BlockApex

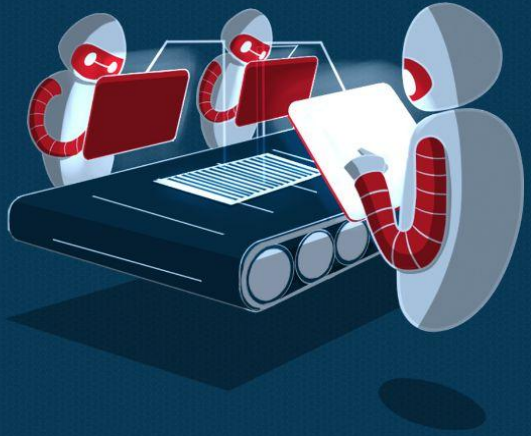
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



PREFACE

Objectives

This document aims to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and the client's intellectual property. It also includes information on potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of the information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly to aid our growing blockchain community; at the client's discretion.

Key understandings

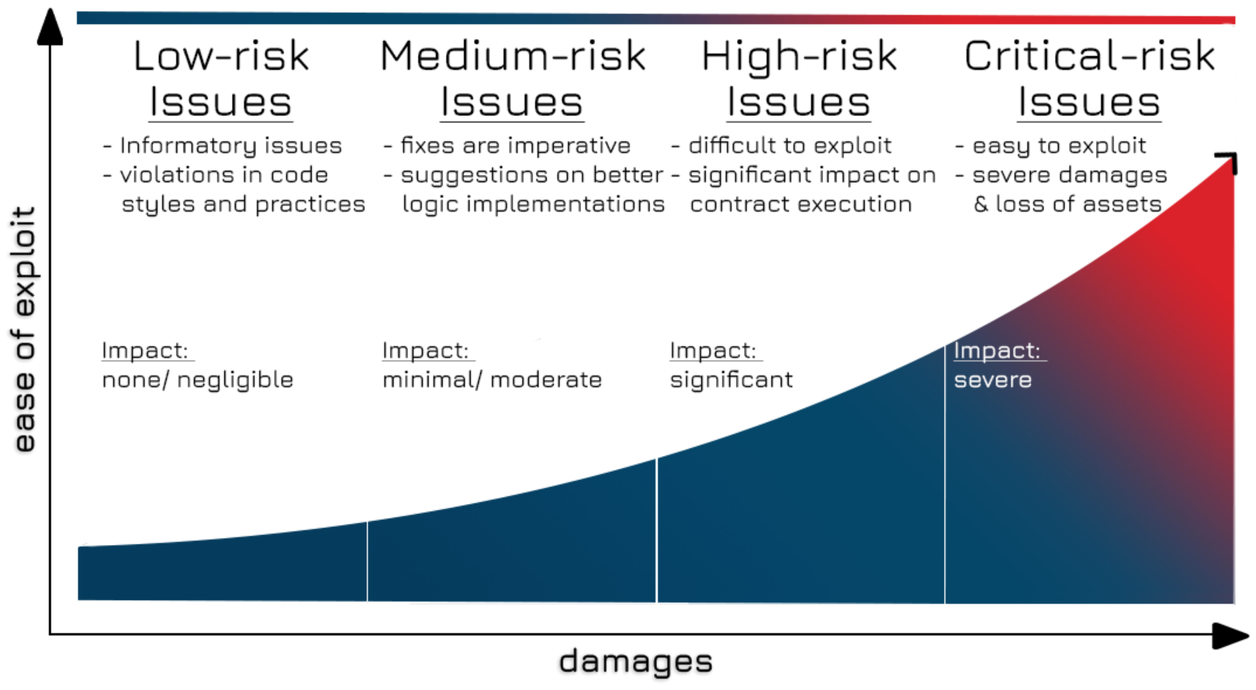


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Mittaria (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis, which started on 13th August '2023

Name
Mittaria
Auditors
Kaif Ahmed | Muhammad Jarir Uddin
Platform
Ethereum and EVM Compatible Chains | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
https://github.com/pellartech/mittaria-blockchain/tree/main/contracts
Website
https://www.mittaria.io/
Document log
Initial Audit Completed: Aug 21, 2023
Final Audit Completed: Aug 25, 2023

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect major issues/vulnerabilities. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

The Genesis NFTs are a set of exclusive hand-drawn spirits residing on Ethereum as ERC-721A tokens. Holding these NFTs will provide users unique utility benefits within the Mittaria Metaverse. Mittaria represents the next frontier in social media platforms, allowing creators to showcase their creativity and share it with the world. They have partnered with professional animators to bring your ideas to life and turn your dreams into reality. By leveraging the power of decentralization, Mittaria platform allows you to fully own your digital assets, enabling seamless interoperability across our entire metaverse ecosystem.

System Architecture

The Mittaria smart contract suite comprises three contracts designed to handle the minting and distribution of NFTs with Royalty.

MittariaGenesis:

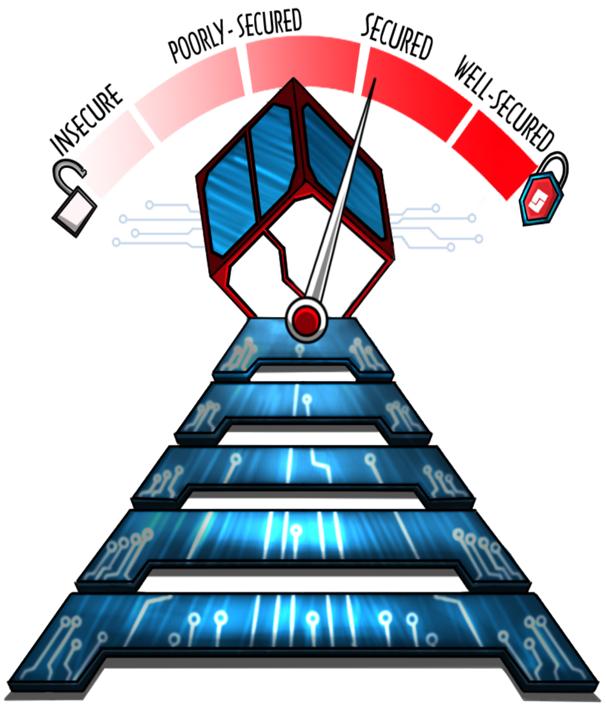
The Genesis contract is designed for the platform's initial token distribution. It orchestrates the minting process across various phases, each with distinct configurations and rules. The contract offers functions to define and modify these phases, ensuring a flexible yet controlled minting process.

Methodology & Scope

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for nine (9) days.

Starting with the recon phase, a basic understanding 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, best practices, and identifying false positives detected by automated analysis tools.

AUDIT REPORT



Executive Summary

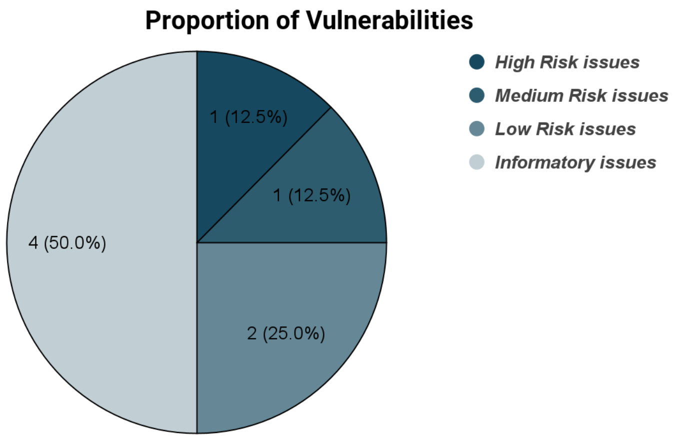
Our team performed a technique called *Filtered Audit*, where two individuals separately audited the Mittaria smart contracts suite.

After a thorough and rigorous manual testing process involving line-by-line code review for bugs, an automated tool-based review was carried out using Slither for static analysis.

All the flags raised were manually reviewed and re-tested to identify the false positives.

Our team found:

#Issues	Severity Level
0	Critical-Risk issue(s)
1	High-Risk issue(s)
1	Medium-Risk issue(s)
2	Low-Risk issue(s)
4	Informatory issue(s)
1	Undetermined issue(s)



Key Findings

#	Findings	Risk	Status
1	Redundant and Risky Withdrawal Mechanism	High	Resolved
2	Inapplicable MaxSupply for tokens	Medium	Fixed
3	Lack of Initial Versioning for Minting Phases	Low	Resolved
4	Insufficient input validations and Lack of Event Emission for Verifier Update	Low	Fixed
5	Insufficient Event Emission	Info	Resolved
6	Insufficient Logs in PhaseModified event	Info	Resolved
7	Ambiguous Naming Conventions in Signature Verification	Info	Fixed
8	Unrestricted Owner Minting Capability	Undetermined	Resolved
9	Hardcoded Withdrawal Address	Info	Resolved

Detailed Overview

High-risk issues

ID	1
Title	Redundant and Risky Withdrawal Mechanism
Path	contracts/Genesis.sol
Severity	High
Function Name	backupWithdraw( )

Description:

The backupWithdraw function in the MittariaGenesis smart contract allows the contract owner to withdraw the entire balance of the contract to the address that calls the function. Having multiple withdrawal functions, especially without clear documentation outlining each purpose, can be a red flag. It raises concerns about the centralization of funds and can be perceived as a potential "rug pull" mechanism, where the developers might withdraw funds unexpectedly, leaving users at a loss.

Code-Affected:

```
function backupWithdraw() public onlyOwner {
    uint256 balance = address(this).balance;
    payable(msg.sender).transfer(balance);
  }
```

Impact:

Such a function can erode trust among users and potential investors. A lack of clarity and the presence of potentially harmful functions can deter user adoption and decrease the perceived legitimacy and security of the contract.

Recommendation:

To address this, the contract should either:

1. Provide comprehensive documentation detailing the specific scenarios and reasons for invoking backupWithdraw and how it differs from other withdrawal functions.
2. If no valid reason exists for its presence, it is recommended to remove this function entirely to reduce complexity and potential centralization risks.

Developer's Response:

We will remove this method.

Auditor's Response:

Acknowledged.

Medium-risk issues

ID	2
Title	Inapplicable MaxSupply for tokens
Path	contracts/Genesis.sol
Severity	Medium
Function Name	setTotalSupply( )

Description:

In the MittariaGenesis contract, the ERC721 token does not place a predefined constraint on the maxSupply of the NFTs. The supporting strategy provided by the protocol shares a disclaimer that initially, the protocol development team has planned on minting 5555 unique NFTs, which will be changed in the future. This plan is not considered fail-safe in the wider NFT community due to the uncertainty that follows for all token holders since a special NFT with a rare availability can have its value boosted due to limited ownership. In case the protocol decides to increase the supply to an absurdly large quantity, there's no way to ensure that the rarity of such a token remains intact via the smart contract implementation.

Code-Affected:

```
function setTotalSupply(uint16 _maxSupply) external onlyOwner {
        maxSupply = _maxSupply;
}
```

Impact:

Having a rare token is all NFTs are about; the price of NFT and exclusiveness offered to holding such a token is an attraction point to users. The uncertainty with this strategy can be a risk factor to NFTs' overall rating leading to an untrusting user base.

Recommendation:

It is recommended that the protocol specifies a concrete maxSupply of tokens in the smart contract and the specifications for each different minting phase to ensure the community's trust is built for the longest period.

Revised-Affected:

```
function setTotalSupply(uint16 _maxSupply) external onlyOwner {
        require(_maxSupply <= MAXSUPPLY); //introduce a constant here for max possible
supply of token
        maxSupply = _maxSupply;
}
```

Developer’s Response:

We will add a maximum of 5555.

Auditor’s Response:

Acknowledged once the above recommendation is enforced.

Low-risk issues

ID	3
Title	Lack of Initial Versioning for Minting Phases
Path	contracts/Genesis.sol
Severity	Low
Function Name	createMintingPhase( )

Description:

In the MittariaGenesis smart contract, when a new minting phase is created using the createMintingPhase function, the version of the phase is implicitly set to zero by default due to the nature of Solidity's default value assignment for unsigned integers. However, in most versioning contexts, the initial version is usually represented as '1' and not '0'. Starting with version '0' can lead to confusion and might be misinterpreted as an uninitialized or incomplete phase, especially for external developers or users unfamiliar with the contract's internals.

Code-Affected:

```
function createMintingPhase(Configs calldata _configs) external onlyOwner {
    _validateMintingPhase(_configs);

    uint256 phaseId = phases.length;
    phases.push();
    phases[phaseId].configs = _configs;
    emit PhaseModified(phaseId, _configs);
 }
```

Impact:

This might lead to misunderstandings about the phase versioning system, potentially causing issues in any systems or dApps that might interface with this contract and rely on phase versioning.

Recommendation:

It is recommended to explicitly set the initial version of a minting phase to '1' when it is created. This would make it clearer and more intuitive for developers and users alike.

Revised-Affected:

```
function createMintingPhase(Configs calldata _configs) external onlyOwner {
    _validateMintingPhase(_configs);

    uint256 phaseId = phases.length;
    phases.push();
    phases[phaseId].configs = _configs;
    phases[phaseId].version = 1;  // Set initial version to 1

    emit PhaseModified(phaseId, _configs);
 }
```

Developer's Response:

Ignored - we will be using 0 - since there won't be any external parties using mint phases (they won't have the cryptographic signature).

Auditor's Response:

Acknowledged with no follow-up required.

ID	4
Title	Insufficient input validations and Lack of Event Emission for Verifier Update
Path	contracts/Genesis.sol; contracts/Badges.sol
Severity	Low
Function Name	setVerifier( )

Description:

In the MittariaGenesis and MittariaBadgesDev smart contract, the setVerifier function allows the owner to set or update the address of the verifier. However, the function lacks the necessary input validation to check whether the provided address is valid or non-zero. This could inadvertently allow the owner to set the verifier address to zero, rendering the verifier functionalities unusable. Furthermore, the function does not emit any event to log the change, making it difficult to track and audit changes to the verifier address over time.

Code-Affected:

```
function setVerifier(address _verifier) external onlyOwner {
    verifier = _verifier;
}
```

Impact:

Without proper input validation and event logging, there's a risk of unintentional mistakes leading to contract misbehavior and a lack of transparency for users and developers interfacing with the contract.

Recommendation:

Adding an input validation check is recommended to ensure the provided verifier address is non-zero. Additionally, emit an event to log the update for transparency and easier tracking.

Revised-Affected:

```
  event VerifierUpdated(address indexed newVerifier);

 function setVerifier(address _verifier) external onlyOwner {
     require(_verifier != address(0), "Invalid verifier address");
     verifier = _verifier;
     emit VerifierUpdated(_verifier);
 }
```

Developer’s Response:

Ok, we will validate the address input.

Auditor’s Response:

Acknowledged with no follow-up required.

Informational issues & Gas Optimizations

ID	5
Title	Insufficient Event Emission
Path	contracts/Genesis.sol
Severity	Informational
Function Name	setPreRevealedURI( ), setBaseURI( ), setBackupURI( ), setHtmlURI( ), setPrimaryRoyalty( ), deleteDefaultRoyalty( ), setRoyaltyInfoForToken( ), resetRoyaltyInforToken( )

Description:

Critical functions in the MittariaGenesis smart contract lack proper event emissions. Events provide transparency and a historical record of contract state changes. The absence of appropriate event loggings in these functions can hinder effective monitoring and auditing of the contract.

Recommendation:

To address this issue, including event emissions against appropriate functionalities is recommended.

Developer's Response:

we will not be using external developments, so those events are not required.

Auditor's Response:

Acknowledged with no follow-up required.

ID	6
Title	Insufficient Logs in PhaseModified event
Path	contracts/Genesis.sol
Severity	Informational
Function Name	-

Description:

In the MittariaGenesis smart contract, the PhaseModified() event emits two parameters for phaseId and configs, respectively. However, in the wider context of how phases work in the Mittaria Protocol, it is observed that updates to a phase play a vital role. During these updates, the new phase detail and config is emitted but no version update is published in the event, which is a vital piece of information to be registered on the blockchain for protocols that employ composability with the Mittaria protocol.

Recommendation:

It is highly recommended to include in the new version update during the PhaseModified event emission.

Developer's Response:

we will not be using external developments, so those events are not required.

Auditor's Response:

Acknowledged with no follow-up required.

ID	7
Title	Ambiguous Naming Conventions in Signature Verification
Path	contracts/Genesis.sol
Severity	Informational
Function Name	mint( )

Description:

In the MittariaGenesis smart contract, the mint function accepts a parameter named _proof. However, within the wider blockchain ecosystem, the terms "proof" and "signature" have distinct definitions and implications. The name _proof might be misleading, especially when associated with cryptographic operations. Additionally, the corresponding verification function is named _verifyProof which further amplifies the ambiguity. Lastly, using the term "invalid signature" in error messages while having a parameter named _proof can cause confusion.

Recommendation:

To ensure clarity and to align with the conventions of the wider blockchain community, it is recommended to rename _proof to _signature and _verifyProof to _verifySignature. Also, ensure that error messages are consistent with the function and parameter names. This will clarify the function's intent and reduce potential ambiguities for developers interfacing with the contract.

Developer's Response:

Thank you, we will change the naming.

Auditor's Response:

Acknowledged with no follow-up required.

ID	8
Title	Unrestricted Owner Minting Capability
Path	contracts/Genesis.sol
Severity	Undetermined
Function Name	mintTo( )

Description:

The mintTo function in the MittariaGenesis smart contract allows the owner to mint a specified amount of tokens to any address. While this function is restricted to the owner, its existence without proper context or documentation might raise concerns about the fairness and decentralization of the token distribution process. Unrestricted minting capabilities can be misused, leading to unpredictable token supply and possibly diluting the value of existing tokens.

Code-Affected:

```
function mintTo(address _to, uint256 _amount) external onlyOwner {
  require(_amount > 0, "Invalid amount");
  _mintToken(_to, uint16(_amount));
}
```

Recommendation:

If the function serves a genuine use case (e.g., for airdrops, rewards, or compensations), it is crucial to provide comprehensive documentation explaining its purpose and the circumstances under which it would be invoked. If there is no valid reason for its existence, consider removing it or adding additional constraints to prevent potential misuse. Proper documentation will ensure transparency and foster trust among the community and token holders.

Developer’s Response: This is a business requirement.

Auditor’s Response: Acknowledged with no follow-up required.

ID	9
Title	Hardcoded Withdrawal Address
Path	contracts/Genesis.sol
Severity	Informational
Function Name	-

Description:

The withdraw function in the MittariaGenesis smart contract directly sends the entire balance of the contract to a hardcoded Ethereum address. This practice lacks flexibility and can pose risks. If, for any reason, the ownership of the hardcoded address changes or becomes compromised, or if the owner loses access to the address, funds may become irretrievable. Furthermore, hardcoded addresses can be opaque to users and other developers, as they may not know the purpose or ownership of the address.

Code-Affected:

```
function withdraw() public onlyOwner {
  uint256 balance = address(this).balance;
  payable(0xe3Bd610f0A53F028F95fB64aFF55Cd65aFdCd1Ef).transfer(balance);
}
```

Recommendation:

Instead of hardcoding the withdrawal address, consider one of the following best practices:

1. Accept the withdrawal address as a function parameter, ensuring that the caller (owner) specifies the address at the withdrawal time.
2. Introduce a public and modifiable state variable for the withdrawal address. This allows for flexibility in changing the address if needed while also providing transparency about the current set address.

Developer's Response: This is a business requirement, thank you.

Auditor's Response: Acknowledged with no follow-up required.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices to date w.r.t cybersecurity vulnerabilities and issues 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, its sale, or any other aspect of the project.

Crypto assets/tokens are the results of 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 for the purpose of making any decisions to buy or sell any token, product, service, or another 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 Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity 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 Solidity that could present security risks.

This audit 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 we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.