BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 15 JAN 2024

PREPARED FOR: ZERO LIQUID

Contents

1 Executive Summary

Mittaria engaged BlockApex for a security review of their protocol's features. A team of two security researchers reviewed the source code for the Presale and NFT smart contract for 7-days of effort. The details of project scope, complexity and coverage are provided in the the following sections of this report.

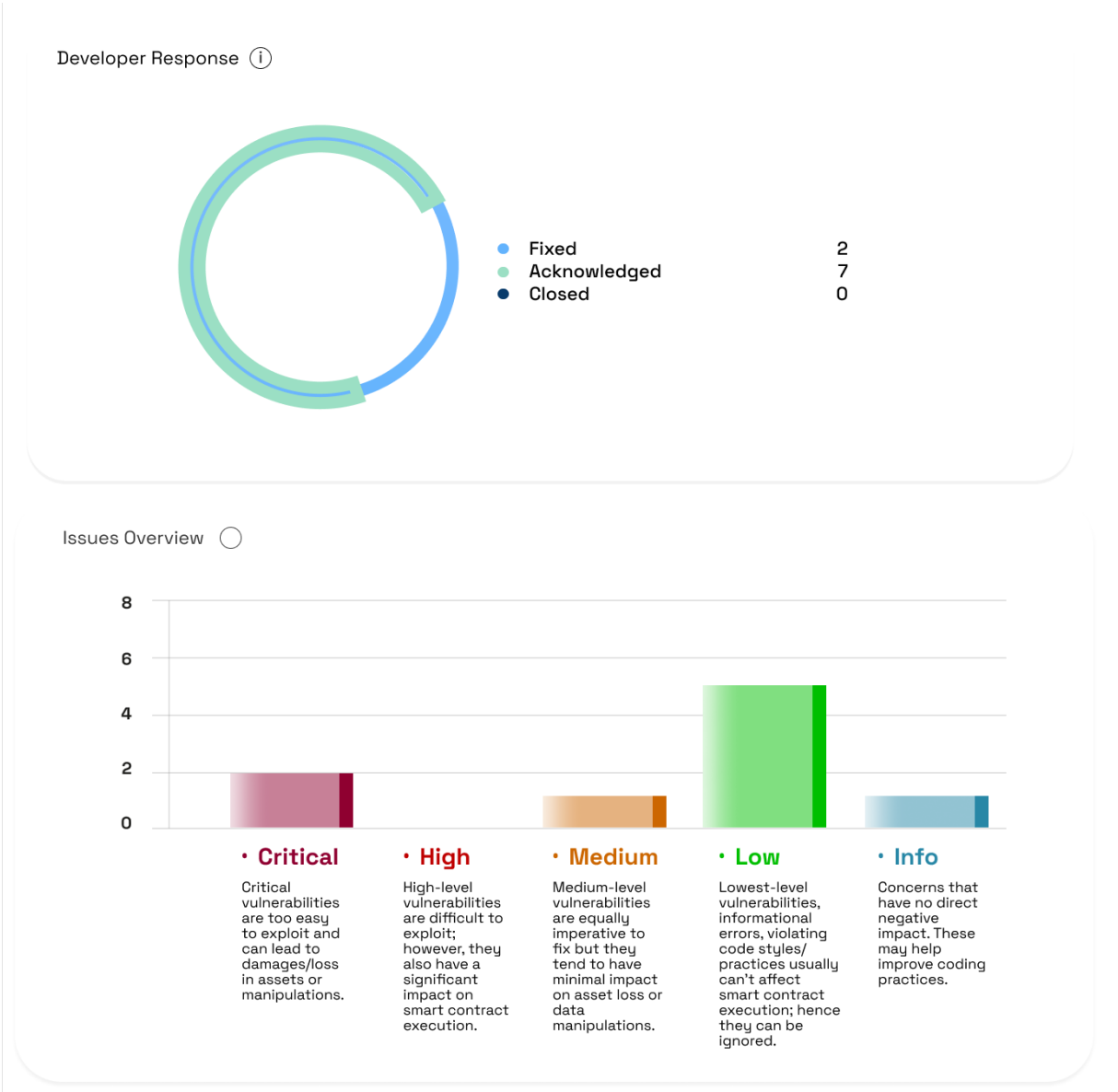

Figure 1: Executive Summary

1.1 Scope

1.1.1 In Scope

1.2 Smart Contracts Overview

MittariaPresale: Presale is a Mittaria protocol's ICO feature implemented as a single smart contract written in Solidity. The contract itself comprises of two externally visible methods facilitating users to interact with assets. Key functionalities include purchase and refund, allowing users to purchase and get refunded for assets bought during the presale launch, token distribution.

MittariaWTG: WTG is a Mittaria protocol's NFT Smart Contract implemented in Solidity. NFTs can be minted via this smart contract's available functions mint and mintTo for users willing to purchase for a fixed price as per the phase. The concept of phases allows sale of tokens over flexible periods as per the business decision of the Mittaria protocol.

Risk Profiling

- **Centralization Factor**: During the assessment of MittariaPresale and the MittariaWTG smart contracts, it was observed that a number of critical functionalities have centralized control from either the owner of the protocol or a backend server used to generate signatures and merkleproofs for the users willing to interact with both smart contracts. *Concerns regarding the centralization risks of the Mittaria Protocol Smart Contracts under review have been conveyed to the client.*

Files in scope:

1. contracts/custom-wtg/Presale.sol
2. contracts/custom-wtg/WTG.sol

Initial Audit Commit Hash: 314d6dcd859689ea8faff6dee1840172eede9ed2

Final Audit Commit Hash: 95f05d97352881f723c3928195301ea987d39a08

1.2.1 Out of Scope

Any features or functionalities not explicitly mentioned in the "In Scope" section are considered outside the scope of this security review.

1.3 Methodology

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning 7 days. Starting with the recon phase, a basic understanding of the code was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/ whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles and best practices.

1.4 Status Description

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

2 Summary of Findings

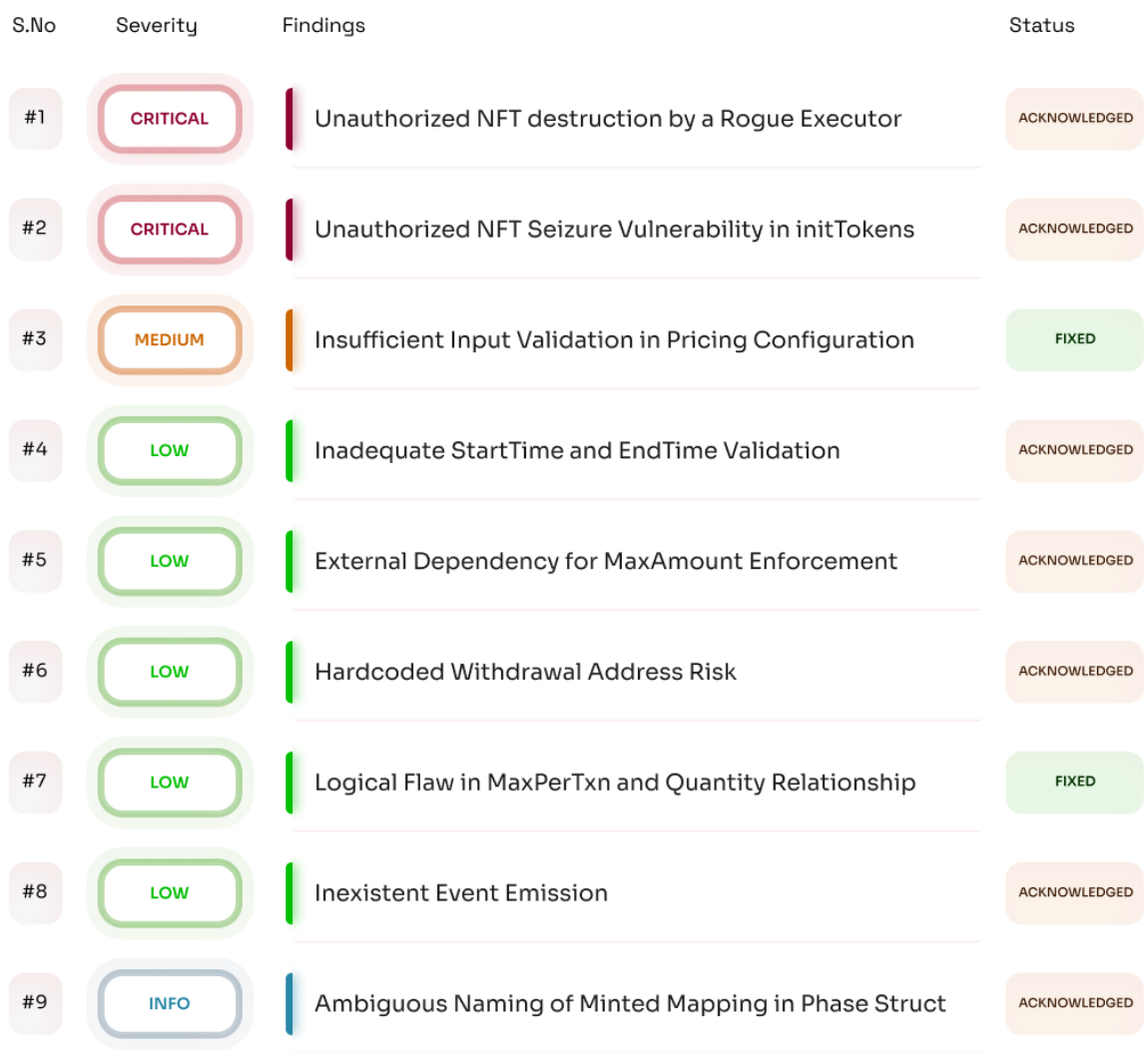

Figure 2: Findings Summary

3 Findings and Risk Analysis

3.1 Unauthorized NFT destruction by a Rogue Executor

Severity: Critical

Status: Acknowledged

Location :

- burn() - Link

Description :

The NFT contract includes a function burnToken that enables an allowed executor, as designated by the contract owner, to burn any NFT token. This function calls _burn(tokenId, false), which bypasses the approval check, allowing the executor to burn tokens regardless of their ownership. This represents a significant security flaw, as it undermines the ownership and control of NFT holders.

Impact :

This vulnerability severely impacts the integrity of the NFT ecosystem by enabling unauthorized destruction of assets. The burnToken function's ability to destroy tokens without owner consent directly contradicts the principles of ownership and trust in the NFT space. It allows for the potential malicious destruction of assets, undermining the value and reliability of the entire system.

Proof of Concept :

The provided PoC demonstrates how an executor, set by the contract owner, can burn a token regardless of its current owner. In the scenario, five different accounts mint tokens, but the contract owner (or an executor set by the owner) is able to burn any one of these tokens without the consent of its owner.

```
    function testBurn() public {
          address[] memory _executors = new address[](1);
          _executors[0] = exec1;


          vm.prank(owner);
          nftContract.setExecutor(_executors, true);


          MittariaWtg.Configs memory _configs = MittariaWtg.Configs({
              quantity: 10,
              maxPerTxn: 5,
              startTime: uint32(block.timestamp + 10 seconds),
              endTime: uint32(block.timestamp + 10 minutes),
              price: 0.1 ether
          });


          vm.prank(exec1);
          nftContract.updateMintingPhase(0, _configs);
```

```
        for (uint256 i = 0; i &lt; users.length; i++) {
            vm.warp(block.timestamp + 1 minutes);
            address account = users[i];
            vm.prank(account);
            deal(account, 10 ether);
            nftContract.mintTo{value: 0.1 ether}(account, 1);
        }


        nftContract.balanceOf(users[2]);


        nftContract.ownerOf(2);


        vm.prank(owner);
        nftContract.burnToken(2);
    }
```

Code Affected:

```
    function burnToken(uint256 tokenId) external onlyAllowedExecutor {
        _burn(tokenId);
    }

  /**
   * @dev Equivalent to `_burn(tokenId, false)`.
   */
  function _burn(uint256 tokenId) internal virtual {
    _burn(tokenId, false);
  }

  /**
   * @dev Destroys `tokenId`.
   * The approval is cleared when the token is burned.
   *
   * Requirements:
   *
   * - `tokenId` must exist.
   *
   * Emits a {Transfer} event.
   */
  function _burn(uint256 tokenId, bool approvalCheck) internal virtual {
    TokenOwnership memory prevOwnership = _ownershipOf(tokenId);


    address from = prevOwnership.addr;


    if (approvalCheck) {
      bool isApprovedOrOwner = (_msgSender() == from || isApprovedForAll(from, _msgSender()
          ) || getApproved(tokenId) == _msgSender());


      if (!isApprovedOrOwner) revert TransferCallerNotOwnerNorApproved();
    }
```

```
      _beforeTokenTransfers(from, address(0), tokenId, 1);


      // Clear approvals from the previous owner
      _approve(address(0), tokenId, from);


      // Underflow of the sender&#x27;s balance is impossible because we check for
      // ownership above and the recipient&#x27;s balance can&#x27;t realistically overflow.
      // Counter overflow is incredibly unrealistic as tokenId would have to be 2**256.
      unchecked {
        AddressData storage addressData = _addressData[from];
        addressData.balance -= 1;
        addressData.numberBurned += 1;


        // Keep track of who burned the token, and the timestamp of burning.
        TokenOwnership storage currSlot = _ownerships[tokenId];
        currSlot.addr = from;
        currSlot.startTimestamp = uint64(block.timestamp);
        currSlot.burned = true;


        // If the ownership slot of tokenId+1 is not explicitly set, that means the burn
            initiator owns it.
        // Set the slot of tokenId+1 explicitly in storage to maintain correctness for
            ownerOf(tokenId+1) calls.
        uint256 nextTokenId = tokenId + 1;
        TokenOwnership storage nextSlot = _ownerships[nextTokenId];
        if (nextSlot.addr == address(0)) {
          // This will suffice for checking _exists(nextTokenId),
          // as a burned slot cannot contain the zero address.
          if (nextTokenId != _currentIndex) {
            nextSlot.addr = from;
            nextSlot.startTimestamp = prevOwnership.startTimestamp;
          }
        }
      }


      emit Transfer(from, address(0), tokenId);
      _afterTokenTransfers(from, address(0), tokenId, 1);


      // Overflow not possible, as _burnCounter cannot be exceed _currentIndex times.
      unchecked {
        _burnCounter++;
      }
    }
```

Recommendation :

Restrict the burnToken Function: This function should be modified to ensure that only the token owner or an authorized party can burn the token. Implementing a requirement for owner consent or approval would mitigate this risk.

Developer Response

We will becareful of admin and executor accounts.

Auditors Response

Acknowledged

3.2 Unauthorized NFT Seizure Vulnerability in initTokens

Severity: Critical

Status: Acknowledged

Location :

- initTokens() - Link

Description :

The NFT contract contains a critical vulnerability within its initTokens() function. This function, when invoked by the contract owner, forcibly transfers the first 5 NFTs to the owner's address without requiring consent or acknowledgement from the current token holders. This action is a severe breach of trust and decentralization principles in blockchain applications.

Impact :

This vulnerability directly undermines the ownership rights of NFT holders and allows the contract owner to seize assets at will. The impact is substantial as it allows the contract owner to unilaterally and forcibly transfer ownership of NFTs from legitimate owners to themselves. This not only breaches the trust of the users but also jeopardizes the integrity of the entire NFT ecosystem associated with this contract.

Proof of Concept :

The provided PoC demonstrates a scenario where after the NFTs are minted by users, the contract owner can execute the initTokens() function to transfer the first 5 NFTs to their address. This is achieved without the consent of the NFT owners, exploiting the centralization flaw in the initTokens() function.

```
        function testUsersCanLoseNFTsByForce() public {
            address[] memory _executors = new address[](1);
            _executors[0] = exec1;


            vm.prank(owner);
            nftContract.setExecutor(_executors, true);


            MittariaWtg.Configs memory _configs = MittariaWtg.Configs({
                quantity: 10,
                maxPerTxn: 5,
                startTime: uint32(block.timestamp + 10 seconds),
                endTime: uint32(block.timestamp + 10 minutes),
                price: 0.1 ether
            });


            vm.prank(exec1);
            nftContract.updateMintingPhase(0, _configs);
```

```
        for (uint256 i = 0; i &lt; users.length; i++) {
            vm.warp(block.timestamp + 1 minutes);
            address account = users[i];
            vm.prank(account);
            deal(account, 10 ether);
            nftContract.mintTo{value: 0.1 ether}(account, 1);
        }


        for (uint256 i = 1; i &lt; 6; i++) {
            nftContract.ownerOf(i);
        }

        vm.prank(owner);
        nftContract.initTokens();


        for (uint256 i = 1; i &lt; 6; i++) {
            nftContract.ownerOf(i);
        }
    }
```

Code Affected

```
    function initTokens() public onlyOwner {
        require(!tokenInited, &quot;Already inited&quot;);
        for (uint8 i = 1; i &lt;= 5; i++) {
            address tokenOwner = ownerOf(i);
            _approve(msg.sender, i, tokenOwner);
            _transfer(tokenOwner, msg.sender, i);
        }
    }
}

 /**
  * @dev Approve `to` to operate on `tokenId`
  *
  * Emits a {Approval} event.
  */
 function _approve(address to, uint256 tokenId, address owner) internal {
   _tokenApprovals[tokenId] = to;
   emit Approval(owner, to, tokenId);
 }
```

Recommendation :

Remove or Restrict the initTokens Function: Consider removing this function entirely, or implementing strict checks and balances. If this function is essential for some administrative purpose, it should be redesigned to require explicit approval from the NFT owners before transferring tokens.

Developer Response

We had to claim the first 5 tokens due to the previous contract, this method is only executed once

Auditors Response

Acknowledged

3.3 Insufficient Input Validation in Pricing Configuration

Severity: Medium

Status: Fixed

Location :

1. `uint128 price;` - contracts/custom-wtg/Presale.sol#L32
2. `uint128 price;` - contracts/custom-wtg/WTG.sol#L22

Description The `Configs` struct includes a price variable intended to set the asset's purchase `price`. However, the `_validateMintingPhase` method fails to validate this price for non-zero values, allowing the creation of phases where assets can be purchased for free.

Impact This oversight can lead to significant financial losses for the contract owner and undermines the intended economic mechanics of the asset distribution. It can also result in an unfair and uncontrolled distribution of assets, affecting the integrity of the entire system.

Proof of Concept

```
    struct Configs {
        // ...
        uint128 price;
    }
```

Recommendation Implement a validation check within `_validateMintingPhase` to ensure that price is `greater than zero`. This prevents zero-cost minting and maintains the economic balance intended by the contract.

3.4 Inadequate StartTime and EndTime Validation

Severity: Low

Status: Acknowledged

Location :

1. _validateMintingPhase() - contracts/custom-wtg/Presale.sol#L160
2. _validateMintingPhase() - contracts/custom-wtg/WTG.sol#L163-L164

Description The `_validateMintingPhase` method checks that `startTime` is greater than zero, but does not compare it against the current `block timestamp`. This allows for the creation of phases that start in the past, rendering the `endTime` check ineffective.

Impact Such a configuration can lead to all purchase transactions reverting, as the time checks become illogical. This could put the contract into a state of denial-of-service (DoS), preventing any meaningful interaction and disrupting intended functionalities.

Proof of Concept

```
      // verifed
      function _validateMintingPhase(Configs calldata _configs) internal pure {
        require(_configs.quantity &gt; 0, &quot;Invalid quantity&quot;);
        require(_configs.maxPerTxn &gt; 0, &quot;Invalid max per txn&quot;);
        require(_configs.startTime &gt; 0, &quot;Invalid start time&quot;);
        require(_configs.endTime &gt; _configs.startTime, &quot;Invalid end time&quot;);
      }
```

Recommendation Enhance validation to ensure that `startTime` is not only `greater than zero` but also `greater than or equal to the current block timestamp`. Additionally, ensure `endTime` is always `greater than startTime`.

Developer Response

We will be careful with setting up start / end times

Auditors Response

Acknowledged with no further rebuttal.

3.5 External Dependency for MaxAmount Enforcement

Severity: Low

Status: Acknowledged

Location :

1. WTG.sol - _verifySignature(... `uint256 _maxAmount`) - Link
2. Presale.sol - _verifySignature(... `uint256 _maxAmount`) - Link

Description The contract relies on an external server to handle and enforce `_maxAmount` per user, creating a potential desynchronization between the contract's state and external controls. This reliance can be exploited for unauthorized purchases.

Impact The desynchronization could lead to unauthorized asset purchases, breaking the contract's intended business logic and potentially causing financial and reputational damage.

Proof of Concept :

Presale.sol:

1. purchase() -

```
    function purchase(uint32 _phaseId, uint32 _amount, uint32 _maxAmount, bytes calldata
        _signature) external payable {
```

2. refund() -

```
    function refund(uint32 _phaseId, uint32 _amount, uint32 _maxAmount, bytes32[] calldata
        _merkleProofs) external nonReentrant {
```

WTG.sol:

```
    function mint(uint256 _phaseId, uint16 _amount, uint256 _maxAmount, bytes calldata
        _signature) external payable {
```

Recommendation Implement logic within the contracts to enforce maxAmount per user, reducing reliance on external systems and enhancing the contract's integrity.

Developer Response

Flexible amount is a business requirement, ignored

Auditors Response

Acknowledged

3.6 Hardcoded Withdrawal Address Risk

Severity: Low

Status: Acknowledged

Location :

Presale.sol:

- `withdraw()`: contracts/custom-wtg/Presale.sol#L201
- `address public verifier`: contracts/custom-wtg/Presale.sol#L43

WTG.sol:

- `withdraw()`: contracts/custom-wtg/WTG.sol#L256-L260
- `verifier`: contracts/custom-wtg/WTG.sol#L57

Description The `withdraw` function transfers funds to a hardcoded address. This design is insecure and against industry best practices. Similarly, the immutable `verifier` address poses a similar risk.

Impact The loss of keys to these hardcoded addresses could lead to funds being frozen or stolen, undermining the trust in the protocol and the security of the funds.

Proof of Concept :

Presale:

```
    address public verifier = 0x9f6B54d48AD2175e56a1BA9bFc74cd077213B68D;

  ...

    function withdraw() public onlyAllowedExecutor {
      uint256 balance = address(this).balance;
      (bool success, ) = 0x961E3f37Ce55799E27C1E8CeE24305fccFE7591C.call{ value: balance }(&
          quot;&quot;);
      require(success, &quot;Transfer failed.&quot;);
    }
```

MittariaWTG:

```
    function initialize() public onlyOwner {
      ...
      verifier = 0x9f6B54d48AD2175e56a1BA9bFc74cd077213B68D;
      ...
    }

  ...

    function withdraw() public onlyOwner {
      uint256 balance = address(this).balance;
      (bool success, ) = address(0x961E3f37Ce55799E27C1E8CeE24305fccFE7591C).call{ value:
          balance }(&quot;&quot;);
      require(success, &quot;Transfer failed.&quot;);
    }
```

Recommendation Implement setter functions for these addresses, allowing for updates if needed. Ensure these functions are protected with appropriate access control.

Developer Response

There will be no funds in this contract, withdrawal address is confirmed multiple times

Auditors Response

Acknowledged

3.7 Logical Flaw in MaxPerTxn and Quantity Relationship

Severity: Low

Status: Fixed

Location :

1. _validateMintingPhase() - contracts/custom-wtg/Presale.sol#L157-L162
2. _validateMintingPhase() - contracts/custom-wtg/WTG.sol#L160-L165

Description The `_validateMintingPhase` method does not enforce a logical relationship between `maxPerTxn` and `quantity`, allowing configurations where `maxPerTxn` is `equal to or greater than quantity` .

Impact This flaw can enable a single transaction to exhaust the entire phase's allocation, negating the intended limits per transaction and potentially leading to unfair distribution.

Proof of Concept

```
    // verifed
    function _validateMintingPhase(Configs calldata _configs) internal pure {
      ...
      require(_configs.maxPerTxn &gt; 0, &quot;Invalid max per txn&quot;);
      ...
    }
```

Recommendation Introduce a validation step to ensure `maxPerTxn` is strictly `less than quantity`, maintaining the intended limitations of the minting process.

3.8 Inexistent Event Emission

Severity: Low

Status: Acknowledged

Location :

1. WTG.sol - Link

Description :

A number of state-altering functions within the `MittariaWtg` contract lack event emissions. Event emissions are crucial in Ethereum contracts for tracking changes of state, especially in a decentralized context where transparency and traceability are paramount. The absence of events in these functions could hinder the ability to monitor and verify contract interactions effectively.

Impact :

The lack of event emissions in these functions may lead to difficulties in tracking contract interactions on-chain. This can make it challenging for users and external systems to monitor changes or detect potential issues or irregularities. While this issue does not pose an immediate security risk, it significantly impacts the contract's transparency and traceability.

Proof of Concept ### Code Affected:

The functions identified below, which alter the contract's state but do not emit events, are affected.

```
    initialize, mint, mintTo, _mintToken, setTotalSupply, burnToken, setExecutor, setVerifier,
         toggleTokenURI, toggleBackupURI, toggleHtmlURI, toggleReveal, setPreRevealedURI,
        setBaseURI, setBackupURI, setHtmlURI, setTokensURI, adminMintTo, withdraw, setName,
        setSymbol, initTokens
```

Recommendation :

It is recommended to introduce relevant event emissions in all state-altering functions. Each event should capture key information pertinent to the state change, ensuring that users and external observers can effectively track and understand the contract's operations.

Developer Response

Since events are not exposed / required for any third parties

Auditors Response

Acknowledged

3.9 Ambiguous Naming of Minted Mapping in Phase Struct

Severity: Info

Status: Acknowledged

Location :

- `minted` - contracts/custom-wtg/Presale.sol#L39

Description The `minted` mapping in the `Phase` struct is misleading as it tracks the number of assets minted to a user. This naming could cause confusion, especially in the context of ERC721 and ERC1155 assets.

Impact The ambiguous naming can lead to misunderstandings about the functionality of the mapping, potentially causing errors in maintenance or further development of the contract.

Proof of Concept

```
struct Phase {
  ...
  mapping(address =&gt; uint32) minted;
}
```

Recommendation Rename the minted mapping to a more descriptive term, such as mintedTo, to clearly convey its purpose and improve code clarity.

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.