What is a reentrancy attack?

A reentrancy attack exploits a contract that makes an external call before updating its own state. The called contract (the attacker's) calls back into the original function before the first call finishes, while the contract still thinks nothing has changed, repeating an action (like a withdrawal) many times to drain funds. It is the bug behind The DAO hack and many since. The fix is the checks-effects-interactions pattern (update state before external calls) and a reentrancy guard.

HOW IT WORKS

01 How it works and example

Classic vulnerable withdraw, the external call happens before the balance is zeroed:

```
function withdraw() public {
  uint bal = balances[msg.sender];
  (bool ok,) = msg.sender.call{value: bal}("");  //
calls attacker first
  balances[msg.sender] = 0;  // state updated too late
}
```

The attacker's `receive()`/`fallback()` calls `withdraw()` again before `balances` is set to 0, so the check still passes and they withdraw repeatedly, draining the contract. Cross-function and read-only reentrancy are variants. Shown for defensive context.

> CHECKS-EFFECTS-INTERACTIONS
>
> *The fix is order: do all checks, then update state (effects), then make external interactions last. Update* `balances[msg.sender] = 0` *before the* `call`*, and the re-entry finds a zero balance.*

HOW TO DEFEND

- Follow checks-effects-interactions: update state before any external call.
- Use a reentrancy guard (a `nonReentrant` modifier) on functions that make external calls.
- Prefer pull-over-push for payments so withdrawals do not call untrusted code mid-state.
- Beware read-only reentrancy: a view function reading mid-update state can mislead other protocols; guard those interactions too.
- Audit and test every external-call path for re-entry, including cross-function cases.

SOURCES

[1] SWC Registry: Smart Contract Weakness Classification

[2] Ethereum.org: Smart contract security

[3] Solidity docs: Security considerations