A Research Into NFT Whitelist Bypass Vulnerability (1/2)

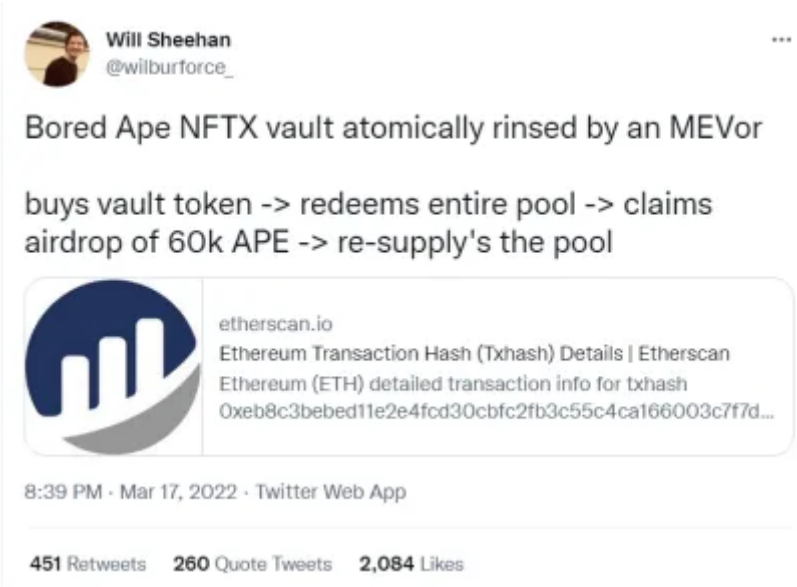
Beosin · Follow
6 min read · Jun 24, 2022

Listen Share

The lifecycle of an NFT is usually divided into phases such as issuance, circulation and burning, among which many of the security issues can be seen in the issuance phase. The purchase of NFT is usually divided into two cases: pre-sale and official sale. In the pre-sale stage, NFT projects usually can set up a whitelist of users who can participate. Despite the settings are made, there are still various types of whitelist bypass vulnerabilities.

Case 1 BAYC airdrop event

On March 17, 2022, a twitter user Will Sheehan tweeted that an MEV bot got over 60,000 APE Coin airdrops through flashloan.



The relevant information is as follows:

Tx:

0xeb8c3bebed11e2e4fcd30cbfc2fb3c55c4ca166003c7f7d319e78eaab9747098

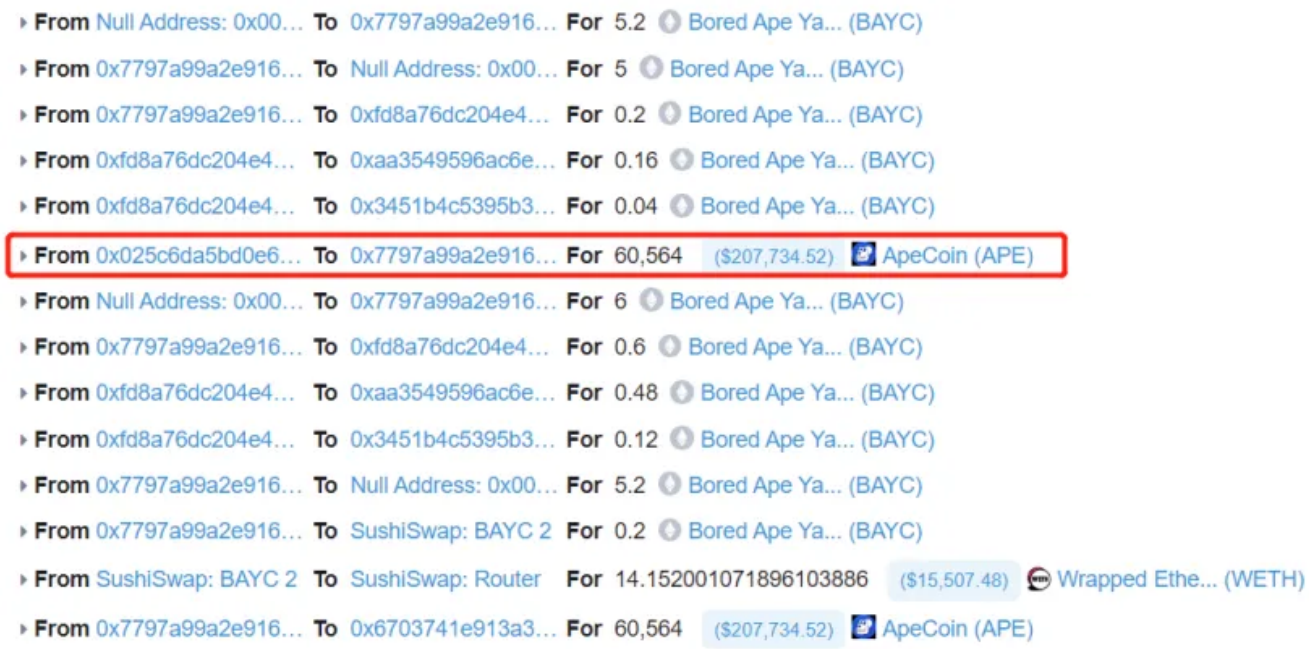
Profited address:

0x6703741e913a30D6604481472b6d81F3da45e6E8

Contract address with vulnerability:

0x025C6da5BD0e6A5dd1350fda9e3B6a614B205a1F

The transaction in which the attacker obtains the relevant APE airdrop is shown in the following figure.



The main steps of the arbitrage:

1. First purchase BAYC NFT with ID 1060 from Opensea.

2. Flashloan 5 BAYC NFTs via NFTX.

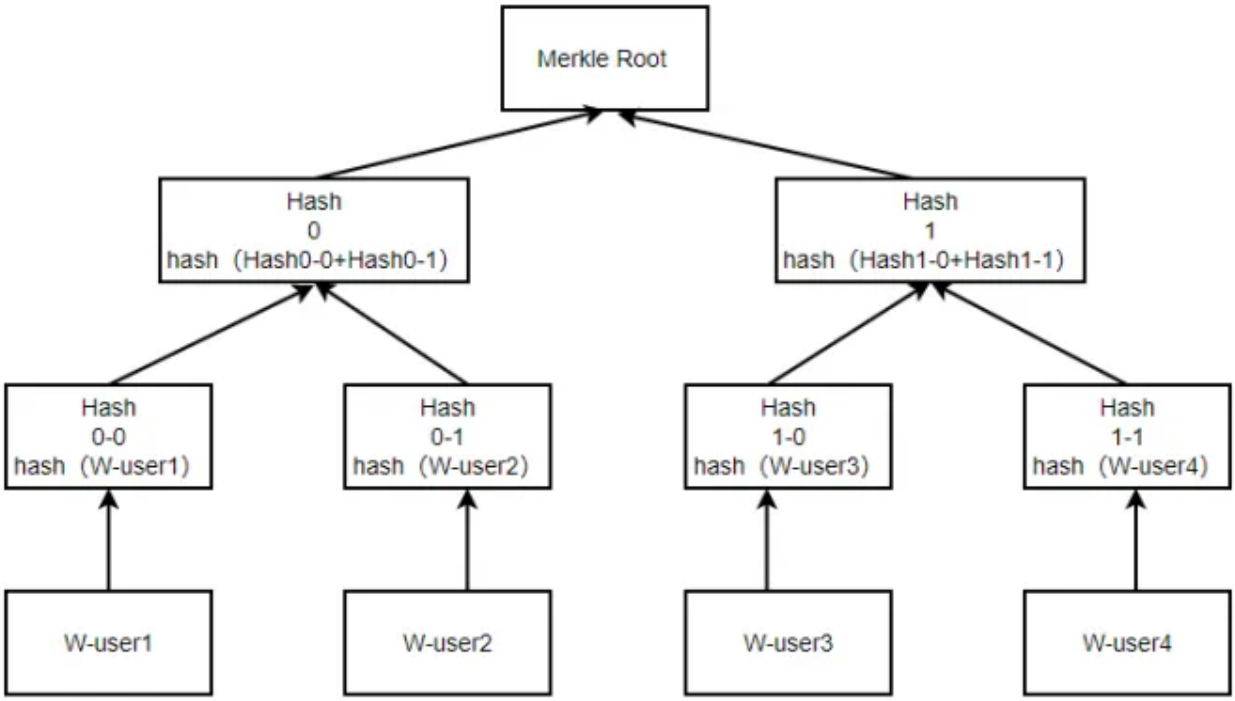
3. Exploit the AirdropGrapesToken vulnerability to claim 6 BAYCs corresponding to the number of APE Coin.

4. Repay the flashloan and transfer of profitable APE Coin.

The code of the AirdropGrapesToken contract:

```
function claimTokens() external whenNotPaused {
    require(block.timestamp >= claimStartTime && block.timestamp < claimStartTime + claimDuration, "Claimable period is finished");
    require((beta.balanceOf(msg.sender) > 0 || alpha.balanceOf(msg.sender) > 0), "Nothing to claim");

    uint256 tokensToClaim;
    uint256 gammaToBeClaim;

    (tokensToClaim, gammaToBeClaim) = getClaimableTokenAmountAndGammaToClaim(msg.sender);

    for(uint256 i; i < alpha.balanceOf(msg.sender); ++i) {
        uint256 tokenId = alpha.tokenOfOwnerByIndex(msg.sender, i);
        if(!alphaClaimed[tokenId]) {
            alphaClaimed[tokenId] = true;
            emit AlphaClaimed(tokenId, msg.sender, block.timestamp);
        }
    }

    for(uint256 i; i < beta.balanceOf(msg.sender); ++i) {
        uint256 tokenId = beta.tokenOfOwnerByIndex(msg.sender, i);
        if(!betaClaimed[tokenId]) {
            betaClaimed[tokenId] = true;
            emit BetaClaimed(tokenId, msg.sender, block.timestamp);
        }
    }

    uint256 currentGammaClaimed;
    for(uint256 i; i < gamma.balanceOf(msg.sender); ++i) {
        uint256 tokenId = gamma.tokenOfOwnerByIndex(msg.sender, i);
        if(!gammaClaimed[tokenId] && currentGammaClaimed < gammaToBeClaim) {
            gammaClaimed[tokenId] = true;
            emit GammaClaimed(tokenId, msg.sender, block.timestamp);
            currentGammaClaimed++;
        }
    }

    grapesToken.safeTransfer(msg.sender, tokensToClaim);

    totalClaimed += tokensToClaim;
    emit AirDrop(msg.sender, tokensToClaim, block.timestamp);
}
```

As shown above, this function uses alpha.balanceOf() and beta.balanceOf() to determine the caller's ownership of the BAYC/MAYC NFT. However, this approach only captures the transient state of the user's ownership of the NFT, which can be manipulated through flashloan. A safer approach is to use a **Merkle** tree based off-chain snapshot approach. This method stores the whitelist on the central server of the off-chain project party. When the user clicks mint on the front-end website, the server generates **Merkle** proof based on the wallet address, and the user then sends a transaction carrying **Merkle** proof to the smart contract and verifies it in the on-chain smart contract.



The approach generally consists of the following three parts.

- Backend generation of **Merkle** trees based on whitelist addresses.

- Uploading **Merkle** Root to the blockchain.

- The front-end generates a **Merkle** Proof based on the current user during validation and passes it into the NFT contract for verification.

For details, please refer to the NFT project Invisible Friends (INVSBLE). Contract address:0x59468516a8259058baD1cA5F8f4BFF190d30E066.

The following is the whitelist minting method mintListed() adopted in the INVSBLE project.

```
// Minting

function mintListed(
  uint256 amount,
  bytes32[] calldata merkleProof,
  uint256 maxAmount
) public payable nonReentrant {
  address sender = _msgSender();

  require(isActive, "Sale is closed");
  require(amount <= maxAmount - _alreadyMinted[sender], "Insufficient mints left");
  require(_verify(merkleProof, sender, maxAmount), "Invalid proof");
  require(msg.value == price * amount, "Incorrect payable amount");

  _alreadyMinted[sender] += amount;
  _internalMint(sender, amount);
}
```

- amount: the number of minting NFTs.

- maxAmount: the maximum number of NFTs that can be minted at this address.

- merkleProof: determine whether a particular whitelisted address node belongs to the required data on the merkle tree, including leaf nodes, paths, and roots.

The function first checks whether the pre-sale is on and whether the caller still has a quota for minting, then performs a whitelist check and verifies that the caller's bid is correct, and finally mints NFT. The following is the code implementation to perform the whitelist verification.

```
function _verify(
  bytes32[] calldata merkleProof,
  address sender,
  uint256 maxAmount
) private view returns (bool) {
  bytes32 leaf = keccak256(abi.encodePacked(sender, maxAmount.toString()));
  return MerkleProof.verify(merkleProof, merkleRoot, leaf);
}
```

In this function, keccak256(abi.encodePacked(sender,maxAmount.toString())) is used to compute the leaf nodes of the Merkle tree, where the whitelisted user address and the maximum number of NFTs each user can mint are used as leaf node attributes. A check is also hidden that the msg.sender must be the whitelist address itself.

MerkleProof validation is calculated using library MerkleProof, the calculation process can be referred to SPV validation, and the source code is as follows.

```
library MerkleProof {
    /**
     * @dev Returns true if a `leaf` can be proved to be a part of a Merkle tree
     * defined by `root`. For this, a `proof` must be provided, containing
     * sibling hashes on the branch from the leaf to the root of the tree. Each
     * pair of leaves and each pair of pre-images are assumed to be sorted.
     */
    function verify(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf
    ) internal pure returns (bool) {
        return processProof(proof, leaf) == root;
    }

    /**
     * @dev Returns the rebuilt hash obtained by traversing a Merklee tree up
     * from `leaf` using `proof`. A `proof` is valid if and only if the rebuilt
     * hash matches the root of the tree. When processing the proof, the pairs
     * of leafs & pre-images are assumed to be sorted.
     *
     * _Available since v4.4._
     */
```



```
            } else {
                // Hash(current element of the proof + current computed hash)
                computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
            }
        }
        return computedHash;
    }
}
```

With this approach, the entire whitelist does not need to be stored in the NFT issuance contract, only the Merkle root. when the transaction sender is a non-whitelisted user, it will not pass the checks because it cannot provide a legitimate MerkleProof.

There is another way of using signatures for off-chain data snapshot verification whitelisting, which is also prone to security problems if contract developers do not set up sufficient security checks at this point, e.g., the NBA arbitrage incident on April 21, 2022.

Case 2 NBA arbitrage incident

On April 21, 2022, the NBAxNFT project tweeted that its pre-sale sold out prematurely due to a problem with its allow list checks.

In this event, the vulnerable contract address is as follows:

0xdd5a649fc076886dfd4b9ad6acfc9b5eb882e83c

```
function mint_approved(
    vData memory info,
    uint256 number_of_items_requested,
    uint16 _batchNumber
) external {
    require(batchNumber == _batchNumber, "!batch");
    address from = msg.sender;
    require(verify(info), "Unauthorised access secret");
    _discountedClaimedPerWallet[msg.sender] += 1;
    require(
        _discountedClaimedPerWallet[msg.sender] <= 1,
        "Number exceeds max discounted per address"
    );
    presold[from] = 1;
    _mintCards(number_of_items_requested, from);
    emit batchWhitelistMint(_batchNumber, msg.sender);
}
```

The above code has two security issues: signature impersonation and signature reuse. Signature reuse means that the same signature can only be used once. Generally, the project will set up a mapping structure in the contract to store whether the signature has been used or not, and the source code is as follows.

mapping(bytes => bool) private usedClaimSignatures

The bytes represent the Hash signature data, and the bool value represents whether the signature has ever been used, but the value is not stored in the mint_approved() function, so there is a signature reuse issue.

We focus on the problem of signature impersonation in whitelist verification here, i.e., the signature can be impersonated due to the vData memory parameter info not being verified by msg.sender when the parameter is passed. The whitelist verification function verify() is as follows:

```
function verify(vData memory info) public view returns (bool) {
    require(info.from != address(0), "INVALID_SIGNER");
    bytes memory cat =
        abi.encode(
            info.from,
            info.start,
            info.end,
            info.eth_price,
            info.dust_price,
            info.max_mint,
            info.mint_free
        );
    // console.log("data-->");
    // console.logBytes(cat);
    bytes32 hash = keccak256(cat);
    // console.log("hash ->");
    //    console.logBytes32(hash);
    require(info.signature.length == 65, "Invalid signature length");
    bytes32 sigR;
    bytes32 sigS;
    uint8 sigV;
    bytes memory signature = info.signature;
    // ecrecover takes the signature parameters, and the only way to get them
    // currently is to use assembly.
    assembly {
        sigR := mload(add(signature, 0x20))
        sigS := mload(add(signature, 0x40))
        sigV := byte(0, mload(add(signature, 0x60)))
    }

    bytes32 data =
        keccak256(
            abi.encodePacked("\x19Ethereum Signed Message:\n32", hash)
        );
    address recovered = ecrecover(data, sigV, sigR, sigS);
    return signer == recovered;
}
```

The above code uses signature verification to verify the whitelist, which is stored on the centralized server. When a user clicks mint from the front-end NFT website, the server will first verify whether the user is a whitelist user. If yes, the server will use the project's private key to sign, and then return the signature data. Finally, the user carries this signed transaction sent to the chain for verification. So here ecrecover() verifies only the address of the project owner, which only proves that the signature data is issued by the project owner, but the signature can be impersonated as the caller in the signature data, i.e. info.from, is not verified.

We will continue to talk about the security issues in NFT platforms in our next research. Stay tuned!

More

1. *How to Steal User's Signature in NFT Phishing Attacks?*

2. How to Ensure the Security of NFT Under the Web 3.0 Boom?

3. DEUS Finance Suffered its Second Flashloan Attack This Year: Beosin's Detailed Analysis

4. Investigation of Common Phishing Attacks in Web 3.0: Discord, Google Ads, Fake Domains and Others

5.「RECAP」AMA About How to Keep Your Smart Contract Secure During Development With Beosin VaaS

6. Hype, Plagiarism, Insider Fraud, NFT Scams on OpenSea and Security Advice

Contact

If you have need any blockchain security services, please contact us:

Website Email Official Twitter Alert Telegram LinkedIn

Blockchain Web3 Nft Crypto Smartcontract Audit

Follow

Written by Beosin

295 Followers

Blockchain Security Audit Service · Official Website: https://beosin.com · https://t.me/beosin

More from Beosin

Beosin

How to Avoid Telegram Scams?

Recently, Telegram, a cross-platform instant messaging (IM) application, has seen a spate of account thefts through illegal means and...

8 min read · Feb 13

11

Beosin

A Closer Look at the Anti-Sybil Mechanism Under the Arbitrum Airdrop Hype

Recently, the airdrop of Arbitrum, a Layer 2 scaling protocol has become a hot topic in the crypto market.

9 min read · Mar 25

94

Beosin

Security Audit | What Are the Common Characteristics of Recent Web3 Attacks, and How Can Projects...

Recently, there have been numerous security attack incidents, and these incidents have had a significant impact on project teams. One of...

7 min read · Oct 16

12

Beosin

A Must-Read for ZKP Projects | Circuit Audit: Are Redundant Constraints Really Redundant?

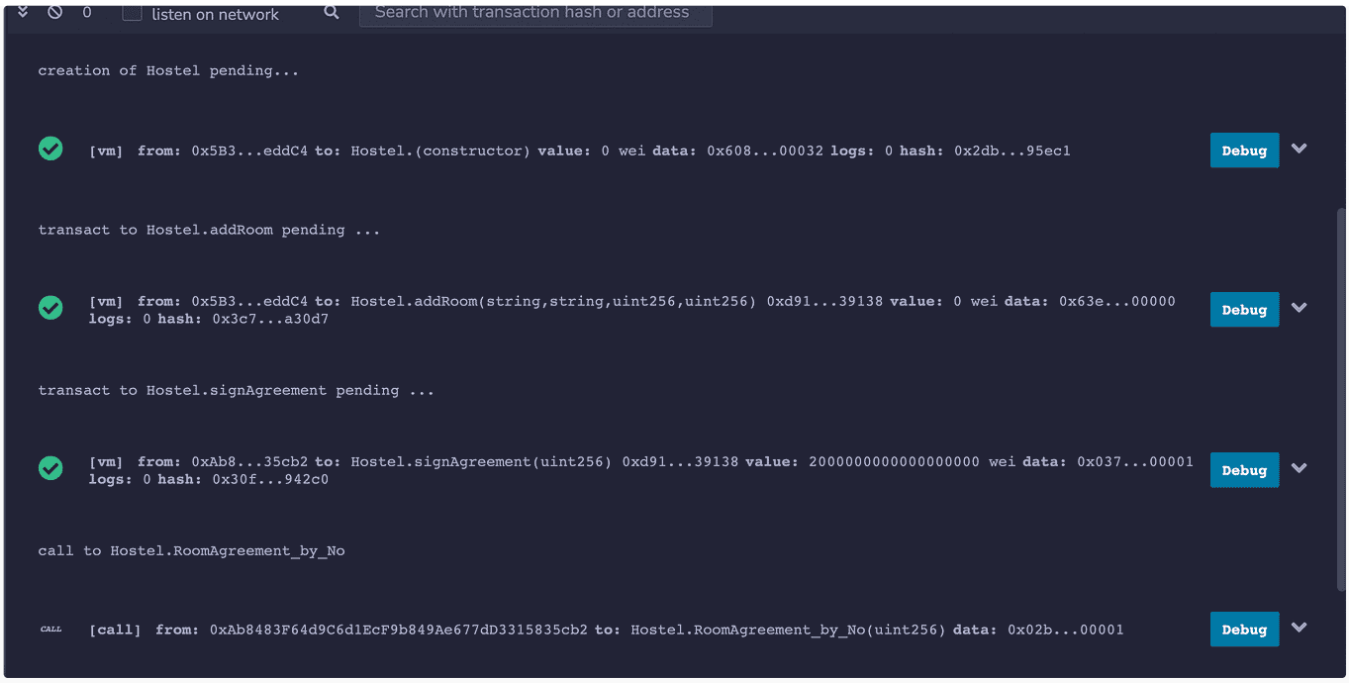
1. Preface

9 min read · Sep 20

5

See all from Beosin

Recommended from Medium



Luis Soares in Coinmonks

Understanding and Preventing Short Address Attacks in Solidity Smart Contracts

What is a Short Address Attack?

· 3 min read · May 22

136 2

Sena aslıbay

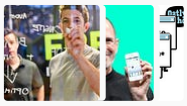
10 Most Common Ethereum Smart Contract Vulnerabilities

1.Missing Access Control

9 min read · Oct 11

16

Lists



Modern Marketing

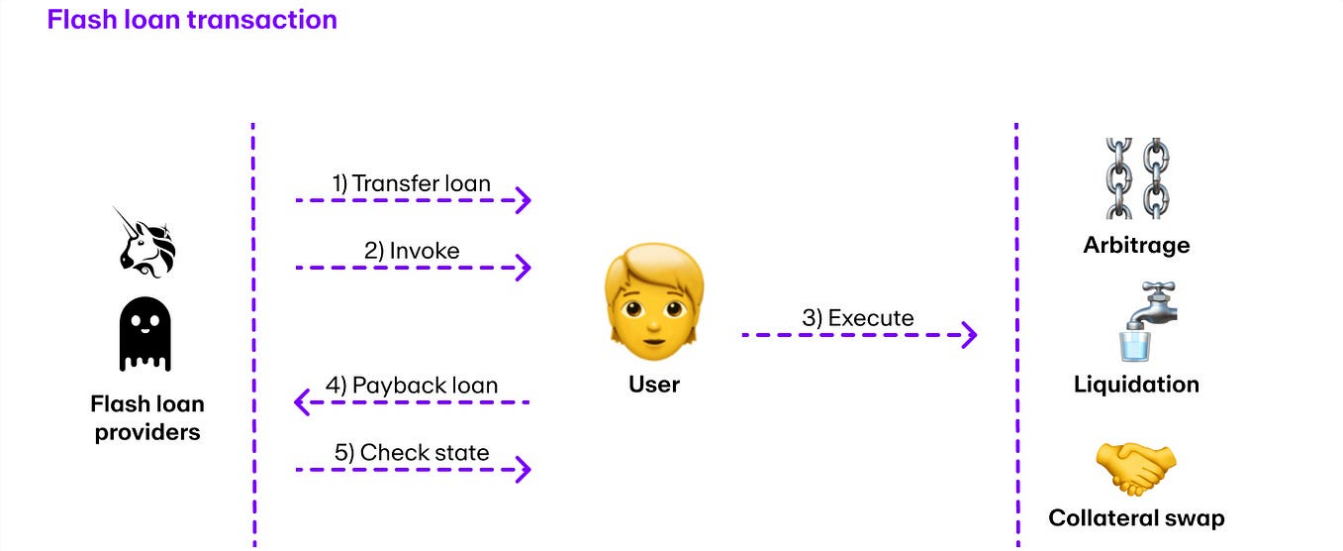
38 stories · 218 saves

My Kind Of Medium (All-Time Faves)

45 stories · 104 saves

Generative AI Recommended Reading

52 stories · 354 saves



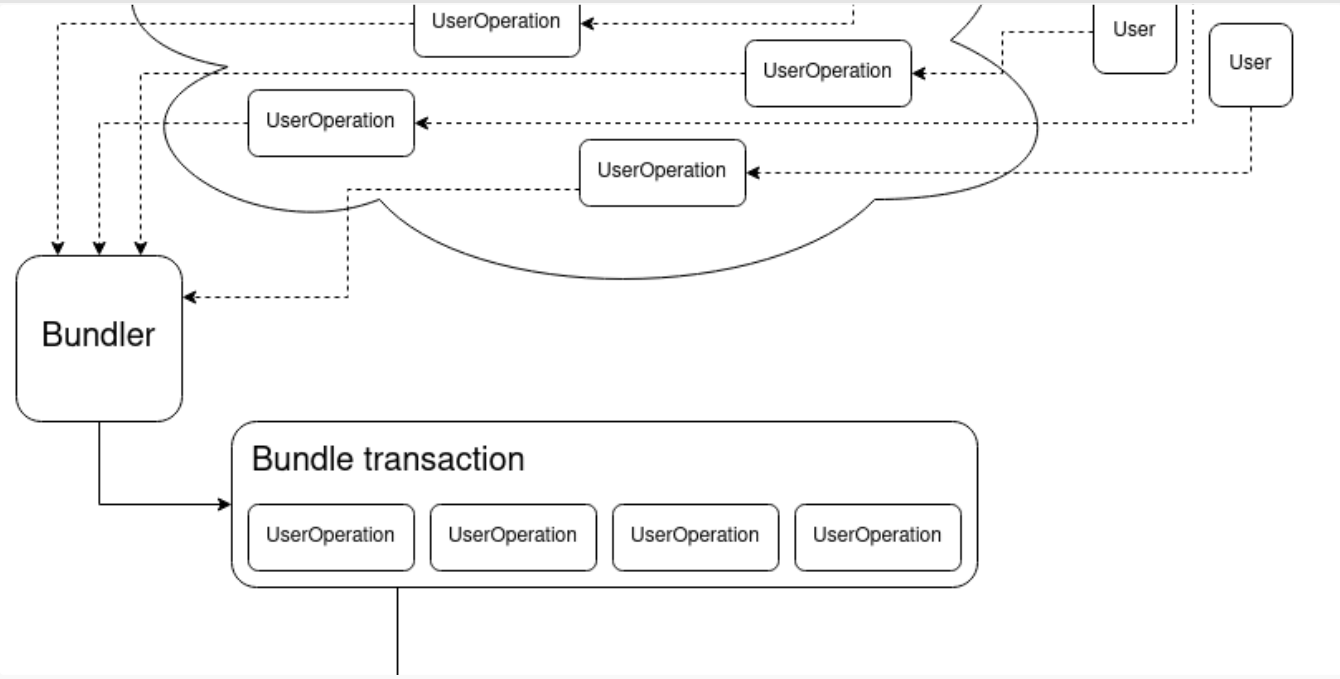
Seyyed Ali Ayati

Understanding Flash Loan Attacks Through a Practical Example

A Comprehensive Case Study of a Flash Loan Attack

10 min read · Aug 30

3



Antonio Viggiano in Oak Security

A deep dive into the main components of ERC-4337: Account Abstraction Using Alt Mempool— Part 1

Account abstraction has been a highly desired feature within the Ethereum developer community for years, and it is seen by many as a...

5 min read · Aug 29

Phuwanai Thummavet in Valix Consulting

Solidity Security By Example #12: Amplification Attack (Double Spending #1)

By Phuwanai Thummavet

8 min read · May 23

67

Encoding Labs

[ZK] Part 4: Guide to zkEVM

In Part 3 of our zero-knowledge series, we explored how zkEVMs work, the technical difficulties in developing them, and the various types...

7 min read · May 9

3

See more recommendations