SECURITY AUDIT OF

NAUTILUS SMART CONTRACT

Public Report

Mar 20, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.
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.
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Polygon	Polygon is a protocol and a framework for building and connecting Ethereum-compatible blockchain networks. Aggregating scalable solutions on Ethereum supporting a multi-chain Ethereum ecosystem.
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.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Mar 20, 2024. We would like to thank the Nautilus Assure 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 Nautilus Smart Contract. 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 identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY **5**

1.1. Audit scope **5**

1.2. Audit methodology **5**

1.3. Disclaimer **6**

1.4. Acceptance Minute **6**

2. AUDIT RESULT **7**

2.1. Overview **7**

2.1.1. NautilusEtherisc contract 7

2.1.2. NautilusInsurace contract 7

2.1.3. NautilusNeptune contract 8

2.2. Findings **8**

2.2.1. HIGH - The `createPolicy` and `withdrawCommission` functions of the contract `NautilusEtherisc` receives the ETH of a user but does not record it. 8

2.2.2. INFORMATIVE - Redundant Check in `BuyCovers` Function of `NautilusNeptune` Contract 10

2.2.3. INFORMATIVE - Some state change functions do not emit events. 12

3. VERSION HISTORY **13**

1. MANAGEMENT SUMMARY

1.1. Audit scope

This audit focused on identifying security flaws in code and the design of the Nautilus Smart Contract. It was conducted on commit `d89004c4e21cf60c3c5219a6467e7b032a3a546c` from git repository link: *https://github.com/nautilusAssure/smart-contracts*

The latest version of the following files were made available in the course of the review:

SHA-1 Sum	File
7d3fdb2816423e53d29be627c52f503ff0d12539	contracts/NautilusEtherisc.sol
e7fd483f9bad76c2b289592b09f6e5b5c34e0dcd	contracts/NautilusInsurace.sol
39b5451e1778e835e08084acec28c8653c190bcb	contracts/NautilusNeptune.sol
c4943a6cdc4812d947b0102196d17b150b505d76	contracts/InsuranceBaseProvider.sol

The Nautilus Assure team has been updated with the findings and recommendations. The team has acknowledged the issues and provided updates on the issues. The team has also responded to the findings and recommendations. The latest version of the code was reviewed on commit `fb2bd9f2a421d6a31aab5eda69b0a754526b6a5b`.

1.2. 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

1.3. Disclaimer

Nautilus Assure 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. Nautilus Assure understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Nautilus Assure 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.

1.4. Acceptance Minute

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

2. AUDIT RESULT

2.1. Overview

The `InsuranceBaseProvider` contract inherits from the `Initializable`, `OwnableUpgradeable`, `PausableUpgradeable`, and `ReentrancyGuardUpgradeable` contracts, thereby enabling proxy initialization, owner management, pausing functionality, and protection against reentrancy vulnerabilities.

The contracts, including NautilusEtherisc, NautilusInsurance, and NautilusNeptune, are structured to be upgradeable, enabling adjustments and enhancements over time. Each contract features an initialize function responsible for setting the initial values upon deployment.

2.1.1. NautilusEtherisc contract

The `NautilusEtherisc` contract extends the functionality of the `InsuranceBaseProvider` contract, providing users with features to create policies, withdraw commissions, and retrieve commission rates.

Internally, the contract interacts with the `DepegDistribution` contract to facilitate the creation of new policies for buyers. Before creating a policy, buyers must first approve a specific amount of `stableToken`.

The `createPolicy` function permits any user to transfer native tokens (ETH) into the contract, initiating the creation of a new policy.

In instances of exceptional circumstances, the contract owner has the authority to pause policy creation temporarily, while retaining the ability to withdraw commissions to the designated `escrowAddress`.

2.1.2. NautilusInsurace contract

The `NautilusInsurance` contract extends the functionality of the `InsuranceBaseProvider` contract, primarily enabling users to purchase insurance covers and access product details and coverage information.

In cases of exceptional circumstances, the contract owner has the ability to pause cover purchases temporarily to prevent users from buying covers.

When a user purchases a cover, the contract interacts with the `InsuranceCover` contract. The caller attaches their ETH value during the invocation of the `buyCovers` function.

2.1.3. NautilusNeptune contract

The `NautilusNeptune` contract extends the functionality of the `InsuranceBaseProvider` contract, primarily facilitating users to purchase insurance covers and retrieve policy premiums.

In exceptional situations, the contract owner can pause cover purchases to restrict user activity temporarily.

Before invoking the function, the caller must approve an amount of `stablecoin` token. Upon purchasing a cover, the contract transfers these tokens to the policy contract.

2.2. Findings

During the audit process, the audit team identified some vulnerable issues in the contract code.

#	Issue	Severity	Status
1	The `createPolicy` and `withdrawCommission` functions of the contract `NautilusEtherisc` receives the ETH of a user but does not record it.	HIGH	Fixed
2	Redundant Check in `BuyCovers` Function of `NautilusNeptune` Contract	INFORMATIVE	Fixed
3	Some state change functions do not emit events.	INFORMATIVE	Fixed

2.2.1. HIGH - The `createPolicy` and `withdrawCommission` functions of the contract `NautilusEtherisc` receives the ETH of a user but does not record it.

Affected files: `NautilusEtherisc.sol`

The `createPolicy` function in the `NautilusEtherisc` contract is designed to facilitate the creation of a new policy for a buyer. However, it's observed that although the function is marked as payable, it does not utilize or record the received ETH value from the user.

This discrepancy poses a potential risk as users may inadvertently lose their ETH without any corresponding record or action within the contract. Furthermore, the inability to compare the user's balance with any specified amounts passed exacerbates the issue, potentially leading to inconsistencies or unintended consequences.

To address this issue, appropriate modifications should be made to the createPolicy function to ensure proper handling and recording of received ETH values, thus enhancing transparency and mitigating the risk of user loss or confusion.

A similar issue in the `withdrawCommission` function allows the user to transfer an ETH value and lock it in. The user will lose their ETH.

```
function createPolicy(
    address buyer,
    address protectedWallet,
    uint256 protectedBalance,
    uint256 duration,
    uint256 bundleId
  ) external payable whenNotPaused nonReentrant returns (bytes32 processId) {
    processId = depegDistribution.createPolicy(
      buyer,
      protectedWallet,
      protectedBalance,
      duration,
      bundleId
    );

    emit CoverPurchased_Etherisc(
      bundleId,
      protectedWallet,
      duration,
      protectedBalance,
      0,
      msg.value
    );
}

function withdrawCommission(uint256 amount) external payable {
    depegDistribution.withdrawCommission(amount);

    require(stableToken.transfer(escrowAddress, amount), "Failed Withdrawal");
}
```

RECOMMENDATION

We recommend that the development team modify the `createPolicy` function to properly handle and record received ETH values, thereby enhancing transparency and mitigating the risk of user loss or confusion. Additionally, the `withdrawCommission` function should be reviewed and modified to prevent user loss of ETH.

In the shortly patch, removing the `payable` keyword from the `createPolicy` function and the `withdrawCommission` function is a temporary solution to prevent user loss of ETH.

UPDATES

- *Mar 18, 2024*: The Nautilus Assure team updated the `createPolicy` functions to remove the `payable` keyword and remain the issue in the `withdrawCommission`. This change will prevent users from losing their ETH when calling these functions.
- *Mar 20, 2024*: The Nautilus Assure team updated the `withdrawCommission` function to remove the `payable` keyword. Complete the fix for the issue.

2.2.2. INFORMATIVE - Redundant Check in `BuyCovers` Function of `NautilusNeptune` Contract

Affected files: `NautilusNeptune.sol`

The `buyCovers` function within the NautilusNeptune contract includes a redundant check related to the availability of the policy contract address. The policy contract address is dynamic, meaning it can change over time. Therefore, it's essential to verify the address's availability when calling functions of the policy contract.

However, this checking is duplicated in both the `buyCovers` and getPremium functions. Consequently, the check within the `buyCovers` function becomes unnecessary. Removing this redundant check can optimize gas costs while ensuring the same level of security and functionality.

By eliminating the duplicate check in the `buyCovers` function, gas consumption can be reduced without compromising the contract's integrity or functionality.

```
function buyCovers(IPolicy.PurchaseCoverArgs memory args) external whenNotPaused
nonReentrant {
    require(args.coverKey > 0, "Invalid key");
    require(args.coverDuration > 0 && args.coverDuration < 4, "Invalid duration");
    require(args.amountToCover > 0, "Invalid protection amount");

    IPolicy policy = getPolicyContract();
    require(address(policy) != address(0), "Fatal: Policy missing");

    IERC20Upgradeable stablecoin = getStablecoin();
    require(address(stablecoin) != address(0), "Fatal: Stablecoin missing");

    // Get fee info
    uint256 premium = getPremium(
      args.coverKey,
      args.productKey,
      args.coverDuration,
      args.amountToCover
    );

    // Transfer stablecoin to this contract
    stablecoin.safeTransferFrom(msg.sender, address(this), premium);

    // Approve protocol to pull the protocol fee
    stablecoin.safeIncreaseAllowance(address(policy), premium);

    args.onBehalfOf = msg.sender;

    // Purchase protection for this user
    policy.purchaseCover(args);

    // Emits cover purchased event
    emit CoverPurchased_Neptune(
      args.coverKey,
      args.productKey,
      msg.sender,
      args.coverDuration,
      args.amountToCover,
      args.referralCode,
      premium
    );
  }
}

function getPremium(
    bytes32 coverKey,
    bytes32 productKey,
    uint256 duration,
    uint256 protection
  ) public view returns (uint256 premium) {
    IPolicy policy = getPolicyContract();
    require(address(policy) != address(0), "Fatal: Policy missing");

    IPolicy.CoverFeeInfoType memory coverFeeInfo = policy.getCoverFeeInfo(
      coverKey,
      productKey,
      duration,
      protection
    );
    premium = coverFeeInfo.fee;
  }
```

RECOMMENDATION

We recommend removing the redundant check in the `buyCovers` function to optimize gas costs while maintaining the same level of security and functionality.

UPDATES

Mar 18, 2024: The Nautilus Assure team acknowledged the issue and removed the redundant check in the `buyCovers` function to optimize gas costs while maintaining the same level of security and functionality.

2.2.3. INFORMATIVE - Some state change functions do not emit events.

Affected files: `NautilusEtherisc.sol`

The state of smart contracts can be changed by calling functions. It is important to emit events when the state of the contract is changed. This allows external applications to listen for these events and react accordingly.

Following setters of `NautilusEtherisc`:

- setDepegDistribution
- setStableToken
- setEscrow

```
function _setEscrow(address _escrowAddress) internal {
    require(_escrowAddress != address(0), "Escrow address cannot be zero address.");
    escrowAddress = _escrowAddress;
}
function _setStableToken(address _tokenAddress) internal {
    require(_tokenAddress != address(0), "Stable token cannot be zero address.");
    stableToken = IERC20Upgradeable(_tokenAddress);
}
function _setDepegDistribution(address _depegDistributionAddress) internal {
    require(_depegDistributionAddress != address(0), "Depeg Distribution cannot be zero
address.");
    depegDistribution = DepegDistribution(_depegDistributionAddress);
}
```

RECOMMENDATION

We suggest that the development team consider emitting events for these state-changing functions to enhance transparency and facilitate external monitoring and integration.

UPDATES

Mar 18, 2024: The Nautilus Assure team acknowledged the issue and added events for the state-changing functions to enhance transparency and facilitate external monitoring and integration.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 14, 2024*	Public Report	Verichains Lab
1.1	*Mar 18, 2024*	Public Report	Verichains Lab
1.2	*Mar 20, 2024*	Public Report	Verichains Lab

Table 2. Report versions history