
    }
}
```

Note that the profitability of this attack depends on how large the sell penalty is.

Recommended mitigation

Consider reducing the maximum sell penalty to smaller than 70% to limit the profitability of such an attack.

Additionally, consider documenting how the sell penalty can be bypassed.

Team response

Acknowledged.

Low severity findings

TRST-L-1: Reentrancy in *DegenAdmin.reap()* allows the owner to drain all ETH in the protocol

- **Category:** Reentrancy attacks
- **Source:** DegenAdmin.sol
- **Status:** Fixed

Description

DegenAdmin.reap() can be called by anyone to transfer the protocol's proceeds to the owner:

```
function reap() external {
    (bool sent,) = owner().call{ value: LibDegen.store().proceeds }("");
    LibDegen.store().proceeds = 0;
    require(sent);
}
```

However, since **proceeds** is reset after ETH is sent to the owner, the function is vulnerable to reentrancy. If the owner address is a contract, it could re-enter *reap()* in its fallback function, which would send ETH to the owner again. This can be repeated as many times as needed to drain all ETH in the protocol.

Recommended mitigation

Set **proceeds** to 0 before sending ETH to the owner:

```
  function reap() external {
-     (bool sent,) = owner().call{ value: LibDegen.store().proceeds }("");
-     LibDegen.store().proceeds = 0;
+     uint256 eth = LibDegen.store().proceeds;
+     LibDegen.store().proceeds = 0;
+     (bool sent,) = owner().call{ value: eth }("");
      require(sent);
  }
```

Team response

Fixed by removing *reap()* and the **proceeds** variable and sending ETH to the owner whenever *gatherProceeds()* is called.

Mitigation review

Compared to the previous implementation, this new implementation is more dangerous. Since the owner address gets a callback whenever *gatherProceeds()* is called, they can re-enter or DOS *create()*, *launch()*, *buy()* and *sell()* whenever they are called.

This introduces poses a much greater centralization risk to the protocol compared to before. Consider reverting this change and implementing the recommended mitigation above.

Team response

Reverted the change and applied the recommended mitigation.

Mitigation review #2

Verified, *reap()* now sets **proceeds** to 0 before sending ETH out, as per the recommended mitigation.

Note that the *DegenAdmin.proceeds()* function that was previously removed was not added back when reverting the change. Consider adding this view function again.

Additional recommendations

TRST-R-1: Ensure token was deployed by the protocol in *DegenTokens.launch()*

Both *FakePools._buy_FakePool()* and *FakePools._sell_FakePool()* check that **pool.token** is not the zero address, which ensures that the **token** address specified by the user was deployed by the protocol. This prevents calling *DegenTokens.buy()* and *DegenTokens.sell()* with any arbitrary token address.

However, *FakePools._launchstats_FakePool()* does not have this check. This allows *DegenTokens.launch()* to be called with any arbitrary token address.

In the current implementation of the codebase, doing so would revert when attempting to calculate **tokens – 1** later on in the function.

Nevertheless, consider adding the same check to *_launchstats_FakePool()*:

```
  function _launchstats_FakePool(address token) external view returns (uint256,
uint256) {
      LibFakePools.FakePool storage pool = LibFakePools.store().poolMap[token];
+     require(pool.token != address(0));
      return (pool.ethReserve - pool.fakeEth, pool.tokenReserve);
  }
```

TRST-R-2: llamaVesting address should only be declared once

In *DegenTokens.launch()*, use the **llamaVesting** variable instead of redeclaring the LlamaVesting address again:

```
  if (teamAlloc.beneficiary == 1) {
      Token(pair).approve(address(llamaVesting), lp);
-     ILlamaVesting(0xB93427b83573C8F27a08A9090045c3e809610411a).deploy_vesting_contract(
+     llamaVesting.deploy_vesting_contract(
```

TRST-R-3: minAnswer/maxAnswer checks in *LibUtils.getOraclePrice()* are redundant

LibUtils.getOraclePrice() fetches **minAnswer** and **maxAnswer** from the price feed's aggregator, and checks if the price returned by the feed is within both values:

```
//fetch the pricefeeds hard limits so we can be aware if these have been reached.
int192 tokenMinPrice = aggregator.minAnswer();
int192 tokenMaxPrice = aggregator.maxAnswer();
//The min/maxPrice is the smallest/largest value the aggregator will post and so if it
is reached we can no longer trust the oracle price hasn't gone beyond it.
require(signedPrice < tokenMaxPrice, "Upper price bound breached");
require(signedPrice > tokenMinPrice, "Lower price bound breached");
```

However, this is a redundant check. When prices are submitted to Chainlink's price feed, they are checked to be within **minAnswer** and **maxAnswer.** As such, **signedPrice** returned by *latestRoundData()* will always be within **minAnswer** and **maxAnswer**.

Additionally, according to Chainlink's documentation, **minAnswer** and **maxAnswer** have been deprecated and are no longer in use for most price feeds.

Therefore, consider removing the **minAnwer** and **maxAnswer** check in *getOraclePrice()*.

Centralization risks

TRST-CR-1: Owner risks

Due to the permissions granted to the owner, the protocol should be considered fully centralized. The owner address can:

- Set any configuration parameter in DegenAdmin.sol to any arbitrary value. These parameters include transaction fees for swapping through fake pools, and fees for creating and launching tokens.
- *DegenTokens.launch()* can only be called by the owner. If the owner chooses not to launch a token after its fake pool is locked, the token's ETH reserves would be stuck in the protocol.

Additionally, the protocol is deployed behind a diamond proxy that is controlled by the owner. This allows the owner to upgrade the protocol's code to whatever they wish.