```
1  contract LendingContract {
2    IERC20 public WETH;
3    IERC20 public USDC;
4    IUniswapV2Pair public pair; // USDC - WETH
5    // debt --> USDC, collateral --> WETH
6    mapping(address => uint) public debt;
7    mapping(address => uint) public collateral;
8
9    function liquidate(address user) external {
10     uint dAmount = debt[user];
11     uint cAmount = collateral[user];
12     require(getPrice() * cAmount * 80 / 100 < dAmount,
13       "the given user's fund cannot be liquidated");
14     address _this = address(this);
15     USDC.transferFrom(msg.sender, _this, dAmount);
16     WETH.transferFrom(_this, msg.sender, cAmount);
17   }
18   function getPrice() view returns (uint) {
19     return (USDC.balanceOf(address(pair)) /
20         WETH.balanceOf(address(pair)))
21   }
22 }
```

Fig. 5: Price oracle manipulation exploit in Deus Finance

invariant law. In Uniswap [92], a leading AMM contract, the invariant is denoted by a constant product formula, expressed as $x \times y = k$, stating that trades must not change the product k of a pair's reserve balances (within the contract) [93], e.g., x for WETH and y for USDC. The price of one asset over the other is hence decided by their ratio, e.g., y/x denoting the price of WETH over USDC. Intuitively, more supply of x leads to its depreciation and y's appreciation. A code snippet from Uniswap, its explanation, and an example are presented in our supplementary material [29] (§IV-A).

Price Oracle Manipulation. Despite being pivotal for DeFi project development, price oracles are occasionally used improperly by application contracts, rendering their price queries vulnerable. It is not a bug in the price oracle contract, but an issue caused by oracle misuse in the application contract. For example, although Uniswap provides an official (and well protected) API for price queries, application contract developers tend to implement their own queries (to Uniswap) to avoid the expensive gas costed by the official API. A common faulty code pattern in the application contract is to simply determine the price by querying the ratio of two assets' instant balances in the oracle contract. Recall that block-chain transactions are atomic so that any action sequence in a single transaction cannot be interrupted or interleaved with other actions. Hence, a malicious user can tamper with the price without the interruptions of arbitragers. It is done by first processing an exchange (with the oracle), then invoking a function in the vulnerable application contract which makes an erroneous query (to the oracle), and finally processing another exchange (with the oracle) which is the counter version of the first one. Essentially, the first exchange imbalances the Uniswap contract in order to manipulate the follow-up price query, while the second exchange re-balances the Uniswap contract to avoid losing the (borrowed) funds used in step one. Note that the three actions are wrapped in a single transaction (a piece of code written by the adversary), guaranteeing that no arbitrage behavior can interfere the attack.

Example. Figure 5 presents a vulnerable code snippet, which is slightly modified from a real-world exploit against Deus Finance causing a loss of $3.1 millions. The bug survived at least one publicly-known audit round [94]. Deus is a lending contract that allows users to deposit WETH as collateral and borrow USDC. Lines 2-4 define the addresses of WETH, USDC, and the Uniswap AMM, respectively. Line 6 defines a mapping `debt`, which denotes the amount of borrowed USDC for each user, and line 7 a mapping `collateral` for the amount of each user's deposited WETH. As a lending contract, Deus supports multiple basic functionalities, including depositing collateral, withdrawing collateral, getting loans, and paying debts. The vulnerability lies in function `liquidate` (line 9) which forces to close a given user's *ill position*, i.e., the user's debts exceed 80% of her collateral. To do so, the function's caller, i.e., `msg.sender`, pays the user's debt and gets her collateral. Specifically, the function first checks whether the position of `user` is ill (lines 10-13) and processes the token transfers (lines 14-16). The price oracle is involved when calculating the real-world value of the collateral, i.e., WETH, through function `getPrice()` (defined in lines 18-21). The function does not use the Uniswap API. Instead, it directly queries the instance balances of USDC and WETH in Uniswap and uses their ratio as the price.

To exploit, the adversary drastically decreases the price of a collateral, forcefully making a victim's position liquidable. She then liquidates a valuable collateral with a much smaller amount of fund. Assume Bob (victim) deposits 100 WETH as collateral and borrows $100,000$ USDC. Also assume that the current price of WETH is $\$4,000$ and the Uniswap AMM holds 100 WETH and $400,000$ USDC. Note that Bob's current position is healthy and cannot be liquidated, since the value of his debt is $\$100,000$ and his collateral worths $\$400,000$. Alice, the adversary, can exploit the aforementioned vulnerability by encapsulating the following three actions into a single transaction. Specifically, she first exchanges 100 WETH for $200,000$ USDC through UniSwap, making the AMM's balances of WETH and USDC 200 and $200,000$, respectively. Note that although the current real-world price of WETH is $\$4,000$, Alice pays 100 WETH for $200,000$ USDC, according to the constant-product invariant, i.e., $100 \times 400,000 = (100 + 100) \times (400,000 - 200,000)$. Alice then invokes `liquidate(Bob)`, which succeeds since Bob's position depreciates with a WETH price of $1000 (due to the instant balances of WETH and USDC in the AMM), i.e., $100 \times 1000 \times 0.8 < 100,000$ at line 12. By paying $100,000$ USDC, Alice gets 100 WETH whose real-world value is $\$400,000$. She acquires a large profit of $\$300,000$. After that, Alice re-balances the AMM by exchanging $200,000$ USDC for 100 WETH, retrieving her initial attack funds. To prevent price oracle manipulation, most on-chain DEXes provide manipulation-resistant APIs for price queries. Time-weighted average price (TWAP) is the most common solution nowadays. It is a pricing algorithm that calculates the average price of an asset over a set period. It provides great resistance against flash loans. Recall that a flash loan has to happen within a single

```
1   contract Vote {
2     struct Proposal {
3       uint160 sTime; address newOwner;
4     }
5     IERC20 votingToken;
6     address owner;
7     Proposal proposal;
8
9     function propose() external {
10      require(proposal.sTime == 0, "on-going proposal");
11      proposal = Proposal(block.timestamp, msg.sender);
12    }
13    function vote(uint amount) external {
14      require(proposal.sTime + 2 days > block.timestamp,
15        "voting has ended");
16      votingToken.transferFrom(
17        msg.sender, address(this), amount);
18    }
19    function end() external {
20      require(proposal.sTime != 0, "no proposal");
21      require(proposal.sTime + 2 days < block.timestamp,
22        "voting has not ended");
23      require(votingToken.balanceOf(address(this))*2 >
24          votingToken.totalSupply(), "vote failed");
25      owner = proposal.newOwner;
26      delete proposal;
27    }
28    function getLockedFunds() external onlyOwner { ... }
29  }
```

Fig. 6: A voting contract vulnerability

transaction and hence the time weight of its manipulated price is 0. There are also other solutions, e.g., volume-weighted average price (VWAP) and time-weighted average tick (TWAT).

Flash Loans. Recall that the aforementioned exploit requires a tremendous amount of initial funds, i.e., 100 WETH with $\$400,000$ real-world value, which seems to hinder the impact of price oracle manipulation. However, *flash loan*, a unique and innovative lending model enabled by blockchain techniques, makes such attacks easily realizable. It allows users to borrow (a tremendous amount of) debts without depositing any collateral. It leverages the atomicity of blockchain transactions, that is, the borrow happens at the beginning of a transaction and the debt is paid off at the end. An example can be found in the supplementary material [29] (§IV-B).

Abstract Bug Model and Remedy. Given a price oracle C_{orc}, an application contract C, and lending contract(s) C_l supporting flash loans, C needs to query C_{orc} for prices which are based on instant balances (or balances within a short time) in C_{orc}, and C_l needs to have sufficient funds to manipulate the balance ratio in C_{orc}. The cost of the attack is minimum, including just gas and fees, as the flash loan is paid off at the end. The profit depends on how much price changes can be induced. To remedy such bugs, developers simply use official APIs strictly following the specification.

B. Privilege Escalation (C5)

These bugs arise when an (unexpected) sequence of functions can be invoked to bypass access control.

Example. Figure 6 presents a real-world case from an anonymized contract (upon developers' request). The code is completely rewritten to ensure anonymity while its essence is retained. This is a voting contract where users can elect a new contract owner by voting. In lines 2-4, the contract defines a data structure `Proposal` to describe a proposal with `sTime` denoting the start time of voting and `newOwner` the proposed new owner. There are three state variables `votingToken`, `owner`, and `proposal` denoting the token used for voting (line 5), the current contract owner (line 6), and an on-going proposal (line 7), respectively. Function `propose` (line 9) allows a user to propose himself as the new owner, which creates a new proposal (at line 11) and sets the current block time as the start time and `msg.sender` the proposed owner. Observe that there can only be one on-going proposal (line 10). Users vote by function `vote`, in which they send their voting tokens to the contract (lines 16-17) to support a proposal. Note that users can only vote in the first two days after the voting starts, guarded by the `require` in lines 14-15. The voting ends two days later, and the decision is made by function `end`. Function `end` first checks whether there is an on-going proposal (line 20) and whether the voting has lasted for at least 2 days (lines 21-22). In lines 23-24, the function then checks whether over 50% `votingToken` holders have voted for the proposal. If so, a critical operation of setting a new contract owner is performed (line 25). At line 28, a privileged function `getLockedFunds` allows the owner to get all the locked funds. Note that both functions `vote` and `end` strictly constrain the invocation time, which constitutes an access control preventing the two functions from being invoked in a single transaction. Otherwise, an adversary could invoke function `vote` with a tremendous amount of flash-loaned `votingToken` and force a malicious proposal to go through (similar to the exploit in §VII-A). However, an unexpected call sequence can evade the access control. Specifically, consider an adversary Alice proposes herself as the owner. When the time is approaching the deadline `proposal.sTime + 2 days`, she launches an attack wrapping the following actions into a single transaction, including 1) flash-loaning a large amount of `votingToken` from its AMM contract, 2) invoking `votingToken.transferFrom`, a fund transfer function provided by all ERC20 tokens to directly transfer the loaned amount to the contract without any access control, 3) invoking `end` to become the owner, 4) getting locked funds by function `getLockedFunds`, and 5) paying off the flash-loan debt. Intuitively, Alice "votes" without calling the `vote` function. The developers did not anticipate such a business flow and hence did not guard properly.

Abstract Bug Model and Remedy. Let a business flow $\mathcal{B}$ be a sequence of transactions t_1, ..., t_n, each denoting an external function invocation, and n the length of flow which may be equal to or larger than 1. Assume $\mathcal{B}$ has some critical operation f guarded by a set of access control checks, denoted as $\mathcal{P}$, a conjunction of multiple checks. However, there exists an (unexpected) business flow t'_1, ... t'_m that can reach f with access control $\mathcal{P}'$ and $\mathcal{P}' < \mathcal{P}$ (here the operator $<$ means weaker-than). The challenges of identifying this type of bugs lie in recognizing sensitive operations, which may require domain knowledge, and finding the multiple paths that can lead to the operations. The fixes are to add the missing access

```
1  contract Lottery {
2    //  user address -> lottery id -> count
3    mapping (address => mapping(uint64 => uint))
4      public tickets;
5    uint64 winningId; // the winning id
6    bool drawingPhase; // whether the owner is drawing
7
8    // invoked every day to reset a round
9    function reset() external onlyOwner {
10       delete tickets;
11       winningId = 0; drawingPhase = false;
12   }
13   function buy(uint64 id, uint amount) external {
14     require(winningId == 0, "already drawn");
15     require(!drawingPhase, "drawing")
16     receivePayment(msg.sender, amount),
17     tickets[msg.sender][id] += amount;
18   }
19   function enterDrawingPhase() external onlyOwner {
20     drawingPhase = true;
21   }
22   // id is randomly chosen off-chain, i.e., by chainlink
23   function draw(uint64 id) external onlyOwner {
24     require(winningId == 0, "already drawn");
25     require(drawingPhase, "not drawing");
26     require(id != 0, "invalid winning number");
27     winningId = id;
28   }
29   // claim reward for winners
30   function claimReward() external {
31     require(winningId != 0, "not drawn");
32     ...
33   }
34   function multiBuy(uint64[] ids, uint[] amounts)
35     external {
36     require(winningId == 0, "already drawn");
37     uint totalAmount = 0;
38     for (int i = 0; i < ids.length; i++) {
39       tickets[msg.sender][ids[i]] += amounts[i];
40       totalAmount += amounts[i];
41     }
42     receivePayment(msg.sender, totalAmount),
43   }
44 }
```

Fig. 7: The PancakeSwap Lottery vulnerability

control checks or prevent the unexpected paths.

C. Atomicity Violations (C6)

This type of bug is caused by the interference between concurrent business flows that are supposed to have high level atomicity (higher than the transaction level atomicity).

Example. Figure 7 presents a real-world vulnerability in the PancakeSwap lottery contract [95]. It was reported by an anonymous whitehat and awarded with an undisclosed bounty [96]. Like the lottery in the physical world, the contract users can buy tickets and the owner randomly draws a winner every day. Lines 3-6 define the key state variables, including a three-level mapping `tickets` indicating the amount of each ticket bought by each user (multiple tickets of the same ID can be bought by the same or different users), the winner (`winningId`), and a boolean variable indicating whether the owner is drawing the winner (`drawing`). Function `reset` (line 9) is a privileged function for the owner to start a new round. Function `buy`, starting from line 13, allows users to buy tickets of a specified ID. It first checks that the owner is not drawing and has not drawn the winner, at lines 14 and 15, and further processes the payment and updates `tickets` accordingly. At line 19, function `enterDrawingPhase` is used to start the lottery drawing phase. Variable `drawingPhase` is essentially a lock for the variable `tickets` to prevent further ticket purchase in this round. After entering the drawing phase, function `draw` (lines 23-28) is invoked to set the winner, which enables `claimReward`. There are three business flows, i.e., buying tickets, drawing winners, and claiming prizes. Note that the business flow of drawing winners comprises two functions (`enterDrawingPhase` and `draw`), and hence two transactions. Such a design is critical. Otherwise, an adversary could observe the winner from the `draw` transition in the *mempool*, and bought a huge amount of tickets with the winner's ID. Note that before being mined and finalized on blockchain, transactions are placed in a mempool and visible to the public [97]. Besides, since paying a high gas fee provides incentives for miners, it allows the adversary to preempt the `draw` transaction with his own, and eventually earn a lot of profit illegally. The contract properly prevents this by separating the business flow to two transactions and using a lock `drawingPhase` to ensure atomicity. However, another purchase function `multiBuy` (lines 34-43) does not respect such atomicity. It is a gas-friendly version of function `buy` which allows buying multiple tickets at a time. It updates `tickets` accordingly within the loop in lines 39-40, and receives the payment for all tickets at line 41. However, it does not use the `drawingPhase` lock, making the aforementioned attack possible. This exploit method (preempting a pending transaction belonging to an atomic business flow by paying a higher gas fee) is also called *front running* [98], whose root cause is usually atomicity violation.

Abstract Bug Model and Remedy. There are multiple business flows $\mathcal{B}_1$, ... and $\mathcal{B}_m$ that access some common state variables (e.g., *tickets* in our example). An atomicity violation bug occurs when concurrent business flows yield an unserializable outcome [99]. In our example, after front-running, the amount of tickets for the winner ID is substantially inflated after the winner is decided and before the prizes are claimed. Such a result cannot be achieved by serializing the business flows of drawing winners and claiming prizes. There are a large body of atomicity violation detection tools for traditional programming languages such as Java and C [100]. They may be adapted to detect violations in smart contracts. However, atomic business flows are usually implicit (suggested by boolean flags serving as locks and explicit time bounds). Such challenges need to be addressed during adaptation. Atomicity violation bugs are usually fixed using lock variables (e.g., `drawingPrase`).

D. Other MUB Types

Other MUB types are detailed in our supplementary material (§V - §VII).

Finding 10: *Five out of the seven MUB categories (accounting for 60% of MUBs), namely, all except (C2) accounting errors and (C7) implementation specific bugs, have general abstract models which may serve as oracles for future automated tools.*