Moonwell: How a Thin Token and a Donation Trick Pulled $8.7 Million Out of a Lending Market on Base

Summary

Before you read the details, here is the short version of what happened.

- **Protocol and code.** Moonwell is a Compound v2 style lending protocol on Base. The market that got hit is the MAMO Core Market. The mMAMO contract address is 0x2F90Bb22eB3979f5FfAd31EA6C3F0792ca66dA32 and the MAMO token is 0x7300B37DfdfAb110d83290A29DfB31B1740219fE. All the code I quote in this post is based on the moonwell-fi/moonwell-contracts-v2 repo at commit `3b32d534778b854d31bd3555c90cc787b2072566`.

- **Type of weakness.** This was not a bug in the code. It was an economic attack. The attacker used three weak spots. One, a spot price oracle on a thin token. Two, an exchange rate that also counts tokens sent straight to the contract. Three, a supply cap that is only checked when you mint. None of these is a bug on its own. But put the three together and you get a hole that can lose about $9 million.
- **Impact.** The attacker used manipulated MAMO collateral to borrow cbBTC, WETH, USDC and wstETH worth about $11.03 million in total. CertiK and PeckShield estimate the loss at about $8.7 million. Even after the liquidations were done, about $9.13 million of bad debt is still left in those four markets.
- **Status.** This happened on 27 August 2026. That same morning Moonwell cut every borrow cap on Base down to 1 wei. On 30 August 2026 Anthias Labs published a detailed post mortem on the governance forum. As of today, 31 August 2026, there is still no plan for how the suppliers who lost money will get it back.

One thing I want to say before we start. Most of the headlines just called this an “oracle hack”. If you say oracle, you are only half right. The other half is a strange accounting habit that lives in every Compound v2 fork. If you send tokens directly into an mToken contract, every existing share becomes worth more. But the supply cap is not checked, and no event comes out. The attacker mixed both of these weaknesses together. So even if you only fix the oracle, the accounting weakness is still sitting there.

Background

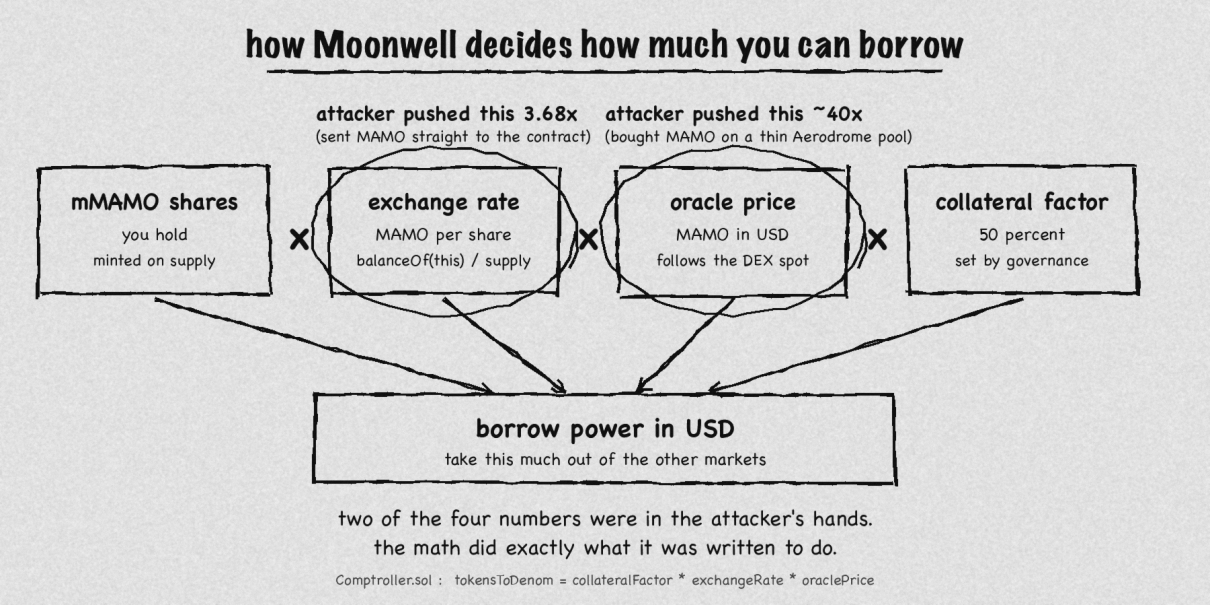
Moonwell is a lending protocol. In simple words, it is a DeFi protocol that gives out loans. When you deposit a token, you get back a receipt token called an mToken. You can then use that deposit as a guarantee and borrow other tokens against it. It is a fork of Compound v2, so if you know how cTokens work, I think this system will be easy to follow.

The way it works is simple:

1. You deposit or supply MAMO into the mMAMO market. The contract gives you mMAMO shares. How many shares you get depends on the exchange rate at that moment.
2. The protocol works out how much your shares are worth in USD. It takes the number of shares, multiplies by the exchange rate, then by the oracle price, then by a safety haircut called the collateral factor (Shares x Exchange Rate x Oracle Price x Collateral

Factor). The collateral factor is a percentage that lowers the value a bit for safety before you are allowed to borrow. For MAMO it was 50 percent. For example, if the protocol values your MAMO collateral at $100,000, then $100,000 x 50% = $50,000 is the most you can borrow.

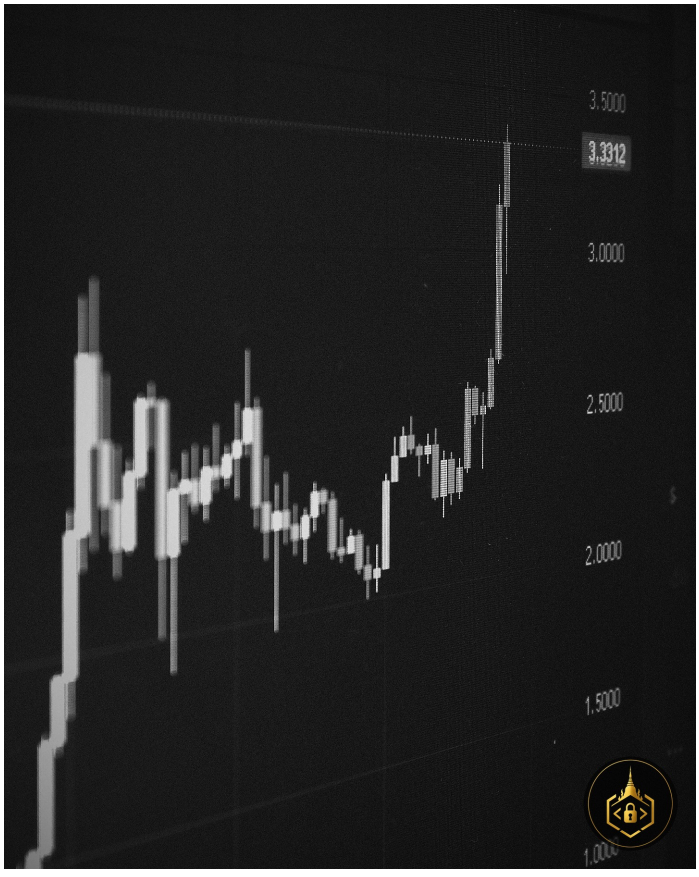
3. Up to that USD value, you can borrow tokens like cbBTC, WETH or USDC from the other markets on Base.



How Moonwell values collateral. Four numbers multiplied together. The attacker controlled two of them.

So for every lender's deposit to be safe, all four of those numbers have to be right. Shares and collateral factor are set by the protocol. The other two, the exchange rate and the oracle price, are data pulled from outside. And that is where the problem starts.

The MAMO token is the token of Mamo, a personal finance app on Base built by the same team behind Moonwell. Not many people trade it and there is not much liquidity. On the day of the attack its whole market cap was only about $6 million, and its 24 hour trading volume was only about $1.18 million. Moonwell allowed MAMO to be used as collateral, set its supply cap at 20 million MAMO, and set the collateral factor at 50 percent.



You can read the docs at https://docs.moonwell.fi/ and the contract list at https://docs.moonwell.fi/moonwell/protocol-information/contracts.

Vulnerable Code

There is no single “vulnerable function” here. You could say there are three parts. If you look at each one on its own, it is just doing its job. But when you put the three together, the attacker ends up in a position where they can set their own borrowing power. Let me show all three parts.

Part 1: the exchange rate trusts the token balance

In `src/MToken.sol`, the exchange rate is worked out like this:

```
// src/MToken.sol, lines 440 to 475 (commit 3b32d53)
function exchangeRateStoredInternal() internal view virtual returns (MathError, uint) {
    uint _totalSupply = totalSupply;
    if (_totalSupply == 0) {
        return (MathError.NO_ERROR, initialExchangeRateMantissa);
    } else {
        /*
         * Otherwise:
         *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
         */
        uint totalCash = getCashPrior();
        // ... add totalBorrows, subtract totalReserves, divide by totalSupply
    }
}
```

And `getCashPrior` in `src/MErc20.sol` is just the raw ERC20 balance:

```
// src/MErc20.sol, lines 205 to 208 (commit 3b32d53)
function getCashPrior() internal view virtual override returns (uint) {
    EIP20Interface token = EIP20Interface(underlying);
    return token.balanceOf(address(this));   // counts EVERY token in the contract
}
```

What this means is, if someone calls `MAMO.transfer(mMAMO, hugeAmount)` without calling `mint`, then `totalCash` goes up, `totalSupply` stays the same as before, and so every existing mMAMO share is suddenly worth more. Think of something that was worth 1,000 and suddenly it is worth 2,000. Moonwell does not emit any event for this. No hook runs. This is just how Compound v2 normally works, and normally there is no danger in it, because sending tokens into a shared pool as a donation is the same as giving your money away to everyone else. But when one person holds almost all the shares, it becomes dangerous.

Part 2: the supply cap only looks at mints

In `src/Comptroller.sol`, this is the cap check inside `mintAllowed`:

```
// src/Comptroller.sol, lines 282 to 296 (commit 3b32d53)
uint supplyCap = supplyCaps[mToken];
if (supplyCap != 0) {
    uint totalCash = MToken(mToken).getCash();
    uint totalBorrows = MToken(mToken).totalBorrows();
    uint totalReserves = MToken(mToken).totalReserves();
    uint totalSupplies = sub_(add_(totalCash, totalBorrows), totalReserves);

    uint nextTotalSupplies = add_(totalSupplies, mintAmount);
```

```
    require(nextTotalSupplies < supplyCap, "market supply cap reached");
}
```

That `mintAllowed` is only called when you mint. If you do a direct transfer, there is no reason for `mintAllowed` to be called at all. So the MAMO cap could not stop the attacker from putting another 53 million MAMO into the contract. The purpose of the cap is to limit how much MAMO the protocol will accept as collateral. In real life it is like this: the front door of the house has a sign saying “only this much allowed”, but the big back door is left open, so they just walked around and came in from the back.

Part 3: the oracle followed a low liquidity DEX price

Collateral value takes the price from `oracle.getUnderlyingPrice(asset)` and then works it out like this:

```
// src/Comptroller.sol, lines 745 to 748 (commit 3b32d53)
// Pre-compute a conversion factor from tokens -> glmr (normalized price value)
vars.tokensToDenom = mul_(
    mul_(vars.collateralFactor, vars.exchangeRate),
    vars.oraclePrice
);
```

The MAMO price feed kept updating to follow the spot trading price on the DEX. The post mortem does not name the feed provider, but it does give the numbers. During the attack MAMO went from $0.010597 up to a peak of $0.43127363. The highest price Moonwell accepted as valid was $0.40248571. There was no time weighted average (TWAP), no sanity band to block a huge jump in price, and no liquidity check that would say “this token cannot go up 40 times in two hours”.

Put the three parts together and the formula in Part 3 becomes `0.5 x (an exchange rate the attacker pushed up 3.68x) x (a price the attacker pushed up 38x)`. That is the main cause of the whole exploit.

Root Cause

The rule that should hold

Every lending market has one basic rule:

The USD value the protocol gives to your collateral should be close to what you would really get if you were liquidated and that collateral was sold on the market.

On 27 August 2026 that rule broke twice at the same time.

- **The oracle side.** The MAMO price feed was showing $0.40. But if you looked at the real market depth, you could not sell even a small part of 94 million MAMO at that price. The liquidators found this out. They seized almost all of the attacker's mMAMO and still could not get most of the debt back, because the MAMO they seized was worth only a small part of what the oracle had shown.
- **The accounting side.** The exchange rate said each mMAMO share was backed by 0.0755 MAMO. Technically that was true, because the attacker really did donate the MAMO into the contract. But that extra MAMO went past the 20 million supply cap the risk managers had set. And as I said before, the whole point of that cap is to stop the protocol from holding too much MAMO.

It is clearer if you split the symptom from the cause:

- Symptom: the attacker borrowed assets worth $11 million while spending only around $7.5 million on buying MAMO, and they got most of that back later by selling the MAMO again.
- Cause: two of the four inputs in the collateral formula were things the attacker could change themselves. And there was no guard rail on either input to stop a sudden huge move.

Why this is not "just an oracle bug"

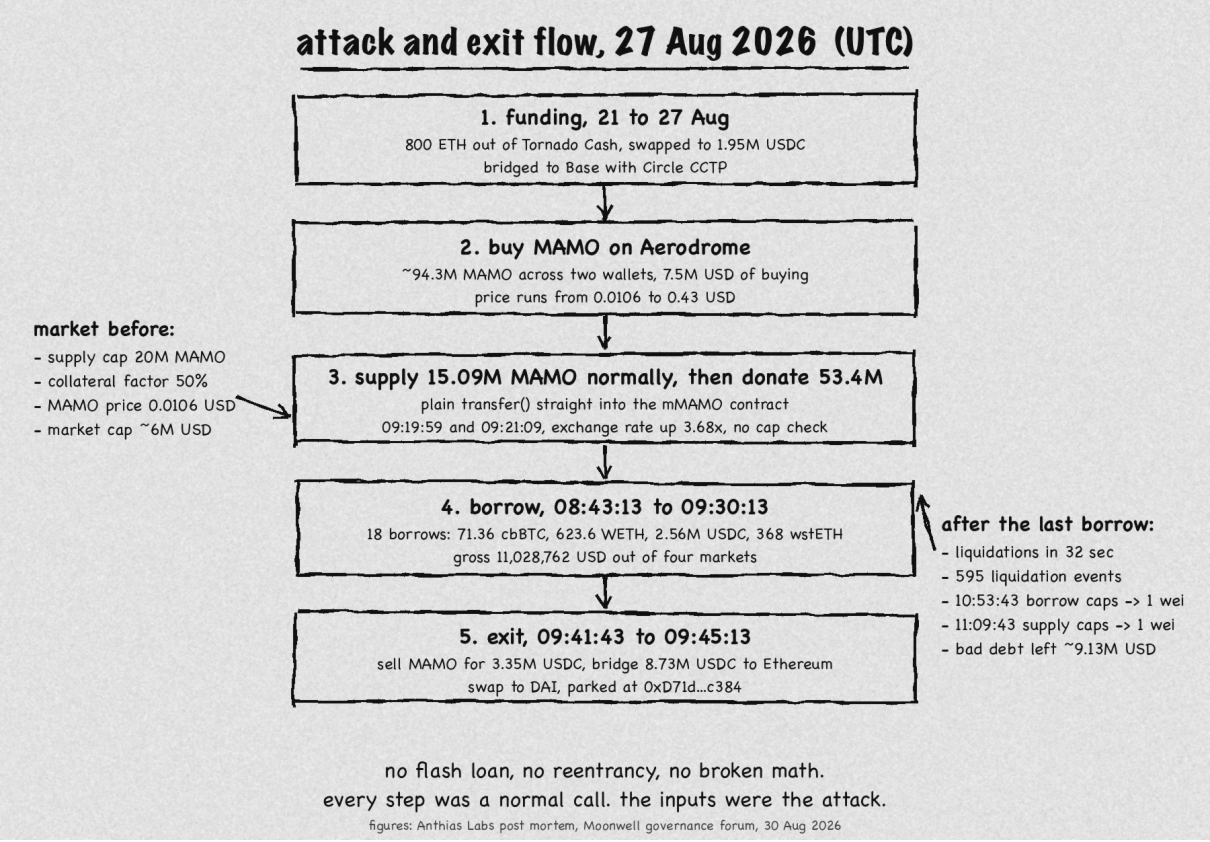
If the oracle had been perfect and only the donation trick existed, the attacker would have got 3.68 times more collateral than the supply cap allowed, at a fair price. That would not have been as big as what we have now, it would have been a much smaller problem. If only the oracle was the problem and the donation trick did not exist, the attacker would have been stuck at the 20 million MAMO supply cap. At a MAMO price of $0.40 and a 50 percent collateral factor, that is 20M x 0.40 x 50% = about $4 million of borrowing, not the $11 million we got. The two weaknesses combined into one big one. That is why the fix has to cover both.

The real design mistake was using a thin, low liquidity token as collateral

Step back one more level and the main problem is that MAMO was accepted as collateral at all. MAMO's market cap was only around $6 million and its daily trading volume was just over a million dollars. Moonwell allowed up to 20 million of it as collateral at 50 percent. An attacker who started with $2 million was able to manipulate the price about 40 times in a single morning. A token where that is possible should not be collateral for real assets like cbBTC and WETH at that size, no matter how good the oracle is.

Exploit Walkthrough

Because this was an economic attack, there is no flash loan and no reentrancy in it. It was done slowly and patiently. If you think about it, it is boring, and honestly a bit funny too, as long as it is not your money.



The full attack and exit flow. Every figure is from the Anthias Labs post mortem.

1. **Funding, 21 to 27 August.** The attacker's funding wallet pulled 800 ETH out of Tornado Cash in several withdrawals. They swapped 799 ETH into 1,947,391 USDC on CoW Swap and bridged it to Base through Circle's CCTP. Ethereum burn: 0x6fd85b1f… 505b7205. Base mint: 0x79195be2…e12d8852.

2. **Buy MAMO.** The main wallet, 0x719eae70…84BE919D, did 47 swaps on the Aerodrome MAMO/USDC pool and collected about 86.9 million MAMO. A second wallet, 0xD71dd9b6…e338c384, bought more. Together they ended up with about 94.3 million MAMO after about $7.5 million of gross buying. Because the pool was so thin, all that buying pushed the price up from about one cent to around 40 times that. The oracle just followed it up.

3. **Supply the normal way.** The attacker supplied about 15.09 million MAMO into the mMAMO market and got shares. The market was already close to its 20 million cap with

other people's deposits, so like I said above, this was about as much as the front door would allow.

4. **Donate from the back.** At 09:19:59 UTC and 09:21:09 UTC the attacker sent 53,393,290 MAMO straight to the mMAMO contract with plain `transfer` calls. The transactions are 0x056597f4…de109ce9 and 0xaf284d7f…61de3cd9. No mint, no cap check. The exchange rate jumped from about 0.020513 to 0.075460 MAMO per share, a 3.68 times increase. Since the attacker held most of the shares, most of that jump went to them.

5. **Borrow.** Between 08:43:13 and 09:30:13 UTC the attacker made 18 borrows across 17 transactions. First borrow: 0x09687d74…a395593e (a small 0.5 cbBTC test). Last borrow at 09:30:13: 0xee2b7564…c18c88b18 (990,000 USDC). In total: 71.36 cbBTC, 623.60 WETH, 2,560,000 USDC and 368 wstETH, worth $11,028,762 at the time.

6. **Exit.** Between 09:41:43 and 09:45:13 UTC the attacker sold MAMO for 3,345,134 USDC, then burned 8,729,453 USDC on Base through Wormhole's CCTPv2 bridge (0x9cd2fbe0…567f1dec and 0xfcb2ff81…e40ecac03), received 8,728,318 USDC on Ethereum, and swapped it to DAI on Velora (0xd8fa6fbe…676ca8622). The DAI is now sitting in the 0xD71d wallet on Ethereum.

7. **Liquidations.** 32 seconds after the last borrow, liquidators started coming in. There were 595 liquidation events across 594 transactions. They seized nearly 100 percent of the attacker's minted mMAMO and got back 17.76 cbBTC, 38.57 WETH, 215,841 USDC and 25.31 wstETH. That alone was nowhere near enough, because once the pump fell back down, the seized MAMO was worth almost nothing.

8. **Emergency brakes.** At 10:53:43 UTC the borrow caps on every Core Market on Base went to 1 wei. At 11:09:43 UTC the MAMO and WELL supply caps went to 1 wei too.

If you are curious, you can look up every one of these addresses on Basescan or Etherscan.

Role	Address	Notes
mMAMO market	0x2F90Bb22eB3979f5FfAd31EA6C3F0792ca66dA32	"Moonwell MAMO" in the docs
MAMO token	0x7300B37DfdfAb110d83290A29DfB31B1740219fE	the thin collateral
Comptroller	0xfBb21d0380beE3312B33c4353c8936a0F13EF26C	holds the caps and collateral factors

Role	Address	Notes
Moonwell price oracle	0xEC942bE8A8114bFD0396A5052c36027f2cA6a9d0	“Moonwell: Chainlink Oracle” on Basescan
Attacker main wallet	0x719eae70d4A83f35bF82A2740699F5db84BE919D	bought MAMO, supplied, donated, borrowed
Attacker second wallet	0xD71dd9b6e634412713c47fe7ae02c628e338c384	holds the DAI on Ethereum

Proof of Concept

Since this is a live protocol with about $9 million of bad debt still open, I am not publishing a working exploit script. But the basic shape of a Foundry test is simple, so I wrote it out here so you can understand the mechanics. Treat it as a study sketch. It is not something you can copy, paste and run.

```
// ILLUSTRATIVE ONLY. Not tested, not runnable as is.
// Idea: on a Base fork before block ~50.5M, show that a plain transfer
// inflates the exchange rate and that the supply cap does not stop it.

function test_donationInflatesExchangeRate() public {
    uint256 before = mMAMO.exchangeRateStored();

    // Step 1: supply up to the cap the normal way. This passes mintAllowed.
    mamo.approve(address(mMAMO), 15_000_000e18);
    mMAMO.mint(15_000_000e18);

    // Step 2: a second mint past the cap reverts, as designed.
    vm.expectRevert("market supply cap reached");
    mMAMO.mint(1_000_000e18);

    // Step 3: a plain transfer of the same tokens does NOT revert.
    mamo.transfer(address(mMAMO), 53_000_000e18);

    // Step 4: every existing share is now worth ~3.68x more MAMO.
    uint256 after_ = mMAMO.exchangeRateStored();
    assertGt(after_, before * 3);

    // Step 5: account liquidity, and so borrow power, went up with it.
    (, uint256 liquidity, ) = comptroller.getAccountLiquidity(address(this));
    // liquidity is now roughly 0.5 * 68M MAMO * oraclePrice, not 0.5 * 20M
}
```

The oracle part is not something you can reproduce in a unit test. It was $7.5 million of real buying on a real pool. You could say the onchain data above is the proof for that part.

Impact

- **Funds lost.** Total assets borrowed were worth about $11,028,762. Even after the liquidations, about $9,131,342 of bad debt is left: 53.60 cbBTC, 585.17 WETH, 2,345,281 USDC and 342.75 wstETH. After taking out what the attacker spent buying MAMO, the stablecoin profit that can be traced is about $6.78 million. CertiK and PeckShield's headline number is around $8.7 million.
- **Who gets hurt.** Everyone who supplied cbBTC, WETH, USDC or wstETH to Moonwell on Base can be hit. Bad debt in a Compound v2 market is not owned by any one person, and there is no rule that says who it gets taken from. So unless the protocol fills the hole from its treasury, the last suppliers to withdraw are the ones who can end up taking the loss. Mamo app users also could not withdraw USDC for a while, because the app puts its deposits into Moonwell.
- **A loss bigger than a year of revenue.** If you add up Moonwell's yearly fees, it comes to only around $8.6 million. One attack in one morning cost more than that.
- **Third time in eleven months.** November 2025, you could say a wrsETH oracle failure, and it left about $3.7 million of bad debt. February 2026, a governance proposal misconfigured the cbETH oracle to use the cbETH/ETH rate alone, so the cbETH price went to about $1.12 and the liquidation bots ate it up. That one left about $1.78 million of bad debt. That incident got extra criticism because the pull requests were written with an AI coding assistant. August 2026, this one. The suppliers who lost money in February were still arguing about who should pay them back when the MAMO attack happened on top of it. When it rains, it pours.
- **Token prices.** WELL dropped about 13 percent in 24 hours. MAMO dropped only about 9 percent, but compared to the 40x pump and dump the attacker had just done, that is still a big move.

- **Still exploitable?** Not for now. With every borrow cap at 1 wei, nobody on Base can borrow anything, attacker or not, nobody can do anything. But that is only a tourniquet to stop the bleeding, it is not a cure. The main weak points, the code and the oracle design, have not changed. They are still live right now.

The Fix

As of 31 August 2026 Moonwell has published a post mortem but not a remediation plan. So what is written below is how I think it should be fixed, split into two parts. This is just my opinion.

Part one: stop the donation trick from creating collateral

The cleanest fix is to stop counting tokens the protocol never accepted. Compound v2 forks use two methods for this problem. The first one is, instead of reading `balanceOf` to work out the underlying token amount, the protocol keeps its own internal ledger of the underlying tokens:

```
// ILLUSTRATIVE. Track what was actually deposited, not what happens to be in the
contract.
```

```
- function getCashPrior() internal view virtual override returns (uint) {
-     return EIP20Interface(underlying).balanceOf(address(this));
- }
+ uint256 internal trackedCash;
+
+ function getCashPrior() internal view virtual override returns (uint) {
+     return trackedCash;                    // donations do not move this
+ }
+ // update trackedCash inside doTransferIn / doTransferOut only
```

The second one is to make the supply cap work not just at mint time but also at valuation time, meaning the moment the value is calculated. If the amount of underlying token in the market is more than the cap allows, the extra part should not count as collateral:

```
// ILLUSTRATIVE. In getHypotheticalAccountLiquidityInternal, cap the value a market can
back.
+ uint cap = supplyCaps[address(asset)];
+ uint held = asset.getCash() + asset.totalBorrows() - asset.totalReserves();
+ if (cap != 0 && held > cap) {
+     // scale this market's collateral down so the whole market is worth at most `cap`
tokens
+     vars.exchangeRate = mul_(vars.exchangeRate, Exp({mantissa: cap * 1e18 / held}));
+ }
```

Fixing either one of these would have stopped the 3.68x multiplier.

Part two: do not let a thin token set its own price

- **For thin assets, use a price that is not easy to manipulate.** A time weighted average, or a feed that is capped so it cannot go up or down more than a set percentage per hour, or both. A 40x jump in two hours should be rejected as a wrong price, not accepted.
- **Set collateral by liquidity, not by hope.** A token with only about $1 million of daily trading volume should have a borrow ceiling somewhere in the low hundreds of thousands of dollars, or a collateral factor of 0. Aave's isolation mode and debt ceilings exist for exactly this kind of risk.
- **Watch for donations.** If there had been a monitor that fires when `balanceOf(mToken)` jumps without a matching `Mint` event, it would have flagged the 09:19:59 transfer more than eight minutes before the last borrow.

What I wrote above is not code that Moonwell has actually merged. It is just my view of what the fix should look like, so it may be missing things or have extra. When Moonwell publishes their real code changes I will update this post.

Takeaways

1. **“No code was hacked” does not mean the code is all fine.** Every function did its own job exactly as written. The problem was in the design. It let the attacker pick the inputs. When you audit a lending market, audit the inputs to the collateral formula as carefully as you audit the formula itself.
2. **Every Compound v2 fork has the donation quirk.** `getCashPrior` reading `balanceOf(address(this))` is found in Compound, Moonwell, Venus, Benqi and most other Compound v2 forks. Normally it is not dangerous. But the moment one account holds most of the shares in a market, it becomes dangerous, and that tends to happen with low liquidity, low volume tokens. Grep your code for `getCashPrior` and `balanceOf(address(this))`.
3. **A supply cap that is only checked on mint cannot really be called a limit.** It is more like a suggestion. If your risk model says “the protocol will never hold more than 20 million of this token”, that limit has to be enforced everywhere the token’s value is used, not only at deposit time.
4. **Low liquidity tokens should not be collateral for big loans.** The real mistake was listing a token with only a $6 million market cap as collateral for cbBTC and WETH at 50 percent. When one wallet with $2 million can manipulate the underlying asset by about 40x, oracle quality does not matter much anymore.
5. **Three oracle incidents in eleven months is a process problem.** The wrsETH, cbETH and MAMO failures were all different in how they happened, but all the same in kind: a price the protocol trusted was wrong. So I think token listing and oracle risk management need the same kind of review culture as Solidity code review.
6. **Follow the money.** You can see the whole story: in from Tornado Cash, CoW Swap, CCTP to Base, 47 token buys, two donations, 18 borrows, money out through CCTPv2, parked as DAI. Once you know how to read transactions on the blockchain, you can figure out a whole attack in a short time.

Disclosure Timeline

Date (UTC)	Event
21 to 27 Aug 2026	Attacker funds up: 800 ETH from Tornado Cash, swapped to 1.95M USDC, bridged to Base.
27 Aug 2026, 08:43:13	First borrow, a 0.5 cbBTC test.
27 Aug 2026, 09:19:59 and 09:21:09	53.4M MAMO sent directly to the mMAMO contract. Exchange rate up 3.68x.
27 Aug 2026, 09:28:43	MAMO price feed peaks at $0.4313.
27 Aug 2026, 09:30:13	Last borrow. Gross out: $11,028,762. Liquidations begin 32 seconds later.
27 Aug 2026, 09:41:43 to 09:45:13	8.73M USDC bridged to Ethereum and swapped to DAI.
27 Aug 2026, ~10:00	CertiK and PeckShield flag the exploit publicly. Moonwell posts that it is investigating.
27 Aug 2026, 10:53:43	Borrow caps on all Base Core Markets set to 1 wei.
27 Aug 2026, 11:09:43	MAMO and WELL supply caps set to 1 wei.
27 Aug 2026	Mamo team states “Mamo was not hacked, and no Mamo contracts were compromised.”
30 Aug 2026	Anthias Labs publishes the post mortem on the Moonwell governance forum.
31 Aug 2026	No remediation or repayment plan published yet. Borrowing on Base still frozen.

Responsible disclosure: every detail here is already public, either on the chain or in the official post mortem, and the protocol is already frozen. So nothing in this post is a live weapon.

Sources

- Moonwell governance forum, Anthias Labs post mortem: https://forum.moonwell.fi/t/post-mortem-mamo-market-incident-on-base/2208

- Moonwell contract addresses: https://docs.moonwell.fi/moonwell/protocol-information/contracts
- Moonwell contracts at commit 3b32d53: https://github.com/moonwell-fi/moonwell-contracts-v2/tree/3b32d534778b854d31bd3555c90cc787b2072566/src
- Basescan, mMAMO market: https://basescan.org/address/0x2F90Bb22eB3979f5FfAd31EA6C3F0792ca66dA32
- Basescan, attacker main wallet: https://basescan.org/address/0x719eae70d4A83f35bF82A2740699F5db84BE919D
- Etherscan, attacker second wallet holding DAI: https://etherscan.io/address/0xD71dd9b6e634412713c47fe7ae02c628e338c384
- The Block: https://www.theblock.co/news/defi/2026-08-27-moonwell-investigates-base-lending-market-issue-412913
- The Defiant: https://thedefiant.io/news/hacks/moonwell-loses-8-7-million-to-mamo-price-manipulation-on-base
- Yahoo Finance: https://finance.yahoo.com/markets/crypto/articles/moonwell-lost-8-7-million-122819291.html
- CryptoDaily on the $9.13M shortfall: https://cryptodaily.co.uk/2026/08/moonwell-mamo-exploit-base-9-13m-borrower-obligations
- TechTimes on the three incidents: https://www.techtimes.com/articles/325839/20260827/moonwell-oracle-exploit-exceeds-full-annual-revenue-third-failure-11-months.htm
- Protos on the February cbETH incident: https://protos.com/defi-meet-claude-moonwells-vibe-coded-oracle-in-1-8m-blowup/
- Moonwell forum, cbETH remediation counter proposal: https://forum.moonwell.fi/t/counter-proposal-fund-cbeth-remediation-from-the-foundation-treasury-not-a-reserve-residual/2175