Smart Contract Audit

Date	August 2021

METASWAP

We conducted a security evaluation of MetaSwap's contract system in August 2020. This service aims to optimize and aggregate trades for MetaMask users.

1 | EXECUTIVE SUMMARY

We conducted a security evaluation of MetaSwap's contract system in August 2020. This service aims to optimize and aggregate trades for MetaMask users.

This assessment was performed between August 3rd to August 10th 2020. Steve Marx was the principal conductor of the engagement. The effort required was only one person per week.

1.1 | Scope

MetaSwap is a combination of smart contracts and web services. This review was focused on smart contracts.

Repository	SHA-1 Hash
Constants.sol	7084fb639abd81dfb3a532ee19395878c9a54fc0
IWeth.sol	f6c553a4c18b191d22b5a3aaefafc46a947b0850
MetaSwap.sol	6516738189f6f4b6490da347b3151d407ed2b9eb
Spender.sol	c1340aaec0be0debb664b00af727fcd2eca958d8
adapters/CommonAdapter.sol	c34588f24f6472ea8ab37eb7276d64ae29bd2612
adapters/WethAdapter.sol	bf3f0172009ab57896c6ee576116f085c1ca428f

2 | RECOMMENDATIONS

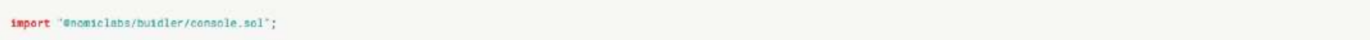
2.1 | Remove unused imports

Description

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

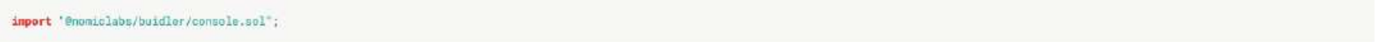
Examples

code/contracts/adapters/WethAdapter.sol:L6



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

code/contracts/MetaSwap.sol:L3



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

Recommendation

It is always a win to reduce code whenever possible. These imports should be removed.

2.2 Use receive() instead fallback()

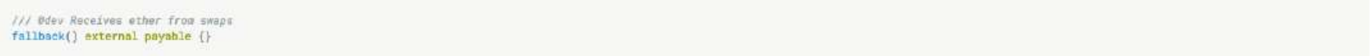
Fixed

MetaSwap's team chose to use fallback() in order to handle cases in which an aggregator might send data with ether.

Description

Spender uses fallback() to receive ether from trades:

code/contracts/Spender.sol:L12-L13



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

Receive() is the best choice if you only need to send simple funds (without paying any load).

Recommendation

Consider using receive() instead.

3 | SECURITY SPECIFICATION

This section describes the security implications of the system being audited. This section is not intended to replace documentation. This section identifies security properties that have been validated by the audit team.

Actors

Below are the relevant actors and their abilities:

MetaSwap: MetaSwap's administrative capabilities are limited:

| They can register new adapters.
| They can pause MetaSwap's contract and stop all trading.

Anyone can use MetaSwap contracts: Every Ethereum address is eligible.

| MetaSwap contracts can be used as a proxy to execute trades.

Trust Model

It is important to establish trust between actors in any smart contract system. The following trust model was created for this audit:

| Users trust the MetaSwap API for good trades, but this is beyond our scope. Although they can inspect trades before execution, this is difficult to do manually. Most users must trust the API or their front-end tools.
| MetaSwap should not have access to users' funds other than during a trade. Users should also be able to change the contracts and adapters they trust 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 is assigned a severity:

- **Minor** problems are subjective. These are usually suggestions about best practices or readability. These issues should be addressed by code maintainers.

- **Medium** issues are objective, but they are not security vulnerabilities. These issues should be addressed, unless there are compelling reasons not to.

- Security vulnerabilities are critical issues that can't be exploited directly or require special conditions to be exploited. All of these **Major** problems should be addressed.

- Security vulnerabilities that could be exploited to cause **Critical** issues need to be addressed.

4.1 | Reentrancy vulnerability in MetaSwap.swap() Major Fixed

This is (xed in ConsenSys/metaswap-contracts@8de01f6.an aggregator might send some data along with ether.

Description

MetaSwap.swap() must have a reentrancy protection.

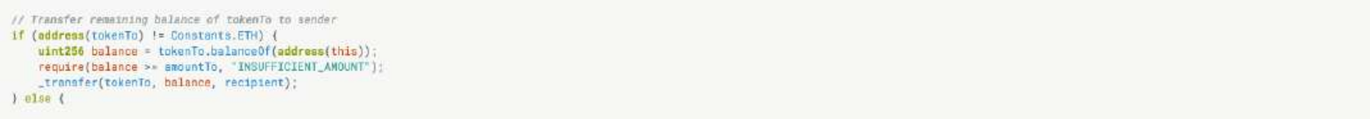
This is how adapters work:

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

An attacker can reenter swap() prior to step 3. This allows them to execute their own trade with the same tokens and obtain all tokens.

This can be partially mitigated by CommonAdapter's check against amountTo. However, it is possible for an attacker slippage so there may still be room for them to siphon off some money while still returning the minimum amount 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

This could be exploited in a number of ways. 0x supports the signature type "EIP1271Wallet", which invokes an external contract that checks whether a trade has been allowed. To reduce their inventories, a malicious maker could front-run the swap. The taker sends more taker assets to MetaSwap than is necessary. The maker can take the excess during the EIP1271 Call.

Recommendation

To prevent MetaSwap.swap() reentrancy, use a simple reentrancy protection such as OpenZeppelin's ReentrancyGuard. Although it might be easier to place this check in Spender.swap() than it is, the Spender contract deliberately does not store any data to prevent interference between adapters.

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

This is (xed in ConsenSys/metaswap-contracts@8de01f6.

Description

MetaSwap contracts are designed to reduce gas consumption when users deal with multiple aggregators. They can approve() the spending of their tokens by MetaSwap or in a later architecture the Spender contract. They can then trade with all supported aggregaters without needing to reapprove.

This design has a downside: a malicious or buggy adapter can have access to valuable assets. Even users who have carefully reviewed all adapter codes before using MetaSwap run the risk of having their money stolen by a malicious adapter.

Recommendation

This type of attack can be mitigated by a variety of design options. After much discussion with the client team, we agreed on a pattern in which the MetaSwap contract is only contract that gets token approval. The Spender contract is then moved to and DELEGATECALLS to the appropriate adapter. This model doesn't allow new adapters to gain access to users' funds.

4.3 | Owner can front-run traders by updating adapters Medium Fixed

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

Description

MetaSwap owners have the ability to front-run users in order to swap their adapter implementation. This could be used to steal user data by malicious or compromised owners.

Adapters can alter storage because they are DELEGATECALLed. Any adapter can modify the logic of any other adapter regardless of the policies that are in place at the contract-level. Because one malicious adapter can change the logic of all adapters, users must trust each adapter.

Recommendation

Allow modification of existing adapters to be disallowed at a minimum. Instead, add new adapters to replace the ones that are already disabled. They should be deleted but the aggregator IDs should not be used again.

However, this is not enough. An attacker could still modify adapters by overwriting the adapter mapping. This issue should be addressed in detail by separating the contract that contains the adapter registry. We came up with this pattern after much discussion and iteration.

1. MetaSwap includes the adapter registry. It allows you to enter a new Spender contract.
2. The Spender contract does not have any storage and can only be used to DELEGATE to the adapter agreements.

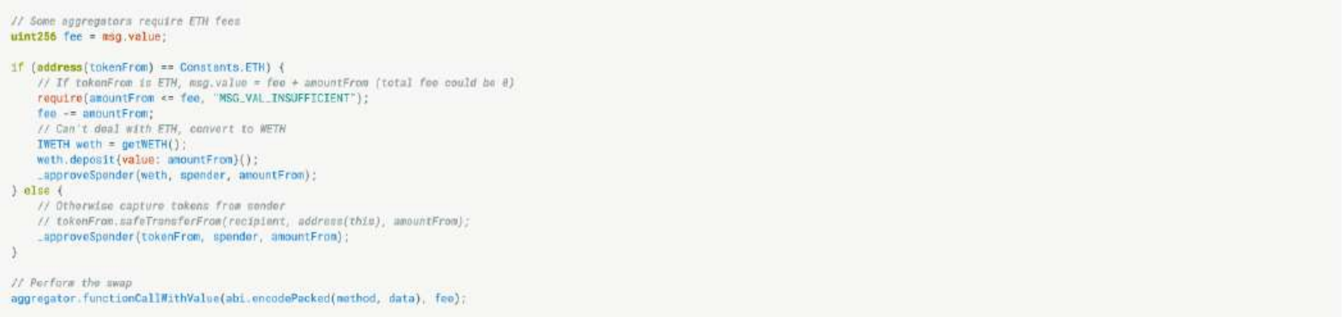
4.4 | Simplify fee calculation in WethAdapter Minor Fixed

ConsenSys/metaswap-contracts@93bf5c6.

Description

WethAdapter performs some math to track how much ether is being offered as a fee and which funds should be transferred into WETH.

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

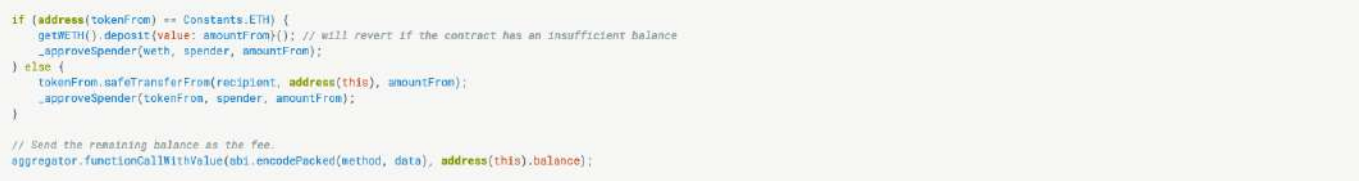


```
// Some aggregators require ETH fees
uint256 fee = msg.value;

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 insufficient balance
    _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.

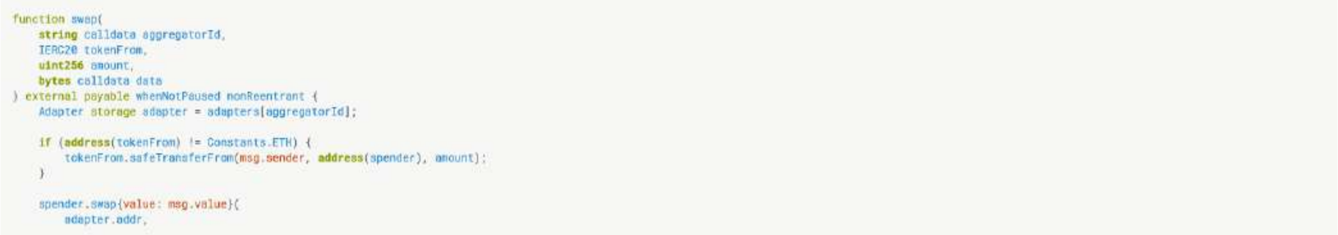
4.5 | Consider checking adapter existence in MetaSwap Minor Fixed

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:

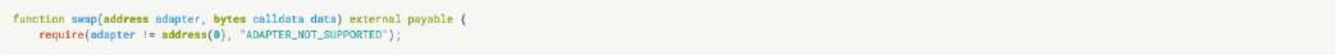
code/contracts/MetaSwap.sol:L87-L100



```
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

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 WethA dapter. 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

These files were included in the scope of the second assessment:

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

5.2 | Security Specification

Security specifications have not changed much since the initial assessment. Please refer to this. Two significant changes have been made to the security model: gas token ownership and fee collection.

The new code collects fees, but they can be considered voluntary by smart contracts. To avoid any fees, users can pass any value as a fee parameter. It is assumed that most users won't bother changing the MetaSwap API fee.

Another significant change is the introduction the CHI gas token. The MetaSwap contract allows you to use gas tokens, opening up a new attack surface. We found that attackers could also use tokens held by contracts for other purposes.

6 | SECOND ASSESSMENT ISSUES

Each issue is assigned a severity:

- **Minor** problems are subjective. These are usually suggestions about best practices or readability. These issues should be addressed by code maintainers.
- **Medium** issues are objective, but they are not security vulnerabilities. These issues should be addressed, unless there are compelling reasons not to.
- Security vulnerabilities are critical issues that can't be directly exploited or may need to be accessed under certain conditions. All of these **Major** problems should be addressed.
- Security vulnerabilities that could be exploited to cause **Critical** issues need to be addressed.

6.1 | Attacker can abuse gas tokens stored in MetaSwap Major Fixed

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 is dangerous because attackers can use any tokens in the contract for other purposes.

Examples

MetaSwap can hold gas tokens and an attacker could use them all by calling swapUsingGasToken(). An attacker could, for example, create a token called EVIL to establish an ETH/EVIL pairing on Uniswap. Implementation for EVIL's transfer() and transferFrom() methods could perform arbitrary gas-heavy operation. The attacker can also invoke swapUsingGasToken() by using the Uniswap adapter with ETH/EVIL as the trading partner. They can use a lot of gas when EVIL's transfer function is called. To offset the gas consumption, swapUsingGasTokens() will make as many CHI gas tokens possible after the operation is completed.

A token that makes external calls (e.g., ERC777) could be used to attack. An ERC777 token or a mechanism in an exchange that makes external calls could be used to attack. Signatures for wallets in 0x.

Recommendation

This vulnerability can be avoided by not transferring CHI gas tokens from MetaSwap. Another option is to allow gas tokens only to be used for approved transactions using the MetaSwap API. One way to do that is to request a signature from MetaSwap API. An attacker could not misuse tokens if such a signature was only available in certain good situations, which are admittedly difficult to identify.

7 | THIRD ASSESSMENT

Between November 7th and 11th 2020, we performed a third assessment. Steve Marx was the principal conductor of the engagement. The effort total was 4 hours.

The third assessment included the FeeDistributor contract. This divides assets between a variety of recipients. It is used in MetaSwap to distribute fees. Each recipient is given a number "shares", which are then divided according to the share of each recipient. Potential assets include ERC20-compatible tokens and ether.

7.1 | Scope for the third assessment

FeeDistributor was the only contract that was within its scope.

Repository	SHA-1 Hash
FeeDistributor.sol	23749a338461db92a96ae87a2fd454d1aa0cbb92

7.2 | Security Specification

The FeeDistributor's initialization includes a list of recipients and corresponding shares.

| Recipients should have the ability to withhold their fair share of the funds.http://www.amazon.com/Any stored asset can be accessed at any time.
| No recipient should be given more than their fair share.

8 | THIRD ASSESSMENT RECOMMENDATIONS

8.1 | Document assumptions about ERC20 tokens

The FeeDistributor contract can be used for most ERC20-compliant tokens. However, it is wise to document certain assumptions made by the contract.

| The token balances are not too large in relation to the shares. Particularly, the token balance must be able multiply the amount of shares held by the recipient by the total token units.

| Token balances are not too small in relation to share amounts. It is impossible to split a balance of 1 between more than one recipient. It is best to ensure that everyone doesn't lose less tokens than totalShares tokens in order to be safe. An asset like ether, for example, would not pose a problem if 1,000,000 shares are held. 1,000,000 wei is an insignificant amount.

| Without an explicit transfer, token balances will not fall. The contract makes the assumption that it can always compute the total received tokens by adding tokenBalance(token) and _totalWithdrawn[token]. If the token balance is externally manipulable, this assumption does not hold.

8.2 | Only allow full withdrawal

Both withdraw() and withdrawalAll() are available in the current code. The former allows partial withdrawals. We recommend that you remove the withdraw() function unless there is a clear reason to do so. Both require a lot of code and withdrawal() is unlikely to be used.

8.3 | Drop the recipient parameter

The recipient is always msg.sender everywhere in the code. It is easier to use msg.sender everywhere.

9 THIRD ASSESSMENT ISSUES

Each issue is assigned a severity:

- Minor problems are subjective. These are usually suggestions about best practices or readability. These issues should be addressed by code maintainers.

- Medium issues are objective, but they are not security vulnerabilities. These issues should be addressed, unless there are compelling reasons not to.

- Security vulnerabilities are critical issues that can't be exploited directly or require special conditions to be exploited. All of these Major problems should be addressed.

- Security vulnerabilities that could be exploited to cause Critical issues need to be addressed.

9.1 Make accounting simpler and manage remainders better Minor Fixed

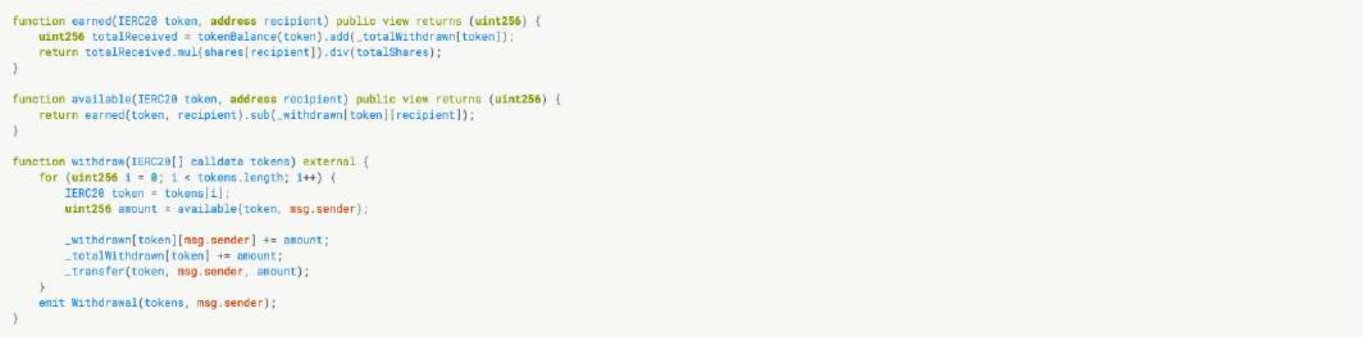
This was (xed in ConsenSys/metaswap-contracts@f0a62e5. This recommendation was followed and the accounting was revised.

Description

To keep track of different pieces of state, the current code performs some very complex and redundant calculations during withdrawal. In particular, the pair of _available[recipient][token] and _totalOnLastUpdate[recipient][token] is difficult to describe and reason about.

Recommendation

We recommend that you track how much money has been taken out for a particular token and recipient. You can easily calculate the rest:



```
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 easy to understand:

| 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 is easy to see that earned() works according to the standard assumptions regarding ERC20 balances.

| It is easy to see that the available() function is correct. It is the earned amount less the amount already withdrawn.

| Remainders can be handled better You can withdraw 1 token unit if you have half of the shares and 1 token unit is not available. If there are 2 token units later, you can withdraw 1. (Under the old code, if 1 token unit was not available and you attempted to withdraw, you could only withdraw 2 units.

APPENDIX 1 - DISCLOSURE

ConsenSys Dialigence ("CD") receives compensation from clients (the Clients) for performing the analysis in these reports (the Reports). Reports can be distributed via ConsenSys publications or other distributions.

Reports are not intended to endorse or indict any project or team. They also do not guarantee security for any project. This Report doesn't consider or have any bearing on the economics of token sales, token sales, or any other product, services, or assets. Cryptographic tokens, which are emerging technologies, carry high technical risks and uncertainties. Any Report does not provide any representation or warranty to Third-Parties in any way. This includes regarding the bug-free nature of code, any business model or proprietors, or the legal compliance of such businesses. The Reports should not be relied upon by any third party, even if it is used to make decisions about buying or selling tokens, products, services, or assets. This Report is not intended as investment advice and should not be relied on as such. It is also not intended to be used as such as an endorsement of this project, team or project. Furthermore, it does not guarantee absolute security. CD is not obligated to any Third-Party for publishing these Reports.

PURPOSE OF THE REPORTS Reports and analysis contained therein are only for Clients. They can be published with their permission. Our review will only cover Solidity code. We are limited to reviewing the Solidity codes we have identified as being included in this report. Solidity language is still under development. It may have flaws and risks. The review does NOT cover the compiler layer or any other areas that could pose security risks beyond Solidity. Cryptographic tokens, which are emerging technologies, carry high technical risk and uncertainty.

CD makes the Reports accessible to clients and other parties (i.e. "third parties") via its website. CD hopes that the public availability of these analyses will help the blockchain ecosystem to develop best practices in this rapidly changing area of innovation.

LINKS TO OTHER WEBSITES FROM THIS WEB site You can, via hypertext or other computer hyperlinks, gain access web sites owned by people other than ConsenSys. These hyperlinks are provided only for your convenience and are not intended to replace the owners of these web sites. ConsenSysys or CD are not responsible or liable for any content or operation of these Web sites. You also agree that ConsenSysys or CD will not be liable for any third-party Web site. Except as stated below, linking from this Web Site to another site does not mean or imply that ConsenSysys or CD endorses that Web site's content or its operator. It is up to you to decide whether or not you can use content from any other websites to which the Reports link. ConsenSys or CD will not be responsible for third-party software used on the Web Site. They also assume no responsibility and will have no liability to any person or entity as regards the accuracy and completeness of any result generated by such software.

TIMELINESS CONTENT. The Reports are current as of the Report's date. However, they can be modified at any time. ConsenSys or CD are the only sources of information, unless otherwise indicated.