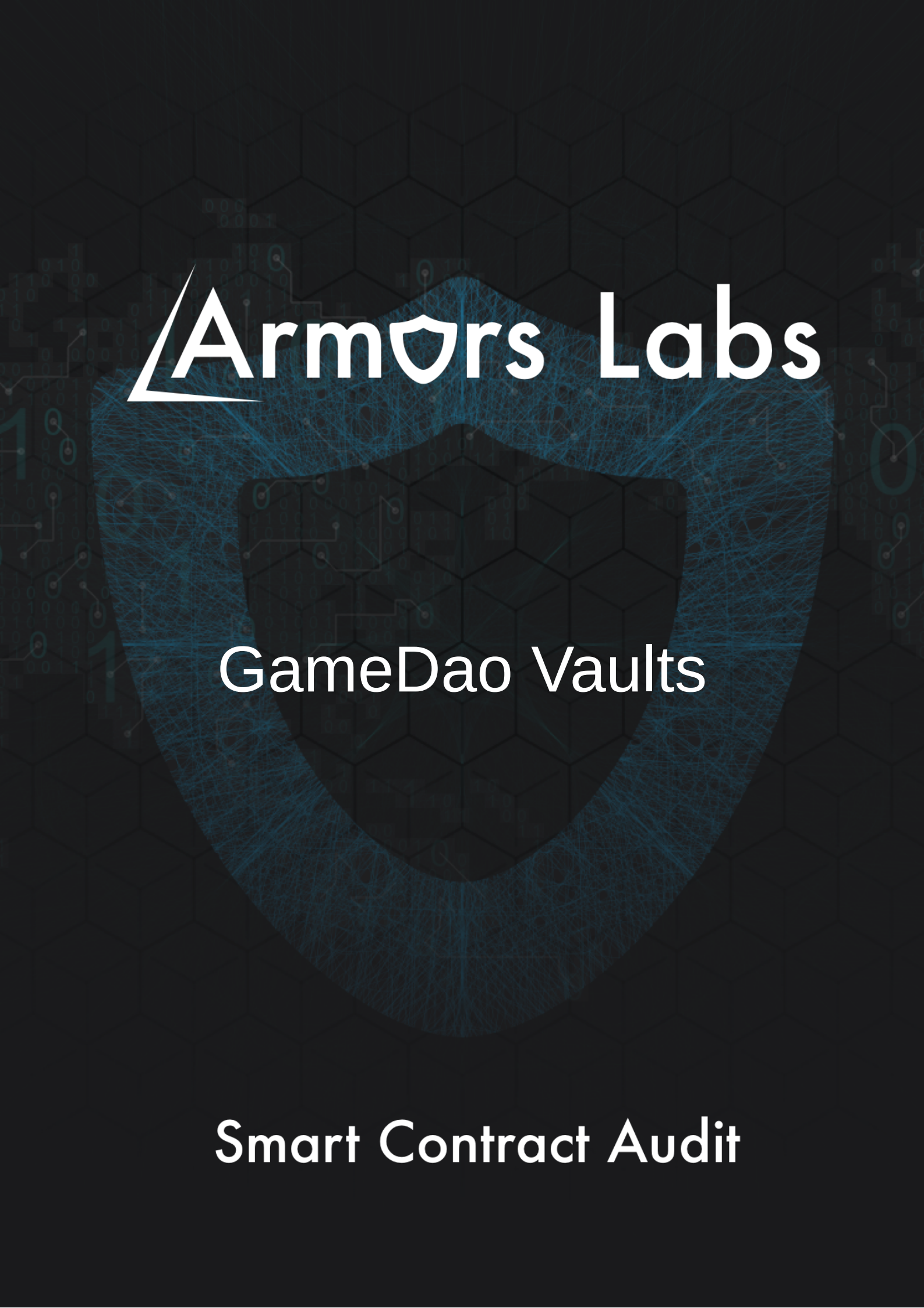

Armors Labs
GameDao Vaults
Smart Contract Audit

GameDao Vaults Audit Summary

Project name : GameDao Vaults Contract

Project address: None

Code URL : None

Projct target : GameDao Vaults Contract Audit

Test result : PASSED

Audit Info

Audit NO : 0X202103120006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

GameDao Vaults Audit

The GameDao Vaults team asked us to review and audit their GameDao Vaults contract on hecochain and ethereum. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
GameDao Vaults Audit	Rock ,Hosea, Rushairer	1.0.0	2021-03-12

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the GameDao Vaults contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 18 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused.)

Audited target file

file	md5
Controller.sol	414d56ca00c239ba00fa92614e53f23b
Gdb.sol	b5897f9b470b47564ab2457a0daca146
base/LPTokenWrapper.sol	a3d341ad0d1252617ca40c30dc868538
distribution/LpPool.sol	0ccbd0a24c786f1a6fc8f32e52c5dd5e
distribution/SinglePool.sol	6e6423062401e2be677abcbcd06a185c
distributor/InitialDistributor.sol	f0efcd7f857b2f0d328e618c26f89976
owner/Operator.sol	8cd1fb45fc4b736e1d1a06fd653c3585
strategy/mdxusdt/StrategyLendHubForMdxusdt.sol	a05de2e9f76ba0a0e824b3605a68c3ec
strategy/mdxusdt/mdxusdtVault.sol	38a341d2eb6b6ec41f614bc3be2979a8
strategy/usdt/StrategyLendHubForUsdt.sol	3e45086be809a6977568b102e72fc90f
strategy/usdt/usdtVault.sol	d2a7fec5f64a639ebf9df0c052f4b942
interface/CETH.sol	0a37a3fe6ec64c19828819f6d8f4fc7e
interface/CToken.sol	ac6e26bb2184bb7f367efc00590f6251
interface/IController.sol	d1bafd56853fd7c662824f9515e3939c
interface/IConverter.sol	c83d0dcdf1f25459096d048a7379cee4
interface/IDistributor.sol	231be56ddbc16a340bfe77bca0223d05
interface/IRewardDistributionRecipient.sol	a1a93cda9a8b648a0adb5cb154748fbe
interface/IStrategy.sol	fd28c0d33b5015b0eb9571fb864c7f5f
interface/IUni.sol	90a647a0b781ecb02c90883b7bda641c
interface/IUnitroller.sol	2c5ebad4ac8bfc90ff89fe159b72f209
interface/IWETH.sol	4e01700cc4bbce14617f29b9b60acff1

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Ether	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe

Contract file

```
.
├── Controller.sol
├── Gdb.sol
├── Migrations.sol
├── base
│   └── LPTokenWrapper.sol
├── distribution
│   ├── LpPool.sol
│   └── SinglePool.sol
├── distributor
│   └── InitialDistributor.sol
├── interface
│   ├── CETH.sol
│   ├── CToken.sol
│   ├── IController.sol
│   ├── IConverter.sol
│   ├── IDistributor.sol
│   ├── IRewardDistributionRecipient.sol
│   ├── IStrategy.sol
│   ├── IUni.sol
│   ├── IUnitroller.sol
│   └── IWETH.sol
├── owner
│   └── Operator.sol
├── strategy
│   ├── mdxusdt
│   │   ├── StrategyLendHubForMdxusdt.sol
```

```
│   │   └── mdxusdtVault.sol
│   └── usdt
│       ├── StrategyLendHubForUsdt.sol
│       └── usdtVault.sol
└── test
    ├── MockToken.sol
    └── MockWETH.sol
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle ether, and as such often send ether to various external user addresses. The operation of calling external contracts, or sending ether to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Unexpected Ether

- **Description:**
  Typically when ether is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where ether can exist in a contract without having executed any code. Contracts which rely on code execution for every ether sent to the contract can be vulnerable to attacks where ether is forcibly sent to a contract.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whether a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending ether to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap ether in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that

contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

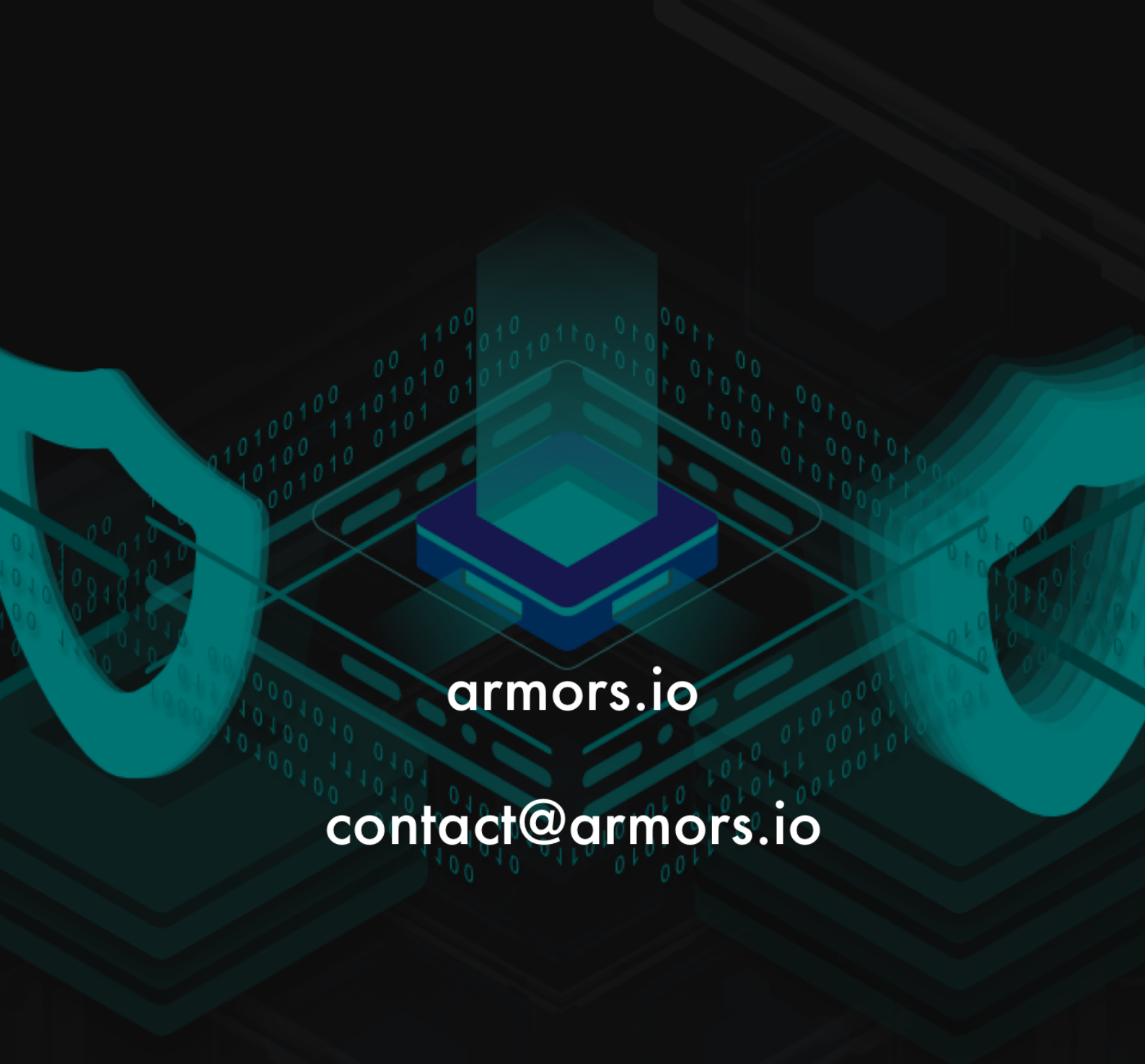
- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.



armors.io

contact@armors.io