SECURITY AUDIT OF BSCEX LAUNCHPOOLX SMART CONTRACTS

PUBLIC REPORT

JAN 30, 2021

✓erichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology >> Forward

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Jan 30, 2021. We would like to thank BSCex to trust Verichains Lab to audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

The assessment identified some issues in BSCex LaunchPoolX smart contract code.

Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

CONTENTS

ACRONYMS AND ABBREVIATIONS

Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
ETH (Ether)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
EVM	Ethereum Virtual Machine.

AUDIT OVERVIEW

ABOUT BSCEX

> *BSCex is a decentralized non-custodial cryptocurrency exchange-centered ecosystem that runs on Binance Smart Chain (BSC). BSCex's mission is to make Binance's off-chain services available on the blockchain, develop the applications on BSC, and promote the features of decentralized finance that let our users earn tokens and gain many other economic benefits.*

ABOUT BSCEX LAUNCHPOOLX

> *LaunchPoolX is the on-chain version of Binance exchange's LaunchPool. LaunchPoolX lets you use your tokens and the liquidity provider (LP) token of the BSCX-BUSD pair to farm (earn) a new token, for free. The amount of tokens you earn each daily is proportional to the number of tokens you have subscribed to the pool vs the total number of tokens subscribed to the pool. The tokens you earn are distributed to you in real-time. You get to accumulate a brand new coin, for free.*

SCOPE OF THE AUDIT

This audit focused on identifying security flaws in code and the design of the smart contracts. It was conducted on commit `adf4d88d3994789762a6c561a73f675c577c27fa`.

Source code repository	https://github.com/Bscex/bscex-launchpoolx-contract
Audit branch	master

Audit commit	`commit adf4d88d3994789762a6c561a73f675c577c27fa (HEAD -> master, origin/master, origin/HEAD)`
	`Author: henry <hoang.nong.nv@gmail.com>`
	`Date: Thu Jan 14 11:57:34 2021 +0000`
	
	`Update transfer referral`

The final report is based on the update patches on commit `fa81b7c272288a54f037c3075357d764c36cfc54` by BSCex on Feb 5, 2021.

Source code repository	https://github.com/Bscex/bscex-launchpoolx-contract
Audit branch	master
Audit commit	commit fa81b7c272288a54f037c3075357d764c36cfc54 (HEAD -> master, origin/master, origin/HEAD) Author: henry <hoang.nong.nv@gmail.com> Date: Fri Feb 5 13:09:58 2021 +0000 Update handle reward forFarmer

AUDIT METHODOLOGY

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- TimeStamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories, depending on their criticality:

LOW	An issue that does not have a significant impact, can be considered as less important
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.

AUDIT RESULTS

The smart contract in this audit implements a mixture of logics for:

- Multi token pool holder: hold user's token and let them earn profit in another token.
- Time-based locked tokens that distribute parts of holding token after pre-configured block numbers.
- Referral logics: with configurable percentage for level 1 and level 2 referrers of each pool.

Most of the above logics were carefully implemented but we managed to identify some issues as described in the following section.

Updates:

- Jan 26, 2021: BSCex fixed all found threats and other related suggestions.
- Feb 5, 2021: BSCex found and fixed an issue in **getPoolReward** function.

VULNERABILITIES FINDINGS

FIXED CRITICAL
WRONG DATA LOCATION USAGE AT **UNLOCK** FUNCTION

Current code of **unlock** function in **BSCXNTS** contract:

```
function unlock(uint256 _pid) public {
    PoolInfo memory pool = poolInfo[_pid];

    /* ... */

    pool.totalLock = pool.totalLock.sub(amount);
    totalLocks[pool.rewardToken] = totalLocks[pool.rewardToken].sub(amount);
}
```

If the **memory** data location is used, then the **pool** object will be a *copy* of **poolInfo[_pid]** object, so all modifications to **pool** object (such as updating **totalLock** is this case) will not affect to the original object in **poolInfo** and will be discarded when the function returns.

RECOMMENED FIXES

Use **storage** instead of **memory** for data location as below:

```
function unlock(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];

    /* ... */

    pool.totalLock = pool.totalLock.sub(amount);
    totalLocks[pool.rewardToken] = totalLocks[pool.rewardToken].sub(amount);
}
```

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a*.

REJECTED CRITICAL
WRONG DEV TEAM ADDRESS AT _TRANSFERREFERRAL FUNCTION

For each pool, the address of dev team was stored at **teamAddresses** array. To send reward to dev team, the contract must check the address at **teamAddresses** first and use this address if it is a valid address. If not, then the contract will use the address at **devaddr**.
But in the **_transferReferral** function, the contract does not check for address stored at **teamAddresses**, only use the address at **devaddr** to send reward to.

```
    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        /* ... */

        if (referAmountForDev > 0) {
            pool.rewardToken.transfer(devaddr, referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
            uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
            farmLock(devaddr, lockAmount, _pid);
        }
    }
```

RECOMMENDED FIXES

Check and use the corresponding dev team address as below:

```
    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        /* ... */

        if (referAmountForDev > 0) {
            if (teamAddresses[_pid] != address(0)) {
                pool.rewardToken.transfer(teamAddresses[_pid], referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
                farmLock(teamAddresses[_pid], lockAmount, _pid);
            } else {
                pool.rewardToken.transfer(devaddr, referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
                farmLock(devaddr, lockAmount, _pid);
            }
        }
    }
```

> **Update: Jan 26, 2021** - BSCex confirmed that this is not a bug and won't fix this.

FIXED CRITICAL
WRONG CALCULATION OF **REFERAMOUNTFORDEV** AT **_TRANSFERREFERRAL** FUNCTION

There are two cases when the calculation for **referAmountForDev** will be wrong:

- (1) when referrer level 2 can receive the reward but referrer level 1 cannot, then the dev team will receive only reward amount for referrer level 1. But in current implementation, the dev team will receive reward amount for both level 1 and level 2.
- (2) when both referrer level 1 and referrer level 2 cannot receive the reward, then all referral reward must be transfered to dev team. But in current implementation, the dev team will receive only reward amount for referrer level 2.

Current code of **_transferReferral** function in **BSCXNTS** contract:

```
    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        PoolInfo storage pool = poolInfo[_pid];
        address referrerLv1 = referrers[address(msg.sender)];
        uint256 referAmountForDev = 0;

        if (referrerLv1 != address(0)) { // IF (C)
            uint256 lpStaked = referralLPToken.balanceOf(referrerLv1);
            if (lpStaked >= stakeAmountLPLv1) { // IF (C.1)
                pool.rewardToken.transfer(referrerLv1, _referAmountLv1.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                farmLock(referrerLv1, lockAmount, _pid);
            } else {
                referAmountForDev = _referAmountLv1.add(_referAmountLv2); // (1) cannot
include _referAmountLv2 if IF (C.2) is true
            }

            address referrerLv2 = referrers[referrerLv1];
            uint256 lpStaked2 = referralLPToken.balanceOf(referrerLv2);
            if (referrerLv2 != address(0) && lpStaked2 >= stakeAmountLPLv2) { // IF (C.2)
                pool.rewardToken.transfer(referrerLv2, _referAmountLv2.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100); //
(3) wrong using of _referAmountLv1, must be _referAmountLv2
                farmLock(referrerLv2, lockAmount, _pid);
            } else {
```

```
                referAmountForDev = _referAmountLv2; // (2) missing referAmountLv1 if (C.1)
is false
            }
        } else {
            referAmountForDev = _referAmountLv1.add(_referAmountLv2);
        }

        if (referAmountForDev > 0) {
            pool.rewardToken.transfer(devaddr, referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
            uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
            farmLock(devaddr, lockAmount, _pid);
        }
    }
```

Consider three IF statements (C), (C.1) and (C.2):

- (1) when (C) is true, (C.1) is false and (C.2) is true, then **referAmountForDev** must be equal to **_referAmountLv1**. But in statement (1), it is equal to sum of **_referAmountLv1** and **_referAmountLv2**.
- (2) when (C) is true and both (C.1) and (C.2) are false, then **referAmountForDev** must be equal to sum of **_referAmountLv1** and **_referAmountLv2**. But in statement (2), the **referAmountLv1** is missing.

Additionally, we noticed another bug in statement (3), where the variable **_referAmountLv2** must be used instead if **_referAmountLv1**.

RECOMMENDED FIXES

Update the calculations for **referAmountForDev** as below:

```
    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        PoolInfo storage pool = poolInfo[_pid];
        address referrerLv1 = referrers[address(msg.sender)];
        uint256 referAmountForDev = 0;

        if (referrerLv1 != address(0)) { // IF (C)
            uint256 lpStaked = referralLPToken.balanceOf(referrerLv1);
            if (lpStaked >= stakeAmountLPLv1) { // IF (C.1)
                pool.rewardToken.transfer(referrerLv1, _referAmountLv1.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                farmLock(referrerLv1, lockAmount, _pid);
            } else {
                // dev team will receive reward of referrer level 1
```

```
                referAmountForDev = referAmountForDev.add(_referAmountLv1);
            }

            address referrerLv2 = referrers[referrerLv1];
            uint256 lpStaked2 = referralLPToken.balanceOf(referrerLv2);
            if (referrerLv2 != address(0) && lpStaked2 >= stakeAmountLPLv2) { // IF (C.2)
                pool.rewardToken.transfer(referrerLv2, _referAmountLv2.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv2.mul(pool.percentLockReward).div(100);
                farmLock(referrerLv2, lockAmount, _pid);
            } else {
                // dev team will receive reward of referrer level 2
                referAmountForDev = referAmountForDev.add(_referAmountLv2);
            }
        }

        if (referAmountForDev > 0) {
            pool.rewardToken.transfer(devaddr, referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
            uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
            farmLock(devaddr, lockAmount, _pid);
        }
    }
```

> **Update: Jan 26, 2021** - BSCex fixed this in commit *498701277bf148e2d6f5896642ede7d2e19503dd.*

FIXED **MEDIUM**

MISSING EVENTS WHEN UPDATING CRITICAL CONTRACT STATES

Almost all of the state-changing methods of the contracts doesn't emit events when mutate the data, such as **setReferralLPToken**, **setTeamAddressPool**, **setAmountLPStakeLevelRefer**, **setPercentLPLevelRefer**… It makes users harder to follow changes made to the contract or verify the contract calculations.

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a.*

FIXED **LOW**

UNNECESSARY COSTLY OPERATION

Some functions in **BSCXNTS** contract contain unnecessary object copy operators just to obtain the object's property:

```
    function totalLockInPool(uint256 _pid) public view returns (uint256) {
        PoolInfo memory pool = poolInfo[_pid];
        return pool.totalLock;
    }

    function lockOf(address _holder, uint256 _pid) public view returns (uint256) {
        UserInfo memory user = userInfo[_pid][_holder];

        return user.lockAmount;
    }

    function lastUnlockBlock(address _holder, uint256 _pid) public view returns (uint256) {
        UserInfo memory user = userInfo[_pid][_holder];

        return user.lastUnlockBlock;
    }

    function canUnlockAmount(address _holder, uint256 _pid) public view returns (uint256) {
        PoolInfo memory pool = poolInfo[_pid];
        /* ... */
    }

    function getNewRewardPerBlock(uint256 pid1) public view returns (uint256) {
        PoolInfo memory pool = poolInfo[pid1];
        /* ... */
    }
```

RECOMMENDED FIXES

Use object's property directly or use **storage** instead of **memory** as the code below:

```
    function totalLockInPool(uint256 _pid) public view returns (uint256) {
        return poolInfo[_pid].totalLock;
    }

    function lockOf(address _holder, uint256 _pid) public view returns (uint256) {
        return userInfo[_pid][_holder].lockAmount;
    }

    function lastUnlockBlock(address _holder, uint256 _pid) public view returns (uint256) {
        return userInfo[_pid][_holder].lastUnlockBlock;
    }

    function canUnlockAmount(address _holder, uint256 _pid) public view returns (uint256) {
```

```
        PoolInfo storage pool = poolInfo[_pid];
        /* ... */
    }

    function getNewRewardPerBlock(uint256 pid1) public view returns (uint256) {
        PoolInfo storage pool = poolInfo[pid1];
        /* ... */
    }
```

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a*.

FIXED **LOW**

REDUNDANT CALCULATIONS

The following calculation pattern can be simplified for gas saving.

```
    // current calculations
    value = total.mul(percent).div(100);
    remain = total.mul(100 - percent).div(100);

    // can be simplified to
    value = total.mul(percent).div(100);
    remain = total.sub(value);
```

(Note: the above code is just for demonstration purpose and does not exist in BSCex LaunchPoolX smart contracts)

We have found this pattern in several places in the **BSCXNTS** contract:

```
    function updatePool(uint256 _pid) public {
        /* ... */
        if (forDev > 0) {
            if (teamAddresses[_pid] != address(0)) {
                pool.rewardToken.transfer(teamAddresses[_pid], forDev.mul(100 -
pool.percentLockReward).div(100));
                farmLock(teamAddresses[_pid], forDev.mul(pool.percentLockReward).div(100),
_pid);
            } else {
                pool.rewardToken.transfer(devaddr, forDev.mul(100 -
pool.percentLockReward).div(100));
                farmLock(devaddr, forDev.mul(pool.percentLockReward).div(100), _pid);
            }
        }
```

```
        /* ... */
    }

    function _harvest(uint256 _pid) internal {
        /* ... */
                uint256 amount = pending.sub(referAmountLv1).sub(referAmountLv2);
                pool.rewardToken.transfer(msg.sender, amount.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = amount.mul(pool.percentLockReward).div(100);
                farmLock(msg.sender, lockAmount, _pid);
        /* ... */
    }

    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        /* ... */
                pool.rewardToken.transfer(referrerLv1, _referAmountLv1.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                farmLock(referrerLv1, lockAmount, _pid);
        /* ... */
                pool.rewardToken.transfer(referrerLv2, _referAmountLv2.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                farmLock(referrerLv2, lockAmount, _pid);
        /* ... */
            pool.rewardToken.transfer(devaddr, referAmountForDev.mul(100 -
pool.percentLockReward).div(100));
            uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
            farmLock(devaddr, lockAmount, _pid);
        }
    }
```

RECOMMENDED FIXES

Simplify the calculations as below:

```
    function updatePool(uint256 _pid) public {
        /* ... */
        if (forDev > 0) {
            uint256 lockAmount = forDev.mul(pool.percentLockReward).div(100);
            if (teamAddresses[_pid] != address(0)) {
                pool.rewardToken.transfer(teamAddresses[_pid], forDev.sub(lockAmount));
                farmLock(teamAddresses[_pid], lockAmount, _pid);
```

```
            } else {
                pool.rewardToken.transfer(devaddr, forDev.sub(lockAmount));
                farmLock(devaddr, lockAmount, _pid);
            }
        }
        /* ... */
    }

    function _harvest(uint256 _pid) internal {
        /* ... */
                uint256 amount = pending.sub(referAmountLv1).sub(referAmountLv2);
                uint256 lockAmount = amount.mul(pool.percentLockReward).div(100);
                pool.rewardToken.transfer(msg.sender, amount.sub(lockAmount));
                farmLock(msg.sender, lockAmount, _pid);
        /* ... */
    }

    function _transferReferral(uint256 _pid, uint256 _referAmountLv1, uint256
_referAmountLv2) internal {
        /* ... */
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                pool.rewardToken.transfer(referrerLv1, _referAmountLv1.sub(lockAmount));
                farmLock(referrerLv1, lockAmount, _pid);
        /* ... */
                uint256 lockAmount = _referAmountLv1.mul(pool.percentLockReward).div(100);
                pool.rewardToken.transfer(referrerLv2, _referAmountLv2.sub(lockAmount));
                farmLock(referrerLv2, lockAmount, _pid);
        /* ... */
            uint256 lockAmount = referAmountForDev.mul(pool.percentLockReward).div(100);
            pool.rewardToken.transfer(devaddr, referAmountForDev.sub(lockAmount));
            farmLock(devaddr, lockAmount, _pid);
        }
    }
```

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a*.

FIXED **LOW**

CONFUSED COMMENT

We found the following comment in the header of **_harvest** function in **BSCXNTS** contract, but the implementation code seems not to follow that comment.

```
    // lock 75% of reward if it come from bounus time
    function _harvest(uint256 _pid) internal {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        if (user.amount > 0) {
            uint256 pending =
user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user.rewardDebt);
            uint256 masterBal = pool.rewardToken.balanceOf(address(this));

            if (pending > masterBal) {
                pending = masterBal;
            }

            if(pending > 0) {
                uint256 referAmountLv1 = pending.mul(percentForReferLv1).div(100);
                uint256 referAmountLv2 = pending.mul(percentForReferLv2).div(100);
                _transferReferral(_pid, referAmountLv1, referAmountLv2);

                uint256 amount = pending.sub(referAmountLv1).sub(referAmountLv2);
                pool.rewardToken.transfer(msg.sender, amount.mul(100 -
pool.percentLockReward).div(100));
                uint256 lockAmount = amount.mul(pool.percentLockReward).div(100);
                farmLock(msg.sender, lockAmount, _pid);

                user.rewardDebtAtBlock = block.number;

                emit SendReward(msg.sender, _pid, amount, lockAmount);
            }

            user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
        }
    }
```

RECOMMENDED FIXES

Either edit/remove the comment, or update the **_harvest** function to satisfy the comment.

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a.*

FIXED **LOW**

TYPOS

- BSCXNTS.sol:

- Line 62: // Total allocation poitns. Must be the sum of all allocation points
- Line 458: require(_amount <= pool.rewardToken.balanceOf(address(this)), "ERC20: lock amount over blance");

> **Update: Jan 26, 2021** - BSCex fixed this in commit *85c544ca62eeb59c6c909c241958eb2c7ee01b2a.*

CONCLUSION

BSCex LaunchPoolX smart contracts have been audited by Verichains Lab using various public and in-house analysis tools and intensively manual code review. The assessment identified some issues and provided some suggestions in BSCex LaunchPoolX smart contracts code. Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

LIMITATIONS

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

APPENDIX I

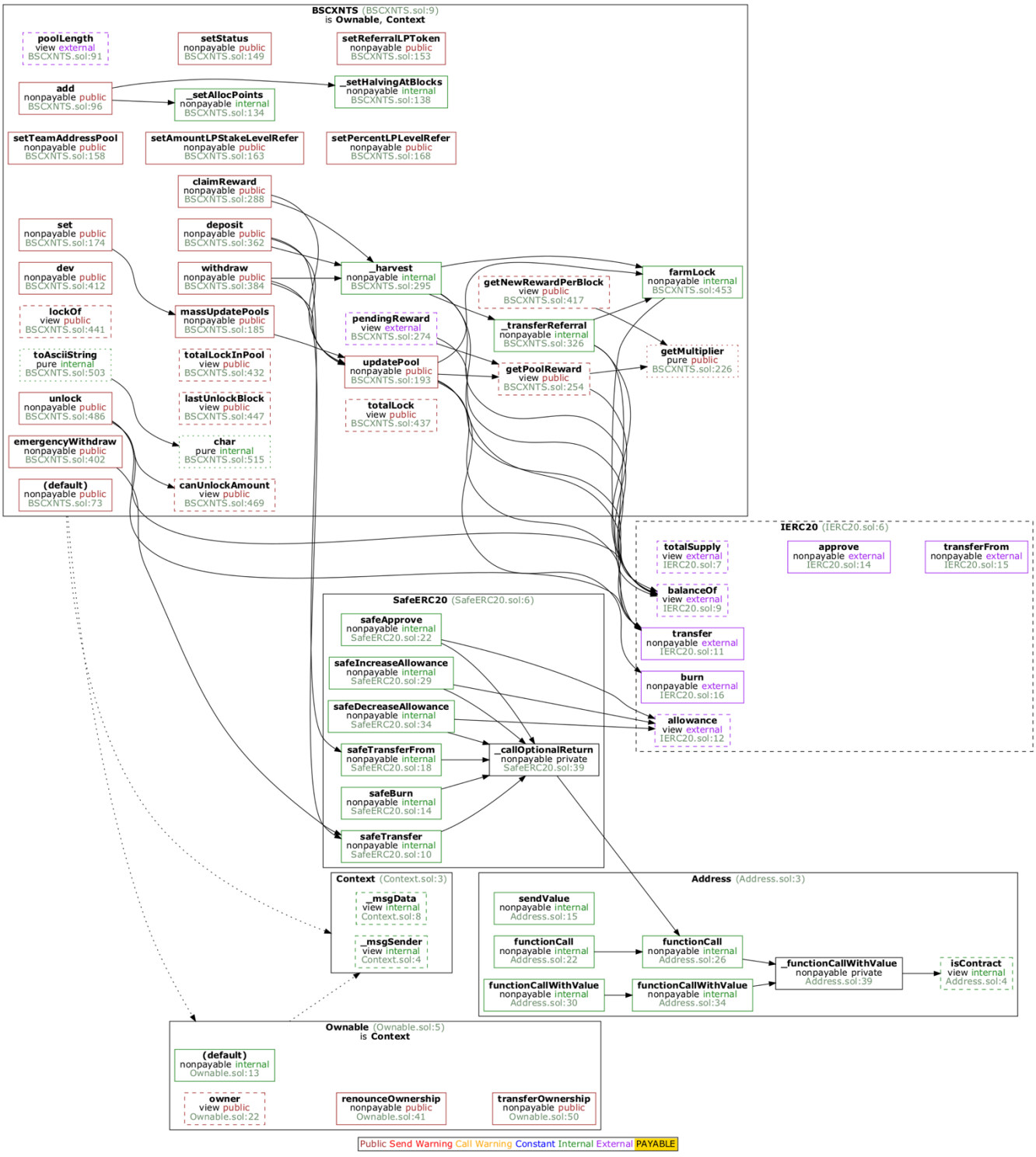

Figure 1: Call-graph of BSCXNTS contract