Woosh Deposit Vault Audit Report

Prepared by Cyfrin

Version 1.0

Lead Auditors

Hans

Assisting Auditors

0kage

September 1, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The protocol intends to allow anyone send crypto to someone who does not have a wallet. Anyone can send funds (native ETH and ERC20 tokens) to Woosh deposit vault and a receiver can claim the funds by providing a signature.

5 Executive Summary

Over the course of 2 days, the Cyfrin team conducted an audit on the Woosh Deposit Vault smart contracts provided by Woosh. In this period, a total of 10 issues were found.

DepositVault.sol

Summary

Project Name	Woosh Deposit Vault
Repository	woosh-contracts
Commit	ac3d05a3f12b...
Audit Timeline	Aug 31th - Sep 1st
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	1
Informational	3
Gas Optimizations	4
Total Issues	10

6 Findings

6.1 Medium Risk

6.1.1 Non-standard ERC20 tokens are not supported

Severity: Medium

Description: The protocol implemented a function `deposit()` to allow users to deposit.

```
DepositVault.sol
37:     function deposit(uint256 amount, address tokenAddress) public payable {
38:         require(amount > 0 || msg.value > 0, "Deposit amount must be greater than 0");
39:         if(msg.value > 0) {
40:             require(tokenAddress == address(0), "Token address must be 0x0 for ETH deposits");
41:             uint256 depositIndex = deposits.length;
42:             deposits.push(Deposit(payable(msg.sender), msg.value, tokenAddress));
43:             emit DepositMade(msg.sender, depositIndex, msg.value, tokenAddress);
44:         } else {
45:             require(tokenAddress != address(0), "Token address must not be 0x0 for token deposits");
46:             IERC20 token = IERC20(tokenAddress);
47:             token.safeTransferFrom(msg.sender, address(this), amount);
48:             uint256 depositIndex = deposits.length;
49:             deposits.push(Deposit(payable(msg.sender), amount, tokenAddress));//@audit-issue
↪  fee-on-transfer, rebalancing tokens will cause problems
50:             emit DepositMade(msg.sender, depositIndex, amount, tokenAddress);
51:
52:         }
53:     }
```

Looking at the line L49, we can see that the protocol assumes `amount` of tokens were transferred. But this does not hold true for some non-standard ERC20 tokens like fee-on-transfer tokens or rebalancing tokens. (Refer to here about the non-standard weird ERC20 tokens)

For example, if token incurs fee on transfer, the actually transferred amount will be less than the provided parameter `amount` and the `deposits` will have a wrong state value. Because the current implementation only allows full withdrawal, this means the tokens will be locked in the contract permanently.

Impact: If non-standard ERC20 tokens are used, the tokens could be locked in the contract permanently.

Recommended Mitigation:

- We recommend adding another field in the `Deposit` structure, say `balance`
- We recommend allow users to withdraw partially and decrease the `balance` field appropriately for successful withdrawals. If these changes are going to be made, we note that there are other parts that need changes. For example, the withdraw function would need to be updated so that it does not require the withdrawal amount is same to the original deposit amount.

Protocol:

Cyfrin:

6.1.2 Use call instead of transfer

Severity: Medium

Description: In both of the withdraw functions, `transfer()` is used for native ETH withdrawal. The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

Impact: The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Recommended Mitigation: Use call() instead of transfer().

Protocol: Cyfrin:

6.2 Low Risk

6.2.1 The deposit function is not following CEI pattern

Severity: Low

Description: The protocol implemented a function `deposit()` to allow users to deposit.

```
DepositVault.sol
37:     function deposit(uint256 amount, address tokenAddress) public payable {
38:         require(amount > 0 || msg.value > 0, "Deposit amount must be greater than 0");
39:         if(msg.value > 0) {
40:             require(tokenAddress == address(0), "Token address must be 0x0 for ETH deposits");
41:             uint256 depositIndex = deposits.length;
42:             deposits.push(Deposit(payable(msg.sender), msg.value, tokenAddress));
43:             emit DepositMade(msg.sender, depositIndex, msg.value, tokenAddress);
44:         } else {
45:             require(tokenAddress != address(0), "Token address must not be 0x0 for token deposits");
46:             IERC20 token = IERC20(tokenAddress);
47:             token.safeTransferFrom(msg.sender, address(this), amount);//@audit-issue against CEI
↪  pattern
48:             uint256 depositIndex = deposits.length;
49:             deposits.push(Deposit(payable(msg.sender), amount, tokenAddress));
50:             emit DepositMade(msg.sender, depositIndex, amount, tokenAddress);
51:
52:         }
53:     }
```

Looking at the line L47, we can see that the token transfer happens before updating the accounting state of the protocol against the CEI pattern. Because the protocol intends to support all ERC20 tokens, the tokens with hooks (e.g. ERC777) can be exploited for reentrancy. Although we can not verify an exploit that causes explicit loss due to this, it is still highly recommended to follow CEI pattern to prevent possible reentrancy attack.

Recommended Mitigation: Handle token transfers after updating the `deposits` state.

Protocol:

Cyfrin:

6.3 Informational Findings

6.3.1 Nonstandard usage of nonce

Severity: Informational

Description: The protocol implemented two withdraw functions `withdrawDeposit()` and `withdraw()`. While the function `withdrawDeposit()` is designed to be used by the depositor themselves, the function `withdraw()` is designed to be used by anyone who has a signature from the depositor. The function `withdraw()` has a parameter `nonce` but the usage of this param is not aligned with the general meaning of nonce.

```
DepositVault.sol
59:     function withdraw(uint256 amount, uint256 nonce, bytes memory signature, address payable
↪  recipient) public {
60:         require(nonce < deposits.length, "Invalid deposit index");
61:         Deposit storage depositToWithdraw = deposits[nonce];//@audit-info non aligned with common
↪  understanding of nonce
62:         bytes32 withdrawalHash = getWithdrawalHash(Withdrawal(amount, nonce));
63:         address signer = withdrawalHash.recover(signature);
64:         require(signer == depositToWithdraw.depositor, "Invalid signature");
65:         require(!usedWithdrawalHashes[withdrawalHash], "Withdrawal has already been executed");
66:         require(amount == depositToWithdraw.amount, "Withdrawal amount must match deposit amount");
67:
68:         usedWithdrawalHashes[withdrawalHash] = true;
69:         depositToWithdraw.amount = 0;
70:
71:         if(depositToWithdraw.tokenAddress == address(0)){
72:             recipient.transfer(amount);
73:         } else {
74:             IERC20 token = IERC20(depositToWithdraw.tokenAddress);
75:             token.safeTransfer(recipient, amount);
76:         }
77:
78:         emit WithdrawalMade(recipient, amount);
79:     }
```

In common usage, nonce is used to track the latest transaction from the EOA and generally it is increased on the user's transaction. It can be effectively used to invalidate the previous signature by the signer. But looking at the current implementation, the parameter `nonce` is merely used as an index to refer the `deposit` at a specific index.

This is a bad naming and can confuse the users.

Recommended Mitigation: If the protocol intended to provide a kind of invalidation mechanism using the nonce, there should be a separate mapping that stores the nonce for each user. The current nonce can be used to generate a signature and a depositor should be able to increase the nonce to invalidate the previous signatures. Also the nonce would need to be increased on every successful call to `withdraw()` to prevent replay attack. Please note that with this remediation, the mapping `usedWithdrawalHashes` can be removed completely because the hash will be always decided using the latest nonce and the nonce will be invalidated automatically (because it increases on successful call).

If this is not what the protocol intended, the parameter nonce can be renamed to `depositIndex` as implemented in the function `withdrawDeposit()`.

Client:

Cyfrin:

6.3.2 Unnecessary parameter amount in withdraw function

Severity: Informational

Description: The function `withdraw()` has a parameter `amount` but we don't understand the necessity of this parameter. At line L67, the amount is required to be the same to the whole deposit amount. This means the user does not have a flexibility to choose the withdraw amount, after all it means the parameter was not necessary at all.

```
DepositVault.sol
59:     function withdraw(uint256 amount, uint256 nonce, bytes memory signature, address payable
↪  recipient) public {
60:         require(nonce < deposits.length, "Invalid deposit index");
61:         Deposit storage depositToWithdraw = deposits[nonce];
62:         bytes32 withdrawalHash = getWithdrawalHash(Withdrawal(amount, nonce));
63:         address signer = withdrawalHash.recover(signature);
64:         require(signer == depositToWithdraw.depositor, "Invalid signature");
65:         require(!usedWithdrawalHashes[withdrawalHash], "Withdrawal has already been executed");
66:         require(amount == depositToWithdraw.amount, "Withdrawal amount must match deposit
↪  amount");//@audit-info only full withdrawal is allowed
67:
68:         usedWithdrawalHashes[withdrawalHash] = true;
69:         depositToWithdraw.amount = 0;
70:
71:         if(depositToWithdraw.tokenAddress == address(0)){
72:             recipient.transfer(amount);
73:         } else {
74:             IERC20 token = IERC20(depositToWithdraw.tokenAddress);
75:             token.safeTransfer(recipient, amount);
76:         }
77:
78:         emit WithdrawalMade(recipient, amount);
79:     }
```

Recommended Mitigation: If the protocol intends to only allow full withdrawal, this parameter can be removed completely (that will help save gas as well). Unnecessary parameters increase the complexity of the function and more error prone.

Client:

Cyfrin:

6.3.3 Functions not used internally could be marked external

Severity: Informational

Description: Using proper visibility modifiers is a good practice to prevent unintended access to functions. Furthermore, marking functions as `external` instead of `public` can save gas.

```
File: DepositVault.sol

37:     function deposit(uint256 amount, address tokenAddress) public payable

59:     function withdraw(uint256 amount, uint256 nonce, bytes memory signature, address payable
↪  recipient) public

81:     function withdrawDeposit(uint256 depositIndex) public
```

Recommended Mitigation: Consider change the visibility modifier to `external` for the functions that are not used internally.

Client:

Cyfrin:

6.4 Gas Optimizations

6.4.1 Use assembly to check for `address(0)`

Saves 6 gas per instance

```
File: DepositVault.sol

40:             require(tokenAddress == address(0), "Token address must be 0x0 for ETH deposits");

45:             require(tokenAddress != address(0), "Token address must not be 0x0 for token deposits");

71:         if(depositToWithdraw.tokenAddress == address(0)){

90:         if(depositToWithdraw.tokenAddress == address(0)){
```

6.4.2 Use calldata instead of memory for function arguments that do not get mutated

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

```
File: DepositVault.sol

55:     function getWithdrawalHash(Withdrawal memory withdrawal) public view returns (bytes32)

59:     function withdraw(uint256 amount, uint256 nonce, bytes memory signature, address payable
↪  recipient) public
```

6.4.3 Use Custom Errors

Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

File: DepositVault.sol

```
38:         require(amount > 0 || msg.value > 0, "Deposit amount must be greater than 0");

40:             require(tokenAddress == address(0), "Token address must be 0x0 for ETH deposits");

45:             require(tokenAddress != address(0), "Token address must not be 0x0 for token deposits");

60:         require(nonce < deposits.length, "Invalid deposit index");

64:         require(signer == depositToWithdraw.depositor, "Invalid signature");

65:         require(!usedWithdrawalHashes[withdrawalHash], "Withdrawal has already been executed");

66:         require(amount == depositToWithdraw.amount, "Withdrawal amount must match deposit amount");

82:         require(depositIndex < deposits.length, "Invalid deposit index");

84:         require(depositToWithdraw.depositor == msg.sender, "Only the depositor can withdraw their
↪  deposit");

85:         require(depositToWithdraw.amount > 0, "Deposit has already been withdrawn");
```

6.4.4 Use != 0 instead of > 0 for unsigned integer comparison

File: DepositVault.sol

```
38:         require(amount > 0 || msg.value > 0, "Deposit amount must be greater than 0");

38:         require(amount > 0 || msg.value > 0, "Deposit amount must be greater than 0");

39:         if(msg.value > 0)

85:         require(depositToWithdraw.amount > 0, "Deposit has already been withdrawn");
```