# Demystifying Exploitable Bugs in Smart Contracts (Supplementary Material)

Zhuo Zhang*, Brian Zhang†, Wen Xu‡§, Zhiqiang Lin‖
*Purdue University, †Harrison High School, ‡Georgia Institute of Technology, §PNM Labs, ‖Ohio State University,
zhan3299@purdue.edu, bzhangprogramming@gmail.com, wen@pnm.xyz, zlin@cse.ohio-state.edu

***Abstract*—Exploitable bugs in smart contracts have caused significant monetary loss. Despite the substantial advances in smart contract bug finding, exploitable bugs and real-world attacks are still trending. In this paper we systematically investigate 516 unique real-world smart contract vulnerabilities in years 2021-2022, and study how many can be exploited by malicious users and cannot be detected by existing analysis tools. We further categorize the bugs that cannot be detected by existing tools into seven types and study their root causes, distributions, difficulties to audit, consequences, and repair strategies. For each type, we abstract them to a bug model (if possible), facilitating finding similar bugs in other contracts and future automation. We leverage the findings in auditing real world smart contracts, and so far we have been rewarded with $102,660 bug bounties for identifying 15 critical zero-day exploitable bugs, which could have caused up to $22.52 millions monetary loss if exploited.**

***Index Terms*—Blockchain, Smart Contract, Vulnerability, Security, Empirical Study**

## I. Background

**Ethereum Blockchain.** *Ethereum* [1] is an advanced framework for the development of custom financial products on the web. This is made possible through the underlying *blockchain* [2], which provides secure information processing and storage by an append-only public ledger that keeps track of transactions. Groups of transactions are collected within a *block*, where transactions can be *mined* by other users known as miners, who use a visible public key and the hash of a transaction to determine whether the transaction is valid. Miners then vote on whether to accept or revert a transaction. This process results in a "consensus view" of all the transactions. Once transactions within a block are finished, the block is appended onto the blockchain. Ethereum requires anyone who submits a transaction to provide an appropriate amount of *gas*, which is a fee paid to miners when they process transactions. Transactions on Ethereum are transparent and decentralized. As of 29 August 2022, Ethereum has a market capitalization of more than $187 billions [3].

**Smart Contracts.** *Smart contracts* are applications that provide functionalities to realize some business model. They are usually implemented by specific programming languages such as *Solidity* [4] or *Serpent* [5], leveraging the primitive services provided by Ethereum. Smart contracts are publicly available for *users* to access and no changes to the internal states of a smart contract can be hidden, hence the service *transparency*. Smart contracts are owned by *contract owners*, usually the developers. They have access to special functions in the smart contract that are not *callable* by other users.

A smart contract has two kinds of functions: *external* and *internal*. The former can be invoked by a user or the owner, and the latter can only be invoked by another function within the contract. A *transaction* starts when a user invokes an external function. A transaction has *atomicity*, meaning that changes within a transaction are not visible to the outside world until it is committed/mined. Usually, the transaction ends when it is mined and the root external function call returns. A transaction may fail due to a variety of reasons. When this happens, the transaction is undone. This is known as a *revert*. In general, the execution model of smart contract allows one atomic transaction at a time, meaning that it does not allow another invocation of an external function (by user) when one is going on. Smart contracts can interact with each other, constituting a *decentralized finance* (DeFi) [6].

**Solidity.** Syntax-wise, Solidity is similar to Java/JavaScript. A *contract* is similar to a class in Java. Solidity provides a *require* operation that asserts a condition. If the assertion fails, an error message is emitted and the current transaction is reverted. Solidity uses *msg.sender* to denote the caller of current (external) function, and *this* to denote the current contract.

**Address.** On Ethereum and the blockchain, entities such as users and smart contracts are represented by an *address*, or a 20 byte value (e.g., `0xe03a2766325d914898cdA00d4EF927A305786Aa7`).

**Tokens and Crypto-currency.** With the introduction of the blockchain, Ethereum, and smart contracts came the need for currency, in order to realize the business models that developers envision. Ethereum resolved this issue with the creation of *Ethereum Request for Comment* (ERC) tokens. Intuitively, assets are denoted by various kinds of tokens. Tokens can be *fungible* or *non-fungible* (i.e., NFTs). ERC20 [7] tokens are fungible, meaning that they are non-unique and interchangeable. An example would be USDC that denotes a real-world dollar bill. ERC721 [8] and ERC1155 [9] tokens are non-fungible, meaning that they are unique in the making. For example, houses and paintings in the physical world can be represented by NFTs on Ethereum. Tokens can be *minted* (created), *transferred*, or *burned* (destroyed) from a *central* contract, influencing tokens' values, which depend on the amount of real-world assets stored within the central contract

```
1  contract ERC20 {
2    // owner => spender => amount
3    mapping (address => mapping (address => uint256))
4      internal _allowances;
5
6    function _approve(address owner, address spender,
7      uint256 allowance) internal {
8      _allowances[owner][spender] = allowance;
9    }
10
11   function transferFrom(address from, address to,
12     uint256 amount) external {
13     require(_allowances[from][msg.sender] >= amount);
14     _approve(from, msg.sender,
15       _allowances[from][to] - amount);
16     _transfer(from, to, amount);
17   }
18 }
```

Fig. 1: The Redacted Cartel exploit

against the amount of tokens in circulation. For example, one could mint 100 fungible tokens to denote the ownership of an asset, namely, each token denotes 1% of ownership. Users can buy/sell tokens by dealing with their central contracts.

**Exploitable Bugs and A Real-world Example.** We call bugs that can cause direct monetary loss *exploitable bugs*. Figure 1 depicts an example in Redacted Cartel [10]. An ethical hacker reported this bug and was rewarded with a $560,000 bounty. Specifically, it is a fungible token contract which piggy-backs on real-world assets (e.g., USDC tokens backed by US Dollars). The first line defines a contract `ERC20`. Lines 2-4 define `_allowances`, a two-level mapping denoting the amount of fungible tokens that the owner allows a spender to spend. Note that the first-level key is the address of owner, and the second-level key is the address of spender. It is an *internal* field that can only be accessed by the contract's functions. Lines 6-9 defines an internal function, `_approve()`, which updates the amount of allowance. It internally updates `_allowances[owner][spender]` at line 8. Lines 11-17 define a function `transferFrom` that transfers `amount` tokens from address `from` to address `to`. It is an *external* function that can be invoked by any parties including users and other smart contracts. At line 13, the function first validates that `msg.sender` has sufficient allowance from address `from`. It is achieved by the `require` operation. Lines 14-15 update the caller's allowance via function `_approve`. Line 16 invokes `_transfer` to update the balances of `to` and `from`. The bug happens at line 15, where the contract mistakenly uses the allowance of `to` instead of `msg.sender`. That is, the correct allowance to update is `_allowances[from][msg.sender]`. Considering that a victim user Alice grants Bob an allowance of 10 tokens, an adversary Eve can invoke `tranferFrom(Alice, Bob, 0)` without any token transferred. However, since line 15 updates Eve's allowance as `_allowance[Alice][Bob] - 0`, Eve illegally gains 10-token allowance of Bob.

Observe that this bug aligns better with functional bugs in traditional software while being exploitable. Human auditors and automatic tools can hardly detect it without understanding the meaning of `_allowances` and `transferFrom`, as well as the business model. The bug survived multiple rounds of auditing where automatic tools have been applied.

TABLE I: Categories of on-chain projects

| Categories | Description |
|---|---|
| **Lending** | Allow users to borrow and lend assets |
| **Dexes** | Allow users to swap/trade crypto-currency |
| **Yield** | Reward users for their staking |
| **Services** | Service providers, e.g., tokenization |
| **Derivatives** | Projects that get the value, risk, and basic term structure from an underlying asset, e.g., options |
| **Yield Aggregator** | Aggregate yield from a set of other projects |
| **Real World Assets** | Projects that associate their values with real-world assets, e.g., stocks |
| **Stablecoins** | Cryptocurrencies that attempt to peg their market value to some external reference, e.g., US Dollar |
| **Indexes** | Projects that have a way to track the performance of a group of related assets |
| **Insurance** | Projects that provide monetary insurance |
| **NFT Marketplace** | Projects where users can buy/sell/rent NFTs |
| **NFT Lending** | Allow users to colletarize NFTs for loans |
| **Cross Chain** | Provide interoperability among blockchains |

## II. Description of Each DeFi Category

This section provides a comprehensive overview of the various categories within the DeFi domain. A summary of these categories can be found in Table I.

**Lending.** Lending projects facilitate the borrowing and lending of assets between users. Lenders deposit their assets into the project, earning interest, while borrowers borrow assets by providing collateral.

**Decentralized Exchanges (Dexes).** Dex projects provide a platform for users to exchange assets in a decentralized manner. Uniswap, which we have discussed in detail in the main text, is one of the most well-known Dex projects.

**Yield.** Yield projects reward users for staking their funds. Users deposit their funds into the project, and the project invests the funds in various other opportunities. All users share in the profits of the investment, based on their staking shares.

**Services.** Service projects offer essential functionalities that can be utilized by other DeFi projects. For example, governance voting is a popular DeFi service, as is the tokenization of seed investors' and founders' funds.

**Derivatives.** Derivative projects, like their traditional finance counterparts, derive their value from the performance of an underlying entity. Futures and options are two common derivative products in DeFi.

**Yield Aggregators.** Yield aggregator projects pool profits from various yield projects, managing user funds for optimal returns. Given the large number of yield projects with different investment strategies, yield aggregators offer a way for users to maximize their returns.

**Real World Assets.** These projects bring real-world assets onto blockchains, providing investment products whose value is determined by real-world assets, such as stocks.

**Stablecoins.** Stablecoins are a type of cryptocurrency whose value is pegged to a real-world currency, such as the US Dollar, making them less volatile than cryptocurrencies like ETH and

TABLE II: Categories of MABs

| ID | Bug Name | Description |
|---|---|---|
| **AF** | Assertion Failure | Assertion is not satisfied. |
| **AW** | Arbitrary Write | Arbitrary storage data gets overwritten due to mismanaged objects or improper proxies |
| **BD** | Block-state Dependency | Ether transfer depends on block states, e.g., `block.timestamp` or `block.number`. |
| **CE** | Compiler Error | The contract mis-behaves due to using an outdated compiler which contains known bugs. |
| **CH** | Control-flow Hijack | Users can arbitrarily control the destination of a control-flow transfer. |
| **EL** | Ether Leak | User can freely retrieve ether from the contract. |
| **FE** | Freezing Ether | No one can retrieve a (large) portion of locked ether from the contract. |
| **GI** | Gas-related Issue | Execution fails due to insufficient gas. |
| **IB** | Integer Bug | Integer overflows or underflows. |
| **ME** | Mishandled Exception | The contract does not check an exception from external function invocations. |
| **PL** | Precision Loss | Significant precision loss during calculation. |
| **RE** | Reentrancy | A victim function gets re-entered by an untrusted callee, leading to state inconsistency. |
| **SC** | Suicidal Contract | An arbitrary user can destroy the contract. |
| **TD** | Transaction-ordering Dependency | The result of an execution trace depends on another trace sent by a different sender. |
| **TO** | Transaction Origin Use | The result of an execution trace depends on `tx.origin` for user authorization. |
| **UV** | Uninitialized Variable | Uses of uninitialized storage variables. |
| **WP** | Weak PRNG | A pseudo-random number generator (PRNG) relies on predictable variables. |

BTC. There are two main types of stablecoins: asset-backed stablecoins, which are backed by reference assets held by central issuers or smart contracts, and algorithmic stablecoins, which maintain price stability through specialized algorithms.

**Indexes.** Index projects track the performance of a group of assets in a standardized way, much like stock market indices in the physical world. These indexes provide investors with a way to understand current asset prices.

**Insurance.** Insurance is a common financial product in both traditional finance and DeFi, offering users protection for their assets in exchange for insurance fees.

**Non-Fungible Token (NFT) Marketplaces.** NFTs can be bought and sold like physical assets, such as houses and paintings. NFT marketplace projects provide platforms for users to trade their NFTs.

**NFT Lending.** In NFT lending projects, individuals have the ability to use their NFTs as collateral for loans. It is important to note that, in comparison to traditional lending projects, NFT lending projects require the proper valuation of NFTs due to the variable value of each NFT.

**Cross Chain.** Cross-chain projects facilitate communication and interaction between different blockchains. It is important to note that, by design, smart contracts on different blockchains (such as Ethereum and Binance Smart Chain) cannot communicate with one another. Cross-chain projects connect these contracts through secure and trustworthy off-chain services.

## III. Description of Machine Auditable Bugs

In this section, we present a comprehensive overview of the various types of MABs. A summary of the information can be found in Table II.

**Assertion Failure.** As in conventional software, an assertion failure signifies a breach of a user-defined assertion statement.

**Arbitrary Write.** Smart contracts store data in "storage" (equivalent to "memory" in conventional software). The contracts themselves must guarantee that only authorized users or contracts have access to sensitive data. However, if an adversary can modify the storage, they can manipulate sensitive data, such as the contract owner.

**Block-state Dependency.** "Ether" (also known as ETH) is the cryptocurrency used within the Ethereum ecosystem. Users and smart contracts can transfer Ether to one another. If a contract's Ether transfers depend on specific block states, it is susceptible to manipulation by miners. It is important to note that most block states are under the control of miners, meaning they can determine the execution of a specific transaction, regardless of the proposed gas price.

**Compiler Error.** There are known bugs in some outdated Solidity compilers [11] that lead to incorrect compilation results and misbehaving contracts. This vulnerability arises when contracts are compiled using these flawed compilers.

**Control-flow Hijack.** Solidity supports "callback functions". A callback function is a function passed as an argument to another function and invoked later. If a user can control any callback function, they can execute arbitrary code.

**Ether Leak.** As previously mentioned, Ether is the native cryptocurrency of Ethereum. If a user can freely retrieve Ether from a contract, they are essentially stealing funds deposited by other users.

**Freezing Ether.** If a large amount of Ether is locked in a contract, users' funds in Ether are permanently frozen.

**Gas-related Issue.** Ethereum imposes a maximum gas limit for each transaction. If a function requires more computational units than the gas limit allows, the function will always revert due to insufficient gas.

**Integer Bug.** Integer bugs refer to integer overflows or underflows.

**Mishandled Exception.** When calling an external function, the smart contract fails to properly validate the return status.

**Precision Loss.** A precision loss that is significantly amplified during calculation can cause the final result to deviate significantly from the expected outcome.

**Reentrancy.** If a function in a contract can be re-entered, it can lead to an inconsistent state of the contract. Reentrancy bugs have been extensively studied and 29 out of 38 existing techniques are capable of detecting such vulnerabilities.

**Suicidal Contract.** Smart contracts can be destructed through the use of the `selfdestruct` operation. If such a privileged

```
1  contract UniswapV2Pair {
2
3    IERC20 token0; IERC20 token1;
4    uint reserve0; uint reserve1;
5
6    function swapToken0ForToken1(
7      uint amount1Out, address to
8    ) external {
9      token1.transfer(to, amount1Out);
10
11     IUniswapV2Callee(to).uniswapV2Call();
12
13     uint balance0 = token0.balanceOf(address(this));
14     uint balance1 = token1.balanceOf(address(this));
15
16     uint amount0In = balance0 - (reserve0 - amount0Out);
17     uint balance0Adj = balance0 * 1000 - amount0In * 3;
18
19     require(
20       balance0Adj * balance1 >=
21         reserve0 * reserve1 * 1000,
22       "insufficient funds transferred back"
23     );
24
25     reserve0 = balance0; reserve1 = balance1;
26   }
27 }
```

Fig. 2: The swap function of Uniswap

operation can be invoked by users, an adversary can deliberately destroy the contract, resulting in a permanent lock of all other users' funds.

**Transaction-ordering Dependency.** A contract's outcomes can be influenced by the order in which transactions are processed. It is important to note that miners have the ability to determine the order of transactions, making such contracts susceptible to manipulation.

**Transaction Origin Use.** If a contract uses `tx.origin` (the address of the user who initiated the current transaction) for authorization purposes, it can become vulnerable. Consider a vulnerable contract $V$ that uses `tx.origin` for authorization, an authorized user $A$, and a malicious contract $M$. If $A$ calls $M$ and $M$ subsequently calls $V$, $A$'s privileges will be transferred to $M$ as `tx.origin` returns $A$.

**Uninitialized Variable.** In Solidity, uninitialized variables are assigned default values based on their data type (e.g., 0 for `uint`). If a developer uses an uninitialized variable, but assumes an incorrect default value, it can lead to unexpected results.

**Weak PRNG.** A subject contract that uses a pseudo-random number generator (PRNG) that relies on predictable variables can be vulnerable. It is important to note that many variables in the Ethereum blockchain are predictably counterintuitive. For example, an adversary can predict the `block.timestamp` in a victim function by wrapping the function and the victim function into a single transaction, making them share the same `block.timestamp`.

## IV. Price Oracle Manipulation

### A. Uniswap

Figure 2 presents a code snippet of Uniswap's swap function, which instantiates the aforementioned exchange rule. It is critical to understand this function as most existing price oracle exploits entail manipulating this contract *in a legal way*. The code is simplified for the illustrative purpose, and hence slightly differs from the real implementation. Starting from line 1, the code declares a contract `UniswapV2Pair`. Lines 3 and 4 define several state variables, including `token0` and `token1` denoting the two assets for exchange, and `reserve0` and `reserve1` standing for the reserve balances of `token0` and `token1`, respectively. A user invokes function `swapToken0ForToken1()` starting from line 6 to exchange `token0` for `token1`, by specifying the amount of `token1` she demands, namely `amount1Out`, and her address `to` to receive `token1` and pay with `token0`. The transaction is between the user and Uniswp which owns both tokens for trading. The main body of the function is divided into three phases, transferring `token1` (line 9), receiving `token0` (line 11), and verifying the constant-product invariant (lines 13-23), respectively. To transfer `token1` to the user's contract, at line 9, a standard `transfer` function of *ERC20* is invoked, which essentially transfers a specified amount of the underlying asset from the UniswapV2Pair contract to the user. At line 11, an external function call, i.e., `uniswapV2Call`, happens upon the user's contract, within which the user transfers a certain amount of `token0` back to Uniswap. The use of external call enables *flash-loan*, a powerful and unique feature of DeFi. We will elaborate more on flash-loan later in the section. Starting from line 13, the contract verifies whether the constant-product invariant is guaranteed after receiving the user's fund, i.e., whether the user sends back a sufficient amount of `token0`. Variables `balance0` and `balance1` denote the current balances of assets (lines 13 and 14), based on which the amount of received `token0` can be calculated (`amount0In` at line 16). Uniswap charges a contract fee of $0.3\%$, reflected as `balace0Adj` at line 17. Note that `balance0Adj` denotes the amount of the current balance after charging the contract fee with a multiplier of 1000. In lines 19-23, the contract compares the product of reserve balances before and after the exchange. If the check fails, i.e., the user does not pay a sufficient amount of `token0`, the exchange fails with the whole transaction reverted. Given the atomicity of block-chain transactions, the token transfers at lines 9 and 11 get reverted as well, without affecting the user's and Uniswap's funds. Also note that the user is allowed to transfer more funds back, which is profitable for the contract. The reserve balances `reserve0` and `reserve1` are updated accordingly at line 25. There is another function `swapToken1ForToken0` for the exchange in the opposite direction.

*Example.* Consider a Uniswap pair of WETH and USDC with reserve balances $100$ and $400,000$, respectively. The current price of WETH in Uniswap is hence $\$4,000 = 400,000/100$. Assume due to the high volatility of cryptocurrency, the price of WETH (in the rest of the world) drops drastically to $\$1,000$. Assume Alice plans to swap out $100,000$ USDC from Uniswap by invoking the contract's exchange function. According to the constant-product invariant, the contract needs

```
1  function swap(uint amount1Out, address to) external {
2    token1.transfer(to, amount1Out);
3    IUniswapV2Callee(to).uniswapV2Call();
4
5    uint balance0 = token0.balanceOf(address(this));
6    uint balance1 = token1.balanceOf(address(this));
7    uint amount0In = balance0 - (reserve0 - amount0Out);
8    uint balance0Adj = balance0 * 10000 - amount0In * 22;
9    require(
10     balance0Adj*balance1 >= reserve0* reserve1* 1000,
11     "insufficient funds transferred back");
12   reserve0 = balance0; reserve1 = balance1;
13 }
```

Fig. 3: The LFW ecosystem exploit

to hold $100 \times 400,000/(400,000 - 100,000) \approx 133$ WETH after the external call. That is, Alice is required to send $133 - 100 = 33$ WETH back to the contract. Taking the 0.3% contract fee into consideration, the total amount that Alice needs to pay is only $33/(1-0.003) \approx 33.1$ WETH. It becomes extremely profitable for Alice, since she pays 33.1 WETH (worth $33100 since the current real-world price of WETH is $1,000) but gets 100,000 USDC (worth $100,000) back. The profit incentive continually attracts arbitrageurs in the wild and pushes the Uniswap pair towards the balanced status of an WETH price of $1,000, i.e., with 200,000 USDC and 200 WETH reserves, explaining Uniswap's business model.

### B. Example of Flash Loan

Uniswap inherently supports flash loans. Specifically, in Figure 2, Alice specifies the debt amount as `amount1Out` and gets the funds at line 9. Within the external call at line 11, Alice not only trades for arbitrage (or launches the aforementioned exploit) with the borrowed `token1`, but also pays the debts after the trading (or exploit). After line 11, both `balance0` and `balance1` remain unchanged, satisfying the repayment check in lines 19-24.

## V. Erroneous Accounting

This type of bugs is due to incorrect implementation of the underlying financial model formulas. They are difficult to find due to the substantial domain knowledge needed.

*Example.* Figure 3 presents a code snippet of the LFW ecosystem, which has been exploited and lost $0.21 millions. LFW is an AMM contract that allows exchange of two types of assets. A user invokes function `swap()` to exchange `token0` (the first asset) for `token1` (the second) with LFW. Variables `balance0` and `balance1` denote the instant balances of the respective tokens, and `reserve0` and `reserve1` their reserve balances (i.e., committed balances). At line 1, the user specifies the amount of `token1` she demands, namely `amount1Out`, and her address `to` to receive `token1` and send `token0`. The main body of the function is divided into three phases, transferring `token1` (line 2), receiving `token0` (line 3), and verifying the constant-product invariant (lines 5-11), respectively. At line 9, the contract verifies whether the user sends back a sufficient amount of `token0` such that the constant-product invariant is respected. The amount of received `token0` can be calculated from the instant and

```
1  contract NFTMarketReserveAuction{
2    mapping(address => mapping(uint => uint)) auctionIds;
3    mapping(uint => ReserveAuction) idAuction;
4    uint auctionId;
5
6    function createReserveAuction(
7      address nftContract, uint tokenId) external ...{
8      auctionId++;
9      _transferToEscrow(nftContract, tokenId);
10     auctionIds[nftContract][tokenId] =
11       auctionId;
12     idAuction[auctionId] = NewAuction(
13        msg.sender, ..., tokenId, ...);
14     ...
15   }
16   function _transferToEscrow(
17     address nftContract, uint tokenId) internal ...{
18     uint auctionId =
19       auctionIds[nftContract][tokenId];
20     if (auctionId == 0) { // NFT is not in auction
21      super._transferToEscrow(nftContract, tokenId);
22      return;
23    } ...
24   }
25 }
```

Fig. 4: The NFTMarketReserveAuction exploit

the reserve balances (`amount0In` at line 7). LFW charges a contract fee of 0.22%, reflected as `balace0Adj` at line 8. The developers use a multiplier of 10,000 at line 8 (to avoid expensive floating point computation). Lines 9-11 are supposed to check the invariant. However, the developers use a wrong multiplier of 1,000. Since the asset prices are determined by the ratio of their reserve balances, this bug leads to substantial pricing errors. Consider the actual invariant checked by the contract, i.e., $(\texttt{balance0} \times 10 - \texttt{amount0In} \times 0.022) \times \texttt{balance1} \geq \texttt{reserve0} \times \texttt{reserve1}$ (reduced from $(\texttt{balance0} \times 10000 - \texttt{amount0In} \times 22) \times \texttt{balance1} \geq \texttt{reserve0} \times \texttt{reserve1} \times 1000$). The adversary pays only one tenth of the expected `token0` to get `token1` he demands.

**Abstract Bug Model and Remedy.** It is hard to derive general abstract models for erroneous accounting bugs, since such bugs are project/implementation specific. This makes them difficult to find. However, it is encouraging to see that audit contests provide an effective way to expose them (these bugs are the most popular kind among the Code4rena bugs, due to the very broad domain expertise brought by the participants. The fixes are usually simple, including changing coefficients and rephrasing arithmetic expressions.

## VI. ID Uniqueness Violations

These bugs are caused by violations of the uniqueness property of ID fields. They are the 3rd most popular bugs in the auditing stage.

*Example.* This is a real case from an auction contract enlisted for audit in Code4rena [12]. A user acting as a seller can put their NFT up for auction. The bug was caught by only one auditor. If exploited, it could make a winning bidder's funds locked within the smart contract with no direct way of recovery. As shown in Figure 4, to initiate an auction, the seller calls the function `createReserveAuction` at line 6 with the parameters of the address of their NFT contract

(nftContract) and the ID of the token they are selling (tokenId). At line 9, the token is transferred to escrow via function _transferToEscrow (defined at line 16). Inside the function, it looks up an auction using the seller's address and the token ID (line 19). It then checks if this is a new auction by checking if auctionID == 0 on line 20. If so, the NFT is transferred to the escrow via the super class function at line 21. Then the token is marked as in storage at line 10 via auctionIds and a new auction is created at line 12 via NewAuction with all the necessary parameters. The bug lies in that the developers are essentially using the seller address and the NFT token ID to denote an auction (their intention can be inferred from line 19). However, they do not ensure the uniqueness of these data fields.

It can hence be exploited as follows. A (malicious) seller invokes createReserveAuction (line 6) twice using the same nftContract and tokenId. The first invocation correctly transfers the NFT and creates the auction. In the second invocation, the lookup at line 19 simply yields the previously created auction and the check at line 20 falls though (not reverting). A new (duplicate) auction is created at line 12. Now, there are two auctions for the same NFT. Then the adversary cancels the first auction to get the NFT back. However, bidders are still bidding in the duplicate auction. Eventually, someone wins the auction and the contract is supposed to transfer the NFT to the winner. However, since the NFT is already gone, the transfer reverts all the time. Essentially, all the highest bidders' funds are locked in the contract forever. The developers fixed the bug by adding a check: before a NewAuction is created, the storage of toTokenIdToAucionId (line 10) is checked to make sure that the tokenID of the NFT on auction has not been placed in the auction storage before.

**Abstract Bug Model and Remedy.** Variables or data structure fields are used as the ID for some entity/asset and the access to some critical operation (e.g., creating/canceling an auction) is granted based on the ID. However, developers do not check the uniqueness of ID fields. Although this type of bugs is not difficult to find in general, it may require nontrivial efforts to infer if some variables are intended to be an ID, especially when the variable names are not informative. Sometimes, the guarded operation is implicit and distant from the ID check. Developers usually fix these bugs by checking for duplication.

## VII. Inconsistent State Updates

In this type of bugs, developers forget to have correlated updates when they update some variable(s), or the updates do not respect their inherent relations.

*Example.* Figure 5 is an example of a real world exploit that was caught during a Code4rena audit. The bug was caught by only one auditor. The contract itself is an AMM for exchanging two types of tokens, and also a part of the Sushi organization, which has a market capitalization of $171 millions [13]. At lines 2-3, reserve0 and reserve1 are the reserve balances of the two tokens. The bug is in the burn

```
1  contract SushiTrident{
2    uint128 internal reserve0;
3    uint128 internal reserve1;
4
5    function burn(bytes calldata data)
6      public override lock returns (...) {
7      (... , uint128 amount, address recipient...) =
8        abi.decode(data,(int24, int24, ..));
9      // calculates amounts of each reserve to be returned
10     (uint amount0, uint amount1) =
11       _getAmountsForLiquidity(..., amount);
12     //calculate fees to burn from amounts to burn
13     (uint amount0fees, uint amount1fees) =
14       _updatePosition(msg.sender, ..., -amount);
15     ...
16     reserve0 -= uint128(amount0fees);
17     reserve1 -= uint128(amount1fees);
18     //returns the reserve tokens
19     _transferBothTokens(recipient, amount0, amount1,...)
20     ...
21   }
22 }
```

Fig. 5: The SushiTrident exploit

function, beginning at line 5, which is supposed to burn a specific type of ownership token called LP Token, a separate token which designates ownership (shares) of the entire pool of the two reserve tokens, and return the amounts of the reserve tokens corresponding to the burned ownership. This is analogous to cashing out stocks in the physical world. First at line 7, variables amount, representing the amount of LP tokens to burn, and recipient address, are unwrapped from the data parameter. Then, based on the amount (of LP token) (amount0, amount1) of the two reserve tokens are generated at line 10. Fees for the burn operation are decided at line 13. Eventually, the amounts of each reserve token are updated at lines 16-17. The burner is then transferred the amounts of each token at line 19. Observe that only the fees are subtracted from the reserves, and not the actual reserves returned to the receiver. As such, there appears to be more tokens within the contract than there actually are, leading to all sorts of problems like incorrect pricing. Developers patched this bug by subtracting amount0 and amount1 as well. The essence is that when the reserves are updated by the fees, they should be updated by the burned amounts as well.

**Abstract Bug Model and Remedy.** Without losing generality, there are two variables $x$ and $y$, their operations (e.g., reads, writes, and arithmetic operations) tend to co-occur due to inherent (and often implicit) relations, such as amount0 and amount0fee in our example. However, developers forget some operations that are supposed to co-occur. Inferring such co-occurrence relations is the key to detecting these bugs. The fixes entail adding/correcting updates.

## References


[1] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[2] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber, "New kids on the block: an analysis of modern blockchains," *arXiv preprint arXiv:1606.06530*, 2016.

[3] "Ethereum market capital 2022." [Online]. Available: https://coinmarketcap.com/currencies/ethereum/

[4] "Solidity documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.8.15/

[5] "Serpent documentation." [Online]. Available: https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm
[6] L. Zhang, X. Ma, and Y. Liu, "Sok: Blockchain decentralization," *arXiv preprint arXiv:2205.04256*, 2022.
[7] "Erc20 token standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-20
[8] "Erc721 non-fungible token standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-721
[9] "Erc1155 multi-token standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155
[10] "Redacted cartel custom approval logic bugfix review." [Online]. Available: https://medium.com/immunefi/redacted-cartel-custom-approval-logic-bugfix-review-9b2d039ca2c5
[11] "List of known - solidity 0.8.16 documentation," 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.16/bugs.html
[12] "Foundation exploit." [Online]. Available: https://code4rena.com/reports/2022-02-foundation/#h-01-nft-owner-can-create-multiple-auctions
[13] "Sushi price 2022." [Online]. Available: https://coinmarketcap.com/currencies/sushiswap/