SMART CONTRACT AUDIT REPORT

for

Arche Network

Prepared By: Yiqun Chen

PeckShield

August 27, 2021

Document Properties

Client	Arche Network
Title	Smart Contract Audit Report
Target	Arche_v1.0_Eros
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 27, 2021	Xuxian Jiang	Final Release
1.0-rc1	August 20, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Arche_v1.0_Eros protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Arche Network

Arche Network is a user-defined open platform based on Ethereum, Polygon and BSC. It aims to empower traders and developers to participate in a customized asset marketplace by providing user-friendly tools and community support that is open, and assessable to all. Arche Network v1.0 contract is called Arche_v1.0_Eros, users can create a token swap with customized parameters.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Arch_v1.0_Eros

Item	Description
Target	Arche_v1.0_Eros
Website	https://arche.network
Type	Ethereum Smart Contract
Platform	ETH/BSC/Polygon Solidity
Audit Method	Whitebox
Latest Audit Report	August 27, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Archenetwork/Arche_v1.0_Eros.git (909f35e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Archenetwork/Arche_v1.0_Eros.git (da9fd06)

And here is the list of contracts that have been deployed after fixing issues reported here:

ETH/BSC/Polygon

ETH

D_Swap_Main, support users to create the order
https://etherscan.io/address/0x31a23b28d955f5ac4e7499670c25c4c13d910882#code
D_Swap_Factory, Factory lib of main contract
https://etherscan.io/address/0xa2d56664a6469d37bec838b01db695f179a85bcc#code
Trading_Charge, Lib of charging rate
https://etherscan.io/address/0xa143a2c9c28b811c12857df846b0778832eb230b#code
FFI_ERC20, Used for contract renewal
https://etherscan.io/address/0x2815d3272baE3ebde5D7c128Eea5f4A8da402783#code

BSC

D_Swap_Main, support users to create the order
https://bscscan.com/address/0xA2d56664a6469d37BEC838b01DB695f179A85bCc#contracts
D_Swap_Factory, Factory lib of main contract
https://bscscan.com/address/0x5d1cFADa5746E00FcFAf9D7fA377f7d5b7D51922#contracts
Trading_Charge, Lib of charging rate
https://bscscan.com/address/0x31A23b28D955F5ac4E7499670c25C4C13D910882#contracts
FFI_ERC20, Used for contract renewal
https://bscscan.com/address/0x2815d3272baE3ebde5D7c128Eea5f4A8da402783#contracts

Polygon

D_Swap_Main, support users to create the order
https://polygonscan.com/address/0x067dE9B39344F207349aa65F9Ccce1Bee9275290#code
D_Swap_Factory, Factory lib of main contract
https://polygonscan.com/address/0x5d1cFADa5746E00FcFAf9D7fA377f7d5b7D51922#code
Trading_Charge, Lib of charging rate
https://polygonscan.com/address/0x577caD5AE15C57a7106700AEC7eAD4d2974699e3#code
FFI_ERC20, Used for contract renewal
https://polygonscan.com/address/0x2815d3272baE3ebde5D7c128Eea5f4A8da402783#code

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Arche_v1.0_Eros` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Possible Overflow Prevention With SafeMath	Coding Practices	Fixed
PVE-003	Low	Incompatibility With Deflationary Tokens	Time and State	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	High	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Fixed
PVE-006	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
```

```
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `ERC20Interface` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address to, uint tokens)virtual public returns (bool success)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `Charging_Transfer_ERC20()` routine in the `D_Swap` contract. If `USDT` is given as `token`, the unsafe version of `ERC20Interface(token).transfer(to,exactly_amount)` (line 400) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `ERC20Interface` interface expects a return value)!

```
390     function Charging_Transfer_ERC20 (address token ,address to ,uint256 amount)private
391     {
392         (address tc_addr)= D_Swap_Main(m_DSwap_Main_Address).m_Trading_Charge_Lib();
393         (address collecter_addr)= D_Swap_Main(m_DSwap_Main_Address).
                m_Address_of_Token_Collecter();
394         uint256 exactly_amount=Trading_Charge(tc_addr).Amount(amount,to);


397         bool res=true;
398         if(exactly_amount>=1)
399         {
400             ERC20Interface(token).transfer(to,exactly_amount);
401         }


404         if(amount.sub(exactly_amount)>=1)
405         {
406            ERC20Interface(token).transfer(collecter_addr,amount.sub(exactly_amount));
407         }

409     }
```

Listing 3.2: `D_Swap::Charging_Transfer_ERC20()`

Note that other routines `Receive_Token()`, `Deposit_For_Tail()` and `Withdraw_Head()` share the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/transferFrom()`.

Status This issue has been fixed in the commit: 7c6b673.

3.2 Possible Overflow Prevention With SafeMath

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `D_Swap`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While analyzing the `D_Swap` contract, we observe it can be improved by taking advantage of improved security from `SafeMath`. In the following, we use the `Deposit_For_Tail()` as the example.

```
    function Deposit_For_Tail(uint256 amount,address referer)public
    {


        if(m_Permit_Mode==true)
        {
            require(m_Permit_List[msg.sender]==true,"NOT PERMITTED");
        }


        require (m_Entanglement==true,"NOT ACTIONABLE");
        require (m_Option_Finish_Tail==false ,"SWAP CLOSED");


        ////calculate exactly amount whitch can be swapped
            ////////////////////////////////////////////////////////////////////////
        if(m_Amount_Tail>=m_Total_Amount_Tail)revert();
        uint256 e_amount= m_Total_Amount_Tail-m_Amount_Tail;
        if(e_amount>amount)
        {
            e_amount=amount;
        }
        bool res=false;
        ////////////////////////////////////////////////////////////////////////////////

        ////Receive tokens of tail and accumulate the variable m_Amount_Tail//////
        Receive_Token(m_Token_Tail,e_amount,msg.sender);
```

```
303         m_Amount_Tail=m_Amount_Tail+e_amount;
304         m_Amount_Tail_Swapped+=e_amount;

306         ////////////////////////////////////////////////////////////////////////////////


309         ////Calculate the amount of how many tokens to be transfered//////////////
310         uint256 amount_back=e_amount*m_Total_Amount_Head/m_Total_Amount_Tail;
311         if(amount_back>=1)
312         {
313             amount_back=amount_back.sub(1);
314         }
315         m_Amount_Head_Swapped+=amount_back;

317         uint256 reward_back=m_Amount_Reward*e_amount/m_Total_Amount_Tail;
318         ...
319     }
```

Listing 3.3: `D_Swap::Deposit_For_Tail()`

Specifically, this function allows to swap `Tail` tokens to `Head` tokens. We notice that the multiplication of `e_amount * m_Total_Amount_Head` in the computation of `amount_back` (line 310) is not guarded for overflow. Other routine `Probe_Deposit_For_Tail()` shares the same issue.

Recommendation Make use of `SafeMath` in the above calculations to better mitigate possible overflows.

Status This issue has been fixed in the commit: b45ca57.

3.3 Incompatibility With Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `D_Swap_Main`, `D_Swap`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [5]

Description

In the `Arche_v1.0_Eros` protocol, the `D_Swap` contract acts as a trustless intermediary between seller and buyer. The seller `claims` the `m_Total_Amount_Head` amount of `m_Token_Head` tokens into the `D_Swap` contract for selling and the buyer `deposits` the `m_Total_Amount_Tail` amount of `m_Token_Tail` tokens into the `D_Swap` contract for buying. After `block.number` is larger than `m_Future_Block`, the user could trigger the `Claim_For_Delivery()` to deliver tokens to the corresponding participants.

For the above two user's operations, i.e., `claim` and `deposit`, the protocol provides low-level routines `Receive_Token()` to transfer assets into the vault (see the code snippet below). These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
251     function Receive_Token(address addr,uint256 value,address from) internal
252     {
253         uint256 t_balance_old = ERC20Interface(addr).balanceOf(address(this));
254         ERC20Interface(addr).transferFrom(from, address(this),value);
255         uint256 t_balance = ERC20Interface(addr).balanceOf(address(this));

257         uint256 e_amount=t_balance.sub(t_balance_old);

259         require(e_amount>=value,"?TOKEN LOST");

261     }
```

Listing 3.4: `D_Swap::Receive_Token()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer` or `transferFrom`. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above `Receive_Token()` will revert at the requirement of `require(e_amount >=value,"?TOKEN LOST")` (line 259) when received deflationary tokens. These balance inconsistencies affects protocol-wide operation and maintenance.

One mitigation is to regulate the set of ERC20 tokens that are permitted into `D_Swap` to `claiming` . However, as a plug-in component, Arche_v1.0_Eros may not have the control of the process. Instead, it can monitor the introduction of such tokens and prevent vaults from using such tokens.

Recommendation Apply necessary mitigation mechanisms to regulate non-compliant or unnecessarily-extended ERC20 tokens.

Status This issue has been confirmed. The team clarifies that deflationary tokens are NOT supported.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In the Arche_v1.0_Eros protocol, there is a privileged owner account of D_Swap_Main contract that plays a critical role in governing and regulating the system-wide operations (e.g., system parameter configuration). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the Set_Arche_Address() and Set_Arche_Amount_Per_Deal() routines in the D_Swap_Main contract. These routines allows the owner account to adjust the m_Address_of_Arche_Token and m_Arche_Amount_Per_Deal without any limitations.

```
    function Set_Arche_Address(address addr) public onlyOwner
    {
        m_Address_of_Arche_Token=addr;
    }
    function Set_Arche_Amount_Per_Deal(uint256 amount) public onlyOwner
    {
        m_Arche_Amount_Per_Deal=amount;
    }
```

Listing 3.5: D_Swap_Main::Set_Arche_Address()and D_Swap_Main::Set_Arche_Amount_Per_Deal()

Also, the owner account of D_Swap_Factory has the privilege to change the implementation of m_DSwap_Main_Address to any other address. As mentioned before, the D_Swap_Main contract plays a critical role in system parameter configuration. To elaborate, we show below the code snippet of related functions.

```
    function Set_DSwap_Main_Address(address addr) public onlyOwner
    {
        m_DSwap_Main_Address=addr;
    }
```

Listing 3.6: D_Swap_Factory::Set_DSwap_Main_Address()

We emphasize that the privilege assignments among D_Swap_Main and D_Swap_Factory are necessary and required for proper protocol operations. However, it is worrisome if the privileged owner account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The team clarifies that they will transfer the privileged owner account to a multisig contract where 5 signatures are needed out of 7 total signatures.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `D_Swap`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there are several occasions where the `checks-effects-interactions` principle is violated. Using the `D_Swap` as an example, the `Impl_Delivery()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interactions with the external contract (line 366) start before effecting the update on the internal state (line 370), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
359 function Impl_Delivery(address user) internal
360 {
```

```

    uint256 head_amount_back=m_Future_Balance_Tail[user];

    if(head_amount_back>=1)
    {
        Charging_Transfer_ERC20(m_Token_Head,user,head_amount_back);
    }
    m_Amount_Head_Deliveryed=m_Amount_Head_Deliveryed.add(head_amount_back);
    m_Total_Future_Balance_Tail=m_Total_Future_Balance_Tail.sub(head_amount_back);
    m_Future_Balance_Tail[user]=0;

    uint256 tail_amount_back=0;
    tail_amount_back=m_Future_Balance_Head;
    m_Future_Balance_Head=0;
    Charging_Transfer_ERC20(m_Token_Tail,owner,tail_amount_back);
    m_Amount_Tail_Deliveryed+=tail_amount_back;

    D_Swap_Main(m_DSwap_Main_Address).Triger_Claim_For_Delivery( address(this) , user);

}
```

Listing 3.7: `D_Swap::Impl_Delivery()`

Note that other routine `Deposit_For_Tail()` shares the same issue.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the commit: b45ca57.

3.6 Improved Precision By Multiplication-Before-Division

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Trading_Charge`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [3]

Description

The lack of `float` support in `Solidity` may introduce a subtle and troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `Amount()` (in `Trading_Charge` contract) as an example. This routine is used to calculate the exactly amount that will be transferred to the user after charing the fees.

```
3     function Amount(uint256 amount ,address to)public view returns(uint256)
4     {
5       uint256 charge=amount/1000;
6       charge=charge*3;
7       uint256 res=amount-charge;
8       return res;
9     }
```

Listing 3.8: Trading_Charge::Amount()

We notice the calculation of the `charge` (line 5 - 6) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `charge = amount*3/1000`.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the commit: 11f7a62.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Arche_v1.0_Eros protocol, which aims to empower traders and developers to participate in a customized asset marketplace by providing user-friendly tools and community support. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[6] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[12] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[13] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[14] PeckShield. PeckShield Inc. https://www.peckshield.com.

[15] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[16] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.