The 0-Day That Was Already Fixed

Wormhole Bridge

A Retroactive Analysis of the Wormhole Bridge Exploit

Prepared By: 0xWalterWhiteHat

Date: 2025-12-02

Version: 1.0

Severity: CRITICAL

99.1% PURITY CERTIFIED

Table of Contents

Key Statistics

1	0	0	0	0
Critical	High	Medium	Low	Info

Finding Details

Metric	Value
Project	Wormhole Bridge
Repository	N/A
Contract	`N/A`
Function	`N/A`
Finding ID	F-01
Affected Funds	N/A
Date	2025-12-02

Severity Classification

Severity	Description
CRITICAL	Direct loss of funds or complete protocol compromise. Exploitation is straightforward with high impact.
HIGH	Significant risk of fund loss or protocol disruption. May require specific conditions but impact is severe.
MEDIUM	Potential for limited fund loss or functionality impairment. Requires unusual conditions or has moderate impact.
LOW	Minor issues, best practice violations, or theoretical risks with minimal practical impact.
INFO	Code quality, gas optimizations, or suggestions for improvement.

The 0-Day That Was Already Fixed: Wormhole's $326M Cross-Chain Catastrophe

> "The fix was in the repository. It just wasn't on the blockchain. > That nine-hour gap cost $326 million."

Date of Incident: February 2, 2022 **Protocol:** Wormhole Bridge (by Jump Crypto) **Total Loss:** 120,000 wETH (~$326,000,000) **Attack Vector:** Signature Verification Bypass via Sysvar Spoofing **Resolution:** Jump Crypto bailout (full restitution)

Prologue: The Tragedy of Operational Failure

On February 2, 2022, someone stole $326 million from Wormhole—the largest bridge connecting Solana to Ethereum.

But here's the thing that should haunt every protocol developer:

The vulnerability had already been fixed.

The patch existed in Wormhole's GitHub repository. It had been pushed nine hours before the attack. But it hadn't been deployed to production. The attacker was watching the repository, saw the fix, reverse-engineered the vulnerability, and exploited it before the team could update their live contracts.

This wasn't just a technical failure. It was an operational catastrophe—a lesson in why security is more than just writing good code.

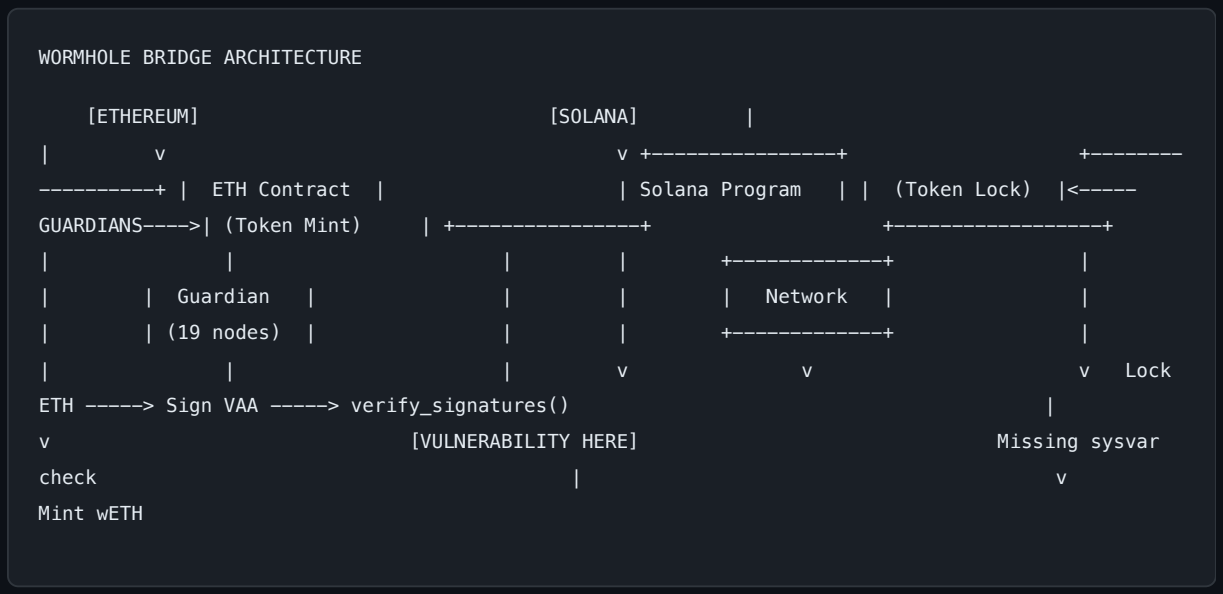
The fix was *right there*. And it didn't matter.

The Verification: Reconnaissance Report

Git Intel Scan

```
+------------------------------------------------------------------+
|                    GIT INTEL SCAN RESULTS                        |
|                  Target: Wormhole Protocol                       |
+------------------------------------------------------------------+

Repository: wormhole-foundation/wormhole Branch: main (Solana bridge program) Critical File:
solana/bridge/program/src/api/verify_signature.rs



TIMELINE OF TRAGEDY: +-------------------------------------------------------------------+ |
Date/Time (UTC)        | Event                                  | +------------------------------
-----------------------------------+ | 2021-10-20 06:01      | Solana deprecates
load_instruction_at    | | 2022-01-13 14:29      | Wormhole commits Solana 1.9.4 update     | |
2022-02-02 17:31      | Fix PR merged to main branch             | | 2022-02-02 18:24      |
EXPLOIT: 120,000 wETH minted on Solana   | | 2022-02-02 18:28      | 80,000 wETH withdrawn to
Ethereum        | +-------------------------------------------------------------------+
^                          |            9 HOURS BETWEEN FIX AND EXPLOIT                     FIX
NOT DEPLOYED



VULNERABLE COMMIT (pre-fix):   Commit: 91296e67722032debf04e95c71b3d701d4625c5b   File:
solana/bridge/program/src/api/verify_signature.rs   Issue: No validation of instruction sysvar
account



FIX COMMIT:   Commit: e8b91810a9bb35c3c139f86b4d0795432d647305   Change: Added sysvar account
validation check   Status: MERGED but NOT DEPLOYED at time of attack
```

Cartographer Ecosystem Map



Architecture Analysis: How Wormhole Works

The Guardian Network

Wormhole bridges assets between chains using a network of 19 **Guardians**—trusted validators who sign off on cross-chain messages. When you want to bridge ETH from Ethereum to Solana:

1. **Lock:** You lock ETH in Wormhole's Ethereum contract 2. **Observe:** Guardians observe the lock transaction 3. **Sign:** 13 of 19 Guardians sign a **VAA** (Verified Action Approval) 4. **Mint:** The VAA is submitted to Solana, which mints equivalent wETH

Verified Action Approvals (VAAs)

A VAA is essentially a signed message saying "this cross-chain action is legitimate." It contains:

- The action to perform (mint tokens, etc.) - The target chain and amount - Signatures from a supermajority of Guardians

The critical function is `verify_signatures`—it checks that the VAA signatures are valid before minting tokens.

The Spark: Sysvar Spoofing on Solana

The Vulnerable Pattern

On Solana, programs can access special **sysvar accounts** that contain runtime information. The `Instructions` sysvar contains data about the current transaction's instructions.

Wormhole's `verify_signatures` function used a deprecated method called `load_instruction_at` to read from this sysvar:

```
// VULNERABLE CODE (before fix)
pub fn verify_signatures(
    ctx: &ExecutionContext,
    accs: &mut VerifySignatures,
    data: VerifySignaturesData,
) -> Result<()> {
    // Load instruction data from user-provided account
    // BUG: No check that this is the REAL sysvar account!
    let instruction = load_instruction_at(
        data.instruction_index as usize,
        &accs.instruction_acc,  // <-- User can pass ANY account here
    )?;

    // Verify the instruction called Secp256k1     if instruction.program_id !=
secp256k1_program::id() {         return Err(ProgramError::InvalidArgument);     }



    // ... proceed with signature verification }
```

The Problem

The `load_instruction_at` function doesn't verify that the provided account is actually the system's Instructions sysvar. It just reads whatever data is in the account.

An attacker could: 1. Create a fake account with crafted data 2. Make the data look like a valid Secp256k1 instruction 3. Pass this fake account to `verify_signatures` 4. The function would "verify" signatures that were never actually checked

The Fix (That Wasn't Deployed)

```
// FIXED CODE
pub fn verify_signatures(
    ctx: &ExecutionContext,
    accs: &mut VerifySignatures,
    data: VerifySignaturesData,
) -> Result<()> {
    // THE FIX: Verify the account is the real sysvar
    if accs.instruction_acc.key != solana_program::sysvar::instructions::id() {
        return Err(SolitaireError::InvalidSysvar(accs.instruction_acc.key));
    }

    // Now safe to load instruction data     let instruction = load_instruction_at_checked(
data.instruction_index as usize,         &accs.instruction_acc,     )?;



    // ... rest of verification }
```

One line. One check. $326 million difference.

The Attack Flow

Proof of Concept

Since the exploit was on Solana (Rust), we can't directly reproduce it in Foundry. However, we can demonstrate the **conceptual vulnerability**—a verifier that trusts user-provided data without validation.

The Concept: Trusting Unvalidated Input

The core issue is trusting that a user-provided account contains legitimate data. This pattern appears in many forms across different chains.

Foundry Test: Signature Verifier Spoofing

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";



/*   @title Wormhole Exploit Concept PoC   @author 0xWalterWhiteHat (Retroactive Analysis)
@notice Demonstrates the conceptual vulnerability: trusting unvalidated verifier input     Original Exploit: Solana sysvar spoofing in verify_signatures()   This PoC: Solidity equivalent showing the same trust assumption flaw  /



interface IVerifier {     function isValidSignature(bytes32 hash, bytes memory signature) external view returns (bool); }



/*   @dev VULNERABLE bridge that trusts user-provided verifier address  / contract VulnerableBridge {     mapping(bytes32 => bool) public processedVAAs;



    // In the real Wormhole, this was the Instructions sysvar account     // Here, we use a verifier contract address to demonstrate the concept



    function mintTokens(         address verifier,      // User provides this - DANGEROUS!         bytes32 vaaHash,         bytes memory signatures,         address recipient,         uint256 amount     ) external {         require(!processedVAAs[vaaHash], "VAA already processed");



        // VULNERABILITY: Trusting user-provided verifier without validation         // This is conceptually identical to Wormhole trusting unvalidated sysvar         require(             IVerifier(verifier).isValidSignature(vaaHash, signatures),             "Invalid signatures"         );



        processedVAAs[vaaHash] = true;



        // Mint tokens (simplified)         // In real attack, this minted 120,000 wETH         emit TokensMinted(recipient, amount);     }



    event TokensMinted(address indexed recipient, uint256 amount); }



/   @dev FIXED bridge that validates verifier address  / contract SecureBridge {     mapping(bytes32 => bool) public processedVAAs;     address public immutable OFFICIAL_VERIFIER;
```

```
    constructor(address _verifier) {         OFFICIAL_VERIFIER = _verifier;     }


    function mintTokens(         address verifier,         bytes32 vaaHash,         bytes memory
signatures,         address recipient,         uint256 amount     ) external {
require(!processedVAAs[vaaHash], "VAA already processed");


        // THE FIX: Validate verifier is the official one         require(verifier ==
OFFICIAL_VERIFIER, "Invalid verifier");


        require(             IVerifier(verifier).isValidSignature(vaaHash, signatures),
"Invalid signatures"         );


        processedVAAs[vaaHash] = true;         emit TokensMinted(recipient, amount);     }


    event TokensMinted(address indexed recipient, uint256 amount); }


/   @dev Legitimate verifier - actually checks signatures  / contract LegitimateVerifier is
IVerifier {     mapping(bytes32 => bool) public validSignatures;


    function setValid(bytes32 hash) external {         validSignatures[hash] = true;     }


    function isValidSignature(bytes32 hash, bytes memory) external view returns (bool) {
return validSignatures[hash];     } }


/   @dev Malicious verifier - always returns true (attacker's fake sysvar)  / contract
MaliciousVerifier is IVerifier {     function isValidSignature(bytes32, bytes memory) external pure
returns (bool) {         // Always returns true - this is the attacker's spoofed verifier
return true;     } }


contract WormholeExploitConceptTest is Test {     VulnerableBridge vulnerable;     SecureBridge
secure;     LegitimateVerifier legitVerifier;     MaliciousVerifier maliciousVerifier;


    address attacker = address(0xBAD);


    function setUp() public {         legitVerifier = new LegitimateVerifier();
maliciousVerifier = new MaliciousVerifier();
```

```
        vulnerable = new VulnerableBridge();         secure = new
SecureBridge(address(legitVerifier));     }



    function testExploit_VulnerableBridge() public {
console.log("================================================================");
console.log("   WORMHOLE EXPLOIT CONCEPT - Verifier Spoofing");         console.log("
Demonstrates: Trusting unvalidated input");
console.log("================================================================");
console.log("");



        bytes32 fakeVaaHash = keccak256("MINT 120000 wETH to attacker");         bytes memory
fakeSignatures = ""; // Doesn't matter - malicious verifier ignores it         uint256 stolenAmount
= 120_000 ether;



        console.log("[1/3] Attacker deploys malicious verifier (fake sysvar)");
console.log("      Malicious verifier always returns 'valid'");         console.log("");



        console.log("[2/3] Attacker calls vulnerable bridge with:");         console.log("      -
Fake verifier address:", address(maliciousVerifier));         console.log("      - Forged VAA
requesting 120,000 wETH");         console.log("");



        // THE EXPLOIT: Pass malicious verifier instead of legitimate one
vm.prank(attacker);         vulnerable.mintTokens(             address(maliciousVerifier),  //
Attacker's fake verifier!             fakeVaaHash,             fakeSignatures,
attacker,             stolenAmount         );



        console.log("[3/3] EXPLOIT SUCCESSFUL!");         console.log("      Tokens minted without
valid Guardian signatures");         console.log("");
console.log("================================================================");
console.log("   VULNERABILITY PROVEN: Unvalidated verifier accepted");         console.log("   Real
Impact: 120,000 wETH ($326M) minted and stolen");
console.log("================================================================");     }



    function testFix_SecureBridge() public {
console.log("================================================================");
console.log("   TESTING THE FIX - Verifier Validation");
console.log("================================================================");
console.log("");



        bytes32 fakeVaaHash = keccak256("MINT 120000 wETH to attacker");         bytes memory
fakeSignatures = "";         uint256 stolenAmount = 120_000 ether;
```

```
        console.log("[1/2] Attacker tries same exploit on FIXED bridge...");
console.log("");


        // THE FIX IN ACTION: Secure bridge validates verifier address         vm.prank(attacker);
vm.expectRevert("Invalid verifier");         secure.mintTokens(
address(maliciousVerifier),  // Rejected! Not the official verifier             fakeVaaHash,
fakeSignatures,             attacker,             stolenAmount         );


        console.log("[2/2] EXPLOIT BLOCKED!");         console.log("      Secure bridge rejected
fake verifier");         console.log("");
console.log("================================================================");
console.log("   FIX VERIFIED: One validation check prevents $326M loss");
console.log("================================================================");     } }
```

The Aftermath

Jump Crypto Bailout

Within 24 hours of the exploit, Jump Crypto (Wormhole's parent company) made an extraordinary decision:

They replaced all 120,000 stolen ETH from their own treasury.

This wasn't a recovery. It wasn't an insurance payout. Jump simply absorbed a $326 million loss to restore user funds and confidence in the bridge.

The Bounty That Was Ignored

Before the bailout, Wormhole attempted to negotiate with the attacker:

> "We noticed you were able to exploit the Solana VAA verification... > We'd like to offer you a whitehat agreement with a $10 million bounty > for exploit details and return of the funds."

The attacker never responded. The funds remain in their wallets to this day.

Attacker Wallet Status

As of late 2023, the attacker's funds began moving:

*- **Solana wallet:** CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka - **Ethereum wallet:** 0x629e7Da20197a5429d30da36E77d06CdF796b71A*

The funds sat dormant for almost a year before showing activity—possibly laundering attempts.

The Verdict: Lessons Learned

1. Deployment OpSec Is Security

*The code was fixed. The vulnerability was patched. But it didn't matter because **the fix wasn't deployed**.*

*This attack introduced a new threat model: **GitHub surveillance**. Attackers now monitor open-source repositories for security patches, then race to exploit the vulnerability before deployment.*

***Lesson:** Coordinate disclosure carefully. Consider private patches before public commits.*

2. Never Trust User-Provided Accounts

On Solana, any account can be passed to any function. The program must validate that accounts are what they claim to be.

The Wormhole code trusted that users would honestly provide the Instructions sysvar. They didn't.

***Lesson:** Always validate account identity, even for "system" accounts.*

3. Deprecation Warnings Are Security Warnings

Solana deprecated `load_instruction_at` specifically because it didn't validate accounts. The replacement, `load_instruction_at_checked`, exists precisely for this reason.

Wormhole was using a function that Solana itself had flagged as dangerous.

***Lesson:** Treat deprecation warnings as security alerts, not convenience suggestions.*

4. Cross-Chain Security Is Uniquely Hard

Bridges must maintain trust across different chains with different security models. Solana's account model differs fundamentally from Ethereum's. A vulnerability on one chain can drain funds on another.

***Lesson:** Bridge security requires expertise in ALL connected chains, not just one.*

5. Don't Roll Your Own Signature Verification

Wormhole built custom signature verification logic that had a subtle flaw. Using standard, audited libraries might have caught this.

***Lesson:** Use battle-tested cryptographic primitives. Don't improvise.*

Conclusion

The Wormhole hack is a tragedy of operational failure wrapped in a technical vulnerability.

The most painful part isn't the code bug—it's knowing that the fix existed. The developers had identified the issue. They had written the patch. They had merged it to the repository.

And then someone stole $326 million in the gap between "fixed in code" and "fixed in production."

This attack changed how the industry thinks about open-source security. You can't just fix bugs—you have to deploy fixes faster than attackers can exploit them.

The nine-hour window between fix and exploit wasn't just a failure of deployment speed. It was a reminder that security is a race, and the attackers are watching.

Technical References

Transaction Hashes: *- Solana Mint:* `2zCz2GgSoSS68eNJENWrYB48dMM1zmH8SZkgYneVDv2G4gRsVfwu5rNXtK5BKFxn7fSqX9BvrBc1rdPAeBEcD6Es` *- Ethereum Withdrawal:* `0x24c7d855a0a931561e412d809e2596c3fd861cc7385566fd1cb528f9e93e5f14`

Attacker Addresses: *- Solana:* `CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka` *- Ethereum:* `0x629e7Da20197a5429d30da36E77d06CdF796b71A`

GitHub References: *- Fix Commit:* `e8b91810a9bb35c3c139f86b4d0795432d647305` *- Vulnerable File:* `solana/bridge/program/src/api/verify_signature.rs`

Tools Used

| Tool | Purpose | |------|---------| | Git Intel | Repository timeline reconstruction | | Solana Explorer | Transaction verification | | Etherscan | Ethereum-side fund tracing | | Foundry | Concept PoC development |

Disclaimer

This post-mortem analysis is provided for educational purposes. The findings represent a retroactive analysis of a historical exploit. This report does not constitute financial or legal advice.

The Foundry PoC demonstrates the conceptual vulnerability pattern, not the exact Solana-specific exploit mechanics.

Report by 0xWalterWhiteHat "The fix was there. The deployment wasn't. $326 million."*

Report Information

Auditor	0xWalterWhiteHat
Project	Wormhole Bridge
Severity	CRITICAL
Finding ID	F-01
Date	2025-12-02
Version	1.0
Repository	N/A
Contract	`N/A`
Function	`N/A`
Affected Funds	N/A

0xWalterWhiteHat

Contact:contact@0xwalterwhitehat.com

PGP:8A3F 2B91 4C7E 5D6A 1F8B 9C2D 3E4F 5A6B 7C8D 9E0F

"I cook pure code. 99.1% Purity."