MDI

Smart Contract Security Audit

V1.0

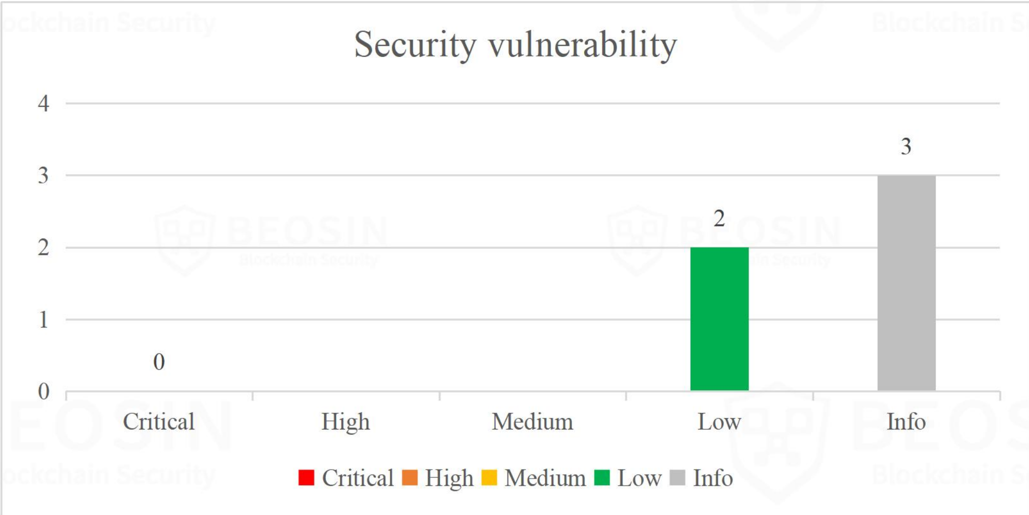
No. 202302101729

Feb 10th, 2023

Contents

Summary of Audit Results

After auditing, 2 Low-risk and 3 Info items were identified in the MDI project. Specific audit details will be presented in the **Findings** section. Users should pay attention to the following aspects when interacting with this project:



***Notes:**

- **Risk Description:**

1. The contract will charge handling fees beside the transfer value.

2. Users may pay expensive costs by fee setting from owner.

- **Project Description:**

1. **Basic Token Information**

Token name	MDI
Token symbol	MDI
Decimals	8
Total supply	1 Billion(mintable and burnable)
Token type	DIP20

2. **Business overview**

MDI is a DIP20 standard token deployed on IC. The initial total supply of MDI is 1 billion, and it has burn and mint functions. In the implementation of transfer, the contract will also charge fees from the account when fee is not 0. In addition, there is also a handling fee in the approve function, the transfer party will pay the handling fee twice in one transaction when the fee is not 0.

The controller of this canister can upgrade the deployed code, and the problems caused by the code update are not within the scope of this audit. Users participating in the project may pay attention to the version update.

1 Overview

1.1 Project Overview

Project Name	MDI
Platform	IC
Audit Scope	https://github.com/MEDICLEMDI/mdi
Commit Hash	8cea107f13b76d81e7c1780e2478f936990c3d78

1.2 Audit Overview

Audit work duration: Feb 3, 2023 – Feb 10, 2023

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team.

2 Findings

Index	Risk description	Severity level	Status
MDI-1	Unreasonable handling fee	Low	Acknowledged
MDI-2	The handling fee has no setting range	Low	Acknowledged
MDI-3	Approve function charge a handling fee	Info	Acknowledged
MDI-4	Cycles consumption problem	Info	Acknowledged
MDI-5	The centralization risk	Info	Acknowledged

Status Notes:

1. MDI-1 is not fixed and will cause additional handling fee is required when the user transfers token. It will have an impact on the actual transfer volume of users and the price calculation of DEX.

2. MDI-2 is not fixed and will cause the owner can control the amount of the fee, and user transfers token due to the high fee.

3. MDI-3 is not fixed and will not cause any issues.

4. MDI-4 is not fixed and will not cause any issues.

5. MDI-5 is not fixed and will not cause any issues.

Finding Details:

[MDI-1] Unreasonable handling fee

Severity Level	Low
Type	Business Security
Lines	token.mo #L158-173 token.mo #L174-194

Description

Under normal circumstances, the handling fee should be deducted from the transfer amount. However, in the *transfer* and *transferFrom* functions, it is charged fees beside the value of transfer. In applications such as DEX, the price of tokens will be impacted due to the inconsistency between the transfer record and the actual transfer amount.

```
158     /// Transfers value amount of tokens to Principal to.
159     public shared(msg) func transfer(to: Principal, value: Nat) : async TxReceipt {
160         if (_balanceOf(msg.caller) < value + fee) { return #Err(#InsufficientBalance); };
161         _chargeFee(msg.caller, fee);
162         _transfer(msg.caller, to, value);
163         ignore addRecord(
164             msg.caller, "transfer",
165             [
166                 ("to", #Principal(to)),
167                 ("value", #U64(u64(value))),
168                 ("fee", #U64(u64(fee)))
169             ]
170         );
171         txcounter += 1;
172         return #Ok(txcounter - 1);
173     };
```

Figure 1 Source code of *transfer* function

```
174
175     /// Transfers value amount of tokens from Principal from to Principal to.
176     public shared(msg) func transferFrom(from: Principal, to: Principal, value: Nat) : async TxReceipt {
177         if (_balanceOf(from) < value + fee) { return #Err(#InsufficientBalance); };
178         let allowed : Nat = _allowance(from, msg.caller);
179         if (allowed < value + fee) { return #Err(#InsufficientAllowance); };
180         _chargeFee(from, fee);
181         _transfer(from, to, value);
182         let allowed_new : Nat = allowed - value - fee;
183         if (allowed_new != 0) {
184             let allowance_from = Types.unwrap(allowances.get(from));
185             allowance_from.put(msg.caller, allowed_new);
186             allowances.put(from, allowance_from);
187         } else {
188             if (allowed != 0) {
189                 let allowance_from = Types.unwrap(allowances.get(from));
190                 allowance_from.delete(msg.caller);
191                 if (allowance_from.size() == 0) { allowances.delete(from); }
192                 else { allowances.put(from, allowance_from); };
193             };
194         };
```

Figure 2 Source code of *transferFrom* function

Recommendations	It is recommended that the handling fee be deducted from the amount transferred.
Status	Acknowledged.

[MDI-2] The handling fee has no setting range

Severity Level	Low
Type	Business Security
Lines	token.mo #L348-352

Description

When the owner sets the handling fee through the setFee function, the handling fee is not limited within a reasonable range. May lead to excessive loss of user transfer.

```
348     public shared(msg) func setFee(_fee: Nat) {
349         assert(msg.caller == owner_);
350         fee := _fee;
351     };
352
```

Figure 3 Source code of *setFee* function

Recommendations	It is recommended to limit the handling fee within a reasonable range.
Status	Acknowledged.

[MDI-3] *Approve* function charge a handling fee

Severity Level	Info
Type	Business Security
Lines	token.mo #L209-233
Description	In the approve function, the token transfer is not performed, but the handling fee is still deducted.

```
209         /// If this function is called again it overwrites the current allowance with value
210         public shared(msg) func approve(spender: Principal, value: Nat) : async TxReceipt {
211             if(_balanceOf(msg.caller) < fee) { return #Err(#InsufficientBalance); };
212             _chargeFee(msg.caller, fee);
213             let v = value + fee;
214             if (value == 0 and Option.isSome(allowances.get(msg.caller))) {
215                 let allowance_caller = Types.unwrap(allowances.get(msg.caller));
216                 allowance_caller.delete(spender);
217                 if (allowance_caller.size() == 0) { allowances.delete(msg.caller); }
218                 else { allowances.put(msg.caller, allowance_caller); };
219             } else if (value != 0 and Option.isNull(allowances.get(msg.caller))) {
220                 var temp = HashMap.HashMap<Principal, Nat>(1, Principal.equal, Principal.ha
221                 temp.put(spender, v);
222                 allowances.put(msg.caller, temp);
223             } else if (value != 0 and Option.isSome(allowances.get(msg.caller))) {
224                 let allowance_caller = Types.unwrap(allowances.get(msg.caller));
225                 allowance_caller.put(spender, v);
226                 allowances.put(msg.caller, allowance_caller);
227             };
228             ignore addRecord(
229                 msg.caller, "approve",
230                 [
231                     ("to", #Principal(spender)),
232                     ("value", #U64(u64(value))),
233                     ("fee", #U64(u64(fee)))
```

Figure 4 Source code of *approve* function

Recommendations	It is recommended to delete the fee collection in the *approve* function.
Status	Acknowledged.

[MDI-4] Cycles consumption problem

Severity Level	Info
Type	Business Security
Lines	token.mo #426-447

Description

When storing and restoring balances and allowances data in the *preupgrade* and *postupgrade* functions, the out of cycles problem may be caused by too long looping operation.

```
426     system func preupgrade() {
427         balanceEntries := Iter.toArray(balances.entries());
428         var size : Nat = allowances.size();
429         var temp : [var (Principal, [(Principal, Nat)])] = Array.init<(Principal, [(Principal, Nat)])>(size, (own
430         size := 0;
431         for ((k, v) in allowances.entries()) {
432             temp[size] := (k, Iter.toArray(v.entries()));
433             size += 1;
434         };
435         allowanceEntries := Array.freeze(temp);
436     };
437
438     system func postupgrade() {
439         balances := HashMap.fromIter<Principal, Nat>(balanceEntries.vals(), 1, Principal.equal, Principal.hash);
440         balanceEntries := [];
441         for ((k, v) in allowanceEntries.vals()) {
442             let allowed_temp = HashMap.fromIter<Principal, Nat>(v.vals(), 1, Principal.equal, Principal.hash);
443             allowances.put(k, allowed_temp);
444         };
445         allowanceEntries := [];
446     };
447 };
448
```

Figure 5 Source code of *preupgrade* and *postupgrade* functions

Recommendations	It is recommended to limit the length of loop.
Status	Acknowledged.

[MDI-5] The centralization risk

Severity Level	Info
Type	Business Security
Lines	token.mo #L75
Description	When the token is deployed, the total amount of tokens will be stored in the owner account. Centralization risk with token distribution.

```
72    private stable var allowanceEntries : [(Principal, [(Principal, Nat)])] =
73    private var balances = HashMap.HashMap<Principal, Nat>(1, Principal.equal,
74    private var allowances =  HashMap.HashMap<Principal, HashMap.HashMap<Princ
75    balances.put(owner_, totalSupply_);
76    private stable let genesis : TxRecord  {
77        caller = ?owner_;
78        op = #mint;
79        index = 0;
80        from = blackhole;
81        to = owner_;
82        amount = totalSupply_;
83        fee = 0;
84        timestamp = Time.now();
85        status = #succeeded;
86    };
```

Figure 6 Source code of balances

Recommendations	It is recommended to use time lock function.
Status	Acknowledged.

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Impact / Likelihood	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Redundant Code
		require/assert Usage
		Cycles Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		Returned Value Security
		Rollback Risk
		Replay Attack
		Overriding Variables
		Call Canister controllable
		Canister upgrade risk
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com