Aperture UniV3Automan

Security Testing and Assessment

May 8, 2023

Prepared for Aperture

Narya.ai

Table of Content

Summary

Overview

From April 17, 2023, to May 8, 2023, Aperture engaged Narya.ai to evaluate the security of its UniV3Automan in the GitHub repository: https://github.com/Aperture-Finance/core-contracts (commit 2a8975e91e1371fa23b268b30c8959f95027dafb).

UniV3Automan is a contract that exposes external functions that allow Aperture's automation service to trigger these actions for Uniswap V3 liquidity position holders:

- Close an existing position.
- Add or Remove Liquidity.
- Rebalance a position to another price range.
- Reinvest the accrued fees into the position's liquidity.

Project Scope

We reviewed and tested the smart contracts that implement the UniV3Automan product. Other security-critical components of UniV3Automan, such as off-chain services and the web front end, are not included in the scope of this security assessment. We recommend a further review of those components.

Summary of Findings

Severity	# of Findings
High	0
Medium	0
Low	3
Informational	0
Gas	1
Total	4

Throughout the testing, we identified 3 low-severity issues and 1 gas issue:

- 1 issue of unequal `msg.value` and amounts in mint parameters that may leave leftover native ETH on the UniV3Automan contract.
- 1 issue of stealing contract's ETH to mint NFT positions.

- 1 issue of Arbitrary function call in `_routerSwap`.
- 1 gas issue of unnecessary Check on the native ETH balance.

Disclaimer

This report should not be used as investment advice.

Narya.ai uses an automatic testing technique to test smart contracts' security properties and business logic rapidly and continuously. However, we do not provide any guarantees on eliminating all possible security issues. The technique has its limitations: for example, it may not generate a random edge case that violates an invariant during the allotted time. Its use is also limited by time and resource constraints.

Unlike time-boxed security assessment, Narya.ai advises continuing to create and update security tests throughout the project’s lifetime. In addition, Narya.ai recommends proceeding with several other independent audits and a public bug bounty program to ensure smart contract security.

Project Overview

Aperture's UniV3 Automan is a single contract exposing the following functionality for Uniswap V3 liquidity position holders:

- Close an existing position.
- Add or remove Liquidity.
- Rebalance a position to another price range.
- Reinvest the accrued fees into the position's liquidity.

Tests

Tested Invariants and Properties

We relied on the Narya engine that used a smart fuzzing approach to test the following 6 invariants and properties of the smart contracts.

ID	Invariant/Property Description	Found Bug(s)
01	When calling mintOptimal() with an encoded function signature and argument, it should not revert with the specific error string of that function. If it does, a controllable arbitrary function call can be made in the whitelister router.	Fail
02	After a successful mint a user should gain one more NFT and have less funds since he/she spent them.	Pass
03	After a successful call to remove liquidity a user should lose one NFT and have more funds as he/she was refunded the liquidity.	Pass
04	After a successful rebalance, the amount of NFT held should not change.	Pass
05	After a successful liquidity increase or reinvest(), the amount of NFT held stays the same, but the position's liquidity goes up.	Pass
06	After a successful liquidity reduction, the amount of NFT stays the same, but the position's liquidity goes down.	Pass

Key Findings and Recommendations

1. Unequal `msg.value` and amounts in mint parameter may leave leftover native ETH on the UniV3Automan contract

Severity: **Low**

Description

Users that mistakenly send a different amount of native ether than what was specified in `amount0Desired` and/or `amount1Desired` to the mint function, won't be able to get back the leftovers. This function will call `pullAndApprove` to pull the exact 2 amounts regardless of what was given in `msg.value`.

Code 1 src/UniV3Automan.sol#L729

```
function mint(
     INPM.MintParams memory params
 )
     external
     payable
     returns (
         uint256 tokenId,
         uint128 liquidity,
         uint256 amount0,
         uint256 amount1
     )
 {
     pullAndApprove(
         params.token0,
         params.token1,
         params.amount0Desired,
         params.amount1Desired
     );
     (tokenId, liquidity, amount0, amount1) = _mint(params);
     emit Mint(tokenId);
 }
```

Impact

The immediate consequence is that users that mistakenly send native ETH to the contract cannot get back the surplus. Another consequence of the contract having a positive balance of native ETH can be related to the 2nd issue (2. Stealing of contract's ETH).

Failed Invariant

Found during code review.

Recommendation

Add a function (like Uniswap's refundETH) to allow users to refund themselves.

Remediation

This issue has been acknowledged by Aperture and fixed at commit [3bf4ac8a4fa2348a18bc54caa4c8ea11d62fbf8f](3bf4ac8a4fa2348a18bc54caa4c8ea11d62fbf8f).

2. Stealing of contract's ETH

Severity: **Low**

Description

If UniV3Automan holds native ETH, an attacker can trick the mint() function into using the contract's balance to mint a new NFT instead of sending any ETH to the UniV3utoman contract. Inside of the pay() function, it only checks for `address(this).balance >= value` which doesn't account for if the user sent any funds at all.

Code 2 src/base/Payments.sol#L32

```
    function pay(
        address token,
        address payer,
        address recipient,
        uint256 value
    ) internal {
        // Receive native ETH
        if (token == WETH9 && address(this).balance >= value) {
            // Wrap it
            WETHCallee.wrap(WETH9).deposit(value);
            // Already received native ETH so return
            if (recipient == address(this)) return;
        }
        if (payer == address(this)) {
            // Send token to recipient
            token.safeTransfer(recipient, value);
        } else {
            // pull payment
            token.safeTransferFrom(payer, recipient, value);
        }
    }
```

Impact

If the contract holds any native ETH balance, an attacker can mint for himself new positions using the contract's balance instead of his/her own funds.

Failed Invariant

Found during code review.

Recommendation

A check should be added to make sure that the user did send the requested amount, instead of checking that the contract holds enough funds.

Remediation

This issue has been acknowledged by Aperture and fixed at commit [78fc4820494f4c8b164cf5b4249d2f32b9e4228e](78fc4820494f4c8b164cf5b4249d2f32b9e4228e).

3. Arbitrary function call in _routerSwap

Severity: **Low**

Description

The SwapData argument is coming from the user (for eg. mintOptimal()) and contains the function signature and arguments. This can result in a low level arbitrary call inside a whitelisted router.

Code 3 src/base/SwapRouter.sol#L149

```
    function _routerSwap(
        PoolKey memory poolKey,
        address router,
        bool zeroForOne,
        bytes calldata swapData
    ) internal returns (uint256 amountOut) {
        address tokenIn;
        address tokenOut;
        if (zeroForOne) {
            tokenIn = poolKey.token0;
            tokenOut = poolKey.token1;
        } else {
            tokenIn = poolKey.token1;
            tokenOut = poolKey.token0;
        }
        uint256 balanceBefore = ERC20Callee.wrap(tokenOut).balanceOf(
            address(this)
        );
        // Approve `router` to spend `tokenIn`
        tokenIn.safeApprove(router, type(uint256).max);
        /*
            If `swapData` is encoded as `abi.encode(router, data)`, the memory
layout will be:
            0x00         : 0x20         : 0x40         : 0x60         : 0x80
            total length : router       : 0x40 (offset): data length  : data
            Instead, we encode it as:
            ```
            bytes memory swapData = abi.encodePacked(router, data);
            ```
            So the memory layout will be:
            0x00         : 0x20         : 0x34
            total length : router       : data
            To decode it in memory, one can use:
            ```
            bytes memory data;
```
```

```
            assembly {
                router := shr(96, mload(add(swapData, 0x20)))
                data := add(swapData, 0x14)
                mstore(data, sub(mload(swapData), 0x14))
            }
            ```
            knowing that `data.length == swapData.length - 20`.
         */
        assembly ("memory-safe") {
            let fmp := mload(0x40)
            // Strip the first 20 bytes of `swapData` which is the router address.
            let calldataLength := sub(swapData.length, 20)
            calldatacopy(fmp, add(swapData.offset, 20), calldataLength)
            // Ignore the return data unless an error occurs
            if iszero(call(gas(), router, 0, fmp, calldataLength, 0, 0)) {
                returndatacopy(0, 0, returndatasize())
                // Bubble up the revert reason.
                revert(0, returndatasize())
            }
        }
        // Reset approval
        tokenIn.safeApprove(router, 0);
        uint256 balanceAfter = ERC20Callee.wrap(tokenOut).balanceOf(
            address(this)
        );
        amountOut = balanceAfter - balanceBefore;
    }
```

## Impact

This issue has a limited impact as the UniV3Automan is not holding any tokens. Furthermore, even if the user approved UniV3Automan to spend tokens on his/her behalf, no calls in UniV3Automan can be used by the router to pull tokens using that user's address.

## Failed Invariant

```
    function invariantArbitraryCall() public {
         uint tokenId = npm.tokenByIndex(0);
         // console.log("id", tokenId);

         (uint96 nonce, address operator, address _token0, address _token1,
uint24 _fee, int24 tickLower, int24 tickUpper, uint128 liquidity, uint256
feeGrowthInside0LastX128, uint256 feeGrowthInside1LastX128, uint128
```
```

```
tokensOwed0, uint128 tokensOwed1) = npm.positions(tokenId);

        // console.log("token0", _token0);
        // console.log("token1", _token1);
        // console.log("fee", _fee);

        INPM.MintParams memory params;
        params.amount0Desired = 0;
        params.amount1Desired = 0;
        params.token0 = _token0;
        params.token1 = _token1;
        params.fee = _fee;
        params.tickLower = tickLower;
        params.tickUpper = tickUpper;

        address router = address(naryaRouter);

        bytes memory swapData = abi.encode(
            router,
            abi.encodeWithSelector(NaryaRouter.arbitraryCall.selector, 42)
        );

        assembly {
            let length := mload(swapData)
            swapData := add(swapData, 0x40)
            mstore(swapData, sub(length, 0x40))
            mstore(add(swapData, 0x20), router)
        }

        try automan.mintOptimal(params, swapData) returns (uint256 tokenId,
uint128 liquidity, uint256 amount0, uint256 amount1) {

        } catch Error(string memory reason) {
            if (keccak256(abi.encodePacked(reason)) ==
keccak256(abi.encodePacked("arbitrary call"))) {
                require(false, "arbitrary call inside NaryaRouter was
called");
            }
        }
    }
}
```

Recommendation

Special attention should be paid to future developments to ensure that the conditions in the impact section are not met.

Remediation

This issue has been acknowledged by Aperture. Regarding the concerns about external router swap, UniV3Automan explicitly checks if the swap router is whitelisted. The risk involved with a malicious router is documented. Moreover even if a malicious router is whitelisted, only whitelisted controllers can call functions that can modify approved LPs and drain user's fund.

4. Unnecessary Check on the native ETH balance

Severity: Gas

Description

Inside refund() in Payments.sol, the first IF block checks if the contract holds enough native ETH to send to the recipient. This is unnecessary as the contract should not hold any native ETH and only uses WETH to issue a refund. The check on the WETH balance is enough for that purpose.

Code 4 src/base/Payments.sol#L54

```
    function refund(address token, address recipient, uint256 value) internal {
        if (token == WETH9) {
            uint256 balance = address(this).balance;
            if (balance < value) {
                uint256 wethBalance = ERC20Callee.wrap(WETH9).balanceOf(
                    address(this)
                );
                if (wethBalance >= value)
                    WETHCallee.wrap(WETH9).withdraw(value);
                else revert InsufficientETH();
            }
            // Send native ETH to recipient
            recipient.safeTransferETH(value);
        } else {
            token.safeTransfer(recipient, value);
        }
    }
```

Impact

The function could be more gas efficient with only the check on the WETH balance.

Failed Invariant

Found during code review.

Recommendation

Remove the check on the native ETH balance as the contract is not supposed to use native ETH for the refund.

Remediation

This issue has been acknowledged by Aperture and fixed at commit [78fc4820494f4c8b164cf5b4249d2f32b9e4228e](78fc4820494f4c8b164cf5b4249d2f32b9e4228e).

Fix Log

ID	Title	Severity	Status
01	1. Unequal `msg.value` and amounts in mint parameter may leave leftover native ETH on the UniV3Automan contract	Low	Fixed at commit 3bf4ac8a4fa2348a18bc54caa4c8ea11d62fbf8f
02	2. Stealing of contract's ETH	Low	Fixed at commit 78fc4820494f4c8b164cf5b4249d2f32b9e4228e
03	3. Arbitrary function call in _routerSwap	Low	Confirmed and Mitigated*
04	4. Unnecessary Check on the native ETH balance	Gas	Fixed at commit 78fc4820494f4c8b164cf5b4249d2f32b9e4228e

* The issues have been acknowledged by Aperture but won't be fully fixed at the time of this report. Check the former description of each issue for details.

Appendix

Severity Categories

Severity	Description
Gas	Gas optimization.
Informational	The issue does not pose a security risk but is relevant to best security practices.
Low	The issue does not put assets at risk such as functions being inconsistent with specifications or issues with comments.
Medium	The issue puts assets at risk not directly but with a hypothetical attack path, stated assumptions, or external requirements. Or the issue impacts the functionalities or availability of the protocol.
High	The issue directly results in assets being stolen, lost, or compromised.