CONTRACT CHECKER
Blockchain Solutions

https://t.me/contractchecker
contact@contractchecker.app
contractchecker.app
Anywhere on the Blockchain

Date: 13.08.2023

Smart Contract Security Audit

XSALE Launchpad

Harry Redelman
General Manager

Project Information

Xsale is a decentralized protocol which is built upon the foundation of incentivising successful crypto startups across all major blockchains.

Audited Contract	Token Launcher
Web Site	https://www.xsalepad.finance/
Twitter	https://twitter.com/xsalepad
Telegram	https://t.me/xsalepad
Email	contact@xsalepad.finance
Platform	Ethereum Network
Language	Solidity
SC Link	Not deployed
Audited Codebase	xsale_launchpad_eth.sol

Audit Result

XSALE Launchpad Smart Contract has PASSED the smart contract audit with some recommendations.

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

Audit Result:	PASSED
Ownership:	Not renounced yet
KYC Verification:	NA at the date of report edition
Audit Date:	August 13, 2023
Audit Team:	CONTRACTCHECKER

Vulnerability Summary

Vulnerability	High	Medium	Low
Finalizing Presale Without Uniswap Pair		⚠	
Preventing Ether Transfers to Contract and Ensuring Contribution Tracking	⚠		
Usage of Ternary Operators for Conditional Expressions			✓
Missing Event Handling			✓

Important Notice for Investors

As the ContractChecker team, our primary objective is to conduct a comprehensive audit of the contract code to assess its functionality and identify any potential risks embedded within the code.

Before making any investment decisions, it is crucial to consider several factors. These include the ownership status, approach of the project team, marketing strategies, general market conditions, liquidity, token holdings, and other relevant aspects.

Investors should always exercise due diligence by conducting their own research and carefully managing their risk, considering the various factors that can impact the success of a project.

CONTRACT CHECKER

Table of Contents

CONTRACT CHECKER

Findings

Finalizing Presale Without Uniswap Pair

Issue Description:

The finalize() function within the PresaleContract may encounter a potential issue related to the finalization process when adding liquidity to a Uniswap pair. If there is no existing Uniswap pair for the specified token and ETH, the attempt to add liquidity will result in a transaction failure, causing the entire finalize() function to revert.

Recommendation:

It is recommended to enhance the finalize() function to include proper checks and handling mechanisms to address the absence of a Uniswap pair for the token. This can be achieved by:

a- Checking for the existence of a Uniswap pair using the Uniswap factory contract before attempting to add liquidity.
b- Emitting informative error messages to provide users with context in case of a transaction failure.
c- Optionally, adjusting the behavior of the finalize() function when no pair exists, such as distributing remaining tokens to participants or updating the contract state accordingly.

Code Example:

```
function finalize() onlyAdmin public {
    require(selfInfo.finalized == 0 && isCancelled == false, "Presale is
finalized or cancelled");
    // ...

    // Check if a Uniswap pair exists for the token
    IUniswapV2Factory uniFactory = IUniswapV2Factory(Router.factory());
    address pairAddress = uniFactory.getPair(selfInfo.addresses[0],
Router.WETH());
    require(pairAddress != address(0), "No Uniswap pair found for token");

    // Add liquidity to Uniswap pair
    address lock = addLiquidity(
        tokens,
        liquidity,
        selfInfo.addresses[0]
    );

    // ...
}
```

Preventing Ether Transfers to Contract and Ensuring Contribution Tracking

Description:

In the current contract implementation, there is a potential issue where users might accidentally or intentionally send ethers directly to the contract address instead of using the designated buy() function for contributions. This can lead to unintended behavior and loss of contributions since the contract does not properly handle direct ether transfers. Additionally, there is no mechanism to track these received ethers as contributions within the contract's functionality.

Recommendation:

It is recommended to Prevent Contract from Receiving Ethers or Track Received Ethers as Contribution

Option 1: Prevent Contract from Receiving Ethers

Fallback Function Enhancement:

Enhance the contract's fallback function (receive() or fallback()) to include a revert statement. This will reject any incoming ether transfers and provide clear feedback to users that direct ether transfers are not allowed.

Warning Message:

Emit a custom event or log a warning message in the fallback function to notify users about the restriction on direct ether transfers and guide them to use the designated buy() function for contributions.

Option 2: Track Received Ethers as Contribution

Fallback Function Enhancement:

Modify the fallback function to record incoming ether transfers in a mapping or data structure, associating the sender's address with the amount of received ethers.

Buy Function Integration:

Extend the buy() function to account for contributions made through direct ether transfers. If the sender's address is found in the mapping from step 1, add the received ether amount to their contribution and calculate token entitlement accordingly.

Event Logging:

Emit an event in both the fallback function and the modified buy() function to log direct ether transfers as contributions. Include the sender's address and the contributed ether amount.

Usage of Ternary Operators for Conditional Expressions

Description:

The ternary operators (?) are used for conditional expressions. While they can make the code concise, too many nested ternary operators can make the code less readable. Consider using regular if-else statements for improved readability, especially if the logic becomes more complex.

```
function getFee(address _user) public view returns (uint256) {
        uint256 tier = getTiers(_user);
        uint256 fee = tier == 1 ? feeForPoolTier1 : tier == 2
            ? feeForPoolTier2
            : feeForPoolTier3;
        return fee;
    }
```

Recommendation

It is recommended to consider using regular if-else statements for improved readability, especially if the logic becomes more complex.

CONTRACT CHECKER

Missing Event Handling

During our audit of the contract code, we identified several missing events that could enhance transparency, tracking, and auditability within the contract. We recommend implementing these events to provide a comprehensive log of actions and activities within the contract. Below are the missing events that we recommend adding:

```
contract PresaleContract {
    // ...

    event PoolEdited(string[] strings, string hash);
    event Claimed(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event TokensPurchased(address indexed user, uint256 amount);
    event PoolFinalized(address indexed poolAddress, uint256 tokensLocked);
    event PoolCancelled(address indexed poolAddress);
    event ForceCancelled(bool status);

    // ...

    function editPool(string[] memory strings, string memory hash) public
onlyAdmin {
        selfInfo.strings = strings;
        selfInfo._hash = hash;
        emit PoolEdited(strings, hash);
    }

    function claim() public {
        // ...

        emit Claimed(msg.sender, netEntitlement);
    }

    function withdraw() public payable {
        // ...

        emit Withdrawn(msg.sender, balance);
    }

    function buy() public payable {
        // ...

        emit TokensPurchased(msg.sender, tokens);
    }

    function finalize() onlyAdmin public {
        // ...
```

```
        emit PoolFinalized(selfInfo._address, tokens);
    }

    function cancel() public onlyAdmin {
        // ...

        emit PoolCancelled(selfInfo._address);
    }

    function forceCancel(bool _status) public onlySuperAdmin {
        isCancelled = _status;
        emit ForceCancelled(_status);
    }

    // ...
}
```

Executive Summary

CONTRACTCHECKER received an application on August 10, 2023, from the project team of XSALE to perform a thorough smart contract security audit. The objective was to identify any vulnerabilities present in the source code of XSALE Launchpad, as well as any dependencies within the contract. The audit process employed a combination of Static Analysis and Manual Review techniques, conducted by our expert team.

The audit primarily focused on the following considerations:

- Functionality Testing: The Smart Contract was subjected to rigorous testing to assess if the intended logic was followed consistently throughout the entire process.
- Line-by-Line Manual Examination: Our experts meticulously reviewed the code, examining each line in detail to identify any potential issues or vulnerabilities.
- Live Testing with Multiple Clients: The Smart Contract underwent live testing using a Testnet environment, involving multiple clients. This allowed for real-world usage scenarios to be simulated and evaluated.
- Failure Analysis: The preparations for potential failures were analyzed to determine how the Smart Contract would perform in the event of bugs or vulnerabilities.
- Library Version Analysis: The versions of all libraries utilized in the code were scrutinized to ensure they were up to date, reducing the risk of known vulnerabilities.
- On-chain Data Security Analysis: The security of on-chain data was thoroughly assessed to identify any weaknesses or potential risks associated with data storage and handling.

Furthermore, as part of the smart contract security audit, CONTRACTCHECKER conducted an in-depth review of the Token Launcher contract of XSale project, focusing on various key aspects. These included access control and authorization, input validation and sanitization, gas optimization and efficiency, event logging and error handling, as well as compliance with best practices and standards.

Through these comprehensive auditing procedures, CONTRACTCHECKER aimed to provide a detailed evaluation of the XSALE Launchpad smart contract security, highlighting any vulnerabilities and recommending appropriate measures for mitigation.

Overview

This audit report provides a comprehensive assessment of the overall security of the XSALE Launchpad smart contract source code. ContractChecker, a trusted security auditing firm, has conducted a thorough analysis of the contract, evaluating the system architecture and codebase to identify potential vulnerabilities, exploitations, hacks, and backdoors. The objective of this audit is to ensure the reliability, correctness, and robustness of the XSALE Launchpad smart contract.

Applied Methodology

The audit process employed by Contract Checker followed industry-leading practices and methodologies. Our expert team utilized a comprehensive approach, including the following key elements:

- Code Design Pattern Analysis: We conducted a thorough analysis of the code design patterns used in the smart contract. This analysis helped identify any design flaws or architectural weaknesses that could potentially impact the contract's security.
- Line-by-Line Inspection: Our experienced auditors performed a meticulous review of the smart contract's code, examining each line in detail. This process aimed to identify any coding errors, vulnerabilities, or potential security risks that could compromise the contract's integrity.
- Unit Testing Phase: We executed a robust unit testing phase, where specific units of the smart contract were tested individually to ensure their functionality and resilience. This phase helped identify and address any functional issues or discrepancies within the contract.
- Automated Testing: Contract Checker employed automated testing techniques to complement the manual review process. Automated tests were designed to simulate various scenarios and interactions with the contract, helping to identify potential vulnerabilities or weaknesses that may not be apparent through manual inspection alone.

These elements were integrated into our methodology to ensure a comprehensive assessment of the smart contract's security. By combining manual inspection, unit testing, and automated testing, we aimed to provide a holistic evaluation of the contract's reliability and correctness.

Security Assessment

During the assessment, Contract Checker scrutinized the smart contract for various security aspects, including but not limited to:

- Vulnerability Identification: We conducted a comprehensive scan of the contract codebase to identify any potential vulnerabilities that could expose the contract to unauthorized access, manipulation, or exploitation.

- Exploitation Analysis: Our team performed rigorous testing and analysis to assess the contract's resistance to common exploitation techniques, ensuring its robustness against potential attacks.

- Backdoor Detection: We meticulously examined the codebase to identify any backdoors or hidden functionalities that could compromise the security and integrity of the smart contract.

Sound Architecture

The smart contract incorporates a robust and efficient architecture. It follows a modular design, separating functionalities into distinct modules for reusability and scalability. Access control mechanisms ensure authorized operations, while optimized data structures minimize storage costs. The event-driven architecture enables real-time communication, and upgradeability allows for future enhancements. The contract's sound architecture reflects best practices, ensuring secure and efficient operations.

Note: While keeping the content concise, it's important to maintain clarity and ensure that essential information is conveyed effectively.

Code Correctness and Quality

The smart contract underwent a comprehensive review of its source code, with a primary focus on accuracy, readability, sections of high complexity, and the quantity and quality of test coverage. Contract Checker conducted a thorough assessment to ensure that the code is error-free, easily understandable, and properly tested. By examining these critical areas, the contract's overall code correctness and quality were evaluated, providing assurance of reliable and robust execution.

CONTRACT CHECKER

Risk Classification

Vulnerabilities are classified in 3 main levels as below based on possible effect to the contract.

High level vulnerability

Vulnerabilities on this level must be fixed immediately as they might lead to fund and data loss and open to manipulation. Any High-level finding will be highlighted with **RED** text

Medium level vulnerability

Vulnerabilities on this level also important to fix as they have potential risk of future exploit and manipulation. Any Medium-level finding will be highlighted with **ORANGE** text

Low level vulnerability

Vulnerabilities on this level are minor and may not affect the smart contract execution. Any Low-level finding will be highlighted with **BLUE** text

Manual Audit:

In the manual audit phase, our developers conducted a comprehensive line-by-line examination of the code. This meticulous process involved a thorough review of each line to identify any potential issues or vulnerabilities. To further validate the contract's functionality, we utilized Remix IDE's JavaScript VM and Kovan networks for testing. By combining manual code inspection with real-world testing environments, we aimed to ensure the accuracy and effectiveness of the smart contract.

Automated Audit

Remix Compiler Warnings

During the audit, the smart contract was tested using Solidity's compiler in Remix. While conducting the analysis, we found no issues or warnings reported by the compiler. This indicates that the contract's codebase complied with Solidity's syntax and best practices, further ensuring the contract's integrity and reliability.

Disclaimer

This report provides a limited overview of our findings based on our analysis, in accordance with industry best practices at the time of this report, regarding cybersecurity vulnerabilities and issues in the smart contract framework and algorithms, as detailed within this report. It is essential for you to read the full report to obtain a comprehensive understanding of our analysis. While we have conducted our analysis and prepared this report to the best of our abilities, it is important to note that you should not solely rely on this report and cannot hold us liable based on its content, production, or lack thereof. It is crucial for you to conduct your own independent investigations before making any decisions. We provide further details and clarification in the following disclaimer, which we advise you to read in its entirety.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to these terms, please discontinue reading this report, delete and destroy all downloaded and/or printed copies. This report is for informational purposes only and should not be relied upon for investment advice. No party shall have the right to rely on this report or its contents. ContractChecker and its affiliates, including holding companies, shareholders, subsidiaries, employees, directors, officers, and representatives (collectively referred to as "ContractChecker"), owe no duty of care to you or any other person and make no warranty or representation regarding the accuracy or completeness of the report. The report is provided on an "as is" basis, without any conditions, warranties, or other terms, except as explicitly stated in this disclaimer. ContractChecker hereby excludes all representations, warranties, conditions, and other terms that might have effect, but for this clause, in relation to the report, including, but not limited to, warranties of satisfactory quality, fitness for a particular purpose, and the use of reasonable care and skill. Except to the extent prohibited by law, ContractChecker excludes all liability and responsibility and disclaims any claims for any kind of loss or damage that may result from the use of this report, including, but not limited to, direct, indirect, special, punitive, consequential, or purely economic loss or damages, loss of income, profits, goodwill, data, contracts, use of money, or business interruption, regardless of whether the claim is based on delict, tort (including negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent), or otherwise, and regardless of the nature or jurisdiction of the claim. The security analysis is solely based on the smart contracts and does not cover the security of applications or operations. No product code has been reviewed. If you have any doubts about the authenticity of this document, please verify using the provided QR code.