SMART CONTRACT AUDIT
FOR
HYPEY

June, 2025

DOC PROPERTIES

Client	HYPEY
Title	Smart contract Audit Report
Version	1.0
Author & Auditor	Alejandro
Classification	Public
Contact	auditblockchain@protonmail.com
Status	Pending review V2 - Failed

CONTENTS

1 | INTRODUCTION

This document presents the cybersecurity audit report for the smart contracts developed for the HYPEY project. The primary goal of the audit is to evaluate the security posture of the deployed smart contract system, assess adherence to best practices, and provide a structured report that can serve as a reference for stakeholders.

2 | SCOPE OF THE AUDIT

The audit covers the following components:

- **Token Smart Contract:** **https://sepolia.basescan.org/address/0x3Abfa38465e0c0b2b2dFcb8Db4676cDF2609d7A0#writeContract**
- **Liquidity Provider (LP) Smart Contract:** **https://sepolia.basescan.org/address/0xB6bb863149DA4913080A2fCb56D96ebDffADB317#writeContract**
- **Vesting Smart Contract:** **https://sepolia.basescan.org/address/0x5dEEabe03Eeea25B94f3BD4770cca0612dF406Bb#writeContract**

Only the smart contract code and its logic are within the scope of this audit. Off-chain components, integrations, and external dependencies are excluded unless directly impacting contract functionality.

3 | AUDIT METHODOLOGY

The audit process follows a structured methodology that includes:

- **Code Review: Manual inspection of smart contract code for vulnerabilities, logic errors, and adherence to coding standards.**
- **Static Analysis: Use of automated tools to detect known patterns of vulnerabilities.**
- **Dynamic Testing: Simulation of contract interactions in test environments to identify unintended behaviors.**
- **Threat Modeling: Evaluation of potential attack vectors and abuse cases specific to the contract's design.**
- **Best Practice Assessment: Comparison of implementation against industry standards and security guidelines.**

4 | PROJECT OVERVIEW

The HYPEY project consists of a smart contract ecosystem built to manage token creation, distribution, and vesting schedules. The project aims to maintain security, transparency, and compliance while ensuring flexibility for future upgrades and governance transitions.

4.1 | SMART CONTRACT ARCHITECTURE

This section describes the architectural design of the smart contract system, including key modules, upgradeability mechanisms, and permission management frameworks.

- Token Smart Contract
- Liquidity Provider (LP) Smart Contract
- Vesting Smart Contract
- UUPS Proxy pattern
- Multisig DAO governance planning

5 | FINDINGS

This section summarizes the vulnerabilities and issues discovered during the audit. Each finding is categorized based on its severity and potential impact. The severity levels are defined as follows:

- **Critical: Issues that may lead to complete contract compromise or loss of funds.**
- **High Severity: Issues that may result in significant financial loss or operational disruption.**
- **Medium Severity: Issues that pose moderate risk and could impact contract behavior or user experience.**
- **Low Severity: Minor issues that may have limited impact but should be addressed for completeness.**
- **Informational: Observations that do not present immediate risk but may improve code quality or security posture.**

5.1 | KEY FINDINGS

This section highlights the most significant and impactful vulnerabilities identified during the audit.

*^*Critical and medium findings are marked in red/orange; the rest are not highlighted in color since their importance is minimal.*

These are findings that require immediate attention or have the highest potential to affect the integrity of the system.

VSC [HYPEY TOKEN] - https://sepolia.basescan.org/address/0x3Abfa38465e0c0b2b2dFcb8Db4676cDF2609d7A0#writeContract

ID	Severity	Title	Category	Status
VSC1	**High**	**Upgradeable Contract Backdoor**	**Centralization**	**Open**
VSC2	**High**	**Reserve Address Hijacking**	**Economic Control**	**Open**
VSC3	**High**	**Overprivileged NFT Contracts**	**Access Control**	**Open**
VSC4	**Low**	**Dynamic Burn Rate Manipulation**	**Tokenomics**	**Open**

VSC5	Low	Dusting Attack Vector	Spam/Phishing	Fixed

V2 - Fixes commit 621043c33184749ae41a298d46b48ecd548f5e6a:

ID	Notes	Status
VSC1	_authorizeUpgrade is still onlyOwner—no timelock or multisig was wired in.	Not Fixed
VSC2	Added zero-address/contract checks, but owner can still swap to any external addr.	Partially Fixed
VSC3	There's a per-call 1% cap and approved list, but no per-user allowance or circuit breaker.	Partially Fixed
VSC4	It remains a hard jump function. No smooth curve or moving-average logic was added.	Not Fixed
VSC5	Now uses an exemption list and a 0.5%-of-balance (min 100 tokens) threshold.	Fixed

XSC [HYPEY VESTING] - https://sepolia.basescan.org/address/0x5dEEabe03Eeea25B94f3BD4770cca0612dF406Bb#writeContract

ID	Severity	Title	Category	Status
XSC1	Medium	Unused Merkle Root Functionality	Access Control	Open
XSC2	Medium	Potential Calculation Errors in Vesting	Business Logic	Open
XSC3	Low	Lack of Input Validation	Input Validation	Open
XSC4	Low	Incomplete Event Coverage	Monitoring	Open
XSC5	Low	Inconsistent Error Messages	Code Quality	Open

V2 - Fixes commit 621043c33184749ae41a298d46b48ecd548f5e6a:

ID	Severity	Status
XSC1	The addVestingScheduleWithProof function now calls MerkleProofUpgradeable.verify(...) against merkleRoot before	Fixed

	adding schedules, so the proof logic is actually enforced.	
XSC2	In addVestingSchedule (and the proof variant) they require cliffUnlockPercent <= 100 and slicePeriodSeconds > 0, preventing out-of-bounds percentages or zero-slice divisions.	Fixed
XSC3	Added most checks (non-zero beneficiary, totalAmount > 0, cliffDuration ≤ duration, etc.), but there’s still no guard that the start timestamp is not in the past (e.g. require(start >= block.timestamp)).	Not Fixed
XSC4	All critical actions now emit events	Fixed
XSC5	Every require(…, "") now uses a clear, descriptive string. There are no custom errors or generic messages left error handling is uniform across the contract.	Fixed

ZSC [TREASURY] - https://sepolia.basescan.org/address/0xB6bb863149DA4913080A2fCb56D96ebDffADB317#code

ID	Severity	Title	Category	Status
ZSC1	High	Ownership Initialization Front-Running Window	Race Condition	Open
ZSC2	High	No Withdrawal Limits	Access Control	Open
ZSC3	Medium	Unbounded Supported Token List	Resource Management	Open
ZSC4	Medium	No Removal from supportedTokenList Array	Data Consistency	Open
ZSC5	Low	Missing Event Emission in initialize	Logging/Auditability	Open
ZSC6	Low	Unused Imports Increasing Bytecode Size	Code Quality	Open
ZSC7	Low	Inconsistent Error-Handling Approach	Error Handling	Open

V2 - Fixes commit 621043c33184749ae41a298d46b48ecd548f5e6a:

ID	NOTES	Status
ZSC1	initialize is a public initializer (even though _disableInitializers runs in the implementation constructor), so an attacker could front-run the real deployer and call initialize(admin, timelock) first. The two-step	Not Fixed

	initializeOwner is never usable because initialize already sets ownerInitialized = true.	
ZSC2	Both disburseToken and disburseETH now include require(amount <= 1_000_000 * 1e18, "Amount exceeds maximum withdrawal limit"), capping each withdrawal to 1 million units.	Fixed
ZSC3	Introduced MAX_SUPPORTED_TOKENS = 50 and require(supportedTokenList.length < MAX_SUPPORTED_TOKENS, .) in addSupportedToken, so the list can never grow beyond 50 entries.	Fixed
ZSC4	removeSupportedToken now loops, swaps the to-remove entry with the last, and calls .pop(), ensuring the on-chain array stays in sync with the mapping.	Fixed
ZSC5	Added emit TreasuryInitialized(admin, timelockAddress); at the end of initialize, so the very first setup is now logged.	Fixed
ZSC6	All imported modules (PausableUpgradeable, UUPSUpgradeable, SafeERC20Upgradeable, IERC20Upgradeable, Initializable, AccessControlUpgradeable, TimelockControllerUpgradeable) are actually used in the contract.	Fixed
ZSC7	Every require() now uses a clear, consistent revert message string. There are no silent failures or mismatched patterns.	Fixed

5.2 | DETAILED RESULTS

This section provides a comprehensive list of all findings identified during the audit, along with their descriptions, severity levels, impacted components, and recommended remediations.

5.2.1 | VSC1 - Upgradeable Contract Backdoor [High]

Description:

The UUPS upgrade pattern allows the owner to replace the entire contract logic without user consent. While the _authorizeUpgrade function is onlyOwner restricted, this creates a single point of failure. A compromised owner could inject malicious code for example, minting functions, fee manipulations) in a new implementation.

Technical Impact:

Full control over token economics
Potential rug-pull vector
Violates trust assumptions in decentralized systems

Recommendations:

Timelock Contracts:

```
// Pseudocode for timelock integration
function _authorizeUpgrade(address newImpl) internal override {
    require(msg.sender == timelock, "Only timelock");
    emit UpgradeScheduled(newImpl, block.timestamp + 48 hours);
}
```

Multi-Sig Threshold: Require 3/5 owner signatures for upgrades.
Transparency Measures: Publish upgrade hashes off-chain before execution.

Analogies:
Nomad Bridge (2022): Upgrade introduced a critical bug leading to $190M exploit.

5.2.2 | VSC2 - Reserve Address Hijacking (High)

Description:

The setReserveBurnAddress function lets the owner redirect burned tokens to any address. This could:
Divert accumulated reserves to a malicious address
Disrupt tokenomics by preventing actual burns

Technical Impact:

Current check only prevents address(0)
No event emitted for changes

Recommendations:

Immutable initialization:

```
constructor(address _reserve) {
    reserveBurnAddress = _reserve; // Immutable
}
```

Fail-Safe design:

Implement a 2-step address change (commit-reveal pattern)

Analogies:
SushiSwap MISO (2021): Auction wallet redirect drained $3M.

5.2.3 | VSC3 - Overprivileged NFT Contracts (High)

Description:

Approved NFT contracts can burn any users tokens without:
Per-user allowances
Per-transaction limits
Expiration mechanisms

Technical Impact:

Owner approves malicious NFT contract
Contract calls burnForNFT(user, MAX_UINT256)

Recommendations:

Implement allowance pattern:

```
mapping(address => mapping(address => uint256)) public nftBurnAllowances;

function approveNFTBurn(address nftContract, uint256 amount) external {
    nftBurnAllowances[msg.sender][nftContract] = amount;
}
```

Circuit breaker:

Global daily burn limit per NFT contract
Emergency revocation function

Analogies:
PolyNetwork (2021): Overprivileged admin key led to $600M exploit.

5.2.4 | VSC4 - Dynamic Rate Manipulation (Low)

Description:

The burn rate jumps abruptly at supply thresholds (1% → 3%), enabling:

1. Front-running large transfers
2. Arbitrage bots exploiting rate changes

Technical Impact:

Distorts fair market pricing
Penalizes large holders disproportionately

Recommendations:

Instead of a sudden jump, use a smooth mathematical function (for example sigmoid or linear interpolation) to adjust the burn rate gradually.

```
function _updateBurnRate() internal {
    uint256 supply = totalSupply();
    // Smooth curve: 1% at 2B supply, 3% at 3B supply (example values)
    burnRateBasisPoints = 100 + (200 * (supply - 2e18) / 1e18);
}
```

Another option is to calculate burn rates based on a 30-day moving average of supply instead of the instantaneous value.

5.2.5 | VSC5 - Dusting attack vector (Low)

Description:

The protocol currently exempts transfers below 100 tokens from the burn tax.
This creates a potential dusting attack vector where attackers can:

- Spam wallets with tiny, exempt transfers (polluting user balances).
- Track wallet activity by sending dust tokens and monitoring on-chain movements.

Technical Impact:

The 100-token exemption bypasses the burn tax, making it cheap for attackers to send dust.

Recommendations:

Instead of a blanket exemption, allow specific addresses (e.g., exchanges, trusted contracts) to bypass the burn tax for small transfers.

```
mapping(address => bool) public isExempt;

function _transferWithBurn(address sender, uint256 amount) internal {
    if (amount < MIN_EXEMPT && !isExempt[sender]) {
        applyBurnTax();
    }
}
```

You can also make the exemption scale with the sender's balance (for ex transfers < 0.1% of balance are exempt).

```
function _transferWithBurn(address sender, uint256 amount) internal {
    uint256 balance = balanceOf(sender);
    if (amount < (balance * 1 / 1000)) { // 0.1% threshold
        applyBurnTax();
    }
}
```

5.2.6 | XSC1 - Unused Merkle Root Functionality [Medium]

Description:

The contract includes a merkleRoot variable and an event for updates (MerkleRootUpdated), but there is no actual merkle proof verification implemented anywhere in the contract. This creates dead code that could potentially be misused in future upgrades if developers assume the verification exists when it doesn't. Same happens with MerkleProofUpgradeable.

Technical Impact:

1. **Dead Code Bloat: Increases contract size unnecessarily**
2. **False Security Assumption: Future developers might assume merkle proofs are enforced**
3. **Upgrade Risk: If activated later without proper testing, could introduce vulnerabilities.**

Recommendations:

Fully implement merkle proof verification for claims (claimWithProof(bytes32[] proof) function) or remove the merkleroot function/MerkleProofUpgradeable if it’s not needed.

5.2.7 | XSC2 - Potential calculation errors vesting [Medium]

Description:

The computeReleasableAmount() function handles complex vesting logic with:

1. Cliff unlocks (percentage-based)
2. Linear vesting post-cliff
3. Multiple time-bound calculations

Technical Impact:

Cliff Percentage Edge Cases:

If cliffUnlockPercent = 100 post-cliff vesting becomes irrelevant
No check preventing cliffUnlockPercent > 100

Time Calculation Precision:

Uses integer division which may truncate small amounts
No safeguards against slicePeriodSeconds = 0

Recommendations:

Add explicit checks:

```
require(cliffUnlockPercent <= 100, "Cliff % > 100");
require(slicePeriodSeconds > 0, "Slice period 0");
```

5.2.8 | XSC3 - Lack of input validation [Low]

Description:

Critical functions like addVestingSchedule() lack validation for:

1. Zero addresses (beneficiary != address(0))
2. Time consistency (start < cliff < duration)
3. Total amount sanity checks (totalAmount > 0)

Technical Impact:

Example of a vulnerable path:

```
addVestingSchedule(address(0), 0, 0, 0, 0, 0, 200); // Would silently accept
```

Recommendations:

Add comprehensive validation:

```
function addVestingSchedule(
    address beneficiary,
    uint256 totalAmount,
    uint256 start,
    uint256 cliffDuration,
    uint256 duration,
    uint256 slicePeriodSeconds,
    uint256 cliffUnlockPercent
) external onlyOwner {
    require(beneficiary != address(0), "Zero address");
    require(totalAmount > 0, "Zero amount");
    require(start >= block.timestamp, "Start in past");
    require(cliffDuration <= duration, "Cliff > duration");
    require(duration > 0 && slicePeriodSeconds > 0, "Invalid duration");
    require(cliffUnlockPercent <= 100, "Invalid %");
```

5.2.9 | XSC4 - Lack of input validation [Low]

Description:

While basic events exist for claims/deposits, these critical actions lack events:

1. **Vesting schedule modification**
2. **Ownership transfers (inherited but not explicitly logged)**
3. **Emergency pauses**

Technical Impact:

Off-chain monitors can't track all state changes
Users cant prove historical schedule terms

Recommendations:

Add events for:

```
event VestingModified(address indexed beneficiary, uint256 index);
event EmergencyAction(string action, address executor);
```

Emit during:

Schedule updates
Admin withdrawals
Pause/unpause

5.2.10 | XSC5 - Inconsistent Error Messages [Low]

Description:

Error messages vary between:

Reverts with strings ("Invalid index")
Custom errors (InvalidInitialization())
Generic requires ("Token transfer failed")

Technical Impact:

Hard to grep/filter logs
Some errors dont have enough context

Recommendations:

Use descriptive strings, for example:

```
require(totalAmount > 0, "HypeyVesting: Zero amount prohibited");
```

5.2.11 | ZSC1 - Ownership Initialization Front-Running [High]

Description:

Between proxy deployment and owner finalization there's a brief window where no owner is set, opening a potential front-running attack by a malicious actor who might initialize as owner first.

Recommendations:

In deployment scripts, immediately invoke initialize in the same transaction as proxy creation or use a factory that atomically sets ownership.

5.2.12 | ZSC2 - No withdrawal limits [High]

Description:

The disburseToken and disburseETH functions grant the contract owner unfettered ability to move any amount of any supported asset at any time:

```
function disburseToken(
    address token,
    address to,
    uint256 amount
) external onlyOwner whenNotPaused { … }

function disburseETH(address payable to, uint256 amount)
    external onlyOwner whenNotPaused {  }
```

Analogies:

Parity Multisig Freeze (2017): A bug allowed a single actor to "kill" a multisig wallet, demonstrating the dangers of relying on one privileged key.

Technical Impact:

Single-Point-of-Failure
If the owner private key is ever compromised through phishing, malware, or insider threat an attacker can drain the entire treasury in one transaction.

No On-Chain Safeguards
There are no built-in checks on maximum per-transaction amounts, no daily or weekly caps, and no delayed execution window.

Reduced Transparency for Users
Even honest owners cant credibly commit to holding funds long-term, since they could on-chain withdraw everything at once.

Recommendations:

Per-Transaction and Daily Caps

- *Track withdrawals with a rolling window.*

- ***For example enforce amount ≤ maxPerTx and dailyTotal[token] + amount ≤ dailyCap[token].***

Multisignature or Timelock

- ***Require n of m owner approvals (using. Gnosis Safe).***
- ***Or route large withdrawals (> threshold) through a timelock contract (24 h delay).***

On-Chain Withdrawal Requests

- ***Owner proposes a withdrawal, which must be "executed" after a delay.***
- ***Allows off-chain stakeholders to audit and react to suspicious requests.***

5.2.13 | ZSC3 - Unbounded Supported Token List [Medium]

Description:

getSupportedTokens returns the entire supportedTokenList array, which can grow without bound and eventually exceed gas limits when iterating or returning the full list.

Recommendations:

Consider adding a reasonable maximum limit or implementing pagination for the getSupportedTokens function.

5.2.14 | ZSC4 - No Removal from supportedTokenList Array [Medium]

Description:

Even after removeSupportedToken flips the mapping flag, the token address remains in supportedTokenList, leading to stale entries and wasted gas on lookups or on-chain UI.

Recommendations:

Implement logic to remove the token from the array or track active/inactive status differently.

5.2.15 | ZSC5 - Missing Event Emission in initialize [Low]

Description:

The initialize function sets up ownership but does not emit any event, making it harder to track critical setup steps on-chain.

Recommendations:

Emit a custom Initialized(address indexed newOwner) event (or leverage OpenZeppelin OwnershipTransferred pattern) at the end of initialization.

5.2.16 | ZSC6 - Unused Imports Increasing Bytecode Size [Low]

Description:

Several OpenZeppelin interfaces (IERC1967, IBeacon) are imported but never used, inflating contract size and deployment costs.

Recommendations:

Remove all unused imports and interfaces; audit for other dead code to stay below bytecode size limits.

5.2.17 | ZSC7 - Inconsistent Error-Handling Approach [Low]

<u>Description:</u>

Mix of require statements and custom errors leads to non-uniform gas usage and harder maintenance.

<u>Recommendations:</u>

Standardize on custom errors (error UnsupportedToken(address token)) for all input checks to save gas and improve clarity.

6. Limitations and Disclaimers

This audit has been conducted based on the provided smart contract source code, documentation, and information shared by the project team up to the audit date. The following limitations and disclaimers apply:

- No Guarantee of Complete Security: While reasonable efforts have been made to identify vulnerabilities, no audit can guarantee the absolute security or correctness of any software, especially in rapidly evolving blockchain ecosystems.
- Scope Limitation: The audit scope was limited to the smart contracts provided. Off-chain systems, front-end applications, APIs, infrastructure, or any external dependencies are outside the scope of this audit unless explicitly stated.
- Code Changes Post-Audit: Any code modifications, updates, or deployment processes executed after the audit are not covered by this report. New vulnerabilities may arise from future code changes or external developments.
- Undiscovered Vulnerabilities: There is always a possibility that undiscovered or undisclosable vulnerabilities may exist due to limitations in testing tools, techniques, time constraints, or emerging attack vectors unknown at the time of the audit.
- Operational Security Responsibility: The project team remains fully responsible for secure key management, governance, access control, deployment practices, and incident response mechanisms after deployment.
- Regulatory Compliance: This report does not constitute legal, financial, or regulatory advice. The project team is responsible for ensuring full compliance with applicable legal frameworks, including but not limited to MiCA, GDPR, and other relevant jurisdictions.
- Reliance on Report: Third parties relying on this report should conduct their own independent assessments before making any decisions. This report is not intended as investment advice or an endorsement of the project.
- No Liability: The auditing firm and its auditors accept no liability for any losses, damages, or regulatory consequences resulting from the use or reliance on this report.