# GlobeDX GDT

## Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **April 18th-23th, 2021**

Visit: **Halborn.com**

# DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 04/21/2021 | Gokberk Gulgun |
| 0.2 | Document Edits | 04/22/2021 | Gabi Urrutia |
| 1.0 | Final Version | 04/23/2021 | Gabi Urrutia |
| 1.1 | Remediation Plan | 04/27/2021 | Gabi Urrutia |

# CONTACTS

| CONTACT | COMPANY | EMAIL |
|---|---|---|
| Rob Behnke | Halborn | Rob.Behnke@halborn.com |
| Steven Walbroehl | Halborn | Steven.Walbroehl@halborn.com |
| Gabi Urrutia | Halborn | Gabi.Urrutia@halborn.com |
| Gokberk Gulgun | Halborn | Gokberk.Gulgun@halborn.com |
| Piotr Cielas | Halborn | Piotr.Cielas@halborn.com |

# EXECUTIVE OVERVIEW

## 1.1 INTRODUCTION

The GlobeDX engaged Halborn to conduct a security assessment on their Smart contract beginning on April 18th, 2021 and ending April 23th, 2021. The security assessment was scoped to the smart contract GDT.sol. An audit of the security risk and implications regarding the changes introduced by the development team at GlobeDX prior to its production release shortly following the assessments deadline.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverables set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

## 1.2 AUDIT SUMMARY

The team at Halborn was provided one week for the engagement and assigned two full time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit to achieve the following:

- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified few security risks, and recommends performing further testing to validate extended safety and correctness in context to the whole of contract. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

# 1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit.While manual testing is recommended to uncover flaws in logic, process,and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart Contract manual code read and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions(solgraph)
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Testnet deployment (Truffle, Ganache)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

**RISK SCALE - LIKELIHOOD**

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.

3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

**RISK SCALE - IMPACT**

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.


**10** - CRITICAL
**9 - 8** - HIGH
**7 - 6** - MEDIUM
**5 - 4** - LOW
**3 - 1** - VERY LOW AND INFORMATIONAL

## 1.4 SCOPE

IN-SCOPE:

Code related to GDT.sol smart contract.

Specific commit of contract:

48f1a087e067c32932e5bdcb42ec201f81a03de1

# 2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 4 |

LIKELIHOOD

IMPACT

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | (HAL-01) | |
| | (HAL-03) | (HAL-02) | | |
| | | | | |
| (HAL-04)<br>(HAL-05)<br>(HAL-06)<br>(HAL-07) | | | | |

# EXECUTIVE OVERVIEW

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
|---|---|---|
| IMPROPER ACCESS CONTROL POLICY | High | SOLVED: 04/27/2021 |
| LACK OF OVERFLOW PROTECTION | Medium | SOLVED: 04/27/2021 |
| LACK OF ADDRESS CONTROL ON THE FUNCTIONS | Low | SOLVED: 04/27/2021 |
| OUTDATED OPENZEPPELIN CONTRACT | Low | RISK ACCEPTED: 04/27/2021 |
| PRAGMA VERSION | Informational | RISK ACCEPTED: 04/27/2021 |
| POSSIBLE MISUSE OF PUBLIC FUNCTIONS | Informational | SOLVED: 04/27/2021 |
| NO TEST COVERAGE | Informational | SOLVED: 04/27/2021 |
| STATIC ANALYSIS | | |
| AUTOMATED SECURITY SCAN RESULTS | | |

# FINDINGS & TECH DETAILS

## 3.1 (HAL-01) IMPROPER ACCESS CONTROL POLICY - HIGH

Description:

Implementing a valid access control policy is an essential step in maintaining the security of a smart-contract. All the features of the smart contract , such as add/remove roles and upgrade contracts are given by Access Control. For instance, Ownership is the most common form of Access Control. In other words, the owner of a contract (the account that deployed it by default) can do some administrative tasks on it. Additional authorization levels are needed to implement the least privilege principle, also known as least-authority, which ensures only authorized processes, users, or programs can access the necessary resources or information. The ownership role is useful in a simple system, but more complex projects require the use of more roles by using Role-based access control.

In the GDT.sol contract, there is no authorization check on the burn function. Each user can burn their own balance on the related function.

Code Location:

GDT.sol Lines #56-66

```
56      function burn(uint256 amount)
57          public
58          returns (bool success)
59      {
60          uint256 accountBalance = balanceOf[msg.sender];
61          require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
62          balanceOf[msg.sender] = accountBalance - amount;
63          totalSupply -= amount;
64          emit Transfer(msg.sender, address(0), amount);
65          return true;
66      }
```

GDT.sol Lines #56-66

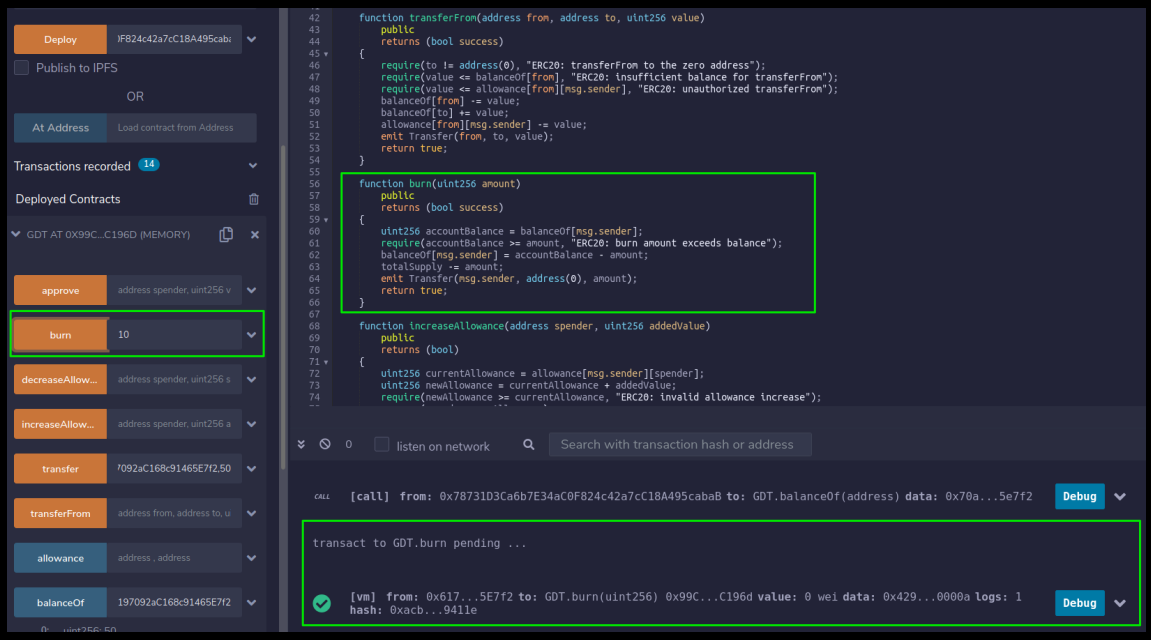


Figure 1: Any user can burn their own balance.

Risk Level:

**Likelihood - 4**
**Impact - 4**

Recommendation:

It's recommended to use role-based access control based on the principle of least privilege to lock permissioned functions using different role. In the other words, A white-listing should be applied on the burn function via access control policy. The access control policy of function may govern who can burn their balances.

Reference: https://www.cyberark.com/what-is/least-privilege/

Remediation Plan:

Solved: GlobeDX team included the initial address check in the function.

## 3.2 (HAL-02) LACK OF OVERFLOW PROTECTION - MEDIUM

### Description:

In the GDT.sol functions, The contract does not perform any precondition checks on the function arguments.

### Code Location:

GDT.sol Lines #20-27

```
20 ▾    function transfer(address to, uint256 value) public returns (bool success) {
21          require(to != address(0), "ERC20: transfer to the zero address");
22          require(balanceOf[msg.sender] >= value, "ERC20: insufficient balance for transfer");
23          balanceOf[msg.sender] -= value;  // deduct from sender's balance
24          balanceOf[to] += value;          // add to recipient's balance
25          emit Transfer(msg.sender, to, value);
26          return true;
27      }
```

GDT.sol Lines #42-54

```
42      function transferFrom(address from, address to, uint256 value)
43          public
44          returns (bool success)
45 ▾    {
46          require(to != address(0), "ERC20: transferFrom to the zero address");
47          require(value <= balanceOf[from], "ERC20: insufficient balance for transferFrom");
48          require(value <= allowance[from][msg.sender], "ERC20: unauthorized transferFrom");
49          balanceOf[from] -= value;
50          balanceOf[to] += value;
51          allowance[from][msg.sender] -= value;
52          emit Transfer(from, to, value);
53          return true;
54      }
55
```

GDT.sol Lines #56-66

```
56      function burn(uint256 amount)
57          public
58          returns (bool success)
59 ▾    {
60          uint256 accountBalance = balanceOf[msg.sender];
61          require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
62          balanceOf[msg.sender] = accountBalance - amount;
63          totalSupply -= amount;
64          emit Transfer(msg.sender, address(0), amount);
65          return true;
66      }
```

GDT.sol Lines #68-77

```
68      function increaseAllowance(address spender, uint256 addedValue)
69          public
70          returns (bool)
71      {
72          uint256 currentAllowance = allowance[msg.sender][spender];
73          uint256 newAllowance = currentAllowance + addedValue;
74          require(newAllowance >= currentAllowance, "ERC20: invalid allowance increase");
75          approve(spender, newAllowance);
76          return true;
77      }
78
```

GDT.sol Lines #79-87

```
79      function decreaseAllowance(address spender, uint256 subtractedValue)
80          public
81          returns (bool)
82      {
83          uint256 currentAllowance = allowance[msg.sender][spender];
84          require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
85          approve(spender, currentAllowance - subtractedValue);
86          return true;
87      }
88  }
```

Risk Level:

**Likelihood - 3**
**Impact - 3**

Recommendation:

A contract should consider using the OpenZeppelin SafeMath library on the decimal calculations and should perform a precondition check to prevent an overflow.

Reference: https://docs.openzeppelin.com/contracts/2.x/api/math

Remediation Plan:

Solved: GlobeDX team will use pragma version 0.8.4.

# 3.3 (HAL-03) LACK OF ADDRESS CONTROL ON THE FUNCTIONS - LOW

## Description:

Address validation in contract GDT.sol is missing. Lack of address validation has been found in the multiple functions.

## Code Location:

GDT.sol Lines #14-18

```
14      constructor(address manager) {
15          initial_address = manager;
16          balanceOf[initial_address] = totalSupply;
17          emit Transfer(address(0), initial_address, totalSupply);
18      }
19
```

GDT.sol Lines #20-27

```
20      function transfer(address to, uint256 value) public returns (bool success) {
21          require(to != address(0), "ERC20: transfer to the zero address");
22          require(balanceOf[msg.sender] >= value, "ERC20: insufficient balance for transfer");
23          balanceOf[msg.sender] -= value;  // deduct from sender's balance
24          balanceOf[to] += value;          // add to recipient's balance
25          emit Transfer(msg.sender, to, value);
26          return true;
27      }
28
```

GDT.sol Lines #35-42

```
35      function approve(address spender, uint256 value)
36          public
37          returns (bool success)
38      {
39          allowance[msg.sender][spender] = value;
40          emit Approval(msg.sender, spender, value);
41          return true;
42      }
```

GDT.sol Lines #42-54

```
42      function transferFrom(address from, address to, uint256 value)
43          public
44          returns (bool success)
45      {
46          require(to != address(0), "ERC20: transferFrom to the zero address");
47          require(value <= balanceOf[from], "ERC20: insufficient balance for transferFrom");
48          require(value <= allowance[from][msg.sender], "ERC20: unauthorized transferFrom");
49          balanceOf[from] -= value;
50          balanceOf[to] += value;
51          allowance[from][msg.sender] -= value;
52          emit Transfer(from, to, value);
53          return true;
54      }
55
```

Risk Level:

**Likelihood - 2**
**Impact - 3**

Recommendation:

Add proper address validation when assigning a value to a variable from user supplied inputs. As a better solution, a white-listing/black-listing should be applied on the related functions.

**Listing 1**

```
modifier validAddress(address addr) {
   require(addr != 0, "Value can not be null");
   require(addr != address(0), "Address cannot be 0x0");
   require(addr != address(this), "Address cannot be contract
       address");
   _;
}
```

Remediation Plan:

Solved: GlobeDX team completed address check on the related functions.

## 3.4 (HAL-04) OUTDATED OPENZEPPELIN CONTRACT - INFORMATIONAL

Description:

The GDT contract used outdated version of the OpenZeppelin contract and was written for Solidity 0.4.21. Even though this outdated version does not pose any high-risk vulnerabilities, some of function implementations have since changed, which may lead to GlobeDX being incompatible.

GDT.sol Lines #68-88

```
68      function increaseAllowance(address spender, uint256 addedValue)
69          public
70          returns (bool)
71      {
72          uint256 currentAllowance = allowance[msg.sender][spender];
73          uint256 newAllowance = currentAllowance + addedValue;
74          require(newAllowance >= currentAllowance, "ERC20: invalid allowance increase");
75          approve(spender, newAllowance);
76          return true;
77      }
78
79      function decreaseAllowance(address spender, uint256 subtractedValue)
80          public
81          returns (bool)
82      {
83          uint256 currentAllowance = allowance[msg.sender][spender];
84          require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
85          approve(spender, currentAllowance - subtractedValue);
86          return true;
87      }
88  }
```

Reference: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/437
Reference: https://ethereum.org/tr/developers/tutorials/erc20-annotated-code/

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

The OpenZeppelin contract should be updated to latest stable version for being compatible with modern ERC-20 tokens.

Remediation Plan:

Risk Accepted: the GlobeDX team has decided to use the older version of ERC20 token.

## 3.5 (HAL-05) PRAGMA VERSION - INFORMATIONAL

Description:

GDT.sol uses one of the latest pragma version (0.8.3) which was released on March 23, 2021. The latest pragma version (0.8.4) was released in April 2021. Many pragma versions have been lately released, going from version 0.6.x to the recently released version 0.8.x. in a short time.

Code Location:

GDT.sol Line #~1

```
1  pragma solidity 0.8.3;
2
3  contract GDT {
4      mapping (address => uint256) public balanceOf;
5
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

Consider if possible, using a stable pragma version (0.6.12-0.7.3) that has been well tested to prevent potential undiscovered vulnerabilities.

Reference: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json

Remediation Plan:

Risk Accepted: GlobeDX team considers the use of pragma 0.8.4 appropriate for the deployment to the mainnet.

## 3.6 (HAL-06) POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation. Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, the function expects its arguments being located in memory when the compiler generates the code for an internal function.

Code Location:

**Listing 2**

```
1 function transfer
2 function transferFrom
3 function burn
4 function increaseAllowance
5 function decreaseAllowance
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

Consider declaring external variables instead of public variables. A best practice is to use external if expecting a function to only be called externally and public if called internally. Public functions are always accessible, but external functions are only available to external callers.

Remediation Plan:

Solved: GlobeDX team marked functions as an external.

## 3.7 (HAL-07) NO TEST COVERAGE - INFORMATIONAL

Risk Level:

**Likelihood - 1**
**Impact - 1**

Description:

Unlike other software programs, smart contracts can not be modified or removed if deployed once into a specific address except if you deploy them with a proxy contract. Checking the code by automated testing (unit testing or functional testing) is a good practice to be sure all lines of the code work correctly. Mocha and Chai are useful tools to perform unit test in Smart Contracts functions. Mocha is a Javascript testing framework for creating both synchronous and asynchronous unit tests. Moreover, Chai is an assertions library with some interfaces such as assert, expect and should to develop custom unit tests.

References:
https://github.com/mochajs/mocha
https://github.com/chaijs/chai
https://docs.openzeppelin.com/learn/writing-automated-tests

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended considering to perform as much as possible test cases to cover all possible scenarios in the smart contract.

Remediation Plan:

Solved: GlobeDX team implemented test cases via another framework.

# 3.8 STATIC ANALYSIS REPORT

## Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

## Results:

```
INFO:Detectors:
GDT.constructor(address).manager (GDT.sol#14) lacks a zero-check on :
                - initial_address = manager (GDT.sol#15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Pragma version0.7.2 (GDT.sol#1) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.2 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Variable GDT.initial_address (GDT.sol#9) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
transfer(address,uint256) should be declared external:
        - GDT.transfer(address,uint256) (GDT.sol#20-27)
transferFrom(address,address,uint256) should be declared external:
        - GDT.transferFrom(address,address,uint256) (GDT.sol#42-54)
burn(uint256) should be declared external:
        - GDT.burn(uint256) (GDT.sol#56-66)
increaseAllowance(address,uint256) should be declared external:
        - GDT.increaseAllowance(address,uint256) (GDT.sol#68-77)
decreaseAllowance(address,uint256) should be declared external:
        - GDT.decreaseAllowance(address,uint256) (GDT.sol#79-87)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:GDT.sol analyzed (1 contracts with 72 detectors), 9 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Figure 2: Slither Results

According to the test results, most of the findings found by slither were considered as false positives. Relevant findings were reviewed by the auditors.

# 3.9 AUTOMATED SECURITY SCAN

MYTHX:

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. Only security-related findings are shown below.

Results:

GDT.sol

Report for Tokens/ERC_20/Solidity/GDT.sol
https://dashboard.mythx.io/#/console/analyses/192faf20-f616-4854-b273-47a26e7e2fbc

| Line | SWC Title | Severity | Short Description |
|---|---|---|---|
| 16 | (SWC-100) Function Default Visibility | Low | Function visibility is not set. |
| 22 | (SWC-000) Unknown | Medium | Function could be marked as external. |
| 44 | (SWC-000) Unknown | Medium | Function could be marked as external. |
| 58 | (SWC-000) Unknown | Medium | Function could be marked as external. |
| 70 | (SWC-000) Unknown | Medium | Function could be marked as external. |
| 81 | (SWC-000) Unknown | Medium | Function could be marked as external. |

THANK YOU FOR CHOOSING

// HALBORN