Smart Contract Audit

Date 12	September 2021

DOGEMAMA

This report presents the results of our engagement with Dogemama to review all Dogemama and Dogemama Family smart contracts.

1 | EXECUTIVE SUMMARY

This report presents the results of our engagement with Dogemama to review all Dogemama and Dogemama Family smart contracts. Main flattened and deployed contract address can be found at: 0x1c9a40db8fba3a9aa87c4eb381f5fca78f39eb61.

The review was conducted by Nicolas Ward between August 10 and August 20, 2021.

2 | SCOPE

The review focused on the commit hash 0x20ea4cb1ea495fde4368d5138f1278e16bd7657e04b716287b17a8b941939116. A cursory review of the Resolver contract for stealth keys was also performed at commit hash 0x5adda5eeae8b90e90d56e61171a6c861b3cc142f6f98b10cfc1788d39b8ad129. The list of files in scope can be found in the Appendix.

3 | SECURITY SPECIFICATION

This audit assumes that the documentation of the Dogemama platform, as well as the Dogemama white/litepapers as well as the actual code have been thoroughly reviewed as well.

Dogemama is an environmentally friendly cryptocurrency, all-in-one bonus/staking platform, and memecoin aggregator. Dogemama Coin features an innovative deflationary token model that is built on Binance Smart Chain, and will compete as a Dogecoin alternative. Within the ecosystem, users are able to use the Dogemama family portal to access passive rewards in the form of Dogecoin, Shiba Inu, Baby Doge, and many other similar cryptocurrencies.

For its deflationary tokenomics model, Dogemama uses a fork of SafeMoon's "liquify" processes - this deducts a certain percentage of all transacted Dogemama after successful signing of a transaction process, and connects it to 2 receiver contracts:

1. A re-distribution contract, and
2. A liquidity contract. Sub contract 1 receives 1/2 of the deduction and sends it to all token holders proportionally to however many Dogemama coins they hold. Sub contract 2 sends all received DM tokens directly to its liquidity address.

4 | FINDINGS & FIXES

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.

- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.

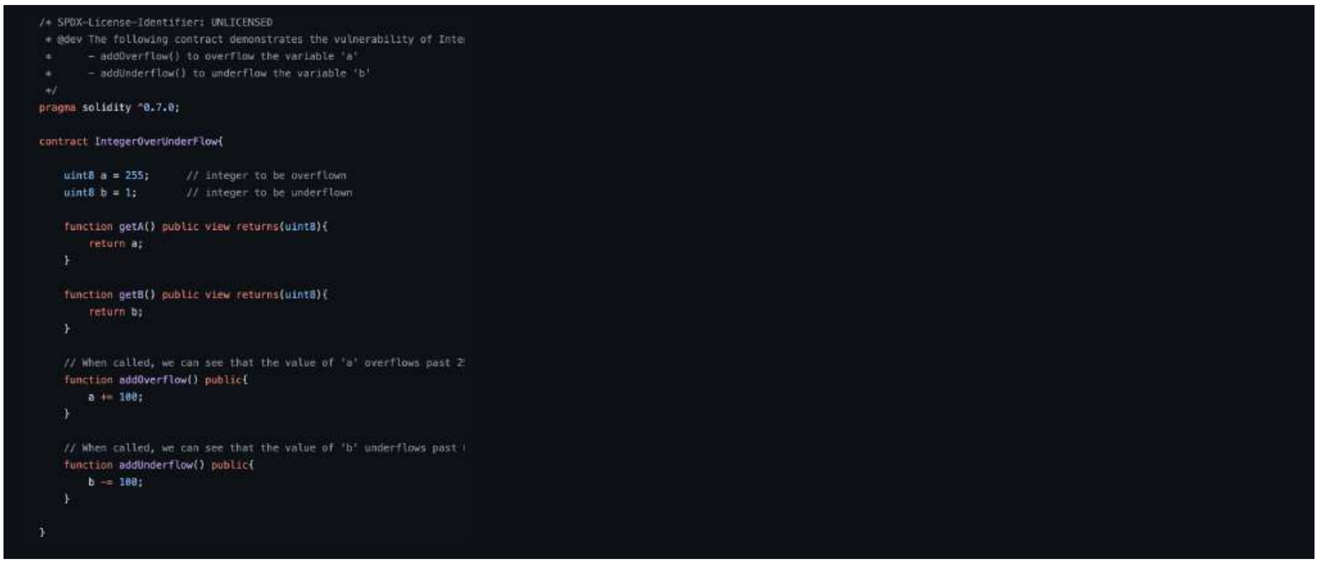
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 | Minor Integer Overflow and Underflow Fixed

An integer over/underflow attack exploits Solidity's counting process of going back to 0 when it goes above the max uint, or inversely jumping to a max value when it decreases past 0. Integer over/under flow should be checked carefully with any contract where users can change a uint value using a call.

The following demonstrates an example of a smart contract that is vulnerable, subject to an integer over/under flow attack:

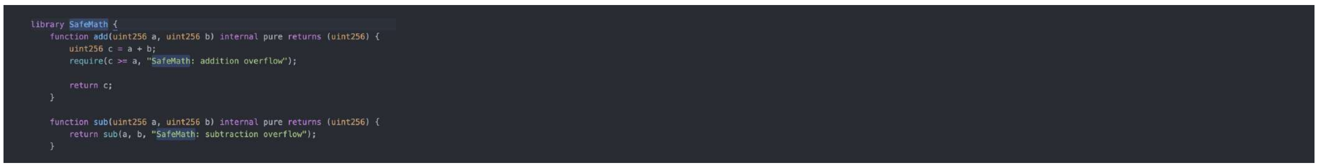


```
/* SPDX-License-Identifier: UNLICENSED
 * @dev The following contract demonstrates the vulnerability of Inte
 *      - addOverflow() to overflow the variable 'a'
 *      - addUnderflow() to underflow the variable 'b'
 */
pragma solidity ^0.7.0;

contract IntegerOverUnderFlow{

    uint8 a = 255;      // integer to be overflown
    uint8 b = 1;        // integer to be underflown

    function getA() public view returns(uint8){
        return a;
    }

    function getB() public view returns(uint8){
        return b;
    }

    // When called, we can see that the value of 'a' overflows past 2
    function addOverflow() public{
        a += 100;
    }

    // When called, we can see that the value of 'b' underflows past
    function addUnderflow() public{
        b -= 100;
    }

}
```

Within Dogemama's core ecosystem of smart contracts, while integer over/underflow was not directly an apparent issue, the previous solution against such a future attack was to rely purely restrict incoming values, which while successful, does not provide a great protection against evolving over/underflow attacks, and best practice would be to import an already existing library for all mathematical processes.

To provide better protection and to solve against potentially being vulnerable post deployment, xsec implemented and tested proper usage of OpenZeppelin's "SafeMath" library within the main DM contract, which has been publicly audited to protect from malicious integer over/under flow attacks. The contract successfully accomplishes this import of the library and safely executes all mathematical operation using the SafeMath library.

TokenPt1.sol L101-L222

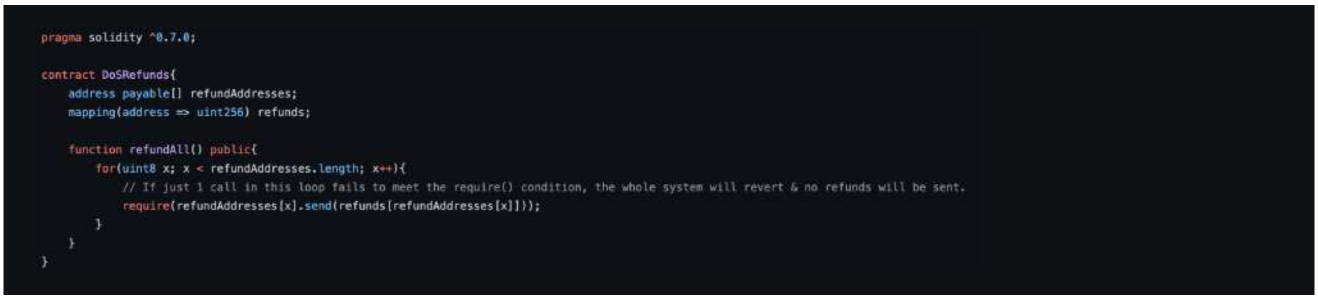


```
library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }
```

4.1 | Minor DoS Fixed

Dos and gas limit attacks are extremely prevalent in the smart contract hacking sector, and traditionally are done by triggering some form of a denial of service on the blockchain the contract is being run on.

The following contract demonstrates an example vulnerability containing a function that sends all owned funds within the contract back to all addresses via 1 single function.



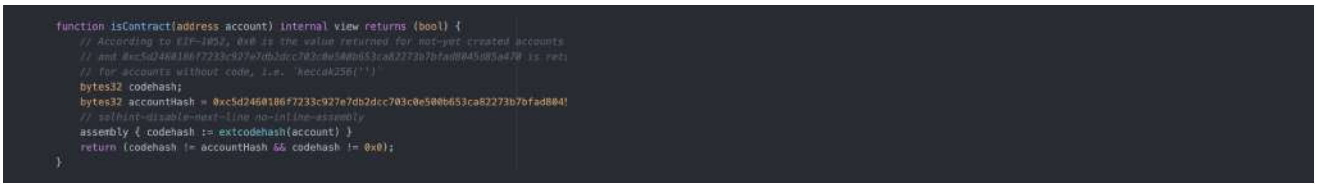
```
pragma solidity ^0.7.0;

contract DoSRefunds{
    address payable[] refundAddresses;
    mapping(address => uint256) refunds;

    function refundAll() public{
        for(uint8 x; x < refundAddresses.length; x++){
            // If just 1 call in this loop fails to meet the require() condition, the whole system will revert & no refunds will be sent.
            require(refundAddresses[x].send(refunds[refundAddresses[x]]));
        }
    }
}
```

This creates a potential flaw, since if one single call within the said function fails, the for loop won't complete, and funds won't be correctly refunded. Therefore, a malicious third party contract manipulating the gas limit interaction to/from the contract (Using a series of miniscule funded addresses in the main contract) could drain all of the user funds in the contract.

To protect from this, first xsec assured that the most common solution to DoS attacks in modern smart contracts, a permission based system for relaying of transactions that could be "griefed".

The oretically, this is already solved and protected against since these contracts are deployed on Binance Smart Chain (With substantially lower gas fees than Ethereum), however direct protection against gas griefing, the potential of sensitive relayed transactions inside of loops failing, and gas-reliant front-running should be addressed within the contract code, just as a secondary precaution.

Xsec achieved this in successfully implementing OpenZeppelin's "address" library, which allows Dogemama to successfully provide whitelisted permissions of relayed transactions to only certain account hashes, as well as assuring all relayed transactions within the ecosystem smart contracts were permissioned rather than directly interactable by third-party malicious addresses. Below demonstrates successful implementation of this at the base layer:



```
function isContract(address account) internal view returns (bool) {
    // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
    // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is ret
    // for accounts without code, i.e. `keccak256('')`
    bytes32 codehash;
    bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad804
    // solhint-disable-next-line no-inline-assembly
    assembly { codehash := extcodehash(account) }
    return (codehash != accountHash && codehash != 0x0);
}
```

4.2 | APPENDIX 1 - Files in Scope

Migrations.sol
Token.sol
SafemoonFork.sol