ENIGMA DARK

Securing the Shadows

Security Review

Flaunch

November 15, 2024

Contents

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Flaunch
The ƒlaunch protocol is a launchpad platform built on Uniswap v4, incorporating advanced mechanics for token launch and trading. It emphasizes sustainability in token economies by introducing features such as Progressive Bid Walls and decentralized revenue-sharing models. These mechanisms aim to create a fair and transparent environment for participants, balancing incentives for developers and traders to prioritize long-term ecosystem stability over short-term speculation.

Engagement Overview

Over the course of 1,6 weeks (1 week and 3 days) starting October 28th 2024, the Enigma Dark team conducted a security review of the Flaunch project. The review was performed by two Lead Security Researchers, vnmrtz & 0xWeiss.

The following repositories were reviewed at the specified commits:

Repository	Commit
flayerlabs/flaunch-contracts	`a994398e8cfb39fcc70bdb673a5be08f971fbbca`

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	2	2	0
Medium	1	1	0
Low	4	4	0
Informational	4	3	1

Findings

Index	Issue Title	Status
H-01	Wrong cached `_beforeSwapTick` value is used by liquidity hooks at feeDistribution, setting bidWall liquidity at wrong ticks	Fixed
H-02	Creator can sandwitch fees during the fair launch period	Fixed
M-01	`canFlaunchNow` does not return the correct value	Fixed
L-01	Malicious action can purposely leave approvals open to steal future tokens.	Fixed
L-02	Excess of flaunch fee allows to brick UI by bypassing pool state updates	Fixed
L-03	Inconsistent Declaration of Position Manager Across the Codebase	Fixed
L-04	Do not hardcode values nor addresses	Fixed
I-01	Consider reverting early on BidWall `disable`	Fixed
I-02	Miscellaneous	Fixed
I-03	BidWall `deposit` liquidity removal flow can be simplified	Fixed
I-04	`_amountSpecified` inside `fillFromPosition` can't be 0	Acknowledged

Detailed Findings

High Risk

H-01 - Wrong cached `_beforeSwapTick` value is used by liquidity hooks at feeDistribution, setting bidWall liquidity at wrong ticks

Severity: High Risk

Technical Details: Three of the protocol hooks call `_distributeFees`: `afterSwap`, `beforeAddLiquidity` & `beforeRemoveLiquidity`. This internal function is able to call bidWalls `deposit` passing `_beforeSwapTick` as parameter. This behaviour works as intended for the swap flows, since the variable is cached on the respective before swap hook. Therefore at the moment of bidWall deposits, the variable is correctly updated to the tick value of the pool just before the swap is executed.

However this update on the before side of hooks does not happen for `beforeRemoveLiquidity` nor `beforeAddLiquidity`. Both functions call `_distributeFees`, which may imply a deposit on the bidWall, if the cached value comes from a different pool that the one that is being interacted a wrong tick will be use for the bidWall liquidity provision.

Impact: BidWalls are placed at wrong ticks on liquidity provision, which could lead to reverts, implying DOS for adding or removing liquidity.

Recommendation: Instead of using a global variable, implement a mapping that stores tick values on a pool basis.

Developer Response: Fixed at commit `57b63a2`. Removed modify liquidity fee distributions.

H-02 - Creator can sandwitch fees during the fair launch period

Severity: High Risk

Technical Details:

The creator of the memecoin can change the creator fee freely in the range of 0%-80% while the fairlaunch period is still active, effectively being able to front-run big swaps to earn more fees and "trick" users to "ape" on the flaunch period by setting a very low creator fee at the start.

Flaunch is intended to be used by users mostly through the front-end. Users will make their on "investment decisions" using several parameters, like the fess for the creator vs the fees for the community.

The flaunch period lasts 10 minutes, where the users will quickly "ape" to buy the token.

The impact is not just the creator "sandwiching" the swaps to get more fees, it also will provoke that users that are investing inside this 10minute period because the creator only has a 1% fee, do regret the investment as the creator can update it to 80% inside the flaunch period.

Impact:

Creator can sandwich swaps to earn more fees and trick users to buy their memecoin by setting a very low fee range at the start.

Recommendation:

Do not allow the creator to change the creator fee inside the flaunch period.

Developer Response: Fixed at commit `525d673`.

Medium Risk

M-01 - `canFlaunchNow` does not return the correct value

Severity: Medium Risk

Technical Details: As per the natspec comment `canFlaunchNow` should return true if the current `block.timestamp` is bigger than `scheduleParams.flaunchTime`and has NOT been launched yet. However the check is missing a `!` which renders relayers not able to check the status of a scheduled flaunch properly.

```
function canFlaunchNow(uint _scheduleId) external view returns (bool) {
    return (block.timestamp >= scheduleParams[_scheduleId].flaunchTime &&
        scheduleParams[_scheduleId].flaunched); //@audit-issue Should check
for not flaunched
}
```

Impact: Relayers cannot check for a flaunch to be ready to be executed.

Recommendation: Fix the boolean clause:

```
function canFlaunchNow(uint _scheduleId) external view returns (bool) {
    return (block.timestamp >= scheduleParams[_scheduleId].flaunchTime &&
        !scheduleParams[_scheduleId].flaunched);
}
```

Developer Response: Fixed. This function has been removed and logic has been moved into the `PositionManager`. We now store `flaunchesAt` which can be checked against the current timestamp.

Low Risk

L-01 - Malicious action can purposely leave approvals open to steal future tokens.

Severity: Low Risk

Technical Details:

The creator of each memecoin can leverage the `executeAction` function inside the memecoin treasury contract to execute a set of actions.

```
 function executeAction(address _action, bytes memory _data) public {

        // Ensure the action is approved
        if (!actionManager.approvedActions(_action)) revert
ActionNotApproved();

        // Make sure the caller is the owner of the corresponding ERC721
        address poolCreator = poolKey.memecoin(nativeToken).creator();
        if (poolCreator != msg.sender) revert Unauthorized();

        // Approve all tokens to be used before execution
        IERC20(Currency.unwrap(poolKey.currency0)).approve(_action,
type(uint).max);
        IERC20(Currency.unwrap(poolKey.currency1)).approve(_action,
type(uint).max);

        // Call the execute function on the action contract
        ITreasuryAction(_action).execute(poolKey, _data);
        emit ActionExecuted(_action, poolKey, _data);

        // Unapprove all tokens after execution
        IERC20(Currency.unwrap(poolKey.currency0)).approve(_action, 0);
        IERC20(Currency.unwrap(poolKey.currency1)).approve(_action, 0);
    }
```

The important point is that the contract approves max token0 and token1 to the action address:

`IERC20(Currency.unwrap(poolKey.currency0)).approve(_action, type(uint).max);`

to further make a callback and restore the approval

```
// Call the execute function on the action contract
      ITreasuryAction(_action).execute(poolKey, _data);
      emit ActionExecuted(_action, poolKey, _data);

      // Unapprove all tokens after execution
      IERC20(Currency.unwrap(poolKey.currency0)).approve(_action, 0);
      IERC20(Currency.unwrap(poolKey.currency1)).approve(_action, 0);
```

This callback is dangerous and could potentially be leveraged by an action which is also a creator of a memecoin.

It could re-enter in `executeAction()` with a different action address than the first time, which would not reset the initial approval from the first action.

In case such action gets unapproved, they will still have an open approval.

Impact:

Approvals can be left uncleared to be later exploited.

Recommendation:

Do add a non-reentrant modifier to the `executeAction` function.

Developer Response: Fixed at commit `ad80cc7`.

L-02 - Excess of flaunch fee allows to brick UI by bypassing pool state updates

Severity: Low Risk

Technical Details:

Memecoin creators must pay a flaunch fee when launching a memecoin. This flaunch fee is directly fetched from the `initialPrice` contract.

In the case a creator sends more `msg.value` than the required, a reimbursement callback will be triggered.

```
       uint flaunchFee = initialPrice.getFlaunchingFee(msg.sender);
       if (flaunchFee != 0) {
            // Check if we have insufficient value provided
            if (msg.value < flaunchFee) {
                revert InsufficientFlaunchFee(msg.value, flaunchFee);
            }

            // Pay the flaunching fee to our fee recipient
            SafeTransferLib.safeTransferETH(protocolFeeRecipient,
flaunchFee);

            // Refund any ETH that was not required
            if (msg.value > flaunchFee) {
                SafeTransferLib.safeTransferETH(msg.sender, msg.value -
flaunchFee);
            }
        }

        emit PoolCreated({
            _poolId: poolId,
            _initialTokenFairLaunch: _initialTokenFairLaunch,
            _fairLaunchEnds: fairLaunchInfo.endsAt,
            _memecoin: memecoin_,
            _memecoinTreasury: memecoinTreasury,
            _tokenId: tokenId,
            _currencyFlipped: currencyFlipped,
            _flaunchFee: flaunchFee,
            _creator: msg.sender
        });

        // After our contract is initialized, we mark our pool as
initialized and emit
        // our first state update to notify the UX of current prices, etc.
        _emitPoolStateUpdate(poolId);
```

This callback can be abused by the creator to re-enter the flaunch function and bypass the event emission that is used to update the UI:

```
        // After our contract is initialized, we mark our pool as
initialized and emit
        // our first state update to notify the UX of current prices, etc.
        _emitPoolStateUpdate(poolId);
```

Impact:

UI can be bricked

Recommendation:

Do make the callback at the end of the function:

```
        uint flaunchFee = initialPrice.getFlaunchingFee(msg.sender);
        if (flaunchFee != 0) {
            // Check if we have insufficient value provided
            if (msg.value < flaunchFee) {
                revert InsufficientFlaunchFee(msg.value, flaunchFee);
            }

            // Pay the flaunching fee to our fee recipient
            SafeTransferLib.safeTransferETH(protocolFeeRecipient,
flaunchFee);

-            // Refund any ETH that was not required
-            if (msg.value > flaunchFee) {
-                SafeTransferLib.safeTransferETH(msg.sender, msg.value -
flaunchFee);
-            }
        }

        emit PoolCreated({
            _poolId: poolId,
            _initialTokenFairLaunch: _initialTokenFairLaunch,
            _fairLaunchEnds: fairLaunchInfo.endsAt,
            _memecoin: memecoin_,
            _memecoinTreasury: memecoinTreasury,
            _tokenId: tokenId,
            _currencyFlipped: currencyFlipped,
            _flaunchFee: flaunchFee,
            _creator: msg.sender
        });

        // After our contract is initialized, we mark our pool as
initialized and emit
        // our first state update to notify the UX of current prices, etc.
        _emitPoolStateUpdate(poolId);

+            // Refund any ETH that was not required
+            if (msg.value > flaunchFee) {
+                SafeTransferLib.safeTransferETH(msg.sender, msg.value -
flaunchFee);
+            }
```

Developer Response: Fixed at commit `3bd90c3`. The ETH logic has been moved below other logic as advised.

L-03 - Inconsistent Declaration of Position Manager Across the Codebase

Severity: Low Risk

Technical Details:

The `positionManager` contract is meant to be an immutable contract that should not be updated.

In some contracts like `DynamicFeeCalculator`, this principle is followed, and `positionManager` is declared as immutable:

`address public immutable positionManager;`

In other contracts, this rule is not followed, and it is declared as a mutable variable that can be updated later:

`PositionManager public positionManager;`

Impact:

Missing functionality to update the `positionManager`

Recommendation:

Do follow a structure when setting immutable/mutable variables across the codebase. If position manager is meant to be updated, then add that functionality within the entire codebase, if not, do the opposite.

Developer Response: Fixed at commit `525d673`.

L-04 - Do not hardcode values nor addresses

Severity: Low Risk

Technical Details:

- In the `createPosition` function, it does check that the initial supply iw within the bounds of 0.1e27 < _initialTokenFairLaunch < 0.69e27

```
  if (_initialTokenFairLaunch < 0.1e27 || _initialTokenFairLaunch > 0.69e27)
{
            revert InvalidInitialSupply(_initialTokenFairLaunch);
        }
```

This values, while purposely set, should not be hardcoded as they might want to be updated in the future.

- Vitalik's address is currently hardcoded to:

`address internal constant VITALIK_ETH = 0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045;`

while unlickely, it could happen that vitalik would lose access to this address, making the tokens to be stuck in that address.

Do have the `VITALIK_ETH` address as a variable instead of hardcoding it.

Impact:

In case a variable needs to be changed or because of external reasons Vitalik can't access his wallet, the contract state can be incorrect.

Recommendation:

Make the recommended changes above.

Developer Response: Partially fixed at commit `5372c91`.

Informational

I-01 - Consider reverting early on BidWall `disable`

Severity: Informational

Technical Details: BidWall `disable` does not revert early when access control does not succeed and a BidWall is disabled. Consider changing the order of the checks to follow the best practice of having early revert for access control checks.

Recommendation: Change the order of the checks to the following:

```
// Ensure that the caller is the pool creator
if (msg.sender != _key.memecoin(nativeToken).creator()) revert
CallerIsNotCreator();/

// We only need to process the following logic if anything is changing
PoolInfo storage _poolInfo = poolInfo[_key.toId()];
if (_disable == _poolInfo.disabled) return;
```

Developer Response: Fixed at commit `48f2447`.

I-02 - Miscellaneous

Severity: Informational

Technical Details: The following issue contains a compilation of typos, wrong comments and bad function naming

- `BidWall` line `237`: function `disable` could be changed to `changeDisabledStatus` since it not only allows to disable BidWalls but also enable them back.
- `FeeDistributor` line `223`: Comment type, `/// less that then traditionally` should be `/// less than the traditionally`.
- `MarketCappedPrice` line `20`: Wrong contract summary, it refers to the `InitialPrice` summary.

Developer Response: Fixed at commit `8aa3a0d`.

I-03 - BidWall `deposit` liquidity removal flow can be simplified

Severity: Informational

Technical Details: When a deposit lands on a bidWall and this one has already been initialised, liquidity is temporarily removed to later be deposited at a fresh tick.

The implementation can be simplified since `tickLower` and `tickUpper` are already known from the previous deposit so there is no need to calculate them again like the call to remove liquidity does:

```
// Find the desired BidWall tick based on if we have flipped key tokens
int24 tick = _nativeIsZero ? _poolInfo.tickUpper :
_poolInfo.tickLower;//@audit todo check this path

// We need to remove tokens from our current position
(ethWithdrawn, memecoinWithdrawn) = _removeLiquidity({
    _key: _poolKey,
    _nativeIsZero: _nativeIsZero,
    _tickLower: _nativeIsZero ? tick - TickFinder.TICK_SPACING :
tick,//@audit-issue INFO not needed since tickLower and tickUpper are
already known
    _tickUpper: _nativeIsZero ? tick : tick + TickFinder.TICK_SPACING
});
```

For reference here are other blocks of code on the bidWall that directly use stored tick values:

```
// Remove all liquidity from the BidWall
(ethWithdrawn, memecoinWithdrawn) = _removeLiquidity({
    _key: _key,
    _nativeIsZero: nativeIsZero,
    _tickLower: _poolInfo.tickLower,
    _tickUpper: _poolInfo.tickUpper
});
```

Recommendation: Implement the call reusing the stored position tick values.

Developer Response: Fixed at commit `3c0ebe`.

I-04 - `_amountSpecified` inside `fillFromPosition` can't be 0

Severity: Informational

Technical Details:

Inside the `fillFromPosition` function there is a check whether the amount specified to be swapped is 0:

```
if (_amountSpecified == 0) {
   return (beforeSwapDelta_, balanceDelta_, info);
       }
```

This can't happen, thus the code is useless, because Uniswap already checks for this amount not to be 0 in their pool manager contract on line 189:

```
if (params.amountSpecified == 0) SwapAmountCannotBeZero.selector.revertWith();
```

Impact:

Redundant code

Recommendation:

Do remove the previously mentioned line of code.

Developer Response: Acknowledged. Although we agree with this as a code redundancy in the lifetime of the transaction, this would allow the code to exit early and save some gas on reverts.

Since this is just informational I think we will leave this in

Disclaimer

This report does not endorse or critique any specific project or team. It does not assess the economic value or viability of any product or asset developed by parties engaging Enigma Dark for security assessments. We do not provide warranties regarding the bug-free nature of analyzed technology or make judgments on its business model, proprietors, or legal compliance.

This report is not intended for investment decisions or project participation guidance. Enigma Dark aims to improve code quality and mitigate risks associated with blockchain technology and cryptographic tokens through rigorous assessments.

Blockchain technology and cryptographic assets inherently involve significant risks. Each entity is responsible for conducting their own due diligence and maintaining security measures. Our assessments aim to reduce vulnerabilities but do not guarantee the security or functionality of the technologies analyzed.

This security engagement does not guarantee against a hack. It is a review of the codebase at a during a specific period of time. Enigma Dark makes no warranties regarding the security of the code and does not warrant that the code is free from defects. By deploying or using the code, the project and users of the contracts agree to use the code at their own risk. Any modifications to the code will require a new security review.