Audit 101 – Smart Contract Vulnerabilities

Let's break down "Audit 101 – Smart Contract Vulnerabilities" into a comprehensive guide, covering the philosophy, the technical stack, and a detailed look at common vulnerabilities.

Part 1: The Philosophy of a Smart Contract Audit :

A smart contract audit is not a bug hunt; it's a **systematic security review**. Unlike traditional software, deployed contracts are often immutable, and bugs can lead to the irreversible loss of millions of dollars. The goal is to identify and remediate security risks, logical flaws, and inefficiencies before deployment.

Core Principles:

- **Trustlessness & Verification:** The code is the law. The audit verifies that the law behaves exactly as intended under all conditions.
- **Defense in Depth:** Multiple layers of security should exist (e.g., access controls, checks, emergency stops).
- **Pessimism:** Assume users are malicious, inputs are malformed, and the system is under active attack.
- **Simplicity:** Complexity is the enemy of security. The simpler the code, the easier it is to verify.

Part 2: The Tech Stack of a Smart Contract Auditor :

An auditor needs to be proficient across a wide range of technologies and tools.

1. Core Development & Blockchain Knowledge

- **Solidity:** The primary language for Ethereum and EVM-compatible chains. Deep knowledge of its quirks, the EVM, and gas optimization is essential.
- **Vyper:** A Pythonic language for the EVM, less common but still relevant.
- **Rust:** For auditing Solana, NEAR, Polkadot, and other non-EVM chains.
- **Cairo:** For StarkNet contracts.
- **Ethereum Virtual Machine (EVM):** Understanding opcodes, storage layouts, calldata, and how transactions are executed is fundamental.
- **Blockchain Fundamentals:** How transactions, blocks, gas, and consensus mechanisms work.

2. Testing & Development Frameworks

- **Hardhat:** The industry standard for Ethereum development, testing, and debugging. Its console.log and stack traces are invaluable.
- **Foundry:** A rapidly growing, powerful framework written in Rust. Its fuzzing capabilities (Forge) and direct EVM manipulation tool (Cast) are game-changers for auditors.
- **Truffle Suite:** A legacy framework, still seen in older codebases.

3. Static Analysis Tools

These tools automatically scan code for known patterns and vulnerabilities.

- **Slither:** The best static analysis framework for Solidity. It detects a wide range of vulnerabilities and provides code quality metrics.
- **Mythril:** A security analysis tool that uses symbolic execution to find bugs.
- **Semgrep:** A generic static analysis tool with custom rules for Solidity.

4. Dynamic Analysis & Fuzzing Tools

These tools execute the code with a vast number of random inputs to find edge-case failures.

- **Foundry Fuzzer (**forge test**):** Integrated fuzzing that is incredibly fast and effective. Becoming the go-to for stateful fuzzing.
- **Echidna:** A sophisticated fuzzer/property-based tester. You define "invariants" (e.g., "the total supply should never decrease"), and it tries to break them.
- **Harvey:** Often used with Mythril for more efficient fuzzing.

5. Formal Verification & Advanced Tools

- **Certora Prover:** A formal verification tool that uses mathematical proofs to check if a contract satisfies specified rules. Used by top-tier protocols but requires significant expertise.
- **Manticore:** A symbolic execution tool for analyzing smart contracts and binaries.

6. Manual Review Tools

- **VS Code with Extensions:** (Solidity, Hardhat, etc.)
- **Code Differencing Tools:** (git diff, Meld) to compare changes between audit rounds.

Part 3: Detailed Breakdown of Common Smart Contract Vulnerabilities :

This is the core of the audit. Vulnerabilities are often categorized by the SWC Registry (Smart Contract Weakness Classification), a curated list for the community.

Category 1: Reentrancy

The most famous DeFi vulnerability, responsible for the DAO hack.

- **What it is:** A malicious contract calls back into the vulnerable contract *before* the first function call has finished, re-entering the code and manipulating state.
- **How it happens:** Using an external call (e.g., sending ETH to an unknown address) *before* updating the internal state.
- **Example (Simplified):**

solidity

```
// VULNERABLE CODE
contract VulnerableBank {
    mapping(address => uint) public balances;
    function withdraw() public {
        uint amount = balances[msg.sender];
        (bool success, ) = msg.sender.call{value: amount}(""); // <-- External call
        require(success);
        balances[msg.sender] = 0; // <-- State update happens AFTER the call
    }
}
contract Attacker {
    function attack() public {
        // This will call back into `withdraw` multiple times before the balance is set to 0.
```

```
        vulnerableBank.withdraw();
    }
    receive() external payable {
        if (address(vulnerableBank).balance >= 1 ether) {
            vulnerableBank.withdraw();
        }
    }
}
```

- **Mitigation:** Use the **Checks-Effects-Interactions** pattern.
    1. **Checks:** Validate conditions (e.g., require(balance > 0)).
    2. **Effects:** Update all internal state *first* (balances[msg.sender] = 0;).
    3. **Interactions:** Perform external calls *last* (msg.sender.call{value: amount}("")).
- **Advanced Mitigation:** Use reentrancy guards (e.g., OpenZeppelin's ReentrancyGuard).

Category 2: Access Control

Failure to properly restrict who can call sensitive functions.

- **What it is:** A function that should be callable only by the owner/admin or specific roles is left open to everyone.
- **How it happens:** Missing or incorrect function modifiers.
- **Example:**

solidity

```
// VULNERABLE CODE
function setAdmin(address newAdmin) public {
    // Missing: require(msg.sender == owner, "Not owner");
    admin = newAdmin;
}
```

- **Mitigation:**
    - Use modifiers like onlyOwner from OpenZeppelin's Ownable contract.
    - For complex roles, use OpenZeppelin's AccessControl.

Category 3: Arithmetic Over/Underflows

Solidity 0.8.x introduced built-in checks, but they are still relevant, especially in older code or with unchecked blocks.

- **What it is:** An unsigned integer (uint) going below 0 causes an underflow (wraps to a massive number). An operation exceeding the maximum uint size causes an overflow.
- **How it happens:** Subtracting more than is available, or multiplying large numbers.
- **Example:**

solidity

```
// VULNERABLE CODE (in Solidity < 0.8)
mapping(address => uint) public balances;

function transfer(address to, uint amount) public {
  require(balances[msg.sender] - amount >= 0); // This underflows!
  balances[msg.sender] -= amount;
  balances[to] += amount;
}
```

- **Mitigation:**
  - Use Solidity >= 0.8.0 (which has built-in checks).
  - For Solidity < 0.8, use SafeMath library from OpenZeppelin.
  - Be extremely careful when using unchecked { } blocks for gas optimization.

Category 4: Oracle Manipulation

Using a single, manipulable data source for critical information like asset prices.

- **What it is:** An attacker can artificially move the price on a DEX (e.g., with a flash loan) to manipulate a price oracle that relies on that DEX, causing a protocol to misprice assets and allowing the attacker to drain funds.
- **Mitigation:**
  - Use **decentralized oracles** like Chainlink, which aggregate data from many sources.
  - Use **Time-Weighted Average Prices (TWAPs)** from DEXes like Uniswap V3, which are harder to manipulate in a short time frame.

Category 5: Frontrunning

Seeing a pending transaction in the mempool and submitting a higher-gas transaction to get executed first.

- **What it is:** Not a "vulnerability" in the classical sense, but a fundamental property of public blockchains. It can be exploited for profit (e.g., sandwich attacks on DEX trades).
- **Mitigation:**
  - Use **Commit-Reveal schemes** where users first commit to an action (with a hash) and later reveal it.
  - **Submarine Sends:** Sending transactions through private relayers (like Flashbots) to avoid the public mempool.
  - Set maximum slippage tolerances on trades.

Category 6: Logic & Business Logic Errors

This is the broadest category and requires deep manual review. It's where the auditor ensures the code does what the specification says.

- **Examples:**
  - **Incorrect Fee Calculation:** A protocol takes a 0.1% fee, but a rounding error in the code means it sometimes takes 0% or 0.2%.
  - **Incorrect Reward Distribution:** A staking contract mints too many or too few rewards due to a flaw in the time-based calculation.

- **Improper Tokenomics:** Allowing the same NFT to be used as collateral across multiple loans simultaneously.

Category 7: Denial of Service (DoS)

Making a contract unusable for legitimate users.

- **How it happens:**
  - **Block Gas Limit:** A function that loops over an array that can grow unbounded (e.g., airdropping to 10,000 users) may eventually exceed the block gas limit, making it impossible to call.
  - **Forcing Reverts:** An attacker can make themselves the recipient of a state-changing action (e.g., in a game) and have their contract always revert on receipt, blocking a critical process.

The Audit Process in Practice

1. **Specification Review:** Understand what the code is *supposed* to do.
2. **Automated Scanning:** Run Slither, Mythril, etc., for a first pass.
3. **Manual Code Review:** Line-by-line analysis, often following function call graphs and data flows.
4. **Testing & Fuzzing:** Write unit tests and invariant tests (with Foundry/Echidna) to break the code.
5. **Reporting:** Create a detailed report with findings, categorized by severity (Critical, High, Medium, Low, Informational). Each finding includes a description, code location, impact, and recommendation.
6. **Remediation & Re-audit:** The developers fix the issues, and the auditor reviews the fixes to ensure they are correct and don't introduce new bugs.

Conclusion

Smart contract auditing is a demanding but critical discipline that sits at the intersection of cryptography, finance, and software engineering. It requires a paranoid mindset, deep technical expertise, and a comprehensive toolkit. By understanding the common vulnerability patterns and the tools to find them, you can begin the journey to securing the next generation of decentralized applications.
