Smart Contract Audit

TwinFinance Audit

January 2025

SecHub

CREATED BY
JPG / SUB7 SECURITY

Contents

1 Confidentiality statement

This document is the exclusive property of Sub7 Security and Sub7 Security. This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires consent of both Sub7 Security and Sub7 Security.

2 Disclaimer

A smart contract security audit is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period.

Time-limited engagements do not allow for a full evaluation of all security controls. Sub7 Security prioritized the assessment to identify the weakest security controls an attacker would exploit. Sub7 Security recommends conducting similar assessments on an annual basis by internal or third-party assessors to ensure the continued success of the controls

3 About Sub7

Sub7 is a Web3 Security Agency, offering Smart Contract Auditing Services for blockchain-based projects in the DeFi, Web3 and Metaverse space.

Learn more about us at https://sub7.xyz

4 Project Overview

TWIN Finance Real world assets on Berachain

Welcome to TWIN, a cutting-edge DeFi protocol that ushers in the future of blockchain-based derivatives, bridging the world of traditional and digital assets.

The TWIN Finance protocol:

Empowers users to create, mint, and trade synthetic assets (TWIN assets) linked to the value of virtually any real-world and crypto assets. Offers a unique and highly appealing tokenomics model, benefitting a vibrant community of liquidity providers, investors, and active DAO governance participants. Operates as a decentralized autonomous organization (TWIN DAO), firmly rooted in community-driven principles and governance decisions. As exclusively governed through transparent voting processes by holders of veTWIN, having staked the TWIN Finance protocol token (TWIN).

5 Executive Summary

Sub7 Security has been engaged to what is formally referred to as a Security Audit of Solidity Smart Contracts, a combination of automated and manual assessments in search for vulnerabilities, bugs, unintended outputs, among others inside deployed Smart Contracts.

The goal of such a Security Audit is to assess project code (with any associated specification, and documentation) and provide our clients with a report of potential security-related issues that should be addressed to improve security posture, decrease attack surface and mitigate risk.

As well general recommendations around the methodology and usability of the related project are also included during this activity

1 (One) Security Auditors/Consultants were engaged in this activity.

5.1 Scope

https://github.com/rondras/twin-smartcontracts-oracle.git

5.2 Timeline

December 2024

5.3 Summary of Findings Identified

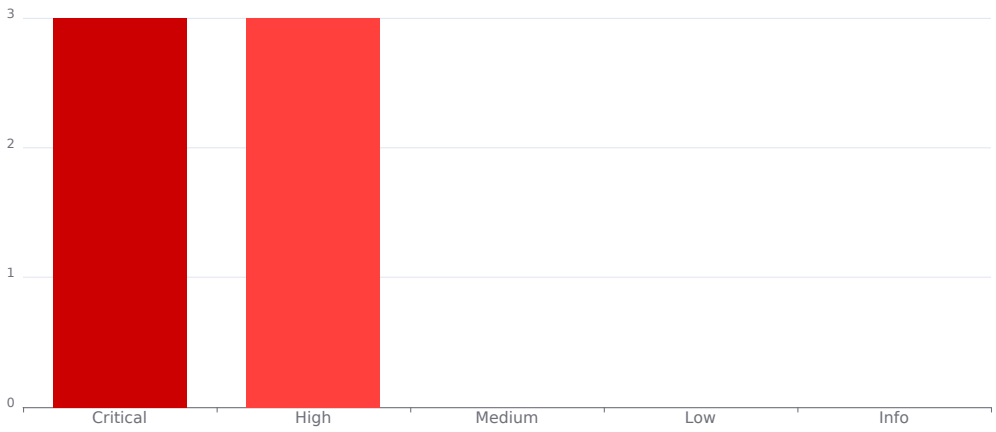

Figure 1: Executive Summary

 # 1 Critical In reward machine rewardTokenNumber is never stored in the mapping so reward will always be 0 – ***Fixed***

 # 2 Critical DOS via DAOVolume Depletion Through Dummy Proposals – ***Fixed***

 # 3 Critical Missing Expired Check in setEndOfLifeValue Allows Multiple Price Updates – ***Fixed***

 # 4 High TWAP Calculation DOS due to Solidity 0.8.0 Overflow Prevention – ***Fixed***

 # 5 High Missing Quorum Check Allows Upgrade with Minimal Participation – ***Fixed***

 # 6 High Missing Quorum Check in Grant Funding Votes – ***Fixed***

5.4 Methodology

SUB7's audit methodology involves a combination of different assessments that are performed to the provided code, including but not limited to the following:

Specification Check

Manual assessment of the assets, where they are held, who are the actors, privileges of actors, who is allowed to access what and when, trust relationships, threat model, potential attack vectors, scenarios, and mitigations. Well-specified code with standards such as NatSpec is expected to save time.

Documentation Review

Manual review of all and any documentation available, allowing our auditors to save time in inferring the architecture of the project, contract interactions, program constraints, asset flow, actors, threat model, and risk mitigation measures

Automated Assessments

The provided code is submitted via a series of carefully selected tools to automatically determine if the code produces the expected outputs, attempt to highlight possible vulnerabilities within non-running code (Static Analysis), and providing invalid, unexpected, and/or random data as inputs to a running code, looking for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Examples of such tools are Slither, MythX, 4naly3er, Sstan, Natspec-smells, and custom bots built by partners that are actively competing in Code4rena bot races.

Manual Assessments

Manual review of the code in a line-by-line fashion is the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found. This intensive assessment will check business logics, intended functionality, access control & authorization issues, oracle issues, manipulation attempts and multiple others.

Security Consultants make use of checklists such as SCSVS, Solcurity, and their custom notes to ensure every attack vector possible is covered as part of the assessment

6 Findings and Risk Analysis

6.1 In reward machine rewardTokenNumber is never stored in the mapping so reward will always be 0

Severity: Critical
Status: Fixed

Description

The createRewards() function calculates weekly rewards but fails to store them properly. While the contract correctly calculates rewardTokenNumber based on weekly rewards and voting escrow total supply, this value is stored only in a state variable and never mapped to the specific reward round.

The calculation:

```
1  solidityCopyrewardTokenNumber = weeklyRewards * votingEscrow.totalSupplyAt(snapshotByRound
       [rewardsRound]) /
2                      (maxGoveranceTokenSupply - governanceToken.balanceOf(address(this)));
```

However, the `claimRewards()` function looks for rewards in rewardTokenNumbersPerRound[round], which is never populated. This causes all reward claims to return 0 since the mapping returns the default value (0) for any round.

This disconnect between where rewards are stored (rewardTokenNumber) and where they are read from (rewardTokenNumbersPerRound) breaks the entire reward distribution mechanism.

Location

main/contracts/RewardsMachine.sol

Recommendation

Add the following line in createRewards() after the reward calculation:

```
1  solidityCopyrewardTokenNumber = weeklyRewards * votingEscrow.totalSupplyAt(snapshotByRound
       [rewardsRound]) /
2                      (maxGoveranceTokenSupply - governanceToken.balanceOf(address(this)));
3  // Add this line:
4  rewardTokenNumbersPerRound[rewardsRound] = rewardTokenNumber;
```

This ensures rewards are properly stored per round and can be accessed by the claim function.

Comments

6.2 DOS via DAOVolume Depletion Through Dummy Proposals

Severity: Critical
Status: Fixed

Description

In the DAO contract, a malicious user holding >100,000 veTokens can create multiple grant proposals using different dummy addresses as receivers to deplete the DAOVolume. Once the DAOVolume is exhausted, legitimate proposals cannot be created. The issue exists in `initiateGrantFundingVote()`:

```
function initiateGrantFundingVote(address _receiver, uint256 _amount, string calldata
    _description) {
    require(voteNumber > 100000*(10**18), 'INSUFFICIENT_ve_BALANCE');
    require(getGrantVotes[_receiver].open == false, 'VOTE_OPEN');
    require(_amount < (100000 * (10**18)), 'AMOUNT_TOO_HIGH');

    DAOVolume = DAOVolume - _amount;  // Immediately decrements DAOVolume
    // ... rest of the function
}
```

A malicious user can: 1. Create multiple EOAs as receivers 2. Create proposals for each receiver up to 100,000 tokens 3. Drain DAOVolume before proposals are voted on/closed 4. Block legitimate proposals as `DAOVolume` will be too low

Location

main/contracts/DAO.sol

Recommendation

Add a limit on active proposals per veToken holder and only deduct from DAOVolume after proposal passes:

```
mapping(address => uint256) public activeProposalsPerUser;
uint256 public constant MAX_ACTIVE_PROPOSALS = 3;

function initiateGrantFundingVote(...) {
    require(activeProposalsPerUser[msg.sender] < MAX_ACTIVE_PROPOSALS, "
        TOO_MANY_ACTIVE_PROPOSALS");
    activeProposalsPerUser[msg.sender]++;
    // Move DAOVolume deduction to closeGrantFundingVote() when proposal passes
    ...
}
```

Comments

6.3 Missing Expired Check in setEndOfLifeValue Allows Multiple Price Updates

Severity: Critical
Status: Fixed

Description

The `setEndOfLifeValue()` function lacks a check for whether an asset is already expired, allowing the end-of-life value to be set multiple times. This occurs in AssetFactory.sol:

```
function setEndOfLifeValue(string calldata _symbol) external notPaused {
    Oracle.OraclePrice memory oracleData = Oracle(oracleAddress).getPrice(_symbol);
    require(block.timestamp - oracleData.publishTime < 1 hours,'ORACLE DATA TOO OLD');
    require(block.timestamp > oracleData.publishTime,'ORACLE DATA INVALID');
    require(block.timestamp > getAsset[_symbol].expiryTime);

    // Missing check: require(getAsset[_symbol].expired == false, "ALREADY_EXPIRED");

    if (oracleData.price > getAsset[_symbol].upperLimit){
        endOfLifeValue = getAsset[_symbol].upperLimit;
    } else {
        endOfLifeValue = oracleData.price;
    }
    getAsset[_symbol].endOfLifeValue = endOfLifeValue;
    getAsset[_symbol].expired = true;
}
```

Even though the function uses oracle data (preventing price manipulation), the end-of-life value could still be updated multiple times after expiry, which violates the intended one-time finality of the expiry price setting.

This means:

- Initial users might have burned their tokens expecting one payout based on upperLimit
- Later users could get different payouts if price dropped and someone called setEndOfLifeValue again
- This creates a direct economic impact where users who burned early could get very different payouts than those who waited
- Could be exploited by waiting for favorable price conditions to “reset” the endOfLifeValue

Location

main/contracts/AssetFactory.sol

Recommendation

Add an expired check at the beginning of the function:

```
function setEndOfLifeValue(string calldata _symbol) external notPaused {
    require(getAsset[_symbol].expired == false, "ALREADY_EXPIRED");
```

```
    // rest of the function...
}
```

Comments

6.4 TWAP Calculation DOS due to Solidity 0.8.0 Overflow Prevention

Severity: High
Status: Fixed

Description

In the MarketPair contract (Uniswap fork), the price accumulator mechanism relies on intentional overflow in the price calculations. However, Solidity ^0.8.0 prevents overflows by default, breaking the intended functionality in `_update()`:

```
if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
    price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) *
        timeElapsed;
    price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) *
        timeElapsed;
}
```

The price accumulator is designed to overflow as part of its TWAP (Time Weighted Average Price) calculation mechanism. The default overflow checks in ^0.8.0 will cause the transaction to revert instead of allowing the intended overflow, leading to a DOS condition.

Location

main/contracts/MarketPair.sol

Recommendation

Add `unchecked` block around the price accumulator calculations:

```
if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
    unchecked {
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) *
            timeElapsed;
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) *
            timeElapsed;
    }
}
```

Comments

6.5 Missing Quorum Check Allows Upgrade with Minimal Participation

Severity: High
Status: Fixed

Description

The Upgrader contract allows contract upgrades to pass with any number of votes as long as yes votes exceed no votes. The issue exists in `closeUpgradeVote()`:

```
if (getUpgradeVotes[_newImplementationAddress].yesVotes > getUpgradeVotes[
    _newImplementationAddress].noVotes) {
    string memory contractName = getUpgradeVotes[_newImplementationAddress].
        contractToUpgrade;
    address payable proxyAddress = contractAddresses[contractName];
    ProxyAdmin(proxyAdminAddress).upgradeAndCall(ITransparentUpgradeableProxy(proxyAddress
        ), _newImplementationAddress, "");
}
```

This means: 1. An upgrade can pass with a single yes vote and zero no votes 2. During periods of low participation, malicious upgrades can be pushed through 3. Critical contract upgrades could be executed without adequate community review

Location

main/contracts/Upgrader.sol

Recommendation

Add a quorum check that requires a minimum percentage of total veToken supply to participate:

```
function closeUpgradeVote(address _newImplementationAddress) external {
    // ... existing checks ...

    uint256 totalVotes = getUpgradeVotes[_newImplementationAddress].yesVotes +
                         getUpgradeVotes[_newImplementationAddress].noVotes;
    uint256 totalSupply = votingEscrow.totalSupply();

    require(totalVotes >= (totalSupply * QUORUM_PERCENTAGE) / 100, "QUORUM_NOT_REACHED");

    if (getUpgradeVotes[_newImplementationAddress].yesVotes > getUpgradeVotes[
        _newImplementationAddress].noVotes) {
        // ... rest of the upgrade logic
    }
}
```

Comments

6.6 Missing Quorum Check in Grant Funding Votes

Severity: High
Status: Fixed

Description

The DAO contract's grant funding mechanism lacks a quorum requirement, allowing proposals to pass with minimal participation. In `closeGrantFundingVote()`, grants are approved solely based on yes votes exceeding no votes:

```
if (getGrantVotes[_receiver].yesVotes > getGrantVotes[_receiver].noVotes) {
    governanceToken.transfer(_receiver, getGrantVotes[_receiver].amount);
    emit grantFundingVoteClosed(_receiver, true);
}
```

This means: 1. A grant can be approved with a single yes vote (100,000 veTokens) 2. During low participation periods, malicious proposals can pass 3. Governance tokens can be distributed without proper community consensus

Location

main/contracts/DAO.sol

Recommendation

Implement a quorum requirement in `closeGrantFundingVote()`:

```
function closeGrantFundingVote(address _receiver) external notPaused {
    // ... existing checks ...

    uint256 totalVotes = getGrantVotes[_receiver].yesVotes + getGrantVotes[_receiver].
        noVotes;
    uint256 totalVeSupply = votingEscrow.totalSupply();

    // Require at least 10% of total veToken supply to participate
    require(totalVotes >= (totalVeSupply * 10) / 100, "QUORUM_NOT_REACHED");

    if (getGrantVotes[_receiver].yesVotes > getGrantVotes[_receiver].noVotes) {
        // ... rest of the grant logic
    }
}
```

Comments

FOLLOW US