Metaswap

Date	August 2020
Auditors	Steve Marx

1 Executive Summary

In August 2020, we conducted a security assessment of the *MetaSwap* contract system: a service that aims to aggregate and optimize trades for MetaMask users.

We performed this assessment between August 3rd and August 10th, 2020. The engagement was conducted primarily by Steve Marx. The total effort expended was 1 person-week.

1.1 Scope

The MetaSwap system consists of several smart contracts and a web service. Our review focused solely on the smart contracts:

File Name	SHA-1 Hash
Constants.sol	7084fb639abd81dfb3a532ee19395878c9a54fc0
IWeth.sol	f6c553a4c18b191d22b5a3aaefafc46a947b0850

File Name	SHA-1 Hash
MetaSwap.sol	6516738189f6f4b6490da347b3151d407ed2b9eb
Spender.sol	c1340aaec0be0debb664b00af727fcd2eca958d8
adapters/CommonAdapter.sol	c34588f24f6472ea8ab37eb7276d64ae29bd2612
adapters/WethAdapter.sol	bf3f0172009ab57896c6ee576116f085c1ca428f

2 Recommendations

2.1 Remove unused imports

Description

`@nomiclabs/buidler/console.sol` is imported in a few contracts that don't use its functionality.

Examples

code/contracts/adapters/WethAdapter.sol:L6

```
import "@nomiclabs/buidler/console.sol";
```

code/contracts/MetaSwap.sol:L3

```
import "@nomiclabs/buidler/console.sol";
```

Recommendation

Reducing code when possible is always a win. Consider removing these imports.

2.2 Consider using `receive()` instead of `fallback()` ✓ Fixed

Resolution

The MetaSwap team decided to intentionally use `fallback()` to cover cases where an aggregator might send some data along with ether.

Description

`Spender` uses a `fallback()` function to receive ether from trades:

code/contracts/Spender.sol:L12-L13

```
/// @dev Receives ether from swaps
fallback() external payable {}
```

If only simple transfers are expected (with no payload), `receive()` is probably the more appropriate choice.

Recommendation

Consider using `receive()` instead.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

Actors

The relevant actors are listed below with their respective abilities:

- MetaSwap: The MetaSwap team itself has limited administrative capabilities:
  - They can register new adapters.
  - They can pause the `MetaSwap` contract, halting all trading.
- Anyone: Any Ethereum address can use the MetaSwap contracts.
  - Users can execute trades using the MetaSwap contracts as a proxy.

Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- *Out of scope for this audit*, users trust the MetaSwap API to provide them with good trades. They do have the ability to inspect those trades before executing them, but this is not easy to do manually, so most users need to trust either the API or the front-end tools they're using.
- The MetaSwap team should not be able to access users' funds outside of a trade, and the specific contracts and adapters a user decides to trust should be immutable once deployed.
- It shouldn't be possible for one user to interfere with another user's transactions (except to the extent allowed by the third-party exchanges/aggregators).

4 Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Reentrancy vulnerability in `MetaSwap.swap()` Major ✓ Fixed

Resolution

This is fixed in ConsenSys/metaswap-contracts@ `8de01f6` .

Description

`MetaSwap.swap()` should have a reentrancy guard.

The adapters use this general process:

1. Collect the *from* token (or ether) from the user.
2. Execute the trade.
3. Transfer the contract's balance of tokens (*from* and *to*) and ether to the user.

If an attacker is able to reenter `swap()` before step 3, they can execute their own trade using the same tokens and get all the tokens for themselves.

This is partially mitigated by the check against `amountTo` in `CommonAdapter`, but note that the `amountTo` typically allows for slippage, so it may still leave room for an attacker to siphon off some amount while still returning the required minimum to the user.

code/contracts/adapters/CommonAdapter.sol:L57-L62

```
// Transfer remaining balance of tokenTo to sender
if (address(tokenTo) != Constants.ETH) {
    uint256 balance = tokenTo.balanceOf(address(this));
    require(balance >= amountTo, "INSUFFICIENT_AMOUNT");
    _transfer(tokenTo, balance, recipient);
} else {
```

Examples

As an example of how this could be exploited, 0x supports an "EIP1271Wallet" signature type, which invokes an external contract to check whether a trade is allowed. A malicious maker might front run the swap to reduce their inventory. This way, the taker is sending more of the taker asset than necessary to `MetaSwap`. The excess can be stolen by the maker during the EIP1271 call.

Recommendation

Use a simple reentrancy guard, such as OpenZeppelin's `ReentrancyGuard` to prevent reentrancy in `MetaSwap.swap()`. It might seem more obvious to put this check in `Spender.swap()`, but the `Spender` contract intentionally does not use any storage to avoid interference between different adapters.

4.2 A new malicious adapter can access users' tokens Medium ✓ Fixed

Resolution

This is fixed in ConsenSys/metaswap-contracts@ `8de01f6`.

Description

The purpose of the `MetaSwap` contract is to save users gas costs when dealing with a number of different aggregators. They can just `approve()` their tokens to be spent by `MetaSwap` (or in a later architecture, the `Spender` contract). They can then perform trades with all supported aggregators without having to reapprove anything.

A downside to this design is that a malicious (or buggy) adapter has access to a large collection of valuable assets. Even a user who has diligently checked all existing adapter code before interacting with `MetaSwap` runs the risk of having their funds intercepted by a new malicious adapter that's added later.

Recommendation

There are a number of designs that could be used to mitigate this type of attack. After discussion and iteration with the client team, we settled on a pattern where the `MetaSwap` contract is the only contract that receives token approval. It then moves tokens to the `Spender` contract before that contract `DELEGATECALL` s to the appropriate adapter. In this model, newly added adapters shouldn't be able to access users' funds.

4.3 Owner can front-run traders by updating adapters Medium

✓ Fixed

Resolution

This is fixed in ConsenSys/metaswap-contracts@ `8de01f6`.

Description

MetaSwap owners can front-run users to swap an adapter implementation. This could be used by a malicious or compromised owner to steal from users.

Because adapters are `DELEGATECALL` ed, they can modify storage. This means any adapter can overwrite the logic of another adapter, regardless of what policies are put in place at the contract level. Users must fully trust *every* adapter because just one malicious adapter could change the logic of all other adapters.

Recommendation

At a minimum, disallow modification of existing adapters. Instead, simply add new adapters and disable the old ones. (They should be deleted, but the aggregator IDs of deleted adapters should never be reused.)

This is, however, insufficient. A new malicious adapter could still overwrite the adapter `mapping` to modify existing adapters. To fully address this issue, the adapter registry should be in a separate contract. Through discussion and iteration with the client team, we settled on the following pattern:

1. `MetaSwap` contains the adapter registry. It calls into a new `Spender` contract.
2. The `Spender` contract has no storage at all and is just used to `DELEGATECALL` to the adapter contracts.

4.4 Simplify fee calculation in `WethAdapter` Minor ✓ Fixed

Resolution
ConsenSys/metaswap-contracts@ `93bf5c6` .

Description

`WethAdapter` does some arithmetic to keep track of how much ether is being provided as a fee versus as funds that should be transferred into WETH:

code/contracts/adapters/WethAdapter.sol:L41-L59

```
// Some aggregators require ETH fees
uint256 fee = msg.value;

if (address(tokenFrom) == Constants.ETH) {
    // If tokenFrom is ETH, msg.value = fee + amountFrom (total fee could be 0)
    require(amountFrom <= fee, "MSG_VAL_INSUFFICIENT");
    fee -= amountFrom;
    // Can't deal with ETH, convert to WETH
    IWETH weth = getWETH();
    weth.deposit{value: amountFrom}();
    _approveSpender(weth, spender, amountFrom);
} else {
    // Otherwise capture tokens from sender
    // tokenFrom.safeTransferFrom(recipient, address(this), amountFrom);
    _approveSpender(tokenFrom, spender, amountFrom);
}

// Perform the swap
aggregator.functionCallWithValue(abi.encodePacked(method, data), fee);
```

This code can be simplified by using `address(this).balance` instead.

Recommendation

Consider something like the following code instead:

```
if (address(tokenFrom) == Constants.ETH) {
    getWETH().deposit{value: amountFrom}(); // will revert if the contract has an insuffi
    _approveSpender(weth, spender, amountFrom);
} else {
    tokenFrom.safeTransferFrom(recipient, address(this), amountFrom);
    _approveSpender(tokenFrom, spender, amountFrom);
}

// Send the remaining balance as the fee.
aggregator.functionCallWithValue(abi.encodePacked(method, data), address(this).balance);
```

Aside from being a little simpler, this way of writing the code makes it obvious that the full balance is being properly consumed. Part is traded, and the rest is sent as a fee.

4.5 Consider checking adapter existence in `MetaSwap` Minor

✓ Fixed

Resolution

The MetaSwap team found that doing the check in `Spender.swap()` actually saves gas, so they're going to stick with the existing implementation.

Description

`MetaSwap` doesn't check that an adapter exists before calling into `Spender`:

code/contracts/MetaSwap.sol:L87-L100

```
function swap(
    string calldata aggregatorId,
    IERC20 tokenFrom,
    uint256 amount,
    bytes calldata data
) external payable whenNotPaused nonReentrant {
    Adapter storage adapter = adapters[aggregatorId];

    if (address(tokenFrom) != Constants.ETH) {
        tokenFrom.safeTransferFrom(msg.sender, address(spender), amount);
    }

    spender.swap{value: msg.value}(
        adapter.addr,
```

Then `Spender` performs the check and reverts if it receives `address(0)`.

code/contracts/Spender.sol:L15-L16

```
function swap(address adapter, bytes calldata data) external payable {
    require(adapter != address(0), "ADAPTER_NOT_SUPPORTED");
```

It can be difficult to decide where to put a check like this, especially when the operation spans multiple contracts. Arguments can be made for either choice (or even duplicating the check), but as a general rule it's a good idea to avoid passing invalid parameters internally. Checking for adapter existence in `MetaSwap.swap()` is a natural place to do input validation, and it means `Spender` can have a simpler model where it trusts its inputs (which always come from `MetaSwap`).

Recommendation

Drop the check from `Spender.swap()` and perform the check instead in `MetaSwap.swap()`.

5 Second Assessment

We performed a second assessment between October 3rd and October 4th, 2020. The engagement was conducted primarily by Steve Marx. The total effort expended was 2 person-days.

This second assessment covered three new features added by the MetaSwap team:

- **Support for the CHI gas token** – This allows users to offset their gas costs by burning gas tokens. These tokens can come from the user or from tokens that are owned by the `MetaSwap` contract itself.
- **Uniswap Adapter** – This adapter allows swaps to be executed via the Uniswap v2 Router directly, rather than going through some other exchange first.
- **Fee collection** – `FeeCommonAdapter` and `FeeWethAdapter` are fee-collecting versions of the original `CommonAdapter` and `WethAdapter`. They support an extra parameter `fee`, indicating the quantity of the from asset to be sent to a fee wallet.

5.1 Scope for the Second Assessment

The following files were in scope for the second assessment:

File Name	SHA-1 Hash
MetaSwap.sol	5d66ea56c131b3ad5246e9fc6c126a0b7ba497fa
adapters/FeeCommonAdapter.sol	1bb0e2b4f7fca8e0d98113cf152eeb6be4ff13c7
adapters/FeeWethAdapter.sol	f844d9e13bd2cbf52a81ae4637b35f214098f3b2
adapters/UniswapAdapter.sol	d0733f6f4567dc58d3caf4af8875e17824a97f2d

5.2 Security Specification

The security specification hasn't change much from the original assessment, so please refer to that. There are two significant changes to the security model: fee collection and gas token ownership.

In the new code, fees are collected, but these fees can be seen as *voluntary* from the perspective of the smart contracts. Users are free to pass any value for the `fee` parameter, including `0` to avoid all fees. The assumption is that most users will not bother to change the fee suggested by the MetaSwap API.

The other significant change is the introduction of the CHI gas token. In particular, the ability to use gas tokens held by the `MetaSwap` contract opens a new potential attack surface. Indeed, we found that an attacker could use contract-held tokens for other purposes.

6 Second Assessment Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Attacker can abuse gas tokens stored in `MetaSwap` Major ✓ Fixed

Resolution

This function was removed in ConsenSys/metaswap-contracts@`75c4454`.

Description

`MetaSwap.swapUsingGasToken()` allows users to make use of gas tokens held by the `MetaSwap` contract itself to reduce the gas cost of trades.

This mechanism is unsafe because any tokens held by the contract can be used by an attacker for other purposes.

Examples

If gas tokens are held by `MetaSwap`, an attacker can use them all up by performing a gas-heavy operation via a call to `swapUsingGasToken()`. For example, an attacker could create a token called EVIL and establish an ETH/EVIL pair on Uniswap. The implementation for EVIL's `transfer()` or `transferFrom()` method could do arbitrary gas-heavy operations. Finally, the attacker can invoke `swapUsingGasToken()`, using the Uniswap adapter and ETH/EVIL as the trading pair. When EVIL's transfer functions are called, they can consume a large amount of gas. When the operation is complete, `swapUsingGasTokens()` will burn as much CHI gas tokens as possible to help offset the gas use.

An attack could also be made by using an existing token that makes external calls (e.g. an ERC777 token) or a mechanism in an aggregated exchange that makes external calls (e.g. wallet signatures in 0x).

Recommendation

The simplest way to avoid this vulnerability is to never transfer CHI gas tokens to `MetaSwap` at all. An alternative would be to only allow gas tokens to be used by approved transactions from the MetaSwap API. A possible mechanism for that would be to require a signature from the MetaSwap API. If such a signature were only provided in known-good situations (which are admittedly hard to define), it wouldn't be possible for an attacker to misuse the tokens.

7 Third Assessment

We performed a third assessment between November 7th and November 10th, 2020. The engagement was conducted primarily by Steve Marx. The total expended effort was 4 person-hours.

This third assessment covered the new `FeeDistributor` contract, which divides assets among a number of recipients. It's used in the MetaSwap system to distribute fees. Each recipient has a number of "shares", and assets are divided according to each recipients portion of share ownership. Potential assets include ether and ERC20-compatible tokens.

7.1 Scope for the third assessment

The only contract in scope was the `FeeDistributor`:

Filename	SHA-1 Hash
FeeDistributor.sol	23749a338461db92a96ae87a2fd454d1aa0cbb92

7.2 Security Specification

At setup, the `FeeDistributor` is initialized with a number of recipients, each with a corresponding number of shares.

- Recipients should be able to withdraw their fair share (`<recipient's shares> / <total shares>`) of any stored asset at any time.
- No recipient should receive more than their fair share of an asset.

8 Third Assessment Recommendations

8.1 Document assumptions about ERC20 tokens

Most ERC20-compatible tokens can be used with the `FeeDistributor` contract, but it's wise to document some assumptions made by the contract:

- Token balances will not be too big (relative to the number of shares). Specifically, the total number of token units received by the contract must be able to be multiplied by the largest share amount held by a recipient.
- Token balances will not be too small (relative to share amounts). It's impossible to divide a balance of 1 among more than 1 recipient. To be safe, it would be good to make sure that no one cares about losing less than `totalShares` token units. For example, if there are 1,000,000 total shares, an asset like ether would not be a problem because 1,000,000 wei is a trivial amount.

- Token balances will not decrease without an explicit transfer. The contract makes the assumption that it can always compute the total received tokens by adding `tokenBalance(token)` and `_totalWithdrawn[token]`. This is not the case if the token balance can be manipulated externally.

8.2 Only allow full withdrawal

The current code has both `withdraw()` and `withdrawAll()`. The former allows for a partial withdrawal. Unless there's a clear use case for this, we recommend removing it. Supporting both requires a significant amount of extra code, and it seems likely that `withdraw()` will never be used.

8.3 Drop the `recipient` parameter

Everywhere in the code, the `recipient` is always `msg.sender`. The code is simpler if `msg.sender` is just used everywhere.

9 Third Assessment Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

9.1 Simplify accounting and better handle remainders Minor ✓ Fixed

Resolution

This was fixed in ConsenSys/metaswap-contracts@ `f0a62e5`. The accounting was reworked according to the recommendation here.

Description

The current code does some fairly complex and redundant calculations during withdrawal to keep track of various pieces of state. In particular, the pair of `_available[recipient][token]` and `_totalOnLastUpdate[recipient][token]` is difficult to describe and reason about.

Recommendation

For a given token and recipient, we recommend instead just tracking how much has already been withdrawn. The rest can be easily calculated:

```
function earned(IERC20 token, address recipient) public view returns (uint256) {
    uint256 totalReceived = tokenBalance(token).add(_totalWithdrawn[token]);
    return totalReceived.mul(shares[recipient]).div(totalShares);
}

function available(IERC20 token, address recipient) public view returns (uint256) {
    return earned(token, recipient).sub(_withdrawn[token][recipient]);
}

function withdraw(IERC20[] calldata tokens) external {
    for (uint256 i = 0; i < tokens.length; i++) {
        IERC20 token = tokens[i];
        uint256 amount = available(token, msg.sender);

        _withdrawn[token][msg.sender] += amount;
        _totalWithdrawn[token] += amount;
        _transfer(token, msg.sender, amount);
    }
    emit Withdrawal(tokens, msg.sender);
}
```

This code is easier to reason about:

- It's easy to see that `withdrawn[token][msg.sender]` is correct because it's only increased when there's a corresponding transfer.
- It's easy to see that `_totalWithdrawn[token]` is correct for the same reason.

- It's easy to see that `earned()` is correct under standard assumptions about ERC20 balances.
- It's easy to see that `available()` is correct, as it's just the earned amount less the already-withdrawn amount.
- Remainders are better handled. If 1 token unit is available and you own half the shares, nothing happens on withdrawal, and if there are later 2 token units available, you can withdraw 1. (Under the previous code, if you tried to withdraw when 1 token unit was available, you would be unable to withdraw when 2 were available.)

Appendix 1 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

AUDITS

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

CAREERS

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

e-mail address