What is integer overflow?

Integer overflow and underflow happen when arithmetic produces a result outside the range a fixed-size integer can hold, so it wraps around: subtracting 1 from 0 in a `uint` becomes the maximum value (underflow), and adding past the maximum returns to 0 (overflow). In smart contracts this can turn a small operation into a huge, attacker-controlled balance. Solidity 0.8.0+ reverts on overflow by default, but older code, `unchecked` blocks, and unsafe casts remain vulnerable. The fix is a modern compiler or SafeMath.

HOW IT WORKS

01 How it works and example

A pre-0.8 transfer that underflows on balance check:

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

Since a `uint` is never negative, the `require` is meaningless, and subtracting more than the balance underflows to a massive number, giving the attacker an enormous balance to spend. Documented for defensive context.

> **0.8 REVERTS, BUT NOT EVERYWHERE**
>
> *Solidity 0.8.0+ checks arithmetic and reverts on overflow/underflow by default. But* `unchecked { }` *blocks, older contracts, and unsafe* `uintN` *casts still wrap, so the bug is far from gone.*

HOW TO DEFEND

- Use Solidity 0.8.0 or later, where checked arithmetic reverts on overflow/underflow by default.
- On older code, use a SafeMath library for all arithmetic.
- Audit every `unchecked { }` block, those opt out of the protection.
- Validate casts between integer sizes (- `uint256` to `uint128`) that can silently truncate.
- Test boundary values (0, max) and fuzz arithmetic-heavy logic.

SOURCES

[1] SWC Registry: Smart Contract Weakness Classification

[2] Solidity docs: Security considerations

[3] MITRE CWE-190: Integer Overflow or Wraparound