Rocket Joe

Audit Report

Prepared by Christoph Michel

February 1, 2022.

Contents

1 Introduction

Rocket Joe is a token launch platform where participants bid to provide liquidity for newly issued tokens.

1.1 Scope of Work

The auditors were provided with a GitHub repository at commit hash 187d6f7.

The task was to audit the contracts, consisting of the following files with their sha1 hashes:

File	SHA1
interfaces/IJoeFactory.sol	5fb31d64e427a924a296ee8db7b6e0db308ce59e
interfaces/IRocketJoeFactory.sol	1a8aed7df3d03672516c493ebb5a933efb6955f4
interfaces/ILaunchEvent.sol	7112aaba121ccbe3fe41108767d32c32f98e0ba1
interfaces/IJoeRouter02.sol	50ce203d02f09605e12e411d6189b953a72e9528
interfaces/IJoeRouter01.sol	9787e8332f5d276528ef13262c0c13fba5e1b1dc
interfaces/IRocketJoeToken.sol	41e241202b0020e6201405f6d40bee52bca960a3
interfaces/IJoePair.sol	08e168e1c986dc99b0d4eaa6ab56437d5e4b41b7
interfaces/IWAVAX.sol	7b96d3af6d7823201497a3b0fb67cb0061b6b394
LaunchEvent.sol	0df41190b279b70acd46197c3f1cf7ec0d9dafa6
RocketJoeToken.sol	85d4fe7f3745b27a5f55401127a31bb97b331fef
RocketJoeFactory.sol	53ecf1314f990aa6495b94ee985c172caf9a69bd
RocketJoeStaking.sol	e20d07dfcdd531646df2024cdc5e77b56bc21dea

The rest of the repository was out of the scope of the audit.

1.2 Security Assessment Methodology

The smart contract's code is scanned both manually and automatically for known vulnerabilities and logic errors that can lead to potential security threats. The conformity of requirements (e.g., specifications, documentation, White Paper) is reviewed as well on a consistent basis.

1.3 Auditors

Christoph Michel

2 Severity Levels

We assign a risk score to the severity of a vulnerability or security issue. For this purpose, we use 4 *severity levels* namely:

MINOR

Minor issues are generally subjective in nature or potentially associated with topics like “best practices” or “readability”. As a rule, minor issues do not indicate an actual problem or bug in the code. The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

LOW

Low-severity issues are generally objective in nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is a clear reason not to.

MEDIUM

Medium-severity issues are bugs or vulnerabilities. These issues may not be directly exploitable or may require certain conditions in order to be exploited. If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations that make the system exploitable.

HIGH

High-severity issues are directly exploitable bugs or security vulnerabilities. If unaddressed, these issues are likely or guaranteed to cause major problems or, ultimately, a full failure in the operations of the contract.

3 Discovered issues

3.1 Wrong token allocation computation for token decimals != 18 if floor price not reached (high)

In `LaunchEvent.createPair`, when the floor price is not reached (`floorPrice > wavaxReserve * 1e18 / tokenAllocated`), the tokens to be sent to the pool are lowered to match the raised WAVAX at the floor price.

Note that the `floorPrice` is supposed to have a precision of 18:

> /// @param _floorPrice Price of each token in AVAX, scaled to 1e18

The `floorPrice > (wavaxReserve * 1e18)/ tokenAllocated` check is correct but the `tokenAllocated` computation involves the `token` decimals:

```
1  // @audit should be wavaxReserve * 1e18 / floorPrice
2  tokenAllocated = (wavaxReserve * 10**token.decimals()) / floorPrice;
```

This computation does not work for `token`s that don't have 18 decimals.

Example

Assume I want to sell `1.0 wBTC = 1e8 wBTC` (8 decimals) at `2,000.0 AVAX = 2,000 * 1e18 AVAX`. The `floorPrice` is `2000e18 * 1e18 / 1e8 = 2e31`

Assume the Launch event only raised `1,000.0 AVAX` - half of the floor price for the issued token amount of `1.0 WBTC` (it should therefore allocate only half a WBTC) - and the token amount will be reduced as: `floorPrice = 2e31 > 1000e18 * 1e18 / 1e8 = 1e31 = actualPrice`. Then, `tokenAllocated = 1000e18 * 1e8 / 2e31 = 1e29 / 2e31 = 0` and no tokens would be allocated, instead of `0.5 WBTC = 0.5e8 WBTC`.

The computation should be `tokenAllocated = wavaxReserve * 1e18 / floorPrice = 1000e18 * 1e18 / 2e31 = 1e39 / 2e31 = 10e38 / 2e31 = 5e7 = 0.5e8`.

Recommendation

The new `tokenAllocated` computation should be `tokenAllocated = wavaxReserve * 1e18 / floorPrice;`.

Response

> Floor price is in 1e18, so numerator is change to be multiplied by `10**token.decimals()`. Check PR#76

The floor price is supposed to be provided in 18 decimals without any token decimal information (`floorPrice = 2000 * 1e18` in the example above) and the `if` condition was fixed instead.

3.2 Launch event creation can be denied (high)

The `RocketJoeFactory.createRJLaunchEvent` requires that no previous launch event was already created for the `_token`.

```
function createRJLaunchEvent(
    address _issuer,
    uint256 _phaseOneStartTime,
    address _token,
    uint256 _tokenAmount,
    uint256 _tokenIncentivesPercent,
    uint256 _floorPrice,
    uint256 _maxWithdrawPenalty,
    uint256 _fixedWithdrawPenalty,
    uint256 _maxAllocation,
    uint256 _userTimelock,
    uint256 _issuerTimelock
) external override returns (address) {
    require(
        // @audit I can frontrun and grief if I own even a single token
            of this
        getRJLaunchEvent[_token] == address(0),
        "RJFactory: token has already been issued"
    );
    require(_token != address(0), "RJFactory: token can't be 0 address"
        );
    require(_token != wavax, "RJFactory: token can't be wavax");
    require(
```

```
        _tokenAmount > 0,
        "RJFactory: token amount needs to be greater than 0"
    );
    require(
        IJoeFactory(factory).getPair(wavax, _token) == address(0),
        "RJFactory: pair already exists"
    );

    // ...
}
```

A griefer who owns a single `_token` amount can call `createRJLaunchEvent` with undesirable parameters and deny a real launch event being created by the `_token` creators.

Recommendation

Consider allowing multiple launch events for the same token.

Response

> Acknowledged. Will control by ensuring launch partners do not release tokens into circulation prior to launch event.

3.3 Launch event creation can be denied 2 (high)

The `RocketJoeFactory.createRJLaunchEvent` requires that no previous pool was created for the `WAVAX <> _token` pair.

```
function createRJLaunchEvent(
    address _issuer,
    uint256 _phaseOneStartTime,
    address _token,
    uint256 _tokenAmount,
    uint256 _tokenIncentivesPercent,
    uint256 _floorPrice,
    uint256 _maxWithdrawPenalty,
    uint256 _fixedWithdrawPenalty,
    uint256 _maxAllocation,
    uint256 _userTimelock,
```

```
12      uint256 _issuerTimelock
13  ) external override returns (address) {
14      require(
15          getRJLaunchEvent[_token] == address(0),
16          "RJFactory: token has already been issued"
17      );
18      require(_token != address(0), "RJFactory: token can't be 0 address"
            );
19      require(_token != wavax, "RJFactory: token can't be wavax");
20      require(
21          _tokenAmount > 0,
22          "RJFactory: token amount needs to be greater than 0"
23      );
24      require(
25          // @audit I can frontrun and grief even if I don't own the
                token by creating this pair
26          IJoeFactory(factory).getPair(wavax, _token) == address(0),
27          "RJFactory: pair already exists"
28      );
29
30      // ...
31  }
```

A griefer who does not even have a `_token` balance can create a pool for the `WAVAX <> _token` pair by calling `JoeFactory.createPair(WAVAX, _token)`. This prevents the real `_token` creators from launching a launch event.

Recommendation

Consider allowing launch events even if the pool already exists. Special attention must be paid if the pool is already initialized with liquidity at a different price than the launch event price.

It would be enough to have a standard min. LP return "slippage" check (using parameter values for `amountAMin`/`amountBMin` instead of the hardcoded ones in `router.addLiquidity`) in `LaunchEvent.createPair()`. The function must then be callable with special privileges only, for example, by the issuer. Alternatively, the slippage check can be hardcoded as a percentage of the raised amounts (`amountADesired = 0.95 * wavaxReserve`, `amountBDesired = 0.95 * tokenAllocated`).

This will prevent attacks that try to provide LP at a bad pool price as the transaction will revert when receiving less than the slippage parameter. If the pool is already initialized, it should just get arbitraged to the auction token price and liquidity can then be provided at the expected rate again.

Response

> There are various ways this can be griefed. E.g. attacker can also create pair, send 1 WAVAX, call `pair.sync()` to update reserves. We address this by replacing the call to `router.addLiquidity()` : PR#82

This has been mitigated by reverting if there is a pool with existing liquidity. The team will ensure that "launch partners do not release tokens into circulation prior to launch event", preventing third parties from providing liquidity.

We'd still like to mention if this assumption is ever to be removed, the current fix introduces a new issue as it directly calls `pool.mint` now with the raised token amounts, not taking into account the actual reserve ratio of a pool that might already exist. It assumes that it is the first to provide liquidity but this does not have to be the case.

An attacker that has a tiny amount of `token`s can steal large quantities of the tokens used for providing liquidity.

The reason is that in Uniswap V2 / Sushiswap / TraderJoe you receive liquidity tokens that correspond to the worst (minimum) ratio that was provided:

```
1 // https://snowtrace.io/address/0
      x9ad6c38be94206ca50bb0d90783181662f0cfa10#contracts
2 // L485: JoePair.mint
3 // amount0, amount1 are the tokens provided for LPing
4 liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul
      (_totalSupply) / _reserve1);
5 // writing it in math terms and factoring out _totalSupply to make it
      easier to read
6 // liquidityToMint = _totalSupply * Math.min(amount0 / _reserve0,
      amount1 / _reserve1)
```

POC

Assume there's a launch event with tokens `WAVAX` and token `B`. For simplicity, assume that the auction has decided that they are worth equal amounts, i.e., `tokenPrice = 1.0`, `1.0 A = 1.0 B`. (The attack works with any `tokenPrice` as the pool's current reserve ratio can be chosen by the attacker and is independent of any of these `LaunchEvent` values.)

Let's say the `LaunchEvent.createPair` will provide `10.0`WAVAX and `10.0`B to the pool.

- Attacker acquires a tiny amount of WAVAX and a large amount of B.
- Attacker observes the `LaunchEvent.createPair` transaction in the mempool

- They frontrun it and create the Traderjoe WAVAX <> B pool and provide initial liquidity of `0.0001 WAVAX <> 10.0B`. They receive the initial LP token supply.
- `LaunchEvent.createPair` provides `10.0 WAVAX <> 10.0B` liquidity which is very different from the current reserves ratio and leads to receiving a horrible rate. They will receive `liquidity = _totalSupply * Math.min(amount0 / _reserve0, amount1 / _reserve1)= _totalSupply * Math.min(10.0A/0.0001A=100_000, 10.0B/10.0B=1)= 1*_totalSupply`.
- Attacker and `LaunchEvent` now each own 50% of the total LP supply.
- Attacker redeems all their LP tokens and receives their fair share (50%) of the pool reserves `5.00005WAVAX <> 10.0B`.

The attacker's profit is ~`5.0WAVAX`. (By further increasing the initial B liquidity, the attacker can steal almost the entire `WAVAX` amount that was provided by `LaunchEvent`.)

> Note that this attack is symmetric in the tokens, i.e., if the attacker would rather steal the `LaunchEvent` token, they can provide liquidity with a large number of WAVAX tokens and a small number of LaunchEvent tokens.

3.4 Pair creation can be denied (high)

The `LaunchEvent.createPair` requires that no previous pool was created for the `WAVAX <> _token` pair.

```
function createPair() external isStopped(false) atPhase(Phase.
    PhaseThree) {
     (address wavaxAddress, address tokenAddress) = (
         address(WAVAX),
         address(token)
     );
     // @audit grief: anyone can create pair
     require(
         factory.getPair(wavaxAddress, tokenAddress) == address(0),
         "LaunchEvent: pair already created"
     );

     // ...
}
```

A griefer can create a pool for the `WAVAX <> _token` pair by calling `JoeFactory.createPair(WAVAX, _token)` while the launch event phase 1 or 2 is running. No liquidity can then be provided and an emergency state must be triggered for users and the issuer to be able to withdraw again.

Recommendation

It must be assumed that the pool is already created and even initialized as pool creation and liquidity provisioning is permissionless. Special attention must be paid if the pool is already initialized with liquidity at a different price than the launch event price.

It would be enough to have a standard min. LP return "slippage" check (using parameter values for `amountAMin`/`amountBMin` instead of the hardcoded ones in `router.addLiquidity`) in `LaunchEvent.createPair()`. The function must then be callable with special privileges only, for example, by the issuer. Alternatively, the slippage check can be hardcoded as a percentage of the raised amounts (`amountADesired = 0.95 * wavaxReserve`, `amountBDesired = 0.95 * tokenAllocated`).

This will prevent attacks that try to provide LP at a bad pool price as the transaction will revert when receiving less than the slippage parameter. If the pool is already initialized, it should just get arbitraged to the auction token price and liquidity can then be provided at the expected rate again.

Response

> Same response as 3.4

3.5 ERC20 return values not checked (`medium`)

The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do **not** revert if the transfer failed but return `false` instead. Tokens that don't actually perform the transfer and return `false` are still counted as a correct transfer.

Recommendation

As the Launch event token can be any token, all interactions with it should follow correct EIP20 checks. We recommend checking the `success` boolean of all `.transfer` and `.transferFrom` calls for the unknown `token` contract.

- `LaunchEvent.withdrawLiquidity`: `token.transfer(msg.sender, amount);`
- `LaunchEvent.withdrawIncentives`: `token.transfer(msg.sender, amount);`
- `LaunchEvent.emergencyWithdraw`: `token.transfer(msg.sender, amount);`
- `LaunchEvent.skim`: `token.transfer(msg.sender, amount);`
- `RocketJoeFactory.createRJLaunchEvent`: `IERC20(_token).transferFrom(msg.sender, launchEvent, _tokenAmount);`

Response

> Instances of `.transfer()` replaced with `.safeTransfer()`.

3.6 Users can lose value in emergency state (medium)

Imagine the following sequence of events:

- `LaunchEvent.createPair()` is called which sets `wavaxReserve = 0`, adds liquidity to the pair and receives `lpSupply` LP tokens.
- `LaunchEvent.allowEmergencyWithdraw()` is called which enters emergency / paused mode and disallows normal withdrawals.
- Users can only call `LaunchEvent.emergencyWithdraw` which reverts as the WAVAX reserve was already used to provide liquidity and cannot be paid out. Users don't receive their LP tokens either. The users lost their entire deposit in this case.

Recommendation

Consider paying out LP tokens in `emergencyWithdraw`.

Response

> `emergencyWithdraw()` changed to also allow withdrawal of LP: PR#99

3.7 `rJoeAmount` can never be less than the `_avaxAmount` (medium)

The `LaunchEvent.rJoePerAvax` variable is an *unscaled* integer value and used to compute the `rJoeAmount` as:

```
1 function getRJoeAmount(uint256 _avaxAmount) public view returns (
    uint256) {
2     return _avaxAmount * rJoePerAvax;
3 }
```

This means the required `rJoeAmount` to burn can never be less than the deposited `avaxAmount`. If a launch event desires to use `0.5 rJoe` per AVAX, this is not possible.

Recommendation

Consider the `rJoePerAvax` value as a value scaled by `1e18` and then divide by this scale in `getRJoeAmount` again.

Response

> `rJoePerAvax` is now scaled to 1e18: PR#101

3.8 Uninitialized `RocketJoeStaking.lastRewardTimestamp` can inflate `rJoe` supply (`medium`)

The `RocketJoeStaking.lastRewardTimestamp` is initialized to zero. Usually, this does not matter as `updatePool` is called before the first deposit and when `joeSupply = joe.balanceOf(address(this))==0`, it is set to the current time.

```
function updatePool() public {
    if (block.timestamp <= lastRewardTimestamp) {
        return;
    }
    uint256 joeSupply = joe.balanceOf(address(this));

    // @audit lastRewardTimestamp is not initialized. can send 1 Joe to
        this contract directly => lots of rJoe minted to this contract
    if (joeSupply == 0) {
        lastRewardTimestamp = block.timestamp;
        return;
    }
    uint256 multiplier = block.timestamp - lastRewardTimestamp;
    uint256 rJoeReward = multiplier * rJoePerSec;
    accRJoePerShare =
        accRJoePerShare +
        (rJoeReward * PRECISION) /
        joeSupply;
    lastRewardTimestamp = block.timestamp;

    rJoe.mint(address(this), rJoeReward);
}
```

However, if a user first directly transfers `Joe` tokens to the contract before the first `updatePool` call, the `block.timestamp - lastRewardTimestamp = block.timestamp` will be a large timestamp value and lots of `rJoe` will be minted (but not distributed to users). Even though they are not distributed to the users, inflating the `rJoe` total supply might not be desired.

Recommendation

Consider tracking the actual total deposits in a storage variable and using this value instead of the current balance for `joeSupply`. This way, transferring tokens to the contract has no influence and depositing through `deposit` first calls `updatePool` and initializes `lastRewardTimestamp`.

Response

> `lastRewardTimestamp` is initialized in `initialize()` : PR#76

3.9 `IssuingTokenDeposited` event not emitted (`low`)

The `IssuingTokenDeposited` event in `LaunchEvent` is not used. Unused code can hint at programming or architectural errors.

Recommendation

Use it or remove it.

Response

> `IssueTokenDeposited` event is moved from `LaunchEvent` to `RocketJoeFactory` and emitted in `createRJLaunchEvent()`: PR#85

3.10 `UserWithdrawn` event not emitted (`low`)

The `UserWithdrawn` event in `LaunchEvent` is not used. Unused code can hint at programming or architectural errors.

Recommendation

Use it or remove it.

Response

> `UserWithdrawn` event is emitted in `LaunchEvent.withdrawAVAX()`: PR#83

3.11 Penalty Collector must be trusted (low)

The `rocketJoeFactory.penaltyCollector()` receives the penalty when users withdraw from the launch event contract. Control is given to this contract in the `LaunchEvent._safeTransferAVAX(rocketJoeFactory.penaltyCollector(), feeAmount)` call and a malicious penalty collector could throw an error and deny all user withdrawals.

Recommendation

Ensure the penalty collector is a trusted smart contract or an EOA.

Response

> Acknowledged. It will be set to our treasury multisig

3.12 Miscellaneous (minor)

- `LaunchEvent.tokenIncentivesPercent`: The math in the comment is wrong: `/// then 105 000 * 1e18 / (1e18 + 5e16)= 5 000 tokens are used for incentives`. It should be `105 000 * 5e16 / (1e18 + 5e16)= 5 000 tokens are used for incentives`
- `RocketJoeFactory.createRJLaunchEvent:`: The naming and comments for `_tokenAmount` and `_tokenIncentivesPercent` could be misinterpreted: The `_tokenAmount` is the *total amount* that will be issued *including* the fees. Consider renaming `_tokenAmount` to `_tokenAmountIncludingFees`. The `_tokenIncentivesPercent` are *not* "a percentage of the issuing tokens", they can be more than 100% as the incentives are computed as `_tokenIncentivesPercent / (1.0 + _tokenIncentivesPercent)* _tokenAmountIncludingFees`. Therefore, if 75% of the tokens are desired for incentives, one must *not* use `_tokenIncentivesPercent = 0.7e18` but `_tokenIncentivesPercent = 300% = 3e18`.

- `LaunchEvent.getReserves`: The comment says: `@notice Returns the current balance of the pool`. The "of the pool" part can be misleading as the `tokenIncentivesBalance` are never part of the *pool pair*. Consider changing this to "Returns the outstanding balance of the launch event contract".
- `RocketJoeStaking.withdraw`: The `_safeRJoeTransfer(msg.sender, pending)` only needs to be performed if `pending > 0`.
- `LaunchEvent.withdrawAVAX`: The `_safeTransferAVAX(msg.sender, amountMinusFee)` call gives control to the caller and it's best if this is the last call being made to avoid potential re-entrancy attacks - even though no immediate issues could be found with this re-entrancy. (They can call `skim` to steal the penalty collector's fee amount but then the `_safeTransferAVAX(rocketJoeFactory.penaltyCollector(), feeAmount)` would fail and the transaction would revert.)

Response

> Recommendations implemented in PR#103

4 Conclusion

An issue involving tokens with non-standard decimals has been identified that leads to an unexpected loss of funds for the users. Several “griefing” attacks have been found where malicious parties can interfere with the correct progression of launch events and deny the successful issuance of LP tokens. Overall, the documentation and the codebase were found to be of high quality. We recommend adding further tests for tokens with non-18 decimals and launch events where pools have already been initialized at a different price.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.