

# Clase 09

- 1 · Solución del quiz de la Clase 08
- 2 · Taller de análisis (Cardano) + cierre del curso

# Quiz

## Amenazas EUTXO

**10 preguntas · una sola opción correcta**

Cada una apunta a una confusión concreta: si la respuesta salió por descarte, conviene volver a la slide que la origina.

## 1 · Double Satisfaction - Dos UTXOs, un solo pago

Una transacción gasta con Claim dos UTXOs lockeados por el validator escrow\_vulnerable, los dos con Bob como beneficiario por 100 ADA, pero tiene un solo output de 100 ADA a Bob. El validator verifica que exista un output que pague al menos 100 a Bob. ¿Por qué aprueba los dos?

- A. El ledger suma los datums de los dos UTXOs y el validator corre una sola vez sobre el total.
- B. El validator corre sólo para el primer input del script; el segundo se gasta sin validar, porque ese código ya se ejecutó una vez en la transacción.
- C. Corre una vez por input. Las dos corridas ven la misma transacción, y cada una encuentra "su" output de 100.
- D. No aprueba: el ledger rechaza dos inputs del mismo script en una transacción.

## 1 · Respuesta: C

Cada input de script dispara su propia ejecución del validator, con la misma transacción como contexto. Ninguna corrida sabe qué contó la otra: las dos buscan un output de 100 a Bob, encuentran el mismo, y devuelven True. Se liberan 200 contra un pago de 100.

"Corre una sola vez" es la intuición de Ethereum, donde una llamada es una ejecución. Y el ledger no prohíbe gastar varios UTXOs del mismo script juntos: es lo normal, y es justamente lo que el validator tiene que tener en cuenta.

## 2 · Validar outputs - El primer arreglo

El validator escrow de la Clase 5 aprueba un Claim con la firma del beneficiario antes del deadline, sin mirar los outputs. Para arreglarlo, se agrega el monto (amount) al datum y se exige que exista un output que pague ese monto al beneficiario. ¿Alcanza con eso?

- A. No: con dos inputs del script, un mismo output satisface las dos corridas del validator.
- B. Sí: con eso el validator ya controla a dónde van los fondos.
- C. Sí, siempre que el monto se compare con  $\geq$  y no con  $=$ .
- D. No, no se puede agregar ese chequeo porque el validator no puede leer los outputs de la transacción, sólo el UTxO que está gastando.

## 2 · Respuesta: A

Es exactamente escrow\_vulnerable.ak: valida el output y sigue siendo double-satisfiable. Verificar que "existe" un pago no ata ese pago a este input. Hace falta atarlo: un solo input del script, sumar lo pagado, o etiquetar cada output.

Comparar con  $\geq$  y no con  $==$  es un buen ítem del checklist (por el min-ADA), pero no tiene que ver con esto. Y el validator sí puede leer los outputs: recibe la transacción entera como contexto, con todos sus inputs y outputs. Por eso puede verificar el pago, y por eso mismo tiene que tener en cuenta los otros inputs.

### 3 · Mitigaciones - Lo que cuesta el fix

Para cerrar la double satisfaction, el Claim del escrow agrega `single_script_input`: busca la dirección del UTXO que se está validando, cuenta los inputs de la transacción con esa misma dirección y exige que sea exactamente uno. ¿Qué le cuesta este fix al contrato?

- A. Que el validator ya no puede verificar firmas.
- B. Que la transacción supera el presupuesto de ExUnits: recorrer todos los inputs es demasiado caro para un validator.
- C. Nada: es una verificación más, sin efectos sobre el uso.
- D. Prohíbe el batching legítimo: reclamar varios UTXOs del escrow en una transacción.

### 3 · Respuesta: D

Si el producto necesita gastar varios UTXOs juntos (por eficiencia, o porque un usuario tiene varios), esta regla lo impide. Para ese caso la mitigación es otra: sumar lo pagado al beneficiario y exigir que cubra el total de todos los inputs del script.

"Nada" es la respuesta de la que hay que desconfiar siempre: no existe el fix gratis. Las ExUnits sí suben, pero poco: filtrar una lista de inputs no se acerca al tope.

## 4 · Datum no autorizado - Un UTXO plantado

Un contrato guarda su estado en el datum de un UTXO que está en la dirección de su validator (por ejemplo, quién es el owner). Cualquiera puede crear otro UTXO en esa misma dirección con el datum que quiera, por ejemplo poniéndose a sí mismo como owner. ¿Cómo distingue el validator el UTXO legítimo del plantado?

- A. El validator corre también al depositar, y rechaza el UTXO si el datum no tiene la forma que espera.
- B. Exige en el UTXO un token de validez, que sólo acuña una policy que verificó el datum al crearlo.
- C. No hace falta: en la dirección de un script, el ledger sólo acepta datums del tipo que declara su blueprint.
- D. Verifica que el UTXO esté en la dirección del script.

## 4 · Respuesta: B

Depositar no ejecuta el validator (Clase 4), así que nada impide crear un UTXO con un datum inventado. La forma estándar de autenticarlo es un token: una minting policy lo acuña sólo si el UTXO nace con un datum correcto, y el validator exige ese token. Sin el token, el UTXO no es suyo aunque esté en su dirección.

La dirección no prueba nada: es justamente lo que cualquiera puede elegir. Y el blueprint es para el off-chain; el ledger no lo conoce.

## 5 · Tiempo - Un rango sin cota superior

Un UTXO del validator escrow de la Clase 5 tiene a Bob como beneficiario y deadline 10.000. El Claim exige la firma de Bob y que `interval.before(deadline) |> interval.includes(ValidityRange)` dé True. Bob firma una transacción de Claim con `ValidityRange`  $[1.000, +\infty)$ . ¿Qué pasa?

- A. Aprueba si el bloque en el que entra cae antes del deadline: el validator ve la hora del bloque.
- B. Aprueba: el rango empieza antes del deadline.
- C. Rechaza: un rango sin cota superior no está contenido en  $(-\infty, \text{deadline}]$ .
- D. Lo decide el nodo: completa el rango con la hora actual antes de evaluar.

## 5 · Respuesta: C

El validator no ve ninguna hora: ve el intervalo que declaró quien armó la transacción. includes exige que TODO el rango esté antes del deadline, y un rango que no termina nunca no lo está. El escrow lo rechaza, y está bien que lo haga.

"Empieza antes" es el error típico de mirar un solo borde: aceptaría una transacción que puede entrar en cualquier momento futuro. Y el nodo no completa nada: si el rango no se declara, es  $(-\infty, +\infty)$ .

## 6 · Minting policies - Una policy que sólo pide firma

La minting policy `token_fragil` sólo verifica que el emisor esté en `tx.extra_signatories`. Con ella, el emisor ya acuñó 1.000 VALE. Ahora firma una transacción que acuña, con la misma policy, 1.000.000 de un token con un nombre que nunca se usó. ¿Qué pasa?

- A. Rechaza: el ledger exige que el nombre del token esté registrado en la policy antes de la primera acuñación.
- B. Aprueba: la policy no mira `tx.mint`, así que pasa cualquier cantidad con cualquier nombre.
- C. Aprueba, pero el ledger limita la cantidad a la oferta inicial.
- D. Rechaza: para volver a acuñar primero hay que quemar.

## 6 · Respuesta: B

La policy es la única regla de emisión, y ésta sólo pide una firma. No limita cuánto se acuña, ni con qué nombre, ni si es acuñar o quemar ( $\text{mint} < 0$ ). Con la firma, todo vale.

No hay registro de nombres ni oferta inicial en el ledger: lo que no verifica la policy, no lo verifica nadie. Es el mismo principio que los outputs del escrow, aplicado a los tokens.

## 7 · Reentrancy - El Vault de la Clase 7, en Cardano

En la Clase 7, ReentrancyAttacker vació VaultVulnerable: withdraw() le mandaba el ETH antes de poner su balance en 0, y el receive() del atacante volvía a llamar a withdraw(). ¿Por qué ese ataque casi no tiene equivalente en un validator de Cardano?

- A. Porque los enteros de Aiken son de precisión arbitraria.
- B. Porque Aiken trae un reentrancy guard incorporado.
- C. Porque el ledger procesa una sola transacción por bloque, así que nadie puede intercalar una llamada en el medio.
- D. Porque el validator no muta estado ni llama a otros contratos: evalúa una vez una transacción ya armada.

## 7 · Respuesta: D

La reentrancy necesita un "durante": código ajeno que corre en medio de una función que todavía no actualizó su estado. Un validator es una función pura sobre una transacción terminada: no hay estado que dejar a medias ni llamada externa por la que volver a entrar.

La precisión arbitraria es la razón por la que desaparece el overflow clásico: otro bug, otra causa. No hay ningún guard: no hace falta. Y el ledger mete muchas transacciones por bloque.

## 8 · EUTXO vs. cuentas - ¿Más seguro?

Con los dos catálogos lado a lado, el de Ethereum de la Clase 6 y el de Cardano de hoy: ¿es el modelo EUTXO más seguro que el modelo de cuentas?

- A. Elimina clases enteras de bugs, pero lo que el validator no verifica no está garantizado.
- B. Sí: sin reentrancy ni overflow, un validator que compila y pasa sus tests es seguro por defecto.
- C. No: es igual de inseguro, sólo cambian los nombres de los bugs.
- D. Sí, porque en Cardano la verificación formal de los validators es obligatoria antes de poder desplegarlos.

## 8 · Respuesta: A

Las dos mitades van juntas. EUTXO elimina reentrancy, overflow y llamadas a código ajeno, y por eso es más fácil de razonar y de verificar. Pero la transacción la arma el adversario, y cada cosa que el validator se olvida de verificar es un agujero. Un contrato no es seguro por estar en EUTXO: es seguro si verifica todo lo que debe.

"Igual de inseguro" niega una ventaja real. "Seguro por defecto" ignora todo el catálogo de hoy. Y no hay verificación formal obligatoria: para Aiken, ni siquiera hay todavía una herramienta formal lista.

## 9 · Herramientas - ¿Un slither para Cardano?

En la Clase 6, Slither le pasaba a un contrato de Solidity más de 100 detectores de patrones vulnerables: análisis estático automático, barato, que se corre siempre. Escribiste un validator en Aiken y querés ese mismo nivel. ¿Con qué contás en Cardano?

- A.** aiken check: además de compilar y correr los tests, trae detectores de vulnerabilidades conocidas.
- B.** Blaster: verifica formalmente cualquier validator escrito en Aiken, con un solo comando.
- C.** Nada equivalente: no hay analizador estático estándar; lo cubren property tests y checklist.
- D.** Un analizador estático oficial de IOG, anunciado en julio de 2025 y ya disponible.

## 9 · Respuesta: C

Es la asimetría de herramientas de la clase: menos comunidad, menos valor en juego, menos herramientas. El nivel automático y barato que en Ethereum es slither . acá no existe; lo cubren aiken check con property tests y un checklist recorrido a mano.

aiken check corre los tests que vos escribís: no trae detectores. Blaster hoy se usa desde Lean4; la integración con Aiken está en marcha. E IOG anunció un analizador estático en 2025, pero no publicó ninguno.

## 10 · Property tests - PASS [after 100 tests]

'aiken check' muestra PASS [after 100 tests] prop\_claim\_rechaza\_sin\_firma. ¿Qué afirma ese resultado?

- A. Que la propiedad es verdadera para todo valor posible del argumento, no sólo para los probados.
- B. Que los 100 valores que generó el fuzzer la cumplieron. Es evidencia, no prueba.
- C. Que pasaron 100 tests unitarios distintos.
- D. Que la propiedad tardó 100 ms en verificarse.

## 10 · Respuesta: B

Un property test corre el cuerpo con 100 valores generados (el default; --max-success lo cambia) y exige True en todos. Si uno falla, muestra el contraejemplo achicado. Que pasen los 100 aumenta la confianza, pero no cubre todos los casos: eso sería verificación formal.

Es la misma diferencia que en la Clase 6 entre el fuzzer de Foundry y halmos: muestrear no es probar. Y no son 100 tests: es uno, corrido 100 veces, que cuenta como 100 checks en el Summary.

# Repaso

| #  |   | Tema                                              |
|----|---|---------------------------------------------------|
| 1  | C | Double Satisfaction - Dos UTXOs, un solo pago     |
| 2  | A | Validar outputs - El primer arreglo               |
| 3  | D | Mitigaciones - Lo que cuesta el fix               |
| 4  | B | Datum no autorizado - Un UTXO plantado            |
| 5  | C | Tiempo - Un rango sin cota superior               |
| 6  | B | Minting policies - Una policy que sólo pide firma |
| 7  | D | Reentrancy - El Vault de la Clase 7, en Cardano   |
| 8  | A | EUTXO vs. cuentas - ¿Más seguro?                  |
| 9  | C | Herramientas - ¿Un slither para Cardano?          |
| 10 | B | Property tests - PASS [after 100 tests]           |

# Clase 09

**Taller de análisis (Cardano)**

**Auditar, demostrar, arreglar + cierre del curso**

## Objetivos de la clase

- Aplicar el checklist de la Clase 8 a un validator que no vieron antes y encontrar sus bugs
- Escribir el **Proof of Concept (PoC) EUTxO**: la transacción que el validator aprueba y no debería
- Escribir una **propiedad de seguridad** (safety) con `fuzz`, leer el contraejemplo y reproducirlo
- Aplicar el fix y **re-verificarlo** con los mismos tests, dados vuelta
- Redactar el hallazgo
- Articular una **comparación final** entre los dos modelos y sus vulnerabilidades existentes

# Los archivos de la clase

Repaso inicial (los vemos juntos, no se modifican):

| Archivo                         | Rol                                                              |
|---------------------------------|------------------------------------------------------------------|
| validators/escrow_vulnerable.ak | double satisfaction, con un test que demuestra la vulnerabilidad |

Taller (lo que ustedes leen, escriben y corren):

| Archivo                          | Rol                                                                                              |
|----------------------------------|--------------------------------------------------------------------------------------------------|
| cardano/Taller-Clase9.md         | las consignas y los comandos                                                                     |
| validators/vesting_vulnerable.ak | el validator a auditar: el vesting versión 2. Los poc_* y los prop_* <b>los escriben ustedes</b> |
| validators/mi_vesting_fixed.ak   | <b>el fix: lo crean ustedes</b> a partir de una copia del vulnerable                             |

SEGMENTO 1

# Repaso: el escrow de la clase 08

## El bug del escrow, demostrado con un test

`escrow_vulnerable.ak` es el escrow de las Clases 04 y 05 con un chequeo más: exige que **exista un output** que le pague al beneficiario el monto. Si alguien crea una tx que gasta dos UTXOs del escrow, **un solo pago satisface a los dos** (Clase 08).

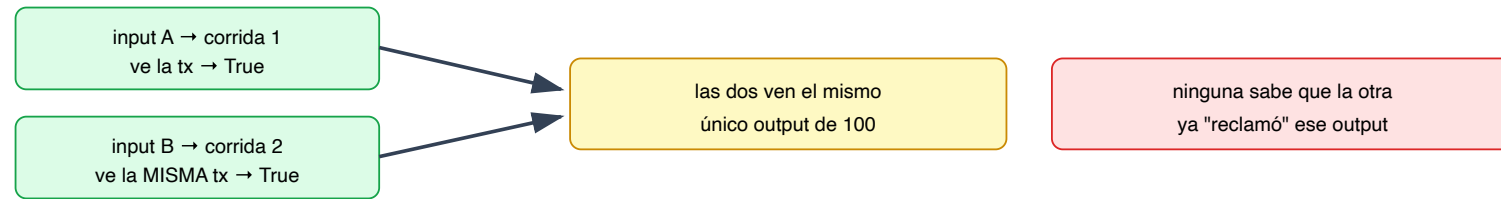
¿Cómo se demuestra? **Construyendo una tx maliciosa** y mostrando que el validator la aprueba.

```
// escrow_vulnerable.ak: este test PASA, y ese pase ES el bug
test vuln_double_satisfaction_se_aprueba_indebidamente() {
  let d = make_datum() // beneficiary = Bob, amount = 100
  let tx = Transaction { ..placeholder,
    inputs: [make_script_input(0), make_script_input(1)], // A y B, 100 c/u
    extra_signatories: [beneficiary_hash],
    validity_range: interval.before(deadline_ms),
    outputs: [output_to_beneficiary(locked)] } // UN solo output de 100

  let aprueba_input_a = can_claim_vulnerable(d, tx) // corrida para A
  let aprueba_input_b = can_claim_vulnerable(d, tx) // corrida para B
  aprueba_input_a && aprueba_input_b // los dos True → double satisfaction
}
```

## ¿Por qué se llama al validator DOS veces?

No es redundancia: **el validator corre una vez por cada input de script** (Clase 4).



El bug vive en que **las dos comparten un efecto sin saberlo**.  
Las dos llamadas del test *son* esas dos corridas independientes.

## El fix que ya vieron, y su re-verificación

El validator sólo aprueba si la tx gasta **un único input lockeado por sí mismo**: así no podemos consumir 2 y realizar un sólo pago al beneficiario.

```
fn single_script_input(own_ref: OutputReference, tx: Transaction) -> Bool {  
    let own_input = resolve_input(tx.inputs, own_ref)  
    let script_addr = own_input.address  
    let script_inputs =  
        list.filter(tx.inputs, fn(i: Input) { i.output.address == script_addr })  
    list.length(script_inputs) == 1  
}
```

```
// el mismo escenario, ahora rechazado: dos inputs de script, un solo output  
test fixed_double_satisfaction_rechazada() { ... !can_claim_fixed(d, script_input_ref(0), tx) }
```

## Y en una blockchain de verdad

El test muestra que el validator **aprueba**. Falta ver que el nodo **acepta**: con fees, colateral y saldos que se mueven. Podemos correrlo en la devnet, con una transacción real:

```
./scripts/devnet.sh up          # terminal 1
cd cardano/offchain && npm run demo-double  # terminal 2
```

```
1 · LOCK: Alice bloquea dos UTXOs de 5 ADA, los dos para Bob
2 · La transacción: dos inputs del escrow, UN solo pago de 5 ADA a Bob
  inputs          2 UTXOs del escrow (10 ADA liberados)
  output          5 ADA → Bob          ← uno solo
  La cadena ACEPTÓ la transacción.
3 · El resultado
  liberado        10 ADA salieron del escrow (0 UTXOs quedan sin gastar)
  a Bob           por el output del escrow: 5 ADA
```

La misma transacción contra el validator con `single_script_input`:

```
$ npm run demo-double -- escrow_fixed
RECHAZADA. El evaluador ejecutó los validators antes de enviarla:
  input 0        False
  input 1        False
```

SEGMENTO 2

**La consigna:  
el vesting, versión 2**

## Lo que el validator dice que hace

`validators/vesting_vulnerable.ak` es el vesting de la Clase 5, reescrito:

- **Unlock** : el beneficiario retira `amount` recién **después** del deadline. La tx tiene que **pagarle** `amount` y gastar **un solo UTXO** de este script.
- **Cancel** : el owner recupera los fondos **en cualquier momento**.

Lo que se le agregó respecto de la Clase 5: el owner y el `Cancel` , la validación del output, y la protección contra double satisfaction.

Hace **dos cosas mal**. La especificación de arriba es contra lo que se audita: cada garantía que promete y no da es un hallazgo.

## El código que importa

```
pub type Redeemer {
  Unlock
  Cancel { signer: ByteArray } // quien cancela declara su hash de clave pública
}

// Después del deadline: si la tx no es de antes, es de después.
fn after_deadline(d: Datum, tx: Transaction) -> Bool {
  !(interval.before(d.deadline) |> interval.includes(tx.validity_range))
}

fn can_unlock(d: Datum, own_ref: OutputReference, tx: Transaction) -> Bool {
  signed_by(tx, d.beneficiary) && after_deadline(d, tx)
  && single_script_input(own_ref, tx)
  && some_output_pays(tx.outputs, d.beneficiary, d.amount)
}

// Cancelar: el que cancela dice quién es, y tiene que haber firmado.
fn can_cancel(signer: ByteArray, tx: Transaction) -> Bool {
  signed_by(tx, signer)
}
```

## Lo que anda: los tests que vienen hechos

```
vesting_vulnerable
PASS [mem: 152.31 K, cpu: 53.44 M] v2_unlock_legitimo_pasa
PASS [mem: 46.74 K, cpu: 13.69 M] v2_unlock_sin_firma_falla
PASS [mem: 136.39 K, cpu: 48.65 M] v2_unlock_sin_pago_al_beneficiario_falla
PASS [mem: 156.91 K, cpu: 55.53 M] v2_unlock_con_dos_inputs_de_script_falla
PASS [mem: 28.41 K, cpu: 8.93 M] v2_owner_recupera_los_fondos_pasa
```

Cinco tests en verde, y los dos bugs adentro. **Tests felices  $\neq$  seguridad**: es el Vault de la Clase 7 otra vez.

Los helpers también vienen hechos: `make_datum()`, `tx_unlock(firmas, rango)`, `tx_cancel(firmas)`, `make_script_input(idx)`, `script_input_ref(idx)`. Los PoCs se escriben con eso.

# Consigna 1: checklist y triage

Aplicar el checklist de la consigna ( Taller-Clase9.md ) a vesting\_vulnerable.ak , ítem por ítem,  /  con una frase.

| Ítem                                                                     |  /  | Evidencia |
|--------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| ¿verifica que los fondos van a la dirección correcta?                    |                                                                                                                                                                           |           |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?           |                                                                                                                                                                           |           |
| ¿ expect Some(d) maneja el caso None ?                                   |                                                                                                                                                                           |           |
| ¿mira <b>los dos bordes</b> del validity_range (y rechaza uno sin cota)? |                                                                                                                                                                           |           |
| ¿ extra_signatories contra el hash correcto?                             |                                                                                                                                                                           |           |

Dos  son bugs. Para cada , el test v2\_\* que lo demuestra. Y una pregunta guía: **¿qué campo del datum no se lee en ningún lado?**

SOLUCIÓN · CONSIGNA 1

El checklist, resuelto

| Ítem                                                                                  |   | Evidencia                                                                                                               |
|---------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------|
| ¿verifica que los fondos van a la dirección correcta?                                 | ✓ | <code>some_output_pays</code> exige la clave del beneficiario;<br><code>v2_unlock_sin_pago_al_beneficiario_falla</code> |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?                        |   |                                                                                                                         |
| ¿ <code>expect Some(d)</code> maneja el caso <code>None</code> ?                      |   |                                                                                                                         |
| ¿mira <b>los dos bordes</b> del <code>validity_range</code> (y rechaza uno sin cota)? |   |                                                                                                                         |
| ¿ <code>extra_signatories</code> contra el hash correcto?                             |   |                                                                                                                         |

SOLUCIÓN · CONSIGNA 1

El checklist, resuelto

| Ítem                                                                                  |   | Evidencia                                                                                                               |
|---------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------|
| ¿verifica que los fondos van a la dirección correcta?                                 | ✓ | <code>some_output_pays</code> exige la clave del beneficiario;<br><code>v2_unlock_sin_pago_al_beneficiario_falla</code> |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?                        | ✓ | <code>single_script_input</code> ; <code>v2_unlock_con_dos_inputs_de_script_falla</code>                                |
| ¿ <code>expect Some(d)</code> maneja el caso <code>None</code> ?                      |   |                                                                                                                         |
| ¿mira <b>los dos bordes</b> del <code>validity_range</code> (y rechaza uno sin cota)? |   |                                                                                                                         |
| ¿ <code>extra_signatories</code> contra el hash correcto?                             |   |                                                                                                                         |

SOLUCIÓN · CONSIGNA 1

El checklist, resuelto

| Ítem                                                                                  |   | Evidencia                                                                                                               |
|---------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------|
| ¿verifica que los fondos van a la dirección correcta?                                 | ✓ | <code>some_output_pays</code> exige la clave del beneficiario;<br><code>v2_unlock_sin_pago_al_beneficiario_falla</code> |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?                        | ✓ | <code>single_script_input</code> ; <code>v2_unlock_con_dos_inputs_de_script_falla</code>                                |
| ¿ <code>expect Some(d)</code> maneja el caso <code>None</code> ?                      | ✓ | <code>expect Some(d)</code> : falla, el UTXO queda bloqueado                                                            |
| ¿mira <b>los dos bordes</b> del <code>validity_range</code> (y rechaza uno sin cota)? |   |                                                                                                                         |
| ¿ <code>extra_signatories</code> contra el hash correcto?                             |   |                                                                                                                         |

SOLUCIÓN · CONSIGNA 1

El checklist, resuelto

| Ítem                                                                                  |   | Evidencia                                                                                                               |
|---------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------|
| ¿verifica que los fondos van a la dirección correcta?                                 | ✓ | <code>some_output_pays</code> exige la clave del beneficiario;<br><code>v2_unlock_sin_pago_al_beneficiario_falla</code> |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?                        | ✓ | <code>single_script_input</code> ; <code>v2_unlock_con_dos_inputs_de_script_falla</code>                                |
| ¿ <code>expect Some(d)</code> maneja el caso <code>None</code> ?                      | ✓ | <code>expect Some(d)</code> : falla, el UTXO queda bloqueado                                                            |
| ¿mira <b>los dos bordes</b> del <code>validity_range</code> (y rechaza uno sin cota)? | ✗ | <code>after_deadline</code> sólo mira que el rango <b>no termine antes</b> : un rango que cruza el deadline pasa        |
| ¿ <code>extra_signatories</code> contra el hash correcto?                             |   |                                                                                                                         |

SOLUCIÓN · CONSIGNA 1

El checklist, resuelto

| Ítem                                                                                  |   | Evidencia                                                                                                               |
|---------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------|
| ¿verifica que los fondos van a la dirección correcta?                                 | ✓ | <code>some_output_pays</code> exige la clave del beneficiario;<br><code>v2_unlock_sin_pago_al_beneficiario_falla</code> |
| ¿rechaza una tx que gasta <b>varios</b> inputs de este script?                        | ✓ | <code>single_script_input</code> ; <code>v2_unlock_con_dos_inputs_de_script_falla</code>                                |
| ¿ <code>expect Some(d)</code> maneja el caso <code>None</code> ?                      | ✓ | <code>expect Some(d)</code> : falla, el UTXO queda bloqueado                                                            |
| ¿mira <b>los dos bordes</b> del <code>validity_range</code> (y rechaza uno sin cota)? | ✗ | <code>after_deadline</code> sólo mira que el rango <b>no termine antes</b> : un rango que cruza el deadline pasa        |
| ¿ <code>extra_signatories</code> contra el hash correcto?                             | ✗ | <code>can_cancel</code> compara contra <code>signer</code> , del redeemer. <code>d.owner</code> <b>no se lee nunca</b>  |

El campo que no se lee es `owner` .

SEGMENTO 3

# **Los PoCs: la tx que aprueba**

## Consigna 2: los PoCs

En `vesting_vulnerable.ak`, escribir los dos tests que arrancan con `todo`, según la consigna de su comentario:

- `poc_unlock_antes_del_deadline_se_aprueba`
- `poc_cualquiera_se_lleva_los_fondos`

```
aiken check -m poc_
```

Los dos tienen que **pasar**. Cada uno arma una tx que el validator **debería rechazar** y afirma que la aprueba. En EUTXO el exploit es una transacción, y el test que la aprueba es la prueba.

Probar el primero también con `interval.everything`. ¿Qué significa que pase?

## SOLUCIÓN · CONSIGNA 2

### Bug 1: el tiempo, con un solo borde

```
!(interval.before(d.deadline) |> interval.includes(tx.validity_range))
```

"Si la tx no es de antes, es de después." Falso: un rango puede **cruzar** el deadline.



El nodo garantiza que el bloque cae **dentro** del rango, no en qué parte. Si el rango empieza antes del deadline, la tx puede ejecutarse antes del deadline. La condición correcta es la de la Clase 5: **todo** el rango después ( `interval.after(deadline) |> includes` ).

## SOLUCIÓN · CONSIGNA 2

### PoC 1: retirar antes de tiempo

```
// Este test PASA, y ese pase ES el bug.
test poc_unlock_antes_del_deadline_se_aprueba() {
  can_unlock(
    make_datum(),
    script_input_ref(0),
    tx_unlock([beneficiary_hash], interval.between(5_000, 15_000)),
  )
}
```

```
| PASS [mem: 164.02 K, cpu: 57.48 M] poc_unlock_antes_del_deadline_se_aprueba
| PASS [mem: 143.26 K, cpu: 50.50 M] poc_unlock_con_rango_abierto_se_aprueba
```

Con `interval.everything` también pasa: el off-chain no puso límites y el validator aprueba **en cualquier momento**, también el día 1. El vesting no vence nada.

En vivo: `./scripts/demo-taller-cardano.sh --escena 2` y `--escena 3`, la tx en pantalla y el validator decidiendo. \*

## SOLUCIÓN · CONSIGNA 2

### Bug 2: autorización con datos del adversario

```
Cancel { signer } -> can_cancel(signer, self)    // signer: lo escribe el que arma la tx  
  
fn can_cancel(signer: ByteArray, tx: Transaction) -> Bool {  
    signed_by(tx, signer)  
}
```

El atacante **declara** quién es en el redeemer, y firma como ese alguien. La verificación es correcta, contra el dato equivocado. `d.owner`, el único que no controla, no se consulta.

Es el `setAdmin` de la Clase 7 con otra forma: ahí cualquiera se **nombraba** admin; acá cualquiera se **declara** owner. En EUTXO todo lo que viene en la tx (redeemer, outputs, datums de otros) lo arma el adversario. Lo que vale es lo que está en el datum **propio** y en el contexto.

## SOLUCIÓN · CONSIGNA 2

### PoC 2: cualquiera se lleva los fondos

```
// Este test PASA, y ese pase ES el bug.  
test poc_cualquiera_se_lleva_los_fondos() {  
    can_cancel(atacante_hash, tx_cancel([atacante_hash]))  
}
```

```
| PASS [mem: 28.41 K, cpu: 8.93 M] poc_cualquiera_se_lleva_los_fondos
```

Dos líneas. Igual que en la Clase 7, el bug de autorización es el **más fácil de explotar** y el de **mayor impacto**: no hace falta esperar ningún deadline ni ser beneficiario de nada.

En vivo: `./scripts/demo-taller-cardano.sh --escena 4`.\*

SEGMENTO 4

# **Las propiedades de seguridad**

## Consigna 3: las propiedades

Un PoC es **una** transacción. Una propiedad dice qué debe pasar con **todas**. Escribir el cuerpo de las dos que arrancan con `todo` (el generador del rango viene hecho):

- `prop_unlock_rechaza_todo_rango_que_empieza_antes`
- `prop_solo_el_owner_recupera`

```
aiken check -m prop_
```

Contra este validator tienen que **fallar**. Leer el contraejemplo: ¿qué rango encontró?

¿Cruza el deadline? ¿Por qué dice `after 1 test` y no `after 100 tests` ? Volver a correr con el `--seed=<n>` que imprime: ¿da el mismo?

**SOLUCIÓN · CONSIGNA 3**

## La propiedad del tiempo

```
// Generador: un rango [lo, hi] que empieza antes del deadline.
fn rango_que_empieza_antes() -> Fuzzer<interval.Interval> {
  fuzz.int_between(0, deadline_ms - 1)
  |> fuzz.and_then(fn(lo) {
    fuzz.int_at_least(lo) |> fuzz.map(fn(hi) { interval.between(lo, hi) })
  })
}

// Para todo rango que empieza antes del deadline, Unlock rechaza.
test prop_unlock_rechaza_todo_rango_que_empieza_antes(
  rango via rango_que_empieza_antes(),
) {
  !can_unlock(make_datum(), script_input_ref(0), tx_unlock([beneficiary_hash], rango))
}
```

`and_then` encadena generadores: `hi` depende de `lo`. Es la composición de la Clase 8.

**SOLUCIÓN · CONSIGNA 3****Lo que devuelve: el contraejemplo, achicado**

```

Simplifying counterexample from 7 choices
Simplified counterexample in 68ms after 98 steps
FAIL [after 1 test] prop_unlock_rechaza_todo_rango_que_empieza_antes
× counterexample
  Interval {
    lower_bound: IntervalBound { bound_type: Finite(8192), is_inclusive: True },
    upper_bound: IntervalBound { bound_type: Finite(10240), is_inclusive: True },
  }
_____ with --seed=1385859528 → 10 tests | 8 passed | 2 failed

```

- `[8192, 10240]` **cruza** el deadline (10 000): exactamente el bug.
- `after 1 test` : casi cualquier rango lo rompe. Si fuera 1 en 100, diría `after 37 tests` .
- El shrinking achica las **decisiones del generador** ( `7 choices` ), no los enteros: por eso salen potencias de dos y no `[9999, 10001]` .
- `--seed=1385859528` reproduce la corrida exacta. Sin eso, un fallo aleatorio no se depura.

### SOLUCIÓN · CONSIGNA 3

## La propiedad del owner

```
// Para toda firma que no es la del owner, Cancel rechaza.
test prop_solo_el_owner_recupera(signer via fuzz.bytearray()) {
  signer == owner_hash || !can_cancel(signer, tx_cancel([signer]))
}
```

[illegible]

El `signer == owner_hash` || descarta el único caso legítimo: el generador podría producir justo al owner, y ahí `Cancel` **debe** aprobar. Una propiedad bien escrita dice exactamente para qué casos vale.

PoC: **una** tx. Propiedad: **todas** las que el generador muestrea. Lo formal (Clase 8): **todas**, sin muestrear. Tres niveles de la misma afirmación.

SEGMENTO 5

# **Fix + re-verificación + mini-reporte**

## Consigna 4: el fix

Copiar `vesting_vulnerable.ak` a `mi_vesting_fixed.ak`, renombrar el validator, y arreglar los dos bugs con **el cambio más chico** que los cierre. El `Datum` y el `Redeemer` quedan como están.

Verificarlo dos veces:

1. **Con las propiedades.** Las dos de la consigna 3, copiadas ahí, pasan `[after 100 tests]`.
2. **Con los PoCs.** Los dos de la consigna 2, copiados y **dados vuelta** (`!can_unlock`, `!can_cancel`), pasan. Y los `v2_*` siguen pasando.

Discutir: ¿su fix de `Cancel` sigue sirviendo si mañana le sacan el campo `signer` al redeemer?  
¿Y el del tiempo, si el off-chain manda un rango **sin cota inferior**?

## SOLUCIÓN · CONSIGNA 4

### Fix 1: el tiempo, con los dos bordes

```
// [deadline, +∞) tiene que contener TODO el rango.  
fn after_deadline(d: Datum, tx: Transaction) -> Bool {  
    interval.after(d.deadline) |> interval.includes(tx.validity_range)  
}
```

Es la línea del vesting de la Clase 5. Un rango sin cota inferior,  $(-\infty, 20\ 000]$ , tampoco pasa: no está contenido en  $[10\ 000, +\infty)$ .

```
| PASS [mem: 110.00 K, cpu: 36.06 M] fixed_unlock_antes_del_deadline_rechazado  
| PASS [mem: 87.23 K, cpu: 28.51 M] fixed_unlock_con_rango_abierto_rechazado  
| PASS [mem: 90.60 K, cpu: 29.52 M] fixed_unlock_sin_cota_inferior_rechazado
```

## SOLUCIÓN · CONSIGNA 4

### Fix 2: el owner es el del datum

```
// El owner es el del datum. Lo que diga el redeemer no importa.  
fn can_cancel(d: Datum, tx: Transaction) -> Bool {  
    signed_by(tx, d.owner)  
}  
  
// en el validator:  
Cancel { .. } -> can_cancel(d, self)
```

Noten que **cambia la firma de `can_cancel`** : el primer parámetro pasa de `signer: ByteArray` a `d: Datum` . Los tests copiados que la llaman (el PoC y la propiedad) hay que ajustarlos.

**Los mismos tests, dados vuelta.** Lo que pasaba demostrando el bug ahora pasa demostrando el fix. Y las propiedades que fallaban al primer intento ahora sobreviven cien.

En vivo: `./scripts/demo-taller-cardano.sh --escena 5` , las tres tx malas contra el fix. \*

## Consigna 5: el hallazgo

```
## [HIGH] Unlock antes del deadline en vesting_vulnerable

**Ubicación:** validators/vesting_vulnerable.ak, after_deadline

**Descripción:** la condición de tiempo sólo exige que el validity_range no
termine antes del deadline. Un rango que empiece antes y termina después
(o un rango sin cotas) se aprueba, y la tx puede ejecutarse antes del deadline.

**Impacto:** el beneficiario cobra antes de tiempo; con un rango abierto, en
cualquier momento. El vesting no garantiza nada.

**Prueba de concepto:** poc_unlock_antes_del_deadline_se_aprueba;
prop_unlock_rechaza_todo_rango_que_empieza_antes falla con [8192, 10240].

**Recomendación:** interval.after(d.deadline) |> interval.includes(tx.validity_range).

**Estado:** corregido en mi_vesting_fixed.ak; verificado por
fixed_unlock_antes_del_deadline_rechazado y por la propiedad, [after 100 tests].
```

El segundo hallazgo lo escriben ustedes. **¿Cuál de los dos es más grave, y por qué?**

# Cierre comparado del curso

Los dos mundos, lado a lado

## Cuentas vs EUTXO: el cuadro completo del curso

|                     | Ethereum (cuentas)                                            | Cardano (EUTXO)                                                            |
|---------------------|---------------------------------------------------------------|----------------------------------------------------------------------------|
| Bug estrella        | Reentrancy                                                    | Double satisfaction                                                        |
| Causa raíz          | estado mutable + llamada re-entrante                          | lo que el validator no verifica de una tx que no armó                      |
| Forma del PoC       | contrato que drena (test que roba)                            | tx que aprueba (test que pasa)                                             |
| El bug "fácil"      | <code>setAdmin</code> sin chequeo: cualquiera se nombra admin | <code>Cancel</code> que confía en el redeemer: cualquiera se declara owner |
| Patrón de fix       | CEI + guard, <code>onlyAdmin</code>                           | atar a un input, mirar los dos bordes, confiar sólo en el datum propio     |
| Herramienta central | Slither + fuzzing + halmos                                    | property testing + checklist                                               |
| No aplica del otro  | double satisfaction no existe                                 | reentrancy / overflow casi no aplican                                      |

## El modelo de ejecución decide de qué se tienen que cuidar

Cómo se **escribe** y cómo se **rompe** un contrato es una consecuencia directa del **modelo de ejecución** de la blockchain.

## Dos consecuencias del mismo principio

- En **cuentas**, el peligro es el **estado mutable compartido**:  
el código corre, muta, y puede ser re-entrado.  
→ reentrancy, control de acceso, orden de operaciones
- En **EUTXO**, el peligro es **lo que el validator se olvida de verificar**  
sobre una tx que él no construye: destino, cantidad de inputs, bordes del tiempo,  
qué dato es suyo y cuál lo trajo el adversario.  
→ double satisfaction, validación de outputs, tiempo, autorización

No son dos listas que memorizar. La **superficie de ataque la define el modelo de ejecución**. Quien entiende el modelo, anticipa los bugs, en cualquier blockchain.

## Lecturas

- **Cardano CTF (Invariant0):** `github.com/Invariant-0/cardano-ctf` . El nivel `02_vesting` es este mismo bug, con transacciones reales. Mismo formato que hoy: romper un validator con un test
- **Vacuumlabs: "Cardano Vulnerabilities" #4, *Time handling* y #1, *Double Satisfaction***
- **Cardano Developer Portal: "Smart Contract Security":** el catálogo oficial; *time handling* y *missing UTxO authentication* son dos de sus cuatro "deep dives"
- **Aiken:** `aiken-lang/fuzz` : generadores, `and_then` , shrinking, `--seed`
- **Tweag: retrospectiva de cinco años de auditorías (2026):** qué falla de verdad hoy
- Repaso final: Clases 1, 6 y 8 (la tabla de los dos catálogos)

# Fin del curso

## Cuentas vs EUTXO

**Un mismo principio, dos catálogos de bugs**

*Quien entiende el modelo de ejecución, anticipa cómo se rompe.*