18 Adversarial Tests That Try to Steal From Our DEX

Most test suites prove that software works when you use it correctly. That's table stakes. The tests that actually matter are the ones that try to **break** it — and fail.

VibeSwap is an omnichain DEX built on LayerZero V2 that eliminates MEV through commit-reveal batch auctions. We had 476 passing tests. Nearly all of them were happy-path: deposit tokens, swap, withdraw, collect fees. Everything works as designed.

But "works as designed" doesn't mean "can't be exploited." So we wrote 18 tests that attempt real attacks against every money path in the system — and assert the attacker walks away with nothing.

The Philosophy: Prove the Defense, Not the Feature

A traditional test says: "*When I add liquidity, I get LP tokens.*"

An adversarial test says: "*When I add liquidity, manipulate the price, then remove liquidity in the same block — can I extract more value than I put in?*"

The difference matters. The first test passes even if the system is vulnerable. The second test **fails if the defense is missing** — which means it's actually testing security, not just functionality.

Every test in this suite follows the same pattern:

1. **Set up the attack** — fund the attacker, position them to exploit
2. **Execute the attack** — try the actual exploit technique
3. **Assert the attacker cannot profit** — balances, invariants, reverts

If you can comment out a security check and watch the test fail, the test is doing its job.

Section 1: AMM Fund Safety

The Flash Loan Sandwich

Attack: Add liquidity, swap, remove liquidity — all in the same block. Classic flash loan sandwich that extracts value from price impact.

Defense: VibeAMM tracks a `sameBlockInteraction` mapping keyed by `(user, poolId, blockNumber)`. The `noFlashLoan` modifier checks this on every liquidity and swap operation. Second interaction in the same block reverts with `SameBlockInteraction()`.

```
// Step 1: Add liquidity — succeeds
amm.addLiquidity(poolId, 5 ether, 10_500 ether, 0, 0);

// Step 2: Swap in same block — BLOCKED
vm.expectRevert(VibeAMM.SameBlockInteraction.selector);
amm.swap(poolId, address(weth), 1 ether, 0, attacker);
```

The attacker's three-step sandwich becomes a one-step deposit. No extraction possible.

First Depositor Inflation

Attack: The classic ERC-4626 vault inflation attack. First depositor adds 1 wei of liquidity, donates a large amount directly to the contract to inflate the share price, then the next depositor's deposit rounds down to 0 shares.

Defense: Two layers. First, `MINIMUM_LIQUIDITY = 10000` means any deposit where `sqrt(amount0 * amount1) <= 10000` reverts with `InitialLiquidityTooLow`. You can't bootstrap the attack with dust. Second, `_checkDonationAttack()` compares tracked balances against actual balances — if they diverge by more than `MAX_DONATION_BPS` (1%), the next interaction reverts with `DonationAttackSuspected()`.

```
// Tiny deposit reverts
vm.expectRevert(); // InitialLiquidityTooLow
amm.addLiquidity(freshPoolId, 1, 1, 0, 0);

// After legitimate deposit + direct token transfer (donation)
weth.transfer(address(amm), 100 ether);

// Next addLiquidity catches the discrepancy
vm.expectRevert(VibeAMM.DonationAttackSuspected.selector);
amm.addLiquidity(freshPoolId, 1 ether, 1 ether, 0, 0);
```

Rounding Theft via Micro-Swaps

Attack: Execute 100 round-trip swaps with tiny amounts (1000 wei each), hoping integer division rounding accumulates in the attacker's favor.

Defense: `BatchMath.getAmountOut` uses `numerator / denominator` which truncates — always rounding **down** the output. The pool keeps the dust. Over 100 round trips, the attacker bleeds value to fees and rounding, never gains.

```
// After 100 round-trip swaps of 1000 wei each
assertLe(attackerWethEnd, attackerWethStart, "Attacker WETH must not increase");

// Pool K can only grow (fees accumulate)
assertGe(kAfter, kBefore, "Pool K must not decrease");
```

Trade Size Limits

Attack: Submit a swap for 15% of pool reserves to cause massive slippage and potential price manipulation.

Defense: `MAX_TRADE_SIZE_BPS = 1000` (10%). Any single swap exceeding 10% of the input reserve reverts with `TradeTooLarge(maxAllowed)`.

Section 2: Auction Fund Safety

Deposit Double-Spend

Attack: Deposit 10 WETH into VibeSwapCore, commit to a batch, then commit again with the same 10 WETH before settlement.

Defense: `commitSwap` calls `safeTransferFrom` which moves actual tokens from the user. After the first commit, the user has 0 WETH. The second commit fails at the ERC-20 transfer level — the accounting is tied to real token movement, not an internal balance that could be manipulated.

Slash Accounting Conservation

Attack: Commit 1 ETH, deliberately don't reveal, check if the slash creates or destroys value.

Defense: `SLASH_RATE_BPS = 5000` (50%). The test verifies exact accounting:

```
// 50% to treasury, 50% refund to user
assertEq(treasuryAfter - treasuryBefore, expectedSlash, "Treasury must receive exactly 50%");
assertEq(attackerEthAfter - attackerEthBeforeSlash, expectedRefund, "Attacker must receive exactly
50% refund");

// Conservation law: deposit = slash + refund
uint256 totalAccounted = (treasuryAfter - treasuryBefore) + (attackerEthAfter -
attackerEthBeforeSlash);
assertEq(totalAccounted, depositAmount, "No value created or destroyed");
```

This is a conservation-of-value test. If the sum doesn't equal the original deposit, money appeared from nowhere or vanished.

Wrong Secret = Immediate Slash

Attack: Commit with secret X, reveal with secret Y.

Defense: The reveal recomputes `keccak256(abi.encodePacked(sender, tokenIn, tokenOut, amountIn, minAmountOut, secret))` and compares it to the stored commit hash. Mismatch triggers `_slashCommitment` immediately — no waiting for settlement. The attacker loses 50% on the spot.

Priority Bid Underpayment

Attack: Claim a 1 ETH priority bid but send `msg.value = 0`.

Defense: `if (msg.value < priorityBid) revert InsufficientPriorityBid()`. You can't claim priority without paying for it.

Section 3: Treasury Fund Safety

Double-Commitment of Funds

This is the most interesting treasury test. `queueWithdrawal` checks `address(this).balance >= amount` at queue time but **doesn't escrow the funds**. So from a 10 ETH treasury, you can queue two withdrawals of 10 ETH each — both pass the balance check.

Attack: Queue 10 ETH to recipient A and 10 ETH to recipient B. Execute both after timelock.

Defense: The first execution drains the treasury. The second execution's `call{value: 10 ether}` fails because there's 0 ETH left, and the `require(success, "ETH transfer failed")` catches it.

```
// First execute drains treasury
treasury.executeWithdrawal(reqA);
assertEq(address(treasury).balance, 0, "Treasury is empty");

// Second execute fails — no ETH to send
```

```
vm.expectRevert("ETH transfer failed");
treasury.executeWithdrawal(reqB);

// Total outflow = exactly 10 ETH, not 20
assertEq(recipientA.balance + recipientB.balance, 10 ether,
    "Total outflow must equal original treasury balance");
```

No double-spend occurs, but the over-commitment is observable on-chain. This is the kind of thing that could matter in a governance context — queued withdrawals that can never execute create noise.

Timelock Enforcement

Attack: Queue withdrawal, immediately execute.

Defense: `require(block.timestamp >= request.executeAfter, "Timelock active")`. The test warps past 2 days + 1 second and confirms execution succeeds only then.

Double Execution

Attack: Execute same withdrawal request twice.

Defense: `request.executed = true` on first execution. `require(!request.executed, "Already executed")` blocks the second.

Recipient Immutability

Attack: Owner queues withdrawal to address A. Attacker calls `executeWithdrawal` hoping funds route to `msg.sender`.

Defense: `executeWithdrawal` is permissionless (anyone can trigger it), but always sends to `request.recipient` which was set at queue time. The attacker's balance doesn't change.

Section 4: Reward Distribution Safety

Shapley Value Double Claim

Attack: Call `claimReward` twice for the same game.

Defense: `claimed[gameId][msg.sender] = true` after first claim. Second attempt reverts with `AlreadyClaimed`.

Overpayment Check

Attack: Verify that Shapley values for 5 participants don't sum to more than `totalValue`.

Defense: The distribution loop gives the last participant the remainder (`totalValue - distributed`), which mathematically guarantees the sum equals exactly `totalValue`. No money created from nothing.

```
// Sum all 5 participants' Shapley values
assertEq(totalAllocated, totalReward, "Sum of Shapley values must equal totalValue exactly");
```

Non-Participant Claim

Attack: Address not in the game tries to claim.

Defense: `shapleyValues[gameId][attacker] == 0`, so `claimReward` reverts with `NoReward`.

Section 5: Oracle and Price Safety

TWAP Deviation Blocking

Attack: Execute a massive swap that moves the spot price far from the TWAP, then continue trading at the manipulated price.

Defense: The `validatePrice` modifier runs after every swap. It consults the TWAP oracle and reverts with `PriceDeviationTooHigh(spotPrice, twapPrice)` if deviation exceeds `MAX_PRICE_DEVIATION_BPS` (5%). The oracle needs warm-up time (multiple observations across different timestamps), but once it has history, manipulation is caught.

Flash Loan Detection

Attack: Add liquidity + swap in the same block (flash loan pattern).

Defense: `noFlashLoan` modifier with per-user, per-pool, per-block tracking. `SameBlockInteraction()` on the second operation.

Max Trade Size

Attack: Swap 15% of reserves in a single trade.

Defense: `MAX_TRADE_SIZE_BPS = 1000` enforces a 10% ceiling. `TradeTooLarge(maxAllowed)` on anything larger.

What These Tests Actually Prove

Each test is a proof-by-contradiction. We assume the attacker can profit, execute the attack, and show the assumption is false. If you disable the defense (comment out the guard), the test fails — confirming it's not a tautology.

The 18 tests cover every path where tokens or ETH move through the system:

Subsystem	Tests	Attack Vectors
AMM	4	Sandwich, inflation, rounding, donation
Auction	4	Double-spend, slash accounting, wrong secret, bid underpayment
Treasury	4	Double-commitment, timelock bypass, double execution, recipient hijack
Rewards	3	Double claim, overpayment, unauthorized claim
Oracle	3	TWAP manipulation, flash loans, oversized trades

Total gas across all 18 tests: ~48M. The heaviest is `roundingTheftManySmallSwaps` at 35M gas (100 round-trip swaps proving rounding always favors the pool). The lightest is `tradeExceedsMaxSize` at 73k gas.

Full suite: 494 tests, 0 failures, 0 regressions.

The Takeaway

Happy-path tests tell you the system works. Adversarial tests tell you it **can't be broken**. If your DeFi protocol doesn't have tests that attempt real attack vectors — sandwich attacks, donation inflation, double-spends, rounding exploitation — then you're testing the marketing, not the security.

Every test in this suite started with the question: "*What if someone tried to steal from this?*" If you can't answer that question with a passing test, you have a vulnerability. You just haven't found it yet.

Test file: `test/security/MoneyPathAdversarial.t.sol` *Source: github.com/WGlynn/VibeSwap*