verichains
CYBERSECURITY

SECURITY AUDIT OF

DATAGRAM

Public Report
Version: 1.0
November 7, 2025

Verichains Lab
info@verichains.io
https://www.verichains.io

ABBREVIATIONS

Name	Description
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on *Nov 7, 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 vulnerable issues in the smart contract code.

TABLE OF CONTENTS

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 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.

Severity levels

1.3. 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.

1.4. 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 Overview

2.1. Audit Scope

This audit focused on identifying security flaws in code and the design of the Datagram. It was conducted on commit `14e5d475797580e94ab4bbfdbe3f0e12963f874d` from git repository: *https://github.com/Datagram-Group/contracts-hub*

SHA256 Sum	File
`0f3114aa03898a8fd5ec5f53cd766e4681c0122dc03f9098d2efe2d066825988`	`./validator-manager/PoAManager.sol`
`38632501ad0059b64a1e8df5f6717b2ef3ac4c23f22602ec2acbf0a1389a5a8c`	`./validator-manager/ValidatorManager.sol`
`373c2a0968ef4998c20c73768e6350b7d0ba6fe7b7c1a8cd7bd9c9d778332d5d`	`./validator-manager/StakingManager.sol`
`fe3b2c4eacf18742ec1fc6ee97dead96f7cc0639c5ef5c3b26d0549b081d8a5b`	`./validator-manager/RewardCalculator.sol`
`24927b0f36b6699db831dac163f7ec4ed876e822300fb00f30faf4670c544887`	`./validator-manager/NativeTokenStakingManager.sol`
`69610201016c2a79b0b04caf69beeebcb58b52c39aa1d6f8c5ebd2a1cabf38d5`	`./validator-manager/ValidatorMessages.sol`
`20072378510d7bae1405ca4845604d96af5ceaf0d74279fd9171cedec0de9df2`	`./validator-manager/ACP99Manager.sol`
`585511d586f8d16b8f7ddca2b8fda8db1c05ccec199a272e21012704cde0d6cf`	`./bridge/CrossChainBridge.sol`

2.2. Overview

The Datagram was written in `Solidity` language, with the required version to be `^0.8.25` (`^0.8.20` for the `CrossChainBridge`). The source code was written based on **OpenZeppelin's library**.

`CrossChainBridge` is a bidirectional native-to-wrapped token bridge secured by **EIP-712** typed signatures and a multi-signature validator scheme. The contract operates in two modes: Source Chain Mode and Destination Chain Mode.

The validator-manager module is a comprehensive **Avalanche L1** validator management framework implementing `ACP-77` and `ACP-99` protocols for sovereign blockchain validation. The system provides three validator management paradigms: `PoAManager`, `NativeTokenStakingManager`, and `ACP99Manager`.

3. Findings

#	Title	Severity	Status
1	Missing Nonce Validation (Replay Attack Vector)	HIGH	Fixed
2	Emergency withdraw cannot drain full contract balance	HIGH	Acknowledged
3	Insufficient balance check in `release` function	MEDIUM	Fixed
4	Requirement always bypassed in `withdrawFees`	MEDIUM	Fixed
5	`emergencyWithdraw` doesn't update `collectedFees`	LOW	Fixed
6	No fee handling in `release` and `mint` mechanisms	INFO	Acknowledged
7	Lack of pausing mechanism	INFO	Fixed

3.1. HIGH - Missing Nonce Validation (Replay Attack Vector) - FIXED

Posision

- `src/bridge/CrossChainBridge.sol`

Description

The contract doesn't track used nonces per user. Users provide their own nonces, but there's no validation that a nonce hasn't been reused. The signature includes the `nonce`, but in `deposit` or `withdraw`: No check that this `nonce` hasn't been used before.

```
    function deposit(
        address recipient,
        uint256 amount,
        uint256 destinationChainId,
        bytes memory userSignature,
        uint256 nonce
    ) external payable nonReentrant {
    ...

        // Verify EIP-712 user signature binding parameters + nonce.
        bytes32 structHash = keccak256(
            abi.encode(
                DEPOSIT_TYPEHASH,
                msg.sender,
                recipient,
                amount,
                destinationChainId,
                block.chainid,
                nonce
            )
        );

    ...
    }
```

```
...

    function withdraw(
        address recipient,
        uint256 amount,
        uint256 destinationChainId,
        bytes memory userSignature,
        uint256 nonce
    ) external payable nonReentrant {
    ...

        // Verify EIP-712 user signature binding parameters + nonce.
        bytes32 structHash = keccak256(
            abi.encode(
                WITHDRAW_TYPEHASH,
                msg.sender,
                recipient,
                amount,
                destinationChainId,
                block.chainid,
                nonce
            )
        );

    ...
    }
```

This might lead to different users using the same `nonce` for the **EIP-712** user signature.

Recommendation

The `nonce` must be tied to the user, with each different user requiring a different `nonce`.

Update

• ***Nov 7, 2025*:** The team has implemented the `nonce` checking for each user.

3.2. HIGH - Emergency withdraw cannot drain full contract balance - ACKED

Posision

• `src/bridge/CrossChainBridge.sol`

Description

`emergencyWithdraw` restricts withdrawals to the "operational balance" only, i.e., `address(this).balance - collectedFees`. Any ETH accounted as fees is excluded and cannot be moved by the withdrawer. Attempting to withdraw the full contract balance reverts with `InsufficientBridgeBalance`, effectively "sticking" ETH if the intent is to evacuate all funds in a single emergency call. This contradicts the expected behavior "emergency withdraw all amount in the contract."

The function also enforces `SourceChainOnly`, so it cannot be used to evacuate the destination-chain instance even if that were desired operationally.

```
    function emergencyWithdraw(
        address payable to,
        uint256 amount
    ) external nonReentrant {
        if (msg.sender != withdrawer) revert UnauthorizedWithdrawer();
        if (to == address(0)) revert InvalidRecipientAddress();
        if (amount == 0) revert ZeroAmountNotAllowed();
        if (!isSourceChain) revert SourceChainOnly();
        if (amount + collectedFees > address(this).balance) {
            revert InsufficientBridgeBalance(
                amount,
                address(this).balance > collectedFees
                    ? address(this).balance - collectedFees
                    : 0
            );
        }

        (bool success, ) = to.call{value: amount}("");
        if (!success) revert ETHTransferFailed();

        emit EmergencyWithdrawn(to, amount);
    }
```

In emergencies, the designated withdrawer cannot drain the entire balance—fee-marked ETH remains blocked.

Recommendation

If the requirement is "withdraw all funds in an emergency," consider changing the function mechanism.

3.3. MEDIUM - Insufficient balance check in `release` function - FIXED

Posision

- **`src/bridge/CrossChainBridge.sol`**

Description

The **`release()`** function checks if **`amount > address(this).balance`**, but it doesn't account for **`collectedFees`**. This allows the release function to transfer fees that should be reserved, breaking the accounting system.

```
function release(
        address sender,
        address payable recipient,
        uint256 amount,
        uint256 withdrawalId,
        uint256 sourceChainId,
        uint256 nonce,
        bytes memory userSignature,
        bytes[] memory validatorSignatures
    ) external nonReentrant onlyOperator {
...
        if (amount > address(this).balance)
            revert InsufficientBridgeBalance(amount, address(this).balance);
```

```
...
}
```

Recommendation

Consider to check the amount by both `collectedFees` and `address(this).balance`

Update

• ***Nov 6, 2025*:**

○ On commit `3279fdcd197036685ae7fd1bdeff9627f0cae729`: The team has updated the condition checking `amount > (address(this).balance - collectedFees)`

○ On commit `ba640562ce0fdb063ff0aa7499b7fe3666cbe772`: The team has updated the condition checking to fix insufficient balance check.

3.4. MEDIUM - Requirement always bypassed in `withdrawFees` - FIXED

Posision

• `src/bridge/CrossChainBridge.sol`

Description

In the `withdrawFees` function, there is a requirement to check fee amount:

```
        if (amount > collectedFees) revert NoFeesToWithdraw();
```

However, later in the function, if `isSourceChain` is satisfied, there is one additional requirement:

```
        if (isSourceChain) {
            uint256 operationalBalance = address(this).balance - collectedFees;
            if (amount > address(this).balance - operationalBalance) {
                revert InsufficientBridgeBalance(amount, address(this).balance -
operationalBalance);
            }
```

This requirement checks `amount > address(this).balance - operationalBalance`, in which `operationalBalance` is `address(this).balance - collectedFees`. This means the requirement effectively checks `amount > collectedFees` again. If a user passes the above statement, they can always bypass this check.

Recommendation

Consider rechecking the requirement logic in the `isSourceChain` case.

Update

• ***Nov 6, 2025*:** The team has removed the `isSourceChain` case checking.

3.5. LOW - `emergencyWithdraw` doesn't update `collectedFees` - FIXED

Posision

- `src/bridge/CrossChainBridge.sol`

Description

The `emergencyWithdraw` function allows withdrawing any amount from the contract balance without updating `collectedFees`:

```
    function emergencyWithdraw(
        address payable to,
        uint256 amount
    ) external nonReentrant {
        if (msg.sender != withdrawer) revert UnauthorizedWithdrawer();
        if (to == address(0)) revert InvalidRecipientAddress();
        if (amount == 0) revert ZeroAmountNotAllowed();
        if (!isSourceChain) revert SourceChainOnly();
        if (amount > address(this).balance)
            revert InsufficientBridgeBalance(amount, address(this).balance);

        (bool success, ) = to.call{value: amount}("");
        if (!success) revert ETHTransferFailed();

        emit EmergencyWithdrawn(to, amount);
    }
```

For example, if `collectedFees = 10 ETH` and `balance = 50 ETH`, an emergency withdrawal of `50 ETH` will leave `collectedFees = 10 ETH` but `balance = 0 ETH`. This breaks the accounting system. However, this function can only be called by the whitelist withdrawers.

Update

- ***Nov 6, 2025*:** The team added `collectedFees` to the checking to ensure the function doesn't withdraw from `collectedFees`.

- ***Nov 7, 2025*:** The team changed the function so it could update `collectedFees` in the case `amount + collectedFees > address(this).balancece`.

3.6. INFO - No fee handling in `release` and `mint` mechanisms - ACKED

Posision

- `src/bridge/CrossChainBridge.sol`

Description

The `mint` and `release` functions have no fee collection mechanism, creating an economic design flaw. While users pay fees upfront in `deposit` and `withdraw`, the operators who execute the corresponding cross-chain operations receive no compensation for their gas costs.

3.7. INFO - Lack of pausing mechanism - FIXED

Posision

- `src/bridge/CrossChainBridge.sol`

Description

The contract provides no mechanism to pause flows. None of the state-changing entry points (`deposit`, `withdraw`, `mint`, `release`) check a pause flag. In incident scenarios (validator key compromise, domain separator misconfig, signature abuse, integration bug, or wrapped token issue), operations cannot be halted on-chain, leaving assets exposed until off-chain coordination occurs.

Update

- ***Nov 7, 2025*:** The team has added pausing mechanism for the contract.

4. Version History

Version	Date	Status/Change	Created by
1.0	*Nov 7, 2025*	Public Report	Verichains Lab

Report versions history