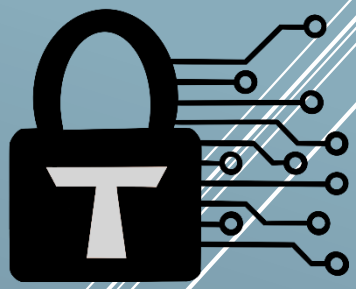
Trust Security

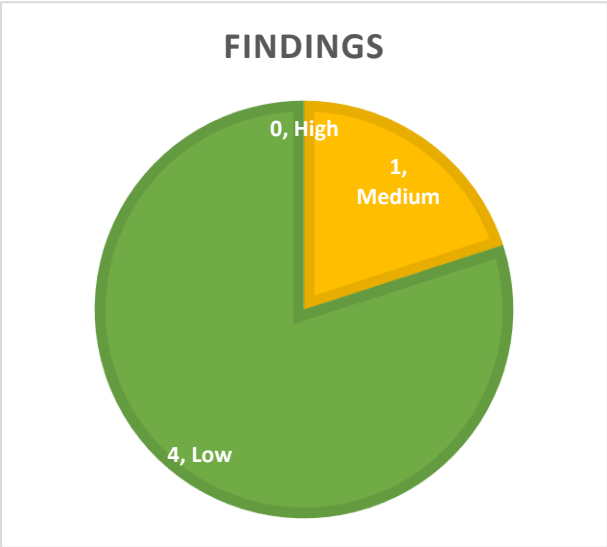


Smart Contract Audit

Universal Swaps

23/02/24

Executive summary

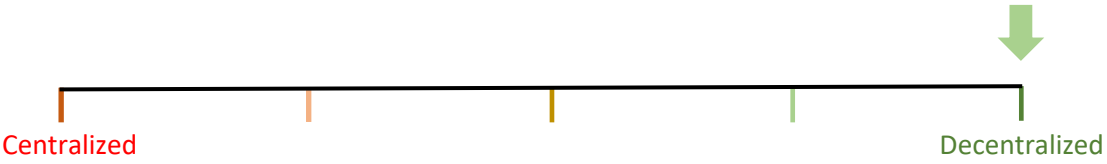


Category	AMM
Auditor	Trust
Time period	14/02/24-16/02/24

Findings

Severity	Total	Fixed	Acknowledged
High	0	-	-
Medium	1	1	-
Low	4	3	1

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.23 16:27:05 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	16/02/24	Client report
0.2	23/02/24	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

Any changes from the base commit hash to the target commit hash are in-scope, except EnumerableSet and ERC165Checker which were declared a copy of the original libraries. Furthermore, the LSP standard contracts and the Uniswap-provided contracts are assumed to be safe.

Repository details

- **Repository URL:** https://github.com/Universal-Swaps/protocol-contracts
- **Base commit hash:** 2b8cbf77d0519f0177a3d961ea84e3d4692b73b0
- **Target commit hash:** a333741a76e52ff8f6d9fa8f3c6a4b8248c79312
- **Mitigation base commit hash:** 6ebba1ddbb49a0d9dd5add1fd113ec2957f2fef2
- **Mitigation target commit hash:** 752f58bc5889082fc78ad5216fee2bc92a93e1be

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks. It is noted that the Universal Receiver functionality creates inevitable complexity.
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Excellent**	Project avoided any significant centralization risks.

Findings

Medium severity findings

TRST-M-1 The deployed protocol fee amounts are incorrect

- **Category:** Logical flaws
- **Source:** UniswapV3Factory.sol
- **Status:** Fixed

Description

The original Uniswap factory was changed to introduce the **deploymentProtocolFee**. It is initialized below:

```
deploymentProtocolFee = 255;
emit DeploymentProtocolFeeChanged(0, 255);
```

This is encoded as (15,15) in the Pool:

```
function setFeeProtocol(uint8 feeProtocol0, uint8 feeProtocol1) external
override lock onlyFactoryOwner {
    require(
        (feeProtocol0 == 0 || (feeProtocol0 >= 4 && feeProtocol0 <= 15)) &&
            (feeProtocol1 == 0 || (feeProtocol1 >= 4 && feeProtocol1 <= 15))
    );
    uint8 feeProtocolOld = slot0.feeProtocol;
    slot0.feeProtocol = feeProtocol0 + (feeProtocol1 << 4);
    emit SetFeeProtocol(feeProtocolOld % 16, feeProtocolOld >> 4,
feeProtocol0, feeProtocol1);
}
```

The issue is that the client intended protocol fees to be 15%, however each value is the denominator when calculating the fee split:

```
// if the protocol fee is on, calculate how much is owed, decrement
feeAmount, and increment protocolFee
if (cache.feeProtocol > 0) {
    uint256 delta = step.feeAmount / cache.feeProtocol;
    step.feeAmount -= delta;
    state.protocolFee += uint128(delta);
}
```

Therefore, fees are off by (15% – 1/15) = 8.33%.

Recommended mitigation

Use the calculation logic explained above to determine the deployment values.

Team response

Reverted the new changes and kept the original fee calculation mechanism.

Mitigation review

The code has been corrected so that the default is 1/6 = 16.666% fees, as intended.

Low severity findings

TRST-L-1 The Owner is not able to add extensions to WLYX contract

- **Category:** Logical flaws
- **Source:** WETH.sol
- **Status:** Acknowledged

Description

The WLYX token (equivalent to WETH in LUKSO) is inherited from LSP7. The LSP7 standard supports contract extensions, meaning the owner can add additional fragments of logic on the token contract. These execute through the *fallback()* function which will match the target selector. WLYX is defined below:

```
// owner can add metadata or add extensions, we can remove it later if we
feel it
constructor() LSP7DigitalAsset("WrappedLYX", "WLYX", msg.sender, 0, false) {
    // No code needed
    // Add metadata later
}
receive() external payable override {
    deposit();
}
fallback() external payable override {
    deposit();
}
```

Note that the LSP7 *fallback()* is overridden, which causes loss of the extension functionality, contrary to the comment above the constructor.

Recommended mitigation

Consider refactoring the *fallback()* to make use of the LSP7 extension logic.

Team response

WLYX should not have extensions, not fixed.

TRST-L-2 The UniversalRouter loses interoperability with fee-on-transfer tokens

- **Category:** Compatibility issues
- **Source:** UniversalRouter.sol
- **Status:** Fixed

Description

The Router operates through an *authorizeOperator()* call by the user on the LSP7 contract, with the swap calldata to forward. This notifies the operator which must make use of the funds immediately, or else the approved amount can be stolen by forging of the **from** parameter in the *universalReceiver()*. This is well-understood by the team.

In order to safeguard the user's assets, it will reset any unspent approval on the Router, through sending the tokens back and forth.

```
// making sure that there are no authorized amount left over and send it
back to owner if that is the case
uint256 remainingAuthorizedAmount =
ILSP7(msg.sender).authorizedAmountFor(address(this), from);
if(remainingAuthorizedAmount != 0) {
    address[] memory tokenOwners = new address[](2);
    tokenOwners[0] = from;
    tokenOwners[1] = address(this);
    address[] memory recipients = new address[](2);
    recipients[0] = address(this);
    recipients[1] = from;
    uint256[] memory amounts = new uint256[](2);
    amounts[0] = remainingAuthorizedAmount;
    amounts[1] = remainingAuthorizedAmount;
    bool[] memory force = new bool[](2);
    force[0] = true;
    force[1] = true;
    bytes[] memory datas = new bytes[](2);
    datas[0] = "";
    datas[1] = "";
    ILSP7(msg.sender).transferBatch(tokenOwners, recipients, amounts, force,
datas);
}
```

The issue is that the same amount is specified in both the swap into the Router and back to the user. When the token has fee-on-transfer mechanism, there will not be enough tokens in the contract to fulfill the transfer out, reverting the operation.

Recommended mitigation

Perform the transfer in two steps (not in batch), and send tokens out by calculating the current *balanceOf()*. It must be documented that the user will lose tax for the two transfers.

Team response

Fixed.

Mitigation review

The code has been fixed as suggested. It is missing documentation of the tax loss behavior.

TRST-L-3 The LSP8 *getOperatorsOf()* function would be calculated incorrectly when there are no operators

- **Category:** Specification issues
- **Source:** NonfungiblePositionManager.sol
- **Status:** Fixed

Description

LSP8 defines *getOperatorsOf()* for the asset. It is implemented in the Position Manager below:

```
function getOperatorsOf(
    bytes32 tokenId
) public view virtual override returns (address[] memory) {
    _existsOrError(tokenId);
```

```
    address[] memory operators_ = new address[](1);
    operators_[0] = _positions[uint256(tokenId)].operator;
    return operators_;
}
```

Note that if *operator* is zero, the code should return an empty array, however due to the changed implemented it would be an array with content of zero. This could harm compatibility with other contracts which would assume the operator is a valid address, if one exists.

Recommended mitigation

Check if the *operator* is zero, and return an empty array in this case.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-4 The PositionManager is prone to reentrancy attacks

- **Category:** Reentrancy attacks
- **Source:** NonfungiblePositionManager.sol
- **Status:** Fixed

Description

The *collect()* function is used to pick up any owed fees from the pool. Note that the tokens owed are cached at the start of the function.

```
(uint128 tokensOwed0, uint128 tokensOwed1) = (position.tokensOwed0,
position.tokensOwed1);
```

They are updated at the end:

```
(amount0, amount1) = pool.collect(
    recipient,
    position.tickLower,
    position.tickUpper,
    amount0Collect,
    amount1Collect
);
// sometimes there will be a few less wei than expected due to rounding down
in core, but we just subtract the full amount expected
// instead of the actual amount so we can burn the token
(position.tokensOwed0, position.tokensOwed1) = (tokensOwed0 -
amount0Collect, tokensOwed1 - amount1Collect);
```

This means any changes to the **tokensOwed0/1** in the position from *pool.collect()* until the update would be wiped, which could mean allowing a user to withdraw owed tokens twice. The LSP7 token design means *collect()* will definitely allow an attacker to execute code at this point. Coincidentally, the core Pool's lock seems to be sufficient at preventing corruption of the **tokensOwed**, however this should not be counted on.

Recommended mitigation

Update the position storage before the *pool.collect()* call.

Team response

Fixed.

Mitigation review

Fixed as suggested.

Additional recommendations

TRST-R-1 Improve fee protocol synchronization

To update the Pool's protocol fee %, *setFeeProtocol()* must be called by the factory owner. This design is not optimal, since in this codebase the fee can be determined from the factory using **deploymentProtocolFee**. Consider allowing anyone to update the fee, which will fetch this value from the factory.

TRST-R-2 Improve safe transfers of LYX

The Periphery contract makes use of TransferHelper, which defines *safeTransferETH()*:

```
function safeTransferETH(address to, uint256 value) internal {
    (bool success, ) = to.call{value: value}(new bytes(0));
    require(success, 'STE');
}
```

This code has a more optimized version in LSP7SafeTransferLib:

```
function safeTransferETH(address to, uint256 amount) internal {
    bool success;
    /// @solidity memory-safe-assembly
    assembly {
        // Transfer the ETH and store if it succeeded or not.
        success := call(gas(), to, amount, 0, 0, 0, 0)
    }
    require(success, "ETH_TRANSFER_FAILED");
}
```

The code does not read any returndata from the contract, offering better protection from gas griefing attacks and better performance.