SECURITY AUDIT OF

DATAGRAM

Public Report

Aug 06, 2025

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.
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 was prepared by Verichains Lab on Aug 06, 2025. We would like to thank the Datagram Network 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 Datagram. 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 vulnerabilities in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY 5

1.1. About Datagram 5

1.2. Audit Scope 5

1.3. Audit Methodology 5

1.4. Disclaimer 6

1.5. Acceptance Minute 7

2. AUDIT RESULT 8

2.1. Overview 8

2.2. Findings 8

2.2.1. LOW - Unnecessary Use of Upgradeable Contract Pattern 8

2.2.2. INFORMATIVE – Commented - Out Event Emissions in `setWhitelistedBatch` 9

3. VERSION HISTORY 10

1. MANAGEMENT SUMMARY

1.1. About Datagram

Datagram is a **Global Hyper-Fabric Network** designed to power the next generation of real-time connectivity applications and **Decentralized Physical Infrastructure Networks (DePIN)**. These applications rely on real-world resources such as compute, bandwidth, and storage. Datagram simplifies the process of launching and scaling such applications and networks, eliminating the need to build complex infrastructure.

By unifying idle hardware bandwidth into a global decentralized network, Datagram delivers fast, secure, and scalable connectivity for modern internet applications—from gaming and AI to telecom and beyond. More information at: https://datagram.network/

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the Datagram.

The audit was conducted on commit `1303f0c328900ad5b77c20eece64199b51a20958` from pull request: https://github.com/Datagram-Group/datagram-ns-contracts

The version of the following files were made available during the review:

SHA256 Sum	File
`70d8b55a11ab21f7d34fff6f72cb1b8949f2e1bc0d66ddffc1f2a86abfbd901b`	`./FullCoreSale.sol`
`f8d7bf9b8f694e569406f3b4a74da2abe1eda0c6955008f06c2a008370adf940`	`./WhitelistManager.sol`
`69dba585536c23aa20719f285a666f5c4560ec188ad103dda70173de0b5282f5`	`./NodeSaleFactory.sol`
`aae7534d3ea76bbe38f9fa085c10c479470026750d7e696f37050ecfa196ba60`	`./interfaces/ICoreSale.sol`
`abdb793e3c11179ee0b5b00403e654273cf59e9ed0aa4ce7434d88d2ecc3ffe0`	`./interfaces/IWhitelistManager.sol`

1.3. 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

1.4. Disclaimer

Datagram Network 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. Datagram Network understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Datagram Network 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.

1.5. Acceptance Minute

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

2. AUDIT RESULT

2.1. Overview

The Datagram smart contracts were written in `Solidity` language, with the required version to be `^0.8.20`.

The Datagram's factory contract extends the `Initializable`, `Ownable`, `Clones`, and `ReentrancyGuardUpgradeable` of **OpenZeppelin's contracts**. The contract uses `SafeERC20` for `IERC20`. It deploys and manages individual node sale instances using minimal proxies. It utilizes **OpenZeppelin's Clones** library to create clones of a core sale implementation.

The Datagram's `FullCoreSale` contract extends the `OwnableUpgradeable`, and `ReentrancyGuardUpgradeable` of **OpenZeppelin's contracts**. This is the core implementation contract for each sale instance, designed for deployment via proxies and using the initializer pattern. It supports buying mechanism for different types of sale, and integrates with the `WhitelistManager` to enforce tier-specific rules.

Note: The `FullCoreSale` contract is **upgradeable** contract.

The `WhitelistManager` contract manages access control for different tiers of participants in the node sale. It supports four tiers—Public, Whitelist, Private, and Preferred—each with configurable start and end times. It provides verification functions for sale contracts to check purchase eligibility based on timing and tier.

2.2. Findings

#	Issue	Severity	Status
1	Unnecessary Use of Upgradeable Contract Pattern	LOW	Open
2	Commented-Out Event Emissions in `setWhitelistedBatch`	INFORMATIVE	Open

2.2.1. LOW - Unnecessary Use of Upgradeable Contract Pattern

Position:

- src/core-sale/NodeSaleFactory.sol

Description:

The `NodeSaleFactory` contract inherits from `ReentrancyGuardUpgradeable`, but the contract itself is not designed to be upgradeable. The contract uses a standard constructor pattern rather than the initializer pattern required for upgradeable contracts. Additionally, the `__ReentrancyGuard_init()` function is not called in the initialization phase.

```
// Current implementation
contract NodeSaleFactory is ReentrancyGuardUpgradeable, Ownable {
    // ...
    constructor(...) Ownable(_initialOwner) { ... }
}
```

RECOMMENDATION

Replace the upgradeable version with the standard version if needed.

2.2.2. INFORMATIVE – Commented - Out Event Emissions in `setWhitelistedBatch`

Position:

- src/core-sale/WhitelistManager.sol

Description:

```
function setWhitelistedBatch(
    SaleType saleType,
    address[] calldata accounts,
    bool allowed
) external onlyOwner {
    mapping(address => bool) storage list = _isWhitelisted[saleType];
    uint256 len = accounts.length;

    for (uint256 i; i < len; ++i) {
        list[accounts[i]] = allowed;
        // emit WhitelistUpdated(saleType, accounts[i], allowed); // Commented out
    }
}
```

Unlike `setWhitelisted()`, the `setWhitelistedBatch` function doesn't emit events for each whitelist change. By commenting out the event, this can break event consistency.

RECOMMENDATION

Consider adding back the event to the `setWhitelistedBatch` function

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Aug 06, 2025*	Public Report	Verichains Lab

Table 2. Report versions history