BLOCKCHAIN & WEB3

Smart Contracts, DeFi & Web3 Security

Writing real Solidity, the EVM and gas, smart contract design patterns, real DeFi mechanics (AMMs, liquidity pools, lending), the dApp/Web3 stack, DAOs, a full vulnerability taxonomy, two real documented exploits (Poly Network and Ronin), and the genuine custody and regulatory risks of Web3 — closing with a capstone that designs, audits, and fixes a real reentrancy vulnerability in a small staking vault.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: WRITING YOUR FIRST SMART CONTRACT: SOLIDITY BASICS

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 1 · Writing Your First Smart Contract: Solidity Basics

Blockchain & Web3 Fundamentals (Course 1) got you as far as knowing what a Contract Account is (Chapter 6) and roughly what the EVM, Solidity, and gas are for. This course builds directly on that foundation and gets hands-on: real Solidity syntax, a real worked contract, and exactly what happens on-chain when you deploy one.

Solidity's Real Origin

Solidity was proposed by **Gavin Wood** (one of Ethereum's own founding team members, per Course 1 Chapter 6) in **August 2014**, and subsequently developed further by Christian Reitwiessner, Alex Beregszaszi, and several other Ethereum core contributors. It's a statically-typed, contract-oriented language, purpose-built for writing code that runs on the EVM — compiling down to the actual bytecode the EVM executes, exactly as Course 1 Chapter 6 described. Its syntax deliberately draws on JavaScript, C++, and Python, making it comparatively approachable for developers coming from any of those backgrounds, while still enforcing static typing that plain JavaScript doesn't have.

The Anatomy of a Contract

Here's a real, complete, minimal Solidity contract — a simple counter that anyone can increment, with its current value publicly readable by anyone:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint public count;

    constructor() {
        count = 0;
    }
```

```
    function increment() public {
        count += 1;
    }

    function getCount() public view returns (uint) {
        return count;
    }
}
```

Breaking down each real piece:

- `pragma solidity ^0.8.0;` — declares which compiler version(s) this contract is written for. The `^` means "this version or any later compatible 0.8.x version" — a real safeguard against a future compiler version silently changing how the code behaves.
- `contract Counter { ... }` — the `contract` keyword defines the contract's own scope, directly comparable to a class definition in an object-oriented language.
- `uint public count;` — a **state variable**. Unlike a local variable inside a function, a state variable's value is permanently stored as part of the Contract Account's own on-chain data (Course 1 Chapter 6), persisting between separate transactions.
- `constructor() { ... }` — code that runs exactly once, at the moment the contract is deployed, and never again afterward.
- `function increment() public { ... }` — a function anyone can call to change the contract's own state (here, increasing `count` by one).

Function Visibility

Solidity has four real visibility keywords, controlling exactly who can call a given function:

Keyword	Who Can Call It
public	Anyone — other contracts, external accounts, and code inside this same contract

external	Only from outside the contract — not callable directly from other functions inside the same contract
internal	Only from inside this contract, or a contract that inherits from it
private	Only from inside this exact contract — not even inheriting contracts can call it

"PRIVATE" DOESN'T MEAN "SECRET"

This is a genuinely important, easy-to-misunderstand distinction: `private` only restricts which *other Solidity code* can directly call a function or read a variable. It does nothing to hide the underlying data itself — every transaction, every state variable's own value, is stored on a public, replicated blockchain (Course 1, Chapters 1 and 3) and can be read directly off the chain by anyone with the right tools, regardless of what visibility keyword Solidity uses. "Private" is an access-control concept for other contract code, not a real confidentiality guarantee.

view and pure: Promising Not to Change (or Even Touch) State

Two further real modifiers describe a function's relationship to the contract's own state:

- `view` — the function can read state variables, but promises not to modify any of them. `getCount()` above is a real example.
- `pure` — the function promises not to even read state variables at all; it only operates on whatever arguments it's given.

This distinction has a genuine, practical payoff: because a `view` or `pure` call doesn't change anything on-chain, it can typically be executed locally by a node without needing to be broadcast as a real, gas-costing transaction at all — you can call `getCount()` for free to simply read the current value, while calling `increment()` genuinely requires a real, signed, gas-paying transaction, since it actually changes the contract's own permanent state.

The address Type and msg.sender

Solidity has a dedicated `address` type representing an Ethereum account — either an Externally Owned Account or a Contract Account (Course 1, Chapter 6). Inside any function, the built-in variable `msg.sender` always holds the address that actually called the currently-executing function — the real mechanism that lets a contract know *who* is interacting with it, since a Contract Account has no private key of its own to check a signature against directly the way an EOA does.

OTHER COMMON BASIC TYPES

Beyond `uint`, `address`, and the boolean `bool`, Solidity's `mapping` type is especially common — a key-value store, most often written as `mapping(address => uint)`, letting a contract associate a value (like a token balance) with each address. Chapter 2's own gas coverage will show exactly why reading and writing to a mapping's own storage has a real, measurable cost.

Deploying a Contract: Creating a Real Contract Account

Writing Solidity code doesn't put it on-chain by itself. Deploying a contract means compiling the source code down to real EVM bytecode, then sending a special transaction (signed with a real private key, exactly like any other transaction from Course 1 Chapter 2) whose destination is empty rather than an existing address. The network responds by creating a brand-new **Contract Account** — the exact account type Course 1 Chapter 6 introduced — at a new address, with the compiled bytecode permanently stored as that account's own code. From that point on, anyone can call the contract's own public functions by sending transactions to that new address.

WHY THIS TIES DIRECTLY BACK TO COURSE 1

Deploying a contract isn't a separate, special blockchain operation — it's still just a signed transaction, broadcast, verified, and eventually mined or validated into a block, exactly like every other transaction Course 1 covered. The only

genuinely new thing is what the transaction's own effect is: instead of moving value between two existing accounts, it creates a brand-new Contract Account and gives it its own permanent code.

Hands-On Exercises

Three exercises reinforcing Solidity's basic building blocks before Chapter 2 covers the EVM and gas in real depth.

EXERCISE 1

Using this chapter's own `Counter` contract as a model, describe (in plain English, no code required) what a `decrement()` function would need to do, and whether it should be marked `view`, `pure`, or neither. Justify your answer.

→ Solution

EXERCISE 2

A colleague says: "I marked this function `private` so nobody else can see the secret value it stores." Using this chapter's own warning about what `private` actually restricts, explain what's wrong with this reasoning.

→ Solution

EXERCISE 3

Explain, using this chapter's own deployment description and Course 1 Chapter 6's account model, exactly what kind of Ethereum account exists immediately before a contract is deployed at a given address, and what kind exists immediately afterward.

→ Solution

Quick Reference

- **Solidity** — proposed by Gavin Wood in August 2014; statically-typed, compiles to EVM bytecode.
- **Contract anatomy** — `pragma` (compiler version), `contract` (scope), state variables (persistent on-chain data), `constructor` (runs once, at deployment), functions.
- **Visibility** — `public`, `external`, `internal`, `private`; `private` restricts other Solidity code, not on-chain data visibility.
- `view` / `pure` — promise not to modify state / not to touch state at all; both can typically be called for free, without a real transaction.
- `address` / `msg.sender` — the type representing an account; the built-in variable holding whoever called the current function.
- **Deployment** — a real, signed transaction with no destination address, which creates a brand-new Contract Account holding the compiled bytecode.

CHAPTER 2: THE ETHEREUM VIRTUAL MACHINE & GAS

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 2 · The Ethereum Virtual Machine & Gas

Chapter 1 said the EVM executes compiled bytecode, and that gas pays for it, without going any deeper. This chapter does: what the EVM actually is under the hood, what a transaction is really paying for, and precisely what happens when a transaction runs out of gas mid-execution.

The EVM: A Real, Stack-Based Virtual Machine

The **Ethereum Virtual Machine (EVM)** is a **stack-based** virtual machine — it executes by pushing and popping values on and off a stack, with a real, specified maximum depth of **1024 items**, where each item is a 256-bit word (a deliberate size choice for compatibility with the cryptographic operations Course 1 Chapter 2 covered).

Crucially, the EVM is formally a **deterministic state transition function**: given the same starting state and the same transaction, it always produces exactly the same resulting state, on any machine, anywhere. This determinism is what lets every node on the network independently execute the exact same contract code and reach identical conclusions about the result — without that guarantee, Chapter 6 of Course 1's own account-balance consensus would be impossible to maintain.

Bytecode and Opcodes

Solidity source code (Chapter 1) compiles down to real EVM bytecode — a sequence of low-level instructions called **opcodes**. Some are ordinary computational operations (`ADD`, `AND`, `XOR`); others are blockchain-specific, reading real context about the transaction or the chain itself (`ADDRESS`, `BALANCE`). A tiny illustrative trace of how the stack changes while evaluating `2 + 3`:

```
PUSH 2                                  stack: [2]
PUSH 3                                  stack: [2, 3]
```

```
ADD                                  stack: [5] — pops both, pushes the sum
```

Why Gas Exists: A Real Solution to a Real Problem

Every opcode has an assigned **gas cost**. This solves two real, distinct problems at once:

- It fairly charges users for the real computational effort their transaction actually demands from every node on the network, which has to independently perform that same work to verify it.
- It gives the EVM a genuine, mechanical way around the **halting problem** — the real, classic computer-science result that there's no general way to determine in advance whether an arbitrary program will ever finish running. Since a smart contract could, in principle, contain an infinite loop, Ethereum needs some way to guarantee execution always eventually stops. Requiring gas for every single operation, with execution forcibly halting the instant available gas runs out, sidesteps the halting problem entirely — not by solving it, but by making it irrelevant: execution is guaranteed to stop within a bounded number of steps no matter what the code does, because it will always run out of purchased gas eventually if it hasn't finished by then.

Gas Price, Gas Limit, and Gas Used

Three real, distinct terms, easy to conflate:

- **Gas limit** — the maximum amount of gas the sender is willing to let this transaction consume. A standard, simple ETH transfer has a real, fixed cost of exactly **21,000 gas** — a useful, concrete reference point.
- **Gas used** — how much gas the transaction actually consumed once executed. This can be less than the gas limit (any unused gas is refunded), but never more.
- **Gas price** — how much the sender is willing to pay per unit of gas, typically quoted in **gwei** (one billionth of an ETH).

RUNNING OUT OF GAS: A REAL, IMPORTANT DETAIL

If a transaction runs out of gas partway through execution, the EVM reverts every state change the transaction attempted to make — but the gas already consumed doing that work up to the point of failure is **not** refunded. This is a deliberate, real design choice: it still costs the network real, genuine computational effort to have attempted (and then unwound) that partial execution, and refunding it would let an attacker probe the network for free by repeatedly triggering failed transactions.

EIP-1559: A Real, Documented Change to How Fees Work

Ethereum's fee mechanism changed in a real, significant way on **5 August 2021**, as part of the **London hard fork**, which activated a proposal called **EIP-1559**. Rather than a single gas price bid, every transaction now pays two real, separate components:

Base fee	set automatically by the protocol itself, and permanently burned (destroyed) rather than paid to anyone
Priority fee (tip)	a real, optional amount the sender adds, paid to whoever validates the block, as an incentive for prompt inclusion

The real, resulting formula for a transaction's total cost:

```
total fee = gas used × (base fee + priority fee)
```

A worked, real example: a simple 1 ETH transfer at 21,000 gas, with a base fee of 10 gwei and a priority fee of 2 gwei, costs `21,000 × (10 + 2) = 252,000 gwei`, or exactly 0.000252 ETH in fees — on top of the 1 ETH actually being sent.

TYING THIS DIRECTLY TO CHAPTER 1'S OWN COUNTER CONTRACT

Chapter 1's `increment()` function writes to a state variable — a genuinely real, gas-costing operation, since permanently updating on-chain storage is one of the more expensive categories of work an EVM opcode can do. Chapter 1's `getCount()` function, marked `view`, doesn't modify anything and can typically be executed locally by a node without ever being broadcast as a real transaction — which is exactly why it doesn't cost any gas at all in ordinary use, even though it's still, technically, running real EVM bytecode to compute its answer.

Hands-On Exercises

Three exercises reinforcing gas mechanics and the EIP-1559 fee model before Chapter 3 covers real, reusable smart contract design patterns.

EXERCISE 1

A transaction has a gas limit of 100,000, but genuinely needs 130,000 gas to fully complete. Using this chapter's own explanation of out-of-gas behavior, describe exactly what happens to (a) the transaction's intended state changes, and (b) the gas the sender paid for.

→ Solution

EXERCISE 2

Calculate the real total fee for a transaction using 45,000 gas, with a base fee of 15 gwei and a priority fee of 3 gwei, using this chapter's own formula. Then explain, in your own words, why the base fee portion doesn't go to the validator who included the transaction.

→ Solution

EXERCISE 3

Explain, using this chapter's own halting-problem reasoning, why Ethereum couldn't simply say "every contract function must finish within 10 seconds" as

> an alternative to gas — what real problem would that alternative rule fail to solve?
>
> → Solution

Quick Reference

- **EVM** — a deterministic, stack-based virtual machine (1024-item stack, 256-bit words) that every node runs identically.
- **Bytecode / opcodes** — Solidity compiles to low-level EVM instructions, each with its own real gas cost.
- **Why gas exists** — charges for real computation, and sidesteps the halting problem by guaranteeing execution always eventually stops.
- **Gas limit / gas used / gas price** — the cap you'll allow, what was actually consumed, and the per-unit price (in gwei) you're paying.
- **Out of gas** — state changes revert, but gas already consumed is not refunded.
- **EIP-1559 (5 August 2021, the London hard fork)** — total fee = gas used × (base fee, burned, + priority fee, paid to the validator).

CHAPTER 3: SMART CONTRACT DESIGN PATTERNS

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 3 · Smart Contract Design Patterns

Chapters 1 and 2 gave you Solidity's syntax and the EVM's real mechanics. This chapter covers something different: the real, established patterns experienced Solidity developers actually reach for, several of which exist specifically because of one of the most expensive lessons in blockchain history — the same DAO hack Course 1 Chapter 6 already introduced, revisited here with the specific technical detail that chapter deliberately left out.

Checks-Effects-Interactions: The Pattern the DAO Hack Made Famous

The **Checks-Effects-Interactions (CEI)** pattern is a real, foundational rule for structuring any function that both updates a contract's own state and calls out to another address: always perform these three kinds of operations in this exact order.

1. **Checks** — validate every condition first (does the caller have permission, is the requested amount valid, and so on).
2. **Effects** — update the contract's own internal state (Chapter 1's state variables) to reflect the outcome, *before* talking to anyone else.
3. **Interactions** — only now, last, call out to an external address or contract (for example, actually sending ETH).

Why the Order Matters: The Real DAO Hack Mechanism

Course 1 Chapter 6 covered the real, documented 2016 DAO hack — roughly $50 million drained from a $150 million smart contract, resolved by a contentious hard fork. What it deliberately didn't cover is precisely *how* the attacker actually did it — and the real mechanism is a textbook violation of this exact ordering.

May 2016	A published security paper had already flagged a real vulnerability related to "recursive calls" in the DAO's own contract code, before the attack ever happened.
17 Jun 2016	An attacker exploits it: the DAO's withdrawal function sent ETH to the caller *before* updating the caller's own internal balance record. The attacker's own contract, receiving that ETH, automatically called straight back into the same withdrawal function again — and because the balance hadn't been updated yet, the DAO happily paid out again. And again. Real, genuine **reentrancy**: the same, not-yet-updated function called recursively, over and over, before its own first invocation had ever finished. Roughly 3.6 million ETH — about a third of the 11.5 million ETH committed to the DAO — was drained this way, worth roughly $50 million at the time.

A REAL, FORTUNATE DESIGN DETAIL

The DAO's own contract happened to include a real 28-day holding period before withdrawn funds could be fully moved elsewhere — not a deliberate security feature against this specific attack, just a structural detail of the DAO's own design. It gave the community real time to respond before the attacker could walk away with the funds entirely, directly enabling the hard fork response Course 1 Chapter 6 already covered. This is worth knowing honestly as a piece of genuine luck in how events unfolded, not a security mechanism anyone had deliberately planned to rely on.

Applying Checks-Effects-Interactions correctly would have prevented this outright: if the DAO's own contract had updated the caller's balance *before* sending any ETH, the second and every subsequent recursive call would have seen a balance already reduced to zero, and simply failed.

VULNERABLE — SENDS ETH BEFORE UPDATING STATE (VIOLATES CEI)

```
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount, "Insufficient balance");
```

```
    // INTERACTION happens before the EFFECT — the bug
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    balances[msg.sender] -= amount; // too late — already sent
}
```

FIXED — FOLLOWS CHECKS, EFFECTS, THEN INTERACTIONS

```
function withdraw(uint amount) public {
    // CHECK
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // EFFECT — update state first
    balances[msg.sender] -= amount;

    // INTERACTION — last
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

In the fixed version, even if the recipient's own contract tries the exact same recursive re-entry trick, `balances[msg.sender]` has already been reduced by the time the external call happens — so a second, recursive call to `withdraw()` now correctly fails the very first `require` check. Chapter 7's own dedicated security chapter covers reentrancy and several other real vulnerability classes in full depth — this chapter's job is establishing the pattern that prevents this one by default.

Pull Over Push Payments

A closely related real pattern: when a contract needs to pay out to multiple recipients, don't loop through them and actively **push** ETH to each one directly. Instead, record how much each address is owed, and let each recipient **pull** their own payment by calling a separate `withdraw()` function themselves, whenever they choose.

This has two real, concrete benefits: a single recipient with a broken or malicious contract can't block payment to everyone else in the same loop (since each pull is now a fully independent transaction), and it naturally reduces how many places in your own code perform a real external interaction — directly shrinking the surface area where a Checks-Effects-Interactions mistake could occur in the first place.

Access Control: Modifiers and the Ownable Pattern

Solidity's real **modifier** feature lets you define a reusable condition, checked before a function's own body runs. The most common real example restricts a function to one specific, privileged address — typically called the **Ownable** pattern:

```
address public owner;

modifier onlyOwner() {
    require(msg.sender == owner, "Not the owner");
    _; // the calling function's own body runs here
}

function withdrawFees() public onlyOwner {
    // only the owner address can ever reach this point
}
```

Pattern	Real Problem It Solves
Checks-Effects-Interactions	Prevents reentrancy by updating state before making external calls
Pull over push	Prevents one broken recipient from blocking payment to everyone else
Ownable / access control	Restricts sensitive functions to one privileged address

A Real, Honest Complication: Contracts Are Immutable by Default

Course 1 Chapter 3 established that a blockchain's history is tamper-evident by design — and the exact same immutability applies to a deployed contract's own code. Once deployed, a contract's bytecode cannot simply be edited later, even by its own original developer, even to fix a genuine bug. This is a deliberate, structural consequence of everything Course 1 already established, not a separate limitation.

Real projects that genuinely need to upgrade their own logic over time use a real, more advanced approach called a **proxy pattern**: users interact with one fixed, permanent "proxy" contract address, which internally delegates its actual logic to a separate implementation contract — and upgrading later means pointing the proxy at a *new* implementation contract, rather than editing any contract's own code after the fact. This is a genuinely more complex pattern with its own real trade-offs and risks, deliberately left out of full depth here — the honest, important point for this chapter is simply that "just fix the bug and redeploy in place" is never actually an option once a contract is live.

OpenZeppelin: The Real, Standard Library

DON'T REINVENT THESE PATTERNS FROM SCRATCH

OpenZeppelin Contracts is a real, widely used, community-reviewed library of battle-tested Solidity code implementing exactly the patterns this chapter covers — a ready-made `Ownable` contract, a `ReentrancyGuard` modifier providing an extra layer of reentrancy protection on top of CEI, and standard, audited ERC-20/ERC-721 implementations (Course 1, Chapter 8). Real, major DeFi protocols and financial platforms rely on it directly rather than hand-writing these same patterns themselves each time. In real-world development, using an established, reviewed library like this for foundational patterns is almost always safer than writing your own from scratch.

Hands-On Exercises

Three exercises applying Checks-Effects-Interactions and this chapter's other real patterns, before Chapter 4 turns to DeFi mechanics built directly on top of them.

EXERCISE 1

Using this chapter's own real DAO hack mechanism, explain step by step why the vulnerable `withdraw()` function in this chapter allows a malicious contract to drain far more than its own actual balance, while the fixed version doesn't.

→ Solution

EXERCISE 2

A contract needs to pay out prize money to 50 different competition winners at once. Using this chapter's own pull-over-push reasoning, explain what could go wrong with a loop that sends ETH directly to all 50 addresses in a single transaction, and how the pull pattern avoids it.

→ Solution

EXERCISE 3

A junior developer says: "If we find a bug in our deployed contract, we'll just push a fix." Using this chapter's own explanation of contract immutability, correct this statement and describe what a real fix would actually require.

→ Solution

Quick Reference

- **Checks-Effects-Interactions** — validate, then update your own state, then call external addresses last — in that exact order.
- **The DAO hack, precisely** — a real reentrancy exploit (17 June 2016) that sent ETH before updating balances, letting a recursive callback drain ~3.6M ETH (~$50M)

before the balance was ever reduced.

- **Pull over push** — let recipients withdraw their own payments individually, rather than pushing to everyone in one loop.
- **Modifiers / Ownable** — reusable pre-conditions (like `onlyOwner`) restricting sensitive functions to a privileged address.
- **Immutability & proxies** — deployed contract code can't be edited in place; real upgradability needs a dedicated proxy pattern instead.
- **OpenZeppelin** — the real, standard, audited library implementing these patterns rather than hand-rolling them.

CHAPTER 4: DECENTRALIZED FINANCE (DEFI): AMMS, LIQUIDITY POOLS & LENDING

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 4 · Decentralized Finance (DeFi): AMMs, Liquidity Pools & Lending

*Chapters 1 through 3 gave you the real tools smart contracts are built from. This chapter applies all of it to **DeFi** — "decentralized finance": real trading, lending, and borrowing services built entirely from smart contracts, with no bank, exchange, or broker sitting in the middle.*

Automated Market Makers: Trading Without an Order Book

A traditional exchange matches buyers and sellers through an order book. DeFi's dominant real alternative is the **Automated Market Maker (AMM)**, pioneered by **Uniswap**, created by **Hayden Adams** and launched on **2 November 2018**, directly inspired by an earlier blog post from Ethereum co-founder Vitalik Buterin (Course 1, Chapter 6).

Instead of matching individual buy and sell orders, an AMM holds a shared pool of two tokens and prices trades using a real, deterministic formula: the **constant product formula**, `x × y = k`, where `x` and `y` are the pool's current quantities of each token, and `k` must stay exactly constant across every trade.

Before a trade	**100 ETH × 200,000 USDC = k (20,000,000)**
A trader adds	**+10 ETH to the pool**
To keep k constant, USDC must fall to	**20,000,000 ÷ 110 ≈ 181,818 USDC**
The trader receives	**200,000 − 181,818 ≈ 18,182 USDC**

Notice what just happened to the real price: before the trade, 1 ETH was worth 2,000 USDC (200,000 ÷ 100). After it, the pool implies roughly 1,653 USDC per ETH (181,818

÷ 110) — the trade itself moved the price, purely as a mechanical consequence of the formula, with no separate "order book" or human market maker involved at all.

Liquidity Pools and Liquidity Providers

The tokens sitting in that pool don't come from Uniswap itself — they're deposited by ordinary users called **liquidity providers (LPs)**, who contribute a matched pair of tokens to the pool and, in exchange, earn a real share of the trading fees every subsequent swap generates.

IMPERMANENT LOSS: A REAL, HONEST RISK FOR LIQUIDITY PROVIDERS

If the real market price of the two pooled tokens shifts after you deposit, the AMM's own constant-product formula automatically rebalances the pool — effectively selling off some of whichever token is becoming more valuable and buying more of whichever is becoming less valuable, exactly as the worked example above shows happening on every single trade. This means a liquidity provider can end up, upon withdrawal, holding a combination of tokens worth less than if they had simply held the original two tokens outside the pool the whole time. It's called "impermanent" because the loss only becomes real once you actually withdraw at an unfavorable price ratio — if prices later return to where they started, the loss disappears on paper. But it's a genuine, real financial risk in the meantime, not a theoretical footnote.

Lending and Borrowing: Real Protocols, No Credit Check

Real DeFi lending protocols — Aave and Compound are two of the most established — let users deposit crypto assets to earn interest, and let other users borrow against posted collateral, with interest rates that adjust automatically based on real-time supply and demand for each asset.

There's a genuine structural reason DeFi lending looks different from a bank loan: **over-collateralization**. Since there's no credit check, no identity verification, and no legal recourse against a pseudonymous blockchain address, these protocols require

borrowers to post *more* collateral value than they're actually allowed to borrow — the collateral itself is the entire basis of trust, replacing everything a credit score or a bank's own underwriting process would normally provide.

LIQUIDATION: CODE, NOT A COURT, ENFORCES THE LOAN

If a borrower's collateral value falls too close to the value of what they've borrowed (because the collateral asset's own market price dropped), the protocol automatically **liquidates** — sells off enough of the collateral to repay the loan, with no human intervention, no negotiation, and no notice period beyond what the smart contract's own code specifies. This is a direct, real-world consequence of Chapter 3's own Checks-Effects- Interactions discipline and Course 1's own "code is law" theme: the loan's own terms are enforced exactly as written, automatically, the moment the code's own conditions are met.

Flash Loans: A Genuinely Novel DeFi Primitive

DeFi introduced a real financial instrument with no direct traditional-finance equivalent: the **flash loan** — an *uncollateralized* loan of any size, borrowed and fully repaid within a single blockchain transaction, or the entire transaction reverts as though it never happened.

1. Borrow a large sum — potentially millions of dollars worth of tokens — with zero collateral posted.
2. Use the borrowed funds for something — commonly, a trade or a series of trades across multiple protocols.
3. Repay the full loan amount, plus a fee, before the same transaction finishes.
4. If step 3 doesn't succeed for any reason, the EVM reverts the *entire* transaction — including the original loan — as if none of it had ever happened at all.

This is only possible because of a real property Chapter 2 already established: a single Ethereum transaction is **atomic** — if execution fails to complete for any reason, every state change it attempted gets reverted, exactly like Chapter 2's own out-of-gas behavior, just triggered here by a deliberate `require` check instead of running out of gas. Because the loan and its repayment are forced into that same atomic unit, the lender needs zero trust in the borrower at all: either the money genuinely comes back by the end of the transaction, or, from the chain's own perspective, the loan simply never happened.

A REAL, HONEST FORWARD REFERENCE

Flash loans are a genuinely powerful, legitimate tool — but their real ability to temporarily command enormous, uncollateralized sums has also made them a real, documented component of several serious DeFi exploits, typically used to briefly manipulate an AMM's own price (exactly the mechanism this chapter's own worked example demonstrated) before another protocol that trusts that price gets exploited. Chapters 7 and 8 cover this in real, specific detail.

DeFi Primitive	Real Purpose	Real Risk
AMM / liquidity pool	Token trading without an order book	Impermanent loss for liquidity providers
Over-collateralized lending	Borrowing without a credit check	Automatic liquidation if collateral value falls
Flash loan	Trust-free, uncollateralized borrowing within one transaction	A real, documented tool in several price-manipulation exploits

Hands-On Exercises

Three exercises applying AMM mechanics, over-collateralization, and flash loan atomicity before Chapter 5 turns to the broader dApp and Web3 stack these protocols live inside.

EXERCISE 1

A pool holds 50 ETH and 100,000 USDC. Using this chapter's own constant product formula, work out roughly how many USDC a trader receives for adding 5 ETH to the pool, and what the pool's implied ETH price is immediately before and after the trade.

→ Solution

EXERCISE 2

A friend says: "DeFi lending is safer than a bank loan, since it doesn't need a credit check at all." Using this chapter's own explanation of over-collateralization, explain what's misleading about this claim.

→ Solution

EXERCISE 3

Explain, using this chapter's own flash loan flow and Chapter 2's own transaction-atomicity/ revert behavior, why a flash loan lender takes on essentially zero counterparty risk — even when lending an enormous, completely uncollateralized sum to a stranger.

→ Solution

Quick Reference

- **AMM** — prices trades via the constant product formula (`x × y = k`) instead of an order book; pioneered by Uniswap (Hayden Adams, 2 November 2018).
- **Liquidity pool / LP** — users deposit paired tokens, earn trading fees, but risk impermanent loss if the price ratio shifts.
- **Over-collateralization** — DeFi lending requires more collateral than borrowed, replacing a credit check.
- **Liquidation** — automatic, code-enforced collateral sale if its value falls too close to the borrowed amount.

- **Flash loan** — an uncollateralized loan that must be repaid within the same atomic transaction, or the whole thing reverts.

CHAPTER 5: DECENTRALIZED APPLICATIONS (DAPPS) & THE WEB3 STACK

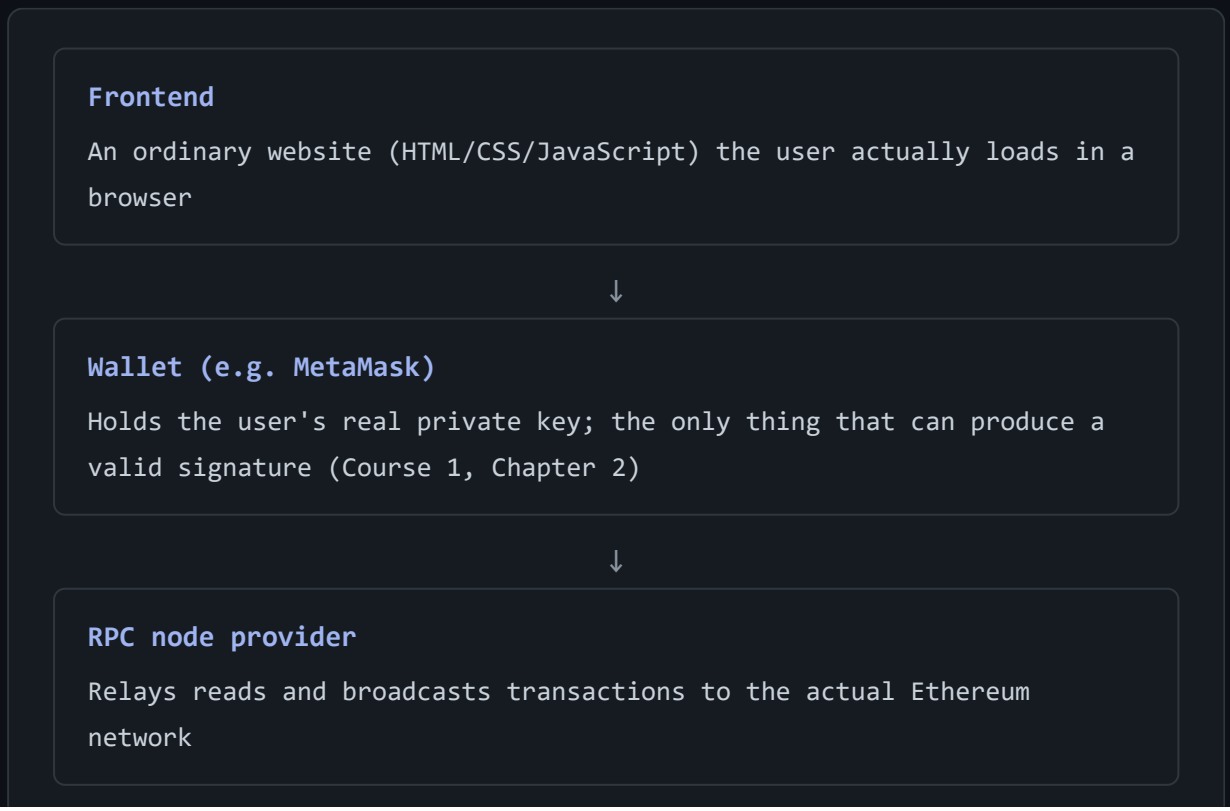
Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 5 · Decentralized Applications (dApps) & the Web3 Stack

Chapters 1 through 4 covered writing, deploying, and reasoning about smart contracts — but a real user never types Solidity into a terminal. This chapter covers what actually sits between an ordinary web page and a deployed contract: how a dApp connects a wallet, reads and writes on-chain state, and where its own frontend and off-chain data actually live.

What a dApp Actually Is

A **decentralized application (dApp)** is, structurally, just an ordinary web frontend — often built with the same tools as any other website — paired with one or more smart contracts (Chapters 1–4) as its real, on-chain backend. What makes it "decentralized" isn't the frontend at all; it's that the actual application logic and data live on the blockchain itself, per every mechanism this course has already covered, rather than on a company's own private servers.



↓

The blockchain itself

Where deployed contracts (Chapter 1) actually run and store state

MetaMask: A Real, Widely Used Wallet Layer

MetaMask, developed by ConsenSys and launched in **2016**, is the real, dominant browser-extension wallet most dApps are built to work with — reporting over 100 million users worldwide as of early 2026. It injects a JavaScript interface into every page a user visits, which a dApp's own frontend code can use to request a connection, read the user's public address, or ask the user to approve a transaction.

THE REAL SECURITY PROPERTY THIS DEPENDS ON

A dApp's own website-hosted JavaScript code *never* has direct access to the user's private key at any point. When a dApp wants the user to sign something, it sends a request through this injected interface, MetaMask displays a real, human-readable confirmation prompt, and only if the user explicitly approves does MetaMask itself — running as a separate browser extension, not as part of the website's own code — actually produce the signature using the key it holds. The website never touches the key directly, exactly matching Course 1 Chapter 7's own "a wallet stores keys, the network never needs to see them" principle.

The Real Interaction Flow

Applying this to Chapter 1's own `Counter` contract, end to end:

1. **Connect wallet** — the user clicks "Connect," MetaMask asks for approval, and the dApp receives the user's public address.

2. **Read state, for free** — the frontend calls the contract's own `getCount()`, a `view` function (Chapter 2), executed locally without needing any signature or gas at all.

3. **Submit a transaction** — the user clicks "Increment," triggering a call to the real, state-changing `increment()` function.

4. **MetaMask prompts for a signature** — showing the real destination contract, the function being called, and the estimated gas cost, before the user approves or rejects it.

5. **Broadcast and confirm** — the signed transaction is sent to the network exactly as Course 1 Chapter 7's own transaction flow described, eventually mined/validated and confirmed.

Talking to the Network: RPC Node Providers

Step 2 and step 5 above both require actually talking to the Ethereum network — but an ordinary website visitor's browser doesn't run a full Ethereum node itself (Course 1's own full-node/decentralization discussion covered why that's a real, meaningful piece of infrastructure to run). Instead, most dApps connect through a real, dedicated **RPC node provider** — a service that runs the actual full nodes on the dApp's behalf and exposes a simple API for reading state and broadcasting transactions. **Infura**, acquired by ConsenSys (MetaMask's own developer) in 2019, and **Alchemy** are two real, widely used examples, together powering a large share of real dApp traffic, including MetaMask's own default connection.

AN HONEST CENTRALIZATION TENSION

This is a genuine, widely discussed point of friction in Web3: a "decentralized" application's own contract logic and data can be as decentralized as Course 1 described, while the specific *path* most real users actually take to reach it — their wallet's default RPC connection — often runs through one of a comparatively small number of centralized infrastructure providers. If a provider like this goes down or chooses to block certain addresses, many real dApps'

own users can be meaningfully affected, even though the underlying blockchain itself keeps running completely unaffected. Running your own full node avoids this dependency entirely, but very few ordinary users actually do.

IPFS: Real, Decentralized Off-Chain Storage

Course 1 Chapter 8 noted that an NFT's own metadata and media typically live off-chain, linked from a URI. **IPFS (InterPlanetary File System)** is a real, genuinely decentralized protocol commonly used for exactly this — a peer-to-peer network where files are identified by **content addressing**: a file's own address is a cryptographic hash of its actual content, rather than a location like a normal URL. Anyone holding a copy of that exact content, on any node in the network, can serve it — and because the address is derived directly from the content itself, the hash also proves the content hasn't been tampered with, reusing exactly the same hash-integrity idea Course 1 Chapter 2 introduced. Storing a dApp's own frontend, or an NFT's metadata, on IPFS reduces (though doesn't fully eliminate) dependence on any single centralized server continuing to stay online.

Hands-On Exercises

Three exercises reinforcing the real Web3 stack before Chapter 6 turns to DAOs — organizations built entirely on top of exactly this stack.

EXERCISE 1

A user reports: "This dApp's website asked me to paste my seed phrase directly into a text box to connect my wallet." Using this chapter's own explanation of how MetaMask actually signs transactions, explain why this is a serious, non-standard red flag rather than normal dApp behavior.

→ Solution

EXERCISE 2

Using this chapter's own five-step interaction flow, explain specifically why step 2 (reading `getCount()`) doesn't trigger a MetaMask signature prompt, while step 3 (`increment()`) does.

→ Solution

EXERCISE 3

A colleague says: "As long as a contract's logic is deployed on-chain, the whole dApp is fully decentralized, with no single point of failure at all." Using this chapter's own centralization tension, explain what this claim overlooks.

→ Solution

Quick Reference

- **dApp** — an ordinary web frontend paired with smart contracts as its real, on-chain backend.
- **MetaMask** — ConsenSys, launched 2016, over 100 million users; signs transactions locally, never exposing the private key to a website.
- **Interaction flow** — connect wallet, read state for free via `view` calls, submit a transaction, sign in MetaMask, broadcast and confirm.
- **RPC node providers** — services like Infura (ConsenSys, acquired 2019) and Alchemy that run full nodes on a dApp's behalf, with a real, honest centralization tension around relying on them.
- **IPFS** — a real, content-addressed, peer-to-peer storage protocol commonly used for dApp frontends and NFT metadata (Course 1, Chapter 8).

CHAPTER 6: DAOS: DECENTRALIZED AUTONOMOUS ORGANIZATIONS

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 6 · DAOs: Decentralized Autonomous Organizations

*Course 1 Chapter 6 and this course's own Chapter 3 both revisited "The DAO" — the specific 2016 project whose real reentrancy exploit is now a permanent part of blockchain history. This chapter covers **a DAO** as a general, ongoing organizational pattern — one that genuinely survived that disaster and is still actively used today, with its own real successes, real failures, and real, unresolved governance problems.*

What a DAO Actually Is

A **Decentralized Autonomous Organization (DAO)** is an organization whose own governance rules — how decisions get made, and how its own treasury gets spent — are encoded directly in smart contracts, rather than run by a traditional board or management hierarchy. In practice, this almost always means **token-weighted governance**: members propose changes, and holders of the organization's own **governance token** (an ERC-20 token, per Course 1 Chapter 8, just used for voting rights rather than as a currency) vote on-chain, with voting power typically proportional to how many tokens each address holds.

MakerDAO: A Real, Established Example

MakerDAO, launched in **December 2017**, is one of the oldest and most significant real DAOs still operating today — its holders govern the real parameters behind DAI, a decentralized stablecoin, and the lending mechanisms that back it. Unlike the original 2016 DAO, MakerDAO has operated continuously for years, a real, concrete demonstration that the underlying pattern itself wasn't the problem in 2016 — a specific, exploitable piece of code was (Chapter 3).

ConstitutionDAO: A Real, Fully Documented Case Study

A genuinely different kind of DAO, and a real, complete story worth knowing in full — both for its remarkable success and its own honest, practical failure.

Nov 2021	**ConstitutionDAO** forms almost overnight, crowdfunding ETH from thousands of strangers with one shared goal: bidding on an original, first-edition copy of the U.S. Constitution at a real Sotheby's auction. It raises a real **$47 million**, with a median individual contribution of roughly **$217**.
Auction day	ConstitutionDAO loses the auction — billionaire Kenneth C. Griffin outbids the group at **$43.2 million**. The organizers, weighing the real added costs of insurance, storage, and transport on top of any further bid, decide not to raise their offer.
Weeks later	The DAO disbands and tries to refund every contributor — and runs directly into Chapter 2's own gas-fee reality at real scale: some contributors paid a real **$70 in gas fees** just to donate or withdraw as little as $200, and roughly $23 million remained genuinely unrefunded weeks after the auction ended, purely because of the accumulated real cost of processing thousands of individual on-chain refund transactions.

A REAL, HONEST, SLIGHTLY ABSURD POSTSCRIPT

Despite the failed mission and the messy refund process, ConstitutionDAO's own now-worthless- seeming governance tokens later became collectible in their own right, with a real, reported total value around $300 million at one point — a genuinely strange, real illustration that a token's own market price and its underlying project's actual real-world success can come apart entirely.

Multisig Treasuries: A More Robust Access-Control Pattern

A DAO's own treasury — potentially millions of dollars in pooled funds — is rarely guarded by a single Chapter 3-style `onlyOwner` address. Instead, real DAOs almost universally use a **multisig wallet**, requiring a real, defined threshold of signatures (commonly written as "M-of-N," e.g. 4-of-7 trusted signers) before any transaction can actually execute. **Safe** (formerly known as Gnosis Safe) is the real, dominant, widely used multisig implementation on Ethereum today.

This is a direct, practical extension of Chapter 3's own access-control discussion: rather than trusting one single private key (a genuine single point of failure), a multisig spreads that trust across several independent keys, so no single compromised or malicious signer can move funds alone.

Real, Honest Legal Status

A DAO's actual legal status remains genuinely unclear in most jurisdictions worldwide — a DAO with no formal legal wrapper can, in some interpretations, functionally expose its own individual members to unlimited liability as an unincorporated general partnership, since there's no recognized corporate structure shielding them. **Wyoming** was the real, first U.S. state to address this directly, passing a law effective **1 July 2021** that legally recognizes DAOs organized as LLCs — with the American CryptoFed DAO receiving the first real, formal recognition under it. This remains a genuinely unsettled area of law almost everywhere else.

Real, Honest Governance Challenges

Challenge	Why It's Real
Plutocracy	Token-weighted voting means influence concentrates wherever token holdings concentrate — the same dynamic Chapter 4's own

	DeFi coverage applies to money applies equally to governance power
Voter apathy	Many token holders simply don't vote on most real proposals, letting a small, active minority effectively decide outcomes for everyone
Legal exposure	Outside jurisdictions like Wyoming, members may carry real, unclear personal liability

THE HONEST TAKEAWAY

A DAO genuinely does remove a traditional company's own board and management hierarchy — but it doesn't automatically remove the underlying human problems those structures existed to manage (concentrated power, disengaged stakeholders, unclear accountability). It relocates them into a different, code-enforced system, with its own real, different set of trade-offs, rather than eliminating them outright.

Hands-On Exercises

Three exercises applying this chapter's real DAO examples and governance critiques, before Chapter 7 turns to smart contract security vulnerabilities in full depth.

EXERCISE 1

Using this chapter's own MakerDAO example, explain why MakerDAO's own continued operation since 2017 is real evidence against the claim that "the 2016 DAO hack proved DAOs themselves are fundamentally unsafe."

→ Solution

EXERCISE 2

Using this chapter's own ConstitutionDAO refund problem and Chapter 2's own gas-fee material, explain specifically why refunding thousands of small

individual contributions on-chain was more expensive, proportionally, than refunding a smaller number of large contributions would have been.

→ Solution

EXERCISE 3

Explain, using this chapter's own multisig explanation and Chapter 3's own access-control material, why a 4-of-7 multisig is generally considered safer for a DAO treasury than a single `onlyOwner` address — and name one real trade-off this added safety comes with.

→ Solution

Quick Reference

- **DAO** — an organization governed by on-chain, token-weighted voting instead of a traditional board.
- **MakerDAO** — launched December 2017, governs the DAI stablecoin, a real, long-running counter-example to "DAOs are inherently unsafe."
- **ConstitutionDAO** — November 2021, raised $47M, lost the Constitution auction at $43.2M, then hit real gas-fee refund problems (some paying $70 in fees on $200 donations).
- **Multisig (Safe/Gnosis Safe)** — requires M-of-N signatures for a treasury transaction, spreading trust beyond a single key.
- **Legal status** — generally unclear worldwide; Wyoming's 1 July 2021 law was the real first US recognition of DAO LLCs.
- **Real governance challenges** — plutocracy (token-concentrated power) and voter apathy.

CHAPTER 7: SMART CONTRACT SECURITY: COMMON VULNERABILITIES

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 7 · Smart Contract Security: Common Vulnerabilities

Chapter 3 introduced Checks-Effects-Interactions as a defense against one specific, real vulnerability — reentrancy. This chapter surveys the broader, real landscape: several more genuine vulnerability classes, each with a real, documented incident behind it, and each teaching a real, specific lesson about how smart contract security differs from ordinary software security.

Reentrancy, Revisited: Beyond a Single Function

Chapter 3 covered the classic case — a function re-entering *itself* before its own state update completes. A subtler real variant is **cross-function reentrancy**: the malicious callback doesn't call the same function again, it calls a *different* function that happens to read or write the same shared state variable, before that state has been updated. Checks-Effects-Interactions still helps, but as real defense-in-depth, OpenZeppelin (Chapter 3) provides a dedicated `ReentrancyGuard` modifier — a real, simple lock flag that blocks *any* re-entry into a protected function, from anywhere, for the duration of the outer call, regardless of which specific function the attacker tries to re-enter through.

Integer Overflow and Underflow: A Real, Language-Level Lesson

Solidity's `uint` types have a fixed bit width, and therefore a fixed maximum value. In Solidity versions **before 0.8.0**, arithmetic that pushed a value past that maximum (overflow) or below zero (underflow) didn't raise an error at all — it silently **wrapped around**, exactly like an odometer rolling over. Subtracting 1 from a `uint` already at 0 didn't fail; it silently became the type's own maximum possible value instead.

PRE-0.8 BEHAVIOR — SILENTLY WRAPS, NO ERROR

```
uint8 balance = 0;
balance -= 1; // wraps silently to 255 — no revert, no warning
```

This was a real, exploitable class of bug in production contracts — a balance check like `require(balance >= amount)` could be defeated entirely if an underflowed balance wrapped around to an enormous number, satisfying the check with a balance that should have been impossible. The real, standard pre-0.8 defense was a library called **SafeMath**, which wrapped every arithmetic operation in an explicit check that reverted on overflow/underflow instead of wrapping.

A REAL, RARE CASE: THE LANGUAGE ITSELF CHANGED

Solidity **0.8.0** made overflow and underflow checks the automatic, built-in default for every arithmetic operation — the exact behavior SafeMath used to bolt on manually now happens without any library at all, and `balance -= 1` on a zero-valued `uint8` today reverts automatically. This is a genuinely notable, real example of a vulnerability class serious enough that an entire language changed its own default behavior specifically to close it, rather than leaving it as a convention developers had to remember to apply themselves.

Access Control: A Real, Permanent Consequence

A real, well-documented incident shows exactly why an access control bug in a smart contract can be categorically worse than the equivalent bug in ordinary software.

Jul 2017	A first, real Parity multisig wallet vulnerability lets an attacker drain a significant amount of ETH directly from several wallets.
Nov 2017	A separate, real incident: many Parity multisig wallets shared one common "library" contract holding their core logic (a design choice made to save on deployment gas costs, per Chapter 2). That library contract's own initialization function had been left callable by anyone. A user triggered it, accidentally becoming its "owner" — and then, apparently in an attempt to understand what had

happened, triggered the library contract's own `selfdestruct`. Because every dependent wallet's own logic pointed at that one now-empty address, this single action instantly and permanently froze every wallet that depended on it — widely reported at the time as well over 500,000 ETH, worth well over $100 million, locked forever.

WHY "PERMANENTLY" ISN'T AN EXAGGERATION

This connects directly to Chapter 3's own immutability discussion: once the shared library contract was destroyed, there was no code left at that address for the dependent wallets to delegate to — not a bug that could be patched, not a function anyone could call to reverse it. The frozen funds are, to this day, still inaccessible. A missing access-control check on an ordinary web application might expose data or allow an unauthorized action that can later be detected and undone; the equivalent gap in an immutable, publicly-callable smart contract can produce consequences with no possible undo at all.

tx.origin vs. msg.sender: A Real, Solidity-Specific Trap

Solidity provides two different ways to identify who's calling a function: `msg.sender` (Chapter 1) — the address that made the *immediate* call — and `tx.origin`, which always resolves to the original, human-controlled address that kicked off the entire transaction, even if it passed through several intermediate contract calls first.

Using `tx.origin` for an authorization check is a real, documented mistake: a malicious contract can trick a legitimate, authorized user into calling *it* (perhaps disguised as something harmless), and then have that malicious contract call the real, sensitive function on the user's behalf. Since `tx.origin` still resolves to the real user's own address no matter how many contracts the call passed through, a check like `require(tx.origin == owner)` passes — even though the actual, immediate caller (`msg.sender`) is the attacker's own malicious contract, not the owner acting directly. Using `msg.sender` instead closes this off entirely, since it would correctly show the malicious contract as the immediate caller.

Oracle Manipulation: Combining Two Chapters Into One Real Attack

Chapter 4 covered how an AMM's own price is a direct, mechanical function of its pool's current token ratio — and Chapter 4 also covered flash loans, which let anyone temporarily command an enormous, uncollateralized sum within one atomic transaction. Put together, they describe a real, documented category of DeFi exploit.

1. A vulnerable lending protocol reads its "price" for some asset directly from a single AMM pool's current ratio, rather than a more robust, manipulation-resistant price source.
2. An attacker takes out a large flash loan and dumps a huge amount of one token into that same AMM pool, temporarily distorting its own price ratio, exactly like Chapter 4's own worked swap example.
3. The vulnerable protocol, reading that now-distorted price, lets the attacker borrow far more than they should genuinely be able to against their real collateral, or liquidate a position that shouldn't actually qualify.
4. The attacker reverses their original AMM trade, repays the flash loan, and keeps the profit — all within the exact same atomic transaction.

This is precisely why Chapter 4 flagged flash loans as a real, documented component of several serious exploits: the vulnerability isn't really the flash loan itself, it's a protocol trusting a price source that can be temporarily moved within a single transaction.

Front-Running and MEV

A submitted transaction doesn't confirm instantly — it sits briefly in a public, visible holding area called the **mempool** before being included in a block. Because anyone can see a pending, profitable transaction before it confirms, a real, documented

practice called **front-running** has emerged: a bot (or, in principle, the very miner or validator deciding block order) can spot a lucrative pending transaction and insert their own transaction ahead of it, paying a higher fee to guarantee it's processed first. This general category of profit extractable purely from a validator or bot's own ability to choose transaction ordering is often called **MEV (Maximal Extractable Value)** — a real, ongoing, actively studied concern across the whole Ethereum ecosystem, not a rare edge case.

Hands-On Exercises

Three exercises applying this chapter's real vulnerability classes before Chapter 8 covers several more real, documented exploits in full case-study depth.

EXERCISE 1

Using this chapter's own real Parity library-freeze incident and Chapter 3's own immutability material, explain specifically why deploying a single shared library contract that many wallets depend on creates a fundamentally different kind of risk than each wallet having its own fully independent copy of the same code.

→ Solution

EXERCISE 2

A contract uses `require(tx.origin == owner)` to protect a sensitive function. Describe, step by step, how a malicious contract could exploit this even though the real owner never intentionally called the sensitive function directly.

→ Solution

EXERCISE 3

Using this chapter's own oracle manipulation flow, explain precisely which real property from Chapter 4 makes step 4 (repaying the flash loan and keeping the profit) reliably possible for the attacker, rather than a risky gamble that might leave them owing an unpaid loan.

→ Solution

Quick Reference

- **Cross-function reentrancy** — a subtler reentrancy variant re-entering a different, shared-state function; `ReentrancyGuard` defends broadly.
- **Integer overflow/underflow** — silently wrapped before Solidity 0.8.0; SafeMath was the real pre-0.8 workaround, now built into the language by default.
- **Access control (the Parity freeze, Nov 2017)** — a callable, uninitialized shared library let a user accidentally trigger `selfdestruct`, permanently freezing well over 500,000 ETH with no possible fix.
- `tx.origin` **vs.** `msg.sender` — `tx.origin` resolves through intermediate contracts, enabling a real phishing pattern; `msg.sender` doesn't.
- **Oracle manipulation** — a flash loan temporarily distorts an AMM price a vulnerable protocol trusts directly, exploited and repaid within one atomic transaction.
- **Front-running / MEV** — visible pending transactions in the public mempool can be reordered ahead of for profit.

CHAPTER 8: REAL, DOCUMENTED EXPLOITS & WHAT THEY TEACH

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 8 · Real, Documented Exploits & What They Teach

Chapter 7 named the vulnerability classes. This chapter applies that taxonomy to two real, fully documented, large-scale exploits — one a genuine smart contract flaw, the other a reminder that "smart contract security" isn't the only real attack surface in Web3 at all.

Case Study 1: The Poly Network Exploit (August 2021)

Poly Network is a real protocol for moving assets between different blockchains. On **10 August 2021**, an attacker exploited a real, genuine access control flaw in its cross-chain contract logic — exactly the vulnerability class Chapter 7 covered — to trick the protocol into treating the attacker's own address as an authorized "keeper" able to direct transfers. The result: over **$610 million** moved out to addresses the attacker controlled, spread across Ethereum, Binance Smart Chain, and Polygon — the largest DeFi hack ever recorded up to that point.

10 Aug 2021	The exploit occurs; over $610 million is drained across three chains.
11 Aug 2021	The attacker publicly announces, one day later, that they intend to return the funds.
13 Aug 2021	$340 million worth has already been returned; the rest moves to a multisig (Chapter 6) jointly controlled by the attacker and Poly Network.
25 Aug 2021	The full return completes — just 15 days after the original exploit. Poly Network publicly offers the anonymous attacker a $500,000 bug bounty and a "chief security advisor" role.

A REAL, GENUINELY DEBATED ENDING

Whether to call this attacker a "white hat" security researcher who made a point, or simply a thief who got cold feet (or feared being traced, since every transaction is permanently public per Course 1's own chapters on transparency), is a real, open question security professionals have genuinely disagreed about publicly. Some raised a real, honest concern that publicly rewarding this outcome risks normalizing "steal it, then negotiate a reward for giving it back" as an acceptable playbook. This chapter isn't resolving that debate — it's worth knowing the ending is genuinely contested, not a clean, uncomplicated resolution.

Case Study 2: The Ronin Network Bridge Hack (March 2022)

Ronin is a real, dedicated blockchain built for the game Axie Infinity, connected back to Ethereum via a "bridge" — a system for moving assets between the two chains. On **23 March 2022**, attackers stole a real, documented **173,600 ETH and 25.5 million USDC**, worth approximately **$620 million** at the time — discovered only **six days later**. The FBI later attributed the attack to the Lazarus Group and APT38, real North Korean state-sponsored hacking groups.

A GENUINELY DIFFERENT KIND OF ATTACK SURFACE

Unlike Poly Network, this wasn't primarily a smart contract code vulnerability of the kind Chapter 7 catalogued. The real, publicly reported account is that the attackers compromised validator infrastructure directly — obtaining enough real, valid private keys to forge the authorizations needed to approve withdrawals, reportedly through social engineering rather than exploiting a flaw in the bridge's own Solidity code. The exact operational specifics (which keys, exactly how many, and exactly how they were obtained) are widely reported in security-industry coverage, but this course couldn't independently re-verify every specific detail this session — the honest, important, well-established

takeaway is the category of attack: this targeted the humans and infrastructure operating the system, not a bug in the deployed contract itself.

Sky Mavis, Ronin's own developer, publicly reimbursed all affected users afterward — a real, centralized company absorbing a loss that a fully decentralized system, per Course 1's own trust-minimization theme, wouldn't have had any single party positioned to guarantee.

What Both Cases Teach Together

	Poly Network	Ronin
Real cause	A genuine access-control flaw in deployed contract code	Compromised validator/infrastructure access, not a contract code bug
Real scale	Over $610 million	Approximately $620 million
Real outcome	Funds fully returned within 15 days	Funds not recovered; users reimbursed by the company
Chapter 7 category	Access control	Outside this course's own contract-code taxonomy entirely

Chapter 7's own vulnerability taxonomy is real and useful — but Ronin is an honest, important reminder that it only covers part of the real attack surface. A perfectly audited, flawlessly written smart contract can still sit behind a system whose real weak point is a human being tricked into handing over access, or a private key stored insecurely. Real Web3 security has to cover both the code and everything operating around it.

Hands-On Exercises

Three exercises applying both real case studies before Chapter 9 turns to regulation, custody, and the broader real risks of Web3.

EXERCISE 1

Using Chapter 7's own vulnerability taxonomy, explain specifically which category the Poly Network exploit falls under, and describe in your own words what the attacker's contract interaction must have achieved to be treated as an authorized "keeper."

→ Solution

EXERCISE 2

A junior developer says: "We had our smart contracts professionally audited, so our project is now fully secure against a major hack." Using this chapter's own Ronin case study, explain what this statement overlooks.

→ Solution

EXERCISE 3

Using this chapter's own comparison table, explain why Ronin's users could be made whole by Sky Mavis's own reimbursement, while a comparable loss on a fully decentralized protocol with no single company behind it typically couldn't be resolved the same way.

→ Solution

Quick Reference

- **Poly Network (10 Aug 2021)** — a real cross-chain access-control flaw, over $610 million drained, fully returned within 15 days after the attacker announced their intent the very next day.
- **Ronin Network (23 Mar 2022)** — approximately $620 million (173,600 ETH + 25.5M USDC) stolen via compromised validator access, attributed to North Korea's Lazarus Group/APT38; users reimbursed by Sky Mavis.

- **The real, shared lesson** — smart contract code vulnerabilities (Chapter 7) are real and serious, but infrastructure and human factors are a genuinely separate, equally real attack surface.

CHAPTER 9: REGULATION, CUSTODY & THE REAL RISKS OF WEB3

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 9 · Regulation, Custody & the Real Risks of Web3

Chapters 7 and 8 covered risk that lives inside a smart contract, or in the infrastructure directly operating one. This chapter steps back further — to a risk category that has nothing to do with code at all, and to the real, still-evolving legal frameworks governments have built in response to it.

Custody: Who Actually Holds Your Assets?

Course 1 covered private keys as the real, sole source of control over an on-chain asset — "not your keys, not your coins" is the community's own well-known shorthand for that principle. A **custodial** exchange breaks that principle by design: when you deposit crypto onto a centralized exchange, the exchange itself holds the private keys, and your account balance is really just an internal database entry the exchange promises corresponds to real, held assets. A **non-custodial** wallet, by contrast, keeps you holding the actual private key the entire time.

THE REAL, PRACTICAL TRADE-OFF

Custodial exchanges are genuinely more convenient — easier onboarding, password resets, no risk of permanently losing a seed phrase. Non-custodial wallets are genuinely more aligned with Course 1's own trust-minimization theme — nothing can freeze, lose, or misuse your assets except you. Neither option is simply "correct"; each trades real convenience against real, different categories of risk.

Case Study: The FTX Collapse (November 2022)

FTX, founded in **May 2019** by Sam Bankman-Fried and Gary Wang, grew into one of the world's largest cryptocurrency exchanges, reaching a real, reported **$32 billion**

valuation before its collapse. The exchange filed for Chapter 11 bankruptcy on **11 November 2022**.

2 Nov 2022	A CoinDesk report reveals that Alameda Research, FTX's own affiliated trading firm, held a large share of its balance sheet in FTT, FTX's own native token — a real, circular dependency.
Early Nov 2022	Binance announces plans to sell its own FTT holdings, triggering a real, rapid loss of confidence and a wave of customer withdrawals.
11 Nov 2022	FTX files for Chapter 11 bankruptcy. New CEO John J. Ray III later calls it "a complete failure of corporate controls."
12 Dec 2022	Bankman-Fried is arrested and charged with fraud and money laundering.
2 Nov 2023	A jury convicts Bankman-Fried on all seven real counts, including wire fraud and conspiracy to commit securities fraud.
28 Mar 2024	He is sentenced to 25 years in prison and ordered to forfeit $11.02 billion.

The real, documented allegation at the center of the case: FTX had lent approximately **$10 billion** in customer deposits to Alameda Research, and Alameda had reportedly been granted a secret exemption from FTX's own automatic liquidation safeguards — the exact kind of privileged-access failure Chapter 7's own access-control category describes, just happening inside a company's internal systems rather than inside deployed Solidity code.

THE REAL IRONY THIS CASE STUDY DELIVERS

FTX was a fully centralized, custodial company — the exact opposite of the trust-minimized, self-custodied model Course 1 spent its own chapters explaining. Depositing funds onto an exchange like FTX meant trusting a company's own internal controls completely, with none of the on-chain transparency or self-custody guarantees a personal wallet actually provides. The collapse is a real,

costly demonstration of exactly the risk "not your keys, not your coins" warns against.

Regulation: The US Approach

US securities regulation still leans heavily on a real, 78-year-old legal test. In **SEC v. W. J. Howey Co.** (328 U.S. 293, decided **27 May 1946**), the Supreme Court held that an *investment contract* — and therefore a regulated security — exists whenever there is (1) an investment of money, (2) in a common enterprise, (3) with an expectation of profit, (4) derived primarily from the efforts of others. The real, ongoing debate over whether a given token counts as a security largely comes down to whether it satisfies all four of these prongs.

Regulation: The EU's MiCA

The European Union took a genuinely different approach: a dedicated, purpose-built regulation rather than an 80-year-old test stretched to fit a new technology. The Markets in Crypto-Assets regulation (**MiCA**) was adopted by the European Parliament on **20 April 2023** and applied in full via two real phases — stablecoins (asset-referenced and e-money tokens) from **30 June 2024**, and other crypto-assets plus crypto-asset service providers (exchanges, custodial wallet providers) from **30 December 2024**.

	US (Howey Test)	EU (MiCA)
Real origin	A 1946 Supreme Court ruling about Florida citrus groves, applied by analogy	A dedicated 2023 regulation written specifically for crypto-assets
Real mechanism	Case-by-case application of a	A direct regulatory framework

	four-part test to determine security status	covering issuers, exchanges, and custodial wallet providers by name
Real effect on this chapter's own FTX case	Would ask whether specific FTX products (like the FTT token) met the Howey criteria	Would have directly regulated FTX's own custodial and exchange functions had it applied at the time

Risks Beyond Code and Custody

Two more real, well-documented Web3-specific risk categories round out this chapter's own survey. A **rug pull** is when a project's own developers abandon it and drain its liquidity pool (Chapter 4) after attracting real investor funds — a risk that has nothing to do with a smart contract bug at all, since the contract may be working exactly as its own creators always intended it to. A `approve()` **phishing exploit** targets a real, genuine ERC-20 feature (Course 1's own token standard) that lets a user grant a smart contract ongoing permission to move tokens on their behalf — a malicious site can trick a user into granting that permission to an attacker-controlled contract, which then drains the approved tokens directly, with no bug in the token contract itself required.

A GENUINE PATTERN ACROSS THIS WHOLE CHAPTER

FTX, rug pulls, and approve() phishing share one real, unifying lesson: none of them require a single line of flawed Solidity code. Chapters 7 and 8 covered risk inside the code; this chapter covers risk around it — who holds your keys, who regulates the platform you're using, and who you trust with a permission you granted. A perfectly secure smart contract cannot protect against any of these on its own.

Hands-On Exercises

Three exercises applying this chapter's own custody, regulation, and risk material before Chapter 10's own capstone brings the full course together.

EXERCISE 1

Explain, using this chapter's own custodial-vs-non-custodial distinction, exactly why depositing funds onto FTX exposed customers to a real risk that holding the same funds in a personal, non-custodial wallet never would have.

→ Solution

EXERCISE 2

Apply the real, four-part Howey Test to a hypothetical new token sold with the pitch "buy this token now, and our development team will build products that make it valuable later." Explain which of the four prongs this pitch appears to satisfy, and why.

→ Solution

EXERCISE 3

Explain why a rug pull and an approve() phishing exploit both fall outside Chapter 7's own smart contract vulnerability taxonomy, even though both can result in a user losing real funds from a Web3 application.

→ Solution

Quick Reference

- **Custodial vs. non-custodial** — who actually holds the private key: the platform, or you.
- **FTX (11 Nov 2022 bankruptcy)** — ~$10 billion in customer funds lent to Alameda Research; SBF convicted on all 7 counts (2 Nov 2023), sentenced to 25 years (28

Mar 2024).

- **Howey Test (1946)** — investment of money, common enterprise, expectation of profit, from others' efforts — the US's own case-by-case security test.
- **MiCA (EU, adopted 20 Apr 2023, applied in full by 30 Dec 2024)** — a dedicated regulatory framework for crypto-assets and service providers.
- **Rug pulls and approve() phishing** — real Web3 risks that require no smart contract bug at all.

CHAPTER 10: CAPSTONE — DESIGNING (AND AUDITING) A SMALL DEFI APPLICATION

Smart Contracts, DeFi & Web3 Security

Course 2 · Chapter 10 · Capstone: Designing (and Auditing) a Small DeFi Application

Nine chapters built every piece separately: writing Solidity, understanding gas, real design patterns, DeFi mechanics, the dApp stack, DAOs, a vulnerability taxonomy, two real documented exploits, and the real risks that live outside the code entirely. This capstone puts it together — designing a small staking vault, auditing an early draft against Chapter 7's own taxonomy, finding a real, genuine vulnerability, fixing it, and then asking Chapter 9's own custody and regulatory questions of the finished contract.

The Design: SimpleStake

The application is deliberately small: users deposit an ERC-20 token, earn no complex yield mechanics (kept out of scope, matching this course's own consistent honesty about what it doesn't cover), and can withdraw their full deposit back out at any time. Even this minimal scope is enough to genuinely exercise every real tool this course has built.

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

contract SimpleStakeDraft {
    mapping(address => uint256) public balances;
    IERC20 public immutable token;

    constructor(address _token) {
        token = IERC20(_token);
    }

    function deposit(uint256 amount) external {
        token.transferFrom(msg.sender, address(this), amount);
        balances[msg.sender] += amount;
    }

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "insufficient balance");
```

```
        token.transfer(msg.sender, amount);
        balances[msg.sender] -= amount;
    }
}
```

The Audit: Applying Chapter 7's Own Taxonomy

Reading this draft against Chapter 7's own real vulnerability categories, one by one:

Chapter 7 Category	Finding Against This Draft
Integer overflow/underflow	Clean — the `pragma solidity ^0.8.20` line means this contract compiles with Solidity's own built-in overflow and underflow checks, active by default in every version since Solidity 0.8.0. `balances[msg.sender] -= amount` would revert automatically rather than underflow.
Access control	Not applicable here — there's no owner-only or privileged function in this small a contract to protect in the first place.
Reentrancy	**A real, genuine finding.** `withdraw()` calls `token.transfer()` before updating `balances[msg.sender]`. See below.

THE REAL VULNERABILITY

This is the exact structural shape Chapter 7 already named: an external call happening before the contract's own internal state is updated. If `token` were a malicious or compromised ERC-20 implementation with a hook that calls back into `SimpleStakeDraft` during `transfer()`, that reentrant call would see `balances[msg.sender]` still at its original, pre-withdrawal value — and could call `withdraw()` again before the first call ever reaches its own final line. This is structurally the same category of flaw behind The DAO hack Chapter 7 used to

introduce reentrancy in the first place, just relocated from ETH transfers to a token contract's own hook.

The Fix: Checks-Effects-Interactions

```
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "insufficient balance");  // Check
    balances[msg.sender] -= amount;                                   // Effect
    token.transfer(msg.sender, amount);                               // Interacti
}
```

Reordering these three lines to match Chapter 3's own checks-effects-interactions pattern closes the vulnerability completely. By the time the external `transfer()` call happens, `balances[msg.sender]` has already been reduced — so even a fully malicious reentrant call sees a balance too low to withdraw against a second time.

AUDITING IN PRACTICE

This is genuinely what a real audit looks like at small scale: reading a specific contract against a known taxonomy of vulnerability classes, one category at a time, rather than searching for bugs with no structure at all. A single real finding — missed here in the draft's very first version — was enough to fully compromise every user's own deposited funds.

Custody and Regulation, Applied to This Contract

Chapter 9's own questions apply directly here too, and the honest answers are more nuanced than "smart contracts are non-custodial by default." While a user's tokens sit inside `SimpleStake`, that user does not hold the private key controlling those specific tokens anymore — the contract's own code does, and the user is trusting that code (not a company, but code all the same) to behave exactly as written. This is a real, genuine form of custody, just one governed by auditable, public logic rather than a company's own internal, unverifiable controls, which is precisely why the fix above matters so much: the contract's own correctness *is* the custody guarantee.

On regulation: if `SimpleStake` paid out yield and were marketed with the promise that the project's own team would keep improving the protocol to increase returns, it would map onto several of the Howey Test's own four prongs directly, the same way Chapter 9's own hypothetical token pitch did. A staking contract that simply returns exactly what a user deposited, with no yield and no promotional promise of future value from anyone else's efforts, sits much further from that real legal test — a genuine, material difference this capstone's own minimal design happens to land on the safer side of.

Chapter-by-Chapter Attribution

Chapter	What It Contributed to This Capstone
1	The Solidity syntax itself — state variables, functions, a constructor, external visibility
2	Why the pragma version matters, and what each function call actually costs in gas
3	The checks-effects-interactions pattern that fixes the real vulnerability found here
4	The deposit/withdraw shape this small vault borrows from real DeFi lending and staking design
5	How a real frontend dApp would call `deposit()` / `withdraw()` through a connected wallet
6	Why a real production version of this contract would likely be owned by a multisig or DAO, not one address
7	The vulnerability taxonomy this capstone's own audit applied directly, category by category
8	The real reentrancy lineage this exact bug shares with a documented historical exploit
9	The custody and regulatory questions applied to this contract's own specific design

WHAT THIS COURSE DOESN'T COVER

This capstone is a genuine, worked demonstration of the audit process at small scale — it is not a substitute for real production security tooling. It doesn't cover static analysis tools like Slither, formal verification, professional third-party audit firms, or the real, ongoing bug-bounty programs serious protocols run continuously rather than as a one-time check. Real capital deployed on a real contract deserves real professional review, not a single pass through one course's own taxonomy.

Hands-On Exercises

Three closing exercises applying the full course toolkit to fresh scenarios.

EXERCISE 1

A colleague proposes fixing the reentrancy vulnerability by adding a `require(!inProgress)` "locking" flag instead of reordering the three lines into checks-effects-interactions order. Explain whether this alternative fix would genuinely work, and which approach this chapter's own fix actually used.

→ Solution

EXERCISE 2

Using this chapter's own custody discussion, explain why a user's tokens sitting inside SimpleStake are not the same thing as "fully non-custodial," even though no company or exchange is involved anywhere in the design.

→ Solution

EXERCISE 3

Redesign SimpleStake so it pays depositors a fixed yield, marketed with the promise that the development team will keep building features to grow returns. Using Chapter 9's own Howey Test material, explain how this specific change shifts the contract's own real regulatory exposure.

→ Solution

Quick Reference — Course Complete

- **Chapters 1–10** — Solidity basics, the EVM and gas, design patterns, DeFi mechanics, dApps, DAOs, a vulnerability taxonomy, two real documented exploits, regulation and custody, and this capstone auditing a real vulnerability end to end.
- **Course complete** — Smart Contracts, DeFi & Web3 Security is now finished, 10/10 chapters.
- **The full Blockchain & Web3 project is now complete** — Blockchain & Web3 Fundamentals (10 chapters) plus Smart Contracts, DeFi & Web3 Security (10 chapters), 20 chapters total across both courses.