#HashEx
BLOCKCHAIN SECURITY

Determinant Finance Presale

smart contracts final audit report

February 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Determinant Finance team to perform an audit of their smart contract. The audit was conducted between 09/02/2024 and 10/02/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available in the Sepolia testnet at 0xBe8a0C4D8e85Ff8cEFa942B7E4A10105E479c20F.

Update. The Determinant Finance team has responded to this report. The updated contract is available in the Ethereum mainnet at 0xbb0f552De41440234f5ffee2b8065DF05D5122a1.

2.1 Summary

Project name	Determinant Finance Presale
URL	https://www.determinant.finance/
Platform	Ethereum
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
Presale	0xbb0f552De41440234f5ffee2b8065DF05D5122a1

3. Project centralization risks

The presale is highly owner-dependent. The project owner can:

- collect payments without providing any actual reward tokens;

- stop all claims by setting contract to active state with `setpresaleEnd(false)`.

4. Found issues

Medium	3 (27%)
Low	6 (55%)
Info	2 (18%)

C9e. Presale

ID	Severity	Title	Status
C9ela9	Medium	Reward amount is calculated during claiming	Acknowledged
C9ela8	Medium	Incorrect return value	Resolved
C9elaa	Medium	Resetting contract does not update users' claim data	Resolved
C9ela5	Low	Hardcoded values	Partially fixed
C9ela6	Low	Oracle response isn't checked for stalling	Acknowledged
C9ela7	Low	Ignoring oracle price for USDT	Acknowledged
C9ela3	Low	Gas optimizations	Acknowledged
C9elab	Low	Lack of events	Acknowledged
C9elac	Low	Error message mismatch for address limit	Acknowledged

C9elad	Info	Max purchase per transaction constraint can be circumvented	Resolved
C9ela4	Info	Inconsistent comments and typographical errors	Resolved

5. Contracts

C9e. Presale

Overview

A simple single round sale contract that accepts payments both in USD-pegged ERC20 token and in native EVM currency. Price conversions are made with use of Chainlink price feeds. Purchased ERC20 tokens are transferred to buyers after the end of the sale in vesting form: 50% are available immediately, other half is split to 5 equal parts to be claimed with at least 1-month periods.

Issues

C9ela9 Reward amount is calculated during claiming

Medium Acknowledged

Reward amount is fixed upon user's request during the first call of the claimDthReward() function. This means spending $1000 worth ETH amount during the sale period may result in an arbitrary amount of DTH tokens at the time of claiming due to ETH/USD price change.

This makes impossible to maintain adequate DTH token balance of the Presale contract since it's impossible to predict the total amount of claimable DTH tokens.

Recommendation

Store purchased amount of DTH tokens instead of total ETH and USDT contributions.

C9ela8 Incorrect return value

Medium Resolved

The `getTotalSoldDth()` returns wrong values in general. Same addresses from both `participants` and `ethparticipants` lists are counted twice distorting the total value.

```
function getTotalSoldDth() public view returns (uint256) {
    uint256 totalSoldDth = 0;
    for (uint i = 0; i < participants.length; i++) {
        totalSoldDth = totalSoldDth.add(calculateTotalDthForParticipant(participants[i]));
    }
    for (uint i = 0; i < ethparticipants.length; i++) {
        totalSoldDth =
totalSoldDth.add(calculateTotalDthForParticipant(ethparticipants[i]));
    }
    return totalSoldDth;
}

function calculateDthForParticipant(address participant) public view returns (uint256) {
    uint256 ethContributionWei = ethcontributions[participant]; // Get the ETH
contribution of the participant in Wei
    uint256 ethPriceWei = getLatestEthPrice(); // Get the latest ETH price in Wei
    uint256 usdContributionWei = ethContributionWei.mul(ethPriceWei); // Convert the ETH
contribution to USD (in Wei)
    uint256 dthRewardWei = usdContributionWei.mul(40); // Calculate the DTH reward (in
Wei)
    uint256 dthReward = dthRewardWei.div(1e8);
    return dthReward;
}

function calculateTotalDthForParticipant(address participant) public view returns
(uint256) {
    uint256 dthRewardEth = calculateDthForParticipant(participant);
    uint256 dthRewardUsdt = calculateDthUsdt(participant);
    uint256 totalDthReward = dthRewardEth.add(dthRewardUsdt);
    return totalDthReward;
}
```

The `getRemainingDthForUser()` function returns wrong data: claimable amount for user is fixed with the first claim, but `getRemainingDthForUser()` re-calculates this amount dynamically with current prices.

```
 function getRemainingDthForUser(address user) public view returns (uint256) {
        uint256 totalDth = calculateTotalDthForParticipant(user);
        uint256 claimedDth = claimedTokens[user];
```

```
        if (totalDth >= claimedDth) {
            return totalDth.sub(claimedDth);
        } else {
            return 0;
        }
    }
```

Recommendation

Remove the `getTotalSoldDth()` function as the `getTotalContributionsInUsd()` function returns correct value.

Consider modifying the reward logic in order to eliminate contribution from price changes.

C9elaa Resetting contract does not update users' claim data

Medium Resolved

The contract contains a function, `resetContract()`, which is intended to reset the contract's state to its initial state. However, not all state variables are reset; for example, `lastClaimTime`, `claimCount`, and `dividedAmount` are not updated. This results in a scenario where a user who has claimed in a previous round will be unable to claim in the next one.

It must be noted that the `resetContract()` function can be called only after every user has claimed their tokens. If `resetContract()` is invoked prematurely, users who have already claimed their tokens will be unable to claim again.

Recommendation

Remove this function and opt for deploying a new contract when a reset is needed. To save on gas costs for deploying similar contracts, the minimal proxy pattern, also known as the EIP-1167 clone pattern, can be employed. This approach involves creating a single, lightweight clone (or proxy) of the master contract for each new instance, significantly reducing the deployment gas costs compared to deploying a full contract each time. This method is particularly effective for deploying multiple instances of the same contract logic while ensuring that each instance can be treated as a separate entity.

C9e1a5 Hardcoded values

Low Partially fixed

The price feed addresses are hardcoded to the ones existing only in the Sepolia testnet.

The price feeds' decimals are hardcoded to value of 8 without safety checks in the `buyTokenWithUsdt()`, `buyTokenWithEth()`, `getUserTotalContributionInUsd()`, `calculateDthForParticipant()` functions.

```
    constructor(
        address _dthToken,
        address _usdtToken,
        address _ethToken

    ) {
        DthToken = IERC20(_dthToken);
        UsdtToken = IERC20(_usdtToken);
        EthToken = IERC20(_ethToken);
        priceFeedETH =
AggregatorV3Interface(0x694AA1769357215DE4FAC081bf1f309aDC325306); // Address of the
Chainlink Price Feed contract for Token1
    priceFeedUsdt = AggregatorV3Interface(0x14866185B1962B63C3Ea9E03Bc1da838bab34C19);
    }

function getUserTotalContributionInUsd(address user) public view returns (uint256) {
    uint256 usdtContribution = contributions[user];
    uint256 ethContribution = ethcontributions[user];

    uint256 usdtInUsd = usdtContribution.mul(getLatestUsdtPrice()).div(1e6); // USDT
contributions converted to USD
    uint256 ethInUsd = ethContribution.mul(getLatestEthPrice()).div(1e18); // ETH
contributions converted to USD

    return usdtInUsd.add(ethInUsd); // Total contributions in USD
}

function buyTokenWithEth() public payable nonReentrant {
    ...
    uint256 transactionValueUsd = (msg.value.mul(ethPrice)).div(1e26); // Adjusted for
Chainlink's 8 decimal places
    ...
```

```
}
```

Recommendation

Use immutable values filled in the constructor. Implement immutable decimals variables.

Update

Price feed addresses made initializable in the constructor, but their decimals remain hardcoded. The deployed contract uses price feeds with correct decimals.

C9ela6 Oracle response isn't checked for stalling

Low Acknowledged

Price feed result is not checked for freshness in the `getLatestUsdtPrice()` and `getLatestEthPrice()` functions. Stalled oracle may return incorrect result.

```
function getLatestEthPrice() public view returns (uint256) {
    (,int price,,,) = priceFeedETH.latestRoundData();
    return uint256(price);
}

interface AggregatorV3Interface {
    function latestRoundData() external view
      returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
answeredInRound);
}
```

C9ela7 Ignoring oracle price for USDT

Low Acknowledged

Reward calculations are made with assumption of strictly pegged USDT price in the `calculateDthUsdt()` function.

```
function calculateDthUsdt(address participant) public view returns (uint256) {
    uint256 contribution = contributions[participant];
    uint256 reward = contribution.mul(40).mul(10**12);
    return reward;
```

```
}
```

C9ela3 Gas optimizations

Low Acknowledged

1. Token and price feed addresses should be declared as immutable.

2. Unnecessary read from storage in the `buyTokenWithUsdt()` and `buyTokenWithEth()` functions: `contributions[msg.sender]` and `ethcontributions[msg.sender]` variables.

3. Multiple reads from storage in the `for()` loops in the `getTotalSoldDth()` function: `participants.length` and `ethparticipants.length` variables.

4. Multiple reads from storage in the `claimDthReward()` function: `lastClaimTime[msg.sender]`, `claimCount[msg.sender]` variables.

5. Multiple reads from storage in the `for()` loops in the `resetContract()` function: `participants.length`, `ethparticipants.length`, `whitelistedAddresses.length` variables.

6. Local variable `maxTransactionValueUsd` can be omitted in the `buyTokenWithEth()` function.

7. Duplicated code: the `calculateTotalDthForParticipant()` function can be replaced by `getUserTotalContributionInUsd()` multiplied by 40.

8. No need to use the SafeMath library with Solidity 0.8

C9elab Lack of events

Low Acknowledged

The contract lacks events for tracking changes in state variables through functions such as `setPresaleStart()`, `setPresaleEnd()`, `setWhitelistStatuses()`, `setWhitelistStatus()`, and `resetContract()`. This omission reduces the transparency of the contract, making it difficult to observe and verify state changes externally.

C9elac Error message mismatch for address limit

Low Acknowledged

The `setWhitelistStatuses()` function has a constraint for a maximum of 200 addresses, but the error message incorrectly states a limit of 100 addresses. This discrepancy between the actual limit and the error message could lead to confusion. It is important to update the error message to accurately reflect the maximum of 200 addresses allowed.

```
function setWhitelistStatuses(address[] calldata _participants) external onlyOwner
nonReentrant {
    require(_participants.length <= 200, "Can't whitelist more than 100 addresses at a
time");
    for (uint i = 0; i < _participants.length; i++) {
        if (!whitelisted[_participants[i]]) {
            whitelisted[_participants[i]] = true;
            whitelistedAddresses.push(_participants[i]);
        }
    }
```

C9elad Max purchase per transaction constraint can be circumvented

Info Resolved

The maximum purchase constraint sets the maximum purchase value only for a single transaction. The same address can make several purchases, exceeding the 2000 limit. Alternatively, a contract may make several purchases in one transaction.

C9ela4 Inconsistent comments and typographical errors

Info Resolved

The terms "mounth" are used in the contract, which is presumably a typographical error. The correct terms should be "month". Misnaming variables can lead to confusion for developers, maintainers, and auditors, potentially obscuring the intent and functionality of the code.

Incorrect comment in L206: "user can claim ... every 3 minutes".

6. Conclusion

3 medium, 6 low severity issues were found during the audit. 2 medium issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

The code was provided without documentation or automated tests. We strongly recommend writing both unit and functional tests as a crucial part of ensuring that the contract functions as intended, and adding NatSpec documentation.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY