ETH Devs

SOLIDITY SECURITY CHECKLIST

Adversarial Programming Mindset: "Assume the Worst"

- Expect all transactions, tx data and public balances to be watched by bad actors
- Expect your contract to be deliberately attacked
- Expect users to 'misuse and abuse' your dApp: assume they will pass parameters that functions don't expect, make calls that aren't needed, disappear when their input is required -- etc.

Sending Ether in Functions

Does it use `.call.value(amount)("")` instead of `send` and `transfer` ?

Does it follow the check-effects-interactions pattern?
(check conditions, update storage variables, send transactions)

Do ether-sending functions lock via mutex until execution is complete?

Don't assume execution reverts if sending ether fails -- check success/failure from the value of the returned boolean.

Use the withdrawal pattern: 'pull', not 'push'. Make users call withdrawal functions to receive funds they're entitled to.

Mathematics

Use SafeMath for uint operations.

Do you need to represent decimal numbers? → Use DeciMath for decimal precision.

Logic that Depends on the Contract Balance

Don't rely on `this.balance` in checks. Track deposited ether in a state variable, and use that in your logic.

Contract Code to be Used in Other Contracts

Make your contract a library if you intend its code to be executed in other contracts via `delegatecall`.

Referencing External Contracts

If feasible, deploy a new instance of the contract you reference, and use that.

Hardcode the deployed referenced contract's address.

Carefully examine external contract calls. Don't assume external deployed contracts 'do what they say they do'. Use EtherScan to verify that a deployed contract's compiled bytecode matches its claimed source code.

Functions / Variables

Does your contract specify visibility for all functions and variables?

Randomness

If random numbers are needed, use RanDAO or a reputable randomness oracle -- e.g. an Oraclize call to Wolfram Alpha. Do not use block variables as a source of randomness -- they can be gamed.

Short Addresses/Parameters

Does your dApp UI properly validate parameter lengths / types before sending the transaction to an Ethereum node?

Transaction Frontrunning -- Exposed Tx Data on the Network

Transaction data is public. Could a bad actor watching pending transactions take advantage of information sent to or from your contract?

Could they jump 'in front' of tx A by sending their own tx B with a higher gas price, that takes advantage of the data in tx A?

To hide information in transactions -- votes in elections, or moves in games --use the* '*commit-reveal' pattern.

1. Users each send tx with `{data: Hash(choice)}`
2. Election or game round ends when all participants have submitted their choice, and all txs confirmed
3. Users send tx with `{data:choice}`

To hide the amount of ether in an auction bid*, *use the 'overpay-reveal-refund' pattern:

1. User sends tx with `{value: amount, data: Hash(willingtoPay)}`.
2. User sends tx with `{data: willingToPay}` after bidding period ends
3. Contract refunds `amount`, or `(amount - willingToPay)` if User's bid was the winning bid.

To stop a contract owner (e.g. an ecommerce merchant) frontrunning purchases, use a mutex:

1. Contract has a `txCount` state variable
2. When merchant calls `setPrice`, it increments `txCount` by one.
3. The user's front-end checks the `txCount` and sends a transaction with `{data: _txCount}` to the contract's 'buy' function. The 'buy' function contains `require( _txCount == txCount )`.

The purchase reverts if merchant tries to front-run a 'buy' transaction by quickly changing the price after the 'buy' tx is sent but before it is mined/confirmed.

User Disappears

If a user disappears, can the contract execution still progress?

Does your contract have contingencies for users 'going dark', such as:

- An owner can reset the contract or refund the user
- Contract execution progresses after a specific time period

Size of Dynamic Arrays

Does you contract have dynamic arrays that can be inflated?

Is it looping over an unlimited size array?

Limit arrays to fixed size, or replace loops with a single-call pattern (e.g. withdrawal pattern).

Time-Dependent Logic

Is logic dependent on `block.timestamp`?

Could miners game `block.timestamp` for an advantage in this contract?

Use `( block.number * average block size)` as a safer proxy metric for time-based ordering.

Storage pointers

Are all storage pointers initialized?

Tx.origin

Does the contract use tx.origin in checks? Use `msg.sender` instead.

Exception: `require( tx.origin == msg.sender)` is a useful check to ensure that the `msg.sender` is an externally owned account.

GENERAL CONTRACT DESIGN

Is it too complex, or does it have non-crucial features? The simpler the code & the less it does ... the better.

Does it re-use proven, tested libraries (OpenZeppelin, DeciMath) for basic functionality?

Is it easy to read and audit?

Is it tested thoroughly before launching?

RISK MITIGATION & DEFENSIVE PROGRAMMING

Some or all of these may be appropriate for your contract.

**Healthchecks -- **Does it have automatic or manual healthchecks that are callable by owner, or triggered automatically after X time, or X transactions?

Use healthchecks to ensure:

- Balances / counts match expected values
- Important flags or switches are set to expected values
- Values are within sensible ranges

**Circuit breakers -- **Contract pauses if there are discrepancies or failed healthchecks. Circuit breakers can be automatic, and/or manual switches callable by the owner.

**Delayed Actions -- **If user takes an action that makes a significant change to balance or state variable, we can have a built-in time delay between the call and the execution.

**Rate Limit Withdrawals -- **limit withdrawal magnitude or frequency -- e.g. $x per day, or N withdrawals per day

Require Approval for Big Changes e.g. owner/admin approval needed for withdrawal > $X

Use this checklist in conjunction with unit testing, human auditing and bug bounties where appropriate.

Remember: any smart contract can be hacked, and if it stores or transmits value, sooner or later, someone will try. Hackers are creative, and there's a first time for everything -- every "known" attack was once a shocking surprise.

Design your contracts to mitigate known attacks, and limit potential losses in the case that an attacker does find an exploit.