verichains

SECURITY AUDIT OF

XELEB PROTOCOL

Public Report

Jul 02, 2025

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	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.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jul 02, 2025. We would like to thank the Xeleb for trusting Verichains Lab in auditing 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 Xeleb Protocol. 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.

During the audit process, the audit team had identified some issues in the source code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Xeleb Protocol

Xeleb is a next-generation platform empowering individuals and businesses to create, own, and monetize AI agents with real-world applications. By combining AI and Web3, Xeleb enables tokenized agent ownership, utility-driven interactions, and community-led growth.

Xeleb protocol introduces a decentralized token economy built around a bonding curve mechanism, allowing for the algorithmic pricing and distribution of its native token. Following an initial sales phase, it automatically provides liquidity to a PancakeSwapV3 pool, fostering a robust trading environment for its users. The ecosystem is further supported by a staking contract, which distributes a portion of the transaction fees to reward long-term token holders. This integrated approach aims to create a self-sustaining financial model that encourages both initial participation and sustained liquidity.

1.2. Audit Scope

This audit focused on identifying security flaws in code of the Xeleb Protocol.

It was conducted on commit `b74d6239487ae15e5288c6c8dec7ad71657b42d4` from git repository *https://github.com/xeleb-dev/Xeleb-Contract/tree/main/contracts*

The latest commit of the repository at the time of the audit was `1f9679f0b18e7c95df0bf77ab59d3f739b769865` and hash of files in the scope listed below:

SHA256 Sum	File
50aa11a4cacf7c69012e49c7f64eca5170e8654aa032c963228f8b220724d427	ArnoldAIerc20.sol
9bf0d7ef15d447c6b8b59bac50f6cf7e209092c19d480854036865ed40e41e1b	BondingCurve.sol
26669dc68eecfb4d58ac423d1627957f84ca26fc76421ee77d49ee2f2bf96d0e	CasanovaAIerc20.sol
aef8c7406221c17fae99cf12cbca7627dcc60359bc374af87d29461609781874	Cashier.sol
c1a9f4a9ac8b0cb5219533ec7a4d1b368f835f8174246e3fdfa7e130658a7ef7	CleopatraAIerc20.sol
a468901d9203639b4f2be709126a9b85995f54f178fd9126a7536462a75bb943	Controller.sol
e70aae20229717adfd1d791598bd4bb4381a90b0f0a17ce4f428d38d0ead6940	DoolittleAIerc20.sol
3e9dba18668387b50d6b38cddeba8cd82bf8e7160923555c651bd7c3905a892a	FreudAIerc20.sol
7a28451d5ce101530ebb1101046e9e36578607ab649af1c3d63d00ef1569675c	GamaAIerc20.sol
564871a656bc7a54c0eecf013cbcde2c675d15ca39b2d438c66eafaabb3048aa	GoatAIerc20.sol

c79ffef31a6a96783dfc397ebbcd28061e87d2f143bc957c32fa9a9e00f630b0	GreteAIerc20.sol
128feb9f3080c603f417e23708400433fa10beb4c0df9a77f111e28f29abcace	MarketPlace.sol
545f019d8d06d20e0600518bb955ed8361afd1463a3b3a651aca2662147433fa	MissAIerc20.sol
e6be4d32c9d21893d6f3ffbcce2917d9e5fdfb622761e1c5a62174db1015c6f2	NikolaAIerc20.sol
16c97d8249f0a5c8feea52ee008d4b02d8925056e29320ebf6bec22f8335f4f2	Staking.sol
8318e471d715f19122a767a1436315b51b29e21ffaf75361c98c158faf1d2b36	TokenERC20.sol
a5d0c2c15777289cf52787460cab0d1990d091df0387b18d8a56230b6bc1b641	Vesting.sol
dd59850b880a283212e43ef523b864da209449aae7f9a5a45291e6a9a7fe0240	WarrenAIerc20.sol
7a10d78af8b629bd09be162f8f6da3fed9f2572fa23f7280132ec8ba5e1e73e2	structs/VestingParam.sol

1.3. 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 were 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 as listed in table below, depending on their severity level:

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

Table 1. Severity levels

1.4. Disclaimer

Xeleb acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Xeleb understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Xeleb agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Xeleb will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Xeleb, the final report will be considered fully accepted by the Xeleb without the signature.

2. AUDIT RESULT

2.1. Overview

The code is written in `Solidity`, requiring a minimum version of `^0.8.20`.

2.1.1. Controller

The `Controller` contract acts as the central administrative authority for the protocol. It is responsible for managing key parameters and deploying core contracts.

- **Address Management:** Stores and provides the addresses for critical contracts, such as `Staking` and `Vesting`.
- **Fee Management:** Sets and manages the protocol's fee and burn percentages, which are used by the `BondingCurve` to calculate transaction costs.
- **Contract Deployment:** Deploys new `BondingCurve` and `TokenERC20` contracts, ensuring they are configured with the correct administrative and operational parameters.

2.1.2. BondingCurve

The `BondingCurve` contract manages the initial sale and distribution of the protocol's native token using an algorithmic price mechanism following the bonding curve formula.

- **Buy Function:** Users with a sufficient staking balance can buy the native token up to a specific cap. This limit can be bypassed with a signature from a protocol admin.
- **Sell Function:** Allows users to sell their native tokens back to the protocol in exchange for the base currency.
- **Liquidity Provision:** Once the sale period is complete, the contract automatically uses the collected base currency and remaining native tokens to provide liquidity to a PancakeSwap V3 pool.
- **Fee Distribution:** A portion of the fees from every buy and sell transaction is sent to the `Staking` contract for reward distribution.

2.1.3. TokenERC20

The `TokenERC20` contract is the template token for sale in the protocol. It has the standard ERC20 and ERC20Burnable logic. Before the bonding curve completes, the token is restricted from being transferred (except the transfer between the bonding curve and user address).

2.1.4. Staking

The `Staking` contract allows users to stake supported tokens to earn rewards and gain eligibility for token sales in `BondingCurve` contract.

- **Staking and Unstaking:** Users can deposit supported tokens to earn rewards and can withdraw them after a specified lock-up period.
- **Reward Calculation:** Stakers earn two types of rewards: a standard APY reward and a share of the protocol fees collected by the `BondingCurve`. The APY reward is distributed on a first-come, first-served basis from the reward pool balance. When the pool is depleted, the claimable amount is reduced to zero.
- **Eligibility:** Maintaining a staking balance is required for users to be eligible to purchase tokens from the `BondingCurve`.

2.1.5. Vesting

The `Vesting` contract manages the time-locked release of tokens for beneficiaries like team members or early investors, ensuring long-term token stability.

- **Vesting Schedules:** An admin can create vesting schedules that define the token amount, cliff period, and total vesting duration for a beneficiary.
- **Activation and Release:** Vesting schedules are typically created at the start of the bonding curve sale and are activated by the `Controller` once the sale is complete. Beneficiaries can then withdraw their vested tokens over time according to their schedule.
- **Emergency Withdraw:** In case of an emergency, admin can withdraw the tokens in the vesting contract.

2.2. Findings

#	Issue	Severity	Status
1	Malicious sqrtPrice when adding liquidity	CRITICAL	Fixed
2	Signature replay in `buy` function	LOW	Acknowledged
3	Front-running to get more fee reward	LOW	Acknowledged
4	Fee Token is burned during the `buy` and `sell` function	INFO	Fixed
5	Using SafeERC20 to avoid error with non-standard ERC20 token	INFO	Fixed

2.2.1. Malicious sqrtPrice when adding liquidity CRITICAL

- Affected file: `contracts/BondingCurve.sol`

When the bonding curve is completed, the contract will add liquidity to the PancakeSwapV3 pool with 100% slippage without any checks. Although the pool was set in the `initialize` function, there is no transferability limit on the token during the bonding curve process. An attacker can use the token bought from this step and `addLiquidity` to the pool to control the `sqrtPrice` of the pool. With a malicious `sqrtPrice`, an attacker can trick the contract into adding a small amount of the token alongside the full amount of the base token. The attacker can then sell some tokens to extract a large amount of the base token from the pool.

```
function _addLiquidity() internal {
    require(bondingComplete, "Bonding not complete");
    require(!isLiquidityAdded, "Liquidity already added");
    isLiquidityAdded = true;

    address pool = IPancakeV3Factory(PANCAKE_V3_FACTORY).getPool(
        address(token),
        BASE_TOKEN,
        POOL_FEE
    );
    //the pool is not checked before adding liquidity, the pool price can be different from
the sqrtPrice in the initialize function.
    require((pool != address(0)), "Pool not deployed!");

    uint finalBaseAmount = Cashier.balanceOf(BASE_TOKEN, address(this));
    if (Cashier.isNative(BASE_TOKEN)) {
        IWETH9(BASE_TOKEN).deposit{value: finalBaseAmount}();
    }

    (address token0, address token1) = address(token) < BASE_TOKEN
        ? (address(token), BASE_TOKEN)
        : (BASE_TOKEN, address(token));

    uint finalLiquiditySupply = token.balanceOf(address(this));
    if (finalLiquiditySupply > LIQUIDITY_SUPPLY) {
        finalLiquiditySupply = LIQUIDITY_SUPPLY;
    }

    (uint256 amount0Desired, uint256 amount1Desired) = address(token) <
        BASE_TOKEN
        ? (finalLiquiditySupply, finalBaseAmount)
        : (finalBaseAmount, finalLiquiditySupply);

    token.approve(NONFUNGIBLE_POSITION_MANAGER, finalLiquiditySupply);
    IERC20(BASE_TOKEN).approve(
        NONFUNGIBLE_POSITION_MANAGER,
        finalBaseAmount
    );

    uint256 tokenId;
```

```
    INonfungiblePositionManager.MintParams
        memory params = INonfungiblePositionManager.MintParams({
            token0: token0,
            token1: token1,
            fee: POOL_FEE,
            tickLower: MIN_TICK,
            tickUpper: MAX_TICK,
            amount0Desired: amount0Desired,
            amount1Desired: amount1Desired,
            amount0Min: 0,
            amount1Min: 0,
            recipient: address(this),
            deadline: block.timestamp + 15 minutes
        });

    (tokenId, , , ) = INonfungiblePositionManager(
        NONFUNGIBLE_POSITION_MANAGER
    ).mint(params);

    INonfungiblePositionManager(NONFUNGIBLE_POSITION_MANAGER).transferFrom(
        address(this),
        0x000000000000000000000000000000000000dEaD,
        tokenId
    );

    emit LiquidityAdded(
        address(this),
        token0,
        token1,
        pool,
        finalLiquiditySupply,
        finalBaseAmount,
        block.timestamp
    );
}
```

RECOMMENDATION

- Ensure the pool's `sqrtPrice` when adding liquidity is the same as the `sqrtPrice` in the `initialize` function. The token's transferability during the bonding curve process should be restricted, only allowing transfers between the user and the bonding curve contract.

UPDATES

- **Jul 01, 2025**: The issue was acknowledged and resolved by the Xeleb team. During the bonding curve process, all tokens are now non-transferable except for transfers between the user and the bonding curve contract.

2.2.2. Signature replay in `buy` function LOW

- Affected file: `contracts/BodingCurve.sol`

In the `buy` function, a user can buy more than the `MAX_BASE_BUY` amount with admin approval via a signature. However, the signature only expires after a certain time (set by the admin), which means an attacker can replay this signature multiple times before it expires.

Besides, the signature does not cover the address of the `BondingCurve` contract. This means an attacker can use the same signature to buy from other `BondingCurve` contracts that have the same admin. With the initial logic in the `Controller` contract, several `BondingCurve` contracts are deployed with the same admin hardcoded, so the likelihood of this issue is high.

```
function buy(
        uint256 _baseAmount,
        uint256 _bonusMaxAmount,
        uint256 signatureExpiredAt,
        bytes memory _signature
    ) external payable nonReentrant returns (uint256) {
        if (_bonusMaxAmount > 0) {
            require(block.timestamp < signatureExpiredAt, "signature expired");
            bytes32 _hash = keccak256(
                abi.encodePacked(
                    "buy",
                    msg.sender,
                    signatureExpiredAt, //the signature is only expired in X time, it means
the attacker can replay this signature multiple times before the signature is expired.
                    _bonusMaxAmount
                )
            );
            bytes32 messageHash = MessageHashUtils.toEthSignedMessageHash(
                _hash
            );
            address signer = ECDSA.recover(messageHash, _signature);

            require(signer == ADMIN_VERIFY_ADDRESS, "invalid signature");
            //the signature is not covered the addresss of the BondingCurve contract, it
means the attacker can use the same signature to buy from other BondingCurver contract
which have the same admin.
        }
        // Call getBaseTokenConfig once and destructure
        (
```

RECOMMENDATION

To prevent signature replay in the current contract, the team should add a nonce to the signature and increment the nonce for each user after a `buy` transaction is completed.

To prevent signature replay on other contracts, the team should include the `BondingCurve` contract's address in the signature.

UPDATES

- **Jul 01, 2025**: The issue has been acknowledged by the Xeleb team. The `buy` function now includes a capped limit for each user, and using signatures across multiple contracts is an intended feature.

2.2.3. Front-running to get more fee reward LOW

- Affected file: `contracts/Staking.sol`

There are two kinds of rewards in `Staking.sol`: APY and FeeDistribute. The `FeeDistribute` reward does not increase with the user's staking time; it only depends on the amount of the user's stake when the reward is deposited through the `receiveFeeDistribution` function. Therefore, an attacker doesn't need to stake for a long time. They can monitor the mempool and front-run a staking transaction before the `receiveFeeDistribution` transaction is sent to the mempool.

Given the withdrawal time limit in `Staking.sol` and that the `feeDistribute` amount from the `BondingCurve` contract is frequently called with a small amount, the impact of this issue is not high. However, if the team plans to use other kinds of rewards through the `receiveFeeDistribution` function, such as staking bonuses, the impact will be higher.

```
function receiveFeeDistribution() external payable {
        if (!distributeFeeToStakers) {
            adminFeeFund += msg.value;
            return;
        }
        require(address(XCX_TOKEN) != address(0), "XCX_TOKEN not set");
        if (msg.value == 0) return;
        TokenInfo storage info = tokenInfo[address(XCX_TOKEN)];
        require(isInitialized[address(XCX_TOKEN)], "XCX_TOKEN not initialized");

        if (info.totalStaked > 0) {
            uint256 rewardPerToken = (msg.value * PRECISION) / info.totalStaked;
            bnbFeeRewardPerToken += rewardPerToken;
            bnbFeeRewardPool += msg.value; //the bnbFeeRewardPool is update by the amount
of the msg.value, not the time.
        } else {
            adminFeeFund += msg.value;
        }
    }

  function _calculateReward(
        address token,
```

```
        address user
    )
        internal
        view
        returns (uint256 actualApyReward, uint256 actualBnbFeeReward)
    {
```

```
        if (token == address(XCX_TOKEN)) {
            bnbFeeReward =
                (userRecord.stakedAmount *
                    (bnbFeeRewardPerToken -
                        userLastBnbFeeRewardPerToken[user])) /
                PRECISION; //the reward based on the current bnbFeeRewardPerToken not the
time.
        }
```

UPDATES

- **Jul 01, 2025**: The issue has been acknowledged by the Xeleb team.

2.2.4. Fee Token is burned during the `buy` and `sell` function INFORMATIVE

- Affected file: `contracts/BondingCurve.sol`

In both the `buy` and `sell` functions, when the token is not the native token, the fee amount is transferred to a `dead` address, which corresponds to burning the fee amount. The team should verify this behavior to avoid the loss of fee revenue.

```
function buy(
        uint256 _baseAmount,
        uint256 _bonusMaxAmount,
        uint256 signatureExpiredAt,
        bytes memory _signature
    ) external payable nonReentrant returns (uint256) {
        (uint256 FEE_PERCENT, uint256 BURN_PERCENT) = IController(CONTROLLER)
                .getFeeAndBurnPercents();
            uint256 fee = (acceptedBaseAmount * FEE_PERCENT) / DEMI;
            uint256 baseUsed = acceptedBaseAmount - fee;

            if (fee > 0) {
                if (Cashier.isNative(BASE_TOKEN)) {
                    IStaking(STAKING_ADDRESS).receiveFeeDistribution{value: fee}();
                } else {
                    IERC20(BASE_TOKEN).transfer(
                        0x000000000000000000000000000000000000dEaD, //the fee token is
burned during the `buy` function.
                        fee
                    );
                }
            }
```

UPDATES

- **Jul 01, 2025**: The issue has been acknowledged by the Xeleb team. The team has updated the fee token logic and added additional fee collection mechanisms with the pool.

2.2.5. Using SafeERC20 to avoid error with non-standard ERC20 token INFORMATIVE

- Affected file: `contracts/BondingCurve.sol`, `contracts/Controller.sol`, `contracts/Vesting.sol`, `contracts/Cashier.sol`

In these contracts, the team uses the `IERC20` interface to transfer tokens. However, the `IERC20` interface is not safe for non-standard ERC20 tokens. The team should use the `SafeERC20` library to prevent errors.

```
 //Vesting.sol
function createVestingSchedule(
     address token,
     address beneficiary,
     uint256 amount,
     uint256 cliffPeriod,
     uint256 vestingTime,
     uint256 unlockPercent,
     bool isBondingCurve
 ) external onlyAdmin whenNotPaused {
     require(
         IERC20(token).transferFrom(msg.sender, address(this), amount), //the IERC20
interface is not safe with non-standard ERC20 token
         "Transfer to contract failed"
     );
     _addVestingSchedule(
         token,
         beneficiary,
         amount,
         cliffPeriod,
         vestingTime,
         unlockPercent,
         isBondingCurve
     );
 }
```

UPDATES

- **Jul 01, 2025**: The issue has been acknowledged and fixed by the Xeleb team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jul 02, 2025*	Public Report	Verichains Lab

Table 2. Report versions history