USETH

Smart Contract Security Assessment

June 11, 2024

VERACITY

Disclaimer

Veracity Security ("Veracity") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocacy for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature.

The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by the Veracity team.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Veracity is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Veracity or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

1 Overview

This report has been prepared for USETH (https://useth.org) . Veracity provides an examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Name	USETH
URL	https://useth.org
Platform	Ethereum
Language	Solidity

USETH consists of 1 contracts:

USETH	ECR20 token with transfer tax, and dev fees.

1.3 Testing

Following an initial pass on all contracts, the client implemented a comprehensive suite of Hardhat unit tests, including automation for contract deployment. This has mitigated the risk of contract locking and initial mis-configurations. However it is not possible to catch all scenarios with these tests. Veracity has implemented a suite of audit tests that also exercise the primary functions of each contract to ensure that no transaction or fund locking occurs.

Tests have been implemented with the Foundry fuzz testing framework and some of the issues discovered are listed in the tables below. No further critical issues were discovered during this secondary process.

1.2 Final Contracts Assessed

Following deployment of the contracts assessed, Veracity compares the contracts that have been deployed, and wired with the contracts that have been audited to guarantee no tampering has been possible between audit report issue and project start.

This gives project owners and community members confidence that what has been deployed matches the findings and resolution status described in this document.

Github hash: 5351ba09171c1733d690f4eabb33a1ab3d30a71d

Deployment network: Ethereum

Links to verified contracts:

Name	Address	Matched
USETH	0x9326c50e4c4754943a2fcd02268066eb86d53837	**YES**

1.3 Findings Summary

Individual issues found have been categorised based on criticality as high, medium, low or informational. The client is required to respond to each issue individually, although it may be by design and therefore simply acknowledged. Additional recommendations may apply to all contracts, but are replicated for each for resolution.

For example an issue relating to centralisation of financial risk may apply to all administration functions, but will be included only once per contract. The table below shows the collected number of issues found and the resolution statuses across all contracts in the project.

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change)
• High	4	4	0	0
• Medium	2	2	0	0
• Low	4	4	0	0
• Informational	4	3	0	1
Total	14	13	0	1

1.3.1 Status Classifications

Severity	Description
• High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
• Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
• Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
• Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.
• Optimization	Suboptimal implementations that may result in additional gas consumption, unnecessary computation or avoidable inefficiencies.

1.3.2 Collected Issues and Statuses

ID	Contract	Severity	Summary	Status
01	USETH	MEDIUM	Rewards may be lost due to division before multiplication precision issues.	RESOLVED
02	USETH	INFO	The `renounced` variable is declared but not used.	RESOLVED
03	USETH	LOW	Multiplication after division in max wallet percentage check can lead to precision loss.	RESOLVED
04	USETH	MEDIUM	The `threshold` parameter in `swapExactTokensForTokensSupportingFeeOnTransferTokens` is a fixed value, which can lead to fund loss or transaction failures.	RESOLVED
05	USETH	HIGH	The deadline parameter in `swapExactTokensForTokensSupportingFeeOnTransferTokens` is set to `type(uint256).max`, effectively disabling the deadline check.	RESOLVED
06	USETH	LOW	Multiple Solidity versions specified in the code.	RESOLVED
07	USETH	LOW	SafeERC20 library not used.	RESOLVED
08	USETH	MEDIUM	Hardcoded gas price check with no function to change `targetGwei` can cause fees not to be distributed for as long as gas price remains above `targetGwei`	RESOLVED

09	USETH	**INFO**	transfer function contains almost similar code to transferFrom.	**ACKNOWLEDGED**
10	USETH	**MEDIUM**	There are multiple centralization attack vectors present in the contract.	**RESOLVED**
11	USETH	**MEDIUM**	Insufficient input validation	**RESOLVED**
12	USETH	**INFO**	The initializer function `flashInitalize` can be run twice.	**RESOLVED**
13	USETH	**LOW**	function `getAmountsOut` does not exist in interface `Univ2`.	**RESOLVED**
14	USETH	**INFO**	Libraries `Address`, `IERC20` and `IERC20Permit` and function `_min` (L188) are not used. Also solidity version is mentioned 4 times `pragma solidity ^0.8.20;`	**RESOLVED**

2. Findings

The contract assessed have been largely authored from scratch rather than using industry tested implementations for ERC20. Standard interfaces have been included inline which adds risk of errors, however code comparison shows no errors have been introduced during this process. This can result in the introduction of vulnerabilities or bugs that have not been seen or addressed in previous projects. However our team has made recommendations and several code sweeps to mitigate the effect of not using industry standard libraries. The following sections outline issues found with individual contracts.

2.1 USETH

This report has been prepared for USETH. Veracity provides an examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

2.1.1 Privileged Roles

The following functions can be called by the deployer or deployerALT of the USETH contract:

- setLPtoken
- flashInitialize
- renounceContract
- configImmuneToMaxWallet
- configImmuneToFee
- editMaxWalletPercent
- editSellFee
- editBuyFee
- editTransferFee
- setThreshold
- updateDevAddress
- setTargetGwei

2.1.2 Initial Token Allocation

On deployment 21000000 (21 Million) tokens are minted to the contract deployer. The purpose of these tokens is to provide liquidity on UniSwap. This is done with the flashInitialize function by the privileged deployer.

2.1.3 Taxes, Rules, Initial Variables.

The amount of tax on buy and sell is set in the USETH contract.

Description	Value
Total Supply	21 Million
Token Decimals	18
Sell Fee Percent	30
Buy Fee Percent	30
Max Wallet Percent	100
Slippage	300

2.1.4 USETH Issues & Recommendations

Issue Number: 1

Impact: Medium, as only a small part of funds are locked.

Likelihood: High, as it will always happen.

Severity: Medium

Summary : Rewards may be lost due to division before multiplication precision issues.
Description : In L468, the code is:

```
uint amt = (address(this).balance/10000);
```

This division operation can lead to precision loss if `address(this).balance` is not a multiple of 10000. The subsequent lines (L470-474) show a multiplication of amt after the division in L468.

```
(bool sent1,) = Dev[0].call{value: amt*9000}("");
(bool sent2,) = Dev[1].call{value: amt*250}("");
(bool sent3,) = Dev[2].call{value: amt*250}("");
(bool sent4,) = Dev[3].call{value: amt*250}("");
(bool sent5,) = Dev[4].call{value: amt*250}("");
```

Proposed Fix:

To ensure no precision is lost, the multiplication should occur before the division. By doing so, the integer division occurs as the last step, minimizing precision loss. The revised code should calculate each developer's share as a fraction of the total balance:

```
uint totalBalance = address(this).balance;
uint amt9000 = (totalBalance * 9000) / 10000;
uint amt250 = (totalBalance * 250) / 10000;

(bool sent1,) = Dev[0].call{value: amt9000}("");
(bool sent2,) = Dev[1].call{value: amt250}("");
(bool sent3,) = Dev[2].call{value: amt250}("");
(bool sent4,) = Dev[3].call{value: amt250}("");
(bool sent5,) = Dev[4].call{value: amt250}("");
```

Issue Number: 2

Severity: Informational

Summary : The `renounced` variable is declared but not used.

`Description`: L300 he variable `renounced` is declared as:

```
bool public renounced;
```

However, this variable is not used anywhere else in the code. This can lead to unnecessary use of storage, which could slightly increase gas costs. Additionally, it may cause confusion for other developers maintaining or reviewing the code.

Proposed Fix: Remove the unused renounced variable to clean up the code and reduce potential gas costs.

Issue Number: 3

Severity: Low

Summary: L512 Multiplication after division in max wallet percentage check can lead to precision loss.

Description: In L391, the code is:

```
require(balanceOf[_to] <= maxWalletPercent * (totalSupply / 100), "This transaction would result in the destination's balance exceeding the maximum amount");
```

This line checks if the `_to` address balance exceeds the maximum wallet percentage. However, the division is performed before the multiplication, which can lead to precision loss if `totalSupply` is not a multiple of 100.

Proposed Fix: To ensure no precision is lost, the multiplication should occur before the division : `require(balanceOf[_to] <= (maxWalletPercent * totalSupply) / 100, "This transaction would result in the destination's balance exceeding the maximum amount");`

Issue Number: 4

Severity: Medium

Summary: The `threshold` parameter in `swapExactTokensForTokensSupportingFeeOnTransferTokens` is a fixed value, which can lead to fund loss or transaction failures.

Description: L461 The line

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(feeQueue, threshold, order, address(proxy), type(uint256).max);
```

uses a fixed `threshold` value as the minimum amount of output tokens that must be received for the transaction not to revert. This approach has significant drawbacks:

- If the `threshold` is set too low, funds can be lost to sandwich bots, especially on the Ethereum mainnet where these attacks are common.
- If the `threshold` is set too high, the transaction might never trigger, leading to potential delays and errors like “UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT”

A fixed `threshold` does not account for market volatility and can lead to suboptimal trading outcomes.

Proposed Fix: Instead of using a fixed `threshold`, dynamically calculate the minimum output tokens using a function like `getAmountsOut` from the router, and include a slippage parameter that can be defined and adjusted. This approach ensures that swaps are executed at acceptable rates, protecting against poor execution and front-running attacks.

Introduce a configurable slippage parameter to the contract.

```
uint256 public slippage; // e.g., 100 = 1% slippage

function setSlippage(uint256 _slippage) external onlyDeployer {
    slippage = _slippage;
}
```

Then Calculate Minimum Output Using **`getAmountsOut`**: Use the Uniswap router's `getAmountsOut` function to dynamically determine the expected output and apply the slippage tolerance.

Add this :

```
function getExpectedOutput(uint256 amountIn, address[] memory path) public view returns (uint256[] memory) {
```

```
    return router.getAmountsOut(amountIn, path);
}
```

Then update this :

```
function swapTokens() external {
    uint256[] memory amountsOut = getExpectedOutput(feeQueue, order);
    uint256 expectedOutput = amountsOut[amountsOut.length - 1];
    uint256 minOutput = (expectedOutput * (10000 - slippage)) / 10000;
    router.swapExactTokensForTokensSupportingFeeOnTransferTokens(feeQueue,
minOutput, order, address(proxy), type(uint256).max);
}
```

By implementing these changes, the contract dynamically sets the `threshold` based on current market conditions, ensuring better protection against front-running attacks and more reliable execution of swaps.

Issue Number: 5

Severity: High

Summary: The deadline parameter in `swapExactTokensForTokensSupportingFeeOnTransferTokens` is set to `type(uint256).max`, effectively disabling the deadline check.

Description: L461 The line:

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(feeQueue, threshold, order, address(proxy), type(uint256).max);
```

uses `type(uint256).max` for the deadline parameter. This effectively disables the deadline check, allowing the transaction to remain in the mempool indefinitely. This can lead to :

1) **Outdated Slippage**: The transaction can be executed much later when the price conditions have drastically changed, leading to significant slippage.
2) **MEV Exploitation**: Malicious bots can exploit these pending transactions, resulting in substantial losses for users.

Proposed Fix: Use a realistic deadline based on `block.timestamp` to ensure that the swap is executed within a reasonable timeframe. This prevents transactions from being executed under outdated market conditions.

```
uint256 deadline = block.timestamp + 1 minutes; // 1 minute from the current block timestamp
```

and

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(feeQueue, threshold, order, address(proxy), deadline);
```

Issue Number: 6

Severity: Low

Summary: Multiple Solidity versions specified in the code.

Description: The codebase contains multiple Solidity version pragmas, which can lead to inconsistencies and potential compatibility issues.

```
pragma solidity ^0.8.1;

pragma solidity ^0.8.0;

pragma solidity >=0.8.0 <0.9.0;
```

Using multiple Solidity versions can cause unexpected behavior, increased maintenance complexity, and potential security vulnerabilities due to differing compiler behavior and bug fixes.

Proposed Fix: Consider using one of the latest versions like ^0.8.20.

Issue Number: 7

Severity: Low

Summary: SafeERC20 library not used.

Description: The SafeERC20 library is included in the codebase but is not utilized for handling ERC20 token transfers. SafeERC20 provides safer methods for interacting with ERC20 tokens, ensuring that all transfers, approvals, and other interactions check for success and revert on failure.

Proposed Fix: Utilize the SafeERC20 library for all ERC20 token interactions to ensure that all operations are checked for success and revert on failure. This enhances the reliability and security of token interactions.

Import the SafeERC20 library and declare its usage:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

using SafeERC20 for IERC20;
```

Use SafeERC20 methods for token transfers and approvals:

```
IERC20(token).safeTransfer(_to, _value);

IERC20(token).safeApprove(_spender, _value);

IERC20(token).safeTransferFrom(_from, _to, _value);
```

Issue Number: 8

Impact: High as it can lead to DoS.

Likelihood: Low as it requires high gas prices.

Severity: Medium

Summary: Hardcoded gas price check with no function to change `targetGwei` can cause fees not to be distributed for as long as gas price remains above `targetGwei` .

Description: L459 The code contains a hardcoded gas price check:

```
require(tx.gasprice < targetGwei * 1000000000, "gas price too high");
```

However, there is no function provided to change the `targetGwei` variable. This can lead to issues if the gas price on the Ethereum network fluctuates significantly, making it difficult to adapt to changing network conditions without modifying and redeploying the contract. This limitation becomes even more critical in the event of a network fork, where gas prices can change dramatically and unpredictably.

Proposed Fix: Add a function to allow the `targetGwei` variable to be updated by authorized users. This will provide flexibility to adjust the acceptable gas price limit based on current network conditions, including situations arising from network forks.

Add a function to update `targetGwei`:

```
function setTargetGwei(uint256 newTargetGwei) external onlyDeployALT {

    targetGwei = newTargetGwei;
}
```

Issue Number: 9

Severity: Informational

Summary: transfer function contains almost similar code to transferFrom.

Description: This is not an issue just a suggestion to reduce code size

Proposed Fix: Run transfer inside transferFrom; Change transferFrom() to this :

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {

        require(balanceOf[_from] >= _value, "Insufficient token balance.");
        if(_from != msg.sender){
            require(allowance[_from][msg.sender] >= _value, "Insufficent approval");
            allowance[_from][msg.sender] -= _value;
        }
transfer(_from, to,value);
}
```

Issue Number: 10

Impact: High as it can lead to a rug pull.

Likelihood: Low, as it requires a compromised or malicious deployer.

Severity: Medium

Summary: There are multiple centralization attack vectors present in the contract.

Description: L328 deployerALT can sweep tokens with SweepToken(). When fees are processed, they are transferred to the contract. deployerALT can remove them before they are sold and sent to fee wallets. deployerALT can also increase parameter `threshold` to a value that can cause a DoS to sendFee(). Also he can update fee wallets essentially making them redundant.

Proposed Fix: Remove the deployerALT concept from the contract and move those functions into the deployer administration function. The deployer user should be operated from a ⅔ multi-sig wallet like https://safe.global/

Issue Number: 11

Impact: High as it can result in user funds loss or system DoS.

Likelihood: Low, as it requires deployer error when configuring parameters.

Severity: Medium

Summary: Insufficient input validation

Description: Many functions lack input validation such as editSellFee/ editTransferFee / editBuyFee. A value of 100 can cause users to lose 100% of the value they wanted to sell/transfer/buy. A value above 100 can create an arbitrary amount of tokens that can be swept after. This is especially important since the deployer can be renounced and after that there is no way to re-configure.

A malicious deployed can also manipulate the fees to facilitate a rug-pull.

Proposed Fix: Add sensible lower and upper boundaries for the values. The upper boundaries for the fees should be fixed and published to the community. For example: **All fees have an upper limit of 5%**

Issue Number: 12

Severity: Informational

Summary : The initializer function `flashInitalize` can be run twice.

`Description`: `flashInitalize` can be run twice. Adding liquidity for a 2nd time with `amountADesired` and `amountBDesired` equal to 0 can be manipulated by MEV bots.

Proposed Fix: To prevent the `flashInitalize` function from being executed more than once, you can introduce a state variable to track if the initialization has already been performed and add a check within the function.

```
bool public isInitialized = false;

function flashInitalize(uint HowManyWholeTokens) onlyDeployer public payable { require(!isInitialized, "Initialization can only be performed once"); isInitialized = true;
```

Issue Number: 13

Severity: Low

Summary : function `getAmountsOut` does not exist in interface `Univ2`.

`Description`: `getExpectedOutput` uses `getAmountsOut` but that doesn't exist in the interface.

Proposed Fix: Add this function in the interface.

```
function getAmountsOut(
    uint amountIn,
    address[] memory path
) external view returns (uint[] memory amounts);
```

Issue Number: 14

Severity: Informational

Summary : Libraries `Address`, `IERC20` and `IERC20Permit` and function `_min` (L188) are not used. Also solidity version is mentioned 4 times `pragma solidity ^0.8.20;`

Proposed Fix: Remove the libraries, _min() and the extra solidity versions to reduce code size.