#17 Security in Practice

Lecture Notes for CS190N: Blockchain Technologies and Security December 1, 2025

This lecture note covers critical security paradigms in DeFi, focusing on the adversarial nature of the mempool, historical exploits like The DAO and Curve Finance, and the shift from low-level bugs to complex economic vulnerabilities.

1 INTRODUCTION: THE IMMUTABLE ADVERSARIAL ENVIRONMENT

The transition from traditional software development to blockchain engineering requires a fundamental psychological shift. In standard web development, the cost of a bug is typically measured in server downtime, data leaks, or user frustration—consequences that, while serious, are often remediable through patches, rollbacks, or database restorations. In the domain of Decentralized Finance (DeFi), the cost of a bug is measured in the immediate, irreversible, and often total loss of economic value. The environment is not merely passive; it is actively adversarial. The Ethereum mempool, the waiting area for pending transactions, operates as a "Dark Forest" where automated agents—arbitrage bots, liquidation keepers, and malicious exploiters—continuously scan unconfirmed transaction data for profitable inconsistencies.

This lecture analyzes the structural and economic vulnerabilities inherent to smart contract programming. We move beyond the theoretical "Money Legos" of previous lectures to examine the "Money Jenga"—a stack where the removal or failure of a single foundational block can precipitate a systemic collapse. Our analysis focuses on two canonical case studies that define the history of DeFi security: The DAO hack of 2016, which illustrated the peril of execution order (Reentrancy), and the Curve Finance exploits of 2023, which demonstrated the danger of state dependency (Price Manipulation and Read-Only Reentrancy). By dissecting these events, we can develop the "security mindset" necessary to engineer robust decentralized protocols.

1.1 The Stakes of DeFi Security

The necessity of rigorous security practices is evidenced by the staggering magnitude of capital lost to exploits. The ecosystem has witnessed a cumulative loss exceeding **$59 billion** across the broader crypto landscape over the last five years, with DeFi protocols bearing a significant portion of this attrition through smart contract failures. This is not a historical anomaly but an accelerating trend.

Data from the first half of 2025 indicates a sharp escalation in both the frequency and severity of attacks. In the first six months of 2025 alone, the total value stolen reached approximately **$2.17 billion**, a figure that had already surpassed the total losses recorded for the entire year of 2024 (see Table 1). While the number of individual incidents has shown variance, the value per incident has risen, driven by the increasing Total Value Locked (TVL) in protocols and the growing sophistication of attackers.

The distribution of these attacks reveals a shifting landscape. While earlier years were dominated by low-level logic errors in unverified contracts, 2025 has seen a resurgence of "access control" exploits and infrastructure compromises, such as the catastrophic $1.5 billion breach of the Bybit exchange. However, from an engineering perspective, the most critical category remains **protocol logic vulnerabilities**. These are failures in the design of the financial mechanism itself—math errors, state management issues, and oracle dependencies—which accounted for a substantial portion of the remaining losses. For example, losses stemming strictly from smart contract vulnerabilities (excluding private key thefts) reached over $263 million in H1 2025, marking the worst period for code-level security since early 2023.

Table 1. Representative DeFi and Crypto Exploits by Category (2016–2025)

Includes earlier canonical examples (e.g., The DAO 2016) for historical context.

Category	**Key Example**	**Mechanism**	**Impact (Approx.)**
Reentrancy	The DAO (2016)	Recursive `splitDAO` calls before balance update	$60M (historical)
Reentrancy	Curve/Vyper (2023)	Compiler bug failing reentrancy locks	$69M
Price Manipulation	Euler Finance (2023)	Donation to pool + Flash Loan manipulation	$197M
Read-Only Reentrancy	dForce/Sentiment (2023)	Reading Curve virtual price during callback	$4.6M+
Access Control	Bybit (2025)	Compromised private keys/infrastructure	$1.5B
Bridge Exploit	Ronin (2022)	Compromised validator keys	$624M

1.2 The Concept of “Blast Radius” in Composable Systems

The gravity of a vulnerability in DeFi is amplified by **composability**. Protocols are designed to interact permissionlessly; a lending market accepts tokens from a decentralized exchange (DEX), which uses price feeds from an oracle, which aggregates data from other DEXs. This interconnectivity means that a bug in a low-level primitive has a massive “blast radius.”

Consider the Curve Finance exploits. Curve is a foundational AMM for stablecoins. When its pools were compromised, it was not just Curve’s liquidity providers who suffered. Lending protocols like Alchemix, JPEG’d, and Metronome, which used Curve LP tokens as collateral or pricing sources, faced immediate insolvency or bad debt accrual. A single failure in a “safe” base layer can cascade upward, destroying protocols that were, in isolation, written securely. This dependency graph turns security from a local problem (securing your own code) into a systemic problem (securing your dependencies), as illustrated in Figure 1.

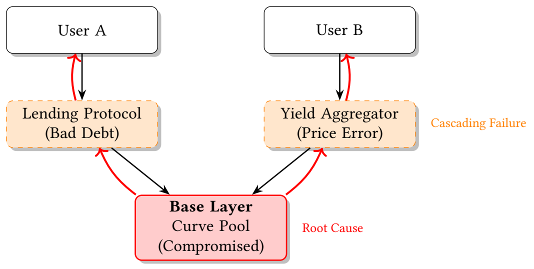

Figure 1. The “Blast Radius”: A failure in the base layer destabilizes all dependent upper layers (red arrows show upward failure propagation).

In Figure 1, a failure in the base Curve pool at the bottom propagates upward. Lending protocols that use Curve LP tokens as collateral misprice positions and accrue bad debt. Yield aggregators on top of them then read these wrong prices, passing the error even further along.

2 THE LANDSCAPE OF VULNERABILITY: A LIGHTWEIGHT TAXONOMY

Before examining specific case studies, it is essential to establish a taxonomy of common vulnerabilities. This framework allows developers to categorize risks and apply appropriate defensive patterns. The following taxonomy groups vulnerabilities by their root cause: arithmetic, ordering, resource exhaustion, and state management.

2.1 Arithmetic Vulnerabilities: Overflow and Underflow

In the Ethereum Virtual Machine (EVM), nearly all numeric types are fixed-size integers, most commonly `uint256`. Unlike high-level languages like Python, which automatically handle large numbers, or Java, which throws exceptions on overflow, early versions of Solidity allowed arithmetic operations to wrap around.

- **The Mechanism:** If a variable `x` is a `uint8` (holding values 0-255) and holds the value `0`, the operation `x - 1` does not yield `-1`. Instead, it underflows to the maximum value, 255. Conversely, `255 + 1` overflows to `0` (see Figure 2). You can think of the `uint8` range [0, 255] as points on a clock: moving one step past 255 wraps around to 0, and moving one step below 0 wraps to 255.
- **The Risk:** This behavior is catastrophic in financial contexts. If a user withdraws 1 token more than they own, an underflow could update their balance to a near-infinite amount ($2^{256} - 1$), effectively allowing them to drain the entire protocol.
- **Modern Mitigation:** Since **Solidity 0.8.0**, the compiler automatically injects checked arithmetic opcodes. Any operation that results in an overflow or underflow causes the transaction to revert. However, developers must still be vigilant when using `unchecked {...}` blocks for gas optimization, or when performing logic that might not strictly overflow but yields incorrect financial results (e.g., precision loss in division).

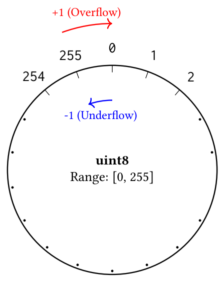

Figure 2. Integer Overflow/Underflow visualized as a circular buffer.

2.2 Transaction Ordering Dependence (Front-Running & MEV)

The blockchain does not execute transactions instantaneously. Users submit transactions to a public mempool, where they sit in a "pending" state until a block producer (miner or validator) selects them for inclusion in a block.

- **The Mechanism:** Because the mempool is public, any observer can see a user's intent before it happens. You can think of the mempool as a public "waiting room" for transactions, and whoever controls the ordering inside that room can sometimes extract profit just by choosing who goes first. If User A submits a transaction to buy a large amount of Token X on a DEX, an attacker can observe this and submit their own buy order with a higher "gas tip." The block producer, incentivized by the higher fee, will order the attacker's transaction *before* User A's.
- **The Sandwich Attack:** This is the most common form of malicious MEV (Maximal Extractable Value). In simple terms, MEV is the extra profit a block producer or attacker can earn purely by reordering, inserting, or censoring transactions.
  (1) **Front-Run:** Attacker buys Token X, driving the price up.
  (2) **Victim:** User A's buy executes at the inflated price, pushing it even higher.

(3) **Back-Run:** Attacker sells Token X immediately after, locking in a risk-free profit at User A's expense (visualized in Figure 3).

- **Defensive Patterns:** A basic defense is to set a slippage tolerance so the trade automatically reverts if the price moves too much. More advanced infrastructure lets users send transactions through private relays, hiding them from the public mempool until they are included on-chain.

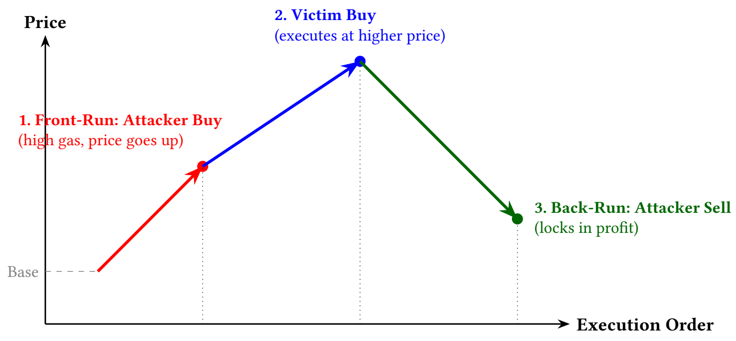

Figure 3. Visualizing the Sandwich Attack execution order and price impact.

2.3 Denial of Service (DoS)

DoS in smart contracts typically results in funds being frozen rather than a server crash.

- **Unbounded Iteration:** If a contract iterates over an array of users to pay dividends (e.g., `for (i=0; i<users.length; i++)`), the gas cost of the function grows linearly with the number of users. In a normal off-chain program, a large for-loop might 'just take longer.' On-chain, the entire loop must finish inside a single block's gas limit. If it needs more gas than a block allows, the whole transaction reverts, and the function effectively becomes impossible to call. Eventually, the gas cost will exceed the block gas limit, making the function impossible to execute, as shown in Figure 4.
- **Griefing (The "King of the Ether" Pattern):** If a contract attempts to send ETH to a user using `.transfer()` or `.send()`, and that user is a smart contract that reverts all incoming transactions, the sending contract might fail. If the sending contract handles this failure by reverting the entire transaction, a single malicious user can halt the system for everyone.
- **Mitigation:** The "Pull over Push" pattern is the standard solution. Instead of the contract sending funds to users (Push), the contract updates a balance mapping, and users must initiate their own withdrawal transactions (Pull), as demonstrated in Listing 1.

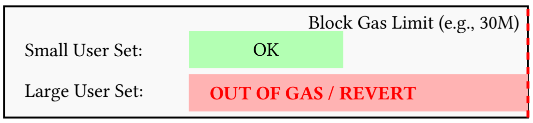

Figure 4. DoS Vulnerability: Unbounded loops hitting the Block Gas Limit.

Listing 1. Secure Pattern: Pull over Push

```
// SECURE PATTERN
```

```
mapping(address => uint) public pendingWithdrawals;

function withdraw() public {
    uint amount = pendingWithdrawals[msg.sender];
    require(amount > 0);
    // Important: Zero out balance BEFORE sending to prevent reentrancy
    pendingWithdrawals[msg.sender] = 0;
    payable(msg.sender).transfer(amount);
}
```

2.4 selfdestruct and Forced Ether

One of the quirks of the EVM is that a contract cannot strictly prevent ETH from being sent to it. While a contract can omit a `payable` fallback function to reject normal transfers, it cannot reject ETH sent via the `selfdestruct` opcode.

- **The Mechanism (historically):** When a contract calls `selfdestruct(targetAddress)`, its remaining ETH is forcibly transferred to `targetAddress`, bypassing all code execution (including fallback functions) on the target.
- **Note (Post-EIP-6780):** On modern Ethereum, `selfdestruct` only truly deletes the contract bytecode if the contract was created within the same transaction. In all other cases, it behaves essentially as a "send-all ETH" operation that transfers all remaining funds without deleting the account, but it still forces ETH into the recipient without triggering a fallback.
- **The Vulnerability:** Developers often write logic assuming `address(this).balance` perfectly tracks the sum of user deposits. For example, a game might end when `address(this).balance == 100 ether`. An attacker can force-send 0.000001 ETH to the contract via `selfdestruct`. The balance becomes `100.000001 ether`, the equality check fails, and the game enters a broken state (see Listing 2).
- **Mitigation:** Never rely on `address(this).balance` for critical internal logic. Always use a separate state variable (e.g., `totalDeposited`) that is only incremented by valid deposit functions.

Listing 2. Vulnerable Logic: Strict Equality Check

```
// VULNERABLE CODE
function endLottery() public {
    // Attack: Send 1 wei via selfdestruct
    // Result: balance is 100.000...01, condition never met
    if (address(this).balance == 100 ether) {
        winner.transfer(reward);
    }
}
```

3 REENTRANCY IN PRACTICE: THE DAO HACK

The concept of **Reentrancy** is foundational to smart contract security. It is not merely a bug; it is a consequence of the synchronous, atomic nature of cross-contract calls in the EVM. The exploitation of this vulnerability in The DAO hack of 2016 remains the single most influential event in Ethereum's history, leading to a hard fork and the creation of Ethereum Classic.

3.1 Mechanics of a Reentrancy Attack

To understand reentrancy, one must think in terms of **control flow**. In standard programming, when Function A calls Function B, Function A pauses until Function B returns. In the EVM, if Function A calls an external contract (Function B), it hands over control of the execution thread. If Function B is malicious, it can leverage this control to call back into Function A before Function A has finished its work.

The vulnerability arises when a contract violates the **Checks-Effects-Interactions** pattern. Specifically, if a contract interacts with the outside world (sends money) *before* it updates its internal state (deducts balance), the state effectively "lies" about reality during the external call.

The Simplified Vulnerable Pattern. Consider a naive vault contract (Listing 3):

Listing 3. Vulnerable Code Pattern

```
// VULNERABLE CODE PATTERN
function withdraw() public {
    uint balance = userBalances[msg.sender];
    require(balance > 0);

    // INTERACTION: Sending ETH happens here
    // Control is handed to the receiver
    (bool success, ) = msg.sender.call{value: balance}("");
    require(success);

    // EFFECT: Balance is updated here
    // This line has not yet executed when reentrancy occurs
    userBalances[msg.sender] = 0;
}
```

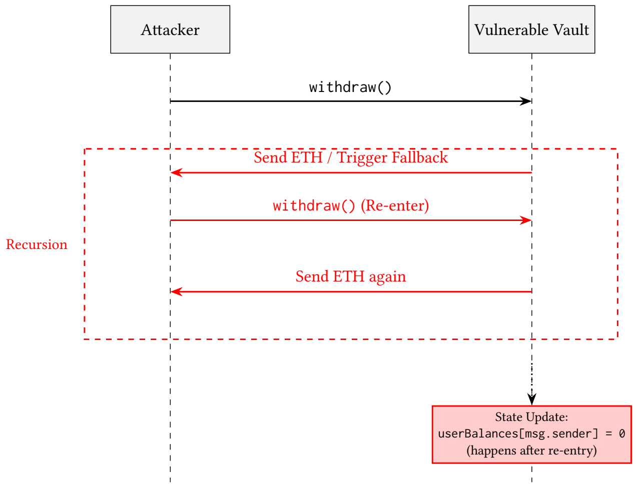

Figure 5. Reentrancy Attack Sequence: Note how the State Update happens after the recursion. Here, the "Vulnerable Vault" on the right is exactly the `withdraw()` function from Listing 3, and the attacker's `fallback()` behavior is implemented by the contract in Listing 4.

An attacker exploits this by deploying a contract with a `fallback` function. When the vault sends ETH, the attacker's `fallback` is triggered. Inside the `fallback`, the attacker calls `withdraw()` again.

Because `userBalances[msg.sender] = 0` has not yet run, the vault sees the *original* positive balance and sends the ETH a second time. This loop continues until the transaction runs out of gas or the vault is drained (see Listing 4 for the attacker's perspective). The sequence of this attack is visualized in Figure 5.

Listing 4. The Attacker's Recursion Logic

```
// Assume 'vault' is an instance of the vulnerable contract from Listing 3.
// ATTACKER CONTRACT
fallback() external payable {
    if (address(vault).balance >= 1 ether) {
        // Recursively call withdraw again
        vault.withdraw();
    }
}

function attack() external payable {
    vault.deposit{value: 1 ether}();
    vault.withdraw(); // Starts the loop
}
```

3.2 The DAO: A "Decentralized Venture Capital Firm"

"The DAO" was launched in April 2016 as a paradigm shift in corporate governance. It was a smart contract system acting as an investor-directed venture capital fund. It raised over **12 million Ether** (worth ~150 million USD at the time), representing roughly 14-15% of all ETH in circulation.

The core functionality allowed token holders to vote on investment proposals. Crucially, to prevent the "tyranny of the majority," the protocol included an exit mechanism called `splitDAO`. This function allowed a user to burn their DAO tokens and receive their proportionate share of ETH into a new "Child DAO".

3.3 The Exploit Logic

The `splitDAO` function contained the critical reentrancy flaw. The simplified logic flow was as follows:

(1) **Check:** The contract calculated how much ETH the user was entitled to based on their token balance.
(2) **Interaction (The Trigger):** The contract called `withdrawRewardFor(msg.sender)` to pay out any accrued rewards and moved the principal ETH to the new Child DAO. This involved sending ETH, which triggered the recipient's code.
(3) **Effect:** *After* the external calls returned, the contract zeroed out the user's DAO token balance and burned the tokens.

The Attack Sequence (June 17, 2016):

(1) The attacker deployed a malicious contract and amassed DAO tokens.
(2) The attacker initiated a `splitDAO` proposal.
(3) When the DAO contract sent the ETH (Step 2), the attacker's contract intercepted the control flow.
(4) From within the callback, the attacker called `splitDAO` again.
(5) Because the DAO tokens had not yet been burned (Step 3 had not executed), the contract believed the attacker still held the full balance.
(6) The contract sent the ETH again.

(7) The attacker repeated this recursion roughly 20-30 times per transaction (limited by the block gas limit and stack depth).

By repeating this process in multiple transactions, the attacker drained approximately **3.6 million ETH** (worth ~60 million at the time) into a Child DAO under their control. In other words, The DAO effectively followed a "Check → Interaction → Effect" order: it sent ETH out before updating its internal state.

3.4 The Hard Fork and Aftermath

The scale of the theft posed an existential threat to Ethereum. If the "code is law" philosophy were strictly upheld, the attacker would control a significant percentage of the total supply.

This led to a contentious community decision. The majority of miners and developers agreed to implement an irregular state change—a **Hard Fork**. This fork moved the stolen funds from the attacker's Child DAO into a recovery contract, allowing original investors to withdraw their ETH. The minority who rejected this intervention continued mining the original chain, which survives today as **Ethereum Classic (ETC)**.

3.5 Lessons: The Checks-Effects-Interactions Pattern

The DAO hack codified the most important rule in smart contract development: the **Checks-Effects-Interactions (CEI)** pattern (Listing 5). The fix is conceptually simple: keep the Check, but move the Effect (state update) before any Interaction (external calls), turning the unsafe "Check → Interaction → Effect" into the safe "Checks → Effects → Interactions" pattern.

Corrected Logic:

(1) **Checks:** Validate inputs and conditions (`require(balance > 0)`).
(2) **Effects:** Update all internal state variables **first** (`balance = 0`).
(3) **Interactions:** Interact with external contracts or send ETH **last** (`msg.sender.call{value: amount}("")`).

Listing 5. Corrected Logic: Checks-Effects-Interactions

```
// SECURE CODE PATTERN
function withdraw() public {
    uint balance = userBalances[msg.sender];

    // 1. CHECKS
    require(balance > 0, "Insufficient balance");

    // 2. EFFECTS (Update state FIRST)
    userBalances[msg.sender] = 0;

    // 3. INTERACTIONS (External call LAST)
    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Transfer failed");
}
```

If The DAO had updated the token balance *before* sending the ETH, the recursive call would have seen a zero balance and failed. Additionally, modern development standards mandate the use of **Reentrancy Guards** (e.g., OpenZeppelin's `nonReentrant` modifier), which uses a mutex flag to lock a function during execution, preventing any re-entry regardless of logic order.

4 PRICE MANIPULATION IN PRACTICE: CURVE-STYLE EXPLOITS

While reentrancy attacks leverage the execution flow of a single contract, **Price Manipulation** attacks exploit the economic dependencies between contracts. These attacks have become the dominant vector in the modern DeFi era, particularly with the advent of **Flash Loans**—uncollateralized loans that must be borrowed and repaid within a single transaction. Flash loans provide attackers with nearly infinite capital to skew market prices temporarily.

4.1 The Vulnerability: Spot Price Dependency

Many DeFi protocols, such as lending markets, require real-time pricing of assets to determine collateral health. If a lending protocol calculates the value of a user's collateral by simply querying the current "spot price" on a Decentralized Exchange (DEX) like Uniswap or Curve, it is vulnerable.

An attacker can:

(1) Borrow huge capital via a Flash Loan.
(2) Dump massive liquidity into the DEX, crashing the spot price of an asset (or pumping it).
(3) Interact with the lending protocol, which reads this manipulated price.
(4) Borrow funds at an undervalued rate or liquidate other users unfairly.
(5) Correct the pool price and repay the flash loan, pocketing the profit.

4.2 Case Study: Curve Finance and Read-Only Reentrancy

Curve Finance is a specialized AMM optimized for stablecoin swaps (e.g., USDC/DAI) and pegged assets (e.g., ETH/stETH). It uses a complex invariant that creates a flatter pricing curve than standard AMMs, offering lower slippage (see Figure 6).

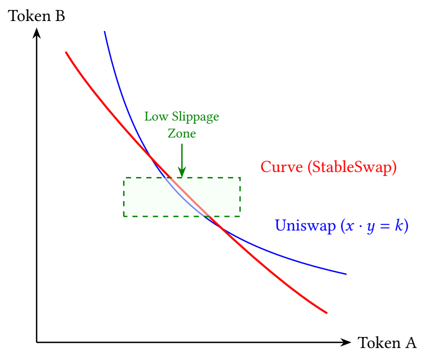

Figure 6. Conceptual comparison of Uniswap vs. Curve invariant curves. Note Curve's flatter central region.

In 2023, a sophisticated class of vulnerabilities known as **Read-Only Reentrancy** emerged, affecting protocols that integrated with Curve. It is crucial to distinguish this from a flaw in Curve's own funds; Curve's pools were generally safe. The victims were the *integrators*—lending protocols like **dForce**, **Sentiment**, and **Midas**—that trusted Curve's price data during a specific, transient state. Crucially, Curve's pools end each transaction in a consistent state and did not lose funds in this read-only reentrancy pattern; the economic loss is borne entirely by the protocols that integrated Curve's `get_virtual_price()` as an oracle.

Difference at a glance (see Table 2):

- **Normal Reentrancy:** The attacker repeatedly withdraws or moves funds before state is updated, directly draining money from the victim contract.

- **Read-Only Reentrancy:** The attacker exploits intermediate state to corrupt views or oracles (e.g., prices or balances), then tricks other protocols in the same transaction into making bad decisions (over-borrowing, unfair liquidations, etc.), so the actual monetary loss occurs in those integrated protocols or LPs.

Table 2. Comparison of Reentrancy Types

Feature	**Classic Reentrancy (The DAO)**	**Read-Only Reentrancy (Curve Integrations)**
Action	Attacker writes data (withdraws funds)	Attacker reads data (price/balance)
State	State is updated *after* the call	State is inconsistent (tokens burned, balances high)
Victim	The contract being called (The DAO)	A third-party contract reading the called contract
Defense	`nonReentrant` modifier + CEI Pattern	Reentrancy check on the *view* function

The Mechanism: `get_virtual_price` *Manipulation.* Protocols often value Curve Liquidity Provider (LP) tokens using the function `get_virtual_price()`. This function calculates the value of one LP token share relative to the underlying assets in the pool. The formula, conceptually, is:

$$\text{Virtual Price} = \frac{D\ \text{(Total Invariant Value)}}{\text{Total Supply of LP Tokens}}$$

Where D represents the total value of assets in the pool.

The vulnerability arises because of how Curve handles the `remove_liquidity` function. When a user removes liquidity (withdraws assets):

(1) **Burn LP Tokens:** Curve burns the user's LP tokens immediately. This reduces the `Total Supply` in the denominator.
(2) **Send Assets:** Curve sends the underlying assets (e.g., ETH) to the user.
(3) **Update Balances:** Curve updates its internal accounting of the pool's balances (the D value).

The attack vector lies in **Step 2**. When Curve sends ETH to the user, it triggers the user's `fallback` function. At this exact moment:

- The **Total Supply** has decreased (Step 1 complete).
- The **Internal Balances** (D) have *not* yet updated (Step 3 pending).

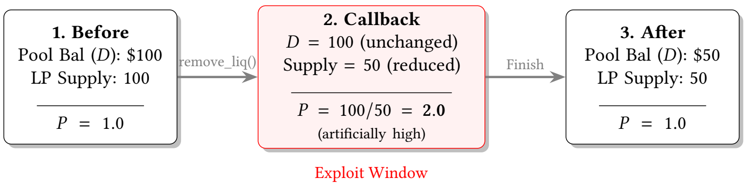

Figure 7. The Logic Error in Read-Only Reentrancy: Inconsistent State Views.

Consequently, the `get_virtual_price` formula calculates:

$$\text{Virtual Price} = \frac{\text{Old (High) } D}{\text{New (Low) Total Supply}}$$

This results in a massive, artificial spike in the Virtual Price (Figure 7). Concretely, in Figure 7: Before, $D = 100$ and Total Supply = 100, so $P = 100/100 = 1.0$. During the callback, D is still 100 but Total Supply has dropped to 50, so $P = 100/50 = 2.0$. After the transaction completes, both D

and Total Supply are 50 again, so P returns to 1.0. The attacker exploits this temporary $P = 2.0$ view during the callback.

The Exploit Steps (dForce / Sentiment Attacks). At a high level, the attacker first inflates the Curve LP price inside a single transaction, then uses that temporarily inflated price as collateral in a lending protocol before the price snaps back.

(1) **Flash Loan:** Attacker borrows substantial ETH.
(2) **Deposit:** Attacker deposits ETH into a Curve pool to mint LP tokens.
(3) **Trigger:** Attacker calls `remove_liquidity` to withdraw ETH.
(4) **Re-entry:** Receiving the ETH triggers the attacker's fallback function.
(5) **The Trap:** Inside the fallback, the attacker calls the victim protocol (e.g., dForce).
(6) **The Deception:** dForce checks the value of the attacker's collateral (Curve LP tokens). It calls `get_virtual_price()`, which returns the manipulated, skyrocketed value.
(7) **The Theft:** dForce believes the attacker is incredibly wealthy and allows them to borrow a massive amount of stablecoins against the (actually worthless) LP tokens.
(8) **Exit:** The transaction finishes, Curve updates the balances, the price returns to normal, and dForce is left with bad debt.

This specific pattern cost dForce roughly **$3.6 million** and Sentiment **$1 million**.

4.3 Distinct Event: The Vyper Compiler Bug (July 2023)

It is vital to distinguish the "Read-Only Reentrancy" attacks (early 2023) from the "Vyper Compiler Bug" that hit Curve directly in July 2023. A comparison of these reentrancy types is provided in Table 2.

While the former was an integration logic error, the latter was a catastrophic failure in the toolchain itself. Specific versions of the **Vyper** programming language (0.2.15, 0.2.16, and 0.3.0) contained a bug where the reentrancy guard (`@nonreentrant`) failed to function correctly due to a compiler error in storage slot allocation. Root cause: the compiler assigned inconsistent storage slots to the reentrancy lock for different functions, so the `@nonreentrant('lock')` modifier looked correct in source code but compiled to bytecode that never actually enforced mutual exclusion.

This allowed attackers to perform a classic reentrancy attack (similar to The DAO) on Curve pools themselves (CRV/ETH, alETH/ETH), draining over **$69 million**. This highlights a deeper layer of security risk: even if the code logic is perfect, the compiler translating it to bytecode effectively introduces a vulnerability. This underscores the importance of "Supply Chain Security" in DeFi—verifying not just the contract, but the tools used to build it. Lesson: even perfect-looking Solidity or Vyper source code is not enough—in DeFi, you must also trust and keep up to date with the compilers and build tools that turn your code into bytecode, a concern often called 'supply chain security.'

5 SYSTEMIC RISKS AND EMERGING VULNERABILITIES

Beyond the canonical examples of Reentrancy and Price Manipulation, the DeFi landscape is populated by a diverse array of vulnerability classes. Building robust systems requires familiarity with these patterns.

5.1 Access Control and Centralization Risks

As highlighted by the 2025 statistics, **Access Control** exploits have surged. Smart contracts often contain privileged functions (e.g., `mint()`, `pause()`, `upgrade()`) protected by modifiers like `onlyOwner`.

- **The Vulnerability:** If the private key controlling the `Owner` account is compromised (via phishing, poor OpSec, or malware), the attacker gains "god mode" over the protocol.
- **The "Rug Pull":** Malicious developers may intentionally leave backdoors or use these privileges to drain user funds.
- **Mitigation:** Decentralization of power. Use **Multi-Signature Wallets (Multisigs)** (requiring M-of-N keys to execute) and **Timelocks** (forcing a delay of 24-48 hours before a privileged action executes, giving users time to exit).

5.2 Flash Loans as Amplifiers

Flash loans are not a vulnerability in themselves; they are a financial primitive. However, they act as a force multiplier for other bugs. A price manipulation attack that requires $100 million capital is impossible for most hackers. With flash loans, any user can borrow $100 million for the cost of a single transaction fee, provided they repay it in the same block. This democratizes the ability to attack protocols, meaning that *any* economic inefficiency that can be exploited with large capital *will* be exploited.

5.3 Denial of Service (DoS) via Gas Limits

Smart contract execution is bounded by the "Block Gas Limit." As detailed in Section 2.3, a "Push" payment system that loops over a large `users` array (Listing 6) will eventually exceed this limit, making the function uncallable and locking funds.

Listing 6. Vulnerable Pattern: Unbounded Loop

```
for (uint i = 0; i < users.length; i++) {
    users[i].transfer(dividend);
}
```

- **Resilience:** Avoid global failure on a single recipient; design batch payouts so that a single revert or high-gas path cannot block the entire system (e.g., log failed payouts for manual retry).

5.4 Signature Replay and EIP-712

Many modern protocols use off-chain signatures (permits) for gasless transactions. A common vulnerability is **Signature Replay**, where a signature intended for one specific action is reused by an attacker. Figure 8 illustrates this with a check-like signed message: without a nonce or chainId, the same 'check' can be cashed multiple times.

- **Risk:** If a signature does not include a `nonce` (a counter) or the `chainId`, an attacker might replay the same signature to execute a withdrawal multiple times or execute a transaction on a different fork of the blockchain (Figure 8).
- **Mitigation:** Always include a nonce and the chainId (and, ideally, a contract-specific domain separator) in the signed message. EIP-712 ('typed structured data') standardizes this pattern so that wallets sign exactly the fields your contract expects, making replay on a different contract or chain much harder.

6 THE SECURITY MINDSET: A DEVELOPER'S CHECKLIST

Security is not a feature; it is a process. While formal verification and professional audits are essential for production code, every developer must internalize a basic checklist to identify "code smells" during the design phase. You do not need to memorize every bullet immediately—think of this checklist as a compact reference to scan when you design or review a contract.

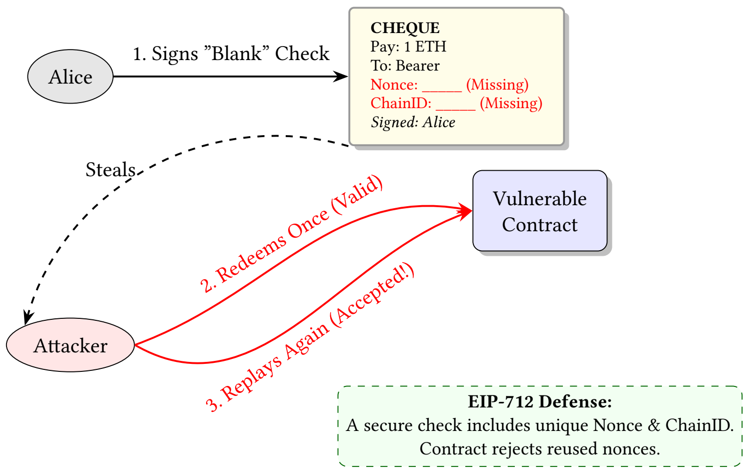

Figure 8. Signature Replay: Reusing a signed message (analogous to a check with missing fields) multiple times.

6.1 The Lightweight Checklist

1. Logic & Flow:

Checks-Effects-Interactions: Are state updates performed *before* any external calls?
Reentrancy Guards: Is the `nonReentrant` modifier applied to all state-changing functions, especially those that transfer assets?
Pull vs. Push: Does the contract avoid iterating over arrays to send funds? Are users required to claim their own assets?

2. Data & Oracles:

Oracle Hardening: Does the protocol rely on a spot price from a single DEX? (Critical Failure). Does it use a TWAP (Time-Weighted Average Price) or a decentralized aggregator like Chainlink?
Stale Data: Does the code check if the oracle data is fresh (last update timestamp)?
Read-Only Safety: If integrating with complex systems (like Curve or Balancer), does the protocol check if the external system is in a read-only reentrancy-safe state?

3. Access Control:

Least Privilege: Is the `Owner` role strictly necessary? Can it be renounced?
Timelocks: Are critical parameter changes delayed by a Timelock contract?
Key Management: Is the admin address a Multisig, not a single EOA (Externally Owned Account)?

4. Arithmetic & Types:

Solidity Version: Is the contract using Solidity 0.8.0+ to prevent silent overflows?
Unit Consistency: Are token decimals handled correctly? (e.g., USDC is 6 decimals, DAI is 18. Mixing them without conversion is a common bug).

6.2 Conclusion: The Path Forward

The history of DeFi is written in the autopsy reports of failed protocols. The DAO taught us that execution order is paramount. The Curve exploits taught us that data validity is conditional on state consistency. The billions lost in 2024 and 2025 serve as a stark reminder that the adversary is constantly evolving.

This domain offers a unique challenge: writing code that must execute correctly in an adversarial, open, and immutable environment. Future lectures will expand on **Formal Verification** (proving code correctness mathematically with tools such as model checkers) and **MEV Mitigation** strategies (protocol designs that reduce the impact of adversarial transaction ordering), which represent the frontier of blockchain security research. But the foundation remains simple: distrust external calls, verify all data, and assume that if your code *can* be broken, it *will* be.

REFERENCES