SECURITY AUDIT OF

THETAN ARENA SMART CONTRACTS

Public Report

Nov 19, 2021

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.
Solc	A compiler for Solidity.
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 prepared by Verichains Lab on Nov 19, 2021. We would like to thank the Wolffun 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 Thetan Arena 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.

During the audit process, the audit team had identified some vulnerable issues in the in the application, along with some recommendations. Wolffun has resolved and updated all the recommendations. Thetan Arena has passed with no Low, Medium, High, or Critical severity issues.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Wolffun

Wolffun is a game development studio founded in 2014 with specialty in Online PvP games for mobile platforms. We started off as a casual game developer with 4 members. Since 2015, we've been focusing on mid-core action genre and published "Tank Raid Online" – our first action title – in 2017. The game is loved by more than 5M players worldwide so far, a Google Editors' Choice and has also achieved many other prizes since its launch. Our second game – Heroes' Strike – is also on its way to come out. We are passionate about, and dedicated to making unique games that bring users the best experiences on mobile.

1.2. About Thetan Arena smart contracts

Thetan Arena is an esport game based on blockchain technology. You can gather your friends, form a team, battle with others and earn money with just your skills.

Thetan Arena smart contracts included ERC721 contract and some contracts which support to trade ERC721 tokens in the Thetan Arena game.

1.3. Audit scope

This audit focused on identifying security flaws in code and the design of the Thetan Arena smart contracts.

1.4. 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 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

1.5. Disclaimer

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.

2. AUDIT RESULT

2.1. Overview

The initial review was conducted on Oct 6, 2021 and a total effort of 7 working days was dedicated to identifying and documenting security issues in the code base of the Thetan Arena smart contracts.

The audit source code is on *https://github.com/WolffunGame/marketplace-smart-contracts*, commit process started on commit 930be41186c5c2773da4c66ec62ce1313cf0df7c.

2.2. Findings

The Thetan Arena smart contracts was written in Solidity language, with the required version to be ^0.8.0.

This section contains a detailed analysis of all the vulnerabilities that were discovered by the audit team during the audit process.

2.2.1. Marketplace.sol - Multiple issues related to the matchTransaction function CRITICAL

Consider the matchTransaction function below, this function accepts multiple parameters including seller address, buyer address, NFT contract address, payment token address, tokenId to buy, token price, native token boolean, percentage of the fee, and the signature signed by the seller.

```
function matchTransaction(
    address[4] calldata addresses,
    uint256[4] calldata values,
    bytes calldata signature,
    uint256 tokenId
) external returns (bool) {
    require(
        paymentTokens[addresses[3]] == true,
        "Marketplace: invalid payment method"
    );

    bytes32 criteriaMessageHash = getMessageHash(
        addresses[2],
        tokenId,
        addresses[3],
        values[1]
    );
```

```
  bytes32 ethSignedMessageHash = getEthSignedMessageHash(
      criteriaMessageHash
  );
  require(
      recoverSigner(ethSignedMessageHash, signature) == addresses[0],
      "Marketplace: invalid seller signature"
  );

  // check current ownership
  IERC721 nft = IERC721(addresses[2]);
  require(
      nft.ownerOf(values[0]) == addresses[0],
      "Marketplace: seller is not owner of this item now"
  );

  // If not native token. isApprovedForAll is required
  if (values[2] == 0) {
      require(
          nft.isApprovedForAll(addresses[0], address(this)),
          "Marketplace: seller did not approve market to transfer his i…
tem"
      );
  }

  // Check payment approval and buyer balance
  IERC20 paymentContract = IERC20(addresses[3]);
  require(
      paymentContract.balanceOf(addresses[1]) >= values[1],
      "Marketplace: buyer doesn't have enough token to buy this item"
  );
  require(
      paymentContract.allowance(addresses[1], address(this)) >= values[…
1],
      "Marketplace: buyer doesn't approve marketplace to spend payment …
amount"
  );

  // We divide by 10000 to support decimal value such as 4.25% => 425 /…
 10000
  uint256 fee = values[3].mul(values[1]).div(10000);
  uint256 payToSellerAmount = values[1].sub(fee);
```

```
    // transfer money to seller
    paymentContract.safeTransferFrom(
        addresses[1],
        addresses[0],
        payToSellerAmount
    );

    // transfer fee to address
    if (fee > 0) {
        paymentContract.safeTransferFrom(addresses[1], feeToAddress, fee);
    }

    // transfer item to buyer
    nft.safeTransferFrom(addresses[0], addresses[1], values[0]);

    // emit sale event
    emitEvent(addresses, values);
    return true;
}
```

Snippet 1. ThetanBoxHub.sol Multiple issues related to the matchTransaction function

After reviewing this function, we can see the above function contains some security vulnerabilities. We will list them all as below:

- Buyer address is got from the addresses[1] without security checking. So, the seller can sign a fake transaction with a random buyer and then submit that transaction. If the buyers have already approved all of their payment tokens, they will be forced to buy this NFT token without their permission. The buyer address must be got from the msg.sender (if the transaction is called by the buyer).
- The parameter tokenId and values[0] both refer to the same value (id of the token to buy). The tokenId variable is checked in with the signature, but the values[0] is the final value when sending the NFT token to the buyer. The dev team needs to review all the duplicated variables like in this case.
- The values[0] (token id) parameter is not included from the signature. So, the signature which allows buying an NFT token with a lower price can be used to buy another item with a higher price.
- The values[3] (fee percentage) parameter cannot be get from the function parameters or the seller signature. It must be set by the admin of this contract.
- Native token is not supported here, consider removing the values[2] (is native token) parameter.

- Signature can be reused since the signature blacklisting feature is not implemented. So, if the seller sell an NFT item and then buy them back in the market. When they sell it for the second time, the old signature (with old price) still can be used to buy this item again.

RECOMMENDATION

The logic of this function should be reviewed carefully, all the above issues should be fixed immediately.

UPDATES

- *2021-11-01*: Wolffun has fixed all the issues of the above function at commit 9ddc1ca5aa8df61d0b397e8f312c22246e1454ee.

2.2.2. ThetanBoxHub.sol - Buying box signatures can be reused multiple times MEDIUM

Current contract implementation in buyBoxWithSignature function doesn't have any mechanism to check if a signature is already used for buying box or not, so the caller can use the old signature to buy boxes token with a specific price.

In addition, the messageHash only composes _type, paymentErc20 and price so a function call with another useraddress can also use this signature to buy boxes.

```
function buyBoxWithSignature(
        uint256 boxId,
        uint256 _type,
        address userAddress,
        uint256 price,
        address paymentErc20,
        bytes calldata signature
    ) external onlyOwner {
        require(
            !isContract(userAddress),
            "ThetanBoxPayment: Only user address is allowed to buy box"
        );
        require(_type > 0, "ThetanBoxPayment: Invalid box type");
        require(price > 0, "ThetanBoxPayment: Invalid payment amount");

        bytes32 criteriaMessageHash = getMessageHash(
            _type,
            paymentErc20,
            price
        );

        bytes32 ethSignedMessageHash = getEthSignedMessageHash(
            criteriaMessageHash
        );

        require(
            recoverSigner(ethSignedMessageHash, signature) == userAdd…
ress,
            "ThetanBoxPayment: invalid seller signature"
        );

        IERC20 paymentToken = IERC20(paymentErc20);
        uint256 allowToPayAmount = paymentToken.allowance(
            userAddress,
            address(this)
        );
        require(
            allowToPayAmount >= price,
            "ThetanBoxPayment: Invalid token allowance"
        );
        // Transfer payment
        paymentToken.safeTransferFrom(
            userAddress,
            paymentReceivedAddress,
            price
        );
        // Emit payment event
        emit ThetanBoxPaid(boxId, userAddress, _type, price, paymentE…
rc20);
    }
```

Snippet 2. ThetanBoxHub.sol Buying box signatures can be reused multiple times

RECOMMENDATION

The messageHash must cover all variables which are used in the transaction. The contract should detect and reject used signatures.

UPDATES

- *2021-11-01*: Wolffun has fixed the issue at commit 9ddc1ca5aa8df61d0b397e8f312c22246e1454ee.

2.3. Additional notes and recommendations

2.3.1. ThetanBoxHub.sol - Inaccurate require message INFORMATIVE

Error message on require statement in buyBoxWithSignature function is inaccurate.

```
bytes32 ethSignedMessageHash = getEthSignedMessageHash(
            criteriaMessageHash
        );

         require(
            recoverSigner(ethSignedMessageHash, signature) == userAddress,
            "ThetanBoxPayment: invalid seller signature"
        );

        IERC20 paymentToken = IERC20(paymentErc20);
        uint256 allowToPayAmount = paymentToken.allowance(
            userAddress,
            address(this)
        );
```

Snippet 3. ThetanBoxHub.sol Inaccurate require message

RECOMMENDATION

Change message in require statement to "ThetanBoxPayment: invalid buyer signature".

UPDATES

- *2021-11-01*: Wolffun has fixed the issue at commit 9ddc1ca5aa8df61d0b397e8f312c22246e1454ee.

2.3.2. Avoid redefining code INFORMATIVE

During the audit process, we found that the contracts use a bunch of functions to verify signatures (getMessageHash, getEthSignedMessageHash, recoverSigner, splitSignature) but they are redefined in all contracts. It will be an inconvenience to change source code in the future if verifying signature mechanism is changed.

RECOMMENDATION

We suggest defining these functions in a library and inherits them in the contracts.

UPDATES

- *2021-11-01*: Wolffun has updated the contract at commit 9ddc1ca5aa8df61d0b397e8f312c22246e1454ee.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*2021-10-26*	Private Report	Verichains Lab
1.1	*2021-11-01*	Private Report	Verichains Lab
1.2	*2021-11-19*	Public Report	Verichains Lab

Table 2. Report versions history