```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

/**
 * @title AdvancedYieldFarm
 * @dev A complex DeFi yield farming contract with multiple
vulnerabilities
 * This contract is intentionally vulnerable for security testing purposes
 */
contract AdvancedYieldFarm is ReentrancyGuard, Ownable {
    using SafeMath for uint256;

    struct UserInfo {
        uint256 amount;
        uint256 rewardDebt;
        uint256 pendingRewards;
        uint256 lastClaimTime;
        bool isVIP;
    }

    struct PoolInfo {
        IERC20 lpToken;
        uint256 allocPoint;
        uint256 lastRewardBlock;
        uint256 accRewardPerShare;
        uint256 depositFee;
        bool isActive;
    }

    IERC20 public rewardToken;
    uint256 public rewardPerBlock;
    uint256 public startBlock;
    uint256 public bonusEndBlock;
    uint256 public constant BONUS_MULTIPLIER = 10;

    PoolInfo[] public poolInfo;
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    mapping(address => bool) public authorizedCallers;
    mapping(address => uint256) public userNonces;

    uint256 public totalAllocPoint = 0;
    uint256 private constant PRECISION = 1e12;
    // Flash loan related
    mapping(address => uint256) public flashLoanAmounts;
    uint256 public flashLoanFee = 9; // 0.09%
    bool public flashLoanEnabled = true;

    // Price oracle (simplified)
    mapping(address => uint256) public tokenPrices;
    address public priceOracle;

    // Emergency functions
    bool public emergencyWithdrawEnabled = false;
    uint256 public emergencyWithdrawFee = 500; // 5%

    event Deposit(address indexed user, uint256 indexed pid, uint256
amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256
amount);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid,
uint256 amount);
    event FlashLoan(address indexed borrower, uint256 amount);

    constructor(
        IERC20 _rewardToken,
        uint256 _rewardPerBlock,
        uint256 _startBlock,
        uint256 _bonusEndBlock
    ) {
        rewardToken = _rewardToken;
        rewardPerBlock = _rewardPerBlock;
        startBlock = _startBlock;
        bonusEndBlock = _bonusEndBlock;
        priceOracle = msg.sender; // VULNERABILITY: Centralized oracle
    }

    // VULNERABILITY 1: Reentrancy in deposit function despite
ReentrancyGuard inheritance
    function deposit(uint256 _pid, uint256 _amount) public {
        // Missing nonReentrant modifier!
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);

        if (user.amount > 0) {
            uint256 pending =
user.amount.mul(pool.accRewardPerShare).div(PRECISION).sub(user.rewardDebt
);
            if (pending > 0) {
                // VULNERABILITY: External call before state update
                safeRewardTransfer(msg.sender, pending);
            }
        }

        if (_amount > 0) {
            // VULNERABILITY 2: No slippage protection
            pool.lpToken.transferFrom(address(msg.sender), address(this),
_amount);

            // VULNERABILITY 3: Fee calculation overflow potential
            uint256 depositFee = _amount.mul(pool.depositFee).div(10000);
            user.amount = user.amount.add(_amount.sub(depositFee));
        }

        user.rewardDebt =
user.amount.mul(pool.accRewardPerShare).div(PRECISION);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // VULNERABILITY 4: Timestamp dependence and front-running opportunity
    function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        // VULNERABILITY: Using block.timestamp for critical logic
        require(block.timestamp > user.lastClaimTime + 1 hours,
"Withdrawal too early");
        require(user.amount >= _amount, "Insufficient balance");

        updatePool(_pid);

        uint256 pending =
user.amount.mul(pool.accRewardPerShare).div(PRECISION).sub(user.rewardDebt
);
        if (pending > 0) {
            safeRewardTransfer(msg.sender, pending);
        }

        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            // VULNERABILITY 5: No withdrawal fee validation
            pool.lpToken.transfer(msg.sender, _amount);
        }

        user.rewardDebt =
user.amount.mul(pool.accRewardPerShare).div(PRECISION);
        user.lastClaimTime = block.timestamp;

        emit Withdraw(msg.sender, _pid, _amount);
    }

    // VULNERABILITY 6: Access control bypass through signature replay
    function authorizedWithdraw(
        uint256 _pid,
        uint256 _amount,
        uint256 _nonce,
        bytes memory _signature
    ) external {
        // VULNERABILITY: Weak signature verification
        bytes32 hash = keccak256(abi.encodePacked(msg.sender, _pid,
_amount, _nonce));
        address signer = recoverSigner(hash, _signature);

        require(authorizedCallers[signer], "Unauthorized signer");
        // VULNERABILITY: No nonce validation against replay attacks

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        require(user.amount >= _amount, "Insufficient balance");

        user.amount = user.amount.sub(_amount);
        pool.lpToken.transfer(msg.sender, _amount);
    }

    // VULNERABILITY 7: Flash loan with inadequate checks
    function flashLoan(uint256 _amount) external {
        require(flashLoanEnabled, "Flash loans disabled");
        require(_amount > 0, "Invalid amount");

        uint256 balanceBefore = rewardToken.balanceOf(address(this));
        require(balanceBefore >= _amount, "Insufficient liquidity");

        flashLoanAmounts[msg.sender] = _amount;

        // VULNERABILITY: No checks-effects-interactions pattern
        rewardToken.transfer(msg.sender, _amount);
        // VULNERABILITY 8: Trusting external call without proper
validation
        IFlashLoanReceiver(msg.sender).executeOperation(_amount);

        uint256 balanceAfter = rewardToken.balanceOf(address(this));
        uint256 feeAmount = _amount.mul(flashLoanFee).div(10000);

        // VULNERABILITY: Integer overflow potential in fee calculation
        require(balanceAfter >= balanceBefore.add(feeAmount), "Flash loan
not repaid");

        delete flashLoanAmounts[msg.sender];
        emit FlashLoan(msg.sender, _amount);
    }

    // VULNERABILITY 9: Price manipulation susceptibility
    function liquidateUser(address _user, uint256 _pid) external {
        UserInfo storage user = userInfo[_pid][_user];
        PoolInfo storage pool = poolInfo[_pid];

        // VULNERABILITY: Using easily manipulated price oracle
        uint256 tokenPrice = tokenPrices[address(pool.lpToken)];
        uint256 userValue = user.amount.mul(tokenPrice);

        // VULNERABILITY 10: Magic numbers and arbitrary liquidation
threshold
        if (userValue < 1000e18) { // Hardcoded threshold
            // Force liquidation
            uint256 liquidationBonus = user.amount.mul(10).div(100); //
10% bonus

            user.amount = 0;
            user.rewardDebt = 0;

            // VULNERABILITY: No slippage protection on liquidation
            pool.lpToken.transfer(msg.sender,
user.amount.add(liquidationBonus));
        }
    }

    // VULNERABILITY 11: Unchecked external call in emergency function
    function emergencyWithdraw(uint256 _pid) public {
        require(emergencyWithdrawEnabled, "Emergency withdraw disabled");

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;

        // VULNERABILITY: Fee calculation without overflow protection
        uint256 fee = amount * emergencyWithdrawFee / 10000;
        uint256 amountAfterFee = amount - fee;

        // VULNERABILITY: Unchecked external call
        pool.lpToken.transfer(msg.sender, amountAfterFee);

        emit EmergencyWithdraw(msg.sender, _pid, amountAfterFee);
    }

    // VULNERABILITY 12: Privilege escalation through admin functions
    function updateRewardPerBlock(uint256 _rewardPerBlock) public
onlyOwner {
        // VULNERABILITY: No limits on reward rate changes
        rewardPerBlock = _rewardPerBlock;
    }

    function setTokenPrice(address _token, uint256 _price) external {
        // VULNERABILITY: Missing access control
        tokenPrices[_token] = _price;
    }

    function addAuthorizedCaller(address _caller) external onlyOwner {
        authorizedCallers[_caller] = true;
    }

    // VULNERABILITY 13: Logic error in pool update
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];

        if (block.number <= pool.lastRewardBlock) {
            return;
        }

        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }

        uint256 multiplier = getMultiplier(pool.lastRewardBlock,
block.number);
```

```
        uint256 reward =
multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);

        // VULNERABILITY: Unbounded accumulation without overflow check
        pool.accRewardPerShare =
pool.accRewardPerShare.add(reward.mul(PRECISION).div(lpSupply));
        pool.lastRewardBlock = block.number;
    }

    // VULNERABILITY 14: Incorrect multiplier calculation
    function getMultiplier(uint256 _from, uint256 _to) public view returns
(uint256) {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from).mul(BONUS_MULTIPLIER);
        } else if (_from >= bonusEndBlock) {
            return _to.sub(_from);
        } else {
            // VULNERABILITY: Potential underflow in edge case
            return
bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(_to.sub(bonusEndBlock))
;
        }
    }

    // VULNERABILITY 15: Unsafe transfer without return value check
    function safeRewardTransfer(address _to, uint256 _amount) internal {
        uint256 rewardBal = rewardToken.balanceOf(address(this));
        if (_amount > rewardBal) {
            // VULNERABILITY: Silent failure instead of revert
            rewardToken.transfer(_to, rewardBal);
        } else {
            rewardToken.transfer(_to, _amount);
        }
    }

    // Helper function for signature recovery (simplified and vulnerable)
    function recoverSigner(bytes32 _hash, bytes memory _signature)
internal pure returns (address) {
        // VULNERABILITY 16: Simplified signature recovery without proper
validation
        require(_signature.length == 65, "Invalid signature length");

        bytes32 r;
        bytes32 s;
        uint8 v;

        assembly {
            r := mload(add(_signature, 32))
            s := mload(add(_signature, 64))
            v := byte(0, mload(add(_signature, 96)))
        }

        return ecrecover(_hash, v, r, s);
    }

    // VULNERABILITY 17: Arbitrary code execution risk
    function executeTransaction(address target, bytes calldata data)
external onlyOwner {
        // VULNERABILITY: Owner can call any contract with any data
        (bool success,) = target.call(data);
        require(success, "Transaction failed");
    }

    // Additional vulnerable functions
    function addPool(
        uint256 _allocPoint,
        IERC20 _lpToken,
        uint256 _depositFee,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }

        uint256 lastRewardBlock = block.number > startBlock ? block.number
: startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);

        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accRewardPerShare: 0,
            depositFee: _depositFee,
            isActive: true
        }));
    }

    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }
    // VULNERABILITY 18: Denial of service through gas limit
    function updateAllUserRewards() external {
        // VULNERABILITY: Unbounded loop that can hit gas limit
        for (uint256 pid = 0; pid < poolInfo.length; pid++) {
            for (uint256 i = 0; i < 1000; i++) { // Arbitrary large number
                // Simulated user processing that could run out of gas
                updatePool(pid);
            }
        }
    }
}

interface IFlashLoanReceiver {
    function executeOperation(uint256 amount) external;
}

// VULNERABILITY 19: Malicious receiver contract example
contract MaliciousReceiver is IFlashLoanReceiver {
    AdvancedYieldFarm public farm;

    constructor(address _farm) {
        farm = AdvancedYieldFarm(_farm);
    }

    function executeOperation(uint256 amount) external override {
        // VULNERABILITY: Could manipulate state during flash loan
        // Could call deposit/withdraw to manipulate pool state
        // Could perform reentrancy attacks
        // Could manipulate price oracles

        // Repay the flash loan
        IERC20 token = farm.rewardToken();
        token.transfer(msg.sender, amount + (amount * 9 / 10000));
```