Security Audit Report

Get Smart

7th June 2022

Disclaimer

The following document may contain confidential information about IT systems and the intellectual property of the Customer as well as information about potential vulnerabilities and methods of their exploitation. This report can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities have been fixed.

Document

Name	Smart Contract Security Audit For Badge Contract
Conducted By	Akshat Mishra – Smart Contract Auditor at Lapits Technologies
Type	ERC – 1155 Token
Platform	Ethereum Mainnet
Methods	Architecture Review, Functional Testing, Manual Review, Automated Testing
Website	http://app.get-smart.net/
Timeline	2 Weeks
Tools	Slither, Mythril

Table of Contents

Introduction

Lapits Technologies was contacted by Dr. J.R. Lange, Serial Founder & Education Problem Solver, to review and perform a security audit of the smart contract. He is expert in higher education and online learning design with a 15-year track record of success.

He has founded GetSmart that aims to completely reverse student debt and fund education worldwide by tokenizing debt-relief through a smart contract that connects a tokenized lottery, cryptolending, and cryptoconnected NFT badges. This report presents the findings of the security assessment of our project's smart contracts.

Scope

The scope of this project are the smart contracts present in the repository

Repository: https://github.com/lapitstechnologies/audits/blob/main/GetSmart
Technical Documentation: https://github.com/lapitstechnologies/audits/blob/main/GetSmart/simple-paper.docx
Type: Simple Paper (Requirements Provided)
Files: Badge.sol

Note - All external dependencies have been treated as black-boxes and are assumed to be free of any vulnerabilities. Unless explicitly mentioned, all such dependencies are outside the scope of this report.

Vulnerability Indicators

Indicator	Risk Level	Description
🔴	**Critical**	Critical vulnerabilities are usually straight-forward to exploit and can lead to asset loss or data manipulations
🟠	**Major**	Major vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions
🟡	**Medium**	Medium-level vulnerabilities are important to fix; however, they cannot lead to asset loss or data manipulations
🟤	**Minor**	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution
🔵	**Informational**	Informational vulnerabilities are problems that do not pose a risk to security. These are merely improvements over the existing code

Checked Items

The provided smart contract has been checked for the following vulnerabilities. Here are a few parameters that have been considered.

Item	Type	Description	Status
Visibility Test	SWC-100, SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	**Passed**
Integer Overflow And Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	**Passed**
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the solidity compiler	**Passed**
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	**Failed**
Unchecked Call Return Value	SWC-104	The Return Value of The Contract Should Be Checked	**Not Relevant**
Access Control & Authorization	CWE-184	There shouldn't be any possibility of illegal takeover of ownership	**Failed**
Self-Destruction Instruction	SWC-106	The contract should not be destroyed until it has funds belonging to users.	**Not Relevant**
Deprecated Solidity Functions	SWC-111	Avoid using deprecated built-in	**Passed**
DoS (Denial of Service)	SWC-113, SWC-128	Contract execution must not be stopped due to a failing call	**Passed**
Race Conditions / Front-running	SWC-114	Unsynchronized Transactions, order dependency	**Passed**
Presence of unused variables	SWC-131	The contract should ideally not contain unused variables	**Failed**
Authorization Through TX Origin	SWC-115	Tx.origin shouldn't be used for Authorization	**Passed**
Unencrypted Private Data on Chain	SWC-136	No private data should be stored on a blockchain	**Passed**
Reentrancy Attack	SWC-107	State variables should be updated before performing a transaction	**Passed**
Improper or Uneven Code Style	Custom	The code should be evenly styled, with proper documentation and identifier casing	**Passed**
Incorrect Order of Inheritance	SWC-125	The contract must follow the correct order of Inheritance	**Passed**
Gas Griefing	SWC-126	Contract should be safe from gas griefing attacks	**Passed**

System Overview

The project GetSmart revolves around the contract Badge. This contract is an implementation of the ERC-1155 standard, which means it can deal with both fungible and non-fungible tokens and has the ability to process transactions in batches.

The contract is used to mint and manage badges issued by the sponsors. Students can apply for badges after which they can be transferred to them by the sponsors; the badges also have ERC-20 tokens associated with them, which the students can claim.

Requirements

Following are the requirements of our contract

1) Creating Badges - Badges can be created by the sponsors with parameters describing the badge price, expiration time and the total no. of copies for the given badge

2) Ability To Renew Badges - Badges can be renewed so that they don't expire. This is a functionality available to the sponsors using which they can extend the lifespan of their Badges.

3) Applying For Badges - This is a functionality which only the students can access. Students can apply for badges through badge Ids after which the sponsors can transfer them the badges.

4) Claiming Badges - Students can claim their tokens for the badge price, after which they'll receive the associated ERC-20 token for their badge.

Test Coverage

Total Functions : 26
Total Coverage : 23%

Unit Tests

A total of 6 unit tests have been conducted for the following functions

| Function Name	File Name	Status
applyBadge(uint badgeIds)	**Badge.sol**	**Passed**
getApplicationByBadgeId(uint ids)	**Badge.sol**	**Passed**
transferBadge(address to, uint[] badgeIds)	**Badge.sol**	**Passed**
claimBadgePrice(uint badgeID)	**Badge.sol**	**Passed**
createBadge(uint numberOfCopies,uint _matiureTime,uint _claimedTime, uint _pricedAmount, string URIs, bool flagFee)	**Badge.sol**	**Passed**
resetBadgeExpiry(uint _reMaturedTime, uint _badgeId)	**Badge.sol**	**Passed**
contractURI()	**Badge.sol**	**N/A**
getTokenPrice()	**Badge.sol**	**N/A**
setTokenPrice(uint256 _totalTokensPerEther)	**Badge.sol**	**N/A**
uri(uint256 id)	**Badge.sol**	**N/A**
setTokenUri(uint256 tokenId, string memory IPFS)	**Badge.sol**	**N/A**
getTokenCounter()	**Badge.sol**	**N/A**
getBadgePrice(uint256 ids)	**Badge.sol**	**N/A**
getBadgeExpiry(uint256 ids)	**Badge.sol**	**N/A**
getCoolingTime(uint256 badgeIds)	**Badge.sol**	**N/A**
isBadgeAssign(uint256 ids)	**Badge.sol**	**N/A**
getCopiesLeft(uint256 ids)	**Badge.sol**	**N/A**
setBadgeFee(uint256 _badgeFee)	**Badge.sol**	**N/A**
getBadgeFee()	**Badge.sol**	**N/A**
tokenIdToURI(uint256 ids)	**Badge.sol**	**N/A**

To make the test coverage equal to 95%, the tests should be conducted for other functions as well . **A minimum of 19 other functions should be tested–for the test coverage to reach the desired goal.**

ownerOf(uint256 tokenId)	**Badge.sol**	**N/A**
totalSupply()	**Badge.sol**	**N/A**
tokenByIndex(uint256 index)	**Badge.sol**	**N/A**
_exists(uint256 tokenId)	**Badge.sol**	**N/A**
tokenOfOwnerByIndex(address owner, uint256 index)	**Badge.sol**	**N/A**
tokenOf(address owner)	**Badge.sol**	**N/A**

Vulnerability Report

Vulnerability Score
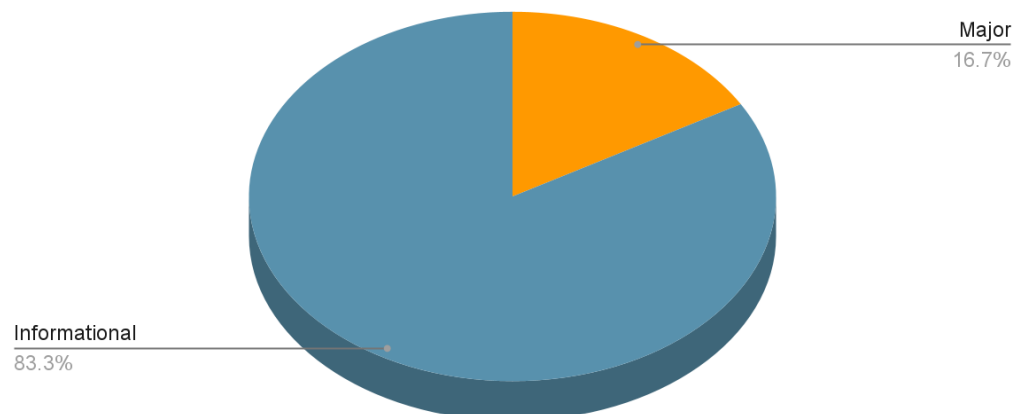
Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Mitigated	Resolved
Critical	0	0	0	0	0	0	0
Major	1	1	0	0	0	0	0
Medium	0	0	0	0	0	0	0
Minor	0	0	0	0	0	0	0
Informational	5	5	0	0	0	0	0

1 – Usage of Long, Lengthy Number Notations

Category : Informational
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : Pending

Description

It has been noticed that the contract makes use of long and lengthy decimal notations for numbers. This is problematic for two reasons; one - it increases the contract size (even though minimally) and second, it makes the number hard to read.

Recommendation

It is suggested that we replace long, lengthy number notation with scientific and hexadecimal ones
Eg.

> Use - uint256 private badgeFee = 9e18;
> Instead of
> uint256 private badgeFee = 9000000000000000000;

2 – Presence of Unused Variables

Category : Informational
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : Pending

Description

It has been noticed that the contract has a public variable declared with the name **tokenUri** of the type string. While this variable has been used in the constructor, it has never been initialized.

Recommendation

It is suggested that the variable either be removed from the contract all-together, or be used in the contract.

3 – Performing Gas Optimisations

Category : Informational
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : Pending

Description

It has been noticed that a lot of code is unoptimised and may lead to increased gas usage. This is primarily due to the use of certain data types and function visibility specifiers.

Recommendation

To reduce gas usage, it is recommended that

1. **Functions that can be made external, should be made external** as they'll save us a lot of gas, this includes most of the getter and setter functions.

   Eg

   ```solidity
   function getBadgeFee() public view returns (uint256) {
       return badgeFee;
   }

   Could be written as

   function getBadgeFee() external view returns (uint256) {
       return badgeFee;
   }
   ```

2. **Avoid using uint8 wherever possible, to reduce gas costs**. This is a strange behavior, this happens because uint8 needs to be processed by the EVM which results in higher opcodes thus higher execution costs.

   This goes for variables used in mappings and structures as well. It is suggested that the code

   ```solidity
   mapping(uint256 => uint8) private isBadgeAssigned;
   ```

   Be Changed To

   ```solidity
   mapping(uint256 => uint32) private isBadgeAssigned;
   ```

4 – Use of Floating Pragma

Category : Informational
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : Pending

Description

The pragma for the Badge contract declares all compiler versions above v 0.8.0 valid. This unlocked pragma can create new problems and induce bugs for our contract for all the versions that our compiler hasn't been tested for.

Recommendation

It is advisable to use only the versions of Solidity compiler that the contract has been tested for
Eg

```solidity
pragma solidity ^0.8.0;
```

Should be

```
pragma solidity 0.8.0;
```

5 – Uneven Documentation

Category : **Informational**
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : **Pending**

Description

It has been noticed that the code is unevenly documented where some functions are missing documentation while others have been poorly documented. Inconsistencies have also been noticed around the use of mixed case for naming variables.

Recommendation

It is suggested that the developers must once go through the code and fix any inconsistencies in it, and follow a generalized code style.

6 – Unrestricted Access To Essential Functions

Category : **Major**
Location : https://github.com/lapitstechnologies/audits/blob/main/GetSmart/Contracts/Badge.sol
Status : **Pending**

Description

The function setBadgeFee(uint fee) has no modifier and this gives any user unrestricted access to the function, using which they can modify the gas fee as they like. This is a major vulnerability as it can drastically affect our contract's functioning.

Recommendation

It is recommended that the function be modified to restrict the access to only the owner.
Eg.

```
function setBadgeFee(uint256 _badgeFee) public{
        badgeFee = _badgeFee;
}

Should Be Changed To
function setBadgeFee(uint256 _badgeFee) public onlyOwner{
        badgeFee = _badgeFee;
}
```

Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices against the most common cybersecurity vulnerabilities and issues in smart contract source code. The details of the diagnosis are disclosed in this report, the audit makes no statements or warranties on the security of the code. It does not guarantee the safety of this contract

It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report. It is important to note that we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts

Get Your Smart Contracts Approved, Tested and Secured By Professionals At Lapits

About Lapits

Cited as one of the best *"Blockchain Services Provider"* companies in India, Lapits is a Blockchain, Web Development and Mobile App development firm, with a long proven history of delivering quality software solutions. Lapits is one of the leaders in the Crypto and Web3 space and has been awarded by multiple national and international renowned bodies.

Lapits houses a powerful team of developers, business analysts and sales managers who know the industry in and out. Be it driving sales, developing apps, or launching an exchange; at lapits, we do it all. For more information on services provided by lapits, head over to Lapits.com