Goldfinger Token & Vesting

Smart Contract Audit

- Goldfinger Audit Summary
- Goldfinger Audit
  - Document information
    - Audit results
    - Audited target file
  - Vulnerability analysis
    - Vulnerability distribution
    - Summary of audit results
    - Contract file
    - Analysis of audit results
      - Re-Entrancy
      - Arithmetic Over/Under Flows
      - Unexpected Blockchain Currency
      - Delegatecall
      - Default Visibilities
      - Entropy Illusion
      - External Contract Referencing
      - Unsolved TODO comments
      - Short Address/Parameter Attack
      - Unchecked CALL Return Values
      - Race Conditions / Front Running
      - Denial Of Service (DOS)
      - Block Timestamp Manipulation
      - Constructors with Care
      - Unintialised Storage Pointers
      - Floating Points and Numerical Precision
      - tx.origin Authentication
      - Permission restrictions

Goldfinger Audit Summary

Project name : Goldfinger Contract

Project address: None

Code URL : https://github.com/yardentamari/GFT/tree/development

Commit : 0553fb0202afc8f03deb2682440deddac0377d8d

Project target : Goldfinger Contract Audit

Blockchain : Polygon

Test result : PASSED

Audit Info

Audit NO : 0X202112010016

Audit Team : Armors Labs

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

Goldfinger Audit

The Goldfinger team asked us to review and audit their Goldfinger contract. 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
Goldfinger Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-12-01

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 Goldfinger 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 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

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. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
GFTTokenVesting.sol	06236c5afc352d63a8f2049a0e349540
GFTToken.sol	8df62bb4f5ef47d26739f301e1431e8d

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 Blockchain Currency	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

Vulnerability	status
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
Permission restrictions	safe

Contract file

GFTToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.10;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract GFTToken is ERC20, Ownable {
    constructor(
        string memory name,
        string memory symbol
    ) payable ERC20(name, symbol) {}

    function mint(address account, uint256 amount) public onlyOwner {
        require(totalSupply() + amount <= 1100000000 * 10 ** decimals());
        _mint(account, amount);
    }

    function burn(uint256 amount) public {
        _burn(msg.sender, amount);
    }
}
```

GFTTokenVesting.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract GFTTokenVesting is Ownable {
  using SafeERC20 for IERC20;

  event Released(address beneficiary, uint256 amount);

  IERC20 public token;
  uint256 public cliff;
  uint256 public start;
```

```
uint256 public duration;
uint256 public period;
uint256 public percent;

mapping (address => uint256) public shares;
mapping (address => uint256) public lastReleaseDate;
mapping (address => uint256) public releasedAmount;

uint256 released = 0;
uint256 BP = 10000;

address[] public beneficiaries;

modifier onlyBeneficiaries {
  require(msg.sender == owner() || shares[msg.sender] > 0, "You cannot release tokens!");
  _;
}

constructor(
  IERC20 _token,
  uint256 _start,
  uint256 _cliff,
  uint256 _duration,
  uint256 _period,
  uint256 _percent
) {
  require(_cliff <= _duration, "Cliff has to be lower or equal to duration");
  token = _token;
  duration = _duration;
  cliff = _start + _cliff;
  start = _start;
  period = _period;
  percent = _percent;
}

function addBeneficiaryes(address[] memory _beneficiaryes, uint256[] memory _sharesAmounts) onlyOwn
  require(_beneficiaryes.length == _sharesAmounts.length);

  for (uint i = 0; i <_beneficiaryes.length; i++) {
    addBeneficiary(_beneficiaryes[i], _sharesAmounts[i]);
  }

  require(totalShares() == 10000, "Invalid shares amount");
}

function addBeneficiary(address _beneficiary, uint256 _sharesAmount) onlyOwner public {
  require(block.timestamp < cliff);
  require(_beneficiary != address(0), "The beneficiary's address cannot be 0");
  require(_sharesAmount > 0, "Shares amount has to be greater than 0");

  if (shares[_beneficiary] == 0) {
    beneficiaries.push(_beneficiary);
  }

  lastReleaseDate[_beneficiary] = cliff;
  shares[_beneficiary] = shares[_beneficiary] + _sharesAmount;
}

function claimTokens() onlyBeneficiaries public {
  uint256 currentBalance = token.balanceOf(address(this));
  uint256 totalBalance = currentBalance + released;

  require(releasedAmount[msg.sender] < calculateShares(totalBalance, msg.sender), "User already rel

  uint256 unreleased = releasableAmount();
```

```
    if (unreleased > 0) {
      uint256 userShare = calculateShares(unreleased, msg.sender);
      released += userShare;
      release(msg.sender, userShare);
      lastReleaseDate[msg.sender] = block.timestamp;
    }
  }

  function userReleasableAmount() public view returns (uint256) {
    return calculateShares(releasableAmount(), msg.sender) - releasedAmount[msg.sender];
  }

  function releasableAmount() public view returns (uint256) {
    return vestedAmount();
  }

  function calculateShares(uint256 _amount, address _beneficiary) public view returns (uint256) {
    return _amount * shares[_beneficiary] / 10000;
  }

  function totalShares() public view returns (uint256 sum) {
    for (uint i = 0; i < beneficiaries.length; i++) {
      sum += shares[beneficiaries[i]];
    }
  }

  function vestedAmount() public view returns (uint256) {
    uint256 currentBalance = token.balanceOf(address(this));
    uint256 totalBalance = currentBalance + released;

    if (block.timestamp < cliff) {
      return 0;
    } else if (block.timestamp >= start + duration) {
      return totalBalance;
    } else {
      if(block.timestamp < lastReleaseDate[msg.sender]) return 0;
      uint256 periodsPassed = (block.timestamp - lastReleaseDate[msg.sender]) / period;
      if (periodsPassed > 0) {
        return totalBalance * (periodsPassed * percent) / BP;
      } else {
        return 0;
      }
    }
  }

  function release(address _beneficiary, uint256 _amount) private {
    token.safeTransfer(_beneficiary, _amount);
    releasedAmount[_beneficiary] += _amount;
    emit Released(_beneficiary, _amount);
  }
}
```


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 Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency 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 Blockchain Currency

- **Description:**
  Typically when Blockchain Currency 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 Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency 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 whBlockchain Currency 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 Blockchain Currency 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 Blockchain Currency 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.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io