Smart Contract Security Audit Report

Tokenlon

1. Contents

2. General Information

This report contains information about the results of the security audit of the Tokenlon (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 03/07/2023 to 14/07/2023.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: https://github.com/consenlabs/tokenlon-contracts. Initial review was done for the commit b5038f48ef7c41fe0a9a57f888ffdb846c91de3b and the re-testing was done for the commit 4b195cd1a9aba8c53a26150d0491985067ad4e27.

The following contracts have been tested:

- AllowanceTarget.sol
- CoordinatedTaker.sol
- GenericSwap.sol
- LimitOrderSwap.sol
- RFQ.sol
- SmartOrderStrategy.sol
- UniAgent.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Liquidity Token owner/contract.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	Actor
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a re-entrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided "as is" and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer's project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have discovered several low security issues. The other suggestions included some best practices and gas optimization recommendations.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of August 02, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Fees at LimitOrderSwap and RFQ are completely controlled by user	contracts/LimitOrderSwap.sol contracts/RFQ.sol	Low	Acknowledged
RFQ offer can be cancelled during execution	contracts/RFQ.sol	Low	Fixed

Issue	Contract	Risk Level	Status
Limit order can be cancelled during execution	contracts/LimitOrderSwap.sol	Low	Fixed
Maker and taker token permit not signed in RFQ	contracts/RFQ.sol	Low	Acknowledged
Modifiers from AdminManagement can be bypassed in SmartOrderStrategy	contracts/SmartOrderStrategy.sol	Low	Acknowledged
Tokens can be swept from LimitOrderSwap	contracts/LimitOrderSwap.sol	Low	Acknowledged
Missing 0x0 address check	contracts/LimitOrderSwap.sol contracts/GenericSwap.sol contracts/RFQ.sol	Low	Fixed
Outdated OpenZeppelin library		Low	Fixed
Unnecessary variable casting	contracts/CoordinatedTaker.sol	Info	Fixed
Unnecessary `memory` argument location	contracts/LimitOrderSwap.sol	Info	Fixed
Lack of makerSettleAmount!=0 check	contracts/RFQ.sol	Info	Fixed
Unnecessary weth variable	contracts/UniAgent.sol	Info	Fixed
Unsafe usage of abi.encodeWithSelector	contracts/UniAgent.sol	Info	Fixed

Issue	Contract	Risk Level	Status
Array length not cached outside of the loop	contracts/AllowanceTarget.sol contracts/SmartOrderStrategy.sol	Info	Fixed
Unnecessary storage access	contracts/LimitOrderSwap.sol	Info	Fixed
Non-optimal conditional statements	contracts/GenericSwap.sol contracts/LimitOrderSwap.sol contracts/RFQ.sol contracts/UniAgent.sol	Info	Fixed
String error messages used instead of custom errors	contracts/AllowanceTarget.sol contracts/SmartOrderStrategy.sol contracts/abstracts/Ownable.sol contracts/libraries/Asset.sol	Info	Fixed
Non-optimal structs	contracts/libraries/GenericSwapData.sol	Info	Acknowledged
Unnecessary checked arithmetic	contracts/LimitOrderSwap.sol contracts/AllowanceTarget.sol contracts/SmartOrderStrategy.sol contracts/UniAgent.sol contracts/abstracts/AdminManagement.sol contracts/RFQ.sol	Info	Fixed
Non-optimal order of operations	contracts/CoordinatedTaker.sol	Info	Fixed
Unused imports	contracts/AllowanceTarget.sol contracts/LimitOrderSwap.sol contracts/SmartOrderStrategy.sol contracts/UniAgent.sol	Info	Fixed

4. General Recommendations

This section contains general recommendations on how to improve overall security level. The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Fees at LimitOrderSwap and RFQ are completely controlled by user

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/LimitOrderSwap.sol
- contracts/RFQ.sol

Description:

Function `_fillLimitOrder` at `LimitOrderSwap` at line 166 calculates fees based on `order.feeFactor` supplied by user.

Function `fillLimitOrderGroup` at `LimitOrderSwap` at line 107 calculates fees based on `order.feeFactor` supplied by user.

Function `_fillRFQ` at `RFQ` at line 133 calculates fees based on `_rfqOffer.feeFactor` supplied by user.

This results in the following issues:

1) Users may supply `feeFactor` equal to 0 which cause protocol getting 0 revenue
2) In case something goes wrong with frontend fees can be as high as `Constant.BPS_MAX`, which is basically 100% of token amount.

Remediation:

Consider setting minimum value for `feeFactor` to prevent 0 fees. Also consider setting maximum value for `feeFactor` to prevent fees as high as 100% from amount.

5.2. RFQ offer can be cancelled during execution

Risk Level: Low

Status: Fixed in the commit af0c69.

Contracts:

- contracts/RFQ.sol

Location: Lines: 74. Function: `cancelRFQOffer`.

Description:

When transferring funds to maker in `_fillRFQ` at line 114, maker in case it's a contract can call `cancelRFQOffer`. This will result in both `CancelRFQOffer` and `FilledRFQ` in a single transaction, what may cause incorrect processing at frontend.

Remediation:

Consider checking that `filledOffer[rfqOfferHash] != true;` before allowing to cancel offer to prevent this. This check will also prevent double canceling of a single offer.

5.3. Limit order can be cancelled during execution

Risk Level: Low

Status: Fixed in the commit c5e363.

Contracts:

- contracts/LimitOrderSwap.sol

Location: Lines: 132. Function: `cancelOrder`.

Description:

When transferring assets to maker in `_fillLimitOrder` at line 179, maker address in case it is contract may call `cancelOrder` function, because it doesn't have `nonReentrant` modifier. This will result in both `OrderCanceled` and `LimitOrderFilled` in a single transaction, what may result in an incorrect processing at frontend.

Remediation:

Consider adding `nonReentrant` modifier to prevent order cancelation during filling.

5.4. Maker and taker token permit not signed in RFQ

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/RFQ.sol

Description:

The `makerTokenPermit` and `takerTokenPermit` parameters of the `fillRFQ()` function in the `RFQ` contract are not included in the signed order hash and therefore can be forged.

This may lead to a griefing attack.

Remediation:

Consider adding the `makerTokenPermit` and `takerTokenPermit` to the order structure.

5.5. Modifiers from AdminManagementcan be bypassed in SmartOrderStrategy

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/SmartOrderStrategy.sol

Description:

The `_call()` function can be used to perform any type of transaction through the GenericSwap contract:

```
contracts/SmartOrderStrategy.sol:
  95:         (bool success, bytes memory result) = _dest.call{ value: _value
}(_data);
```

The `approveTokens()` and `rescueTokens()` functions that were inherited from `AdminManagement` are redundant and their functional with `onlyOwner` modifier could be bypassed and executed through the `_call()` function of the `SmartOrderStrategy` contract.

Remediation:

Consider removing the inheritance from the `AdminManagement` abstract contract.

5.6. Tokens can be swept from LimitOrderSwap

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/LimitOrderSwap.sol

Description:

Orders of the `GroupFill` type can be used to group many orders and fulfill them together. The motivation to look for and submit it would be the ability for a user to identify this type of group with profits. Arbitrageur can set the `profitTokens` array during the call to the `fillLimitOrderGroup()` function to extract the profit from executed orders.

```
contracts/LimitOrderSwap.sol:
  124:         // any token left is considered as profit
  125:         for (uint256 i = 0; i < profitTokens.length; ++i) {
  126:             uint256 profit = profitTokens[i].getBalance(address(this));
  127:             profitTokens[i].transferTo(payable(msg.sender), profit);
  128:         }
```

However, the profit couldn't be paid in WETH because all WETH tokens were unwrapped earlier:

```
contracts/LimitOrderSwap.sol:
  113:         // unwrap extra WETH in order to pay for ETH taker token and
profit
  114:         uint256 wethBalance = weth.balanceOf(address(this));
  115:         if (wethBalance > wethToPay) {
  116:             weth.withdraw(wethBalance - wethToPay);
  117:         }
```

In case when the user got profit in WETH and didn't pass `ETH_ADDRESS` or `ZERO_ADDRESS` in `profitTokens` array the profit may be lost. Thereby the profit will be collected by another user.

Remediation:

Consider mentioning in the docs that all profit that could be made in WETH from the execution of the `fillLimitOrderGroup()` function would be withdrawn and should be taken in raw ETH.

5.7. Missing 0x0 address check

Risk Level: Low

Status: Fixed in the pull request.

Contracts:

- contracts/LimitOrderSwap.sol
- contracts/GenericSwap.sol
- contracts/RFQ.sol

Description:

There is no check that `takerParams.recipient` and `_swapData.recipient` addresses are not equal to zero.

Also, the `_feeCollector` variable in the `LimitOrderSwap` and `RFQ` constructors should be checked. In case when `feeCollector` is equal to `address(0)` fees may be stuck in `RFQ`. If `feeCollector` is equal to `address(0)` in `LimitOrderSwap` fees may be swept from it.

```
contracts/LimitOrderSwap.sol:
  31:     constructor(
  32:         address _owner,
  33:         address _uniswapPermit2,
  34:         address _allowanceTarget,
  35:         IWETH _weth,
  36:         address payable _feeCollector
  37:     ) Ownable(_owner) TokenCollector(_uniswapPermit2, _allowanceTarget)
{
  38:         weth = _weth;
  39:         feeCollector = _feeCollector;
  40:     }
```

```
  167:         order.makerToken.transferTo(payable(takerParams.recipient),
makerSpendingAmount - fee);

contracts/GenericSwap.sol:
  79:         _outputToken.transferTo(_swapData.recipient, returnAmount);

contracts/RFQ.sol:
  32:     constructor(
  33:         address _owner,
  34:         address _uniswapPermit2,
  35:         address _allowanceTarget,
  36:         IWETH _weth,
  37:         address payable _feeCollector
  38:     ) Ownable(_owner) TokenCollector(_uniswapPermit2, _allowanceTarget)
{
  39:         weth = _weth;
  40:         feeCollector = _feeCollector;
  41:     }
```

Remediation:

Consider checking that the input address is not zero to prevent users from losing their funds.

5.8. Outdated OpenZeppelin library

Risk Level: Low

Status: Fixed in the commit 4ba0b0.

Description:

There is an outdated `OpenZeppelin` library in the dependencies (version 4.8.1). This library contains several known vulnerabilities that may have a security impact when new features are added to the project.

Remediation:

Consider updating the dependencies on time to avoid the risk of known vulnerabilities.

References:

- https://github.com/OpenZeppelin/openzeppelin-contracts/tree/0457042d93d9dfd760dbaa06a4d2f1216fdbe297
- https://github.com/OpenZeppelin/openzeppelin-contracts/security

5.9. Unnecessary variable casting

Risk Level: Info

Status: Fixed in the commit 7a0201.

Contracts:

- contracts/CoordinatedTaker.sol

Description:

There is a casting to `uint64` of the `block.timestamp` variable in `CoordinatedTaker`.

```
contracts/CoordinatedTaker.sol:
  61:             if (crdParams.expiry < uint64(block.timestamp)) revert
ExpiredPermission();
```

Remediation:

Consider removing unnecessary casting to save gas.

5.10. Unnecessary memory argument location

Risk Level: Info

Status: Fixed in the commit 3e860e.

Contracts:

- contracts/LimitOrderSwap.sol

Location: Lines: `258`. Function: `_emitLimitOrderFilled`.

Description:

In the `_emitLimitOrderFilled()` function, `_order` argument data is defined as `LimitOrder memory` while `calldata` location should be used because data is not modified.

```
contracts/LimitOrderSwap.sol:
  257:     function _emitLimitOrderFilled(
  258:         LimitOrder memory _order,
  259:         bytes32 _orderHash,
  260:         uint256 _takerTokenSettleAmount,
  261:         uint256 _makerTokenSettleAmount,
  262:         uint256 _fee,
```

```
263:         address _recipient
264:     ) internal {
```

Remediation:

Change `LimitOrder memory` to `LimitOrder calldata` to optimize gas usage. This change is recommended when a function is not modifying the value of an argument and only reads it. Using `calldata` instead of `memory` can improve gas efficiency.

5.11. Lack of makerSettleAmount!=0 check

Risk Level: Info

Status: Fixed in the commit `7d3134`.

Contracts:

- contracts/RFQ.sol

Description:

There is no check that `makerSettleAmount` variable during the execution of the `_fillRFQ()` function is not equal to zero.

Remediation:

Consider adding a comparison that `makerSettleAmount` is not equal to zero.

5.12. Unnecessary weth variable

Risk Level: Info

Status: Fixed in the commit `956ec9`.

Contracts:

- contracts/UniAgent.sol

Description:

The `UniAgent` contract contains the `weth` public immutable variable and `_weth` constructor parameter. This variable is never used and bloats the contract size.

Remediation:

Delete the unused variable.

5.13. Unsafe usage of abi.encodeWithSelector

Risk Level: Info

Status: Fixed in the commits 12db4f and 7328da.

Contracts:

- contracts/UniAgent.sol

Description:

The call data for a call to an external contract is created using `abi.encodeWithSelector`. The usage of mismatched types for the input parameters is not guaranteed by this function.

```
contracts/UniAgent.sol:
    46:    tokens[i].call(abi.encodeWithSelector(IERC20.approve.selector,
v2Router, type(uint256).max));
  47:    tokens[i].call(abi.encodeWithSelector(IERC20.approve.selector,
v3Router, type(uint256).max));
    84:    (bool apvSuccess, bytes memory apvResult) =
inputToken.call(abi.encodeWithSelector(IERC20.approve.selector, routerAddr,
type(uint256).max));
```

Remediation:

To guarantee arguments type safety it is recommended to use `abi.encodeCall` instead of `abi.encodeWithSelector`.

5.14. Array length not cached outside of the loop

Risk Level: Info

Status: Fixed in the commit 2760a0.

Contracts:

- contracts/AllowanceTarget.sol
- contracts/SmartOrderStrategy.sol

Description:

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

There are the following instances:

- contracts/AllowanceTarget.sol:18
- contracts/SmartOrderStrategy.sol:54

Remediation:

Example of a not optimized code:

```
for (uint256 i = 0; i < tokens.length; i++) {
```

Consider saving array length before the loop:

```
uint256 l = tokens.length;
for (uint256 i = 0; i < l; i++) {
```

References:

- https://github.com/byterocket/c4-common-issues/blob/main/0-Gas-Optimizations.md/#g002—cache-array-length-outside-of-loop
- https://github.com/code-423n4/2021-11-badgerzaps-findings/issues/36

5.15. Unnecessary storage access

Risk Level: Info

Status: Fixed in the commit dfa54d.

Contracts:

- contracts/LimitOrderSwap.sol

Description:

The `feeCollector` state variable reads in the loop. This variable should not change through the iterations during the execution of the `fillLimitOrderGroup()` function so it can be cached.

```
contracts/LimitOrderSwap.sol:
  108:             order.makerToken.transferTo(feeCollector, fee);
```

Remediation:

Replace state variable reads within loops with local variable reads.

5.16. Non-optimal conditional statements

Risk Level: Info

Status: Fixed in the commit bfc933.

Contracts:

- contracts/GenericSwap.sol
- contracts/LimitOrderSwap.sol
- contracts/RFQ.sol
- contracts/UniAgent.sol

Description:

Using nested conditions is cheaper than using `&&` multiple check combinations. There are more advantages, such as easier to read code and better coverage reports.

```
contracts/GenericSwap.sol:
  67:         if (_inputToken.isETH() && msg.value !=
_swapData.takerTokenAmount) revert InvalidMsgValue();

contracts/LimitOrderSwap.sol:
   93:                 if (makingAmount == 0 && takerTokenAmounts[i] == 0)
revert ZeroTokenAmount();
  230:         if (takerSpendingAmount == 0 && makerSpendingAmount == 0)
revert ZeroTokenAmount();
  239:         if (_order.taker != address(0) && msg.sender != _order.taker)
revert InvalidTaker();

contracts/RFQ.sol:
  92:         if ((_rfqOffer.flags & FLG_ALLOW_CONTRACT_SENDER == 0) &&
(msg.sender != tx.origin)) revert ForbidContract();
  93:         if ((_rfqOffer.flags & FLG_ALLOW_PARTIAL_FILL == 0) &&
(_rfqTx.takerRequestAmount != _rfqOffer.takerTokenAmount)) revert
ForbidPartialFill();

contracts/UniAgent.sol:
  92:         if (inputToken.isETH() && msg.value != inputAmount) revert
InvalidMsgValue();
```

Remediation:

Example of a non-optimal code:

```
if (amount > 0 && from != to) {
        userInfo.updateBalances(from, to, amount, farmedPerToken());
}
```

Consider separate check:

```
if (amount > 0) {
    if(from != to) {
                userInfo.updateBalances(from, to, amount, farmedPerToken());
        }
}
```

References:

- https://github.com/code-423n4/2022-01-xdefi-findings/issues/128

5.17. String error messages used instead of custom errors

Risk Level: Info

Status: Fixed in the pull request.

Contracts:

- contracts/AllowanceTarget.sol
- contracts/SmartOrderStrategy.sol
- contracts/abstracts/Ownable.sol
- contracts/libraries/Asset.sol

Description:

The contracts make use of the `require()` to emit an error. While this is a perfectly valid way to handle errors in Solidity, it is not always the most efficient.

```
contracts/AllowanceTarget.sol:
  38:         require(authorized[msg.sender], "AllowanceTarget: not
authorized");

contracts/SmartOrderStrategy.sol:
```

```
  80:         require(_inputRatio <= Constant.BPS_MAX, "invalid BPS");

contracts/abstracts/Ownable.sol:
  14:         require(_owner != address(0), "owner should not be 0");
  19:         require(msg.sender == owner, "not owner");
  26:         require(msg.sender == nominatedOwner, "not nominated");
  37:         require(nominatedOwner == address(0), "pending nomination
exists");

contracts/libraries/Asset.sol:
  34:             require(address(this).balance >= amount, "insufficient
balance");
  36:             require(success, "unable to send ETH");
```

Remediation:

Consider using custom errors as they are more gas efficient while allowing developers to describe the error in detail using NatSpec.

References:

- https://blog.soliditylang.org/2021/04/21/custom-errors/

5.18. Non-optimal structs

Risk Level: Info

Status: Acknowledged

Contracts:

- contracts/libraries/GenericSwapData.sol

Description:

`makerTokenAmount` field from GenericSwapData is never used in any calculations or checks (except `getGSDataHash()`) in GenericSwap contract.

```
contracts/libraries/GenericSwapData.sol:
  13: struct GenericSwapData {
  14:     address payable maker;
  15:     address takerToken;
  16:     uint256 takerTokenAmount;
  17:     address makerToken;
  18:     uint256 makerTokenAmount;
```

```
  19:     uint256 minMakerTokenAmount;
  20:     uint256 expiry;
  21:     uint256 salt;
  22:     address payable recipient;
  23:     bytes strategyData;
  24: }
```

Remediation:

Consider removing `makerTokenAmount` field from the struct to save gas.

5.19. Unnecessary checked arithmetic

Risk Level: Info

Status: Fixed in the pull request.

Contracts:

- contracts/LimitOrderSwap.sol
- contracts/AllowanceTarget.sol
- contracts/SmartOrderStrategy.sol
- contracts/UniAgent.sol
- contracts/abstracts/AdminManagement.sol
- contracts/RFQ.sol

Description:

In several locations, the `unchecked` blocks are not used although the overflow/underflow is impossible:

```
contracts/LimitOrderSwap.sol:
   83:         for (uint256 i = 0; i < orders.length; ++i) {

  115:         if (wethBalance > wethToPay) {
  116:             weth.withdraw(wethBalance - wethToPay);
  117:         }

  119:         for (uint256 i = 0; i < orders.length; ++i) {

  125:         for (uint256 i = 0; i < profitTokens.length; ++i) {
```

```
contracts/AllowanceTarget.sol:
     18:         for (uint256 i = 0; i < trustedCaller.length; ++i) {

contracts/SmartOrderStrategy.sol:
     54:         for (uint256 i = 0; i < ops.length; ++i) {

contracts/UniAgent.sol:
     34:         for (uint256 i = 0; i < tokens.length; ++i) {
     43:         for (uint256 i = 0; i < tokens.length; ++i) {

contracts/abstracts/AdminManagement.sol:
     18:         for (uint256 i = 0; i < tokens.length; ++i) {
     19:         for (uint256 j = 0; j < spenders.length; ++j) {
     26:         for (uint256 i = 0; i < tokens.length; ++i) {
```

There is a check that `_rfqOffer.feeFactor` is less than `Constant.BPS_MAX` so the fee always would be less than `makerSettleAmount` and operation on the line 134 could be placed in unchecked block.

```
contracts/RFQ.sol:
     94:         if (_rfqOffer.feeFactor > Constant.BPS_MAX) revert
InvalidFeeFactor();
  133:         uint256 fee = (makerSettleAmount * _rfqOffer.feeFactor) /
Constant.BPS_MAX;
  134:         uint256 makerTokenToTaker = makerSettleAmount - fee;
```

Remediation:

Consider using the unchecked blocks to save gas.

5.20. Non-optimal order of operations

Risk Level: Info

Status: Fixed in the commit [3f405e](3f405e).

Contracts:

- contracts/CoordinatedTaker.sol

Description:

To save gas, the `submitLimitOrderFill()` function should perform an expiry check before computing the order hash. See the following code:

```
contracts/CoordinatedTaker.sol:
  57          // validate fill permission
  58          {
  59              bytes32 orderHash = getLimitOrderHash(order);
  60
  61:             if (crdParams.expiry < uint64(block.timestamp)) revert
ExpiredPermission();
```

Remediation:

Consider changing the order of operations.

5.21. Unused imports

Risk Level: Info

Status: Fixed in the commit [a390e4](a390e4).

Contracts:

- contracts/AllowanceTarget.sol
- contracts/LimitOrderSwap.sol
- contracts/SmartOrderStrategy.sol
- contracts/UniAgent.sol

Description:

```
contracts/AllowanceTarget.sol:
  10: import { Constant } from "./libraries/Constant.sol";

contracts/LimitOrderSwap.sol:
  4: import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";

contracts/SmartOrderStrategy.sol:
  10: import { IUniswapPermit2 } from "./interfaces/IUniswapPermit2.sol";

contracts/UniAgent.sol:
  7: import { EIP712 } from "./abstracts/EIP712.sol";
```

Remediation:

Consider removing unused imports.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.