Pashov Audit Group

WishWish Security Review

November 4th 2025 - November 6th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About WishWish

WishWish is an NFT platform that enables creators to launch NFT collections through a dual-contract system of boxes and ccollectables, allowing users to mint boxes, reveal them into collectables, and generate royalties for creators. It integrates the WISH ERC20 token for payments, fee distribution, and redeemable creator rewards, with a factory-and-manager architecture handling contract deployment and minting permissions.

5. Executive Summary

A time-boxed security review of the **Sofamon/wishwish-contracts** repository was done by Pashov Audit Group, during which **0x37**, **JCN**, **0xbepresent**, **Hunter** engaged to review **WishWish**. A total of **12** issues were uncovered.

Protocol Summary	
Project Name	WishWish
Protocol Type	NFT factory
Timeline	November 4th 2025 - November 6th 2025

Review commit hash:

- ce7b00f10fb26e207a785331ffaa1d13f44db766 (Sofamon/wishwish-contracts)

Fixes review commit hash:

- 58b0c86fc7ea13795b05ed9c08ab2e558af61a52 (Sofamon/wishwish-contracts)

Scope

`RoyaltyManager.sol` `SwapImpl.sol` `SwapProxy.solWishWishBox.sol` `WishWishCollectable.sol` `WishWishFactory.sol` `WishWishManager.sol` `WishWishManagerProxy.sol` `WishWishToken.sol` `WishWishTokenProxy.sol` `IWETH9.sol`

6. Findings

Findings count

Severity	Amount
Critical	1
Medium	1
Low	10
Total findings	**12**

Summary of findings

ID	Title	Severity	Status
[C-01]	Anyone can re-initialize the `swapProxy`	Critical	**Resolved**
[M-01]	Stale `swapProxy` reference in `RoyaltyManager` contracts	Medium	**Resolved**
[L-01]	Suggeset verifying L2 sequencer's status	Low	**Resolved**
[L-02]	Deprecated `answeredInRound` is used in `SwapImpl`	Low	**Resolved**
[L-03]	`SwapImpl` lacks upgradeability despite proxy pattern indicating upgradeable design	Low	**Resolved**
[L-04]	Missing storage gaps in upgradeable contracts	Low	**Resolved**
[L-05]	Manager rotation breaks legacy collection functionality	Low	**Resolved**
[L-06]	Individual token royalty functionality no longer supported	Low	**Resolved**
[L-07]	Missing stale feed check for USDC	Low	Acknowledged
[L-08]	`protocolCut` not actually burned during `_mintBox`	Low	**Resolved**
[L-09]	Using `block.timestamp` as swap deadline removes MEV protection	Low	Acknowledged
[L-10]	permissionless `distribute()` allows self sandwiching for profit	Low	**Resolved**

Critical findings

[C-01] Anyone can re-initialize the `swapProxy`

Severity

Impact: High

Likelihood: High

Description

The `SwapImpl::initialize` function lacks access control and can be successfully invoked more than once, allowing arbitrary reinitialization of the proxy. This enables an attacker to update critical state variables, such as the `router` and `permit2` addresses, with custom malicious contracts. Doing so can allow an attacker to steal all `ETH` and `WETH` that gets sent to the `RoyaltyManager`.

After the `SwapImpl` and `SwapProxy` are deployed, an attacker can call `SwapProxy::initialize` again and specify a custom contract as the `router` and `permit2`:

```
    function initialize(
        address _universalRouter,
        address _permit2,
        address _weth,
        address _usdc,
        address _feedETH,
        address _feedUSDC,
        uint24 _feeTier,
        uint16 _slippageBps,
        uint32 _maxAgeSec
    ) external {
    ...
        $.router = IUniversalRouter(_universalRouter);
        $.permit2 = _permit2;
        $.WETH = IWETH9(_weth);
        $.USDC = IERC20(_usdc);
        $.FEED_ETH = AggregatorV3Interface(_feedETH);
        $.FEED_USDC = AggregatorV3Interface(_feedUSDC);
        $.FEE_TIER = _feeTier;
        $.SLIPPAGE_BPS = _slippageBps;
        $.MAX_AGE_SEC = _maxAgeSec;
...
        $.WETH.safeApprove(_permit2, type(uint256).max);
        IPermit2(_permit2).approve(address($.WETH), _universalRouter, type(uint160).max,
type(uint48).max);
    }
```

As we can see above, after this reinitialization, the `SwapProxy` contract will approve the attacker's malicious contract to handle any of its `WETH` tokens.

After royalties (which can be denominated in `ETH`, `WETH`, or `USDC`) are sent to the `RoyaltyManager`, any entity can call `RoyaltyManager::distribute` to swap any available `ETH` / `WETH` into `USDC`:

```
    function distribute() external nonReentrant {
        // 1. Swap ETH to USDC if any ETH balance
        uint256 ethBalance = address(this).balance;
        if (ethBalance > 0) {
            swapProxy.swapETHToUSDC{ value: ethBalance }();
        }

        // 2. Swap WETH to USDC if any WETH balance
        uint256 wethBalance = WETH.balanceOf(address(this));
        if (wethBalance > 0) {
            swapProxy.swapWETHToUSDC(wethBalance);
        }

        // 3. Distribute USDC if any balance
        uint256 usdcBalance = USDC.balanceOf(address(this));
        if (usdcBalance >= 2e6) {
            USDC.approve(address(manager), usdcBalance);
            manager.distributeRoyalty(usdcBalance);
        }
    }
```

The `RoyaltyManager` will then transfer its entire `ETH` balance to the `SwapProxy` via the `swapETHToUSDC` function call, and this function will in turn transfer all of the `ETH` to attacker's malicious contract via `router::execute`:

```
    function swapETHToUSDC() external payable nonReentrant returns (uint256 amountOut) {
        SwapImplStateStorage storage $ = _getStorage();

        uint256 amountIn = msg.value;
        if (amountIn == 0) revert AmountZero();
...
        $.router.execute{ value: amountIn }(commands, inputs, block.timestamp);
...
    }
```

Similarly, the `RoyaltyManager` will invoke the `swapWETHToUSDC` function, which will lead to the `SwapProxy` pulling the manager's entire `WETH` balance, and then the function will call into the attacker's malicious `router` contract:

```
    function swapWETHToUSDC(uint256 amountIn) external nonReentrant returns (uint256 amountOut)
{
        SwapImplStateStorage storage $ = _getStorage();

        if (amountIn == 0) revert AmountZero();

        $.WETH.safeTransferFrom(msg.sender, address(this), amountIn);
...
```

```
        $.router.execute(commands, inputs, block.timestamp);
...
    }
```

Since the reinitialization resulted in the `SwapProxy` approving the malicious contract to handle all of its `WETH` tokens, the contract can now simply pull all of the `WETH` tokens from the `SwapProxy`.

Recommendations

Prevent the `initialize` function from being called more than once. This can be accomplished by following the same pattern seen in other protocol contracts, such as `WishWishToken`:

```
    function init(address _usdc, address _usdcReceiver, address _redemptionSigner, address
_owner) external {
        if (owner() != address(0)) revert AlreadyInitialized();
    ...
        _initializeOwner(_owner);
    }
```

Similar to the code above, the `SwapImpl::initialize` function should revert if the owner has already been initialized and initialize the owner at the end of the function call.

Medium findings

[M-01] Stale `swapProxy` reference in `RoyaltyManager` contracts

Severity

Impact: Medium

Likelihood: Medium

Description

The `WishWishManager` contract allows the owner to update the `swapProxy` address via the `setSwapProxy` function. However, `RoyaltyManager` contracts are deployed once per creator in the `getRoyaltyManager` function and store the `swapProxy` as an immutable variable in their constructor. Once a creator launches their first collection, subsequent collections reuse the same `RoyaltyManager` instance, which retains the old proxy address even after updates. Consider the next scenario:

1. **CreatorA launches first collection**: Calls `launchNewCollection`, which invokes `getRoyaltyManager(msg.sender)` (code line 256). Since no `RoyaltyManager` exists for CreatorA, a new one is deployed with the current `swapProxy` (e.g., `0x123`) (code line 406). This instance is stored in `royaltyMap[CreatorA]`.

```
File: WishWishManager.sol
256:        address royaltyManager = getRoyaltyManager(msg.sender);
...
405:        if ($.royaltyMap[creator] == address(0)) {
406:            RoyaltyManager rm = new RoyaltyManager(address(this), $.swapProxy);
```

1. **Admin updates swap proxy**: Owner calls `WishWishManager.setSwapProxy(0x456)` to update to a new proxy implementation (e.g., due to bug fixes or optimizations).

2. **CreatorA launches second collection**: Calls `launchNewCollection` again, which calls `getRoyaltyManager(msg.sender)`. Since `royaltyMap[CreatorA]` already contains an address, no new `RoyaltyManager` is deployed - the existing one (with old proxy `0x123`) is reused.

```
File: WishWishManager.sol
403:    function getRoyaltyManager(address creator) public returns (address) {
404:        WishWishManagerStateStorage storage $ = _getStorage();
405:        if ($.royaltyMap[creator] == address(0)) {
            ...
```

```
        }
        return $.royaltyMap[creator];
    }
```

1. **Royalty distribution fails**: When `distribute()` is called on the `RoyaltyManager`, it attempts to swap using the stale proxy address `0x123`. If this proxy is deprecated, compromised, or incompatible with the new swap logic, the transaction may revert, preventing royalty distribution.

   Note that new creators will get a `RoyaltyManager` with the `new proxy`; the problem is with creators who already had a RoyaltyManager created.

Recommendations

New collections should use the `new proxy` in `RoyaltyManager` instances. To achieve this, modify getRoyaltyManager to redeploy RoyaltyManager contracts when the proxy has changed since the creator's existing instance was deployed.

Consider to add an admin function to update the `swapProxy` reference in existing `RoyaltyManager` contracts.

Low findings

[L-01] Suggeset verifying L2 sequencer's status

Wish contracts are deployed on Base Chain. Base chain, as one L2 chain, as one best practice, use the L2 sequencer feed to verify the status of the sequencer when running applications on L2 networks.

```
    function _minOutFromFeeds(uint256 amountInWETH) internal view returns (uint256) {
        SwapImplStateStorage storage $ = _getStorage();
        // ethPxE18 = 4000e18
        (uint256 ethPxE18, uint256 ethAge) = _readE18($.FEED_ETH);
        (uint256 usdcPxE18, ) = _readE18($.FEED_USDC);
        // check stale price.
        if (ethAge > $.MAX_AGE_SEC) revert StalePrice();
        // usdcPxE18 = 1e18, rateE18 = 4000e18
        uint256 rateE18 = (ethPxE18 * 1e18) / usdcPxE18;
        // 1e18 * 4000 e18 / 1e18 = 4000 e18
        uint256 expectedOutE18 = (amountInWETH * rateE18) / 1e18;
        // USDC_DECIMALS = 6
        // 4000e6
        uint256 expectedOut = expectedOutE18 / (10 ** (18 - $.USDC_DECIMALS));
        // add one slippage here.
        return (expectedOut * (10_000 - $.SLIPPAGE_BPS)) / 10_000;
    }
```

Recommendation: Check L2 sequencer status, refer to https://docs.chain.link/data-feeds/l2-sequencer-feeds.

[L-02] Deprecated `answeredInRound` is used in `SwapImpl`

In SwapImpl, we will fetch the current token price to calculate the slippage. In function `_readE18`, we will use `answeredInRound` to check the return value's validity.

The problem here is that, according to https://docs.chain.link/data-feeds/api-reference#functions-in-aggregatorv3interface, the `answeredInRound` is deprecated - Previously used when answers could take multiple rounds to be computed.

```
    function _readE18(AggregatorV3Interface feed) internal view returns (uint256 pxE18, uint256 
age) {
        // https://docs.chain.link/data-feeds/api-reference#functions-in-aggregatorv3interface
        (, int256 ans, , uint256 updatedAt, uint80 answeredInRound) = feed.latestRoundData();

        if (answeredInRound == 0 || ans <= 0) revert BadFeed();
    }
```

Recommendation: Avoid using the deprecated `answeredInRound` to check the return value's validity.

[L-03] `SwapImpl` lacks upgradeability despite proxy pattern indicating upgradeable design

The `SwapImpl` contract does not inherit `UUPSUpgradeable`, breaking the established proxy pattern used throughout the codebase where proxy contracts delegate to upgradeable implementations.

Similar to `WishWishManagerProxy` and `WishWishTokenProxy`, `SwapProxy` is designed as an EIP-1967 compliant proxy that should delegate to an upgradeable implementation. However, `SwapImpl` does not inherit `UUPSUpgradeable`, preventing in-place upgrades of swap logic.

```
File: SwapProxy.sol
/// @dev This contract is a proxy contract for SwapImpl.
contract SwapProxy {
```

This follows the same pattern as other proxies in the codebase:

```
File: WishWishManagerProxy.sol
/// @dev This contract is a proxy contract for WishWishManager.
contract WishWishManagerProxy {
```

Where `WishWishManager` properly inherits `UUPSUpgradeable`:

```
File: WishWishManager.sol
contract WishWishManager is Ownable, EIP712, ReentrancyGuard, UUPSUpgradeable {
```

But `SwapImpl` is missing this inheritance:

```
File: SwapImpl.sol
contract SwapImpl is ReentrancyGuard, Ownable {
```

This architectural inconsistency means swap logic cannot be upgraded if bugs are discovered or new features are needed, unlike other core protocol contracts.

- Make `SwapImpl` inherit `UUPSUpgradeable` to enable upgrades.
- Add storage gap to `SwapImplStateStorage` for future upgrade safety.
- Implement `_authorizeUpgrade` function with proper owner authorization.
- Ensure upgradeability aligns with the established proxy pattern used for other core contracts.

[L-04] Missing storage gaps in upgradeable contracts

The `WishWishToken` and `WishWishManager` contracts are designed to be upgradeable via UUPS proxy pattern, but they do not include reserved storage gaps at the end of their storage layouts.

Both contracts store their state in custom storage structs (`WishWishTokenStateStorage` and `WishWishManagerStateStorage`), future versions may introduce new variables in the base contracts. Without reserved gaps, such additions could lead to storage layout conflicts in future upgrades.

Add a reserved storage gap to `WishWishTokenStateStorage` and `WishWishManagerStateStorage` .

[L-05] Manager rotation breaks legacy collection functionality

The `WishWishManager` uses a proxy pattern where the proxy address remains immutable while the implementation can be upgraded. However, `WishWishToken` and `WishWishFactory` expose `setManager` functions that allow independent manager rotation, breaking the fundamental assumption that the manager is the proxy address.

This creates architectural confusion: the proxy provides upgradeability for the manager role, but individual contracts can override this by changing their manager references independently. The proxy's immutability suggests manager upgrades should only occur through implementation upgrades, not through separate manager fields.

Consider the next scenario:

1. Owner deploys WishWishManager behind a proxy with fixed address 0x123.
2. Owner calls `setManager` on `WishWishToken` , changing manager from 0x123 to 0x456.
3. Legacy collection tries to mint box; WishWishManager calls `token.creditCreator()`

```
File: WishWishManager.sol
348:            WishWishToken(address($.wishToken)).creditCreator(c.creatorAddress,
creatorCut);
```

1. WishWishToken `reverts` with `NotManager` because caller (0x123) ≠ is the current manager (0x456).

```
File: WishWishToken.sol
135:     function creditCreator(address creator, uint256 amount) external onlyManager {
```

1. Legacy and new collections become unusable.

Also, the functions `revealBox` and `distributeRoyalty` may be affected.

- Remove `setManager` functions from `WishWishToken` and `WishWishFactory` since the manager should always be the proxy address.
- If manager rotation is needed, implement it through proxy upgrades rather than independent manager fields.

[L-06] Individual token royalty functionality no longer supported

The `WishWishManager` is the owner of all `WishWishBox` and `WishWishCollectable` contracts deployed during `WishWishManager::launchNewCollection`. The `WishWishBox` and `WishWishCollectable` contracts inherit from `ERC2981`, which means that they are able to hold state relating to royalty information. These contracts can store a default royalty, which will pertain to all `tokenIds` derived from the contract, or they can store a specific token royalty, which only pertains to the specified `tokenId`.

These royalty values can only be set by the `WishWishManager` contract:

```
    function setGlobalRoyalty(uint96 _feeNumerator) external onlyOwner {
        royaltyFeeNumerator = _feeNumerator;
        _setDefaultRoyalty(royaltyReceiver, _feeNumerator);
    }

    /**
     * @notice Sets per-token royalty information.
     */
    function setTokenRoyalty(uint256 _tokenId, address _receiver, uint96 _feeNumerator)
external onlyOwner {
        _setTokenRoyalty(_tokenId, _receiver, _feeNumerator);
    }
```

However, the `WishWishManager` only exposes a `setGlobalRoyalty` function, which means that per-token royalties can no longer be set in the `WishWishBox` and `WishWishCollectable` contracts. If this is intended, consider removing the `setTokenRoyalty` functions altogether and removing the `_resetTokenRoyalty` calls from the `WishWishBox::burnBox` and `WishWishBox::batchBurnBox` functions.

If the contracts are expected to support per-token royalties, then expose a `setTokenRoyalty` function in the `WishWishManager` contract.

[L-07] Missing stale feed check for USDC

The `SwapImpl::_minOutFromFeeds` function is invoked during swapping operations to calculate the acceptable minimum amount out for the swap given prices obtained from the ETH and USDC oracle feeds. During this call, the code validates the age of the ETH price, however, it does not validate the age of the USDC price. Although the price of USDC is not anticipated to vary from time to time as much as ETH, it is still recommended to validate the price age to protect against unforeseen market events.

```
// SwapImpl
    function _minOutFromFeeds(uint256 amountInWETH) internal view returns (uint256) {
        SwapImplStateStorage storage $ = _getStorage();
        (uint256 ethPxE18, uint256 ethAge) = _readE18($.FEED_ETH);
        (uint256 usdcPxE18, ) = _readE18($.FEED_USDC);

        if (ethAge > $.MAX_AGE_SEC) revert StalePrice();

        uint256 rateE18 = (ethPxE18 * 1e18) / usdcPxE18;

        uint256 expectedOutE18 = (amountInWETH * rateE18) / 1e18;
        uint256 expectedOut = expectedOutE18 / (10 ** (18 - $.USDC_DECIMALS));

        return (expectedOut * (10_000 - $.SLIPPAGE_BPS)) / 10_000;
    }
```

Consider validating the age of the USDC price as well.

[L-08] `protocolCut` not actually burned during `_mintBox`

Code comments suggest that the `WishWishManager::_mintBox` function calculates a `protocolCut` meant to be burned when users mint boxes. However, instead of invoking a burn function, the code simply transfers the tokens to `address(0)`, which does not reduce the `totalSupply`. As a result, the tokens remain in circulation.

```
// WishWishManager.sol
    function _mintBox(MintData calldata mintData, bytes calldata signature) internal {
...
        if (mintData.fee != 0) {
            uint256 protocolCut = (mintData.fee * $.MINT_FEE_PERCENT) / 1 ether;
            creatorCut = mintData.fee - protocolCut;
            $.wishToken.transferFrom(msg.sender, address(0), protocolCut); // burn protocol fee
            $.wishToken.transferFrom(msg.sender, c.creatorAddress, creatorCut);
            WishWishToken(address($.wishToken)).creditCreator(c.creatorAddress, creatorCut);
        }
...
```

Note that it is still possible for the protocol to recover the `protocolCut` tokens sent to `address(0)` since the `$.manager` state variable can be updated and the `ERC20::_transfer` function (inherited from Solady) does not have any restrictions against transferring to or from `address(0)`:

```
// WishWishToken.sol
    function transferFrom(address from, address to, uint256 amount) public override returns
(bool) {
        WishWishTokenStateStorage storage $ = _getStorage();
        if (msg.sender != $.manager) revert TransfersRestricted();
        _transfer(from, to, amount);
        return true;
    }
```

```
// ERC20.sol
    /// @dev Moves `amount` of tokens from `from` to `to`.
    function _transfer(address from, address to, uint256 amount) internal virtual {
        _beforeTokenTransfer(from, to, amount);
        /// @solidity memory-safe-assembly
        assembly {
            let from_ := shl(96, from)
            // Compute the balance slot and load its value.
            mstore(0x0c, or(from_, _BALANCE_SLOT_SEED))
            let fromBalanceSlot := keccak256(0x0c, 0x20)
            let fromBalance := sload(fromBalanceSlot)
            // Revert if insufficient balance.
            if gt(amount, fromBalance) {
                mstore(0x00, 0xf4d678b8) // `InsufficientBalance()`.
                revert(0x1c, 0x04)
            }
            // Subtract and store the updated balance.
            sstore(fromBalanceSlot, sub(fromBalance, amount))
            // Compute the balance slot of `to`.
            mstore(0x00, to)
            let toBalanceSlot := keccak256(0x0c, 0x20)
            // Add and store the updated balance of `to`.
            // Will not overflow because the sum of all user balances
            // cannot exceed the maximum uint256 value.
            sstore(toBalanceSlot, add(sload(toBalanceSlot), amount))
            // Emit the {Transfer} event.
            mstore(0x20, amount)
            log3(0x20, 0x20, _TRANSFER_EVENT_SIGNATURE, shr(96, from_), shr(96, mload(0x0c)))
        }
        _afterTokenTransfer(from, to, amount);
    }
```

If the intention is to actually burn the `protocolCut`, then consider introducing a `burnFrom` function in `WishWishToken.sol`, callable only by the manager, that internally invokes `ERC20::_burn()` to properly destroy tokens and decrement the total supply.

[L-09] Using `block.timestamp` as swap deadline removes MEV protection

The `SwapImpl` contract uses `block.timestamp` as the deadline parameter when calling the Uniswap Universal Router's `execute` function in both `swapETHToUSDC` (line 107) and `swapWETHToUSDC` (line 130). This effectively disables deadline protection since `block.timestamp` always represents the current block time when the transaction is mined, regardless of how long it sat in the mempool.

```
File: SwapImpl.sol
106:        uint256 balBefore = $.USDC.balanceOf(msg.sender);
107:        $.router.execute{ value: amountIn }(commands, inputs, block.timestamp);
108:        uint256 balAfter = $.USDC.balanceOf(msg.sender);
...
...
129:        uint256 balBefore = $.USDC.balanceOf(msg.sender);
130:        $.router.execute(commands, inputs, block.timestamp);
131:        uint256 balAfter = $.USDC.balanceOf(msg.sender);
```

Consider the following scenario:

1. Attacker observes a pending `distribute()` call with 10 ETH to swap.
2. Oracle-based `minOut` was calculated when ETH = $2000, expecting ~$20,000 USDC (minus slippage).
3. Attacker delays transaction execution until ETH drops to $1800.

4. Transaction executes at $1800 rate, getting only ~$18,000 USDC, but `minOut` from $2000 price still passes because slippage tolerance is percentage-based.
5. Creators lose ~$2000 in royalty value.

Recommendations

Add a user-controlled `deadline` parameter. This allows callers to specify a reasonable deadline ensuring swaps execute within an acceptable timeframe or revert, protecting against MEV extraction and stale price execution.

[L-10] permissionless `distribute()` allows self sandwiching for profit

`RoyaltyManager::distribute()` being permissionless allows value extraction up to the slippage % either through a mev bot or through the creator himself via self sandwiching, since base doesn't have a public memepool.

The attack will be as follows: - attacker listens for royalty distribution events. - calls distribute() and manipulate the pool to extract up to the slippage amounts of the royalties.

So basically it will be guaranteed to lose the slippage amount.

Attacker here can also be the creator, since if he extract the slippage amounts from the royalties received, he will pay less `ROYALTY_FEE_PERCENT`, which will be applied on the remaining amounts after the attack.

For a 30 usdc distribution and a slippage of 10% and `ROYALTY_FEE_PERCENT` = 5%, the economics become the following: - slippage amounts extracted = 3$. - protocol cut = 1.35$. - lost protocol cut 0.15$. avg txn fees: 0.015 -> 0.030$.

This is in terms of protocolCut loss, for users loss it will be guaranteed he loses his slippage everytime.

So it will be economically profitable for an attacker and economically feasible if the protocol chooses to handle distribution.

Recommendations

Guard the distribution to be by an off-chain backend bot address controlled by the team.