Solidity Programming

A Standalone Mini-Course

BSc Blockchain Course

Why Does a Single Bug in Solidity Cost More Than a Thousand Bugs in Java?

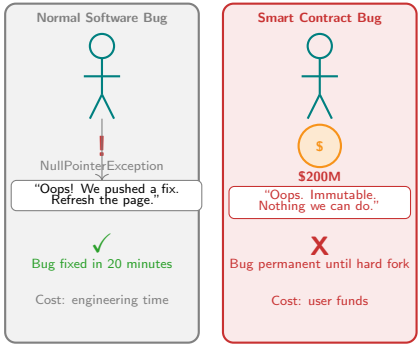
Every software engineer has shipped a bug. In most domains, bugs are embarrassing and fixable. In smart contracts, bugs are **permanent** and **expensive**.

Two properties make Solidity uniquely dangerous:

- **Immutability** – deployed bytecode cannot be edited. There is no "push fix to production." The only options are a proxy upgrade pattern (if pre-planned) or deploying a new contract and migrating users.
- **Financial stakes** – production contracts hold $10M–$1B of user funds. The Ronin Bridge held $624M when compromised. Wormhole held $326M.

What changes as a result:

- Security audit before mainnet (not after)
- Formal verification for critical arithmetic
- Bug bounty programmes (up to $10M rewards)
- Time-locked admin functions



The discipline of Solidity development is not about writing cleverly. It is about writing verifiably.

Solidity requires a pre-production mindset. Every line is a commitment. The difference between a typo in a Word document and a typo in a `transfer()` function is eight figures.

Sourc

Chainalysis 2023 Crypto Crime Report. Ronin $624M (March 2022), Wormhole $326M (Feb 2022), Poly Network $611M (Aug 2021).

What Would You Do If Your Code Drained Sixty Million Dollars?

These are not theoretical risks. Each case is a post-mortem of a real exploit where a discoverable code flaw drained user funds. None could be patched after the fact.

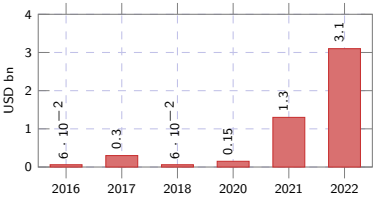
1. **The DAO Hack — $60M (June 2016).** Reentrancy in `splitDAO()`: balance decremented *after* external send. Fix required a hard fork of Ethereum mainnet. **Lesson:** CEI was not yet standard. Now it is the first rule taught.
2. **Parity Multisig Freeze — $300M (Nov 2017).** Library's `initWallet()` was left `public`; anyone could claim ownership and `kill()` it, permanently freezing all dependent wallets. **Lesson:** Visibility must default to `private`.
3. **Ronin Bridge Hack — $624M (March 2022).** Compromised validator keys; only 5-of-9 threshold controlled by attacker. **Lesson:** Contracts are only as secure as their off-chain key management.

Pattern across all three cases:

- The vulnerability was visible in the source code before the exploit
- No audit had been performed, or findings were not acted upon
- The fix in retrospect was trivially simple:

Case	Root Cause	Trivial Fix
DAO	Wrong order	CEI pattern
Parity	Wrong visibility	`private initWallet`
Ronin	Too few signers	9-of-15 keys

Cumulative DeFi exploit losses (USD bn):



Illustrative, based on published post-mortems.

Sourc

Rekt News (rekt.news) post-mortem database. Chainalysis Crypto Crime Report 2023. Trail of Bits vulnerability research.

What Are the Building Blocks of Every Solidity Contract?

A Solidity contract holds state, emits events, enforces access rules, and exposes functions. The type system determines how data is stored and how much it costs.

Value types (copied on assignment):

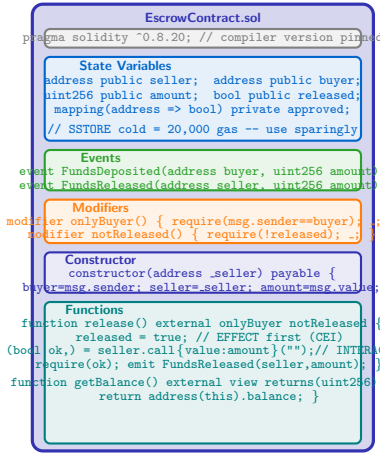
- `uint256`, `int256` – unsigned/signed integers, 256-bit
- `bool` – `true`/`false`
- `address` – 20-byte Ethereum address; `address payable` can receive ETH
- `bytes32` – fixed-size byte array; cheaper than `string` for short data
- `enum` – user-defined set of named constants

Reference types (need explicit data location):

- `mapping(K => V)` – hash map; keys not iterable; most common data structure
- `array` – dynamic arrays in storage can grow
- `struct` – bundle of named fields; packs into fewer slots if ordered
- `string`, `bytes` – variable-length; expensive in storage

Data location (reference types only):

- `storage` – persistent, on-chain, expensive to write
- `memory` – temporary, within a single call, cheap
- `calldata` – read-only input, cheapest for external params



A Solidity contract is a self-contained program: state variables persist on-chain, events create an audit trail, modifiers enforce preconditions, and functions define the public interface. Every component has a gas cost determined by its data location.

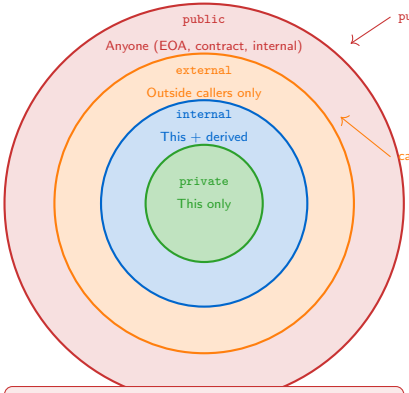
How Did One Public Function Freeze Three Hundred Million Dollars?

Solidity has four visibility levels for functions and state variables. Choosing the wrong one is the second most common class of vulnerability.

The four levels (most open to most restricted):

`public`	Callable by anyone: external accounts, other contracts, and from within the contract. State variables get a free getter. Use only for intentional public API.
`external`	Callable only from outside the contract. Cheaper than public for large array arguments because calldata is not copied into memory.
`internal`	Callable from this contract and any contract that inherits from it. Similar to `protected` in Java/C++. The right default for helpers.
`private`	Callable only from this contract. Not accessible in derived contracts. Apply by default; upgrade only when needed.

Security rule of thumb:
Start `private`. If a derived contract needs it, promote to `internal`. If the front-end needs it, promote to `external`. Never use `public` unless you mean "this is the public API."



Solidity does not enforce a safe default. The industry convention is to declare everything `private` or `internal` first and promote only when the interface demands it.

Sourc

Solidity Documentation — Visibility. Parity Multisig Wallet Hack Post-Mortem, November 2017.

Where Does Data Live Inside the EVM and What Does Each Location Cost?

In Solidity, **where** data lives determines its cost and lifetime. Reference types (`array`, `struct`, `mapping`) require an explicit data location.

The cost hierarchy:

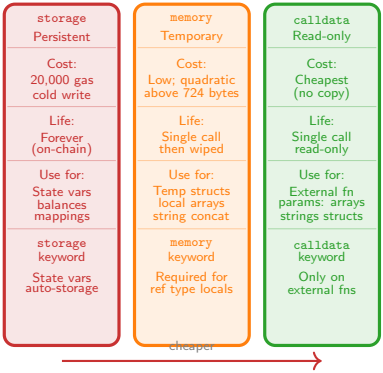
- **storage writes** are the most expensive EVM operation: 20,000 gas for a cold write (new slot), 2,900 gas for a warm write
- **memory** allocates fresh space each call; costs grow quadratically above 724 bytes but is still far cheaper than storage
- **calldata** is read-only input data that never enters EVM memory; the cheapest option for function arguments

A common gas-saving pattern:
Cache a storage variable in a memory local before a loop:

✗ `for(...) { total += balances[i]; }` – SLOAD every iteration

✓ `uint256[] memory b = balances;` then loop over `b[i]`

Calldata for external params:
`function f(uint256[] calldata arr)` avoids copying into memory. Use for large arrays passed to external functions.



The cost rule is simple: storage is forever (expensive), memory is per-call (cheap), calldata is read-only input (cheapest). Choose the cheapest location that satisfies the lifetime requirement.

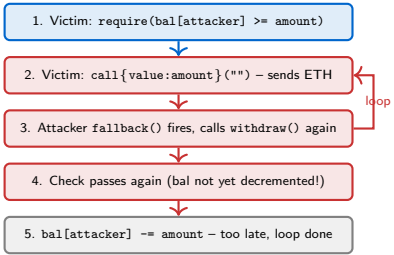
Sourc

Ethereum Yellow Paper (Appendix H: Gas Costs). EIP-2929: Increasing gas cost for state access opcodes (Berlin hardfork, 2021).

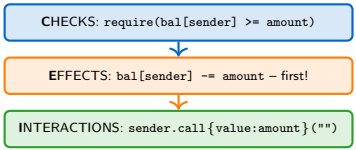
Why Must You Update the Ledger Before Sending the Money?

Reentrancy exploits the fact that a function can be interrupted by an external call before it finishes updating its own state.

Attack sequence (reentrancy):



CEI defence:



✓ Re-enter: balance already 0, require reverts.

Code comparison:

```
// VULNERABLE: interaction before effect
function withdraw(uint256 amount) external {
  require(balances[msg.sender] >= amount);
  (bool ok,) = msg.sender.call{value:  amount}("");
  require(ok);
  balances[msg.sender] -= amount; // too late!
}

// SAFE: CEI order
function withdraw(uint256 amount) external {
  require(balances[msg.sender] >= amount); // C
  balances[msg.sender] -= amount;       // E
  (bool ok,) = msg.sender.call{value:  amount}(""); // I
  require(ok);
}

// BELT + SUSPENDERS: ReentrancyGuard
import "@openzeppelin/.../ReentrancyGuard.sol";
contract SafeVault is ReentrancyGuard {
  function withdraw(uint256 amount)
    external nonReentrant { // plus CEI
  }
}
```

Sourc

Which Five Vulnerability Classes Have Cost Billions of Dollars?

The five most exploited vulnerability classes:

Vulnerability	Example	Mitigation	Sev.
Reentrancy	DAO 2016 $60M; Cream $25M	CEI pattern; `ReentrancyGuard`	Critical
Integer Overflow	BEC token 2018; batch overflow	Solidity 0.8 auto-reverts; `SafeMath`	Critical
Access Control	Parity 2017 $300M; Poly Network	`onlyOwner`; role AC; minimal visibility	High
Oracle Manip.	Mango $117M; Beanstalk $182M	TWAP oracles; multiple sources; circuit breakers	High
Flash Loan	PancakeBunny $45M; Value DeFi	TWAP prices; multi-block oracle; slippage limits	High

How vulnerabilities evolve over time:

- **Reentrancy (pre-2020):** Now well-understood; most audits catch it immediately. Still appears in code written without security awareness.
- **Overflow (pre-Solidity 0.8):** Solidity 0.8+ added built-in overflow checking. Legacy contracts on 0.7 still need `SafeMath`. Check your `pragma`.
- **Oracle manipulation (2020–present):** The dominant attack vector in DeFi. Any protocol that prices assets using a single on-chain AMM pool is vulnerable to same-block flash loan manipulation.
- **Flash loan attacks:** Not a vulnerability in flash loans themselves. Flash loans amplify existing vulnerabilities in oracle design or governance mechanisms.

The pattern: each generation learns the previous generation's mistakes. The next attack always comes from a category that feels too exotic to worry about.

Security is not a feature to add at the end. It is a design constraint present from the first line of code. Retrofitting security into a deployed immutable contract is, by definition, impossible.

Sourc

SWC Registry (SmartContractSecurity.github.io). Rekt News post-mortem database. Trail of Bits "Building Secure Smart Contracts" 2023.

How Do Professional Teams Test Contracts Before Risking Real Money?

Two frameworks dominate professional Solidity development.
Choosing one is a team preference, not a right/wrong decision.

Hardhat (JavaScript/TypeScript ecosystem):

- Test in JS/TS with Mocha/Chai or Waffle
- Excellent plugin ecosystem (`hardhat-etherscan`, `hardhat-gas-reporter`)
- Familiar to web developers; easy to integrate with front-ends
- `console.log()` inside Solidity during tests

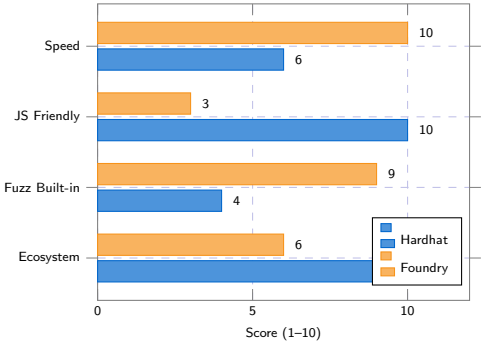
Foundry (Rust-based, tests in Solidity):

- Fastest compilation and test execution
- Tests written in Solidity – no context switch
- Built-in fuzzer (`forge fuzz`) and formal verification hooks
- `cast` CLI for interacting with mainnet/testnet

Static analysis tools:

- `Slither` (Trail of Bits) – fast, 70+ detectors, CI-friendly
- `Mythril` – symbolic execution, finds deep logic bugs
- `Echidna` – property-based fuzzing, writes contracts as invariants

Framework comparison (score 1–10):



CI pipeline recommendation:

- Run `slither .` on every pull request (fast, catches common issues)
- Run `forge test` for unit tests in Solidity
- Run `npx hardhat test` for integration and front-end scripts

Professional teams run both: Hardhat for integration tests and front-end scripts, Foundry for unit tests and fuzzing. Add Slither to CI so every pull request is automatically scanned.

Six Things Every Solidity Developer Must Remember

1. **Immutability is the constraint that makes everything else matter.** A deployed contract cannot be patched. There is no "push fix to production." Front-load all quality work before the first transaction is signed.

2. **Checks-Effects-Interactions is not optional.** Every function that moves value must update state *before* any external call. The DAO hack in 2016 proved the cost of skipping this order: **$**60M and a chain split.

3. **Default all visibility to `private`.** Promote only when the public interface explicitly requires it. The Parity multisig freeze – **$**300M permanently locked – happened because one initialisation function was accidentally left `public`.

4. **Storage writes are your most expensive operation.** A cold `SSTORE` costs 20,000 gas. Cache state variables in `memory` before loops, use `calldata` for external parameters, and pack smaller types into the same 32-byte slot.

5. **Test unhappy paths, not just happy paths.** Attackers only need one code path that fails to revert when it should. Every `require` condition deserves a dedicated failing test. Fuzz arithmetic with Foundry or Echidna.

6. **Professional audit is not optional for mainnet.** A **$**100k security audit that prevents a **$**10M exploit delivers a 100x return. Run Slither on every pull request; submit to a formal audit before any contract holds significant user funds.

Sourc

Chainalysis 2023 Crypto Crime Report. OpenZeppelin "Writing Secure Smart Contracts." Trail of Bits "Building Secure Smart Contracts" (2023).

Your Challenge

You have studied reentrancy, visibility, storage costs, the CEI pattern, and the testing toolchain. Now apply it. Below is a simplified escrow contract with four deliberate flaws.

Your Challenge

```
contract Escrow {
  mapping(address => uint256) public balances;
  function deposit() external payable { balances[msg.sender] += msg.value; }
  function withdraw(uint256 amt) public {
    (bool ok,) = msg.sender.call{value: amt}(""); require(ok);
    balances[msg.sender] -= amt;
  }
  function initOwner(address o) public { owner = o; }
  uint256 fee; uint8 tier; bool active; uint64 ts;
}
```

1. **Identify the reentrancy flaw.** Which two lines are in the wrong order in `withdraw()`? Write the corrected version following the CEI pattern.
2. **Fix the visibility bug.** Which function must never be `public` after deployment? What visibility should it have, and why?
3. **Fix the storage layout.** The four state variables `fee`, `tier`, `active`, `ts` waste slots. Reorder them to pack into a single 32-byte slot.
4. **Write one test.** Using Foundry or Hardhat, write a test that proves `withdraw()` reverts when the caller has insufficient balance.
5. **Run Slither.** What command would you run on this file? What output would you expect for the reentrancy detector?

Sourc

OpenZeppelin "Writing Secure Smart Contracts." Trail of Bits "Building Secure Smart Contracts" (2023). ConsenSys Diligence Best Practices.