# #6 Smart Contract Design Patterns

Lecture Notes for CS190N: Blockchain Technologies and Security — October 15, 2025

This lecture develops a practical vocabulary of smart-contract design patterns for adversarial and immutable environments. We begin by motivating why patterns matter with well known incidents and by framing each pattern with its intent, actors, and invariants. We then build four modules from compact Solidity fragments. First, reentrancy defenses establish order with Checks-Effects-Interactions and non-reentrant gates. Second, an escrow separates decision from money movement using pull-over-push payouts, a single withdrawal path, and explicit liveness and conservation. Third, a voting governor turns collective intent into scheduled, auditable change through snapshot weighting, quorums and thresholds, allowlisted targets, and a timelock that decouples decision from execution. Fourth, a commit-reveal auction encodes a time-based state machine, one-time commitments, on-chain verification, and centralized payouts. Along the way we analyze typical failure modes and show how disciplined state transitions and restricted external calls make outcomes predictable and funds retrievable. By the end, you will be able to read and write minimal yet safe contracts that surface intent early, route value through a single withdrawal function, schedule governance actions with confidence, and reason about correctness via clear invariants and timelines.

## 1 INTRODUCTION

*Why design patterns?* Public code and expensive rollbacks mean small ordering mistakes can move large amounts of value. Several incidents emphasize why explicit patterns matter: The DAO (2016) [2] exploited a reentrancy window to withdraw repeatedly (about $50–$60 M); Parity Multisig (2017) [4] involved a delegatecall/library initialization flaw that affected roughly 153,000 ETH (≈ $30–34 M at the time); bZx (2020) [1] used flash loans to manipulate on-chain prices for profitable trades (≈ $8 M in a major attack); Harvest Finance (2020) [3] saw fast trades distort AMM prices and drain vault liquidity (≈ $24 M). The common lesson is to encode order, trust boundaries, and value movement so that reviewers can reason about every path.

*Roadmap.* We begin with reentrancy as the base defensive habit; then an escrow that separates decision from transfer; then a voting governor that decides early and executes late; and finally a time-gated commit–reveal auction. Each chapter opens with a figure sketching the flow, lists the patterns, shows minimal function-level code, and explains how the code realizes each pattern.

## 2 REENTRANCY

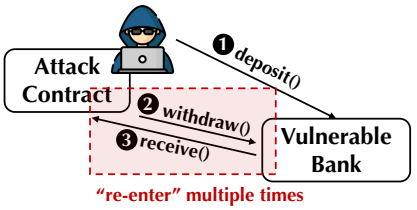


Fig. 1. Reentrancy: deposit once, then loop withdraw() and receive().

*What is reentrancy?* As shown by Figure 1, a reentrancy bug appears when a contract makes an external call before its own state is finalized. The receiver can re-enter while stale state still holds, repeating a transition such as a withdrawal. The diagram shows one deposit followed by a loop between `withdraw()` in the victim and `receive()` in the attacker.

*Patterns in this chapter.*

- **Checks–Effects–Interactions (CEI):** finalize internal state, then interact externally.
- **Non-reentrant gate:** block nested entries on interaction-heavy functions.

*How it works: pattern-to-code.* Start from a minimal vulnerable core, then apply the patterns.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract VulnerableBank {
    mapping(address => uint256) public balanceOf;

    function deposit() external payable {
        require(msg.value > 0, "no ether");
        balanceOf[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 bal = balanceOf[msg.sender];
        require(bal > 0, "no balance");
        (bool ok, ) = msg.sender.call{value: bal}("");
        require(ok, "send failed");
        balanceOf[msg.sender] = 0;
    }
}
```

This `withdraw()` violates CEI because the external call happens while the sender's balance is still positive. A receiver that implements `receive()` can call back into `withdraw()` and observe the old balance again.

Applying CEI closes the window:

```
function withdraw() external {
    uint256 bal = balanceOf[msg.sender];
    require(bal > 0, "no balance");
    balanceOf[msg.sender] = 0;
    (bool ok, ) = msg.sender.call{value: bal}("");
    require(ok, "send failed");
}
```

Now any re-entry sees the post-withdraw state and fails the balance check. A simple non-reentrant gate adds a second layer when a function must call out:

```
bool internal locked;
modifier noReentrant() {
    require(!locked, "no re-entrancy");
    locked = true; _;
    locked = false;
}
```

*Why these lines implement the patterns.* CEI is realized by moving the state update *before* the external call. That makes the transition atomic with respect to reentry: nested calls cannot observe the pre-update balance. The non-reentrant modifier prevents the same entry point (directly or indirectly) from running while it is already active, eliminating complex interleavings when a path has to interact multiple times.

*Summary.* Think in terms of order: effects first, interactions later. Treat every external call as a potential reentry point; use CEI on all interacting paths and a gate for extra defense when needed.

## 3 ESCROW

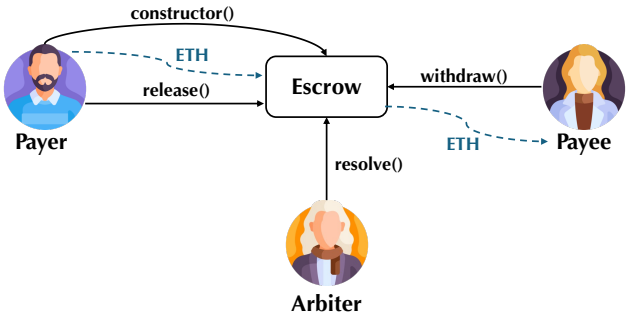


Fig. 2. Escrow roles and calls. Payer funds in the constructor, may release; arbiter can resolve; the winner later withdraws.

*What is an escrow?* As shown by Figure 2, an escrow holds a payer's deposit for a payee. The deal resolves to exactly one outcome: the payer approves, a deadline passes, or an arbiter decides. No Ether is sent during the decision; value moves only when the winner later pulls it from the contract.

*Patterns in this chapter.*

- **Pull-over-Push:** record entitlements; receivers pull later via `withdraw()`.
- **CEI on terminal paths:** set state and credits before any external call.
- **Single withdrawal path:** centralize all transfers in one function.

*How it works: pattern-to-code.* We separate deciding the outcome from moving Ether. The program state fixes roles and amount at deployment, and exposes only one path that moves funds.

```
enum State { Funded, Resolved }
State s;
address payer; address payee; address arbiter;
uint256 amount; uint256 deadline;
mapping(address => uint256) credit;

constructor(address _payee, address _arbiter, uint256 _deadline) payable {
    require(msg.value > 0, "no funds");
    payer = msg.sender; payee = _payee; arbiter = _arbiter;
    amount = msg.value; deadline = _deadline; s = State.Funded;
}
```

*Why these lines matter.* Fixing `payer`, `payee`, `arbiter`, and `amount` in the constructor removes ambiguity about roles and supply. The `require(msg.value > 0)` rejects empty deals that would make later branches degenerate. Setting `s = State.Funded` encodes that exactly one terminal transition is still allowed; all terminal functions will check this.

Terminal calls decide who is entitled; they do not send Ether. Decision and payment are different steps.

```
function release() external {
    require(msg.sender == payer, "only payer");
    require(s == State.Funded && block.timestamp < deadline, "bad state");
    s = State.Resolved;
    credit[payee] += amount;
}

function refund() external {
    require(msg.sender == payer, "only payer");
    require(s == State.Funded && block.timestamp >= deadline, "bad state");
    s = State.Resolved;
    credit[payer] += amount;
}

function resolve(bool toPayee) external {
    require(msg.sender == arbiter, "only arbiter");
    require(s == State.Funded, "bad state");
    s = State.Resolved;
    credit[toPayee ? payee : payer] += amount;
}
```

*Pull-over-Push and CEI on terminal paths.* The assignments to `credit[...]` implement Pull-over-Push: they *record* an entitlement instead of sending Ether now. This makes the outcome independent of receiver behavior—if the receiver's wallet reverts today, the credit persists and can be claimed later. CEI appears in the first write `s = State.Resolved`. Finalizing the state before crediting removes the possibility that any later interaction (direct or indirect) could change the branch. Requiring `s == State.Funded` ensures idempotence: the first terminal call wins, subsequent attempts fail. The deadline split across `release` and `refund` guarantees liveness: before the deadline the payer can close cooperatively; after the deadline the payer can unilaterally recover; at any time while funded the arbiter can decide. In each branch exactly one ledger entry increases by `amount`, enforcing the single-outcome rule.

All value leaves through a single entry. This is the only function that interacts with the outside world.

```
function withdraw() external {
    uint256 due = credit[msg.sender];
    require(due > 0, "nothing");
    credit[msg.sender] = 0;
    (bool ok, ) = msg.sender.call{value: due}("");
    require(ok, "transfer failed");
}
```

*Single withdrawal path and CEI inside it.* Centralizing transfers ensures that state transitions (`release`, `refund`, `resolve`) never call out, which keeps decision logic free of reentrancy surfaces. Inside `withdraw()`, the first write `credit[msg.sender] = 0` is the CEI effect: it prevents double collection if `msg.sender` re-enters before the call returns. If the external call fails, the final `require(ok)` reverts the function and *rolls back* the zeroing, so the entitlement remains intact. The result is predictable: each credited address can withdraw at most once; failed transfers do not lose money; and there is exactly one code path to audit for value movement.

*Why these lines implement the patterns.* Pull-over-Push is the deliberate shift from immediate sends to credit assignments in terminal functions. CEI on those terminal paths appears as state

resolution before any other effect, and because the terminals never interact, the paths are reentrancy-free by construction. The single withdrawal path gathers all external calls into one function and uses CEI again (clear, then call) to preclude double withdrawal and to make failure atomic. These choices encode three invariants the reviewer can check: conservation (credits and balance account for the full deposit), single outcome (only one party is credited by `amount`), and liveness (some call—`release`, `refund`, or `resolve`—is always available to end the deal, followed by `withdraw`).

*Summary.* Escrow turns business decisions into ledger entries and moves value later in one place. That is exactly what Pull-over-Push, CEI, and a single withdrawal path are designed to achieve; together they make the contract predictable to reason about and resilient to misbehaving receivers.

## 4 VOTING AND GOVERNANCE

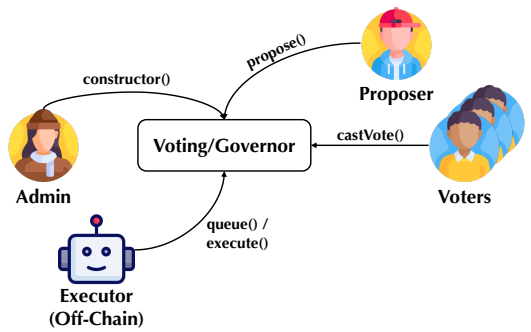


Fig. 3. Governor flow: propose, vote with snapshot, queue, then execute to an allowlisted target.

*What is a governor?* As shown by Figure 3, a governor turns collective intent into a scheduled, auditable change. A proposal fixes the payload up front; voters decide within a window using past-block weights; if thresholds pass, the proposal is queued with the earliest execution time; execution later calls only allowlisted targets. An off-chain executor typically listens to events and triggers `execute()` when legal.

*Patterns in this chapter.*

- **Snapshot voting:** read weight from a past block to prevent short-lived balance manipulation.
- **Timelock delay:** decide now, execute later so users can react.
- **Allowlisted execution:** restrict side effects to reviewed targets.
- **Event-driven trigger:** off-chain executor reacts to on-chain events.

*How it works: pattern-to-code.* A proposal records what will be called and when voting runs. Fixing these fields early is what makes later steps deterministic and auditable.

```
function propose(
    address target, uint256 value, bytes memory callData,
    uint256 startBlock, uint256 endBlock
) external returns (bytes32 id) {
    return id;
}
```

*Fix the intent early.* The parameters `target`, `value`, and `callData` are the exact payload the governor will attempt to execute if the proposal passes. Recording `startBlock` and `endBlock` establishes a closed voting window. Once stored against `id`, these fields should not change; immutability here is what allows reviewers to verify, in advance, the call that will later be made and to simulate its effects. In a concrete implementation, `id` is commonly a deterministic hash of these components so that anyone can recompute the identifier from the proposal contents.

Voting sums support and opposition using weights measured at a past block. The single-vote rule must also be enforced so that each address contributes at most once.

```
interface IVotes { function getPastVotes(address who, uint256 at) external view returns (uint256); }

function castVote(bytes32 id, bool support, IVotes token) external {
    uint256 w = token.getPastVotes(msg.sender, startBlock[id]);
    if (support) forVotes[id] += w; else againstVotes[id] += w;
}
```

*Snapshot voting.* The call to `getPastVotes(msg.sender, startBlock[id])` fixes voting power at the block where voting opened. That choice severs the manipulation path where someone borrows tokens just for the vote and returns them afterward. Because weight does not depend on current balances, transfers during the window cannot inflate a voter's influence. To complete the rule *one address, one vote*, the contract must also record that `msg.sender` has already voted on `id` and reject any second attempt; this prevents a single balance from being counted multiple times.

Scheduling and execution separate decision time from effect time and confine where effects may go.

```
mapping(bytes32 => uint256) eta;
uint256 minDelay;
mapping(address => bool) allowedTarget;

function queue(bytes32 id) external {
    require(forVotes[id] > againstVotes[id] && forVotes[id] >= quorum, "not passed");
    eta[id] = block.timestamp + minDelay;
}

function execute(bytes32 id) external {
    require(block.timestamp >= eta[id], "too early");
    require(allowedTarget[target[id]], "target not allowed");
    (bool ok, ) = target[id].call{value: value[id]}(callData[id]);
    require(ok, "exec failed");
}
```

*Quorum and majority.* In `queue()`, the conditions `forVotes[id] > againstVotes[id]` and `forVotes[id] >= quorum` encode two distinct gates. The first is the majority threshold: more weight must support than oppose. The second is participation: even a majority cannot pass with too little turnout. These numbers are visible before queueing, so observers can predict whether a proposal will advance.

*Timelock delay.* The assignment `eta[id] = now + minDelay` expresses *decide now, execute later*. This creates a visible window where users and integrators can react, and where off-chain monitors can stage checks or mitigation. The time guard in `execute()` enforces that the call cannot happen before the earliest execution time.

*Allowlisted execution.* The guard `allowedTarget[target[id]]` confines the set of callable addresses. This is a containment boundary: even if a malicious or mistaken payload is proposed, execution is

restricted to a reviewed set of contracts. Auditors can then focus on a small allowlist rather than the entire address space.

*Deterministic payload.* The call in `execute()` uses exactly the stored triple `(target, value, callData)`. Because these were fixed at proposal time, everyone can simulate the effect during the delay and check that it matches intent. The boolean `ok` is required to be true; otherwise the whole `execute()` reverts, preventing partial effects or silent failures.

*Event-driven trigger.* Although not shown in the snippet, a practical system emits events on `propose`, `castVote`, and `queue`. An off-chain agent subscribes to these events, waits until `eta`, verifies that the allowlist condition holds, and then calls `execute()`. The important property is that the trigger is observable and can be reproduced by any watcher.

*Why these lines implement the patterns.* Snapshot voting is realized by measuring weight at `startBlock[id]`, which blocks short-lived balance games during the window. The timelock is the explicit `eta` set at queue time and the time check in `execute()`, turning surprises into scheduled actions. Allowlisted execution is enforced by the `allowedTarget` predicate, narrowing side effects to reviewed targets. The event-driven trigger follows from the public schedule and emitted events: any observer can reconstruct the pipeline from proposal to execution and verify that the executed call equals the recorded payload.

*Summary.* Governance becomes a predictable pipeline: fix the payload, tally with snapshots, queue with a delay, and execute only to allowlisted targets. These patterns make the system auditable end-to-end and remove classes of surprises such as flash-loan voting swings, immediate behind-the-scenes changes, and arbitrary-call execution.

## 5 COMMIT–REVEAL AUCTION

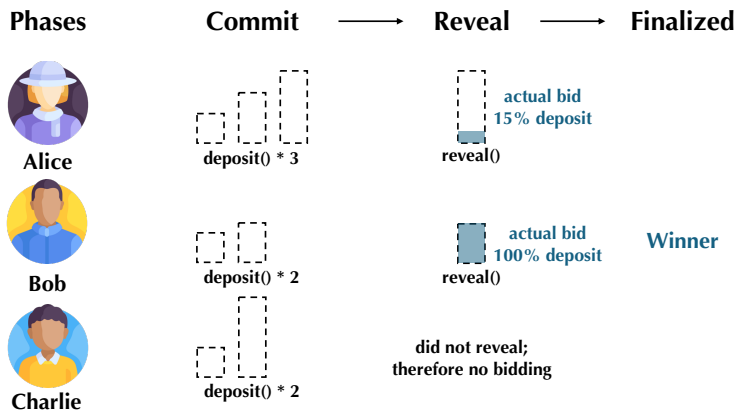


Fig. 4. Commit–reveal: hide bids first, then prove them; the auction uses it in the Reveal phase.

*What is the mechanism?* As shown by Figure 4, commit–reveal separates secrecy from verification. In Commit, a bidder publishes only a hash of their bid and salt with a deposit. In Reveal, the bidder shows `(bid, salt)`; the contract recomputes the hash and enforces a deposit rule.

*What is the auction?* As shown by Figure 5, the auction is a time-gated state machine: Commit → Reveal → Finalized. Bidders interact only through these phases; the seller later claims payment; value moves once at the end through a single withdrawal entry.

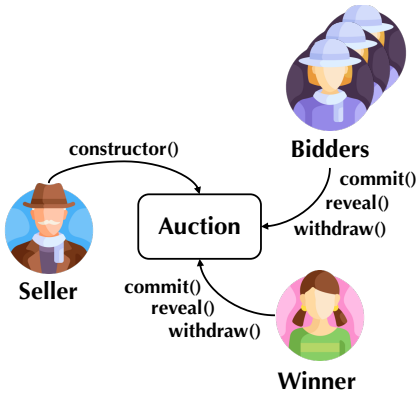


Fig. 5. Auction as a time-gated state machine with a single withdrawal point.

*Patterns in this chapter.*

- **State machine with time-based transitions:** encode phases and advance by timestamps.
- **Commit–reveal:** hide bids first, prove them later.
- **One-time commitment:** clear commitments after reveal to prevent reuse.
- **Single withdrawal path:** all transfers happen after finalization.

*How it works: pattern-to-code.* Program state captures phases and deadlines; functions check the phase; an internal helper advances when time crosses bounds. The following fragment fixes the legal phases up front so that each function can guard on what is allowed *now*.

```
enum Phase { Commit, Reveal, Finalized }
Phase phase; uint64 commitEnd; uint64 revealEnd;
address seller; address highestBidder; uint256 highestBid;

constructor(uint64 _commitEnd, uint64 _revealEnd, address _seller) {
    require(_commitEnd < _revealEnd, "bad bounds");
    commitEnd = _commitEnd; revealEnd = _revealEnd; seller = _seller;
    phase = Phase.Commit;
}

modifier inPhase(Phase p) { require(phase == p, "wrong phase"); _; }
function _advance() internal {
    if (phase == Phase.Commit && block.timestamp >= commitEnd) phase = Phase.Reveal;
    if (phase == Phase.Reveal && block.timestamp >= revealEnd) phase = Phase.Finalized;
}
```

*State machine with time-based transitions.* The variable `phase` encodes the current stage, and `commitEnd`/`revealEnd` set the time gates. The guard `inPhase(p)` makes legality a program property: attempts to reveal during Commit or to withdraw before Finalized are rejected by code, not social convention. The helper `_advance()` moves the system forward once a boundary is crossed, so a late caller who invokes any entry point still observes the correct phase. Together, these lines turn the timeline in Figure 5 into explicit state.

Commit and reveal implement secrecy-then-proof and enforce the deposit rule. The *commit* step binds a bidder to a specific value without revealing it; the *reveal* step proves that value later and checks that the bidder was able to fund it.

```
mapping(address => bytes32) commitments;
mapping(address => uint256) deposits;

function commit(bytes32 c) external payable inPhase(Phase.Commit) {
    require(commitments[msg.sender] == 0, "duplicate");
    require(msg.value > 0, "no deposit");
    commitments[msg.sender] = c;
    deposits[msg.sender] += msg.value;
}

function reveal(uint256 bid, bytes32 salt) external inPhase(Phase.Reveal) {
    bytes32 c = commitments[msg.sender];
    require(c != 0, "no commit");
    require(keccak256(abi.encode(bid, salt)) == c, "bad reveal");
    require(deposits[msg.sender] >= bid, "insufficient deposit");
    if (bid > highestBid) { highestBid = bid; highestBidder = msg.sender; }
    commitments[msg.sender] = 0;
}
```

*Commit–reveal.* The assignment `commitments[msg.sender] = c` records a *commitment* to one exact pair `(bid, salt)`. Only the hash `c` is stored, so the numeric bid remains hidden during Commit. In Reveal, the equality `keccak256(abi.encode(bid, salt)) == c` is the on-chain proof that the bidder is opening the same value they committed to earlier. Using `abi.encode` on a fixed-width pair `(uint256, bytes32)` makes the encoding unambiguous; the salt must be unpredictable so that small bids cannot be brute-forced from the hash.

*Deposit policy.* The line `require(deposits[msg.sender] >= bid)` enforces that the bidder can actually pay what they reveal. Because `deposits[msg.sender]` is funded in `commit()`, a bidder cannot reveal a number they did not back with Ether. This is what blocks the non-payable winner: the leader can only be one whose deposit covers their revealed bid.

*Leader update and ties.* The conditional `if (bid > highestBid)` updates both `highestBid` and `highestBidder`. Equal bids leave the earlier leader in place, which provides a deterministic tie-break: the first valid highest bid retains leadership unless someone strictly exceeds it.

*One-time commitment.* The final line of `reveal()` sets `commitments[msg.sender] = 0`. Clearing the commitment after a successful reveal prevents replay of the same proof in later calls, which could otherwise disturb the leader calculation or waste gas for others.

All transfers occur at the end. Finalization centralizes value movement in one entry point so that state transitions remain interaction-free and easy to audit.

```
function withdraw() external {
    _advance();
    require(phase == Phase.Finalized, "not finalized");

    if (msg.sender == seller) {
        require(highestBid > 0, "nothing");
        (bool ok, ) = seller.call{value: highestBid}("");
        require(ok, "seller transfer failed");
        return;
    }

    uint256 amt = deposits[msg.sender];
    require(amt > 0, "nothing");
```

```
    if (msg.sender == highestBidder) { amt -= highestBid; }
    deposits[msg.sender] = 0;
    (bool ok2, ) = msg.sender.call{value: amt}("");
    require(ok2, "transfer failed");
}
```

*Single withdrawal path.* The `require(phase == Phase.Finalized)` guard ensures that no Ether moves during Commit or Reveal; those phases manipulate only commitments and leader state. When Finalized, there are two exclusive cases. For the seller, the check `highestBid > 0` prevents a meaningless claim when no valid reveal occurred, and the transfer pays exactly the winning amount. For bidders, `amt = deposits[msg.sender]` returns their deposit; if the caller is the winner, the subtraction `amt -= highestBid` returns only the *change* beyond the winning price. Crucially, the first write in the bidder branch is `deposits[msg.sender] = 0`, which clears entitlement before the external call; any attempted re-entry observes zero and fails, so funds cannot be withdrawn twice. Centralizing every transfer here means the phase logic stays free of external calls, which keeps it robust against reentrancy and makes reviews shorter.

*Why these lines implement the patterns.* The state machine is the trio `phase`, `commitEnd`/`revealEnd`, and the gate `inPhase`, with `_advance()` ensuring progress when time passes. Commit–reveal is the pair that stores a hash in Commit and checks it in Reveal; secrecy first, proof later, and verifiable by anyone. One-time commitments are the clearing writes after reveal, which forbid reuse and keep leader updates well-defined. The single withdrawal path is the explicit choice to route all Ether through `withdraw()`, with CEI ordering inside it: clear internal accounting first, then call out. This separation is what makes the invariant easy to check: the seller receives exactly the highest bid once; the winner receives any change; everyone else receives their full deposit; and no external call can change who is the winner after Reveal.

*Summary.* The auction reads like a schedule: time controls legal actions; verification follows secrecy; value moves once through a single entry point.

## REFERENCES


[1] Coinbase. [n. d.]. Around The Block — Issue 3. https://www.coinbase.com/ru/learn/market-updates/around-the-block-issue-3. Accessed: 2025-10-15.

[2] Klint Finley. 2016. A $50 Million Hack Just Showed That the DAO Was All Too Human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/. Accessed: 2025-10-15.

[3] INCIBE-CERT. 2020. Cybercriminal steals $24 million from cryptocurrency service Harvest Finance. https://www.incibe.es/en/incibe-cert/publications/cybersecurity-highlights/cybercriminal-steals-24-million-cryptocurrency-service. Accessed: 2025-10-15.

[4] Mashable Staff. 2017. Ethereum Parity Bug. https://mashable.com/article/ethereum-parity-bug. Accessed: 2025-10-15.