JANUARY 30, 2021

CONFIDENTIAL

Smart Contract Audit Report

This document details the process and results of a Smart contract Audit performed by CyStack on behalf of VNDC between January 01, 2021 and January 30, 2021

PREPARED FOR VNDC HOLDING PTE LTD

CyStack

Contents

Confidentiality Statement 3

Disclaimer 3

Contact Information 3

Assessment Overview 4

- **About VIDB** 4
- **About CyStack** 5
- **Methodology** 5

Assessment Overview 9

- **State of Security** 9
- **Recommendations** 10

Findings 11

- **Overview** 11
- **Details** 17

Conclusion 24

Appendix 25

- **Basic Coding Bugs** 25
- **Semantic Consistency Checks** 27
- **Additional Recommendations** 28

Confidentiality Statement

This document is the exclusive property of **VNDC** and **Viet Nam CyStack Joint Stock Company (CyStack)**. This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires the consent of both **VNDC** and **CyStack**.

CyStack may share this document with auditors under non-disclosure agreements to demonstrate audit requirement compliance.

Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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.

Contact Information

Name	Title	Contact Information
VNDC		
Vuong Le Vinh Nhan	CEO	
CyStack		
Vo Huyen Nhi	Account Manager	Email: nhivh@cystack.net

Assessment Overview

From January 01, 2021 to January 30, 2021, VNDC engaged CyStack to evaluate the security posture of the VIDB Token of their contract system. The findings and recommendations are presented here in this initial report.

NOTE: The VNDC's mitigation efforts of the audit findings are currently still in progress. The report will be subject to updates to reflect issues that can be considered closed.

About VIDB

VIDB (VNDC International Digital Banking) is a utility token in the VNDC financial ecosystem, including: VNDC Wallet, VNDC Exchange, VNDC Borrow, VNDC Staking, VNDC Farming, VNDC P2P,...VIDB is used for payment of transaction fees, collateral for partners, account rating,...

In addition, VIDB is considered as an asset presenting the value of the VNDC – a platform with more than 800,000 users and transaction volume is over 150 million dollars (more than 3,000 billion VND) per month. Investors owning VIDB will be shared profit from VNDC business activities.

The basic information of VIDB is as follows:

Item	Description
Issuer	VNDC
Website	https://vndc.io
Source code	https://etherscan.io/address/0xbfce0c7d3ba3a7f7a039521fe371a87bf84baad4#code
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox

Table 1: Basic information of VIDB Token

In the scope of this project, CyStack focuses on audit the smart contract which allocated at :
https://etherscan.io/address/0xbfce0c7d3ba3a7f7a039521fe371a87bf84baad4#code

About CyStack

CyStack is a leading security company in Vietnam with the goal of building the next generation of cybersecurity solutions to protect businesses against threats from the Internet. CyStack is a member of Vietnam Information Security Association (VNISA) and Vietnam Alliance for Cybersecurity Products Development.

CyStack's researchers are known as regular speakers at well-known cybersecurity conferences such as BlackHat USA, BlackHat Asia, Xcon, T2FI, etc and are talented bug hunters who discovered critical vulnerabilities in global products and acknowledged by their vendors.

Methodology

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

- **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 classified into four categories accordingly, i.e., Critical, High, Medium, Low shown in Table 2.

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

Table 2: Vulnerability Severity Classification

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 additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 3.

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

Category	Check Item
Advanced DeFi Scrutiny	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

Table 3: The Full List of Check Items

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 practice.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699), 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 4 to classify our findings.

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.

Table 4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Executive Summary

State of Security

Here is a summary of our findings after analyzing the design and implementation of the VIDB Token. 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.

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 that 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 Findings.

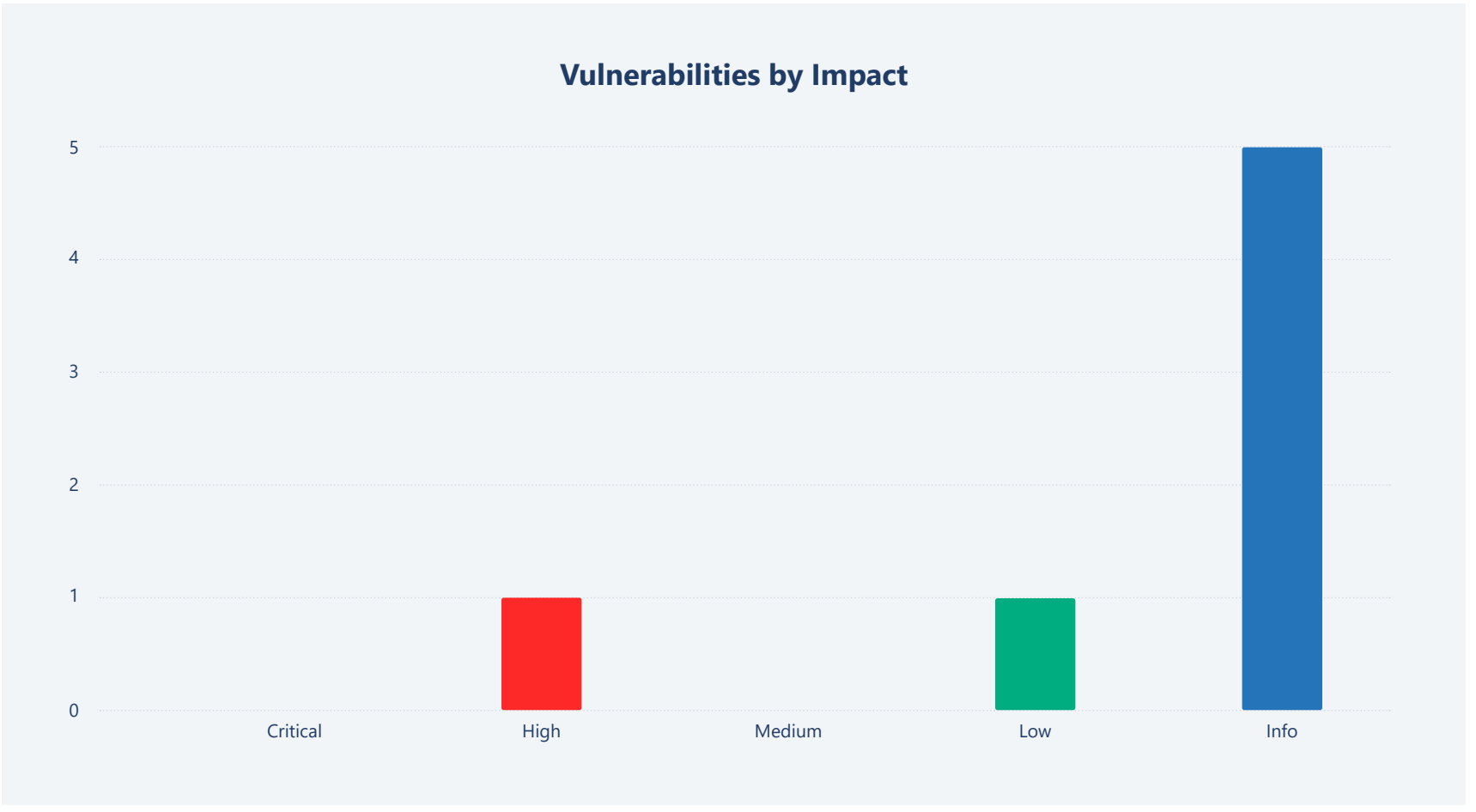

Figure 1: Vulnerabilities by severity

Recommendations

Based on the results of this audit, CyStack has the following high-level key recommendations

	KEY RECOMMENDATION
Key Issue	Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 7) including 1 high-severity vulnerability, 1 low-severity vulnerability, and 5 informational recommendations.
Recommendation	Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to the Findings section for details.
Resources	https://docs.soliditylang.org/en/latest/security-considerations.html

Table 5: Key Recommendation

Findings

Overview

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 8). During the engagement, **7** unique vulnerabilities were found across **2** different vulnerability categories. Vulnerabilities of the following kind were identified:

- Resource Management
- Coding Practices

The following table shows a visualization of how assessment components performed against the most common types of vulnerabilities as defined by the CWE Classifications :

CWE CATEGORY	TEST RESULT	FINDINGS
Configuration	✓	0
Data Processing Issues	✓	0
Numeric Errors	✓	0
Security Features	✓	0
Time and State	✓	0
Error Conditions, Return Values, Status Codes	✓	0
Resource Management	⊗	1
Behavioral Issues	✓	0
Business Logics	✓	0
Initialization and Cleanup	✓	0
Arguments and Parameters	✓	0
Expression Issues	✓	0
Coding Practices	⊗	6

Table 6: Vulnerabilities by CWE category

We use a list of test cases for ensuring the smart contract meets the minimum requirements of security. The result is as below

#	Property	Description	Result
1	abiencoderv2-array	Storage abiencoderv2 array	Passed
2	array-by-reference	Modifying storage array by value	Passed
3	incorrect-shift	The order of parameters in a shift instruction is incorrect	Passed
4	multiple-constructors	Multiple constructor schemes	Passed
5	name-reused	Contract's name reused	Passed
6	public-mappings-nested	Public mappings with nested variables	Passed
7	rtlo	Right-To-Left-Override control character is used	Passed
8	shadowing-state	State variables shadowing	Passed
9	suicidal	Functions allowing anyone to destruct the contract	Passed
10	uninitialized-state	Uninitialized state variables	Passed
11	uninitialized-storage	Uninitialized storage variables	Passed
12	unprotected-upgrade	Unprotected upgradeable contract	Passed
13	arbitrary-send	Functions that send Ether to arbitrary destinations	Passed
14	controlled-array-length	Tainted array length assignment	VIDB-001
15	controlled-delegatecall	Controlled delegatecall destination	Passed
16	reentrancy-eth	Reentrancy vulnerabilities (theft of ethers)	Passed
17	storage-array	Signed storage integer array compiler bug	Passed
18	weak-prng	Weak PRNG	Passed
19	enum-conversion	Detect dangerous enum conversion	Passed

#	Property	Description	Result
20	erc20-interface	Incorrect ERC20 interfaces	Passed
21	erc721-interface	Incorrect ERC721 interfaces	Passed
22	incorrect-equality	Dangerous strict equalities	Passed
23	locked-ether	Contracts that lock ether	VIDB-002
24	mapping-deletion	Deletion on mapping containing a structure	Passed
25	shadowing-abstract	State variables shadowing from abstract contracts	Passed
26	tautology	Tautology or contradiction	Passed
27	boolean-cst	Misuse of Boolean constant	Passed
28	constant-function-asm	Constant functions using assembly code	Passed
29	constant-function-state	Constant functions changing the state	Passed
30	divide-before-multiply	Imprecise arithmetic operations order	Passed
31	reentrancy-no-eth	Reentrancy vulnerabilities (no theft of ethers)	Passed
32	reused-constructor	Reused base constructor	Passed
33	tx-origin	Dangerous usage of tx.origin	Passed
34	unchecked-lowlevel	Unchecked low-level calls	Passed
35	unchecked-send	Unchecked send	Passed
36	uninitialized-local	Uninitialized local variables	Passed
37	unused-return	Unused return values	Passed
38	incorrect-modifier	Modifiers that can return the default value	Passed
39	shadowing-builtin	Built-in symbol shadowing	Passed

#	Property	Description	Result
40	shadowing-local	Local variables shadowing	Passed
41	uninitialized-fptr-cst	Uninitialized function pointer calls in constructors	Passed
42	variable-scope	Local variables used prior their declaration	Passed
43	void-cst	Constructor called not implemented	Passed
44	calls-loop	Multiple calls in a loop	Passed
45	events-access	Missing Events Access Control	Passed
46	events-maths	Missing Events Arithmetic	Passed
47	incorrect-unary	Dangerous unary expressions	Passed
48	missing-zero-check	Missing Zero Address Validation	Passed
49	reentrancy-benign	Benign reentrancy vulnerabilities	Passed
50	reentrancy-events	Reentrancy vulnerabilities leading to out-of-order Events	Passed
51	timestamp	Dangerous usage of block.timestamp	VIDB-003
52	assembly	Assembly usage	Passed
53	boolean-equal	Comparison to boolean constant	Passed
54	deprecated-standards	Deprecated Solidity Standards	Passed
55	erc20-indexed	Un-indexed ERC20 event parameters	Passed
56	function-init-state	Function initializing state variables	Passed
57	low-level-calls	Low level calls	Passed
58	missing-inheritance	Missing inheritance	Passed
59	naming-convention	Conformity to Solidity naming conventions	Passed

#	Property	Description	Result
60	pragma	If different pragma directives are used	Passed
61	redundant-statements	Redundant statements	Passed
62	solc-version	Incorrect Solidity version	VIDB-004
63	unimplemented-functions	Unimplemented functions	Passed
64	unused-state	Unused state variables	Passed
65	assert-state-change	Assert state change	Passed
66	costly-loop	Costly operations in a loop	Passed
67	reentrancy-unlimited-gas	Reentrancy vulnerabilities through send and transfer	Passed
68	similar-names	Variable names are too similar	Passed
69	too-many-digits	Conformance to numeric notation best practices	VIDB-005
70	constable-states	State variables that could be declared constant	VIDB-006
71	external-function	Public function that could be declared external	VIDB-007

Table 7: Test case results

The following table contains all the issues discovered during the audit. The issues are ordered based on their severity. More detailed descriptions of the levels of severity can be found in Appendix 4

ID	Severity	Title	Category	Status
VIDB-001	HIGH	Dead Amount Possibility	Resource Management	Confirmed
VIDB-002	LOW	Locked Ether	Coding Practices	Confirmed
VIDB-003	INFORMATIONAL	Block Timestamp	Coding Practices	Confirmed
VIDB-004	INFORMATIONAL	Allowing Older Solidity Versions	Coding Practices	Confirmed
VIDB-005	INFORMATIONAL	Too Many Digits	Coding Practices	Confirmed
VIDB-006	INFORMATIONAL	State Variables That Could Be Declared Constant	Coding Practices	Confirmed
VIDB-007	INFORMATIONAL	Public function that could be declared external	Coding Practices	Confirmed

Table 8: Key findings

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to the section Details.

Details

1. Dead Amount Possibility

ID	#VIDB-001
Severity	HIGH
Impact	HIGH
Likelihood	HIGH
Status	Confirmed
Category	Resource Management
CWE subcategory	CWE-399

Description

In transfer and transferFrom function, the contract checks for require(_amount <= getAvailableBalance(_from));. And getAvailableBalance function call getLockedAmount. In getLockedAmount, it loops for all items in lockList[lockedAddress].

```
719      for(uint256 j = 0; j<lockList[lockedAddress].length; j++) {
720          if(now < lockList[lockedAddress][j].releaseDate) {
721              uint256 temp = lockList[lockedAddress][j].amount;
722              lockedAmount += temp;
723           }
724      }
```

[VIDBToken.getLockedAmount()] (contract.sol#719-724)

Array **lockList** gets increasing over time and keeps increasing the gas that is used for **getLockedAmount** and also increases the gas for the transfer function. When it reaches a certain number, the gas cost for **transfer** will become over the limit of gas for a transaction and make it unable to transfer tokens.

Recommendation

Make a storage mapping for storing user locked amounts.

2. Locked Ether

ID	#VIDB-002
Severity	LOW
Impact	LOW
Likelihood	LOW
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

The contract has a **payable** function but without a withdrawal capacity. Every Ether sent to the contract will be lost.

```
786      function () payable external {
787          revert();
788      }
```

[VIDBToken.fallback()] (contract.sol#786-788)

Recommendation

Remove the payable attribute or add a withdraw function.

3. Block Timestamp

ID	#VIDB-003
Severity	INFORMATIONAL
Impact	N/A
Likelihood	N/A
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

Dangerous usage of **block.timestamp**, which can be manipulated by miners. Exploit Scenario: Bob's contract relies on **block.timestamp** for its randomness. Eve is a miner and manipulates **block.timestamp** to exploit Bob's contract.

```
722      if(now < lockList[lockedAddress][j].releaseDate) {
...          ...
725      }
```

[VIDBToken.getLockedAmount(address)](contract.sol#722)

Recommendation

Avoid relying on **block.timestamp**.

4. Allowing Older Solidity Versions

ID	#VIDB-004
Severity	INFORMATIONAL
Impact	N/A
Likelihood	N/A
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

Pragma version **^0.5.0** allows old versions and **solc-0.5.0** is not recommended for deployment. **solc** frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements.

```
5        pragma solidity ^0.5.0;
```

[VIDBToken.getLockedAmount(address)](contract.sol#722)

Recommendation

Deploy with any of the following Solidity versions:

- 0.5.11 - 0.5.13,
- 0.5.15 - 0.5.17,
- 0.6.8,
- 0.6.10 - 0.8.0. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

5. Too Many Digits

ID	#VIDB-005
Severity	INFORMATIONAL
Impact	N/A
Likelihood	N/A
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

Literals with many digits are difficult to read and review. While 1_ether looks like 1 ether, it is 10 ether. As a result, it's likely to be used incorrectly.

```
669      totalTokens = 1000000000 * 10 ** uint256(decimals());
672      ERC20.transfer(investorWallet, 500000000 * 10 ** uint256(decimals()));
673      ERC20.transfer(airdropWallet, 10000000 * 10 ** uint256(decimals()));
674      ERC20.transfer(advisorWallet, 25000000 * 10 ** uint256(decimals()));
677      transferWithLock(reserveWallet, 125000000 * 10 ** uint256(decimals()), reserveMap[i]);
689      transferWithLock(advisorWallet, 40000000 * 10 ** uint256(decimals()), advisorMap[1]);
```

[VIDBToken.constructor()](contract.sol#669,672,673,674,677,689)

Recommendation

Use:

- Ether suffix:
  https://docs.soliditylang.org/en/latest/units-and-global-variables.html#ether-units
- Time suffix:
  https://docs.soliditylang.org/en/latest/units-and-global-variables.html#ether-units
- The scientific notation:
  https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals

6. State Variables That Could Be Declared Constant

ID	#VIDB-006
Severity	INFORMATIONAL
Impact	N/A
Likelihood	N/A
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

Constant state variables should be declared constant to save gas.

```
615     address investorWallet = 0x278406d5a5198203ECc54B6a4b3612F174A73f69;
616     address reserveWallet = 0x72EBac03226b1937094c09ca3c181b52630695d5;
617     address foundationWallet = 0xb6f85f280e30c4f2b2739E62Da8166471a170D23;
618     address airdropWallet = 0x638551D8B1a5c582beC4cA978A894CA1B830157E;
619     address advisorWallet = 0x5b4774C795A35269FBc858f84B2242d86fEF75Ed;
```

(contract.sol#L615-619)

Recommendation

Add the **constant** attributes to state variables that never change.

7. Public Function That Could Be Declared External

ID	#VIDB-007
Severity	INFORMATIONAL
Impact	N/A
Likelihood	N/A
Status	Confirmed
Category	Coding Practices
CWE subcategory	CWE-1006

Description

Public functions that are never called by the contract should be declared external to save gas.

```
337     function approve(address spender, uint256 amount) public returns (bool) {}
372     function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {}
391     function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {}
510     function name() public view returns (string memory){}
518     function symbol() public view returns (string memory) {}
591     function renounceOwnership() public onlyOwner {}
600     function transferOwnership(address newOwner) public onlyOwner {}
736     function getLockedAddresses() public view returns (address[] memory) {}
740     function getNumberOfLockedAddresses() public view returns (uint256 _count) {}
752     function getLockedAddressesCurrently() public view returns (address[] memory) {}
774     function getCirculatingSupplyTotal() public view returns (uint256 _amount) {}
778     function getBurnedAmountTotal() public view returns (uint256 _amount) {}
782     function burn(uint256 _amount) public {}
```

(contract.sol#...)

Recommendation

Use the **external** attribute for functions never called from the contract.

Conclusion

In this audit, we have analyzed the design and implementation of the VIDB Token. We are impressed by the design and implementation of this token. The current code base is well organized and those identified issues are promptly confirmed and fixed. As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedback or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

Appendix

1. Basic Coding Bugs

1.1. Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

1.2. Ownership Takeover

- Description: Whether the set owner function is not protected
- Result: Not found
- Severity: Critical

1.3. Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function
- Result: Not found
- Severity: Critical

1.4. Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities
- Result: Not found
- Severity: Critical

1.5. Reentrancy

- Description: Reentrancy [20] is an issue when code can call back into your contract and change state, such as withdrawing ETHs
- Result: Not found
- Severity: Critical

1.6. Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address
- Result: Not found
- Severity: High

1.7. Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out
- Result: Not found
- Severity: High

1.8. Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address
- Result: Not found
- Severity: Medium

1.9. Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected revert
- Result: Not found
- Severity: Medium

1.10. Unchecked External Call

- Description: Whether the contract has any external call without checking the return value
- Result: Not found
- Severity: Medium

1.11. Gasless Send

- Description: Whether the contract is vulnerable to gasless send
- Result: Not found
- Severity: Medium

1.12. Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer
- Result: Not found
- Severity: Medium

1.13. Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception
- Result: Not found
- Severity: Medium

1.14. (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries
- Result: Not found
- Severity: Medium

1.15. (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated
- Result: Not found
- Severity: Medium

1.16. Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions
- Result: Not found
- Severity: Medium

1.17. Deprecated Uses

- Description: Whether the contract use the deprecated tx.origin to perform the authorization
- Result: Not found
- Severity: Medium

2. Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract
- Result: Not found
- Severity: Critical

3. Additional Recommendations

3.1. Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of byte[], as the latter is a waste of space.
- Result: Not found
- Severity: Low

3.2. Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables
- Result: Not found
- Severity: Low

3.3. Make Type Inference Explicit

- Description: Do not use keyword var to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop
- Result: Not found
- Severity: Low

3.4. Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from calls() [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing transfer() of ERC20 tokens)
- Result: Not found
- Severity: Low

4. Severity

Informational

The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth.

Low

Low issues are generally subjective in nature or potentially deal with topics like “best practices” or “readability”. Low issues will in general not indicate an actual problem or bug in code.

The maintainers should use their own judgment as to whether addressing these issues improves the codebase.

Medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

High

High issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable or may require a certain condition to arise in order to be exploited.

Left unaddressed, these issues are highly likely to cause problems with the operation of the contract or to lead to a situation that allows the system to be exploited in some way.

Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed, these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract system.