g8keep Security Review

Pashov Audit Group

Conducted by: rvierdiiev, ast3ros, Shaka

July 10th 2024 - July 13th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **g8keep/audit** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About g8keep

g8keep protocol is a system for deploying smart contracts that create Uniswap V2 pools with fixed buy and sell fees, locked liquidity, and mechanisms to prevent early buyers from unfairly acquiring tokens. It enforces specific rules for fee collection, liquidity provisions, and snipe protection to ensure fair token distribution and stability.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 9a105c5b0758cf2421c3378c6884962d6b5708f0

fixes review commit hash - bc6d948a68b37767e1e5a0db03768c52eff58263

Scope

The following smart contracts were in scope of the audit:

- `g8keepToken`
- `g8keepVester`
- `g8keepFactory`

7. Executive Summary

Over the course of the security review, rvierdiiev, ast3ros, Shaka engaged with g8keep to review g8keep. In this period of time a total of **13** issues were uncovered.

Protocol Summary

Protocol Name	g8keep
Repository	https://github.com/g8keep/audit
Date	July 10th 2024 - July 13th 2024
Protocol Type	Token factory

Findings Count

Severity	**Amount**
High	1
Medium	5
Low	7
Total Findings	**13**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Preventing token claims until vesting period ends	High	Resolved
[M-01]	Deployer fees cannot be higher than 255 basis points	Medium	Resolved
[M-02]	Bypass minimumDeployVestTime due to vestingEnd overflow	Medium	Resolved
[M-03]	The system is not fully usable with the Uniswap Widget	Medium	Acknowledged
[M-04]	Non-ERC20 tokens are not supported as paired tokens	Medium	Acknowledged
[M-05]	Tokens that require non-zero allowance cannot be disabled	Medium	Resolved
[L-01]	No check if msg.value is equal to _initialLiquidity	Low	Resolved
[L-02]	Deadline is set to the current block timestamp	Low	Acknowledged
[L-03]	Fees are charged during liquidity managing	Low	Acknowledged
[L-04]	Incompatibility with fee-on-transfer tokens	Low	Acknowledged
[L-05]	DOS attack on g8keepFactory token deployments	Low	Resolved
[L-06]	_treasuryWallet can be set to address(0)	Low	Resolved

[L-07]	Users can receive fewer g8keep tokens	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Preventing token claims until vesting period ends

Severity

Impact: Medium

Likelihood: High

Description

The `g8keepVester` contract, responsible for managing token vesting schedules, contains an error in its implementation that prevents users from claiming any tokens until the entire vesting period has elapsed.

The issue stems from the `deploymentVest` function not initializing the `lastClaim` timestamp when creating a new vesting schedule:

```
function deploymentVest
      (address _deployer, uint256 _tokensToDeployer, uint256 _vestTime)
        external
        returns (uint256 vestingId)
    {
        ...

                 DeploymentVesting storage deploymentVesting = deploymentVestings[vest
        deploymentVesting.recipient = _deployer;
        deploymentVesting.token = msg.sender;
        deploymentVesting.amount = _tokensToDeployer;
        deploymentVesting.vestingStart = uint40(block.timestamp);
        deploymentVesting.vestingEnd = uint40
        //(block.timestamp + _vestTime); // @audit lastClaim is not set
        ...
    }
```

This omission causes the `_vested` function to calculate an inflated `vestedAmount`:

```
function _vested(uint256 _id) internal view returns
      (DeploymentVesting storage vesting, uint256 vestedAmount) {
        vesting = deploymentVestings[_id];

        uint256 vestingStart = vesting.vestingStart;
        if (block.timestamp < vestingStart) return (vesting, 0);

        uint256 vestingEnd = vesting.vestingEnd;
        uint256 vestingAmount = vesting.amount;
        uint256 vestingClaimed = vesting.claimed;
        if (block.timestamp >= vestingEnd) return (vesting,
          (vestingAmount - vestingClaimed));

        uint256 timeSinceLastClaim = block.timestamp - vesting.lastClaim; //
        // @audit inflated since lastClaim is 0
        uint256 vestingPeriod = vestingEnd - vestingStart;
        vestedAmount =
        //(vestingAmount * timeSinceLastClaim) / vestingPeriod; // @audit vestedAmount
    }
```

Consequently, when a user attempts to claim tokens via the `claim` function, the transaction will revert due to insufficient balance, as the calculated `vestedAmount` exceeds the total vesting amount.

Recommendations

Initialize the `lastClaim` timestamp in the `deploymentVest` function:

```
function deploymentVest
      (address _deployer, uint256 _tokensToDeployer, uint256 _vestTime)
        external
        returns (uint256 vestingId)
    {
        ...

                 DeploymentVesting storage deploymentVesting = deploymentVestings[vest
        ...
+       deploymentVesting.lastClaim = uint40(block.timestamp);
        ...
    }
```

8.2. Medium Findings

[M-01] Deployer fees cannot be higher than 255 basis points

Severity

Impact: Low

Likelihood: High

Description

`g8keepFactory.deployToken` function allows the deployer to set the buy and sell fees for the token. Initially, the maximum value for these fees is 500 basis points. However, the function signature uses `uint8` for these values, which means that the maximum value that can be set is 255 basis points. This means that the deployer cannot set fees higher than 255 basis points.

```
function deployToken(
        uint256 _initialLiquidity,
        string memory _name,
        string memory _symbol,
        uint256 _totalSupply,
        address _treasuryWallet,
@>      uint8 _buyFee,
@>      uint8 _sellFee,
```

Recommendations

```
function deployToken(
        uint256 _initialLiquidity,
        string memory _name,
        string memory _symbol,
        uint256 _totalSupply,
        address _treasuryWallet,
-       uint8 _buyFee,
+       uint16 _buyFee,
-       uint8 _sellFee,
+       uint16 _sellFee,
```

[M-02] Bypass `minimumDeployVestTime` due to vestingEnd overflow

Severity

Impact: Medium

Likelihood: Medium

Description

The `g8keepFactory` contract implements a minimum vesting period for token deployers through the `minimumDeployVestTime` parameter. However, an error in the g8keepVester contract allows this check to be bypassed due to potential timestamp overflow.

In the `deployToken` function of `g8keepFactory`, there's a check to ensure the vesting time meets the minimum requirement:

```
function deployToken(
        uint256 _initialLiquidity,
        string memory _name,
        string memory _symbol,
        uint256 _totalSupply,
        address _treasuryWallet,
        uint8 _buyFee,
        uint8 _sellFee,
        address _pairedToken,
        uint256 _deployReserve,
        uint256 _deployVestTime,
        uint256 _snipeProtectionSeconds,
        bytes32 _tokenSalt
    ) external payable returns (address _tokenAddress) {
        ...
        if (

                         _buyFee > maxBuyFee || _sellFee > maxSellFee || _deployVestTi
            ...
        ) {
            revert InvalidDeploymentParameters();
        }
        ...
    }
```

However, in the `g8keepVester` contract, the `deploymentVest` function stores the vesting end time as a `uint40`, which can lead to an overflow. This overflow can result in a much shorter vesting period than intended, potentially allowing immediate token claims.

```
function deploymentVest
      (address _deployer, uint256 _tokensToDeployer, uint256 _vestTime)
        external
        returns (uint256 vestingId)
    {
        ...

                 DeploymentVesting storage deploymentVesting = deploymentVestings[vest
        deploymentVesting.recipient = _deployer;
        deploymentVesting.token = msg.sender;
        deploymentVesting.amount = _tokensToDeployer;
        deploymentVesting.vestingStart = uint40(block.timestamp);
        deploymentVesting.vestingEnd = uint40
        //(block.timestamp + _vestTime); // @audit vestingEnd can overflow
        ...
    }
```

This vulnerability allows malicious deployers to bypass the minimum vesting period set by g8keep. They could set a very large `_deployVestTime` that causes an overflow, resulting in a near-immediate vesting end time. This undermines the intended token distribution model and could lead to an unexpected token dump.

Recommendations

Check if `block.timestamp + _vestTime` is more than `type(uint40).max` and revert if it is.

[M-03] The system is not fully usable with the Uniswap Widget

Severity

Impact: Low

Likelihood: High

Description

One of the requirements for the protocol is that `the system must be fully usable with the Uniswap Widget`. However, the following incompatible features have been found:

1. During the snipe protection period, quotes on buys of the g8keep token might not return the correct amount, and

`maxSnipeProtectionBuyWithoutPenalty` is expected to be used to calculate the maximum amount of tokens that can be bought without penalty. This flow is not supported by the Uniswap Widget.

2. There is no mention in the documentation about support on fee-on-transfer tokens and, after reviewing the source code of the Uniswap Widget, there has not been found the possibility for the developer to support these tokens.

The only instance of such support in the Uniswap SDK's was found in the v2-sdk router

```
/**
         * Whether any of the tokens in the path are fee on transfer tokens, which sho
    */
    feeOnTransfer?: boolean

@>  const useFeeOnTransfer = Boolean(options.feeOnTransfer)

    let methodName: string
    let args: (string | string[])[]
    let value: string
    switch (trade.tradeType) {
      case TradeType.EXACT_INPUT:
        if (etherIn) {
@>
          methodName = useFeeOnTransfer ? 'swapExactETHForTokensSupportingFeeOnTransfe
```

However, it has not been found how to use this feature in the Uniswap Widget.

As such, using the Uniswap Widget to buy or sell the g8keep token might not work as expected.

Recommendations

Given that the sniping protection and fees on transfer are essential features of the protocol, it is recommended to use a more flexible alternative to the Uniswap Widget, or build a custom widget using Uniswap's `v2-sdk`.

[M-04] Non-ERC20 tokens are not supported as paired tokens

Severity

Impact: Medium

Likelihood: Medium

Description

In order to transfer funds from the deployer to the factory, `g8keepFactory.deployToken()` function uses erc20 `transferFrom()` function.

```
IERC20(_pairedToken).transferFrom(msg.sender, address(this), _initialLiquidity);
```

In case if there is a need to add non erc20 token as a paired token, then such a call will revert, thus such tokens are not supported by the factory.

Another place in the code, where erc20 functions are used is `g8keepFactory.withdrawToken` and `g8keepToken.withdrawToken` functions. Those functions won't work with non erc20 tokens as well.

Recommendations

Use SafeERC20 or a similar library.

[M-05] Tokens that require non-zero allowance cannot be disabled

Severity

Impact: Medium

Likelihood: Medium

Description

In the `g8keepFactory` contract, the `setPairedTokenSettings` function allows the owner to enable or disable a token as a valid paired token. At the end of the function, the Uniswap router is approved to spend an unlimited amount of the token.

```
function setPairedTokenSettings(
      addresspairedToken,
      boolallowed,
      uint256minimumLiquidity
    ) external onlyOwner {
        if (lockedTokens[pairedToken]) revert TokenLocked();

        allowedPairs[pairedToken] = allowed;
        pairedTokenMinimumLiquidity[pairedToken] = minimumLiquidity;
@>      IERC20(pairedToken).approve(UNISWAP_V2_ROUTER, type(uint256).max);
    }
```

Some tokens, such as USDT on mainnet, will revert when both the current and new allowance are non-zero, causing the transaction to revert, making it impossible to disable the token as a paired token.

```
function approve(address _spender, uint _value) public onlyPayloadSize
      (2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
@>      require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Recommendations

```
function setPairedTokenSettings(
      addresspairedToken,
      boolallowed,
      uint256minimumLiquidity
    ) external onlyOwner {
        if (lockedTokens[pairedToken]) revert TokenLocked();

        allowedPairs[pairedToken] = allowed;
        pairedTokenMinimumLiquidity[pairedToken] = minimumLiquidity;
+       if (allowed) {
            IERC20(pairedToken).approve(UNISWAP_V2_ROUTER, type(uint256).max);
+       } else {
+           IERC20(pairedToken).approve(UNISWAP_V2_ROUTER, 0);
+       }
    }
```

8.3. Low Findings

[L-01] No check if `msg.value` is equal to `_initialLiquidity`

In `g8keepFactory.deployToken` when the paired token is WETH, `msg.value` is wrapped into WETH.

```
if (_pairedToken == WETH) {
@>          (bool success,) = WETH.call{value: msg.value}("");
            if (!success) revert FailedToDepositWETH();
        } else if (msg.value > 0) {
            revert ValueNotAllowedForNonWETHPairs();
```

However, it is not checked if `msg.value` is equal to `_initialLiquidity`, so if a user sends more than `_initialLiquidity` the excess will be lost. Another user can take the excess by sending a calling `deployToken` with a `msg.value` lower than `_initialLiquidity`.

Recommendations

```
if (_pairedToken == WETH) {
++          if (msg.value != _initialLiquidity) revert ValueSentNotValid();
            (bool success,) = WETH.call{value: msg.value}("");
            if (!success) revert FailedToDepositWETH();
        } else if (msg.value > 0) {
            revert ValueNotAllowedForNonWETHPairs();
```

[L-02] Deadline is set to the current block timestamp

Severity

Impact: Medium

Likelihood: Medium

Description

`g8keepFactory.sellTokens` can be called by the factory owner to sell the tokens received as fees. The function uses `swapExactTokensForTokensSupportingFeeOnTransferTokens` to swap the tokens and uses `block.timestamp` as the deadline. As a result, the transaction can be executed at any time in the future, when the price of the tokens may be different.

```
try IUniswapV2Router02
              (UNISWAP_V2_ROUTER).swapExactTokensForTokensSupportingFeeOnTransferToken
@>
                sale.amountToSell, sale.amountOutMin, path, sale.recipient, block.time
            ) {} catch {}
```

Recommendations

It is recommended to receive the deadline as a parameter and use it in the swap function.

[L-03] Fees are charged during liquidity managing

`g8keepToken` is going to charge users when selling/buying from uniswap pair contracts. This functionality is implemented in the overridden `_transfer` function of `g8keepToken` contract.

```
function _transfer(address from, address to, uint256 amount) private {
    ...
    if (!(from == G8KEEP || to == G8KEEP)) {
        if (to == UNISWAP_V2_PAIR) {
            toAmount = _applyFees(from, amount, SELL_FEE);
        } else if (from == UNISWAP_V2_PAIR) {
            toAmount = _applyFees(from, amount, BUY_FEE);
        }
    }

    ...
}
```

However, the same function will be used when a user adds or burns liquidity in the pair contract, thus they will be charged fees as well.

[L-04] Incompatibility with fee-on-transfer tokens

The g8keepFactory contract's deployToken function is not designed to handle fee-on-transfer tokens. The issue arises because the function assumes that the exact amount of `_initialLiquidity` tokens will be received when transferring the paired token.

However, if the `_pairedToken` is a fee-on-transfer token, the `deployToken` function will receive less than `_initialLiquidity` amount due to the fee. Later when adding liquidity to the Uniswap pool, the `addLiquidity` function will revert because of insufficient balance since the contract has `_initialLiquidity - valueToG8keep - fee` instead of `_initialLiquidity - valueToG8keep`.

```
function deployToken(
        ...
    ) external payable returns (address _tokenAddress) {
        ...
        } else {
            IERC20(_pairedToken).transferFrom(msg.sender, address
            //(this), _initialLiquidity); // @audit will receive less than _initialLiq
        }
        ...

        // add liquidity to LP
        IUniswapV2Router02(UNISWAP_V2_ROUTER).addLiquidity(
            address(token),
            _pairedToken,
            token.balanceOf(address(this)),
            _initialLiquidity - valueToG8keep, // @audit addLiquidity will
            // revert because of insufficient balance
            0,
            0,
            address(this),
            block.timestamp
        );
        ...
    }
```

Check the actual token transfer by subtracting the after-transfer balance from the before-transfer balance and setting it as `actualReceived` amount.

```
function deployToken(
        ...
    ) external payable returns (address _tokenAddress) {
        ...
        } else {
+           uint256 balanceBefore = IERC20(_pairedToken).balanceOf(address
+ (this));
            IERC20(_pairedToken).transferFrom(msg.sender, address
              (this), _initialLiquidity);
+           uint256 balanceAfter = IERC20(_pairedToken).balanceOf(address
+ (this));
+           uint256 actualReceived = balanceAfter - balanceBefore;
        }
        ...

        // add liquidity to LP
        IUniswapV2Router02(UNISWAP_V2_ROUTER).addLiquidity(
            address(token),
            _pairedToken,
            token.balanceOf(address(this)),
-           _initialLiquidity - valueToG8keep,
+           actualReceived - valueToG8keep,
            0,
            0,
            address(this),
            block.timestamp
        );
        ...
    }
```

[L-05] DOS attack on g8keepFactory token deployments

The g8keepFactory contract is vulnerable to a Denial of Service (DOS) attack due to its use of `CREATE2` for deploying new `g8keepToken` instances. An attacker can predict the address of the to-be-deployed token and front-run the `deployToken` transaction to create a Uniswap V2 pool in advance, causing the deployment to fail. The original `deployToken` transaction will revert with `UniswapV2: PAIR_EXISTS` error.

The token deployments can be consistently blocked, causing financial loss to users in terms of gas fees spent on the failed deployment transactions. The attack cost is the gas fee to create the Uniswap V2 pool in advance.

[L-06] `_treasuryWallet` can be set to `address(0)`

When `updateTreasuryWallet` is called, the `newAddress` is verified to be different to address(0), which implies that the `treasuryWallet` cannot be `address(0)`. However, the `g8keepFactory` contract allows setting the `treasuryWallet` to the `address(0)` during token deployment.

It is recommended that a zero-address check be added to the `deployToken` function.

[L-07] Users can receive fewer g8keep tokens

In the snipe protection period, when users try to buy an amount of g8keep tokens that would decrease the balance below the expected balance, the more paired tokens the buyer sends, the fewer g8keep tokens they receive.

For example:

- At the time `t` calling `maxSnipeProtectionBuyWithoutPenalty` returns 100 g8keep tokens, that are priced at 1 WETH.
- If the buyer sends 1 WETH, they receive 100 g8keep tokens (minus fees).
- If the buyer sends 2 WETH, they receive 70 g8keep tokens (minus fees).

While buyers are expected to protect themselves from slippage with the `amountOutMin` parameter, the behavior described above seems counterintuitive and it is not clear if it is intended.

The documentation describes the following:

> Purchases that would decrease the balance below the expected balance are penalized exponentially by reducing the output amount.

However, it is not clear if the penalization should be applied on the excess amount or on the total amount of the operation.

Also, the following requirement

> Deployer or early buyers cannot be allowed to snipe a large percentage of the supply at low cost

Seems to suggest that a large amount of tokens could be bought in the initial phase, but at a high cost.