medium.com

SIREN Incident Report - Siren - Medium

Dalakos

8-10 minutos

Dear SIREN community, we first want to reassure you that we have stopped the recent exploit. Thank you deeply to all those who helped with the situation.

Our community and users are our priority. We are committed to making this right for those impacted by the exploit. Below details the events of the exploit and our next steps.

What Happened?

TL;DR: On 3 September 2021 at around 12:17 AM UTC

several SIREN AMM pools were exploited via a reentrancy attack. As a result, approximately $3.5M worth of assets were drained from the AMM pools. The core settlement layer wasn't affected, all open options positions are fully collateralized and can be traded or exercised as soon as the security patch is deployed and the protocol is unpaused via the DAO multisig.

00:17: On 3 September (UTC) 2021, a series of transactions were executed by the attacker draining the UNI, KNC, WETH, WMATIC, USDC and SUSHI pools.

An example exploit transaction is linked here.

01:27: Immunify, who helps maintain our $100K security bounty, notified us that several suspicious transactions were withdrawing funds from Polygon to Mainnet. The original tip came from the 0x team.

01:53: A SIREN team member saw the message and immediately began an investigation.

02:02: A war room was created for SIREN contributors across multiple time zones to resolve the issue.

02:39: A transaction to pause ERC1155Controller was executed via the DAO Admin multisig.

03:15: A transaction to pause SeriesController was executed via the DAO Admin multisig. This transaction wasn't necessary, because pausing the ERC1155Controller is enough to prevent any token movements, and was executed out of precaution while the team investigated the core issue.

03:20: Transactions with the message “Offering bounty for return of funds. Please contact admin@sirenmarkets.com or SirenBounty on Telegram” are sent to multiple known hacker addresses on Mainnet with the stolen funds.

04:10: Community members identified that the gas for the attack was sent from an exchange address. SIREN team immediately contacts the exchange via multiple channels. Still awaiting more information.

Technical Details

This exploit was confirmed to be a classical reentrancy attack via MinterAmm.withdrawCapital, MinterAmm._sellOrWithdrawActiveTokens and ERC1155.safeTransferAcceptanceCheck in the ERC1155 option token contract implementation.

Step by Step:

1) The attacker’s Exploit Deployer address calls the exploit contract, which uses a flashloan on Aave to acquire the liquidity

2) Calls MinterAmm.bTokenBuy with an argument of 1000 bTokenAmount. This mints bToken and wToken, sends the bToken to the attacker, and keeps the wToken in the AMM. This will be important 3 steps ahead in MinterAmm._sellorWithdrawActiveTokens where the exploit needs the AMM to have a nonzero balance of wToken.

3) Calls MinterAmm.provideCapital to acquire lpToken. This deposits liquidity in the AMM in exchange for lpToken. Later on this lpToken will be burned in return for a greater amount liquidity than the attacker provided

4) Calls MinterAmm.withdrawCapital with sellTokens equal to **false.** In line 355 the contract defines **collateralTokenBalance**. This will prove fatal later on when the contract logic fails to update this value prior to interacting with external contracts inside of MinterAmm._sellorWithdrawActiveTokens.

```
336     function withdrawCapital(
337         uint256 lpTokenAmount,
338         bool sellTokens,
339         uint256 collateralMinimum
340     ) public {
341         require(!sellTokens || collateralMinimum > 0, "E12");
342         // First get starting numbers
343         uint256 redeemerCollateralBalance =
344             collateralToken.balanceOf(msg.sender);
345
346         // Get the lpToken supply
347         uint256 lpTokenSupply = IERC20Lib(address(lpToken)).totalSupply();
348
349         // Burn the lp tokens
350         lpToken.burn(msg.sender, lpTokenAmount);
351
352         // Claim all expired wTokens
353         claimAllExpiredTokens();
354
355         uint256 collateralTokenBalance =
356             collateralToken.balanceOf(address(this));
357
358         // Withdraw pro-rata collateral token
359         // We withdraw this collateral here instead of at the end,
360         // because when we sell the residual tokens to the pool we want
361         // to exclude the withdrawn collateral
362         uint256 ammCollateralBalance =
363             collateralTokenBalance -
364                 ((collateralTokenBalance * lpTokenAmount) / lpTokenSupply);
365
366         // Sell pro-rata active tokens or withdraw if no collateral left
367         ammCollateralBalance = _sellOrWithdrawActiveTokens(
368             lpTokenAmount,
369             lpTokenSupply,
370             msg.sender,
371             sellTokens,
372             ammCollateralBalance
373         );
374
```

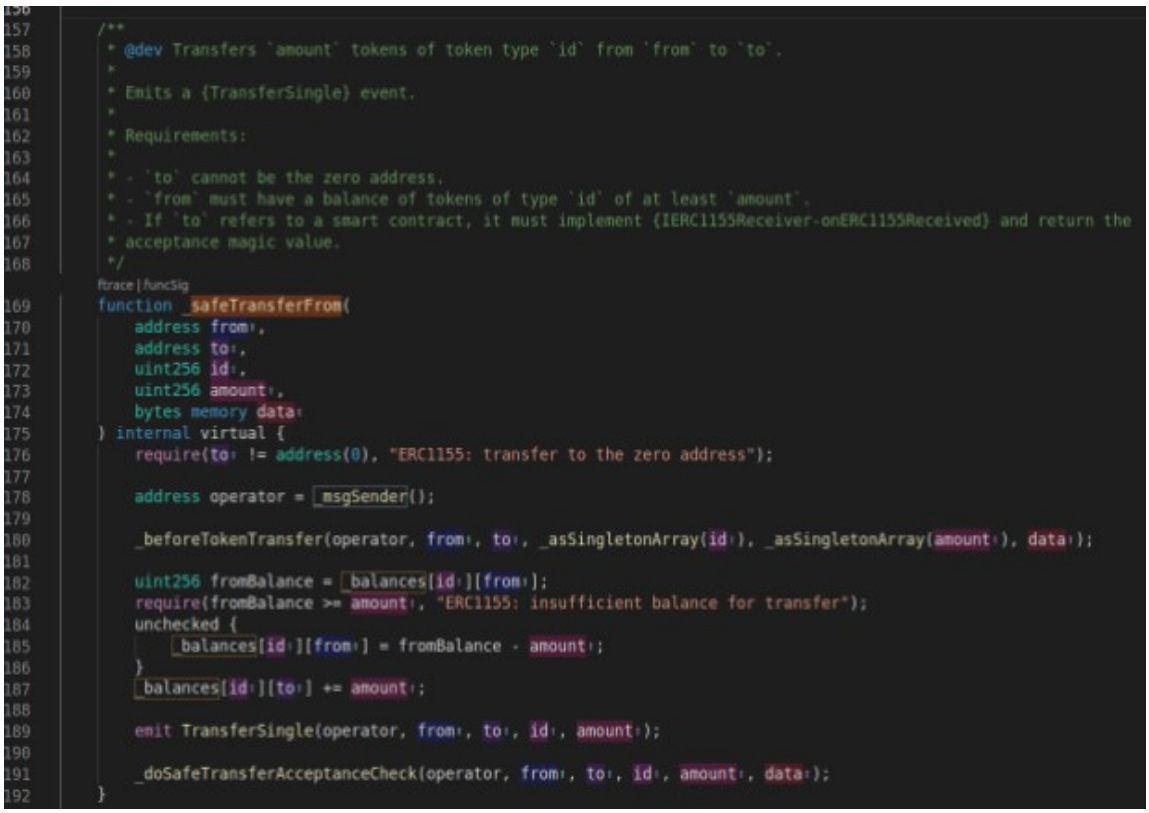
5) Inside of MinterAmm._sellorWithdrawActiveTokens execution follows the branch at line 470 because

sellTokens is equal to **false**. Then, execution follows the branch at line 483 because of the wTokens that were minted in the MinterAmm.bTokenBuy call up in step 2. Here the reentrancy attack begins with the call to erc1155Controller.safeTransferFrom.

```
448     function _sellOrWithdrawActiveTokens(
449         uint256 lpTokenAmount,
450         uint256 lpTokenSupply,
451         address redeemer,
452         bool sellTokens,
453         uint256 collateralLeft
454     ) internal returns (uint256) {
455         for (uint256 i = 0; i < openSeries.length(); i++) {
456             uint64 seriesId = uint64(openSeries.at(i));
457             if (
458                 seriesController.state(seriesId) ==
459                 ISeriesController.SeriesState.OPEN
460             ) {
461                 uint256 bTokenIndex = SeriesLibrary.bTokenIndex(seriesId);
462                 uint256 wTokenIndex = SeriesLibrary.wTokenIndex(seriesId);
463 
464                 uint256 bTokenToSell =
465                     (erc1155Controller.balanceOf(address(this), bTokenIndex) *
466                         lpTokenAmount) / lpTokenSupply;
467                 uint256 wTokenToSell =
468                     (erc1155Controller.balanceOf(address(this), wTokenIndex) *
469                         lpTokenAmount) / lpTokenSupply;
470                 if (!sellTokens || lpTokenAmount == lpTokenSupply) {
471                     // Full LP token withdrawal for the last LP in the pool
472                     // or if auto-sale is disabled
473                     if (bTokenToSell > 0) {
474                         bytes memory data;
475                         erc1155Controller.safeTransferFrom(
476                             address(this),
477                             redeemer,
478                             bTokenIndex,
479                             bTokenToSell,
480                             data
481                         );
482                     }
483                     if (wTokenToSell > 0) {
484                         bytes memory data;
485                         erc1155Controller.safeTransferFrom(
486                             address(this),
487                             redeemer,
488                             wTokenIndex,
489                             wTokenToSell,
490                             data
```

6) Siren's ERC1155Controller.sol contract inherits from OpenZeppelin's ERC1155Upgradeable.sol. ERC1155Upgradeable.safeTransferFrom calls the internal function ERC1155Upgradeable._safeTransferFrom, the final line of which is a call to the internal function. ERC1155Upgradeable._doSafeTransferAcceptanceCheck.

And here an external call to the user's address (contract) is made with the IERC1155ReceiverUpgradeable.onERC1155Received. Recall that the ERC1155 standard uses the **onERC1155Received** callback to protect against ERC1155 tokens being sent to a contract where they would be locked forever, because that contract cannot call ERC1155Upgradeable.safeTransferFrom. By forcing contract recipients to implement **onERC1155Received**, only contracts who explicitly opt in to signal receiving ERC1155 tokens will have **safeTransferFrom** with them as a recipient succeed. However, the **onERC1155Received** function is a non-view non-pure function, and so it is able to make state changes and call other contract's functions, including re-entering back into MinterAmm.withdrawCapital. Which is exactly what this exploit does.



```
/**
 * @dev Transfers `amount` tokens of token type `id` from `from` to `to`.
 *
 * Emits a {TransferSingle} event.
 *
 * Requirements:
 *
 * - `to` cannot be the zero address.
 * - `from` must have a balance of tokens of type `id` of at least `amount`.
 * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-onERC1155Received} and return the
 * acceptance magic value.
 */
ftrace|funcSig
function _safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: transfer to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, to, _asSingletonArray(id), _asSingletonArray(amount), data);

    uint256 fromBalance = _balances[id][from];
    require(fromBalance >= amount, "ERC1155: insufficient balance for transfer");
    unchecked {
        _balances[id][from] = fromBalance - amount;
    }
    _balances[id][to] += amount;

    emit TransferSingle(operator, from, to, id, amount);

    _doSafeTransferAcceptanceCheck(operator, from, to, id, amount, data);
}
```

7) Back in **MinterAmm.withdrawCapital**, the same value of **collateralTokenBalance** on line 355 gets used as in the original call to **MinterAmm.withdrawCapital**. This is the crucial exploit of the reentrancy attack, because usually subsequent calls to **MinterAmm.withdrawCapital** should use smaller and smaller values of **collateralTokenBalance**, because the collateral token gets transferred out of the AMM in line 376. But because the function call is reentrant, the AMM is tricked into removing the same amount of collateral token in each of the 2 calls to **MinterAmm.withdrawCapital**. By making multiple sets of calls to **MinterAmm.provideCapital** followed by **MinterAmm.withdrawCapital** reentered, the attacker was able to drain collateral token from each AMM.

Mitigation

We will wrap each function which interacts with an ERC1155 token in a ReentrancyGuard. Now any attempt to re-enter into **MinterAmm.withdrawCapital** or other state-changing functions will revert.

Current Status

The current total damage is assessed to be ~$3.5M at the time of the attack:

- 266,708 WMATIC
- 689,083 USDC

- 50,959 SUSHI
- 185,392 KNC
- 268 WETH
- 11,995 UNI

  The exploited funds are sitting in these four addresses. We are closely monitoring the movement of these addresses and would appreciate extra community support to do the same.

- 0x07Ba7e8947f8Fb4d33f3C7E25c2CB35B858F02Eb
- 0xfAc4088BbA1fA090FD3F1F52fd691a45C30AC053
- 0xf834eFE5B959E52E3b78cB28c4BC501b52CE41da
- 0x99da8fb52f74b7a3e38d9c75c634f6386f1649c7

Audit

An audit was performed on the SIRENv2 code, including the affected contract, and the code was specifically analyzed for reentrancy attacks. However, this reentrancy was performed in an unusual location and both the SIREN team and auditors missed the exploit. The audit report is available here.

To those impacted

The team is currently working with investors and exchanges to identify the attacker and keep them from moving the funds. The team is offering a 10% bounty for

return of funds. Please contact admin@sirenmarkets.com or @SirenBounty on Telegram if you have any information to contribute.

26 of LP addresses on Polygon were affected by the attack. The SIREN team plans to mint a redemption/I Owe You (IOU) token that will be issued to those addresses proportionate to their share of the affected funds. More details regarding the IOU redemption will be released next week.

This approach of an IOU token has been taken before when a hack is of material impact that exceeds the project's ability to immediately make whole, for example by Harvest in October of 2020.*(Read here for a detailed explanation of the Harvest situation.)*

Going Forward

We are actively pursuing the attacker and will continue to take every measure to monitor and analyze their movements. We will use this exploit as an opportunity to strengthen our protocol. Our plan is to add more security measures and tooling to prevent future exploits. We will continue to fulfill our mission to bring DeFi options to the masses.

V2 / Polygon pools and trading will remain paused through the weekend as we attempt to open communication with the hacker to negotiate. We will continue to update everyone as we work through this situation.

Thank you, as always, for sticking with us on our journey.

SIREN