Hack Analysis: Nomad Bridge, August 2022

Immunefi · Follow
Published in Immunefi
7 min read · Jan 11

Introduction

The Nomad bridge was hacked on August 1st, 2022, and $190m of locked funds were drained. After one attacker first managed to exploit the vulnerability and struck gold, other dark forest travelers jumped to replay the exploit in what eventually became a colossal, “crowdsourced” hack.

A routine upgrade on the implementation of one of Nomad’s proxy contracts marked a zero hash value as a trusted root, which allowed messages to get automatically proved. The hacker leveraged this vulnerability to spoof the bridge contract and trick it to unlock funds.

That first successful transaction alone, which can be seen here, drained 100 WBTC from the bridge–around $2.3m at the time. There was no need for a flashloan or other complex interaction with another DeFi protocol. The attack simply called a function on the contract with the right message input, and the attacker continued throwing blows at the protocol's liquidity.

Unfortunately, the simple and replayable nature of the transaction led others to collect some of the illicit profit. As Rekt News put it, "Staying true to DeFi Principles, this hack was permissionless — anyone could join in."

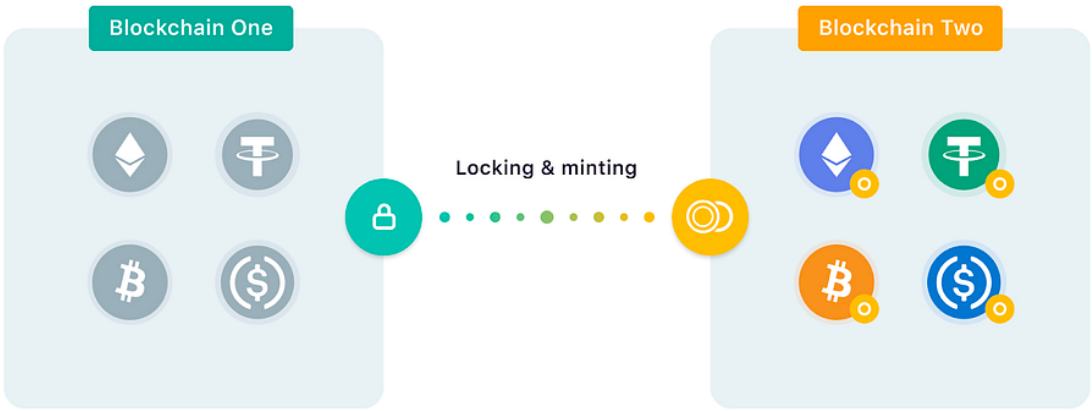
In this article, we will be analyzing the exploited vulnerability in the Nomad bridge's *Replica* contract, and then we'll create our own version of the attack to drain all the liquidity in one transaction, testing it against a local fork. You can check the full PoC here.

This article was written by gmhacker.eth, an Immunefi Smart Contract Triager.

Background

Nomad is a cross-chain communication protocol allowing, among other things, bridging of tokens between Ethereum, Moonbeam and other chains. Messages sent to Nomad contracts are verified and transported to other chains through off-chain agents, following an optimistic verification mechanism.

Like most cross-chain bridging protocols, Nomad's token bridge is able to transfer value through different chains by a process of locking tokens on one side and minting representatives on the other. Because those representative tokens can eventually be burned to unlock the original funds (i.e. bridging back to the token's native chain), they function as IOUs and have the same economic value as the original ERC-20s. That aspect of bridges in general leads to a big accumulation of funds inside a complex smart contract, rendering it a much-desired target for hackers.



Locking & minting process, src: MakerDAO's blog

In Nomad's case, a contract called `Replica`, which is deployed on all supported chains, is responsible for validating messages in a Merkle tree structure. Other contracts in the protocol rely on this for authentication of inbound messages. Once a message is validated, it is stored in the Merkle tree, generating a new committed tree root which gets confirmed to be processed.

Root Cause

Having a rough understanding of what the Nomad bridge is, we can dive into the actual smart contract code to explore the root cause vulnerability that was leveraged in the various transactions of the August 2022 hack. To do that, we need to go deeper into the `Replica` contract.

`Snippet 1: process` function on Replica.sol

The `process` function in the `Replica` contract is responsible for dispatching a message to its final recipient. This will only be successful if the input message has already been proven, which means that the message has already been added to the Merkle tree, leading to an accepted and trustworthy root. That check is done against the message hash, using the `acceptableRoot` view function, which will read from the confirmed roots mapping.

Snippet 2: initialize function in Replica.sol

When an upgrade happens on the implementation of a given proxy contract, the upgrading logic may execute a one-time-call initialization function. This function will set some initial state values. In particular, a routine April 21st upgrade was made, and the value 0x00 was passed as the pre-approved committed root, which gets stored into the `confirmAt` mapping. This is where the vulnerability appeared.

Going back to the `process()` function, we see that we rely on checking for a message hash on the `messages` mapping. That mapping is responsible for marking messages as processed, so that attackers cannot replay the same message.

A particular aspect of an EVM smart contract storage is that all slots are virtually initialized as zero values, which means that if one reads an unused slot in storage, it won't raise an exception but rather it will return 0x00. A corollary to this is that every unused key on a Solidity mapping will return 0x00. Following that logic, whenever the message hash is not present on the `messages` mapping, 0x00 will be returned, and that will be passed to the `acceptableRoot` function, which in turn will return true given that 0x00 has been set as a trusted root. The message will then be marked as processed, but anybody can simply change the message to create a new unused one and resubmit it.

The input message encodes various different parameters in a given format. Among those, for a message to unlock funds from the bridge, there's the recipient address. So after the first attacker executed a successful transaction, anyone that knew how to decode the message format could simply change the recipient address and replay the attack transaction, this time with a different message that would give profit to the new address.

Proof of Concept

Now that we understand the vulnerability that compromised the Nomad protocol, we can formulate our own proof of concept (PoC). We will craft specific messages to call the `process` function in `Replica` function once for each specific token we want to drain, leading to protocol insolvency in just one single transaction.

We'll start by selecting an RPC provider with archive access. For this demonstration, we will be using the free public RPC aggregator provided by Ankr. We select the block number 15259100 as our fork block, 1 block before the first hack transaction.

Our PoC needs to run through a number of steps on a single transaction to be successful. Here is a high-level overview of what we will be implementing in our attack PoC:

1. Select a given ERC-20 token and check the balance of the Nomad ERC-20 bridge contract.
2. Generate a message payload with the right parameters to unlock funds, among which our attacker address as the recipient and the full token balance as the amount of funds to be unlocked.
3. Call the vulnerable `process` function, which will lead to a transfer of tokens to the recipient address.
4. Loop through various ERC-20 tokens with a relevant presence on the bridge's balance to drain those funds in the same fashion.

Let's code one step at a time, and eventually look at how the entire PoC looks. We will be using Foundry.

The Attack

Snippet 3: The start of our attack contract

Let's begin by creating our Attacker contract. The entry point to our contract will be the `attack` function, which is as simple as a for loop going through various different token addresses. We check `ERC20_BRIDGE`'s balance of the specific token that we're dealing with. This is the address of the Nomad ERC-20 bridge contract, which holds the locked funds on Ethereum.

After that, the malicious message payload is generated. The parameters that will change in each loop iteration are the token address and the amount of funds to be transferred. The generated message will be the input to the `IReplica.process` function. As we already established, this function will forward the encoded message to the right end contract on the Nomad protocol to bring the unlock and transferring request to fruition, effectively tricking the bridge logic.

Snippet 4: Generate the malicious message with the right format and parameters

The generated message needs to be encoded with various different parameters, so that it gets properly unpacked by the protocol. Importantly, we need to specify the forwarding path of the message — the bridge router and the ERC-20 bridge addresses. We must flag the message as a token transfer, hence the `0x3` value as the type.

Finally, we have to specify the parameters that will bring the profit to us–the right token address, the amount to be transferred, and the recipient of that transfer. As we've seen already, this will surely create a brand new original message that will never have been processed by the `Replica` contract, which means that it will actually be seen as valid, according to our previous explanation.

Quite impressively, this completes the entire exploit logic. If we had some Foundry logs, our PoC still amounts to only 87 lines of code.

If we run this PoC against the forked block number, we will get the following profits:

- 1,028 WBTC
- 22,876 WETH

- 87,459,362 USDC
- 8,625,217 USDT
- 4,533,633 DAI
- 119,088 FXS
- 113,403,733 CQT

Conclusion

The Nomad Bridge exploit was one of the biggest hacks of 2022. The attack stresses the importance of security throughout the entire protocol. In this particular case, we've learned how a single routine upgrade on a proxy implementation can cause a critical vulnerability and compromise all locked funds. Furthermore, during development one needs to be careful regarding the 0x00 default values on storage slots, specially in logic involving mappings. It's also good to have some unit testing setup for such common values that might lead to vulnerabilities.

It should be noted that some scavenger accounts that drained portions of the funds returned them to the protocol. There are plans to relaunch the bridge, and the returned assets will be distributed to users through pro-rata shares of those recovered funds. Any stolen funds can be returned to Nomad's recovery wallet.

As previously pointed out, this PoC actually enhances the hack and drains all TVL in one transaction. It is a simpler attack than what actually took place in reality. This is what our entire PoC looks like, with the addition of some helpful Foundry logs:

Snippet 5: all code

Nomad Bridge | Immunefi | Bug Bounty | Hacks | Hacking

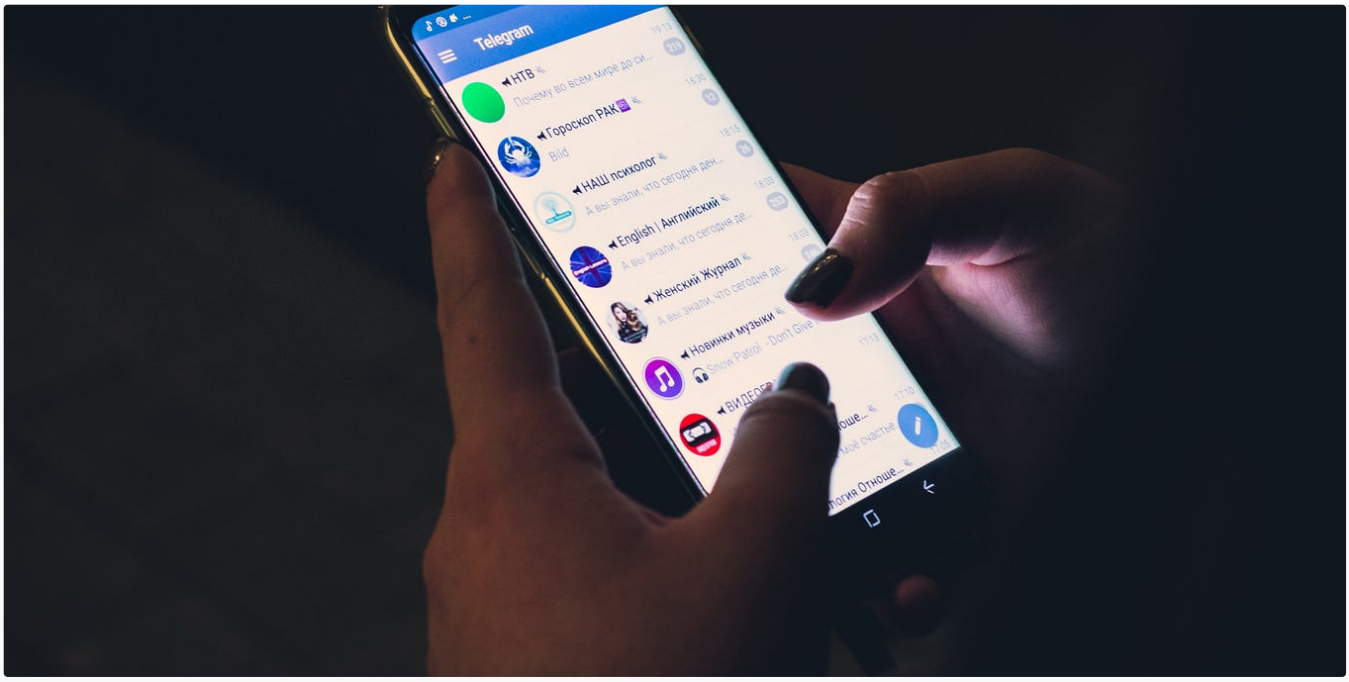
Follow

Written by Immunefi

3.7K Followers · Editor for Immunefi

Immunefi is the premier bug bounty platform for smart contracts, where hackers review code, disclose vulnerabilities, get paid, and make crypto safer.

More from Immunefi and Immunefi



Immunefi in Immunefi

How Not to Get Hacked on Telegram

The lightweight chat client Telegram is one of the most common methods of communication in crypto, and there's a good reason for that. SIM...

5 min read · Jul 28, 2021

919 5

Immunefi in Immunefi

Balancer Rounding Error Bugfix Review

Summary

8 min read · Oct 13

35

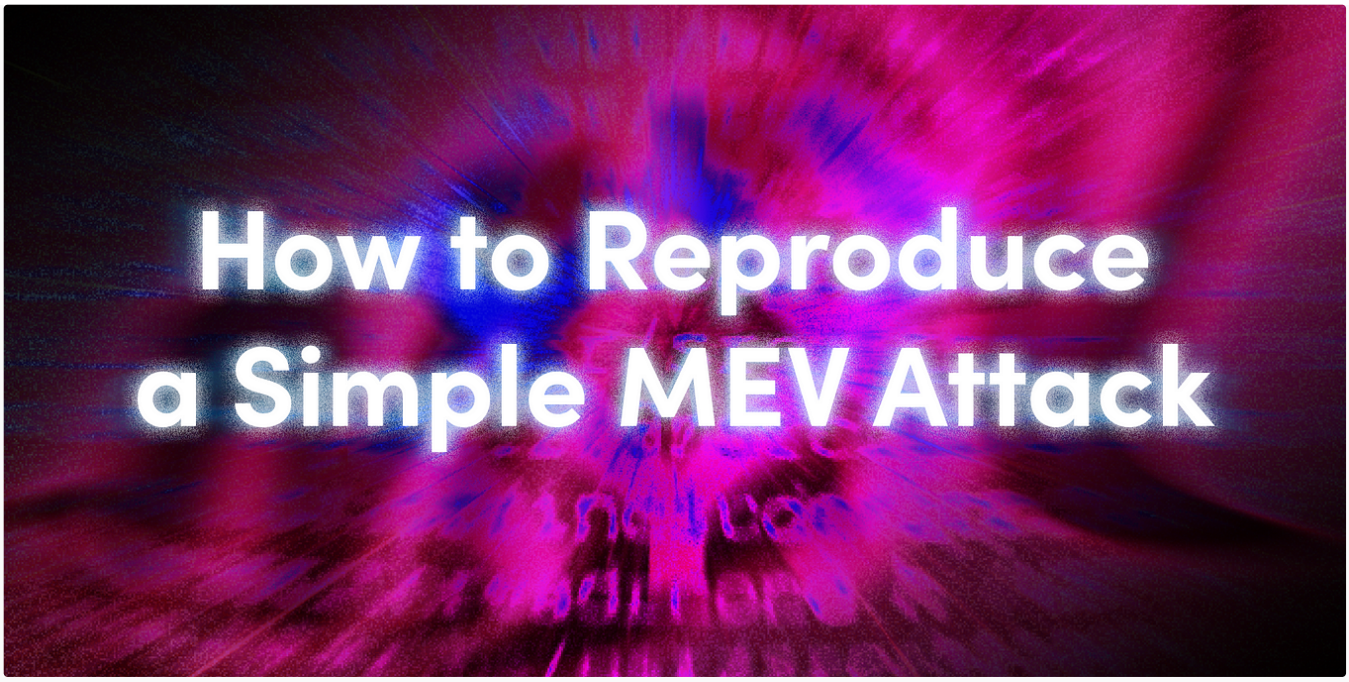
Immunefi in Immunefi

Sui Temporary Total Network Shutdown Bugfix Review

Summary

5 min read · Oct 14

9



Immunefi in Immunefi

How To Reproduce A Simple MEV Attack

Introduction

6 min read · Jul 21

63 2

See all from Immunefi

See all from Immunefi

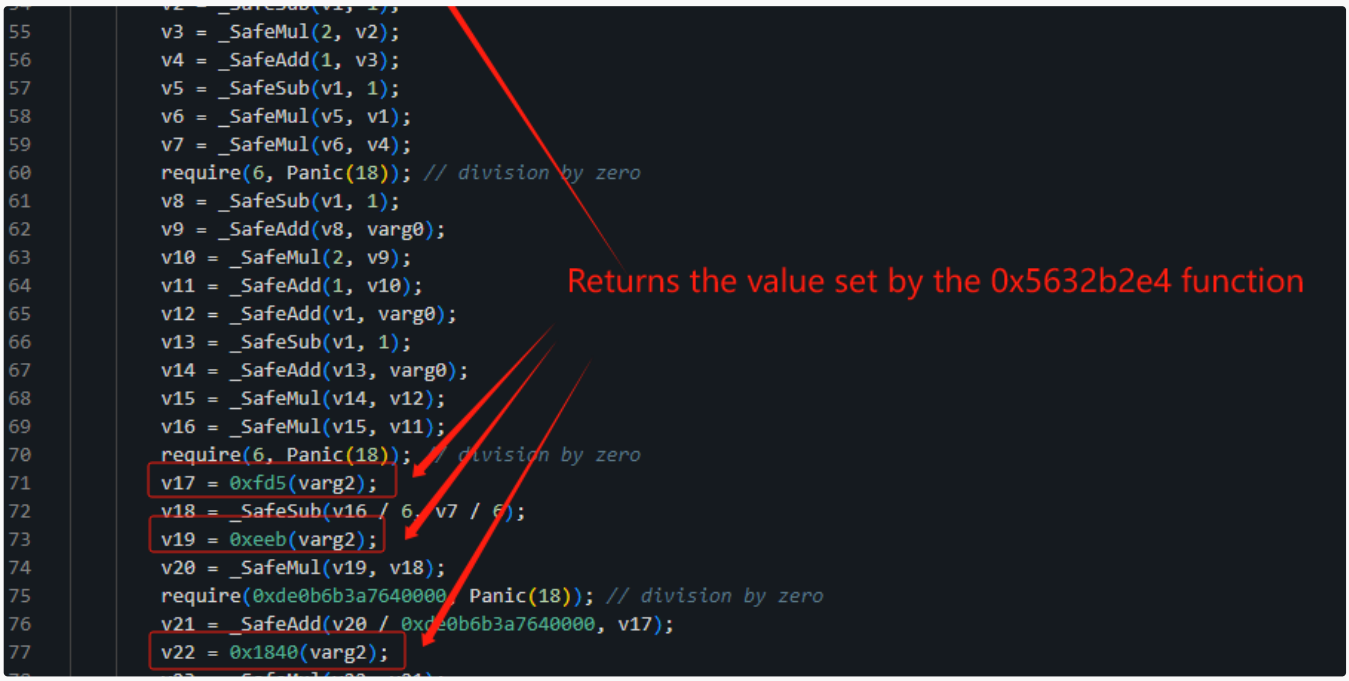
Recommended from Medium

Immunefi in Immunefi

Silo Finance Logic Error Bugfix Review

Summary

6 min read · Jun 16



Shashank in SolidityScan

StarsArena Hack Analysis

Overview:

2 min read · Oct 13

18

Lists

Medium Publications Accepting Story Submissions

154 stories · 884 saves

```
on executeFlashLoan(uint256 amount) external onlyOwne
dress asset = address(pool.asset());
ol.flashLoan(
  this,
  asset,
  amount,
  bytes("")
```

Kundan Kumar Kushwaha in CoinsBench

#1. Unstoppable—Damn Vulnerable DeFi

To pass the challenge make the Vault stop offering flash loans.

6 min read · Jun 18

4 1

Heuss

Critical NFT Bridge Vulnerability : Potential Theft of Deposited NFTs

Already back with my second post, this vulnerability will be easier to explain (and also was easier to catch as I reported it the same day...

3 min read · Jul 27

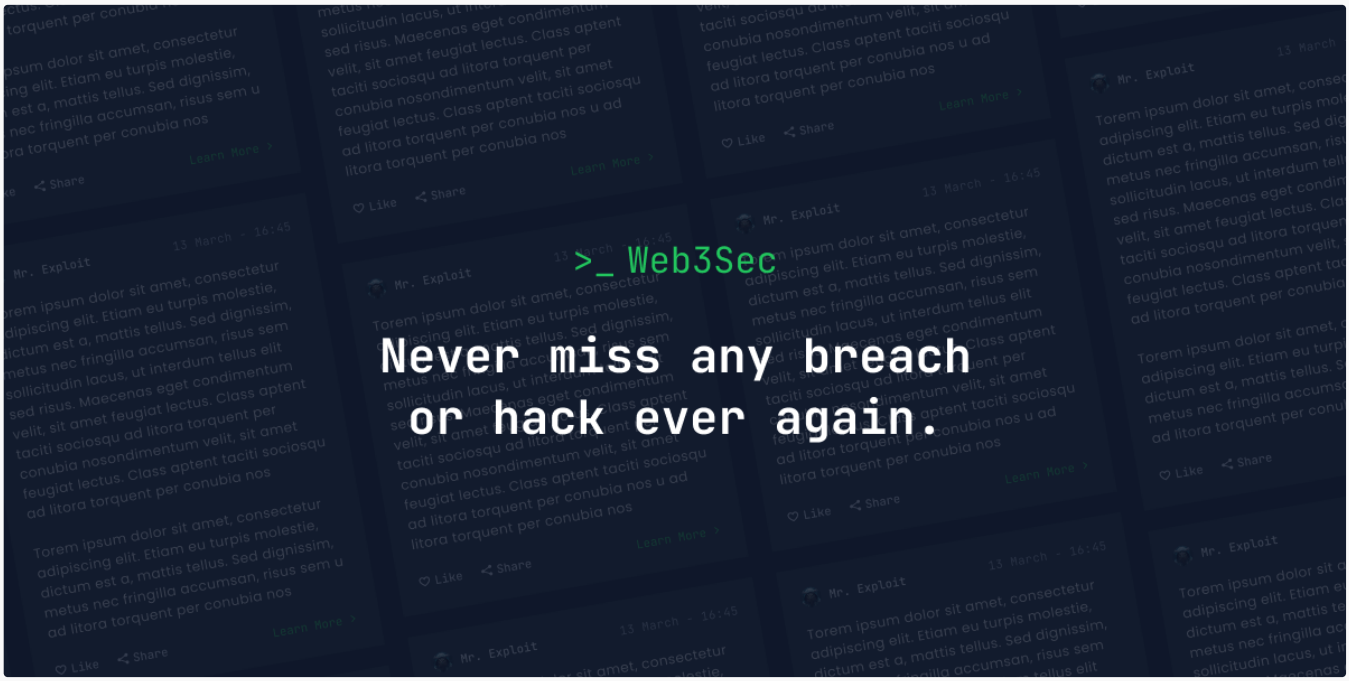
68

Midas Author in Midas Capital

Midas Exploit Post-Mortem

At 5:35 pm UTC on June 17h, 2023, the isolated pool on Midas Capital aimed at supporting the partners from Ankr and Helio Finance on BNB...

5 min read · Jun 20



Chirag Agrawal in InfoSec Write-ups

Smart Contract Best Practice

Top 20 Smart Contract Security Best Practices Checklist

4 min read · Jun 23

26

See more recommendations