L43: Smart Contract Security

Module F: Advanced Topics

Blockchain & Cryptocurrency Course

December 2025

The Stakes of Smart Contract Security

- **Code is Law**: Smart contracts are immutable and self-executing
- **High-Value Targets**: DeFi protocols hold billions in assets
- **Irreversibility**: Bugs cannot be patched without upgradeable contracts
- **Losses to Date**: \$3B+ stolen from smart contract exploits (2016-2024)
- **Asymmetry**: One vulnerability can drain entire protocol
- **Composability Risk**: Vulnerabilities cascade across protocols

- **Paradigm Shift**: Traditional software debugging $\rightarrow$ Formal verification $+$ economic incentives

Top Smart Contract Vulnerabilities

1. **Reentrancy**: Recursive external calls before state updates
2. **Integer Overflow/Underflow**: Arithmetic beyond type limits
3. **Access Control**: Unauthorized function execution
4. **Oracle Manipulation**: Reliance on manipulable price feeds
5. **Front-Running/MEV**: Transaction ordering exploitation
6. **Denial of Service**: Gas limit attacks, unbounded loops
7. **Timestamp Dependence**: Miner-manipulable block timestamps
8. **Uninitialized Storage**: Default values exploited
9. **Delegate Call Injection**: Context preservation attacks
10. **Signature Replay**: Reusing valid signatures

Reentrancy: The DAO Hack (2016)

- **The DAO**: Decentralized autonomous organization, \$150M raised
- **Vulnerability**: Reentrancy in withdrawal function
- **Vulnerable Pattern**:

```
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);
    msg.sender.call{value: amount}("");  // External call BEFORE state update
    balances[msg.sender] -= amount;      // State updated AFTER
}
```

- **Attack**: Attacker's fallback function recursively calls `withdraw()`
- **Result**: Drained \$60M before running out of gas
- **Aftermath**: Ethereum hard fork (ETH vs ETC split)

Reentrancy: Attack Mechanism

1. Attacker deposits 1 ETH, balance = 1
2. Calls `withdraw(1)`
3. Contract sends 1 ETH to attacker
4. **Attacker's fallback function executes**:
   - Balance still = 1 (not yet updated)
   - Calls `withdraw(1)` again
   - Contract checks balance (still 1), sends another 1 ETH
5. Recursion continues until gas limit or contract drained
6. **Final state**: Attacker withdrew N ETH, balance decremented only once

Root Cause: State updated after external call (violates Checks-Effects-Interactions pattern)

Reentrancy: Defense Mechanisms

1. **Checks-Effects-Interactions Pattern**:

```
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);  // Check
    balances[msg.sender] -= amount;           // Effect (state update FIRST)
    msg.sender.call{value: amount}("");       // Interaction (external call LAST)
}
```

2. **Reentrancy Guard (Mutex)**:

```
bool locked;
modifier noReentrant() {
    require(!locked);
    locked = true;
    _;
    locked = false;
}
```

3. **OpenZeppelin ReentrancyGuard**: Industry-standard implementation

Integer Overflow/Underflow

- **Problem**: Solidity ¡0.8.0 does not check arithmetic overflow
- **Overflow Example**:

```
uint8 balance = 255;
balance += 1;  // Wraps to 0 (255 + 1 = 256 mod 256)
```

- **Underflow Example**:

```
uint8 balance = 0;
balance -= 1;  // Wraps to 255 (0 - 1 = -1 mod 256)
```

- **Real Attack**: BeautyChain (BEC) token (2018) – overflow allowed minting billions of tokens
- **Mitigation**: Use SafeMath library or Solidity 0.8.0+ (built-in overflow checks)

Access Control Vulnerabilities

Common Mistakes

- Missing `onlyOwner` modifier
- Default function visibility (`public`)
- Unprotected `selfdestruct`
- Constructor typo (pre-Solidity 0.5.0)

Example: Parity Wallet Hack

- `initWallet()` function unprotected
- Attacker became owner
- Called `selfdestruct`, froze $300M

Secure Pattern

```
address public owner;

modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}

function withdraw()
    public
    onlyOwner
{
    // Protected function
}
```

Best Practice:

- Use OpenZeppelin `Ownable`
- Explicit visibility modifiers
- Role-based access control (RBAC)

Oracle Manipulation

- **Problem**: DeFi protocols rely on external price data (oracles)
- **Vulnerable Pattern**: Using single DEX as price oracle

```
uint price = getReserveRatio(uniswapPair);  // Manipulable!
```

- **Attack**: Flash loan to manipulate pool price
  1. Borrow 10,000 ETH
  2. Buy all TOKEN in pool (inflates price 10x)
  3. Oracle reads manipulated price
  4. Exploit protocol logic (borrow, liquidate, mint)
  5. Restore pool state, repay flash loan
- **Real Examples**: Harvest Finance ($34M), Cream Finance ($130M)

Oracle Manipulation: Defenses

1. **Time-Weighted Average Price (TWAP)**:

```
uint twap = uniswapV2Oracle.consult(token, period);  // Average over N blocks
```

   - Uniswap V2 accumulator design
   - Attacker must manipulate price over multiple blocks (expensive)
2. **Chainlink Decentralized Oracles**:
   - Multiple independent data sources
   - Aggregated via median/consensus
   - Crypto-economic security
3. **Multiple Oracle Sources**: Combine Chainlink + TWAP + Band Protocol
4. **Circuit Breakers**: Pause if price deviation ¿X%

Front-Running and MEV

- **MEV (Maximal Extractable Value)**: Profit from transaction ordering
- **Front-Running Attacks**:
  1. Attacker monitors mempool for profitable transactions
  2. Submits identical transaction with higher gas price
  3. Miner includes attacker's tx first
- **Sandwich Attacks**:
  1. Victim submits large swap (A $\rightarrow$ B)
  2. Attacker front-runs: Buy B (raises price)
  3. Victim's swap executes at worse price
  4. Attacker back-runs: Sell B (profit from price difference)
- **Impact**: \$600M+ MEV extracted in 2023

MEV Mitigation Strategies

1. **Commit-Reveal Schemes**:
   - Commit hash of transaction in block N
   - Reveal actual transaction in block N+1
   - Too slow for time-sensitive operations
2. **Flashbots**: Private transaction pool (MEV-Boost)
   - Users submit bundles directly to block builders
   - Bypass public mempool
   - MEV democratization
3. **Slippage Protection**: Set maximum acceptable price deviation
4. **Batch Auctions**: Aggregate orders, execute at uniform price
5. **Threshold Encryption**: Decrypt transactions only after inclusion

Denial of Service (DoS) Attacks

Gas Limit DoS

```
function distribute() public {
    for (uint i=0; i<users.length; i++) {
        users[i].transfer(amount);
    }
}
```

- Unbounded loop
- Attacker adds many addresses
- Gas cost exceeds block limit
- Function permanently fails

Mitigation: Pull Over Push

```
mapping(address => uint) public balances;

function withdraw() public {
    uint amount = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(amount);
}
```

- Users withdraw individually
- No loops over unbounded arrays
- Batch processing with pagination

Timestamp Dependence

- **Problem**: Relying on `block.timestamp` for critical logic

```
require(block.timestamp > deadline);  // Miner-manipulable!
uint randomness = uint(keccak256(block.timestamp));  // Predictable!
```

- **Miner Power**: Can manipulate timestamp by 900 seconds
- **Attack Vector**: Gambling contracts, auctions, time-locks
- **Mitigation**:
  - Use block numbers instead of timestamps (when possible)
  - Chainlink VRF for randomness
  - Tolerate 15-minute timestamp variance in design

Delegate Call Vulnerabilities

- **Delegate Call**: Execute code in another contract's context (preserves `msg.sender`, storage)
- **Danger**: Storage layout must match exactly

```
// Proxy Contract
address implementation;  // Storage slot 0

function upgradeTo(address newImpl) public {
    implementation = newImpl;
}

fallback() external payable {
    implementation.delegatecall(msg.data);
}
```

- **Attack**: If implementation writes to slot 0, proxy's `implementation` variable overwritten
- **Parity Wallet Hack (2017)**: Delegate call to uninitialized library, $30M stolen
- **Mitigation**: Careful storage layout design, use upgradeable proxy patterns (TransparentProxy, UUPS)

Signature Replay Attacks

- **Problem**: Valid signature can be reused across different contexts
- **Attack Scenario**:
  1. User signs message authorizing token transfer
  2. Attacker captures signature
  3. Replays signature on different chain or contract
- **Mitigation – EIP-712 Typed Data**:

```
struct Transfer {
    address to;
    uint amount;
    uint nonce;
    uint chainId;
}
bytes32 digest = keccak256(abi.encode(domainSeparator, Transfer));
```

  - Include `nonce` (increments per signature)
  - Include `chainId` (prevents cross-chain replay)
  - Include contract address in domain separator

Security Tools: Static Analysis

Tool	Type	Detects
Slither	Static analyzer	Reentrancy, overflow, access control
Mythril	Symbolic execution	Integer bugs, unchecked calls
Securify	Automated verifier	Compliance with security patterns
Manticore	Symbolic execution	Path exploration, concrete exploits
Echidna	Fuzzer	Property-based invariant violations

- **Slither**: Fast, easy integration (Solidity AST-based)
- **Mythril**: Deep analysis, can find complex bugs
- **Echidna**: Randomized testing, property checking

Slither: Usage Example

- **Installation**:

  ```
  pip3 install slither-analyzer
  ```

- **Run Slither**:

  ```
  slither contracts/MyToken.sol
  ```

- **Sample Output**:

  ```
  Reentrancy in MyToken.withdraw (contracts/MyToken.sol#15-19):
      External call: msg.sender.call{value: amount}("")
      State variable written after the call: balances[msg.sender] -= amount
  Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy
  ```

- **Integration**: CI/CD pipelines (fail build on high-severity findings)

Mythril: Symbolic Execution

- **Approach**: Explore all execution paths symbolically
- **Usage**:

```
myth analyze contracts/MyToken.sol --solv 0.8.0
```

- **Capabilities**:
  - Unchecked return values
  - Integer overflows (pre-0.8.0)
  - Unprotected `selfdestruct`
  - State access after external call (reentrancy)
- **Limitation**: Can be slow for large contracts (path explosion)
- **Best Practice**: Use Slither (fast) + Mythril (thorough) in combination

Formal Verification

- **Definition**: Mathematical proof that code satisfies specifications
- **Tools**:
  - **Certora Prover**: Write formal specifications in CVL (Certora Verification Language)
  - **K Framework**: Executable formal semantics of EVM
  - **SMT Solvers**: Z3, CVC4 for constraint solving
- **Example Specification**:

```
rule balanceNeverIncreases(address user) {
    uint balanceBefore = balances[user];
    withdraw(amount);
    assert balances[user] <= balanceBefore;
}
```

- **Adoption**: High-value protocols (Aave, Compound, MakerDAO)
- **Challenge**: Requires formal methods expertise

Audit Process

1. **Internal Review**: Developer self-audit, peer review
2. **Automated Tools**: Slither, Mythril, Echidna
3. **Manual Audit**: Security firm review (2-4 weeks, $50k-$500k)
   - Leading firms: Trail of Bits, ConsenSys Diligence, OpenZeppelin, Quantstamp
4. **Economic Audits**: Mechanism design, incentive analysis
5. **Formal Verification**: High-value or critical contracts
6. **Bug Bounty**: Community testing (Immunefi, HackerOne)
7. **Monitoring**: Real-time anomaly detection (Forta, OpenZeppelin Defender)
8. **Post-Deployment**: Incident response plan, insurance (Nexus Mutual)

Bug Bounty Programs

- **Purpose**: Incentivize white-hat hackers to find vulnerabilities
- **Platforms**: Immunefi, HackerOne, Code4rena
- **Payouts**: $1k (low severity) to $10M+ (critical)
- **Record Payout**: Wormhole $10M bounty (2022)
- **Notable Programs**:
  - Ethereum Foundation: Up to $250k
  - MakerDAO: Up to $10M
  - Compound: Up to $500k
- **Best Practice**: Continuous bounty (not just pre-launch)
- **ROI**: $1 spent on bounties prevents $100+ in losses

Upgradeable Contracts: Proxy Patterns

- **Problem**: Immutable contracts cannot be patched
- **Solution**: Proxy pattern (separate storage and logic)
- **Transparent Proxy**:
  - Proxy contract holds storage, delegates to implementation
  - Admin can upgrade implementation address
  - Users call proxy, which delegates to logic contract
- **UUPS (Universal Upgradeable Proxy Standard)**:
  - Upgrade logic in implementation (not proxy)
  - Smaller proxy contract, lower deployment cost
- **Risk**: Admin key compromise $\rightarrow$ malicious upgrade
- **Mitigation**: Multi-sig admin, timelock delays, immutable after maturity

Security Best Practices Checklist

1. Use latest Solidity version (0.8.0+ for overflow protection)
2. Follow Checks-Effects-Interactions pattern
3. Apply reentrancy guards (OpenZeppelin)
4. Explicit visibility modifiers for all functions
5. Use SafeMath (if Solidity ¡0.8.0)
6. Decentralized oracles (Chainlink) or TWAP
7. Pull over push for payments
8. Avoid loops over unbounded arrays
9. EIP-712 for signature verification
10. Run Slither + Mythril in CI/CD
11. Professional audit before mainnet
12. Bug bounty program
13. Monitoring and incident response plan
14. Upgradeable contracts with timelock governance

Summary

- **Smart contract security is critical**: \$3B+ lost to exploits
- **Top vulnerabilities**: Reentrancy, overflow, access control, oracle manipulation, MEV
- **The DAO hack**: Reentrancy drained \$60M, led to Ethereum hard fork
- **Defense in depth**: Secure coding patterns + automated tools + audits + formal verification
- **Tools**: Slither (fast static analysis), Mythril (symbolic execution), Echidna (fuzzing)
- **Audit process**: Internal review → Tools → Professional audit → Bug bounty
- **Upgradeability**: Proxy patterns allow bug fixes, but introduce admin key risk
- **Continuous vigilance**: Monitoring, incident response, insurance