Hacks Averted

Duncan Townsend, CTO Immunefi
duncan@immunefi.com

ArmorFi - Background

`redeemClaim` is called by a covered individual after a hack to claim the coverage they purchased. `PlanManager.checkCoverage` validates that the caller has purchased the correct coverage. If the check passes, the user should receive the requested amount of Ether.

Fair warning, this example is unfair.

ArmorFi - Vulnerable Method

```
function redeemClaim(address _protocol, uint256 _hackTime, uint256 _amount)
  external
{
    bytes32 hackId = keccak256(abi.encodePacked(_protocol, _hackTime));
    require(confirmedHacks[hackId], "No hack with these parameters has been confirmed.");

    (uint256 planIndex, bool covered) = IPlanManager(getModule("PLAN")).checkCoverage(msg.sender, _protocol,
_hackTime, _amount);
    require(covered, "User does not have valid amount, check path and amount");

    // Put Ether into 18 decimal format.
    uint256 payment = _amount * 10 ** 18;
    msg.sender.transfer(payment);
}
```

ArmorFi - Vulnerability

```
function redeemClaim(address _protocol, uint256 _hackTime, uint256 _amount)
  external
{
    bytes32 hackId = keccak256(abi.encodePacked(_protocol, _hackTime));
    require(confirmedHacks[hackId], "No hack with these parameters has been confirmed.");

    (uint256 planIndex, bool covered) = IPlanManager(getModule("PLAN")).checkCoverage(msg.sender, _protocol,
_hackTime, _amount);
    require(covered, "User does not have valid amount, check path and amount");

    // Put Ether into 18 decimal format.
    uint256 payment = _amount * 10 ** 18;
    msg.sender.transfer(payment);
}
```

`_amount` is *already* in Wei. The lines in red make the amount sent to the caller 10**18 times too large. This would be a total wipeout. I told you this was an unfair example.

ArmorFi - How Did This Happen?

- Refactoring from xDai
- Unit testing
- Confusing comments
- Ether/Wei not used consistently

https://medium.com/immunefi/armorfi-bug-bounty-postmortem-cf46eb650b38

Charged Particles - Background

Charged Particles is a full-featured NFT marketplace supporting interest-bearing NFTs, NFTs representing bundles of ERC20 assets, and NFTs that generate rewards to their owners each time they’re sold. The vulnerable method allows for on-chain sale of interest-bearing NFTs with a royalty paid to the creator.

Charged Particles - Vulnerable Method

```
function _buyProton(uint256 tokenId) internal {
  uint256 ownerAmount = _tokenSalePrice[tokenId];
  require(msg.value >= ownerAmount, "Proton:E-414");

  // Creator Royalties
  uint256 creatorAmount;
  address royaltiesReceiver = _creatorRoyaltiesReceiver(tokenId);
  uint256 royaltiesPct = _tokenCreatorRoyaltiesPct[tokenId];
  uint256 lastSellPrice = _tokenLastSellPrice[tokenId];
  if (royaltiesPct > 0 && lastSellPrice > 0 && salePrice > lastSellPrice) {
    creatorAmount = (salePrice - lastSellPrice).mul(royaltiesPct).div(PERCENTAGE_SCALE);
    ownerAmount = ownerAmount.sub(creatorAmount);
  }
  _tokenLastSellPrice[tokenId] = salePrice;

  // Transfer Payment
  payable(ownerOf(tokenId)).sendValue(ownerAmount);
  if (creatorAmount > 0) {
    payable(royaltiesReceiver).sendValue(creatorAmount);
  }

  // Transfer Token
  _transfer(ownerOf(tokenId), _msgSender(), tokenId);
}
```

Charged Particles - Vulnerable Method

```
function _buyProton(uint256 tokenId) internal {
  uint256 ownerAmount = _tokenSalePrice[tokenId];
  require(msg.value >= ownerAmount, "Proton:E-414");

  // Creator Royalties
  uint256 creatorAmount;
  address royaltiesReceiver = _creatorRoyaltiesReceiver(tokenId);
  uint256 royaltiesPct = _tokenCreatorRoyaltiesPct[tokenId];
  uint256 lastSellPrice = _tokenLastSellPrice[tokenId];
  if (royaltiesPct > 0 && lastSellPrice > 0 && salePrice > lastSellPrice) {
    creatorAmount = (salePrice - lastSellPrice).mul(royaltiesPct).div(PERCENTAGE_SCALE);
    ownerAmount = ownerAmount.sub(creatorAmount);
  }
  _tokenLastSellPrice[tokenId] = salePrice;

  // Transfer Payment
  payable(ownerOf(tokenId)).sendValue(ownerAmount);
  if (creatorAmount > 0) {
    payable(royaltiesReceiver).sendValue(creatorAmount);
  }

  // Transfer Token
  _transfer(ownerOf(tokenId), _msgSender(), tokenId);
}
```

Charged Particles - Vulnerable Method

```
pragma solidity 0.6.12;
contract Ransom {
   bool internal ransomEnabled = true;
   constructor() public {}
   function unlockNFT() public payable {
      if(msg.value >= 1 ether) {
         ransomEnabled = false;
      }
   }
   fallback() external payable {
      require(!ransomEnabled);
   }
}
```

Charged Particles - How did this Happen?

- Always treat user-supplied contracts as hostile
  - Where else could they call?
  - Can they DoS by reverting?
  - Can they consume unbounded amounts of gas?
- Always use a pull-pattern for payment
  - Only ever send payment to the caller

https://medium.com/immunefi/charged-particles-griefing-bug-fix-postmortem-d2791e49a66b

PancakeSwap - Background

PancakeSwap operates a lottery. The lottery proceeds in phases. First, purchasing of tickets is opened. Anyone can purchase a ticket if they've approved CAKE tokens to the lottery contract. Then the drawing happens. On-chain sources of "randomness" are combined with an off-chain random number. Then the lottery is over and the winner claims their earnings. Repeat.

PancakeSwap - Vulnerable Method

```
function multiBuy(uint256 _price, uint8[4][] memory _numbers) external {
    require (!drawed(), 'drawed, can not buy now');
    uint256 totalPrice  = 0;
    for (uint i = 0; i < _numbers.length; i++) {
        uint256 tokenId = lotteryNFT.newLotteryItem(msg.sender, _numbers[i], _price, issueIndex);
        userInfo[msg.sender].push(tokenId);
        totalPrice = totalPrice.add(_price);
    }
    cake.safeTransferFrom(address(msg.sender), address(this), totalPrice);
}
```

PancakeSwap - Vulnerability

```
function multiBuy(uint256 _price, uint8[4][] memory _numbers) external {
    require (!drawed(), 'drawed, can not buy now');
    require(!drawingPhase, 'drawing, can not buy now');
    uint256 totalPrice  = 0;
    for (uint i = 0; i < _numbers.length; i++) {
        uint256 tokenId = lotteryNFT.newLotteryItem(msg.sender, _numbers[i], _price, issueIndex);
        userInfo[msg.sender].push(tokenId);
        totalPrice = totalPrice.add(_price);
    }
    cake.safeTransferFrom(address(msg.sender), address(this), totalPrice);
}
```

Did you notice the un-fixed error in the description of the lottery?

PancakeSwap - How did this Happen?

- Copy/paste error
- `multiBuy` wasn’t exposed through the Web3 UI
- Unit testing didn’t cover the “sad” path

https://medium.com/immunefi/pancakeswap-lottery-vulnerability-postmortem-and-bug-4febdb1d2400

Zapper - Background

Zapper is a platform for DeFi portfolio management. Some of their contracts allow easy management of UniswapV2 LP positions. The vulnerable contract allows users to easily remove liquidity from their position. UniswapV2 implements a variant EIP-2612 that allows users to set a token approval without submitting a transaction on chain, saving gas.

Zapper - Vulnerable Method

```
function ZapOutWithPermit(
    address toTokenAddress,
    address fromPoolAddress,
    uint256 incomingLP,
    uint256 minTokensRec,
    bytes memory permitData,
    address[] memory swapTargets,
    bytes[] memory swapData,
    address affiliate
) public stopInEmergency returns (uint256) {
    // permit
    _validatePool(fromPoolAddress);
    (bool success, ) = fromPoolAddress.call(permitData);
    require(success, "Could Not Permit");

    return (
        ZapOut(
            toTokenAddress,
            fromPoolAddress,
            incomingLP,
            minTokensRec,
            swapTargets,
            swapData,
            affiliate,
            false
        )
    );
}
```

Zapper - Vulnerability

```
function ZapOutWithPermit(
    address toTokenAddress,
    address fromPoolAddress,
    uint256 incomingLP,
    uint256 minTokensRec,
    bytes memory permitData,
    address[] memory swapTargets,
    bytes[] memory swapData,
    address affiliate
) public stopInEmergency returns (uint256) {
    // permit
    _validatePool(fromPoolAddress);
    (bool success, ) = fromPoolAddress.call(permitData);
    require(success, "Could Not Permit");

    return (
        ZapOut(
            toTokenAddress,
            fromPoolAddress,
            incomingLP,
            minTokensRec,
            swapTargets,
            swapData,
            affiliate,
            false
        )
    );
}
```

Zapper - How did this happen?

- Use of `.call` is extremely dangerous, especially with user-supplied calldata.
- Avoid the use of `.call` whenever possible.
- If use of `.call` is unavoidable, carefully control the calldata
  - Whitelist selectors
  - Whitelist contracts
  - Construct the calldata on-chain from structured, user-supplied input
- Use a linter or static analyzer

https://immunefi.medium.com/zapper-arbitrary-call-data-bug-fix-postmortem-d75a4a076ae9

https://medium.com/zapper-protocol/post-mortem-sushiswap-uniswap-v2-zap-out-exploit-84e5d34603f0

Fei Protocol - Background

FEI is an algorithmic stablecoin that uses the assets (ETH) controlled by the protocol to maintain the pricing peg of 1 FEI = 1 USD. There are a variety of methods, but in this case we're using only the method that allows users to mint FEI at $1.01, putting a hard cap on the maximum price of FEI. The protocol-controlled value is used to provide liquidity in the UniswapV2 pool.

Fei Protocol - Vulnerable Methods

```
function purchase(address to, uint256 amountIn)
    external
    payable
    override
    postGenesis
    whenNotPaused
{
    require(
        msg.value == amountIn,
        "Bonding Curve: Sent value does not equal input"
    );
    updateOracle();

    amountOut = getAmountOut(amountIn);
    _incrementTotalPurchased(amountOut);
    fei().mint(to, amountOut);
}
```

(this can be invoked from outside, `ethAmount` is the amount of ETH held in the contract)

```
function _addLiquidity(uint256 ethAmount, uint256 feiAmount) internal {
    _mintFei(feiAmount);
    uint256 endOfTime = uint256(-1);
    router.addLiquidityETH{value: ethAmount}(
        address(fei()),
        feiAmount,
        0, // slippage
        0,
        address(this),
        endOfTime
    );
}
```

Fei Protocol - Vulnerability

```
function purchase(address to, uint256 amountIn)
    external
    payable
    override
    postGenesis
    whenNotPaused
{
    require(
        msg.value == amountIn,
        "Bonding Curve: Sent value does not equal input"
    );
    updateOracle();

    amountOut = getAmountOut(amountIn);
    _incrementTotalPurchased(amountOut);
    fei().mint(to, amountOut);
}
```

(this can be invoked from outside, `ethAmount` is the amount of ETH held in the contract)

```
function _addLiquidity(uint256 ethAmount, uint256 feiAmount) internal {
    _mintFei(feiAmount);
    uint256 endOfTime = uint256(-1);
    router.addLiquidityETH{value: ethAmount}(
        address(fei()),
        feiAmount,
        0, // slippage
        0,
        address(this),
        endOfTime
    );
}
```

Fei Protocol - How Did This Happen?

- Interaction with a DEX is always dangerous - avoid it if possible
- Always use a stable, manipulation resistant pricing oracle
- Never trust the spot price of DEX

https://medium.com/immunefi/fei-protocol-flashloan-vulnerability-postmortem-7c5dc001affb

Hacks Averted

Duncan Townsend, CTO Immunefi
duncan@immunefi.com

Backup slides

Primitive Finance - Background

Primitive Finance allows users to buy and sell options on underlying ERC20 tokens. It contains an adapter to permit the use of flash loans while swapping and trading tokens.

Primitive Finance - Vulnerable Method

```
function uniswapV2Call(
    address sender,
    uint256 amount0,
    uint256 amount1,
    bytes calldata data
) external override {
    address token0 = IUniswapV2Pair(msg.sender).token0();
    address token1 = IUniswapV2Pair(msg.sender).token1();
    assert(msg.sender == factory.getPair(token0, token1));
    (bool success, bytes memory returnData) = address(this).call(data);
    require(
        success &&
            (returnData.length == 0 || abi.decode(returnData, (bool))),
        "ERR_UNISWAPV2_CALL_FAIL"
    );
}
```

Primitive Finance - Vulnerability

```
function uniswapV2Call(
    address sender,
    uint256 amount0,
    uint256 amount1,
    bytes calldata data
) external override {
    address token0 = IUniswapV2Pair(msg.sender).token0();
    address token1 = IUniswapV2Pair(msg.sender).token1();
    require(sender == address(this));
    assert(msg.sender == factory.getPair(token0, token1));
    (bool success, bytes memory returnData) = address(this).call(data);
    require(
        success &&
            (returnData.length == 0 || abi.decode(returnData, (bool))),
        "ERR_UNISWAPV2_CALL_FAIL"
    );
}
```

Primitive Finance - How Did This Happen?

- Insufficient validation
- Whitelist *everything*

https://primitivefinance.medium.com/postmortem-on-the-primitive-finance-whitehack-of-february-21st-2021-17446c0f3122

Fei Protocol - Vulnerable Method

```
/// @notice get the burn amount of a sell transfer
/// @param amount the FEI size of the transfer
/// @return penalty the FEI size of the burn incentive
/// @return _initialDeviation the Decimal deviation from peg before a transfer
/// @return _finalDeviation the Decimal deviation from peg after a transfer
/// @dev calculated based on a hypothetical sell, applies to any ERC20 FEI transfer to the pool
function getSellPenalty(uint256 amount) public view override {
    int256 signedAmount = amount.toInt256();

    (
        Decimal.D256 memory initialDeviation,
        Decimal.D256 memory finalDeviation,
        Decimal.D256 memory peg,
        uint256 reserveFei,
        uint256 reserveOther
    ) = _getPriceDeviations(signedAmount);

    // if trade ends above peg, it was always above peg and no penalty needed
    if (finalDeviation.equals(Decimal.zero())) {
        return (0, initialDeviation, finalDeviation);
    }

    uint256 incentivizedAmount = amount;
    // if trade started above but ended below, only penalize amount going below peg
    if (initialDeviation.equals(Decimal.zero())) {
        uint256 amountToPeg = _getAmountToPegFei(reserveFei, reserveOther, peg);
        incentivizedAmount = amount.sub(amountToPeg, "UniswapIncentive: Underflow");
    }

    Decimal.D256 memory multiplier =
        _calculateIntegratedSellPenaltyMultiplier(initialDeviation, finalDeviation);
    penalty = multiplier.mul(incentivizedAmount).asUint256();
    return (penalty, initialDeviation, finalDeviation);
}
```

Fei Protocol - Vulnerable Method

```
function swap(uint amount0Out, uint amount1Out, address to) external lock {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves();
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');

    uint balance0;
    uint balance1;
    {
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
    }

    uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
    uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
    require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
    {
        uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
        uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
        require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
    }

    _update(balance0, balance1, _reserve0, _reserve1);
}

function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private {
    uint32 blockTimestamp = uint32(block.timestamp % 2**32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast;
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
    }
    reserve0 = uint112(balance0);
    reserve1 = uint112(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}
```

Fei Protocol - Vulnerability

```
/// @notice get the burn amount of a sell transfer
/// @param amount the FEI size of the transfer
/// @return penalty the FEI size of the burn incentive
/// @return _initialDeviation the Decimal deviation from peg before a transfer
/// @return _finalDeviation the Decimal deviation from peg after a transfer
/// @dev calculated based on a hypothetical sell, applies to any ERC20 FEI transfer to the pool
function getSellPenalty(uint256 amount) public view override {
    int256 signedAmount = amount.toInt256();

    (
        Decimal.D256 memory initialDeviation,
        Decimal.D256 memory finalDeviation,
        Decimal.D256 memory peg,
        uint256 reserveFei,
        uint256 reserveOther
    ) = _getPriceDeviations(signedAmount);

    // if trade ends above peg, it was always above peg and no penalty needed
    if (finalDeviation.equals(Decimal.zero())) {
        return (0, initialDeviation, finalDeviation);
    }

    uint256 incentivizedAmount = amount;
    // if trade started above but ended below, only penalize amount going below peg
    if (initialDeviation.equals(Decimal.zero())) {
        uint256 amountToPeg = _getAmountToPegFei(reserveFei, reserveOther, peg);
        incentivizedAmount = amount.sub(amountToPeg, "UniswapIncentive: Underflow");
    }

    Decimal.D256 memory multiplier =
        _calculateIntegratedSellPenaltyMultiplier(initialDeviation, finalDeviation);
    penalty = multiplier.mul(incentivizedAmount).asUint256();
    return (penalty, initialDeviation, finalDeviation);
}
```

Fei Protocol - How Did This Happen?

- Explicitly define your system invariants
- Check those invariants at every step of the system
- Use a fuzzer

https://medium.com/immunefi/fei-protocol-vulnerability-postmortem-483f9a7e6ad1

Vesper/BT Finance - Background

A common pattern in yield farming is to periodically swap some underlying asset (e.g. Ether, stablecoin, etc.) for the yield token on a DEX, then burn the yield token or hold it in the contract. `harvest` purchases TokenB (e.g. the yield token) with some TokenA (e.g. Ether, stablecoin).

Vesper/BT Finance - Vulnerable Method

```
function harvest() public {
  withdrawTokenA();
  uint256 reward = TokenA.balanceOf(address(this));
  unirouter.swapExactTokensForTokens(reward, 0, pathTokenAB, this, now.add(1800));
  depositTokenB();
}
```

This isn’t verbatim code from either project.

Vesper/BT Finance - Vulnerability

```
function harvest() public onlyOwner {
  require(msg.sender == tx.origin);
  withdrawTokenA();
  uint256 reward = TokenA.balanceOf(address(this));
  unirouter.swapExactTokensForTokens(reward, 0, pathTokenAB, this, now.add(1800));
  depositTokenB();
}
```

1. Flash loan underlying token
2. Uniswap underlying token -> yield token
3. Call `harvest`
4. Uniswap yield token -> underlying token
5. Return flash loan

Vesper/BT Finance - How did this happen?

- “What’s the harm? Whoever calls it pays gas, only to have the contract collect its rightful yield.”
- The use case of calling from another contract wasn’t considered.
- Any interaction with a DEX must be guarded this way because flash loans allow attackers to arbitrarily manipulate market conditions

https://medium.com/dedaub/yield-skimming-forcing-bad-swaps-on-yield-farming-397361fd7c72