SOLIDITY SMART CONTRACTS

From Zero to Production

A Developer's Guide to Building, Securing, and Shipping Smart Contracts with Foundry

```
01  // SPDX-License-Identifier: MIT
02  pragma solidity ^0.8.20;
03
04  contract Counter {
05      uint256 public count;
06
07      event Increment(address indexed by, uint256 newCount);
08
09      constructor(uint256 _initial) {
10          count = _initial;
11      }
12
13      function increment() external {
14          unchecked {
15              count += 1;
16          }
17          emit Increment(msg.sender, count);
18      }
19
20      function getCount() external view
21        returns (uint256) {
22          return count;
23      }
24  }
```

MD Ayaan Siddiqui

</> | Solidity ^0.8.20 | Foundry-First | Free Edition

About This Book

This is a code-first guide to Solidity smart contract development, written for developers who already know how to program but are new to Ethereum, the EVM, and Web3 tooling.

The approach here is deliberately practical: every concept is introduced through complete, compilable code before the theory behind it is explained. You will not find long stretches of abstract discussion disconnected from a working contract. If a pattern is worth learning, it is worth seeing in full, testing with Foundry, and understanding well enough to modify with confidence.

The book is organized to move the way a real project moves: fundamentals, then design patterns, then token standards, then DeFi mechanics, then the security and gas-optimization work that separates a contract that runs from one that survives production. Foundry is treated as the primary toolchain throughout, since it is the standard most serious Solidity teams have converged on.

This book is completely free. It exists specifically to help newcomers and aspiring blockchain developers — people with no budget for a paid course or bootcamp — get a genuine, practical start in Solidity and smart contract development. If it helps even one person land their first Web3 role, it's done its job.

This book was written entirely by Claude (Anthropic's AI), working from the author's outline, structure, and direction, with the author reviewing and guiding every chapter.

> **A NOTE ON SOURCES.** Every technical claim, standard, code pattern, and historical fact in this book is drawn from publicly available sources: official documentation (Solidity, Foundry, Ethereum, OpenZeppelin), published EIP/ERC specifications, and public reporting on real events referenced throughout. No private, proprietary, confidential, or non-public information of any kind was used in writing this book.

Table of Contents

How to Use This Book

A few conventions are used consistently across every chapter so you always know what you are looking at.

Tooling and Versions

Every example in this book was written and tested against these versions:

Tool	Version Used In This Book
Solidity	^0.8.20
Foundry	latest stable (forge, cast, anvil)
OpenZeppelin Contracts	^5.x
Node.js (tooling scripts only)	18+

Callout Box Legend

Four kinds of callout boxes appear throughout the book. Each has a fixed meaning, so you can scan a chapter and know exactly what to pay attention to:

> ⚠ **SECURITY** A security-critical point — a vulnerability class, an attack vector, or a rule that protects funds.

> ⛽ **GAS TIP** A concrete way to reduce gas cost, usually shown with a before/after comparison.

> ⚡ **PITFALL** A common mistake developers make with this pattern, and how to avoid it.

> ℹ **INFO** Extra context or a side-note that is useful but not essential to keep moving.

Every chapter closes with a short "Key Takeaways" box summarizing what was covered.

CHAPTER 1

What Are Smart Contracts

1.1 A Brief History

"Smart contract" is an older term than most people expect — it predates Bitcoin by fourteen years, and predates Ethereum by nearly two decades. The table below is the fast version of that history; you do not need to memorize it, but the shape of it explains why Solidity looks the way it does.

Year	Milestone
1994	Nick Szabo describes the concept of a "smart contract" — self-executing digital agreements, illustrated with a vending machine analogy.
2008–09	The Bitcoin whitepaper and network launch. Scripting exists but is deliberately limited — not Turing-complete.
2013	Vitalik Buterin publishes the Ethereum whitepaper, proposing a blockchain with a built-in, Turing-complete programming environment.

2015	Ethereum mainnet launches. Smart contracts become programmable, general-purpose, and composable for the first time.
2016	The DAO hack: a reentrancy vulnerability drains roughly a third of a major fund, reshaping how the industry thinks about contract security.
2017	The ERC-20 token standard drives a token-issuance boom, onboarding a wave of new developers to Solidity.
2020	"DeFi Summer": AMMs, lending protocols, and yield farming turn smart contracts into real financial infrastructure.
2021–22	Rollup-centric scaling matures. Optimistic and ZK rollups reach mainnet, extending the EVM beyond Ethereum L1.
Today	Foundry-based tooling and account abstraction (ERC-4337, and increasingly EIP-7702) are the production standard this book teaches.

Figure 1.1 — Smart contracts, from Szabo's 1994 concept to today's production stack

1.2 Szabo's Original Idea

Nick Szabo, a computer scientist and legal scholar, wanted a way to embed the logic of a contract directly into a system, so that performance of the agreement did not depend on trusting the other party — or a court — to enforce it. His standard illustration was a vending machine: you insert the correct amount, and the machine mechanically dispenses the item. No cashier, no invoice, no possibility of the machine changing its mind. The rules are enforced by the mechanism itself.

That is the core idea a smart contract borrows: encode the rules once, and let the execution environment enforce them automatically, for everyone, every time, without a trusted intermediary in the loop. Szabo described this decades before there was a blockchain capable of running it — Ethereum is, in large part, the vending machine he was imagining, generalized into a programmable computer.

1.3 What a Smart Contract Actually Is, Today

Set aside the history for a moment. In practical, 2026 terms, a smart contract is:

- Code and persistent state, deployed at a fixed address on a blockchain
- Executed deterministically — the same inputs always produce the same outputs, on every node that verifies it
- Publicly verifiable — the bytecode (and usually the source, once verified) is visible to anyone
- Immutable by default once deployed — the code that goes live is the code that runs, unless it was deliberately built to be upgradeable

> **i INFO** Immutability is a default, not a law of physics. Chapter 3 covers proxy and UUPS patterns, which let a team change contract logic later without changing the contract's address. Whether to use them is a real design decision with real tradeoffs, not a free win.

1.4 The Execution Environment: The EVM

Every Ethereum-compatible smart contract runs inside the Ethereum Virtual Machine, or EVM — a sandboxed, stack-based virtual machine that every full node runs identically. Solidity does not execute directly; it compiles down to EVM bytecode, a sequence of low-level opcodes (PUSH, SSTORE, CALL, and so on) that the EVM interprets one instruction at a time.

You will rarely write opcodes by hand, but understanding that they exist — and that each one has a cost — is what makes the next section make sense.

Figure 1.2 — From Solidity source to executed state change

1.5 Gas, Explained

If code ran for free, anyone could write an infinite loop and stall every node on the network forever. Gas exists to solve exactly that problem: every operation an EVM contract performs has a fixed gas cost, and a transaction must pay for the total gas it consumes. This turns unbounded computation into a bounded, priced resource.

Four terms show up constantly, and it is worth being precise about each one:

- Gas limit — the maximum gas you authorize a transaction to consume. Set it too low and the transaction reverts (and you still pay for the gas already used).
- Gas used — the gas the transaction actually consumed. Always less than or equal to the gas limit.
- Base fee — a network-determined price per unit of gas (in gwei) that adjusts with demand, and is burned rather than paid to anyone.
- Priority fee (tip) — an additional amount per unit of gas that you offer the validator, on top of the base fee.

Put together: suppose a transaction has a gas limit of 100,000, but only uses 21,000 gas — the fixed cost of a plain ETH transfer. If the base fee at that moment is 20 gwei and you add a 2 gwei tip, your effective gas price is 22 gwei. Your total cost is 21,000 × 22 gwei = 462,000 gwei, or 0.000462 ETH. You are only charged for gas used, not the full gas limit — the difference is never taken from you in the first place.

(The gwei figures above are an illustrative example, not a claim about any specific moment in time — real network fees move constantly with demand.)

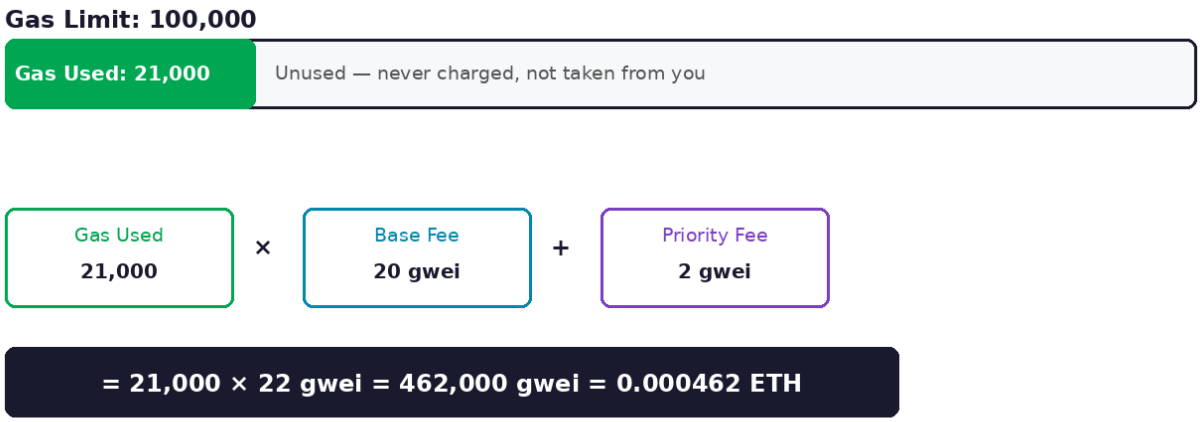

Figure 1.3 — Gas limit vs. gas used, and how total transaction cost is calculated

> **i INFO** Gas is a cost you can actively design against. Chapter 8 is dedicated entirely to gas optimization — storage packing, custom errors, immutable variables, and more — each shown with a measured before/after gas cost.

1.6 Comparing Smart Contract Platforms

Solidity and the EVM are not the only way to build programmable blockchain applications. The table below compares the three environments you are most likely to encounter as a working developer.

	Ethereum L1	Solana	Ethereum L2s (Rollups)
Execution model	EVM — stack-based, sequential state transitions	Sealevel — parallelized execution across accounts	Mostly EVM-equivalent; executes off-chain, posts data/proofs back to L1
Primary language(s)	Solidity, Vyper	Rust, C/C++ (commonly via Anchor)	Solidity (same as L1 — this is the point of EVM-equivalence)
Fee model	Gas priced in gwei; base fee + priority tip (EIP-1559), fluctuates with demand	Low, fairly predictable fees priced in compute units	Much lower than L1; cost driven mainly by L1 data-posting cost
Finality	Full finality in a few epochs (~12–15 min); soft confirmation much faster	Fast — typically well under a second per confirmation	Optimistic rollups: withdrawal challenge period (often ~7 days). ZK-rollups: faster, once a validity proof is verified on L1
Security model	Secured directly by Ethereum's own validator set	Secured by its own independent validator set	Inherits Ethereum L1 security for finality/data availability in most designs
Ecosystem maturity	Largest DeFi TVL, most audited tooling and libraries	Rapidly growing, especially consumer and high-throughput apps	Inherits most of Ethereum's tooling; fast-

			growing DeFi migration from L1

1.7 Why This Book Focuses on Solidity and the EVM

EVM-equivalence is now the dominant standard across the ecosystem — not just Ethereum L1, but the great majority of L2 rollups, and a growing number of otherwise-independent chains that chose to stay EVM-compatible for exactly this reason. Solidity skill is the most transferable skill in the space: a contract you understand on Ethereum L1 is, with minor adjustments, a contract you can deploy on Arbitrum, Optimism, Base, or any of a dozen other EVM chains. That transferability, plus the deepest tooling and audit ecosystem in the industry, is why this book teaches Solidity first.

□ TRY IT YOURSELF — Do the Gas Math

A transaction has a gas limit of 150,000 and actually uses 84,000 gas. The base fee is 15 gwei and you set a 1.5 gwei priority tip. What is the effective gas price, and what is the total transaction cost in ETH? (Work it out before checking — the method is exactly the one used in section 1.5.)

Hint: Effective gas price = base fee + priority tip. Total cost = gas used × effective gas price, converted from gwei to ETH by dividing by 1,000,000,000.

KEY TAKEAWAYS

- "Smart contract" as a concept predates blockchains by decades — Szabo's core idea is rules enforced by a mechanism, not a trusted party.
- A smart contract today is deployed code with persistent state, executed deterministically and immutably by default.
- The EVM executes compiled bytecode one opcode at a time; every opcode has a fixed gas cost.
- Gas limit, gas used, base fee, and priority fee together determine what a transaction actually costs.
- Ethereum, Solana, and L2 rollups differ in execution model, fees, and finality — but EVM-equivalence makes Solidity the most transferable skill across most of them.

□ DID YOU KNOW?

Nick Szabo never framed the smart contract concept around any grand crypto ambition — he described it as a natural extension of thinking about contract law in computational terms. Ironically, Szabo himself has long been the subject of speculation that he might be Satoshi Nakamoto, Bitcoin's pseudonymous creator — a claim he has repeatedly and directly denied.

CHAPTER 2

Solidity Fundamentals

2.1 Your First Contract

Here is a complete, compilable contract. It is small enough to hold in your head, but it already uses every concept this chapter covers: a state variable, a constructor, an event, a custom error, a modifier, and three

functions with different visibility and mutability. Read it once top to bottom before the explanations below break it apart piece by piece.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract SimpleCounter {
    address public immutable owner;
    uint256 private count;

    event CountChanged(uint256 newCount, address changedBy);

    error NotOwner(address caller);

    modifier onlyOwner() {
        if (msg.sender != owner) revert NotOwner(msg.sender);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function increment() external {
        count += 1;
        emit CountChanged(count, msg.sender);
    }

    function reset() external onlyOwner {
        count = 0;
        emit CountChanged(count, msg.sender);
    }

    function getCount() external view returns (uint256) {
        return count;
    }
}
```

Nothing here is decorative. `owner` is set once, in the constructor, and never again — that is what `immutable` buys you. `onlyOwner` is a reusable guard. `NotOwner` is a custom error, not a string. `CountChanged` is how the outside world (a frontend, an indexer) finds out state changed, without ever reading storage directly. The rest of this chapter is these six ideas, one at a time, in more depth.

2.2 Value Types

Value types are copied whenever they are assigned or passed to a function — you always get an independent copy, never a reference. These are the ones you will use constantly:

Type	Size	Holds	Default
bool	1 byte	true / false	false
uint256 (uint)	32 bytes	0 to 2^256 − 1	0
int256 (int)	32 bytes	signed integer	0
address	20 bytes	an account or contract address	address(0)
address payable	20 bytes	address that can receive ETH via .transfer/.send	address(0)

bytes32	32 bytes	fixed-size raw byte data	0x00...00

```
bool public isActive = true;
uint256 public totalSupply = 1_000_000;
int256 public temperatureDelta = -12;
address public admin = msg.sender;
address payable public treasury;
bytes32 public merkleRoot;
```

> **i INFO** Solidity has no floating-point type. Percentages, prices, and ratios are handled with integer math and a fixed number of decimal places (this is exactly what ERC-20's `decimals` is for) — a pattern you'll use constantly starting in Chapter 4.

2.3 Reference Types

Structs, arrays, and mappings do not get copied the way value types do — how they behave depends entirely on where they live, which is exactly the subject of the next section.

```
struct User {
    string name;
    uint256 balance;
    bool isVerified;
}

mapping(address => User) public users;
uint256[] public recentAmounts;
address[3] public founders;

function registerUser(string calldata name) external {
    users[msg.sender] = User(name, 0, false);
}
```

A `mapping` is really a hash table that exists only in storage — you cannot iterate over it, and every possible key already "exists" and returns the type's default value until something is written to it. This is why the `users` mapping above never needs to be initialized.

2.4 Storage, Memory, and Calldata

Every reference type in Solidity lives in exactly one of three places, and the EVM prices each one very differently.

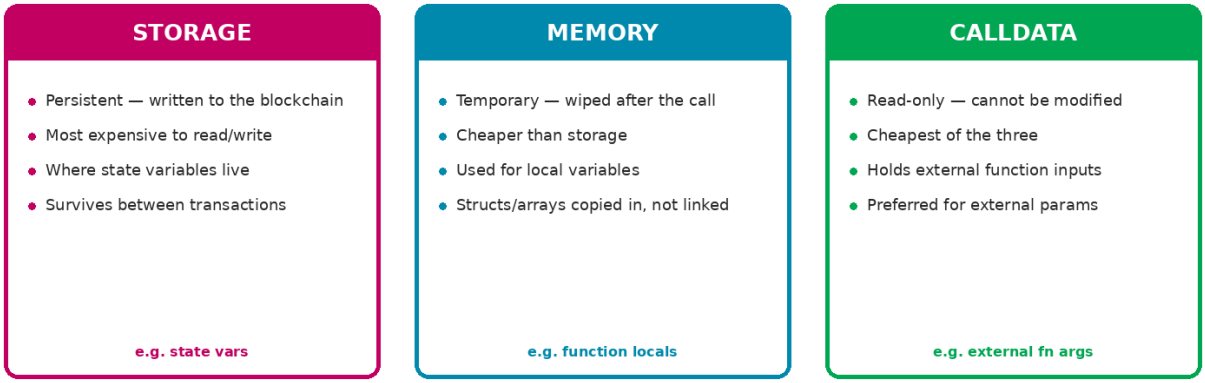

Figure 2.1 — The three data locations, and what each is for

The gotcha almost every new Solidity developer hits at least once:

```
function badUpdate(uint256 userIndex) external {
    // BUG: this COPIES the struct into memory.
    // Changing it here does not touch storage at all.
    User memory u = usersList[userIndex];
    u.balance += 100;
}

function goodUpdate(uint256 userIndex) external {
    // storage reference — this actually writes to state
    User storage u = usersList[userIndex];
    u.balance += 100;
}
```

⚡ **PITFALL** Declaring a struct or array as `memory` inside a function that reads from storage creates a full copy. Modifying the copy does nothing to the actual stored data — the function will compile, run without reverting, and silently do nothing useful. Declare it `storage` when you intend to modify the original.

2.5 Functions

Every function signature makes two independent decisions: who can call it, and what it is allowed to do to state.

Visibility	Callable by
public	Anyone — external accounts, other contracts, and this contract
external	Anyone, but only via an external call (cheaper for large inputs than public)
internal	This contract and any contract that inherits from it
private	Only this exact contract — not even child contracts
Modifier	**Meaning**
view	Reads state but does not modify it
pure	Touches no state at all — reads nothing, writes nothing
payable	Can receive ETH along with the call
(none)	Can freely read and write state

```
function viewBalance(address account) external view returns (uint256) {
    return users[account].balance;
}

function calculateFee(uint256 amount) external pure returns (uint256) {
    return amount / 100;
}

function deposit() external payable {
    users[msg.sender].balance += msg.value;
}

function _internalHelper() internal {
    // callable only from within this contract or derived contracts
}
```

⛽ **GAS TIP** `external` is cheaper than `public` for functions that are only ever called from outside the contract, because `external` arguments are read directly from calldata instead of being copied into memory first. Default to `external` unless the function is also called internally.

2.6 Modifiers

A modifier wraps a function with a check that runs before (and optionally after) the function body. The `_;` marks exactly where the wrapped function's code executes. Modifiers can take their own parameters, just like functions:

```
modifier minAmount(uint256 amount, uint256 minimum) {
    require(amount >= minimum, "Amount below minimum");
    _;
}

function withdraw(uint256 amount) external minAmount(amount, 0.01 ether) {
    // withdrawal logic runs only after the modifier's check passes
}
```

2.7 Events

Storage is expensive to write and expensive to read back from within a contract — but events are cheap to emit and are built specifically to be read by things outside the chain: a frontend listening for updates, a subgraph indexing transfers, a block explorer. Marking a parameter `indexed` (up to three per event) lets off-chain tools filter for it efficiently.

```
event Transfer(address indexed from, address indexed to, uint256 value);

function transfer(address to, uint256 amount) external {
    users[msg.sender].balance -= amount;
    users[to].balance += amount;
    emit Transfer(msg.sender, to, amount);
}
```

2.8 Custom Errors

Before Solidity 0.8.4, the idiomatic way to fail was `require(condition, "some string")`. Custom errors do the same job for meaningfully less gas, and they can carry structured data instead of just a message:

```
// Custom error — cheaper than a require string, and can carry data
error InsufficientBalance(uint256 available, uint256 required);

function withdrawStrict(uint256 amount) external {
    uint256 bal = users[msg.sender].balance;
    if (bal < amount) {
        revert InsufficientBalance(bal, amount);
    }
    users[msg.sender].balance = bal - amount;
}
```

GAS TIP A require string is stored in the deployed bytecode in full, every time it appears, and costs gas to return on revert. A custom error's identifier is encoded as a compact 4-byte selector — smaller to deploy, and cheaper to revert with. Prefer custom errors over require strings everywhere in this book from here on.

□ TRY IT YOURSELF — Extend SimpleCounter

Add a new function `decrement()` to the SimpleCounter contract from section 2.1. It should be callable by anyone (not just the owner), it must revert with a custom error — not a require string — if count is already 0, and it should emit CountChanged just like the other functions.

Hint: You'll need a new error, e.g. `error CountIsZero();`, and an `if` check at the top of the function before you subtract.

KEY TAKEAWAYS

- Value types (bool, uint, address, bytes32...) are always copied; reference types (structs, arrays, mappings) depend on where they're declared.
- Storage is persistent and expensive; memory is temporary and cheaper; calldata is read-only and cheapest — use calldata for external function parameters by default.
- Assigning a storage struct to a memory variable copies it — edits to the copy never reach the chain. Use `storage` when you mean to mutate the original.
- Function signatures combine visibility (who can call it) with mutability (what it can do to state) — external + view/pure whenever possible is the cheapest combination.

- Custom errors replace require strings in every example going forward: same safety, less gas, and they can carry structured data.

□ DID YOU KNOW?

Solidity's original design is credited to Gavin Wood, one of Ethereum's founding developers, who also wrote the Ethereum Yellow Paper — the formal specification defining exactly how the EVM executes every opcode. Wood later went on to found Polkadot, a separate blockchain built around connecting many independent chains together.

CHAPTER 3

Smart Contract Design Patterns

The problems in this chapter — who's allowed to call what, how to pause a contract in an emergency, how to stop a reentrancy attack, how to deploy many copies of a contract, how to upgrade logic after deployment — have all been solved already, publicly, and audited by thousands of eyes. OpenZeppelin's implementations are the industry default for a reason. The goal of this chapter is not to talk you into writing your own; it's to make sure you understand what each pattern is actually doing, so that using the audited version is a decision, not a superstition.

3.1 Ownable — Basic Access Control

The simplest access-control pattern there is: one address, the owner, is allowed to call certain functions, and nobody else is.

```
contract Ownable {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    error NotOwner(address caller);
    error ZeroAddress();

    constructor() {
        _owner = msg.sender;
```

```
        emit OwnershipTransferred(address(0), msg.sender);
    }

    modifier onlyOwner() {
        if (msg.sender != _owner) revert NotOwner(msg.sender);
        _;
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function transferOwnership(address newOwner) external onlyOwner {
        if (newOwner == address(0)) revert ZeroAddress();
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

> **i INFO** This is exactly the pattern `onlyOwner` in Chapter 2's SimpleCounter was hand-rolling. OpenZeppelin's real `Ownable` adds a two-step `transferOwnership` + `acceptOwnership` flow in most modern versions, specifically so you cannot accidentally lock yourself out by mistyping an address.

3.2 Pausable — The Circuit Breaker

A pause switch you hope you never need, but want available the day you do — most often flipped the moment an audit or a monitoring system flags something wrong mid-incident.

```
contract Pausable {
    bool private _paused;

    event Paused(address account);
    event Unpaused(address account);

    error ContractPaused();
    error ContractNotPaused();

    modifier whenNotPaused() {
        if (_paused) revert ContractPaused();
        _;
    }

    function _pause() internal {
        _paused = true;
        emit Paused(msg.sender);
    }

    function _unpause() internal {
        _paused = false;
        emit Unpaused(msg.sender);
    }

    function paused() public view returns (bool) {
        return _paused;
    }
}
```

In practice, `whenNotPaused` is added as a modifier to whichever functions move funds or change critical state — deposits and withdrawals, typically — while view functions stay callable even while paused.

3.3 ReentrancyGuard — Stopping the Attack That Started It All

This is the exact vulnerability class behind the 2016 DAO hack from Chapter 1. Here is the bug, in its smallest possible form:

```
// VULNERABLE — do not deploy this. Shown to illustrate the bug.
contract VulnerableVault {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 amount = balances[msg.sender];
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
        balances[msg.sender] = 0;   // state updated AFTER the external call
    }
}
```

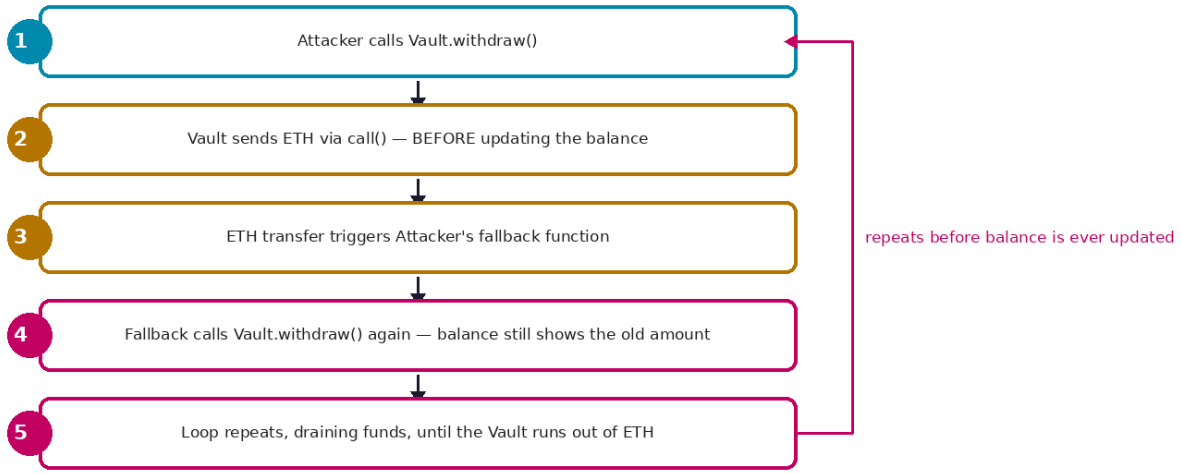

Figure 3.1 — How a reentrancy attack drains a vault before its balance is ever updated

> ⚠ **SECURITY** The root cause is ordering, not the external call itself. `withdraw` sends ETH before setting the caller's balance to zero. If the recipient is a contract, sending it ETH hands it execution — and its fallback function can call `withdraw` again, and again, each time seeing the old, un-zeroed balance.

Two independent fixes, and production code should use both:

- Checks-Effects-Interactions — update all state before making any external call, not after
- A reentrancy guard — a lock that makes any nested call into the same function revert outright

```
contract ReentrancyGuard {
    uint256 private constant NOT_ENTERED = 1;
    uint256 private constant ENTERED = 2;
    uint256 private status = NOT_ENTERED;

    error ReentrantCall();

    modifier nonReentrant() {
        if (status == ENTERED) revert ReentrantCall();
        status = ENTERED;
```

```
        _;
        status = NOT_ENTERED;
    }
}

contract SafeVault is ReentrancyGuard {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external nonReentrant {
        uint256 amount = balances[msg.sender];
        balances[msg.sender] = 0;   // Checks-Effects-Interactions: update state
FIRST
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

3.4 Factory Pattern

When your application needs to deploy many identical (or near-identical) contract instances — one token per project, one vault per user — a factory contract deploys them on demand with `new`, and keeps a registry of what it has created.

```
contract SimpleToken {
    string public name;
    address public creator;

    constructor(string memory _name, address _creator) {
        name = _name;
        creator = _creator;
    }
}

contract TokenFactory {
    address[] public deployedTokens;

    event TokenCreated(address indexed tokenAddress, string name, address indexed
creator);

    function createToken(string calldata name) external returns (address) {
        SimpleToken token = new SimpleToken(name, msg.sender);
        deployedTokens.push(address(token));
        emit TokenCreated(address(token), name, msg.sender);
        return address(token);
    }

    function totalTokens() external view returns (uint256) {
        return deployedTokens.length;
    }
}
```

GAS TIP For high-volume factories deploying many small contracts, look at EIP-1167 minimal proxy clones (`Clones.sol` in OpenZeppelin) once you reach Chapter 9 — they deploy a tiny redirect contract instead of the full bytecode every time, at a fraction of the gas cost.

3.5 Proxy Patterns and Upgradeability

Chapter 1 called immutability a default, not a law of physics. This is how teams opt out of it: split a contract into a Proxy (a fixed address, holding all the storage, that never changes) and an Implementation (the logic, which can be swapped for a new version later).

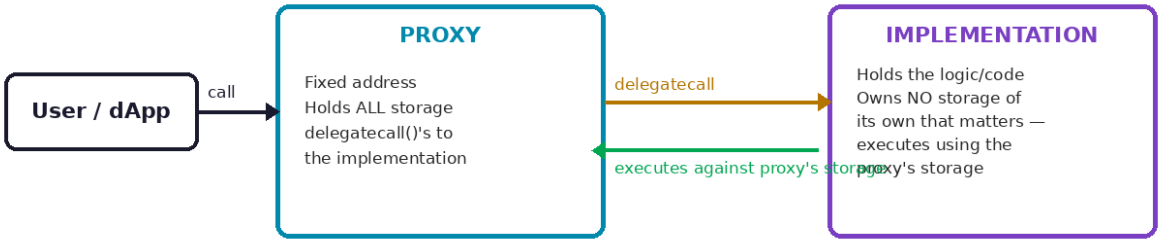

Figure 3.2 — A user always calls the Proxy; the Proxy executes the Implementation's code against its own storage

The mechanism that makes this work is `delegatecall` — it runs another contract's code, but every SLOAD and SSTORE it performs reads and writes the caller's storage, not its own. That is precisely why the Implementation contract in the diagram above is described as owning no storage that matters: whatever it reads or writes happens inside the Proxy.

```
// Simplified for illustration. In production, use OpenZeppelin's audited
// proxy implementations (ERC1967Proxy, UUPSUpgradeable) — never hand-roll this.
contract MinimalProxy {
    // EIP-1967: a pseudo-random slot, chosen specifically so it can never
    // collide with the implementation contract's own storage variables.
    bytes32 private constant IMPLEMENTATION_SLOT =
        0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bb;

    constructor(address implementation) {
        assembly { sstore(IMPLEMENTATION_SLOT, implementation) }
    }

    fallback() external payable {
        assembly {
            let impl := sload(IMPLEMENTATION_SLOT)
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

Because storage is shared by slot position, not by variable name, the Proxy and every version of the Implementation must agree on storage layout — same variables, same order, forever. This is why the implementation address itself lives at a deliberately obscure, hash-derived slot (EIP-1967) instead of slot 0: slot 0 is exactly where the Implementation contract's own first state variable would naturally want to live.

> ⚡ **PITFALL** Reordering, resizing, or deleting a state variable in a new implementation version — instead of only appending new ones at the end — corrupts every existing storage slot the moment the proxy delegates to it. This is the single most common cause of \"working upgrade, broken contract\" incidents.

One more consequence of this split: a constructor only ever runs once, at the moment a contract is deployed — and the Implementation contract is deployed on its own, separately from the Proxy, so its constructor never

runs in the Proxy's storage context. Upgradeable contracts replace the constructor with a plain function, guarded to run exactly once:

```
// A constructor only runs once, at the moment the LOGIC contract itself
// is deployed — never when a user calls through the proxy. So logic
// contracts use a plain function, guarded to run exactly once, instead.
contract LogicV1 {
    bool private _initialized;
    address public owner;
    uint256 public value;

    error AlreadyInitialized();

    modifier initializer() {
        if (_initialized) revert AlreadyInitialized();
        _initialized = true;
        _;
    }

    function initialize(address _owner) external initializer {
        owner = _owner;
    }
}
```

i INFO UUPS (EIP-1822) and the older Transparent Proxy pattern differ mainly in where the upgrade logic itself lives — in the Implementation for UUPS, in the Proxy for Transparent. UUPS is the more common choice in new projects today because it's cheaper to deploy at scale. OpenZeppelin's `UUPSUpgradeable` and `Initializable` implement all of this correctly — reach for them before writing any of it by hand.

□ TRY IT YOURSELF — Combine Two Patterns

SafeVault from section 3.3 already has a reentrancy guard. Add Pausable to it as well: make SafeVault inherit from Pausable, add `whenNotPaused` to both `deposit` and `withdraw`, and add an `onlyOwner`-gated function that lets the contract owner call `_pause()`. You'll need to combine three patterns from this chapter in one contract.

Hint: You'll need Ownable's constructor and modifier too, since `_pause()` should only be callable by an owner — SafeVault will end up inheriting from all three: Ownable, Pausable, and ReentrancyGuard.

KEY TAKEAWAYS

- Ownable, Pausable, ReentrancyGuard, Factory, and Proxy patterns are all solved problems — know how they work, but use OpenZeppelin's audited implementations in real code.
- Reentrancy is a bug in ordering: always finish updating state (Checks-Effects-Interactions) before making an external call, and add a nonReentrant guard as a second layer of defense.
- A factory deploys contract instances on demand with `new` and keeps a registry of what it created; minimal proxy clones (Chapter 9) cut the gas cost at scale.
- A proxy holds storage at a fixed address and delegatecalls into an implementation that holds only logic; storage layout must stay compatible across every version, forever.
- Upgradeable contracts replace constructors with a run-once `initializer` function, since the implementation's own constructor never executes in the proxy's storage context.

□ DID YOU KNOW?

The 2016 DAO hack that opened this chapter's reentrancy discussion didn't just cost roughly a third of the fund's value — it split Ethereum itself. The community disagreed sharply over whether to hard-fork the chain to reverse the theft, and that disagreement is exactly why two separate chains exist today: Ethereum (ETH), which forked to recover the funds, and Ethereum Classic (ETC), whose community kept the original, unaltered chain on the principle that "code is law."

CHAPTER 4

Token Standards

A token standard is a contract interface everyone agreed on in advance. A wallet doesn't need custom code to display your token's balance, and an exchange doesn't need a special integration to list it — both already know what `balanceOf` and `transfer` mean, because the interface is standardized. This chapter builds all three major standards from the interface up, so "just inherit from OpenZeppelin" stops being a magic incantation and starts being an informed choice.

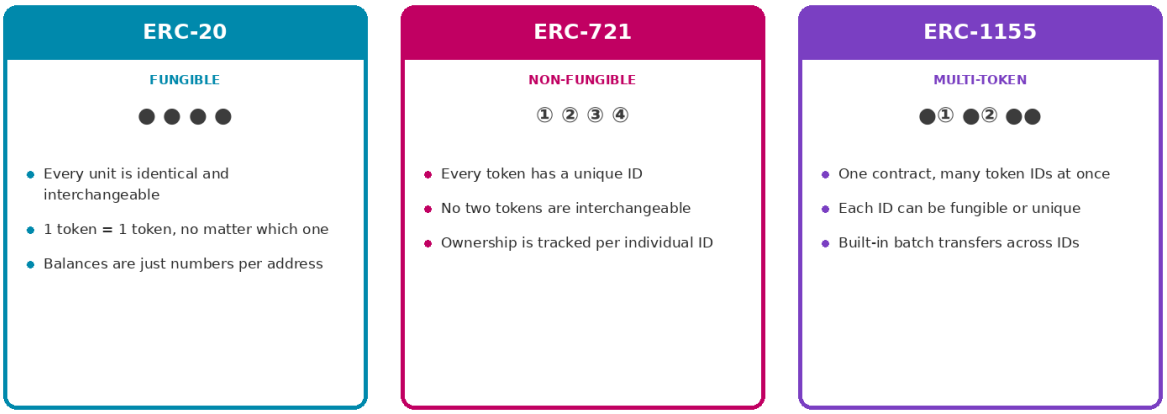

Figure 4.1 — Fungible, non-fungible, and multi-token, side by side

4.1 ERC-20 — The Fungible Token Standard

Fungible means interchangeable — your 10 tokens and my 10 tokens are worth exactly the same thing, with no identity of their own. This is the standard behind essentially every cryptocurrency, governance token, and stablecoin you've interacted with.

```
interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns
(uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external
returns (bool);
}

contract MyToken is IERC20 {
    string public name = "MyToken";
    string public symbol = "MTK";
    uint8 public constant decimals = 18;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    error InsufficientBalance(uint256 available, uint256 required);
    error InsufficientAllowance(uint256 available, uint256 required);
    error ZeroAddress();

    constructor(uint256 initialSupply) {
        _mint(msg.sender, initialSupply);
    }

    function totalSupply() external view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        _transfer(msg.sender, to, amount);
        return true;
    }

    function allowance(address owner, address spender) external view returns
(uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) external returns (bool) {
        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
```

```
    }

    function transferFrom(address from, address to, uint256 amount) external
returns (bool) {
        uint256 currentAllowance = _allowances[from][msg.sender];
        if (currentAllowance < amount) revert
InsufficientAllowance(currentAllowance, amount);
        _allowances[from][msg.sender] = currentAllowance - amount;
        _transfer(from, to, amount);
        return true;
    }

    function _transfer(address from, address to, uint256 amount) private {
        if (to == address(0)) revert ZeroAddress();
        uint256 fromBalance = _balances[from];
        if (fromBalance < amount) revert InsufficientBalance(fromBalance, amount);
        _balances[from] = fromBalance - amount;
        _balances[to] += amount;
        emit Transfer(from, to, amount);
    }

    function _mint(address to, uint256 amount) private {
        _totalSupply += amount;
        _balances[to] += amount;
        emit Transfer(address(0), to, amount);
    }
}
```

> ⚡ **PITFALL** Calling `approve` to change an existing non-zero allowance has a known race condition: a spender watching the mempool can front-run the new approval and spend the old allowance right before it updates. OpenZeppelin's real implementation exposes `increaseAllowance` / `decreaseAllowance` to avoid this — reach for those instead of calling `approve` twice.

4.2 ERC-721 — The Non-Fungible Token Standard

Non-fungible means each token has its own identity — a `tokenId` — and ownership is tracked per ID rather than as a single balance number. Token #4471 is not interchangeable with token #12, even from the same contract.

```
interface IERC721 {
    event Transfer(address indexed from, address indexed to, uint256 indexed
tokenId);
    event Approval(address indexed owner, address indexed approved, uint256
indexed tokenId);
    event ApprovalForAll(address indexed owner, address indexed operator, bool
approved);

    function balanceOf(address owner) external view returns (uint256);
    function ownerOf(uint256 tokenId) external view returns (address);
    function transferFrom(address from, address to, uint256 tokenId) external;
    function approve(address to, uint256 tokenId) external;
    function getApproved(uint256 tokenId) external view returns (address);
    function setApprovalForAll(address operator, bool approved) external;
    function isApprovedForAll(address owner, address operator) external view
returns (bool);
}

contract MyNFT is IERC721 {
    string public name = "MyNFT";
    string public symbol = "MNFT";
```

```
    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;
    mapping(uint256 => address) private _tokenApprovals;
    mapping(address => mapping(address => bool)) private _operatorApprovals;
    uint256 private _nextTokenId;

    error NotOwnerOrApproved();
    error TokenDoesNotExist(uint256 tokenId);
    error ZeroAddress();

    function balanceOf(address owner) external view returns (uint256) {
        if (owner == address(0)) revert ZeroAddress();
        return _balances[owner];
    }

    function ownerOf(uint256 tokenId) public view returns (address) {
        address tokenOwner = _owners[tokenId];
        if (tokenOwner == address(0)) revert TokenDoesNotExist(tokenId);
        return tokenOwner;
    }

    function approve(address to, uint256 tokenId) external {
        address tokenOwner = ownerOf(tokenId);
        bool allowed = msg.sender == tokenOwner ||
_operatorApprovals[tokenOwner][msg.sender];
        if (!allowed) revert NotOwnerOrApproved();
        _tokenApprovals[tokenId] = to;
        emit Approval(tokenOwner, to, tokenId);
    }

    function getApproved(uint256 tokenId) external view returns (address) {
        ownerOf(tokenId);
        return _tokenApprovals[tokenId];
    }

    function setApprovalForAll(address operator, bool approved) external {
        _operatorApprovals[msg.sender][operator] = approved;
        emit ApprovalForAll(msg.sender, operator, approved);
    }

    function isApprovedForAll(address owner, address operator) external view
returns (bool) {
        return _operatorApprovals[owner][operator];
    }

    function transferFrom(address from, address to, uint256 tokenId) public {
        address tokenOwner = ownerOf(tokenId);
        bool allowed = msg.sender == tokenOwner
            || _tokenApprovals[tokenId] == msg.sender
            || _operatorApprovals[tokenOwner][msg.sender];
        if (!allowed) revert NotOwnerOrApproved();
        if (to == address(0)) revert ZeroAddress();

        delete _tokenApprovals[tokenId];
        _balances[from] -= 1;
        _balances[to] += 1;
        _owners[tokenId] = to;
        emit Transfer(from, to, tokenId);
    }

    function mint(address to) external returns (uint256) {
        uint256 tokenId = _nextTokenId++;
```

```
        _balances[to] += 1;
        _owners[tokenId] = to;
        emit Transfer(address(0), to, tokenId);
        return tokenId;
    }
}
```

> ⚠ **SECURITY** Real ERC-721 also defines `safeTransferFrom`, omitted above for space — it performs the same transfer, then checks that the recipient, if it's a contract, actually implements `onERC721Received`. Skipping that check (using plain `transferFrom` to send to a contract that can't handle NFTs) is a common, real way for tokens to become permanently stuck. Prefer `safeTransferFrom` whenever the recipient might be a contract.

4.3 ERC-1155 — The Multi-Token Standard

One ERC-1155 contract can hold thousands of distinct token IDs simultaneously, and each ID can behave as fungible (mint a large supply of ID #1 as an in-game currency) or effectively non-fungible (mint exactly one of ID #900 as a unique item) — the contract doesn't force a choice.

```
interface IERC1155 {
    event TransferSingle(address indexed operator, address indexed from, address
indexed to, uint256 id, uint256 value);
    event ApprovalForAll(address indexed owner, address indexed operator, bool
approved);

    function balanceOf(address account, uint256 id) external view returns
(uint256);
    function balanceOfBatch(address[] calldata accounts, uint256[] calldata ids)
external view returns (uint256[] memory);
    function setApprovalForAll(address operator, bool approved) external;
    function isApprovedForAll(address account, address operator) external view
returns (bool);
    function safeTransferFrom(address from, address to, uint256 id, uint256
amount, bytes calldata data) external;
}

contract MyMultiToken is IERC1155 {
    mapping(uint256 => mapping(address => uint256)) private _balances;
    mapping(address => mapping(address => bool)) private _operatorApprovals;

    error NotOwnerOrApproved();
    error InsufficientBalance(uint256 available, uint256 required);

    function balanceOf(address account, uint256 id) public view returns (uint256)
{
        return _balances[id][account];
    }

    function balanceOfBatch(address[] calldata accounts, uint256[] calldata ids)
        external view returns (uint256[] memory)
    {
        uint256[] memory batchBalances = new uint256[](accounts.length);
        for (uint256 i = 0; i < accounts.length; i++) {
            batchBalances[i] = balanceOf(accounts[i], ids[i]);
        }
        return batchBalances;
    }

    function setApprovalForAll(address operator, bool approved) external {
        _operatorApprovals[msg.sender][operator] = approved;
        emit ApprovalForAll(msg.sender, operator, approved);
```

```
    }

    function isApprovedForAll(address account, address operator) external view
returns (bool) {
        return _operatorApprovals[account][operator];
    }

    function safeTransferFrom(address from, address to, uint256 id, uint256
amount, bytes calldata data) external {
        bool allowed = from == msg.sender || _operatorApprovals[from][msg.sender];
        if (!allowed) revert NotOwnerOrApproved();
        uint256 fromBalance = _balances[id][from];
        if (fromBalance < amount) revert InsufficientBalance(fromBalance, amount);

        _balances[id][from] = fromBalance - amount;
        _balances[id][to] += amount;
        emit TransferSingle(msg.sender, from, to, id, amount);
    }

    function mint(address to, uint256 id, uint256 amount) external {
        _balances[id][to] += amount;
        emit TransferSingle(msg.sender, address(0), to, id, amount);
    }
}
```

GAS TIP The batch functions — `balanceOfBatch`, `safeBatchTransferFrom` — are the actual reason to reach for ERC-1155 over deploying many separate ERC-20 or ERC-721 contracts: one transaction moving five different item types costs meaningfully less gas than five separate transfer transactions.

4.4 Choosing the Right Standard

	ERC-20	ERC-721	ERC-1155
Best for	Currencies, points, shares — anything interchangeable	Art, collectibles, deeds — anything unique	Game items, editions — mixed fungible + unique
Balance model	One number per address	One owner per token ID	One number per (address, ID) pair
Transfer call	transfer(to, amount)	transferFrom(from, to, tokenId)	safeTransferFrom(from, to, id, amount, data)
Batch operations	Not built in	Not built in	Native — safeBatchTransferFrom

4.5 Token Metadata

Neither ERC-721 nor ERC-1155 stores an image or description on-chain by default — both define a `tokenURI(uint256 tokenId)` function that returns a URL, and that URL points to a JSON file following a standard schema: `name`, `description`, `image`, and an optional `attributes` array for traits. A wallet or marketplace fetches that JSON to render everything you actually see when you look at an NFT.

⚡ PITFALL Pointing tokenURI at a centralized web server means the NFT's metadata (and even its image) can disappear the moment that server goes down — the token itself survives on-chain forever, but what it's supposed to represent doesn't. Production collections store metadata on IPFS or Arweave specifically so the content is content-addressed and doesn't depend on any one company staying online.

□ TRY IT YOURSELF — Add tokenURI

Add a tokenURI(uint256 tokenId) function to MyNFT (section 4.2) that returns a string. It should revert with TokenDoesNotExist if the token hasn't been minted, and otherwise return a placeholder like "ipfs://bafybase/" concatenated with the tokenId as a string.

Hint: You'll need ownerOf(tokenId) for the existence check (it already reverts correctly), and Solidity's string(abi.encodePacked(...)) or a library like OpenZeppelin's Strings.toString() to convert the uint256 into text.

□ TRY IT YOURSELF — Add a Burn Function

Add a `burn(uint256 amount)` function to MyToken (section 4.1) that destroys tokens from the caller's own balance: decrease their balance, decrease totalSupply, and emit a Transfer event to address(0) — the same convention _mint uses in reverse.

Hint: You'll need a custom error for when the caller tries to burn more than they hold, following the same InsufficientBalance pattern _transfer already uses.

KEY TAKEAWAYS

- A token standard is a shared interface — wallets and exchanges support any token that implements it, with zero custom integration work.
- ERC-20 tracks one balance number per address; transfers move value between two numbers.
- ERC-721 tracks an owner per unique tokenId; prefer safeTransferFrom so contracts that can't handle NFTs are rejected instead of trapping them forever.
- ERC-1155 tracks balances per (address, id) pair in one contract, mixing fungible and unique items, with native batch transfers.
- In production, inherit from OpenZeppelin's audited implementations of all three rather than deploying hand-written versions like the ones in this chapter.
- tokenURI points to off-chain JSON metadata for ERC-721/1155 — store it on IPFS or Arweave, not a centralized server, or the token can outlive what it's supposed to represent.
- None of this chapter's contracts implement safeTransferFrom's receiver check or the full metadata extension — treat them as a foundation to understand, not a foundation to deploy as-is.

□ DID YOU KNOW?

ERC-20 was proposed on GitHub by developer Fabian Vogelsteller in November 2015 — but it wasn't formally finalized as EIP-20 until 2017, nearly two years later, once Vitalik Buterin co-authored the official specification. In that gap, the standard was already being used informally across the ecosystem; the formal spec caught up to practice, not the other way around.

DeFi Protocol Development

Every major category of DeFi — exchanges, price feeds, lending — rests on a small number of core mechanisms. This chapter builds the three you'll meet in almost every protocol you read: the constant product formula behind AMMs, how a contract gets real-world price data at all, and how a lending protocol decides a position is safe or must be liquidated.

5.1 AMMs and the Constant Product Formula

An automated market maker needs no order book and no counterparty — it holds two token reserves and prices trades using a formula. The one nearly every AMM since Uniswap V2 is built on is deceptively short: x · y = k. The product of the two reserves must stay constant across every trade (fees aside).

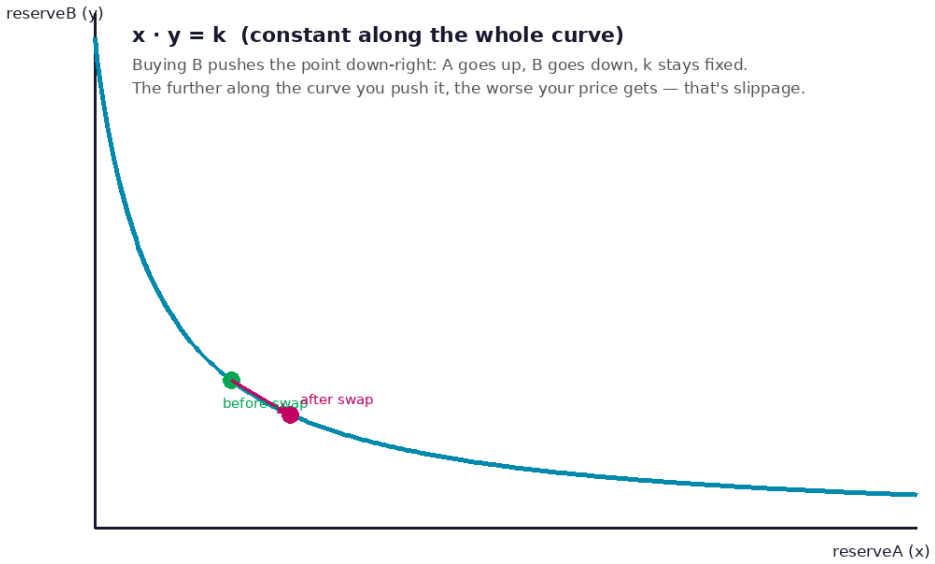

Figure 5.1 — Every swap moves along the curve; k never changes, only the split between x and y does

```
interface IERC20 {
    function transferFrom(address from, address to, uint256 amount) external
returns (bool);
    function transfer(address to, uint256 amount) external returns (bool);
}

contract SimpleAMM {
    IERC20 public immutable tokenA;
    IERC20 public immutable tokenB;

    uint256 public reserveA;
    uint256 public reserveB;

    event LiquidityAdded(address indexed provider, uint256 amountA, uint256
amountB);
    event Swap(address indexed trader, uint256 amountIn, uint256 amountOut);

    error InsufficientLiquidity();
    error InsufficientOutput(uint256 got, uint256 wanted);

    constructor(address _tokenA, address _tokenB) {
        tokenA = IERC20(_tokenA);
```

```
        tokenB = IERC20(_tokenB);
    }

    function addLiquidity(uint256 amountA, uint256 amountB) external {
        tokenA.transferFrom(msg.sender, address(this), amountA);
        tokenB.transferFrom(msg.sender, address(this), amountB);
        reserveA += amountA;
        reserveB += amountB;
        emit LiquidityAdded(msg.sender, amountA, amountB);
    }

    // x * y = k must hold (after fees) before and after every swap
    function swapAForB(uint256 amountAIn, uint256 minAmountBOut) external {
        if (reserveA == 0 || reserveB == 0) revert InsufficientLiquidity();

        uint256 amountAInWithFee = amountAIn * 997;   // 0.3% fee, same as Uniswap
V2
        uint256 numerator = amountAInWithFee * reserveB;
        uint256 denominator = (reserveA * 1000) + amountAInWithFee;
        uint256 amountBOut = numerator / denominator;

        if (amountBOut < minAmountBOut) revert InsufficientOutput(amountBOut,
minAmountBOut);

        tokenA.transferFrom(msg.sender, address(this), amountAIn);
        reserveA += amountAIn;
        reserveB -= amountBOut;
        tokenB.transfer(msg.sender, amountBOut);

        emit Swap(msg.sender, amountAIn, amountBOut);
    }
}
```

> ⚡ **PITFALL** `minAmountBOut` isn't a formality — without it, a trade sent to the mempool can be sandwiched: someone buys ahead of you to push the price against you, lets your trade execute at the worse price, then sells right after. Always compute an acceptable minimum off-chain first and pass it in; never accept whatever the contract happens to return.

5.2 Chainlink Oracles — Getting Price Data On-Chain

A smart contract cannot make an HTTP request. It can only read what's already on-chain. Chainlink's price feeds solve this by having a decentralized network of node operators independently fetch off-chain prices and post them, aggregated, into a contract your contract can then read — an oracle, in the general sense.

```
interface AggregatorV3Interface {
    function decimals() external view returns (uint8);
    function latestRoundData() external view returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    );
}

contract PriceConsumer {
    AggregatorV3Interface public immutable priceFeed;
    uint256 public constant MAX_STALENESS = 3600;   // 1 hour

    error InvalidPrice();
    error StalePrice(uint256 lastUpdated, uint256 now_);
```

```
    constructor(address _priceFeed) {
        priceFeed = AggregatorV3Interface(_priceFeed);
    }

    function getLatestPrice() public view returns (int256) {
        (, int256 answer, , uint256 updatedAt, ) = priceFeed.latestRoundData();

        if (answer <= 0) revert InvalidPrice();
        if (block.timestamp - updatedAt > MAX_STALENESS) {
            revert StalePrice(updatedAt, block.timestamp);
        }
        return answer;
    }
}
```

> ⚠ **SECURITY** Two checks in `getLatestPrice` are not optional. Skipping the staleness check means trusting a price the network stopped updating — during exactly the kind of market stress that makes a feed lag, a stale price is the most dangerous kind. And more broadly: a huge share of historical DeFi exploits are variations of *reading a manipulable price at exactly the wrong moment* — a flash-loan-inflated spot price, not a proper time-weighted or Chainlink-aggregated one. Never price anything from a pool's own instantaneous reserves if real value depends on it.

5.3 Lending Protocol Concepts

A lending protocol's entire safety model comes down to one repeated question: is this borrower's collateral still worth enough, relative to what they owe? That ratio is usually called the health factor.

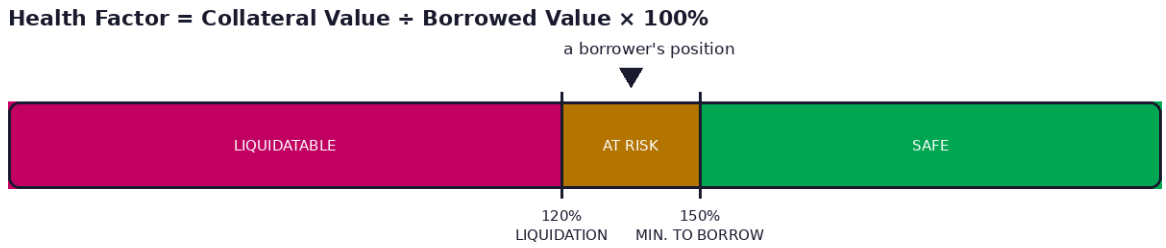

Figure 5.2 — A position's health factor determines whether it can still borrow, is at risk, or can be liquidated right now

```
contract SimpleLending {
    IERC20 public immutable collateralToken;
    IERC20 public immutable borrowToken;
    PriceConsumer public immutable priceFeed;

    uint256 public constant MIN_COLLATERAL_RATIO = 150;   // must deposit 150% of
borrowed value
    uint256 public constant LIQUIDATION_THRESHOLD = 120;  // liquidatable once
health drops below this

    mapping(address => uint256) public collateralBalance;
    mapping(address => uint256) public borrowBalance;

    error InsufficientCollateral(uint256 have, uint256 need);
    error PositionHealthy(uint256 currentRatio);

    function depositCollateral(uint256 amount) external {
        collateralToken.transferFrom(msg.sender, address(this), amount);
        collateralBalance[msg.sender] += amount;
    }
```

```
    function borrow(uint256 amount) external {
        uint256 newBorrowBalance = borrowBalance[msg.sender] + amount;
        uint256 ratio = _healthRatio(collateralBalance[msg.sender],
newBorrowBalance);
        if (ratio < MIN_COLLATERAL_RATIO) {
            revert InsufficientCollateral(ratio, MIN_COLLATERAL_RATIO);
        }
        borrowBalance[msg.sender] = newBorrowBalance;
        borrowToken.transfer(msg.sender, amount);
    }

    function liquidate(address borrower) external {
        uint256 ratio = _healthRatio(collateralBalance[borrower],
borrowBalance[borrower]);
        if (ratio >= LIQUIDATION_THRESHOLD) revert PositionHealthy(ratio);
        // A real implementation: liquidator repays the debt, and receives the
        // borrower's collateral at a discount, funded by the repayment. Left out
        // here to keep focus on the health-ratio mechanics above.
    }

    function _healthRatio(uint256 collateral, uint256 borrowed) internal view
returns (uint256) {
        if (borrowed == 0) return type(uint256).max;
        uint256 collateralValue = uint256(priceFeed.getLatestPrice()) *
collateral;
        return (collateralValue * 100) / borrowed;
    }
}
```

Three numbers do all the work: the minimum ratio required to borrow in the first place (150% here — over-collateralization is the norm in DeFi lending, not the exception), the threshold below which anyone can liquidate the position (120%), and the health ratio itself, recomputed from live collateral and debt values every time it matters.

> ⚠ **SECURITY** This is where 5.1 and 5.2 connect: `_healthRatio` depends entirely on `priceFeed.getLatestPrice()`. If that price can be briefly manipulated — say, via a large trade against a thin pool some other part of the protocol trusts — an attacker can make healthy positions look liquidatable, or borrow more than real collateral allows. Manipulable price source plus lending logic that trusts it is the most common root cause behind major DeFi exploits.

> **i INFO** Real-world case: in October 2022, Mango Markets lost over $100 million this exact way. An attacker split about $10 million across two accounts, then used leveraged positions to push the price of MNGO — the protocol's own token — sharply upward within minutes. Mango's lending logic read that inflated price as genuine collateral value and let him borrow far more than his real collateral was worth. He was later prosecuted for market manipulation — "the oracle just reported what the market said" was not a defense, on-chain or in court.

5.4 Putting It Together

Sections 5.1 through 5.3 are not independent — a real protocol cross-checks them against each other constantly. Here's the pattern in code: before trusting an AMM pool's own price for anything collateral-related, compare it against a trusted oracle first.

```
// A borrower's collateral is itself an LP position in SimpleAMM. Before
// approving a larger loan, the protocol re-checks the live, oracle-verified
// value of that collateral — not whatever the pool's own reserves claim.
contract CollateralValuation {
    SimpleAMM public immutable pool;
    PriceConsumer public immutable trustedOracle;
```

```
    error ValueMismatchTooLarge(uint256 poolImplied, uint256 oracleValue);

    constructor(address _pool, address _oracle) {
        pool = SimpleAMM(_pool);
        trustedOracle = PriceConsumer(_oracle);
    }

    // Cross-checks the AMM pool's implied price against Chainlink before
    // trusting it for anything collateral-related — exactly the discipline
    // section 5.3's SimpleLending should apply in production.
    function verifiedCollateralValue(uint256 lpTokenAmount) external view returns
(uint256) {
        uint256 poolImpliedPrice = pool.reserveB() * 1e18 / pool.reserveA();
        uint256 oraclePrice = uint256(trustedOracle.getLatestPrice());

        uint256 deviation = poolImpliedPrice > oraclePrice
            ? poolImpliedPrice - oraclePrice
            : oraclePrice - poolImpliedPrice;

        // reject if the pool's price has drifted more than 5% from the oracle —
        // a strong signal someone is actively manipulating this specific pool
        if (deviation * 100 / oraclePrice > 5) {
            revert ValueMismatchTooLarge(poolImpliedPrice, oraclePrice);
        }

        return lpTokenAmount * oraclePrice / 1e18;
    }
}
```

GAS TIP This cross-check costs extra gas on every call — an external call to the oracle, plus the comparison math. For a lending protocol, that's a small, worthwhile price: the alternative is exactly the Mango Markets failure mode above.

5.5 From Toy Example to Production

Every contract in this chapter strips away exactly the complexity that would distract from the core mechanism. Here's what the real, audited version of each one adds on top:

This chapter's example	Production equivalent	What production adds
SimpleAMM (5.1)	Uniswap V2 / V3	Real pools add protocol fees, price oracles built from trade history, and concentrated liquidity (V3)
PriceConsumer (5.2)	Chainlink Data Feeds	Production feeds also expose heartbeat intervals and deviation thresholds per asset pair
SimpleLending (5.3)	Aave / Compound	Real protocols add interest rate curves, multiple collateral types, and partial liquidations

5.6 DeFi Terms You'll See Everywhere

A handful of terms show up in nearly every DeFi conversation, article, and audit report from here on. Worth having pinned down before Chapter 6:

Term	What it actually means
TVL (Total Value Locked)	The combined value of every asset currently deposited in a protocol — a rough size/trust indicator, not a safety guarantee

Impermanent Loss	What an AMM liquidity provider gives up compared to just holding the two tokens, whenever their price ratio moves apart
APR vs. APY	APR is a simple annualized rate; APY compounds that rate — APY is always higher for the same underlying return whenever compounding applies
Slippage Tolerance	The maximum acceptable gap between a trade's expected and executed price — this is exactly what minAmountBOut enforces in section 5.1
Flash Loan	A loan that must be borrowed and repaid within a single transaction — if repayment fails, the entire transaction reverts, so the lender risks nothing

□ TRY IT YOURSELF — Add a Second Trading Direction

SimpleAMM (section 5.1) only has swapAForB. Write swapBForA(uint256 amountBIn, uint256 minAmountAOut) that does the same thing in the other direction — same 0.3% fee, same constant-product math, just with reserveA and reserveB swapped in the formula.

Hint: Copy swapAForB's formula and swap every occurrence of reserveA with reserveB and vice versa — the math is symmetric, only the direction of the trade changes.

□ TRY IT YOURSELF — Extend the Deviation Check

CollateralValuation (section 5.4) reverts if the pool's implied price drifts more than 5% from the oracle. Add a second constructor parameter, maxDeviationPercent, so each deployment can set its own tolerance instead of a hardcoded 5 — a stablecoin pair might want 1%, a volatile pair might reasonably need 10%.

Hint: Replace the hardcoded 5 in the deviation check with a state variable set once in the constructor and stored as immutable, exactly like pool and trustedOracle already are.

KEY TAKEAWAYS

- An AMM prices trades with $x \cdot y = k$; every swap moves along the curve, and the further it moves, the worse the price gets — that's slippage.
- Always pass a minimum-output guard into a swap; without one, a transaction sitting in the mempool can be sandwiched for a worse price.
- Contracts can't fetch external data themselves — oracles like Chainlink post aggregated off-chain prices on-chain for contracts to read.
- Always check an oracle price for staleness before trusting it; a frozen feed during market stress is the most dangerous kind of bad data.
- A lending protocol's core safety check is the health factor — collateral value over debt value — and it depends entirely on a manipulation-resistant price source.
- This isn't theoretical: Mango Markets lost over $100M in 2022 to exactly this pattern — a token price pushed up on a thin market, then borrowed against as if it were genuine collateral value.
- Cross-check an AMM pool's own implied price against a trusted oracle before using it for anything collateral-related — a large deviation between the two is itself a signal something is being manipulated.

- None of the contracts in this chapter are production-ready as written — real protocols add fees, multiple collateral types, and interest curves on top of the same core mechanisms shown here.

□ DID YOU KNOW?

Uniswap, the protocol behind this chapter's constant product formula, started after its founder Hayden Adams was laid off from his job as a mechanical engineer at Siemens in mid-2017. A friend pointed him toward Vitalik Buterin's writing on automated market makers, and Adams — with no prior Solidity experience — spent the next year teaching himself to code by building it, with direct feedback from Buterin along the way.

Every contract in this chapter — the AMM, the oracle consumer, the lending pool — was written for clarity, not for defense against a determined attacker. Chapter 6 turns to exactly that: the top vulnerability classes beyond reentrancy and oracle manipulation, the Checks-Effects-Interactions discipline applied systematically, and the checklist a real audit actually runs through.

CHAPTER 6

Security Best Practices

Reentrancy (Chapter 3) and oracle manipulation (Chapter 5) are two of the most damaging vulnerability classes in DeFi's history, but they're far from the only ones. This chapter works through the rest of the list

every serious audit checks for — each with the vulnerable version first, so the fix means something, followed by the pattern that actually prevents it.

6.1 Access Control Failures

The most common security bug in smart contracts isn't exotic — it's simply forgetting to check who's calling a function that changes something important.

```
// VULNERABLE — anyone can call this, not just the intended admin
contract VulnerableConfig {
    address public admin;
    uint256 public protocolFeeBps;

    constructor() {
        admin = msg.sender;
    }

    function setFee(uint256 newFeeBps) external {
        protocolFeeBps = newFeeBps;   // no check on who's calling!
    }
}
```

```
contract FixedConfig {
    address public admin;
    uint256 public protocolFeeBps;

    error NotAdmin(address caller);

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        if (msg.sender != admin) revert NotAdmin(msg.sender);
        _;
    }

    function setFee(uint256 newFeeBps) external onlyAdmin {
        protocolFeeBps = newFeeBps;
    }
}
```

6.2 Unchecked External Call Return Values

`send()` and low-level `call()` return a boolean instead of reverting on failure — and if you don't check it, your contract silently continues as if the transfer worked.

```
// VULNERABLE — return value of send() is ignored
contract VulnerablePayout {
    function payout(address payable recipient, uint256 amount) external {
        recipient.send(amount);   // fails silently if it reverts!
        // code here assumes the payout succeeded — it might not have
    }
}
```

```
contract FixedPayout {
    error PayoutFailed(address recipient, uint256 amount);

    function payout(address payable recipient, uint256 amount) external {
        (bool success, ) = recipient.call{value: amount}("");
```

```
        if (!success) revert PayoutFailed(recipient, amount);
    }
}
```

> ⚡ **PITFALL** `.transfer()` and `.send()` also forward a hard 2300 gas stipend, which breaks against any recipient whose fallback function needs more than that (common with smart contract wallets). `call{value: amount}("")` forwards all remaining gas and is the modern default — but that means it MUST be paired with a return-value check and, ideally, a reentrancy guard, since unlike `.transfer()` it doesn't limit what the recipient can do.

6.3 tx.origin Authentication

`tx.origin` is the address that originally signed and sent the transaction, no matter how many contracts it passed through. `msg.sender` is whoever (or whatever) called the current function directly. That difference is exactly what makes `tx.origin` phishable.

```
// VULNERABLE — phishable authentication
contract VulnerableWallet {
    address public owner;

    constructor() { owner = msg.sender; }

    function transfer(address payable to, uint256 amount) external {
        require(tx.origin == owner, "Not owner");   // looks safe. isn't.
        to.transfer(amount);
    }
}
```

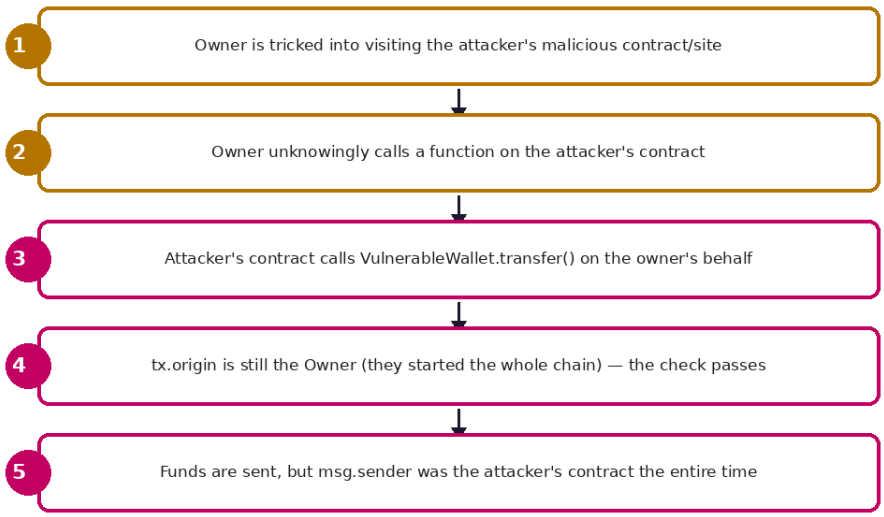

Figure 6.1 — tx.origin stays the same across the whole call chain, which is exactly the problem

```
contract FixedWallet {
    address public owner;

    error NotOwner(address caller);

    constructor() { owner = msg.sender; }

    function transfer(address payable to, uint256 amount) external {
        if (msg.sender != owner) revert NotOwner(msg.sender);
        to.transfer(amount);
```

```
    }
}
```

> ⚠ **SECURITY** Never use tx.origin for authorization, ever — not "usually," not "unless you're careful." msg.sender is the correct check in every ordinary case.

6.4 Weak Randomness

Nothing derived purely from block data is unpredictable to the entity producing that block.

```
// VULNERABLE — every input here is known or predictable in advance
contract VulnerableLottery {
    function pickWinner(uint256 participantCount) external view returns (uint256)
{
        return uint256(
            keccak256(abi.encodePacked(block.timestamp, block.prevrandao))
        ) % participantCount;
        // a validator can see this outcome before finalizing the block
    }
}
```

> ⚠ **SECURITY** A validator proposing a block can see the exact outcome of `pickWinner` before deciding whether to include the transaction at all — and simply not include it if the result doesn't favor them. Real randomness needs a source outside the chain's own control, like Chainlink VRF, which provides a value together with a cryptographic proof that it wasn't tampered with.

6.5 Integer Overflow and unchecked Blocks

Versions before Solidity 0.8 wrapped silently on overflow, and entire exploits were built on that. Every version this book uses reverts by default — but `unchecked {}` exists specifically to opt back out, for gas savings you'll see justified in Chapter 8.

```
// Solidity 0.8+ reverts on overflow/underflow BY DEFAULT — this is safe:
function safeSum(uint8 a, uint8 b) external pure returns (uint8) {
    return a + b;   // reverts automatically if the result would exceed 255
}

// unchecked{} opts back OUT of that protection, for gas savings (Chapter 8).
// Only use it where overflow is mathematically provably impossible:
function unsafeSum(uint8 a, uint8 b) external pure returns (uint8) {
    unchecked {
        return a + b;   // 200 + 100 silently wraps to 44 — no revert at all
    }
}
```

6.6 Signature Replay

A signature only proves who signed a message — it says nothing about how many times that message should be allowed to execute.

```
// VULNERABLE — the same valid signature can be submitted forever
contract VulnerablePermit {
    mapping(address => uint256) public balances;

    function withdrawWithSig(address user, uint256 amount, bytes memory signature)
external {
        bytes32 hash = keccak256(abi.encodePacked(user, amount));
        require(_recover(hash, signature) == user, "Invalid signature");
        balances[user] -= amount;
        payable(msg.sender).transfer(amount);
```

```
        // nothing here stops this exact signature being replayed tomorrow
    }

    function _recover(bytes32 hash, bytes memory sig) internal pure returns
(address) { /* ... */ }
}
```

```
contract FixedPermit {
    mapping(address => uint256) public balances;
    mapping(address => uint256) public nonces;

    error InvalidSignature();

    function withdrawWithSig(address user, uint256 amount, uint256 nonce, bytes
memory signature) external {
        if (nonce != nonces[user]) revert InvalidSignature();
        bytes32 hash = keccak256(abi.encodePacked(user, amount, nonce,
address(this)));
        if (_recover(hash, signature) != user) revert InvalidSignature();

        nonces[user]++;              // this exact signature can never be replayed
again
        balances[user] -= amount;
        payable(msg.sender).transfer(amount);
    }

    function _recover(bytes32 hash, bytes memory sig) internal pure returns
(address) { /* ... */ }
}
```

GAS TIP Including `address(this)` inside the signed hash, as FixedPermit does, also prevents a signature from being replayed against a different deployment of the same contract — a real attack against contracts deployed at predictable addresses across multiple chains.

6.7 Denial of Service via Unbounded Loops

Looping over an array whose size a user controls is a liability twice over: it can exceed the block gas limit as it grows, and a single failing element can block the entire operation for everyone else.

```
// VULNERABLE — one reverting recipient blocks the payout for everyone
contract VulnerableDistributor {
    address[] public recipients;

    function distribute() external payable {
        uint256 share = msg.value / recipients.length;
        for (uint256 i = 0; i < recipients.length; i++) {
            payable(recipients[i]).transfer(share);   // one revert stops the
whole loop
        }
    }
}
```

```
// Pull-over-push: each recipient claims their own share, on their own
transaction.
// One broken recipient can no longer block anyone else's payout.
contract FixedDistributor {
    mapping(address => uint256) public owed;

    function credit(address recipient, uint256 amount) external {
        owed[recipient] += amount;
    }
```

```
    function withdraw() external {
        uint256 amount = owed[msg.sender];
        owed[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

6.8 Checks-Effects-Interactions, Applied Systematically

Chapter 3 introduced this pattern specifically for reentrancy. It's worth restating as a general habit, because nearly every vulnerability in this chapter is some variation of doing things in the wrong order:

Figure 6.2 — The order that makes an entire category of bugs impossible

Validate first. Commit every state change second. Only then talk to the outside world — another contract, a token transfer, an external call whose behavior you don't fully control. Nearly every fix in this chapter is this ordering, applied to a different kind of "outside world": a signature, a price feed, a loop over user-controlled data.

6.9 The Audit Checklist

Before shipping anything to mainnet, or before ever paying for a professional audit, run through this yourself:

Category	Ask this
Access Control	Every state-changing function — who is allowed to call it, and is that actually enforced?
Reentrancy	Does every function follow Checks-Effects-Interactions? Is nonReentrant applied where it matters?
External Calls	Is every call's return value checked? Could a failing call block other users (DoS)?
Price / Oracle Data	Is every price source staleness-checked? Can it be manipulated in a single transaction?
Arithmetic	Any unchecked {} blocks — and is overflow truly, provably impossible there?
Authentication	Any use of tx.origin for authorization? (There shouldn't be any.)
Randomness	Any on-chain “random” value derived from block data alone?
Signatures	Can any valid signature be replayed? Is there a nonce, and is the contract address included in the signed hash?
Upgrades	If upgradeable: is storage layout preserved? Is initialize() protected from being called twice?

□ TRY IT YOURSELF — Find and Fix

Here's a one-function contract: a Vault with a mapping of balances, and a setBalance(address user, uint256 amount) function that just writes directly into that mapping — external, no modifier, no checks. It has exactly one of this chapter's vulnerabilities. Name which section it matches, then rewrite the function to fix it.

Hint: Nothing checks who is calling setBalance — compare it against section 6.1's VulnerableConfig.

□ TRY IT YOURSELF — Add Replay Protection

VulnerablePermit (section 6.6) is missing more than just a nonce check — walk through what would happen if the SAME signature were submitted to two different deployments of this exact contract on two different chains. Then write the one-line change to FixedPermit's hash that already prevents it.

Hint: Look again at what _recover's hash includes in FixedPermit versus VulnerablePermit — one extra value closes this exact gap.

KEY TAKEAWAYS

- Access control failures are the most common smart contract bug — check who's calling before anything else, every time.
- Always check the return value of send() and low-level call(); a silent failure is worse than a loud one.
- Never use tx.origin for authorization — it stays constant across an entire call chain, which is exactly what makes it phishable.
- Nothing derived from block.timestamp or block.prevrandao alone is unpredictable to whoever produces that block; use an external randomness source like Chainlink VRF instead.
- unchecked{} opts out of Solidity 0.8's default overflow protection — only use it where overflow is provably impossible.
- A signature alone doesn't prevent replay; pair it with a nonce, and include the contract's own address in the signed hash to stop cross-chain and cross-contract replay too.
- Never loop over a user-controlled array to send funds; let each recipient pull their own share instead, so one failure can't block everyone else.
- Checks-Effects-Interactions isn't just for reentrancy — validate, then update state, then talk to the outside world, in that order, everywhere.

□ DID YOU KNOW?

A meaningful share of the largest DeFi exploits in history were never actually kept by the attacker. In several high-profile cases, so-called "whitehat" or grey-hat hackers who found a critical bug returned most or all of the funds after the protocol negotiated with them publicly on-chain — sometimes even keeping a percentage as an informal bug bounty. This is now common enough that some protocols include an explicit "whitehat safe harbor" clause in their terms, offering a reward in advance for exactly this outcome.

CHAPTER 7

Development Environment

Every code example so far has assumed Foundry. This chapter explains why, sets up an actual project, and walks through the toolchain you'll use for the rest of the book.

7.1 Hardhat vs Foundry — Two Philosophies

Hardhat is a JavaScript/TypeScript framework — tests, scripts, and configuration are all JS. Foundry takes a different approach entirely: tests are written in Solidity itself, and the whole toolchain is a set of Rust binaries with no Node.js dependency at all.

	Hardhat	Foundry
Language	TypeScript / JavaScript	Solidity (tests are contracts too)
Test speed	Slower — JS/EVM boundary on every call	Very fast — Rust-native EVM, no JS bridge
Local node	Hardhat Network (built in)	anvil (built in)
Fuzz testing	Third-party plugins	Built in, first-class (Chapter 12)
Scripting deploys	JS/TS deploy scripts	Solidity scripts, or cast for one-offs
Ecosystem	Larger plugin ecosystem, more tutorials historically	Faster-growing, now the default for new serious projects

i INFO Hardhat isn't obsolete — plenty of serious teams still use it, and it remains genuinely stronger for projects with heavy JavaScript/TypeScript tooling needs (complex frontend-integrated deploy pipelines, for instance). This book uses Foundry because writing tests in Solidity means you're never context-switching languages, and its fuzzing and gas reporting are first-class rather than bolted on.

7.2 The Foundry Toolchain

"Foundry" is actually four separate command-line tools, installed together:

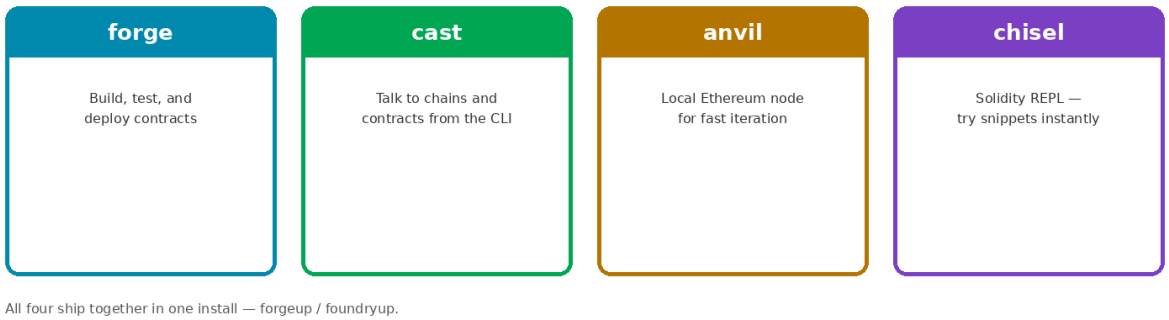

Figure 7.1 — The four tools inside Foundry, and what each one is for

7.3 Setting Up a Project

Installing Foundry (once): `curl -L https://foundry.paradigm.xyz | bash`, then `foundryup`. From there, a new project:

```
forge init my-project
cd my-project
forge build
```

`forge init` scaffolds a standard layout — `src/` for contracts, `test/` for tests, `script/` for deploy scripts, and `lib/` for dependencies (added with `forge install`, not npm). Configuration lives in one file at the project root:

```
[profile.default]
src = "src"
out = "out"
libs = ["lib"]
solc_version = "0.8.20"
optimizer = true
optimizer_runs = 200

[rpc_endpoints]
mainnet = "${MAINNET_RPC_URL}"
sepolia = "${SEPOLIA_RPC_URL}"
```

7.4 Writing and Running Your First Test

Foundry tests are ordinary Solidity contracts that inherit from `forge-std`'s `Test`. Here's a real test against Chapter 2's SimpleCounter:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test} from "forge-std/Test.sol";
import {SimpleCounter} from "../src/SimpleCounter.sol";

contract SimpleCounterTest is Test {
    SimpleCounter public counter;
    address public user = address(0xBEEF);

    function setUp() public {
        counter = new SimpleCounter();
    }

    function test_IncrementIncreasesCount() public {
        counter.increment();
        assertEq(counter.getCount(), 1);
```

```
    }

    function test_RevertWhen_NonOwnerResets() public {
        vm.prank(user);
        vm.expectRevert();
        counter.reset();
    }
}
```

Run it with `forge test`, or `forge test -vvv` for a full trace on any failure. `vm.prank(user)` makes the next call appear to come from `user` instead of the test contract itself — exactly what's needed to test that `reset()` correctly rejects non-owners.

> **GAS TIP** `forge test --gas-report` prints exactly how much gas every function costs, function by function. Run it before and after any optimization from Chapter 8 to prove the change actually helped.

7.5 Cast — Talking to Chains from the Command Line

`cast` is how you poke at a live or local chain without writing a single line of script — reading state, sending transactions, and converting units, all from the terminal.

Command	What it does
cast call <addr> "balanceOf(address)" <user>	Read a view function — no gas, no transaction
cast send <addr> "transfer(address,uint256)" <to> <amt>	Send a state-changing transaction
cast balance <address>	Check an address's ETH balance
cast to-wei 1 ether	Convert a human-readable amount to wei
cast sig "transfer(address,uint256)"	Get a function's 4-byte selector

7.6 Coming From Hardhat? A Few Adjustments

- Dependencies install with forge install user/repo, creating a git submodule under lib/ — not npm install, and there's no package.json driving anything.
- Remappings (so import "@openzeppelin/..." resolves correctly) go in remappings.txt or foundry.toml, generated automatically by forge install in recent versions.
- There's no describe/it nesting — every test is just a public function on a contract, named test_WhatItChecks by convention.
- console.log works inside contracts during tests via import "forge-std/console.sol" — genuinely useful for debugging a failing test's intermediate values.

□ TRY IT YOURSELF — Write a Second Test

Add a test_RevertWhen_ResetCalledByOwnerSucceeds — wait, name it test_OwnerCanReset — that proves the OPPOSITE of the existing revert test: prank as the actual deployer (the test contract itself, so no prank needed), call reset() after incrementing a few times, and assert the count is back to 0.

Hint: You don't need vm.prank for this one — inside a test function, msg.sender is already the test contract, which is exactly who deployed SimpleCounter in setUp().

□ TRY IT YOURSELF — Read a Real Contract with Cast

Using cast call against a mainnet RPC endpoint (or Sepolia testnet), read the totalSupply() of any real deployed ERC-20 token you know the address of. You'll need to pass --rpc-url pointing at a provider (Alchemy, Infura, or a public endpoint).

Hint: The command shape is: cast call <token_address> "totalSupply()" --rpc-url <your_rpc_url> — the result comes back as a raw hex value; cast --to-dec converts it to a readable number.

KEY TAKEAWAYS

- Foundry tests are Solidity, not JavaScript — no context-switching, and fuzzing/gas reporting are built in rather than bolted on.
- forge builds and tests, cast talks to any chain from the command line, anvil runs a local node, and chisel is a Solidity REPL — all four install together.
- forge init scaffolds src/, test/, script/, and lib/ — dependencies are added with forge install, not npm.
- vm.prank(address) makes the next call appear to come from a specific address — essential for testing access control like Chapter 6 covers.
- forge test --gas-report turns "this should be cheaper" into a number you can actually check before and after a change.

- Coming from Hardhat: dependencies are git submodules via forge install, not npm packages, and console.log needs an explicit forge-std import inside the contract itself.

□ DID YOU KNOW?

Foundry was originally built by the team behind Paradigm, a crypto investment firm, and its Rust foundation is exactly why it's dramatically faster than JS-based tooling — running the actual EVM natively instead of going through a JavaScript-to-EVM bridge for every single test call. The name "Foundry" and its tool names (forge, cast, anvil, chisel) all follow a consistent blacksmithing metaphor — literally, tools for forging contracts.

CHAPTER 8

Gas Optimization

Every technique in this chapter assumes Chapters 1–7 already happened — correctness and security come first, always. Gas optimization is a discipline you apply to code that already works and has already been reviewed, not a substitute for either. And it's measured, not guessed: run `forge test --gas-report` (Chapter 7) before and after every change in this chapter, on your own contracts, to confirm it actually helped.

> **i INFO** The gas figures in this chapter are approximate, illustrative numbers based on well-established EVM cost constants — actual costs shift slightly with Solidity version, optimizer settings, and whether a storage slot was already "warm" earlier in the same transaction. Treat every number here as directionally correct, not a guarantee, and verify against your own gas report.

Operation	Approximate cost
SLOAD (cold — first access this tx)	~2,100 gas
SLOAD (warm — same slot, already accessed)	~100 gas
SSTORE (zero → non-zero)	~20,000 gas
Deployed bytecode	~200 gas per byte
Reading a local variable / stack value	~3 gas

8.1 Storage Packing

A storage slot is always 32 bytes, whether you use all of it or not. Solidity packs consecutive smaller-than-32-byte variables into the same slot automatically — but only if you declare them next to each other, and only if they actually fit together.

```
// BEFORE — three storage slots
contract Unpacked {
    uint256 public lastUpdated;   // slot 0 — full 32 bytes used
    uint256 public balance;       // slot 1 — full 32 bytes used
    bool public isActive;         // slot 2 — 1 byte used, 31 wasted
}
```

```
// AFTER — one storage slot instead of three
contract Packed {
    uint128 public balance;       // slot 0, bytes 0-15
    uint96 public lastUpdated;    // slot 0, bytes 16-27
    bool public isActive;         // slot 0, byte 28 — all three fit together
}
```

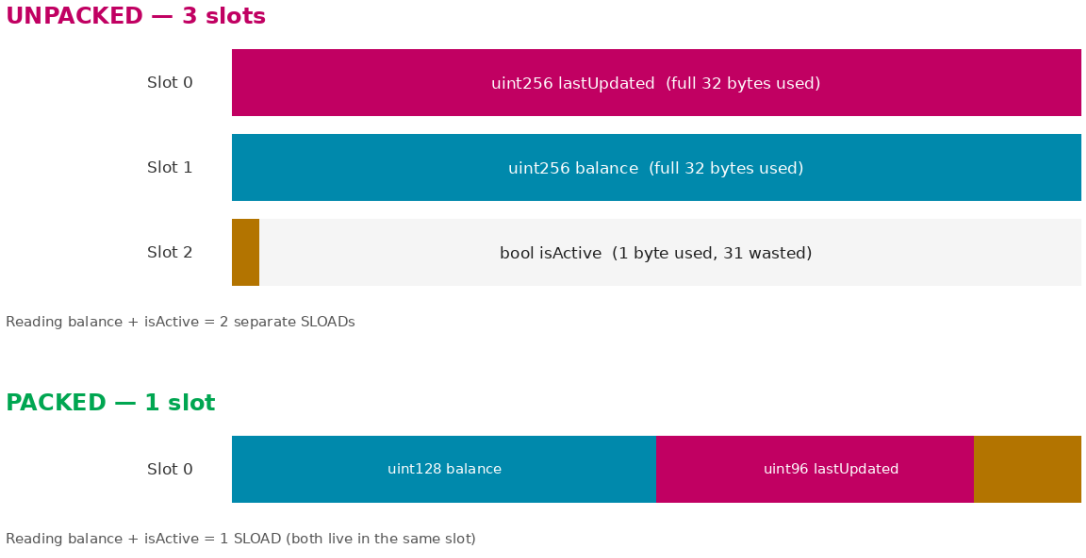

Figure 8.1 — Three variables in three slots versus the same three variables sharing one

GAS TIP The reliable, guaranteed saving from packing is on reads: fetching balance and isActive from Unpacked costs two separate SLOADs; fetching the same two values from Packed costs exactly one, since both now live in the same slot.

⚡ PITFALL Packing only helps if the smaller types are declared adjacently — inserting an unrelated uint256 between two packable fields breaks the packing entirely, silently, with no compiler warning.

8.2 Custom Errors, Quantified

Chapters 2 and 6 established custom errors as the default over require strings. Here's specifically why they're cheaper:

```
// A require string is encoded into deployment bytecode in full, every
// time it appears — at roughly 200 gas per byte deployed.
require(balance >= amount, "Insufficient balance for withdrawal");  // 36 chars ≈
7,200+ gas to deploy

// A custom error's identifier is a 4-byte selector — a fixed, small cost
// no matter how descriptive the name is.
error InsufficientBalance(uint256 available, uint256 required);
if (balance < amount) revert InsufficientBalance(balance, amount);
```

8.3 immutable and constant

A regular storage variable costs a real SLOAD every time your code reads it — even if it never changes after the constructor runs. immutable variables skip storage entirely: the value is written directly into the deployed bytecode at deploy time.

```
contract WithStorage {
    address public owner;                // stored in storage — costs an SLOAD
every read
    constructor() { owner = msg.sender; }
    function checkOwner() external view returns (bool) {
        return msg.sender == owner;      // ~2,100 gas (cold) or ~100 gas (warm)
just to read it
    }
}

contract WithImmutable {
    address public immutable owner;      // embedded directly into deployed
bytecode
    constructor() { owner = msg.sender; }
    function checkOwner() external view returns (bool) {
        return msg.sender == owner;      // no SLOAD at all — the value is baked
into the code
    }
}
```

8.4 unchecked Blocks in Loops

Chapter 6 introduced unchecked{} as something to use only where overflow is provably impossible. A loop counter bounded by array length is the textbook example — it cannot exceed the array's length, so the overflow check on every increment is pure overhead.

```
// BEFORE — every iteration re-checks for overflow on i++
function sumBefore(uint256[] calldata values) external pure returns (uint256
total) {
    for (uint256 i = 0; i < values.length; i++) {
        total += values[i];
```

```
    }
}

// AFTER — the increment can't overflow (loop bound guarantees it), so skip the
check
function sumAfter(uint256[] calldata values) external pure returns (uint256 total)
{
    uint256 len = values.length;
    for (uint256 i = 0; i < len;) {
        total += values[i];
        unchecked { i++; }
    }
}
```

⚡ **PITFALL** This is safe specifically because len is fixed and i is bounded by the loop condition — never wrap arithmetic on values that come from user input or external calls in unchecked{} without separately proving the bound yourself.

8.5 Caching Repeated Storage Reads

Reading the same storage slot more than once in a single function is pure waste — read it once into a local variable, and every subsequent use is a near-free stack read instead of another SLOAD.

```
uint256 public balance;

// BEFORE — reads storage three separate times
function bad() external view returns (uint256) {
    return balance + balance + balance;
}

// AFTER — reads storage once, reuses the value from the stack
function good() external view returns (uint256) {
    uint256 cached = balance;
    return cached + cached + cached;
}
```

8.6 Putting It Together

Technique	What it does	Why it saves gas
Storage packing	Fits related fields into one 32-byte slot	One SLOAD instead of several, when reading multiple fields together
Custom errors	4-byte selector instead of a full string	Smaller deployed bytecode; cheaper to revert with
immutable / constant	Value baked into bytecode, not storage	Zero SLOADs — the read cost disappears entirely
unchecked loop counters	Skips the overflow check where it's impossible	Small saving per iteration, compounds over large loops
Caching storage reads	Read once into a local variable, reuse it	Every reuse costs ~3 gas instead of another ~100 gas SLOAD

□ TRY IT YOURSELF — Pack a Real Struct

Here's an unpacked struct: struct Position { uint256 amount; uint256 openedAt; bool isLong; address trader; }. Reorder and resize its fields so amount and openedAt fit as uint128 each (still enough range for realistic use), and isLong packs alongside them — aim for 2 slots total instead of 4.

Hint: address is 20 bytes and needs its own slot regardless (nothing else fits alongside a full address in most layouts) — focus on packing amount, openedAt, and isLong together in the remaining slot.

□ TRY IT YOURSELF — Measure It Yourself

Take the Unpacked and Packed contracts from section 8.1, write a Foundry test for each that sets all three fields, and run forge test --gas-report. Compare the actual numbers you get against this chapter's claims about read costs.

Hint: You'll need forge-std's Test contract (Chapter 7) and a setUp() that deploys both contracts, then two test functions that write to and read from each.

KEY TAKEAWAYS

- Gas optimization comes after correctness and security, applied to reviewed code — never the other way around.
- Storage packing fits multiple small fields into one 32-byte slot, turning several SLOADs into one when you read them together.
- Custom errors are smaller in deployed bytecode than require strings and cheaper to revert with, regardless of how long the message would have been.
- immutable and constant values are baked into bytecode at deploy time — reading them costs no SLOAD at all.
- unchecked{} around a loop counter is safe only when the bound is mathematically guaranteed elsewhere — never on values derived from user input.
- Cache a storage read you'll need more than once in a local variable; every reuse after the first is a near-free stack read instead of another SLOAD.
- Always verify a gas optimization with forge test --gas-report on your own contract — the numbers in this chapter are illustrative, not a substitute for measuring.

□ DID YOU KNOW?

Solidity's storage packing optimization isn't automatic guesswork by the compiler — it's a deterministic layout algorithm that assigns byte offsets in the exact declaration order you write. This is also exactly why OpenZeppelin's own contracts are carefully ordered internally, and why adding a new state variable to an existing contract in the wrong position is such a common way to accidentally break packing (or, for upgradeable contracts from Chapter 3, corrupt storage entirely).

CHAPTER 9

Advanced Topics

This chapter is a survey, not a deep dive — each of these is a full specification with its own audited reference implementations. The goal here is recognition: know what each one is for, how it fits into the ecosystem, and where to look further when a project actually calls for it.

9.1 L2 Deployment Considerations

Deploying to Arbitrum, Optimism, or Base is not simply "the same contract, cheaper." A handful of practical differences matter every time:

Consideration	What changes vs. L1
Gas costs	Much lower than L1, but non-zero — cost is driven mainly by L1 data-posting, not L2 execution
Block time / finality	Faster soft confirmations; optimistic rollups add a withdrawal challenge period back to L1
Contract verification	Each L2's own explorer (Arbiscan, Optimistic Etherscan, Basescan) needs separate verification
RPC endpoints	Distinct per network — a script hardcoded to one L2's RPC silently fails (or worse, succeeds) elsewhere
Precompiles / opcodes	Most are EVM-equivalent, but always check the specific L2's documented differences before assuming 1:1 parity

9.2 Account Abstraction — ERC-4337

ERC-4337 lets a smart contract, not a private key, be the thing that authorizes transactions — without any change to the Ethereum protocol itself. A new mempool of "UserOperations" gets bundled and submitted through a singleton EntryPoint contract.

```
struct UserOperation {
    address sender;
    uint256 nonce;
    bytes initCode;
    bytes callData;
    uint256 callGasLimit;
    uint256 verificationGasLimit;
    uint256 preVerificationGas;
    uint256 maxFeePerGas;
    uint256 maxPriorityFeePerGas;
    bytes paymasterAndData;
    bytes signature;
}

interface IAccount {
    // The EntryPoint calls this on YOUR smart account before executing
    // anything — this is where custom signature schemes, session keys,
    // or social recovery logic actually lives.
    function validateUserOp(
        UserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external returns (uint256 validationData);
```

```
}
```

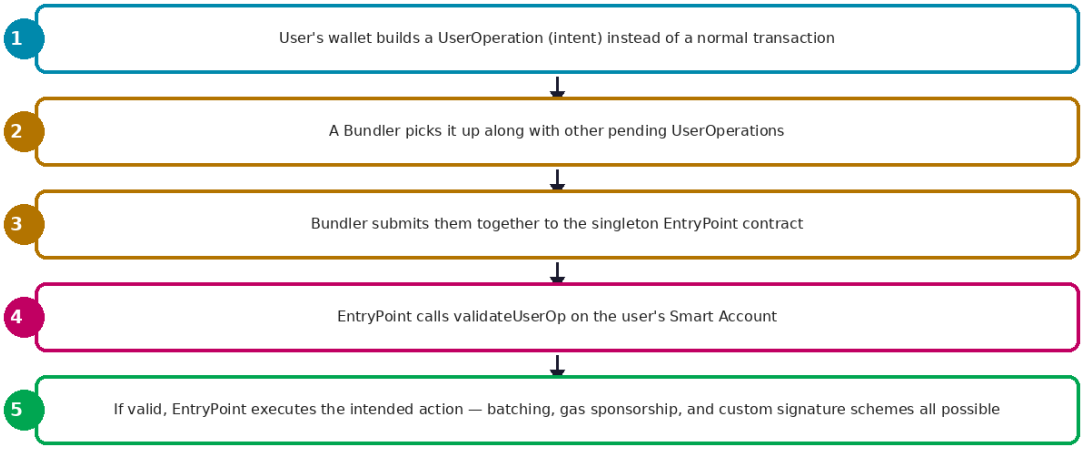

Figure 9.1 — How a UserOperation reaches your smart account

9.3 EIP-7702 — Upgrading Existing Wallets

ERC-4337 requires migrating to a new smart account address. EIP-7702, live since the Pectra upgrade, takes a different approach: it lets an existing EOA (a plain private-key wallet) temporarily run smart contract code at its own address.

```
// EIP-7702 introduces a new transaction type (0x04) carrying an
// authorization list: [chainId, contractAddress, nonce, signature].
//
// Once authorized, calls TO THE EOA'S OWN ADDRESS execute the
// authorized contract's code — with the EOA's real address as
// msg.sender the whole time. No new address, no migration, no asset
// transfer. A plain wallet gains batching, gas sponsorship, and
// spending limits for the duration of the authorization.
```

> **i INFO** 4337 and 7702 aren't competitors — they're increasingly used together. A 7702-authorized EOA can point at a 4337-compatible implementation, getting account abstraction's features without ever leaving its original address.

9.4 ERC-4626 — Tokenized Vaults

Before ERC-4626, every lending protocol and yield vault invented its own share-accounting scheme, and no two were interchangeable. ERC-4626 standardizes it: deposit an asset, receive shares; redeem shares, receive your proportional claim on whatever the vault holds now.

```
interface IERC4626 {
    function asset() external view returns (address);
    function totalAssets() external view returns (uint256);
    function convertToShares(uint256 assets) external view returns (uint256);
    function convertToAssets(uint256 shares) external view returns (uint256);
    function deposit(uint256 assets, address receiver) external returns (uint256
shares);
    function withdraw(uint256 assets, address receiver, address owner) external
returns (uint256 shares);
}

contract SimpleVault {
    IERC20 public immutable asset;
    uint256 public totalShares;
```

```
    uint256 public totalAssetsHeld;
    mapping(address => uint256) public sharesOf;

    constructor(address _asset) { asset = IERC20(_asset); }

    function deposit(uint256 assets) external returns (uint256 shares) {
        shares = totalShares == 0 ? assets : (assets * totalShares) /
totalAssetsHeld;
        asset.transferFrom(msg.sender, address(this), assets);
        totalAssetsHeld += assets;
        totalShares += shares;
        sharesOf[msg.sender] += shares;
    }

    function withdraw(uint256 shares) external returns (uint256 assets) {
        assets = (shares * totalAssetsHeld) / totalShares;
        sharesOf[msg.sender] -= shares;
        totalShares -= shares;
        totalAssetsHeld -= assets;
        asset.transfer(msg.sender, assets);
    }
}
```

GAS TIP This is exactly the share-accounting math Chapter 5's lending concepts were built on top of — ERC-4626 is the standardized, composable version of the same idea, which is why aggregators and other protocols can plug into ANY 4626 vault without custom integration work.

9.5 ERC-6551 — Token Bound Accounts

An NFT normally just points at metadata. ERC-6551 gives every individual NFT its own smart contract account — deterministically computed, no deployment needed until it's actually used.

```
interface IERC6551Registry {
    // Every (implementation, chainId, tokenContract, tokenId) combination
    // deterministically produces the SAME account address, computed the
    // same way every time — no need to store it anywhere.
    function createAccount(
        address implementation,
        bytes32 salt,
        uint256 chainId,
        address tokenContract,
        uint256 tokenId
    ) external returns (address account);
}

// The NFT itself becomes the "owner" of that account and everything
// inside it — trade the NFT, and every asset its token-bound account
// holds goes with it, without a separate transfer.
```

9.6 ERC-7683 — Cross-Chain Intents

Traditional cross-chain bridging asks the user to specify the exact mechanical path: bridge contract, target chain, gas on the other side. Intents flip this: the user states the outcome they want, and lets competition sort out how.

```
struct CrossChainOrder {
    address settlementContract;
    address swapper;
    uint256 nonce;
    uint32 originChainId;
```

```
    uint32 fillDeadline;
    bytes orderData;   // encodes the desired OUTCOME, not the exact path
}

// A user signs an intent once. Independent "solvers" compete to fill it
// however is cheapest or fastest — bridging, swapping, routing through
// whatever liquidity they have access to — and are reimbursed on the
// origin chain only once the fill is proven to have happened.
```

9.7 ERC-3643 — Permissioned Tokens for Real-World Assets

A tokenized bond or share has legal obligations an ordinary ERC-20 has no way to express — who's allowed to hold it, whether a transfer needs a compliance check first. ERC-3643 builds that gating directly into the transfer path.

```
interface IERC3643 {
    // Every transfer checks this BEFORE moving any tokens — is the
    // recipient KYC'd? Is this jurisdiction permitted? Is a lockup over?
    function canTransfer(address from, address to, uint256 amount) external view
returns (bool);
    // ...the rest of the interface mirrors ERC-20, but transfer() and
    // transferFrom() both call canTransfer() internally and revert if it fails
}
```

> **⚠ SECURITY** This is the opposite security model from everything else in this book: an ERC-3643 token deliberately restricts who can hold or receive it. Getting canTransfer wrong in either direction — too permissive, breaking compliance, or too restrictive, freezing legitimate transfers — is the central risk specific to this standard.

9.8 ERC-2612 — Permit (Gasless Approvals)

Every ERC-20 interaction in Chapter 4 assumed a separate approve() transaction before the real action. Permit collapses that into one.

```
interface IERC2612 {
    function permit(
        address owner, address spender, uint256 value, uint256 deadline,
        uint8 v, bytes32 r, bytes32 s
    ) external;
    function nonces(address owner) external view returns (uint256);
}

// Normally: approve() (one transaction) THEN the actual action (a second
// transaction) — two transactions, two gas payments, from the user.
// With permit(): the user signs an EIP-712 message off-chain (free, no
// tx), and whoever is acting on their behalf submits that signature
// alongside the real action — approval and action happen in ONE transaction.
```

9.9 ERC-8004 — Trustless Agent Identity

As AI agents increasingly act autonomously on-chain — trading, negotiating, hiring other agents — they need the same primitives humans take for granted: a persistent identity, a track record, and a way to prove specific work was actually done. ERC-8004, live on Ethereum mainnet since January 2026, defines exactly that, as three separate registries:

```
interface IIdentityRegistry {
    // Registering an agent mints it an ERC-721 — the agent's identity
    // IS a token, transferable and ownable like any other NFT.
```

```
    function register(string calldata agentURI) external returns (uint256
agentId);
    function resolve(uint256 agentId) external view returns (address owner, string
memory agentURI);
}

interface IReputationRegistry {
    function submitFeedback(uint256 agentId, uint8 score, string calldata
evidenceURI) external;
    function getReputation(uint256 agentId) external view returns (uint256
totalScore, uint256 feedbackCount);
}

interface IValidationRegistry {
    function requestValidation(uint256 agentId, bytes32 taskHash) external returns
(uint256 requestId);
    function submitValidation(uint256 requestId, bool passed, string calldata
proofURI) external;
}
```

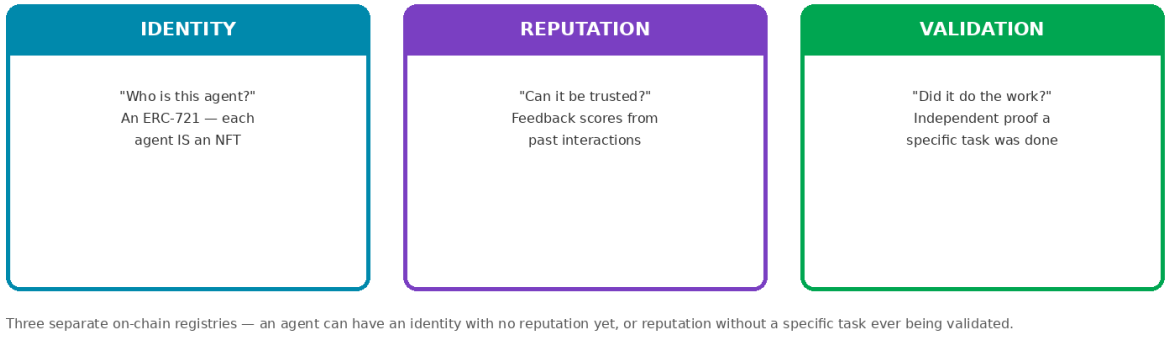

Figure 9.2 — Identity, Reputation, and Validation are deliberately independent concerns

> **i INFO** This standard is genuinely new ground — the author's own published research (see the About the Author section) works directly in this space, on coupling agent identity with cross-chain payment settlement.

9.10 x402 — Agent Payments Protocol

HTTP has had a status code reserved for payment since the 1990s — 402 Payment Required — and it sat unused for thirty years. x402 finally gives it a job: letting an API demand payment inline, in the HTTP response itself, without accounts, API keys, or subscriptions.

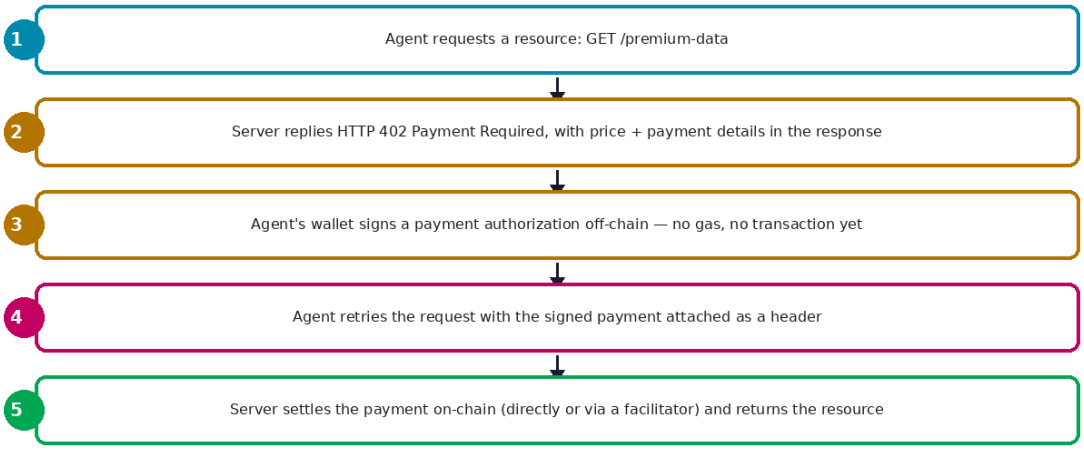

Figure 9.3 — A 402 response carries everything needed to pay and retry, no account required

This matters specifically for autonomous agents: a human can't realistically sign up for a subscription mid-task, but an agent's wallet can sign a stablecoin payment authorization in milliseconds and retry the request immediately — machine-speed micropayments, settled on-chain.

9.11 Production Deployment Checklist

Before anything in this chapter — or this book — reaches mainnet with real funds at risk:

Checklist item	What it means in practice
Testnet first	Full deployment and usage flow tested on a public testnet before mainnet
Contracts verified	Source verified on every relevant block explorer — users should never have to trust bytecode alone
Admin keys secured	Multisig, not a single EOA, controls anything privileged
Timelock on upgrades	Any upgrade or privileged parameter change has a public delay before it takes effect
Monitoring live	Alerts on unusual transaction volume, large withdrawals, or price deviations before launch, not after
Bug bounty live	A funded bug bounty (Chapter 6) is active before mainnet users have real funds at risk
Gradual rollout	Deposit or exposure caps for the first period, raised only after the code has survived real usage

□ TRY IT YOURSELF — Trace a UserOperation

Using Figure 9.1, write out in your own words what happens if step 4 (validateUserOp) returns a failure — where does execution stop, and who (if anyone) pays gas for the attempt?

Hint: The EntryPoint contract is what calls validateUserOp — think about what a caller does when a function it calls indicates failure, and who initiated that outer call in the first place.

TRY IT YOURSELF — Design a Registry Interaction

Sketch (in comments or pseudocode, not full Solidity) how an agent marketplace might use all three ERC-8004 registries together before accepting a bid from an unfamiliar agent: what would it check, and in what order?

Hint: Think about which check should short-circuit the others — is there a reputation or validation check worth running at all if the agent's identity can't even be resolved?

KEY TAKEAWAYS

- L2 deployment differs from L1 in gas structure, finality, and verification — never assume 1:1 parity without checking.
- ERC-4337 routes transactions through a UserOperation mempool and a singleton EntryPoint, letting a smart contract itself authorize execution.
- EIP-7702 lets an existing EOA temporarily run smart contract code at its own address — no migration, and increasingly paired with 4337.
- ERC-4626 standardizes vault share accounting so aggregators and other protocols can integrate with any compliant vault without custom code.
- ERC-6551 gives individual NFTs their own deterministic smart contract accounts, so assets move with the NFT automatically.
- ERC-7683 lets users sign intents describing a desired outcome, leaving competing solvers to find the cheapest path — instead of manually bridging.
- ERC-3643 builds compliance checks directly into token transfers, the standard of choice for tokenized real-world assets.
- ERC-2612's permit() collapses approve-then-act into a single transaction via an off-chain signature.
- ERC-8004 gives autonomous agents identity, reputation, and validation as three deliberately separate on-chain registries.
- x402 uses HTTP's long-dormant 402 status code to let APIs demand and receive stablecoin payment inline, without accounts — built for machine-speed agent payments.

DID YOU KNOW?

HTTP 402 Payment Required has existed in the HTTP specification since 1997, reserved from the very beginning "for future use" — nearly three decades passed before x402 gave it an actual, working implementation in 2025. It's one of the longest gaps between a protocol reserving a feature and someone actually shipping it.

CHAPTER 10

Real World Projects: MultiSig Wallet

Everything so far has been one concept at a time. This chapter is different: one complete, real project, requiring several chapters' worth of ideas working together — access control, custom errors, events, and Checks-Effects-Interactions, all in a contract genuinely worth deploying (after the additions section 10.6 mentions).

10.1 The Complete Contract

An M-of-N multisig: N owners are configured at deployment, and any transaction needs at least M of them to confirm before it can execute. Read it once end to end — every section after this walks through one part of it.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract MultiSigWallet {
    event Deposit(address indexed sender, uint256 amount, uint256 balance);
    event SubmitTransaction(address indexed owner, uint256 indexed txIndex,
address indexed to, uint256 value, bytes data);
    event ConfirmTransaction(address indexed owner, uint256 indexed txIndex);
    event RevokeConfirmation(address indexed owner, uint256 indexed txIndex);
    event ExecuteTransaction(address indexed owner, uint256 indexed txIndex);

    error NotOwner(address caller);
    error TxDoesNotExist(uint256 txIndex);
    error TxAlreadyExecuted(uint256 txIndex);
    error TxAlreadyConfirmed(uint256 txIndex, address owner);
    error TxNotConfirmed(uint256 txIndex, address owner);
    error InsufficientConfirmations(uint256 have, uint256 need);
    error ExecutionFailed(uint256 txIndex);
    error InvalidOwnerCount();
    error InvalidThreshold();
    error DuplicateOrZeroOwner(address owner);

    address[] public owners;
    mapping(address => bool) public isOwner;
    uint256 public required;

    struct Transaction {
        address to;
        uint256 value;
        bytes data;
        bool executed;
        uint256 confirmations;
    }

    Transaction[] public transactions;
    mapping(uint256 => mapping(address => bool)) public isConfirmed;

    modifier onlyOwner() {
        if (!isOwner[msg.sender]) revert NotOwner(msg.sender);
        _;
    }

    modifier txExists(uint256 txIndex) {
        if (txIndex >= transactions.length) revert TxDoesNotExist(txIndex);
        _;
    }
```

```
    modifier notExecuted(uint256 txIndex) {
        if (transactions[txIndex].executed) revert TxAlreadyExecuted(txIndex);
        _;
    }

    constructor(address[] memory _owners, uint256 _required) {
        if (_owners.length == 0) revert InvalidOwnerCount();
        if (_required == 0 || _required > _owners.length) revert
InvalidThreshold();

        for (uint256 i = 0; i < _owners.length; i++) {
            address owner = _owners[i];
            if (owner == address(0) || isOwner[owner]) revert
DuplicateOrZeroOwner(owner);
            isOwner[owner] = true;
            owners.push(owner);
        }
        required = _required;
    }

    receive() external payable {
        emit Deposit(msg.sender, msg.value, address(this).balance);
    }

    function submitTransaction(address to, uint256 value, bytes calldata data)
        external onlyOwner returns (uint256 txIndex)
    {
        txIndex = transactions.length;
        transactions.push(Transaction({ to: to, value: value, data: data,
executed: false, confirmations: 0 }));
        emit SubmitTransaction(msg.sender, txIndex, to, value, data);
    }

    function confirmTransaction(uint256 txIndex) external onlyOwner
txExists(txIndex) notExecuted(txIndex) {
        if (isConfirmed[txIndex][msg.sender]) revert TxAlreadyConfirmed(txIndex,
msg.sender);

        transactions[txIndex].confirmations += 1;
        isConfirmed[txIndex][msg.sender] = true;
        emit ConfirmTransaction(msg.sender, txIndex);
    }

    function revokeConfirmation(uint256 txIndex) external onlyOwner
txExists(txIndex) notExecuted(txIndex) {
        if (!isConfirmed[txIndex][msg.sender]) revert TxNotConfirmed(txIndex,
msg.sender);

        transactions[txIndex].confirmations -= 1;
        isConfirmed[txIndex][msg.sender] = false;
        emit RevokeConfirmation(msg.sender, txIndex);
    }

    function executeTransaction(uint256 txIndex) external onlyOwner
txExists(txIndex) notExecuted(txIndex) {
        Transaction storage transaction = transactions[txIndex];
        if (transaction.confirmations < required) {
            revert InsufficientConfirmations(transaction.confirmations, required);
        }

        transaction.executed = true;   // Checks-Effects-Interactions: state first
```

```
        (bool success, ) = transaction.to.call{value:
transaction.value}(transaction.data);
        if (!success) revert ExecutionFailed(txIndex);

        emit ExecuteTransaction(msg.sender, txIndex);
    }

    function getOwners() external view returns (address[] memory) {
        return owners;
    }

    function getTransactionCount() external view returns (uint256) {
        return transactions.length;
    }
}
```

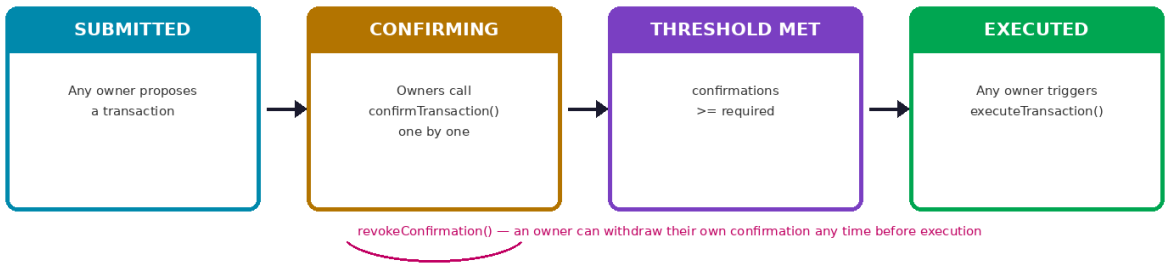

Figure 10.1 — A transaction's life from proposal to execution

10.2 The Constructor: Validating Owners and Threshold

Three things get checked once, at deployment, and never again: there's at least one owner, the required threshold is achievable (between 1 and the owner count), and no owner address is duplicated or zero. Getting any of these wrong at deploy time would be permanent — there's no fixing a misconfigured owner list after the fact without deploying an entirely new wallet.

10.3 Submitting and Confirming

submitTransaction doesn't move any funds — it just records what an owner is proposing, as a Transaction struct with zero confirmations. Each subsequent confirmTransaction call from a different owner increments that count by exactly one, and isConfirmed tracks who specifically has already confirmed so the same owner can't inflate the count by confirming twice.

> ⚡ **PITFALL** revokeConfirmation deliberately has no separate check beyond "this owner confirmed it and it isn't executed yet" — an owner can change their mind about a pending transaction for any reason, at any time, right up until execution. That's a feature: it's exactly what stops a compromised or hasty confirmation from being irreversible.

10.4 Executing — Checks-Effects-Interactions in Practice

executeTransaction is where Chapter 3 and Chapter 6's central lesson earns its keep. Notice the order: confirmations are checked against required first (Checks), transaction.executed is set to true next (Effects), and only after that does the actual external call happen (Interactions). If call() somehow reentered this same function, executed is already true, and notExecuted's modifier check stops it cold — this contract is reentrancy-safe by construction, not by adding a separate guard on top.

10.5 Testing the Full Lifecycle

A contract this stateful deserves tests that walk through real multi-step scenarios, not just single function calls in isolation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test} from "forge-std/Test.sol";
import {MultiSigWallet} from "../src/MultiSigWallet.sol";

contract MultiSigWalletTest is Test {
    MultiSigWallet public wallet;
    address owner1 = address(0x1);
    address owner2 = address(0x2);
    address owner3 = address(0x3);

    function setUp() public {
        address[] memory owners = new address[](3);
        owners[0] = owner1;
        owners[1] = owner2;
        owners[2] = owner3;
        wallet = new MultiSigWallet(owners, 2);   // 2-of-3
        vm.deal(address(wallet), 10 ether);
    }

    function test_ExecutesAfterThresholdReached() public {
        vm.prank(owner1);
        uint256 txIndex = wallet.submitTransaction(address(0xBEEF), 1 ether, "");

        vm.prank(owner1);
        wallet.confirmTransaction(txIndex);
        vm.prank(owner2);
        wallet.confirmTransaction(txIndex);

        vm.prank(owner1);
        wallet.executeTransaction(txIndex);

        assertEq(address(0xBEEF).balance, 1 ether);
    }

    function test_RevertWhen_ExecutingBelowThreshold() public {
        vm.prank(owner1);
        uint256 txIndex = wallet.submitTransaction(address(0xBEEF), 1 ether, "");

        vm.prank(owner1);
        wallet.confirmTransaction(txIndex);

        vm.prank(owner1);
        vm.expectRevert();
        wallet.executeTransaction(txIndex);
    }

    function test_RevokedConfirmationNoLongerCounts() public {
        vm.prank(owner1);
        uint256 txIndex = wallet.submitTransaction(address(0xBEEF), 1 ether, "");

        vm.prank(owner1);
        wallet.confirmTransaction(txIndex);
        vm.prank(owner2);
        wallet.confirmTransaction(txIndex);
```

```
        vm.prank(owner2);
        wallet.revokeConfirmation(txIndex);    // back down to 1 confirmation

        vm.prank(owner1);
        vm.expectRevert();
        wallet.executeTransaction(txIndex);
    }
}
```

GAS TIP test_RevokedConfirmationNoLongerCounts exists specifically because "confirmations went up, then down, then someone tried to execute anyway" is exactly the kind of multi-step interaction a single unit test per function would never catch. Foundry's speed (Chapter 7) makes tests like this cheap enough to write generously.

10.6 What Production MultiSigs Add

This contract is genuinely sound, but Safe (formerly Gnosis Safe) — the multisig standard most real treasuries actually use — adds several things deliberately left out here to keep the core logic legible: modular ownership changes (adding/removing owners without redeploying), transaction batching, gas refunds for the executor, support for smart contract signatures (so a multisig can itself be an owner of another multisig), and a well-audited, years-battle-tested codebase securing billions of dollars today. Understanding this chapter's version is what makes reading Safe's actual source code approachable instead of overwhelming.

□ TRY IT YOURSELF — Add Owner Removal

Add a removeOwner(address ownerToRemove) function, callable only through the multisig's own execution flow (i.e., it must itself be submitted, confirmed, and executed like any other transaction — not a direct onlyOwner function). Update isOwner, remove the address from the owners array, and lower required if it would otherwise exceed the new owner count.

Hint: Since this must go through the normal transaction flow, the multisig would call this function on ITSELF — meaning "to" in submitTransaction would be address(this), and "data" would be the encoded removeOwner call.

□ TRY IT YOURSELF — Test the Reentrancy Guarantee

Write a Foundry test that deploys a malicious contract whose receive() function tries to call back into executeTransaction for the same txIndex. Confirm the reentrant call reverts because notExecuted correctly sees executed already set to true.

Hint: You'll need a small attacker contract with a receive() function that calls back into the wallet — vm.expectRevert() around the outer executeTransaction call confirms the whole thing reverts cleanly rather than double-spending.

KEY TAKEAWAYS

- An M-of-N multisig requires at least M confirmations from N configured owners before any transaction executes — configured once, permanently, at deployment.
- submitTransaction only records a proposal; no funds move until a separate executeTransaction call succeeds after enough confirmations.
- isConfirmed[txIndex][owner] exists specifically to stop the same owner inflating the confirmation count by confirming more than once.
- revokeConfirmation lets any owner withdraw their own confirmation any time before execution — an intentional safety valve, not an edge case to prevent.
- executeTransaction sets executed = true before making the external call — Checks-Effects-Interactions applied directly, making this contract reentrancy-safe by construction.
- Production multisigs like Safe add modular ownership, batching, and years of audits on top of this same core pattern — understand this chapter before reading that code.

□ DID YOU KNOW?

Gnosis Safe (now just "Safe") has secured well over $100 billion in assets at various points and is the multisig nearly every major DAO treasury, protocol team, and serious crypto fund uses to hold its own funds — including, almost certainly, funds held by teams whose contracts you've read about in earlier chapters of this book.

CHAPTER 11

Smart Contract Auditing

11.1 What an Audit Actually Is (and Isn't)

An audit is a time-boxed, structured review by people who did not write the code, looking specifically for the vulnerability classes Chapter 6 covered plus logic errors specific to what the protocol is trying to do. It is not a guarantee of no bugs — "audited" means "reviewed by qualified people within a defined scope," not "mathematically proven safe." Protocols with multiple audits from reputable firms have still been exploited; an audit reduces risk, it doesn't eliminate it.

11.2 The Audit Process

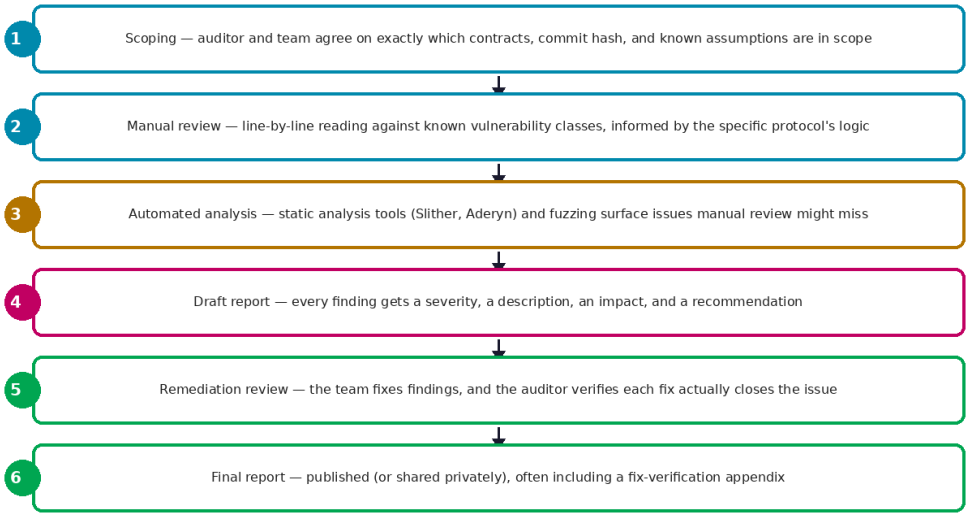

Figure 11.1 — What actually happens between "send us your code" and a published report

11.3 Severity Classification

Every real audit report classifies findings into a small number of severity tiers, so a reader can triage instantly without reading every word first.

Severity	What it means	Example from this book
Critical	Funds can be directly stolen or permanently locked, exploitable right now	Reentrancy draining a vault (Ch. 3); oracle manipulation enabling under-collateralized borrowing (Ch. 5)
High	Funds at risk, but only under specific, realistic conditions	Missing access control on a function that moves funds (Ch. 6)
Medium	Limited impact, or requires an unusual/costly precondition	Denial of service blocking one function without endangering funds directly (Ch. 6)
Low	Best-practice violation with minimal direct impact today	Using tx.origin where no current exploit path exists, but the pattern is fragile (Ch. 6)
Informational / Gas	Code quality or efficiency suggestions, no security impact	require strings instead of custom errors (Ch. 2); unpacked storage layout (Ch. 8)

> **i INFO** Severity is about realistic impact, not how interesting the bug is to find. A subtle, clever issue that can't actually be triggered profitably is Low; a boring, obvious missing access-control check that lets anyone drain the treasury is Critical.

11.4 Anatomy of a Finding

A real finding has a consistent shape — title, severity, exact location, description of the bug, impact if exploited, and a concrete recommendation. Here's one written for Chapter 3's reentrancy example, in the format an actual report would use:

```
Title: Reentrancy in withdraw() allows full drainage of contract funds

Severity: Critical
```

```
Location: VulnerableVault.sol, withdraw(), line 12

Description:
withdraw() sends ETH via a low-level call() before updating the caller's
balance to zero. A malicious contract's receive() function can call
withdraw() again before the first call returns, repeating this as many
times as the vault's ETH balance allows.

Impact:
An attacker can drain the entire contract balance in a single transaction,
regardless of their own deposited amount.

Recommendation:
Apply Checks-Effects-Interactions: set the caller's balance to zero BEFORE
making the external call, not after. Additionally, apply a nonReentrant
guard as defense in depth.
```

11.5 Static Analysis Tools

No serious audit is manual review alone — automated tools catch entire classes of mechanical mistakes faster than any human reading line by line.

Tool	What it's for	Typical invocation
Slither	Static analysis — fast, broad pattern-matching across known vulnerability classes	slither src/Contract.sol
Aderyn	Rust-based static analyzer from Cyfrin, built with Foundry projects in mind	aderyn .
Mythril	Symbolic execution — explores actual execution paths, slower but deeper than pattern matching	myth analyze src/Contract.sol
Foundry fuzzing	Property-based testing (Ch. 7) — throws random inputs at invariants until one breaks	forge test --fuzz-runs 10000

```
pip install slither-analyzer
slither src/VulnerableVault.sol
```

> ⚡ **PITFALL** Static analysis tools produce false positives constantly — a clean Slither run is a starting point, not a clean bill of health, and a flagged issue still needs a human to judge whether it's actually exploitable in context.

11.6 A Worked Example: Auditing VulnerableDistributor

Let's run the actual process from Figure 11.1 against a contract this book already showed you — Chapter 6's VulnerableDistributor. Treating it as a real engagement makes every earlier section concrete.

- Scoping: one contract, one function in question (distribute()), commit hash would be whatever the team submitted — for this exercise, the version printed in section 6.7.
- Manual review: distribute() loops over a caller-supplied recipients array and calls .transfer() on each one, inside a single loop, with no isolation between iterations.
- Automated analysis: Slither flags unbounded loops with external calls inside them as a known pattern — this would surface even without a human reading closely.
- Severity: one recipient that reverts (a contract with no payable fallback, or one that deliberately reverts) blocks the payout for every other recipient in the same call. Funds aren't stolen, but legitimate

users are denied access to funds intended for them — this lands as Medium, not Critical, using section 11.3's table.

- Recommendation: exactly what section 6.7 already showed — pull-over-push. Recipients withdraw their own share individually, so one failure can't block anyone else's payout.

> **i INFO** Notice this audit didn't discover anything new — it applied the same process from Figure 11.1 to a bug this book already explained. That's the actual point: methodology is what turns "I recognize this pattern" into a documented, defensible finding someone else can act on.

11.7 Learning Resources

Reading about vulnerabilities and finding them yourself, under time pressure, in code you've never seen, are very different skills. These are the platforms serious auditors actually train on:

Cyfrin Updraft — https://updraft.cyfrin.io

Ethernaut — https://ethernaut.openzeppelin.com

Damn Vulnerable DeFi — https://www.damnvulnerabledefi.xyz

Code4rena — https://code4rena.com

Sherlock — https://sherlock.xyz

Cyfrin Updraft is free, structured coursework, taught by working auditors. Ethernaut and Damn Vulnerable DeFi are hands-on capture-the-flag style challenges — Ethernaut for general contract vulnerabilities, Damn Vulnerable DeFi specifically for DeFi-flavored exploits (flash loans, oracle manipulation, governance attacks). Code4rena and Sherlock are competitive audit platforms where researchers compete for bounties finding real bugs in real, soon-to-launch protocols — the actual proving ground once the fundamentals feel solid.

□ TRY IT YOURSELF — Write a Finding

Chapter 6's VulnerableConfig (section 6.1) has a missing access-control check on setFee. Using section 11.4's format exactly, write a complete finding for it: Title, Severity, Location, Description, Impact, and Recommendation.

Hint: Compare your severity choice against section 11.3's table — does an attacker directly profit from calling setFee, or does it only indirectly enable something worse?

□ TRY IT YOURSELF — Run a Real Tool

Install Slither (pip install slither-analyzer) and run it against any contract from an earlier chapter of this book — VulnerableDistributor from Chapter 6 is a good candidate. Read through what it flags, and compare its findings against what this book already told you was wrong with that contract.

Hint: Slither needs the contract to compile first — run it from inside a Foundry project (forge init if you haven't already) with the file copied into src/.

KEY TAKEAWAYS

- An audit is a time-boxed structured review, not a guarantee — "audited" reduces risk, it doesn't eliminate it.
- The audit process runs scoping, manual review, automated analysis, a draft report, remediation review, and a final report — in that order.
- Severity classification (Critical/High/Medium/Low/Informational) is about realistic exploitability and impact, not how clever or subtle a bug is.
- A real finding always includes title, severity, location, description, impact, and a concrete recommendation.
- Static analysis tools like Slither and Aderyn catch mechanical patterns fast, but produce false positives — they inform manual judgment, they don't replace it.
- Ethernaut and Damn Vulnerable DeFi build hands-on skill; Code4rena and Sherlock are where that skill gets tested against real protocols for real bounties.
- Good methodology turns "I recognize this pattern" into a documented, defensible finding — that discipline matters as much as spotting the bug itself.

□ DID YOU KNOW?

Code4rena and Sherlock both structure their competitions as "the first researcher to report a unique bug gets full credit" — meaning dozens of skilled auditors race, in parallel, against the same codebase, and duplicate findings are common. It's not unusual for a single contest to receive more total scrutiny-hours from independent researchers than a traditional single-firm audit would, for a fraction of the cost.

Everything in this chapter assumes you already have working, deployable code to audit in the first place. Chapter 12 goes one level earlier: the fuzz testing, invariant testing, and fork testing that catch entire categories of these bugs automatically, before a human auditor — or an attacker — ever needs to find them by hand.

i INFO If you take away one habit from this chapter specifically: read real, public audit reports. Trail of Bits, OpenZeppelin, and Spearbit all publish theirs — reading how an actual finding is scoped, worded, and severity-rated against a real, unfamiliar codebase teaches judgment no checklist alone can.

CHAPTER 12

Testing with Foundry

Chapter 7 introduced Foundry testing at the level needed to follow along. This chapter goes to the level a real production contract actually needs — the gap between a tutorial's test suite and a protocol holding real funds is almost entirely the three techniques covered here.

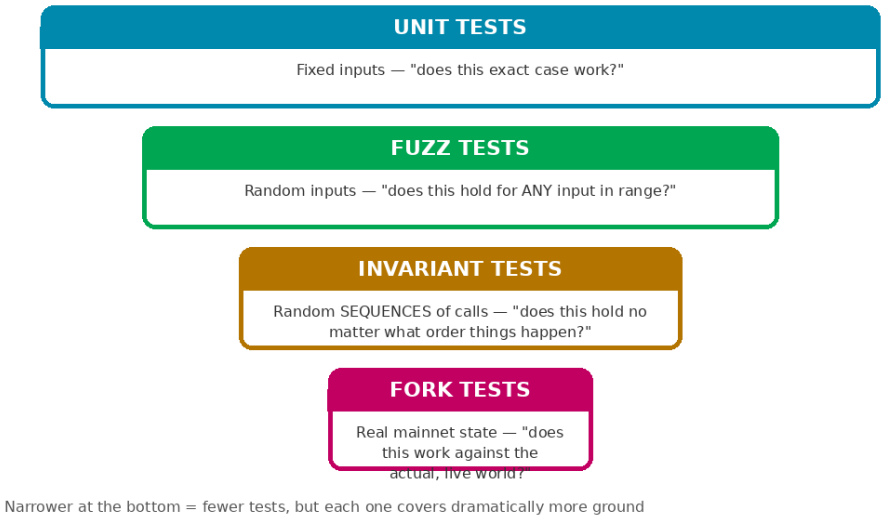

Figure 12.1 — Each layer trades test count for how much ground a single test covers

12.1 Unit Tests: What They Can't Catch

Every test in this book so far has been a unit test: fixed inputs, one specific scenario, one specific assertion. That's necessary but not sufficient — a unit test that checks swapAForB(100, 0) works correctly says nothing about swapAForB(1, 0) or swapAForB(999999999e18, 0). Edge cases you didn't think to write a test for are, by definition, invisible to unit tests.

12.2 Fuzz Testing

Give a test function a parameter instead of a hardcoded value, and Foundry runs it hundreds of times with different values automatically — by default searching for the specific input that breaks your assertion.

```
// forge-std's Test gives every function parameter a fuzzed value
// automatically — no special syntax needed beyond the parameter itself.
function testFuzz_SwapNeverOutputsMoreThanReserve(uint256 amountAIn) public {
    // bound() clamps the fuzzed value into a sane range instead of
    // rejecting out-of-range runs outright — far more efficient than vm.assume
    amountAIn = bound(amountAIn, 1, 1_000_000e18);

    tokenA.mint(address(this), amountAIn);
    tokenA.approve(address(amm), amountAIn);

    uint256 reserveBBefore = amm.reserveB();
    amm.swapAForB(amountAIn, 0);

    assertLt(amm.reserveB(), reserveBBefore);   // must decrease, never go
negative/wrap
}
```

> **GAS TIP** bound() is almost always the right choice over vm.assume() for constraining fuzzed ranges — vm.assume() discards out-of-range runs entirely (wasting them), while bound() maps any input into your target range, so every single run still contributes a real test case.

12.3 Invariant Testing

A fuzz test still only checks one function call in isolation. An invariant test goes further: Foundry calls a sequence of functions, in random order, with random arguments, and after every single call checks that a property you defined still holds — no matter what happened to get there.

```
// A "handler" wraps the real contract with bounded, valid-shaped calls —
// Foundry calls random sequences of the handler's functions, in random
// order, with random arguments, then checks every invariant still holds.
contract MultiSigHandler is Test {
    MultiSigWallet public wallet;

    constructor(MultiSigWallet _wallet) { wallet = _wallet; }

    function confirm(uint256 txIndexSeed, uint256 ownerSeed) external {
        uint256 txCount = wallet.getTransactionCount();
        if (txCount == 0) return;
        uint256 txIndex = txIndexSeed % txCount;

        address[] memory owners = wallet.getOwners();
        address owner = owners[ownerSeed % owners.length];

        vm.prank(owner);
        try wallet.confirmTransaction(txIndex) {} catch {}
    }
}

contract MultiSigInvariantTest is Test {
    MultiSigWallet public wallet;
    MultiSigHandler public handler;

    function setUp() public {
        // ...deploy wallet with 3 owners, required = 2...
        handler = new MultiSigHandler(wallet);
        targetContract(address(handler));
    }

    // This must hold no matter what sequence of confirms/revokes/executes ran
    function invariant_ConfirmationsNeverExceedOwnerCount() public view {
        uint256 ownerCount = wallet.getOwners().length;
        for (uint256 i = 0; i < wallet.getTransactionCount(); i++) {
            (, , , , uint256 confirmations) = wallet.transactions(i);
            assertLe(confirmations, ownerCount);
        }
    }
}
```

⚡ **PITFALL** An invariant test is only as good as the handler's realism. A handler that calls functions with completely unconstrained random addresses and amounts mostly just tests revert paths — bound and constrain the handler's inputs to the same range real usage would actually produce, or the invariant runs spend most of their budget on scenarios that could never happen in production.

12.4 Fork Testing

Some bugs only exist at the boundary between your contract and something you don't control — a real Chainlink feed, a real Uniswap pool, a real token with unusual transfer behavior. Fork testing runs your tests against an actual snapshot of mainnet (or any live chain) state, fetched via RPC, so "the real world" is what your test executes against instead of a mock.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test} from "forge-std/Test.sol";

interface AggregatorV3Interface {
    function latestRoundData() external view returns (
```

```
        uint80, int256, uint256, uint256, uint80
    );
}

contract ChainlinkForkTest is Test {
    // The real, live ETH/USD feed on Ethereum mainnet
    AggregatorV3Interface constant ETH_USD_FEED =
        AggregatorV3Interface(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419);

    function setUp() public {
        vm.createSelectFork(vm.rpcUrl("mainnet"));
    }

    function test_RealFeedReturnsFreshReasonablePrice() public view {
        (, int256 price, , uint256 updatedAt, ) = ETH_USD_FEED.latestRoundData();

        assertGt(price, 0);
        assertLt(block.timestamp - updatedAt, 24 hours);
    }
}
```

⚠ **SECURITY** This is exactly the kind of test that would have caught a bad assumption about Chapter 5's oracle staleness check — mocked price feeds always return whatever the test tells them to, but a fork test against the real feed proves your staleness threshold is actually compatible with how often that specific feed genuinely updates.

12.5 Coverage: What It Does and Doesn't Tell You

forge coverage reports what percentage of your code's lines and branches executed during the test suite. It's a useful floor, not a target: 100% coverage means every line ran at least once, not that every meaningful input or sequence was tested — a fuzz test with a weak assertion can hit 100% coverage while checking almost nothing useful.

□ TRY IT YOURSELF — Write a Real Fuzz Test

Take Chapter 2's SimpleCounter and write testFuzz_IncrementNTimesEqualsCount(uint8 n) that calls increment() n times in a loop, then asserts getCount() equals n. Bound n to a reasonable range so the test doesn't take forever on huge values.

Hint: uint8 already caps the fuzzed value at 255, so you likely don't even need bound() here — just loop from 0 to n calling increment().

□ TRY IT YOURSELF — Design an Invariant for SimpleAMM

Chapter 5's SimpleAMM should maintain reserveA * reserveB >= k_previous after every swap (fees mean it should only ever grow, never shrink). Sketch — in comments, not full code — what a handler function and the invariant check for this would look like.

*Hint: Your handler needs at least a swap function with bounded amounts; the invariant itself just needs to store the k value after setUp() and compare current reserveA * reserveB against it on every check.*

KEY TAKEAWAYS

- Unit tests only check the exact scenarios you thought to write — anything you didn't imagine is invisible to them.
- Fuzz testing runs a test function hundreds of times with random inputs, searching for the specific value that breaks an assertion.
- bound() maps any fuzzed input into your target range so every run contributes a real test case, unlike vm.assume() which discards out-of-range runs.
- Invariant testing calls random sequences of functions through a handler and checks a property holds after every single call, not just one isolated function.
- A handler's realism determines an invariant test's value — unconstrained random inputs mostly just test revert paths, not real usage patterns.
- Fork testing runs against real, live chain state fetched via RPC — the only way to catch bugs that only exist at the boundary with something you don't control.
- 100% test coverage is a floor, not a target — it proves every line ran once, not that the meaningful inputs and sequences were actually tested.

□ DID YOU KNOW?

Foundry's fuzzer doesn't just generate purely random values — it uses a technique borrowed from general-purpose fuzzing tools like AFL, biasing toward values that are more likely to trigger edge cases: zero, the maximum value for a type, values just above and below round numbers, and values close to ones seen in previous failing runs. This is why Foundry fuzz tests tend to find real bugs faster than naive random generation would.

CHAPTER 13

Glossary and Resources

13.1 Glossary

Every term below appeared somewhere in Chapters 1 through 12 — this is a quick-reference, not a re-explanation. Each entry points back to where it was actually covered in depth.

Term	Definition
Account Abstraction	Letting a smart contract, not just a private key, authorize transactions — see ERC-4337 and EIP-7702 (Ch. 9)
AMM	Automated Market Maker — prices trades using a formula against pooled reserves instead of an order book (Ch. 5)

Bytecode	The compiled, low-level instructions the EVM actually executes — what Solidity source compiles down to (Ch. 1)
Calldata	Read-only data location for external function parameters — the cheapest of the three data locations (Ch. 2)
Checks-Effects-Interactions	Validate first, update state second, only then call outside contracts — the pattern that prevents reentrancy (Ch. 3, 6)
Custom Error	A gas-cheaper alternative to require() strings, defined with the error keyword (Ch. 2)
DeFi	Decentralized Finance — financial applications (exchanges, lending, derivatives) built as smart contracts (Ch. 5)
Delegatecall	Executes another contract's code using the CALLER's storage — the mechanism behind proxy patterns (Ch. 3)
EIP	Ethereum Improvement Proposal — the formal process for proposing protocol or standard changes
ERC	Ethereum Request for Comment — an EIP specifically defining an application-level standard, like ERC-20
EVM	Ethereum Virtual Machine — the sandboxed runtime that executes contract bytecode identically on every node (Ch. 1)
Fork Testing	Running tests against a real snapshot of live chain state, fetched via RPC (Ch. 12)
Fuzz Testing	Running a test function many times with randomly generated inputs to find edge cases (Ch. 12)
Gas	The unit that prices computation on the EVM — every operation has a fixed, published cost (Ch. 1)
Health Factor	A lending position's collateral value divided by its debt value — determines if it can be liquidated (Ch. 5)
Immutable	A variable set once at deployment and baked into bytecode — no storage read needed to access it (Ch. 8)
Impermanent Loss	What an AMM liquidity provider gives up versus just holding, when pooled token prices diverge (Ch. 5)
Invariant Testing	Calling random sequences of functions and checking a property holds after every single call (Ch. 12)
Liquidation	Forcibly closing an under-collateralized lending position, usually at a discount to the liquidator (Ch. 5)
Mapping	A hash-table-like storage structure where every possible key already exists at its type's default value (Ch. 2)
Memory	Temporary data location wiped after a function call ends — cheaper than storage, costlier than calldata (Ch. 2)
MultiSig	A wallet requiring M-of-N owner confirmations before any transaction executes (Ch. 10)
Nonce	A number that increments with each use, prevents the same signed message from being replayed (Ch. 6)

Oracle	A mechanism bringing external, off-chain data (like a price) onto the chain for contracts to read (Ch. 5)
Proxy Pattern	Splitting a contract into a fixed-address Proxy (storage) and a swappable Implementation (logic) (Ch. 3)
Reentrancy	A vulnerability where an external call lets the callee re-enter the calling function before state updates (Ch. 3)
Rollup	An L2 that executes transactions off-chain and posts data/proofs back to L1 for security (Ch. 1, 9)
Slippage	The gap between a trade's expected and executed price, caused by the trade's own price impact (Ch. 5)
Storage	Persistent, on-chain data location — the most expensive of the three, since it's written to the blockchain (Ch. 2)
Storage Slot	A fixed 32-byte unit of storage; Solidity packs smaller variables into shared slots automatically (Ch. 8)
TVL	Total Value Locked — the combined value of assets deposited in a protocol (Ch. 5)
Unchecked Block	Opts out of Solidity 0.8's default overflow protection, for gas savings, only where overflow is impossible (Ch. 6, 8)
UserOperation	The transaction-like structure that ERC-4337 account abstraction routes through a bundler and EntryPoint (Ch. 9)
Vault (ERC-4626)	A standardized deposit/share accounting pattern letting any protocol integrate without custom code (Ch. 9)

13.2 Essential Documentation

Solidity Documentation — https://docs.soliditylang.org *the canonical language reference — syntax, semantics, every built-in*

Foundry Book — https://getfoundry.sh *the complete reference for forge, cast, anvil, and chisel*

OpenZeppelin Contracts Docs — https://docs.openzeppelin.com/contracts *audited implementations of every standard pattern in this book*

EIPs Repository — https://eips.ethereum.org *the actual specification text for every ERC and EIP mentioned in this book*

Ethereum.org Developer Docs — https://ethereum.org/developers *broader conceptual reference beyond Solidity specifically*

13.3 Community and Continued Learning

Solidity by Example — https://solidity-by-example.org *short, focused code examples for nearly every language feature and pattern*

Ethereum Stack Exchange — https://ethereum.stackexchange.com *searchable Q&A for the specific error message or edge case you're stuck on*

Foundry GitHub Discussions — https://github.com/foundry-rs/foundry/discussions *where Foundry's own maintainers and power users answer tooling questions directly*

None of these replace reading real, deployed contracts on Etherscan — once a standard's mechanics feel familiar from this book, the fastest way to deepen it further is opening a verified contract you actually use (a DEX, a lending protocol, your own wallet's contract) and reading it end to end.

13.4 Where This Book Ends and Real Practice Begins

Thirteen chapters is enough to read production Solidity fluently, understand why it's shaped the way it is, and build something real with reasonable confidence — it is not enough to skip the practices that catch what a book can't: run every contract you write through the audit process in Chapter 11, write the fuzz and invariant tests Chapter 12 covers before anything holds real funds, and reach for OpenZeppelin's audited implementations over hand-written versions every time production code is on the line.

Every contract in this book, from Chapter 2's SimpleCounter to Chapter 10's MultiSigWallet, was built the same way: a complete, working example first, then an explanation of every piece, then the mistake version and why it's wrong. That's not a teaching gimmick — it's the actual habit worth keeping. When the next unfamiliar pattern shows up, in someone else's codebase or your own, the same approach still works: get it running, break it down piece by piece, and ask what happens when it's used wrong before trusting that it's used right.

That's the whole book. Go build something.

> **□ DID YOU KNOW?**
>
> The term "gas" itself is a deliberate metaphor from Ethereum's earliest design documents — computation is priced the way fuel is, consumed as you go, with a "tank" (the gas limit) that runs out if a transaction tries to do more work than it paid for. The metaphor has held up remarkably well for a concept coined years before anyone had really used it at scale.

About the Author

MD Ayaan Siddiqui is a Full Stack Blockchain Developer with over two years of international remote experience across DeFi, smart contracts, and Web3. He started as a technical co-founder at a Netherlands-based Web3 startup that was submitted to Y Combinator, then worked at a hardware wallet company, followed by a company building a Bitcoin DeFi stablecoin ecosystem. He currently works independently as a blockchain researcher.

His core stack spans Solidity, Foundry, Next.js, TypeScript, and Python, with a research interest in agentic infrastructure on blockchain — specifically the intersection of agent identity and cross-chain micro-settlement protocols.

He holds a B.Tech in Computer Science and an online MBA in Blockchain Management from Guglielmo Marconi University, Italy.

Ayaan is also an independent blockchain researcher. His solo-authored paper, “A Cross-Chain Architecture for Coupling ERC-8004 Agent Identity with x402 Micro-Settlement Across Solana and Ethereum,” is a published research preprint on Zenodo:

https://zenodo.org/records/21394562 *(DOI: 10.5281/zenodo.21394562)*

Acknowledgments

This book leans heavily on work the wider Ethereum community has spent years getting right — the Solidity core team, OpenZeppelin's contributors, and the Foundry team at Paradigm chief among them. Every pattern in

this book that's described as “the audited version” exists because of that work. Thank you also to everyone who has ever filed an issue, answered a question on Ethereum Stack Exchange, or written up a post-mortem after an exploit — that transparency is most of how this entire field learns.

Connect

Portfolio: https://moayaan.com

GitHub: https://github.com/moayaan1911

LinkedIn: https://linkedin.com/in/ayaaneth

Found an error in this book, or built something using it worth sharing? Reach out through any of the above — always happy to hear from readers.