SANS

HACKFEST SUMMIT 2023
WORKSHOP

BLOCKCHAIN SECURITY FOR RED TEAMS

STEVEN WALBROEHL

CO-FOUNDER @ HALBORN -

AUTHOR & INSTRUCTOR || SEC554:
BLOCKCHAIN & SMART CONTRACT SECURITY

AGENDA

- ACCESS CONTROL FOR EVM
- DEBUGGING ACCESS CONTROL FAILURES ON-CHAIN
- BREAKING THE MERKLE TREE FOR SMART CONTRACT VALIDATION
- WORKSHOP – STEAL THIS NFT

Access control is incredibly important in the world of smart contracts.

Access controls govern who can mint tokens, vote on proposals, freeze transfers, and execute privileged functions.

Access can be defined on a function, given to a user/wallet, a group of users via multi-signatures or another smart contract.

Implementation failure can allow someone to take over your whole system, or bypass business logic.

Types of access control in Solidity

1) Native function/variable access levels

Variable	Description
`public`	Default for functions. Enables the state variable or function to be accessed from within the contract and also from external contracts or clients.
`internal`	Default for variables. The contract and imported/inherited contracts can access the variable or function.
`private`	Only from within the calling contract itself can the variable/function be accessed.
`constant`	Only applies to state variables. Set state variables to a value that isn't coming from storage or blockchain.
`external`	Only applies to functions. Can be accessed by external contracts or clients, but not from within the contract.

```
function validateLimit(uint256 _amount) private
returns (bool) {
  if (_amount < maxLimit)
    return true;
  return false;
}
```

Types of access control in Solidity

2) OpenZeppelin Core contract libraries

Core

- `Ownable`
- `Ownable2Step`
- `IAccessControl`
- `AccessControl`

- `AccessControl` provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts.
- `Ownable` is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

```
import "@openzeppelin/contracts/access/Ownable.sol";

import "@openzeppelin/contracts/access/AccessControl.sol";
```

Ownable.sol

Contract module which provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions.

The initial owner is set to the address provided by the deployer.

This can later be changed with `transferOwnership`.

This module is used through inheritance. It will make available the modifier `onlyOwner`, which can be applied to your functions to restrict their use to the owner.

MODIFIERS

`onlyOwner()`

FUNCTIONS

`constructor(initialOwner)`

`owner()`

`_checkOwner()`

`renounceOwnership()`

`transferOwnership(newOwner)`

`_transferOwnership(newOwner)`

Ownable.sol

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol

```
event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

/**
 * @dev Initializes the contract setting the deployer as the initial owner.
 */
constructor(address initialOwner) {
    _transferOwnership(initialOwner);
}

/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
    _checkOwner();
    _;
}

/**
 * @dev Returns the address of the current owner.
 */
function owner() public view virtual returns (address) {
    return _owner;
}

/**
 * @dev Throws if the sender is not the owner.
 */
function _checkOwner() internal view virtual {
    if (owner() ≠ _msgSender()) {
        revert OwnableUnauthorizedAccount(_msgSender());
    }
}
```

AccessControl.sol

This library enables implementing role-based access control.

Its usage is straightforward: for each role that you want to define, you will create a new *role identifier* that is used to grant, revoke, and check if an account has that role.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract AccessControlERC20MintBase is ERC20, AccessControl {
    // Create a new role identifier for the minter role
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    error CallerNotMinter(address caller);

    constructor(address minter) ERC20("MyToken", "TKN") {
        // Grant the minter role to a specified account
        _grantRole(MINTER_ROLE, minter);
    }

    function mint(address to, uint256 amount) public {
        // Check that the calling account has the minter role
        if (!hasRole(MINTER_ROLE, msg.sender)) {
            revert CallerNotMinter(msg.sender);
        }
        _mint(to, amount);
    }
}
```

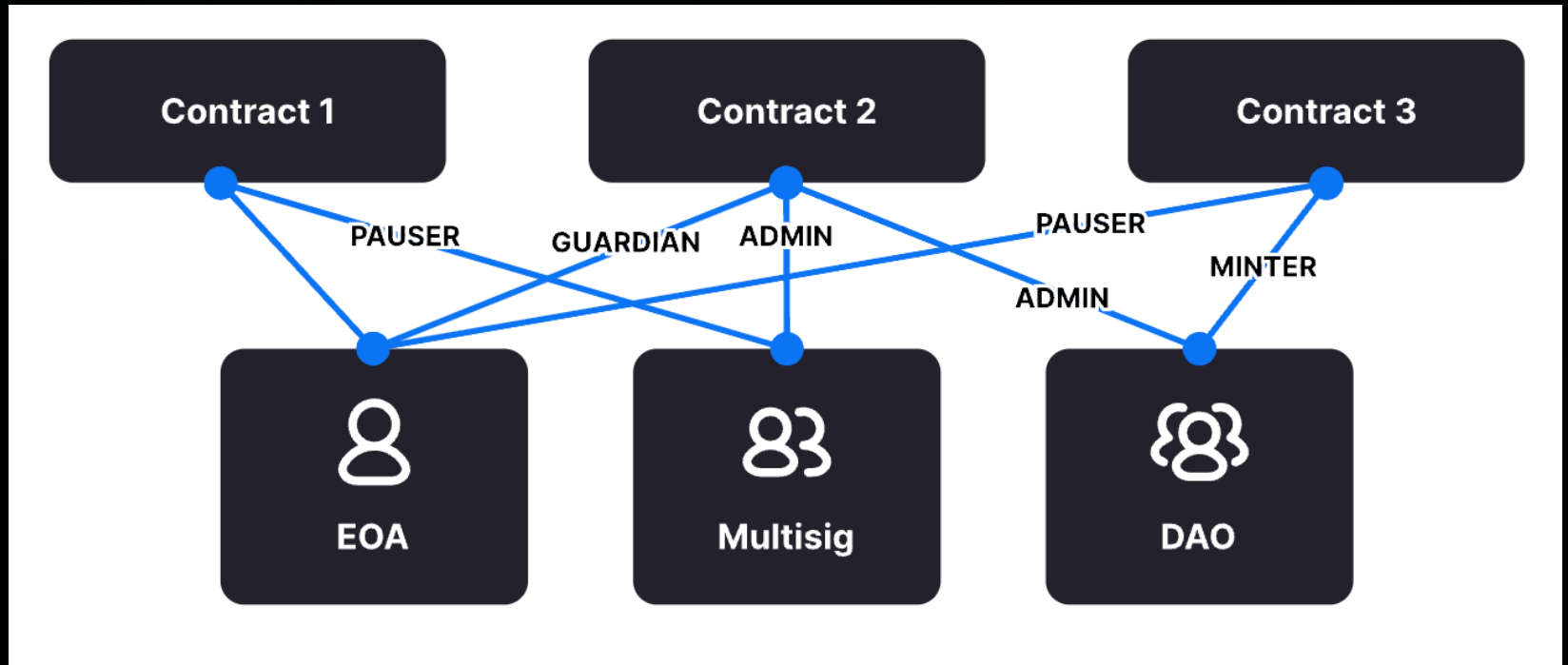
AccessControl.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract AccessControlERC20MintMissing is ERC20, AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");

    constructor() ERC20("MyToken", "TKN") {
        // Grant the contract deployer the default admin role: it will be able
        // to grant and revoke any roles
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE) {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) public onlyRole(BURNER_ROLE) {
        _burn(from, amount);
    }
}
```

Here, no accounts are granted the '`minter`' or '`burner`' roles. However, because those roles' admin role is the default admin role, and *that* role was granted to `msg.sender`, that same account can call grantRole to give minting or burning permission, and `revokeRole` to remove it.

Although AccessControl is a dynamic solution for adding permissions to contracts, decentralized protocols tend to become more complex after integrating many with other systems in a moduler envrionment.

This increases the complexity of permissions management and monitoring across the systems which can leave many gaps for preivlege access, and business logic flaws not intended to be executed by certain roles (or red-teamers) .



LETS LOOK AT SOME ACCESS CONTROL FAILURES THAT PREVIOUSLY OCCURRED ON-CHAIN

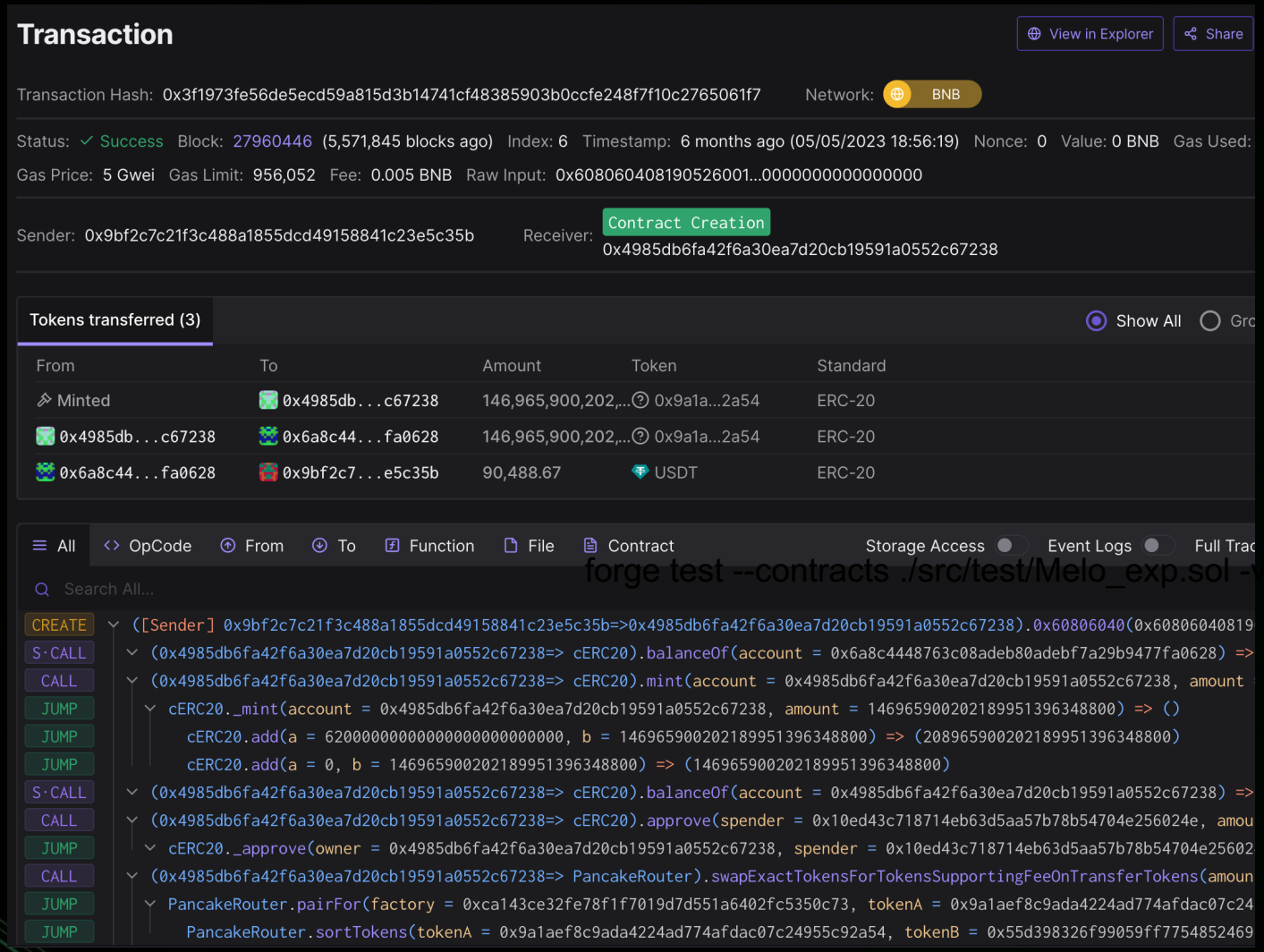
IF YOU WANT TO FOLLOW ALONG ON THE DEBUGGING AND VULNERABILITY RESEARCH, DOWNLOAD FOUNDRY ON ZIION OR ON YOUR MACHINE

```
$ curl -L https://foundry.paradigm.xyz | bash
```

```
$ foundryup
```

And download:

https://github.com/HalbornAcademy/DeFiHackLabs

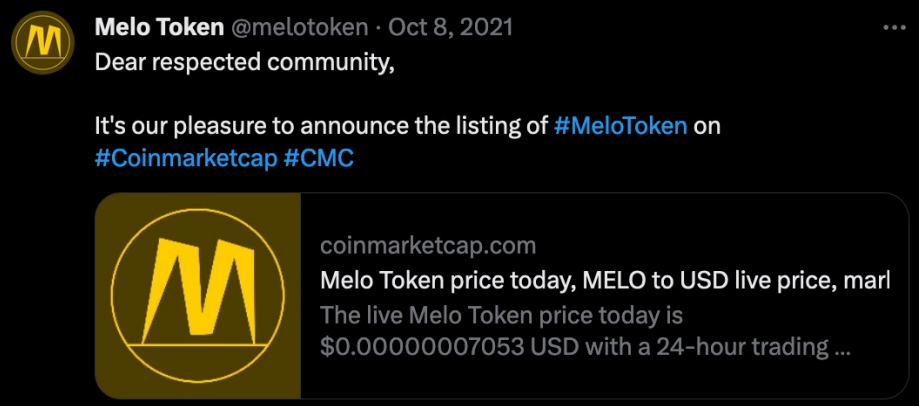


PROJECT:
Melo Token

ON-CHAIN TX:
https://bscscan.com/tx/0x3f1973fe56de5ecd59a815d3b14741cf48385903b0ccfe248f7f10c2765061f7

ROOT CAUSE:
Un-protected “mint” function left public.

TWEET:
https://twitter.com/peckshield/status/1654667621139349505



SANS HALBORN
HACKFEST SUMMIT 2023

Public mint function allowed attacker to create a large amount of tokens to liquddate and convert to USDT.

PROJECT:
Melo Token

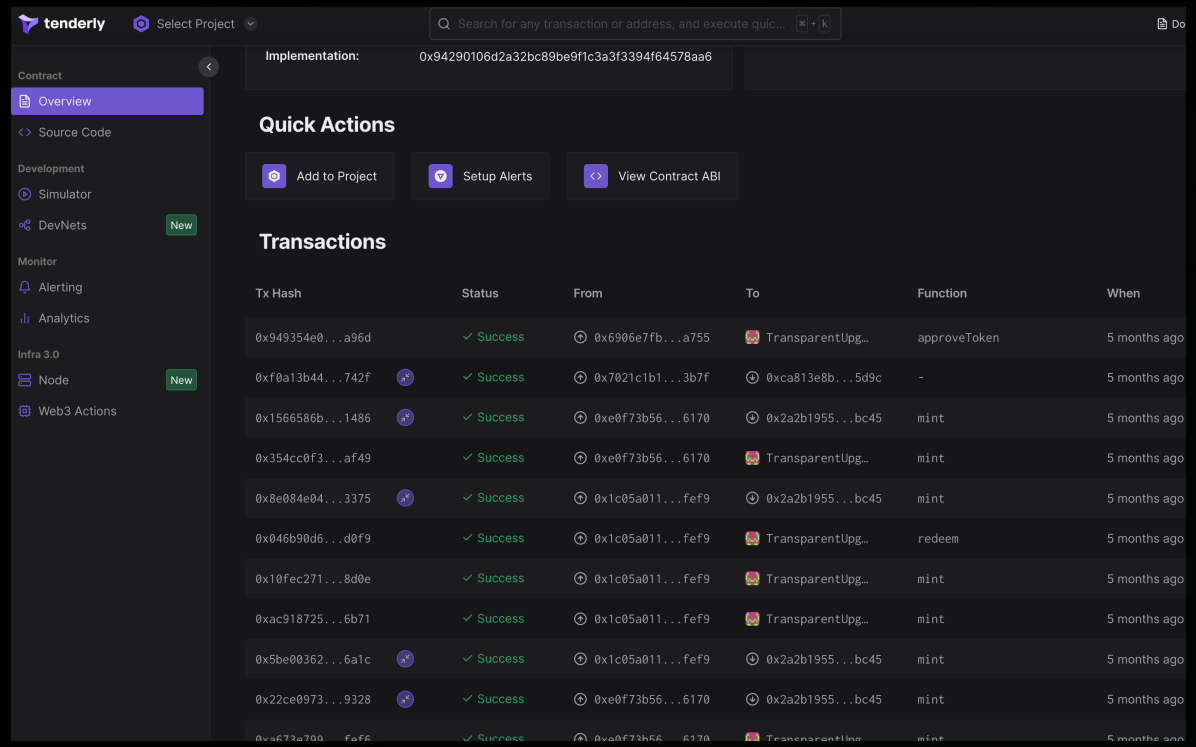
ON-CHAIN TX:
https://bscscan.com/tx/0x3f1973fe56de5ecd59a815d3b14741cf48385903b0ccfe248f7f10c2765061f7

ROOT CAUSE:
Un-protected “mint” function left public.

TWEET:
https://twitter.com/peckshield/status/1654667621139349505

forge test --contracts ./src/test/Melo_exp.sol -vvvv

https://dashboard.tenderly.co/tx/bnb/0x3f1973fe56de5ecd59a815d3b14741cf48385903b0ccfe248f7f10c2765061f7



PROJECT:
DEPUSDT_LEVUSDC

ON-CHAIN TX:
https://etherscan.io/tx/0xf0a13b445674094c455de9e947a25bade75cac9f5176695fca418898ea25742f

ROOT CAUSE:
Approval of a malicious contract

TWEET:
https://twitter.com/NumenAlert/status/1669277328416051201

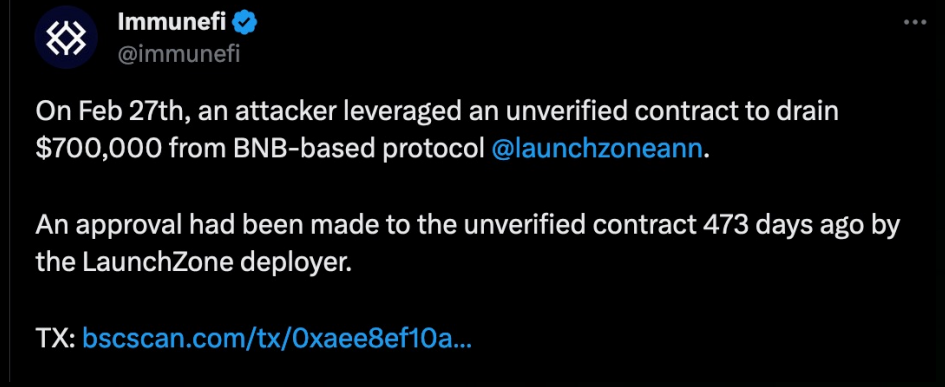
Txn Type: 2 (EIP-1559) | Nonce: 38 | Position In Block: 90

#	Name	Type	Data
0	token	address	0x7b190a928Aa76EeCE5Cb3E0f6b3BdB24fcDd9b4f
1	spender	address	0xF248C52ebDDB098E53bE91365DDe4Ccb48873285
2	_amount	uint256	100000000000000000000000000

Switch Back

SANS HALBORN
HACKFEST SUMMIT 2023

forge test --contracts ./src/test/DEPUSDT_LEVUSDC_exp.sol -vvvv



The attacker called the function `0x4f1f05bc` on the unverified contract, which had been approved for access by the LaunchZone deployer prior.

This allowed the attacker to transfer 9,886,961 LZ of LaunchZone's funds to the Biswap LZ-BUSD pool.

The attacker then swapped 50 BUSD in the same pool to extract 9,886,999.87 LZ. After forcing LaunchZone's contract to perform a bad swap and draining the pool, the attacker exchanged the stolen LZ tokens for 87,911.041 BUS

PROJECT:
LaunchZone

ON-CHAIN TX:
https://bscscan.com/tx/0xaee8ef10ac816834cd7026ec34f35bdde568191fe2fa67724fcf2739e48c3cae

ROOT CAUSE:
Approval of an unverified contract to call priveleged functions.

TWEET:
https://twitter.com/immunefi/status/1630210901360951296

The unverified contract https://bscscan.com/address/0x6d8981847eb3cc2234179d0f0e72f6b6b2421a01 has already now been used to attack 3 separate projects which have given it approvals.

0x1C2B102f22c08694EEe5B1f45E7973b6EACA3e92

forge test --contracts src/test/LaunchZone_exp.sol -vvv

PROJECT:
TrustPad

ON-CHAIN TX:
https://bscscan.com/address/0x1694d7fabf3b28f11d65deeb9f60810daa26909a

ROOT CAUSE:
The receiveUpPool function did not verify msg.sender, allowing the attacker to manipulate newlockstartTime

TWEET:
https://twitter.com/BeosinAlert/status/1721800306101793188

The receiveUpPool function did not verify msg.sender, allowing the attacker to manipulate newlockstartTime.

The attacker repeatedly called receiveUpPool() and withdraw() to collect rewards, then called stakePendingRewards to convert the rewards into staking amounts.

Finally, the attacker withdrew the rewards via withdraw().

forge test --contracts ./src/test/TrustPad_exp.sol -vvvv

```
    │   └─ ← ()
    ├─ [116418] LaunchpadLockableStaking::receiveUpPool(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], 41295609937 [4.129e10])
    │   ├─ [115587] 0x129F4ac88b0446f9B46B176c93531e6cF4687657::receiveUpPool(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], 41295609937 [4.129e10]) [delegatecall]
    │   │   ├─ [450] ContractTest::isLocked(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
    │   │   │   └─ ← true
    │   │   ├─ [2020] ContractTest::depositLockStart(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
    │   │   │   ├─ [828] LaunchpadLockableStaking::depositLockStart(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [delegatecall]
    │   │   │   │   ├─ [0] 0x0000000000000000000000000000000000000000::depositLockStart(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [delegatecal
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← ()
    │   │   │   └─ ← 0
    │   │   ├─ emit Locked(param0: ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], param1: 41295609937 [4.129e10], param2: 94608000 [9.46e7], param30)
    │   │   ├─ [83619] TPAD::transferFrom(ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], LaunchpadLockableStaking: [0xE613c058701C768E0d04D1bf8e6a61a0C6d48A], 41295609937 [4.129e10])
    │   │   │   ├─ emit Transfer(from: ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], to: LaunchpadLockableStaking: [0xE613c058701C768E0d04D1bf8e6ac1a0C6d48A], value: 41295609937 [4.129e10])
    │   │   │   ├─ emit Approval(owner: ContractTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], spender: LaunchpadLockableStaking: [0xE613c058701C768E0d04D1
```

SANS HALBORN

HACKFEST SUMMIT 2023

The root cause is bad access check because the attacker was able to bypass the permit check in the leverage function of the DebtManager contract by directly passing a fake market address without validation, and changing the _msgSender to the victim address. Then, in the untrusted external call, the attacker reentered the crossDeleverage function in the DebtManager contract and steal the collaterals from the _msgSender.

```
101   ) external permit(market, borrowAssets, marketPermit) permit(market.asset(), depo
102     leverage(market, deposit, ratio);
103   }
104
105   /// @notice Leverages the floating position of `_msgSender` a certain `ratio` by
106   /// from Balancer's vault.
107   /// @param market The Market to leverage the position in.
108   /// @param deposit The amount of assets to deposit.
109   /// @param ratio The number of times that the current principal will be leveraged
110   /// @param borrowAssets The amount of assets to allow this contract to borrow on
111   /// @param marketPermit Arguments for the permit call to `market` on behalf of `_
112   /// Permit `value` should be `borrowAssets`.
113   function leverage(
114     Market market,
115     uint256 deposit,
116     uint256 ratio,
117     uint256 borrowAssets,
118     Permit calldata marketPermit
119   ) external permit(market, borrowAssets, marketPermit) msgSender {
120     market.asset().safeTransferFrom(msg.sender, address(this), deposit);
121     noTransferLeverage(market, deposit, ratio);
122   }
123
124   /// @notice Leverages the floating position of `_msgSender` a certain `ratio` by
125   /// from Balancer's vault.
126   /// @param market The Market to leverage the position in.
127   /// @param deposit The amount of assets to deposit.
```

PROJECT:
Exactly

ON-CHAIN TX:
https://explorer.phalcon.xyz/tx/optimism/0x3d6367de5c191204b44b8a5cf975f257472087a9aadc59b5d744ffdef33a520e?line=277&debugLine=277

ROOT CAUSE:
The attacker was able to bypass the permit check in the leverage function of the DebtManager contract by directly passing a fake market address without validation

TWEET:
https://twitter.com/BlockSecTeam/status/1692533280971936059

FAKE MARKET CONTRACT –

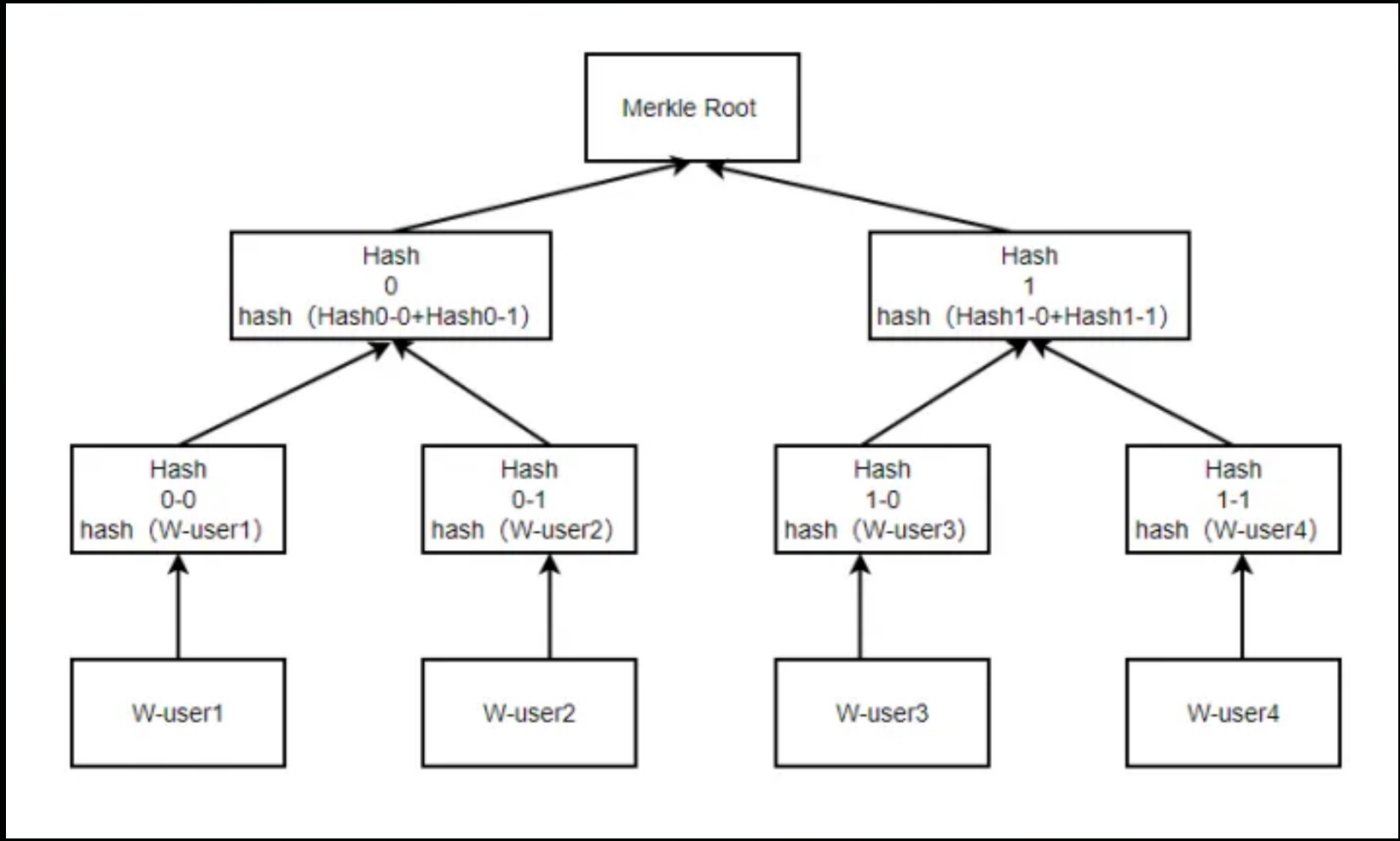
https://optimistic.etherscan.io/address/0x9b88627ec60a7a45482c593e0f5d88da9bf66dbd#code

Types of access control in Solidity

3) Merkle Tree Validation

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.



3) Merkle Tree Validation

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.

```
// 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.

3) Merkle Tree Validation

```
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);
}
```

`keccak256(abi.encodePacked(sender,maxAmount.toString()))`

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.

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.

3) Merkle Tree Validation

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 The multiproof provided is not valid.
     */
    error MerkleProofInvalidMultiproof();

    /**
     * @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 pur
        return processProof(proof, leaf) == root;
    }

    /**
     * @dev Calldata version of {verify}
     */
    function verifyCalldata(bytes32[] calldata proof, bytes32 root, bytes32 leaf) internal pure returns (bool) {
        return processProofCalldata(proof, leaf) == root;
```

```
    /**
     * @dev Sorts the pair (a, b) and hashes the result.
     */
    function _hashPair(bytes32 a, bytes32 b) private pure returns (bytes32) {
        return a < b ? _efficientHash(a, b) : _efficientHash(b, a);
    }

    /**
     * @dev Implementation of keccak256(abi.encode(a, b)) that doesn't allocate or expand memory.
     */
    function _efficientHash(bytes32 a, bytes32 b) private pure returns (bytes32 value) {
        /// @solidity memory-safe-assembly
        assembly {
            mstore(0x00, a)
            mstore(0x20, b)
            value := keccak256(0x00, 0x40)
        }
    }
}
```

SANS HALBORN

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/MerkleProof.sol

3) Merkle Tree Validation - GONE WRONG

The 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.


```
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);
}
```


https://etherscan.io/address/0xdd5a649fc076886dfd4b9ad6acfc9b5eb882e83c#code

3) Merkle Tree Validation - GONE WRONG

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;
}
```

Vulnerability = signature impersonation on whitelist access verification via Merkle Proof.

The signature can be impersonated due to the vData memory parameter info not being verified by msg.sender when the parameter is passed.

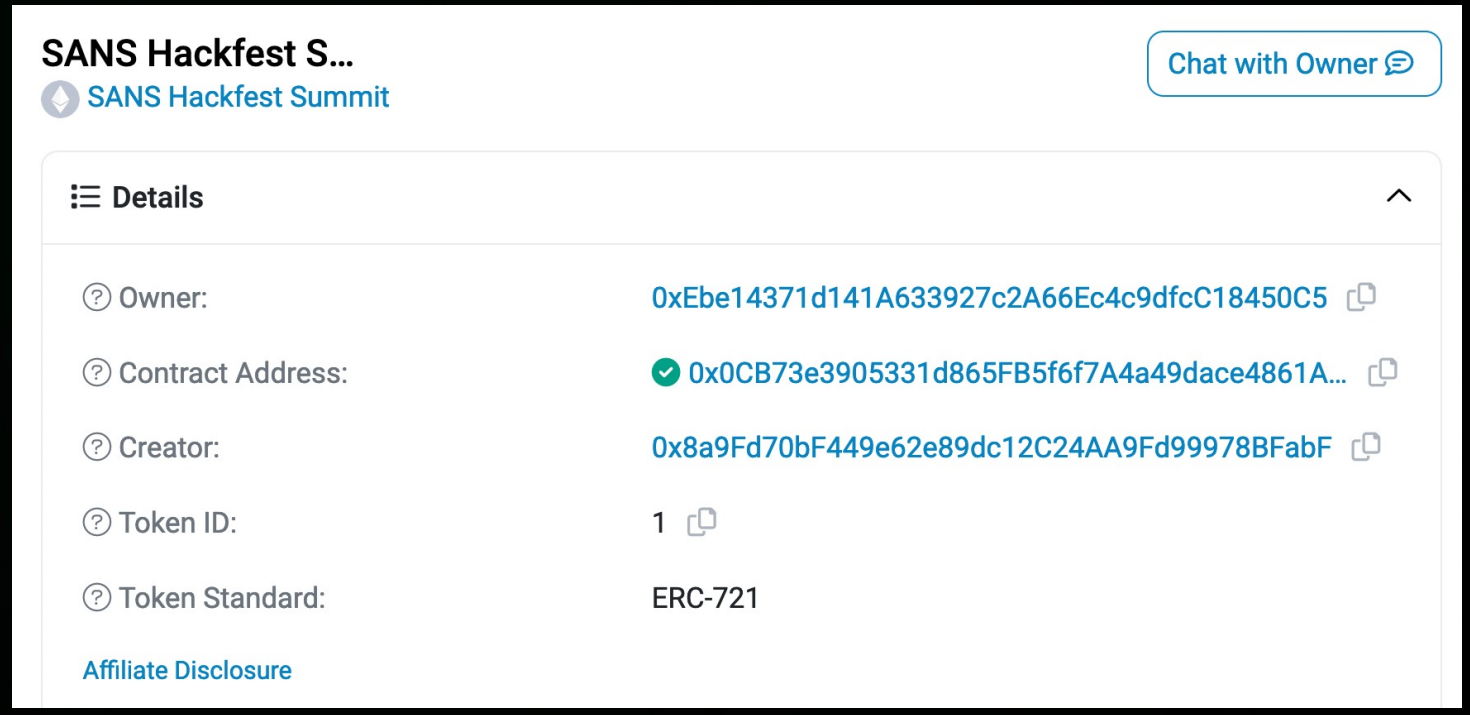
➢WORKSHOP

Get Code and isntructions at:

https://github.com/HalbornAcademy/HackFest2023NFT

https://ipfs.io/ipfs/QmeEc1fTfRofeiAo2BbhytadnHQivq4n5tprBbxRaxETcP/hackfestgif_14MB.gif

Bypass the Merkle Root Validation to Mint a SANS Hackfest 2023 NFT from the smart contract.



SANS HALBORN

QUESTIONS?
COME TALK TO ME!

THANK YOU!

STEVEN WALBROEHL

CO-FOUNDER @ HALBORN