QuillAudits

AUDIT REPORT

December, 2024

For

Table of Content

Executive Summary

Project name	Dayvidende
Overview	DVE facilitates the exchange of a DVE token for a BUSD. It incorporates a referral system, rewarding users with tokens upon each sale. Owner can distribute dividends, which are claimable by users.
Audit Scope	The Scope of Audit is to analysis the Correctness, Code Quality and Security of Dayvidende Smart Contracts
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Commit Hash	1a0350e7d55a5b7edf0093f7aa9dd8402f6023f8
Language	Solidity
Blockchain	EVM
Review 1	12 November 2024 - 22 November 2024
Updated Code Received	2nd January 2025
Review 2	3rd January 2025
Fixed in	ef755c1422bfa5817939a01ca5582d720dc86f3f

Number of Issues per Severity

High	2 (25.00%)
Medium	3 (37.50%)
Low	1 (12.50%)
Informational	2 (25.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**2**	**3**	**1**	**2**
Acknowledged	0	0	0	0
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception Disorder
- Gasless Send
- Balance Equality
- Byte Array
- Transfer Forwards All Gas
- ERC20 API Violation
- Compiler Version Not Fixed
- Redundant Fallback Function
- Send Instead of Transfer
- Style Guide Violation
- Unchecked External Call
- Unchecked Math
- Unsafe Type Inference
- Implicit Visibility Level
- Access Management
- Centralization of Control
- Arbitrary Write to Storage
- Ether Theft
- Improper or Missing Events
- Logical Issues and Flaws
- Arithmetic Computations Correctness
- Race Conditions/Front Running
- Malicious Libraries
- SWC Registry
- Address Hardcoded
- Divide Before Multiply

- [x] Integer Overflow/Underflow
- [x] Dangerous Strict Equalities
- [x] Missing Zero Address Validation
- [x] Revert/Require Functions
- [x] Private Modifier
- [x] Upgradeable Safety
- [x] Using Delegatecall

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Vesting DVE tokens will also be sold in DVESale:buyTokens() due to use of DVE.balanceOf(address(this))

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVESale.sol#L52-L55

Function

buyTokens()

Description

Users can buy DVE tokens using buyTokens(), which give some percentage of DVE tokens to the referral. Tokens that referral gets are vested over a period of 1 year.

Now the problem is, tokens that are allocated to referrals will also be sold to new users because buyTokens() uses the `balanceOf(address(this))` while selling the DVE tokens. This doesn't separate/reserves the tokens that are already allocated/vested to the referrals.

1. Suppose there are 100 DVE tokens & user1 bought 50 DVE tokens.
2. His referrals get 5 DVE tokens, vested over 1 year period
3. Now, user2 should only be able to buy 45 DVE tokens but he can buy complete 50 DVE tokens ie including referral's 5 DVE tokens

```
function buyTokens(address ref, uint256 amount) external whenNotPaused {
    require(ref != msg.sender, "Cannot refer yourself");
    require(ref != address(0), "Referral have to be entered");
    require(amount > 0, "Must purchase a positive number of tokens");
@>  require(amount < DVE.balanceOf(address(this)), "There are not enough tokens left to pur- chase");
    bool paymentSuccessful = BUSD.transferFrom(msg.sender, address(this), amount); require(paymentSuccessful, "Payment was not successful");
    bool success = DVE.transfer(msg.sender, amount); require(success, "Sending failed");
    giveReferrals(ref, amount);
    amountPurchsed = amountPurchsed + amount;
    emit BuyTokens(msg.sender, amount);
    }
```

Recommendation

Track the amount of DVE tokens allocated to referrals and subtract it from DVE.balanceOf(address(this)) in buyTokens()

Referral DVE amount in buyTokens() became substantially high than the actual DVE buy amount

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVESale.sol#L76-L131

Function

giveReferrals()

Description

When a user buys DVE tokens then level one/two/three referral gets some percentage of DVE tokens. Now the problem is, when combining the level one/two/three referral DVE amount, it became substantially higher than the actual buy amount.

Let's see this via example:

1. Suppose there are 100 DVE tokens are for sale and user1 bought all 100 tokens.
2. Level1 referral will get 10% of amount ie `10 DVE` tokens
3. Level1 referrer has 10 Level2 referrals & they get 5% of amount ie 5 * 10 = `50 DVE` tokens
4. Now, each Level2 referrer has 10 referrals & they get 2.5% of amount ie 2.5 * 10 * 10 = `250 DVE` tokens
5. Total referral DVE tokens = 10(Level1) + 50(Level2) + 250(Level3) = 310 DVE tokens, whereas only 100 DVE tokens were sold

Impact would be that referral amount will be substantially high than the actual amount ie a large amount of DVE tokens will be vested, which are not present in the contract

Recommendation

Instead of giving `each` Level2 referral 5% of amount, divide 5% of amount among all level2 referral ie instead of giving 5 DVE tokens to each L2 referral, give 0.5(5 tokens/ L2.length) to each, which will add upto 5% of the amount. Same goes for Level3 referrals

Medium Severity Issues

No withdraw function for withdrawing DVE tokens from DVESale.sol

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVESale.sol#L1-L221

Description

`DVESale` contract is used for selling DVE tokens using buyTokens(). However there can be situations when some DVE tokens are left `unsold` then there is no withdraw function for owner to withdraw those extra DVE tokens from the contract.

Impact would be such that unsold DVE tokens will be `stuck` in the contract

Recommendation

Make a withdrawDVE() function for withdrawing DVE tokens or modify the existing withdraw() function by pasing an `address token` argument.

referees will receive referral DVE amount instead of 10 in buyTokens()

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVESale.sol#L117-L123

Function

giveReferrals()

Description

When a user buys DVE tokens then referrals receive some percentage of purchased DVE amount. Only first 10 level2 & level3 referrals should receive the referral DVE tokens. But due to wrong position of if-statement(which breaks the loop), 11 referrals recives the DVE token amount instead of 10.

```
 function giveReferrals(address l1Ref, uint256 amount) internal {
...
 address[] memory l2Refs = previousReferrals[l1Ref];
 for (uint256 i = 0; i < l2Refs.length; i++) {
 address l2Ref = l2Refs[i];
 addClaim(l2Ref, l2Amount);
 total += l2Amount;
 address[] memory l3Refs = previousReferrals[l2Ref];
 for (uint256 j = 0; j < l3Refs.length; j++) {
 address l3Ref = l3Refs[j];
 addClaim(l3Ref, l3Amount);
 total += l3Amount;
// Safety check to prevent gas limit issues
@>              if (j >= 10) {
 break;
}
}
// Safety check to prevent gas limit issues
@>          if (i >= 10) {
 break;
}
}
}
```

Due to the `=` sign & nature of array, loop will run form 0 to 11 ie [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Impact would be such that One extra referral/vesting will be created for level2 & level3

Recommendation

If-statements are used at the last of the loop, use/call them at the start of the loop

```diff
 for (uint256 i = 0; i < l2Refs.length; i++) {
+        if (i >= 10) {
+            break;
+        }
 address l2Ref = l2Refs[i];
 addClaim(l2Ref, l2Amount);
 total += l2Amount;
 address[] memory l3Refs = previousReferrals[l2Ref];
 for (uint256 j = 0; j < l3Refs.length; j++) {
+          if (i >= 10) {
+              break;
+          }
 address l3Ref = l3Refs[j];
 addClaim(l3Ref, l3Amount);
 total += l3Amount;
// Safety check to prevent gas limit issues
-             if (j >= 10) {
-                 break;
-             }
}
// Safety check to prevent gas limit issues
-         if (i >= 10) {
-             break;
-         }
}
```

Stale price is not checked in DVE:getLatest-BnbToUsd()

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVE.sol#L208-L212

Function

getLatestBnbToUsd()

Description

AggregatorV3Interface(bnbUsdFeed).latestRoundData() returns a variable `updatedAt`, which is the last timestamp when price was updated. For bnb/usd price oracle, heartbeat is 27s ie after every 27s price of bnb is updated.

```solidity
function getLatestBnbToUsd() public view returns (uint256) {
(, int256 price,,,) = AggregatorV3Interface(bnbUsdFeed).latestRoundData();
return uint256(price); // 8 decimals
}
```

If stale price is not checked, code will execute with prices that don't reflect the current pricing.

Recommendation

```diff
function getLatestBnbToUsd() public view returns (uint256) {
-       (, int256 price,,,) = AggregatorV3Interface(bnbUsdFeed).latestRoundData();
+       (, int256 price, , uint256 updatedAt, ) = AggregatorV3Interface(bnbUsdFeed).latestRoundData();
+       if (updatedAt < block.timestamp - 27 seconds) {
+           revert("stale price feed");
+       }
return uint256(price); // 8 decimals
}
```

Low Severity Issues

Full DVE tokens can't be sold in DVESale:buyTokens()

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVESale.sol#L53

Function

buyTokens()

Description

Full DVE tokens can't be sold in DVESale:buyTokens() due to use of `<` instead of `<=`

```solidity
 function buyTokens(address ref, uint256 amount) external whenNotPaused {
 require(ref != msg.sender, "Cannot refer yourself");
 require(ref != address(0), "Referral have to be entered");

 require(amount > 0, "Must purchase a positive number of tokens");
@>      require(amount < DVE.balanceOf(address(this)), "There are not enough tokens left to purchase");

 bool paymentSuccessful = BUSD.transferFrom(msg.sender, address(this), amount);
 require(paymentSuccessful, "Payment was not successful");

....
}
```

Recommendation

Use `<=` instead of `<` in require statement

Informational Severity Issues

dividend mapping is not used in DVESale.sol

Resolved

Description

dividend mapping is not used in DVESale.sol

```solidity
 mapping(address => uint256) public dividend;
```

Recommendation

Either use it in the buyTokens() to store the user's DVE token amount or completely remove it

Double check for address(0) & sale in recalculate-Claim()

Resolved

Path

https://github.com/parlour-dev/dayvidende-contracts-audit/blob/main/contracts/DVE.sol#L169

Function

recalculateClaim() and calculateDividend()

Description

When recalculateClaim() is called, it checks for address(0) & sale address. But when it calls calculateDividend() internally, it also checks the for address(0) & sale address.

```solidity
 function recalculateClaim(address who) internal {
// null user doesn't get dividends
@>      if (who == address(0) || who == sale) {
 return;
}
...

 uint256 oldDividend = claim.unclaimed;
@>      uint256 newDividend = calculateDividend(who, claim);
...
}
```

```solidity
function calculateDividend(address who, Claim storage claim) internal view returns (uint256 dividend)
{
// null and the sale contract don't get dividends
@>      if (who == address(0) || who == sale) {
 return 0;
}
...
}
```

Recommendation

Remove the check from calculateDividend()

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Dayvidende contracts. We performed our audit according to the procedure described above.
\ Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

In The End, Dayvidende Team Resolved all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Dayvidende smart contract. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Dayvidende smart contract. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Dayvidende Team to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

6+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1K+** Projects Secured

Follow Our Journey

AUDIT REPORT

December, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com