

Clase 8

Intro análisis Cardano

Modelo de amenazas EUTXO y herramientas

Tesis de la clase

El catálogo de bugs de EUTXO es **distinto** al de cuentas porque el modelo de ejecución es distinto.

Reentrancy y overflow casi desaparecen; aparecen double satisfaction, validación de outputs y datums no autorizados. Y **no existe un Slither maduro para Cardano**.

Objetivos de aprendizaje

Al terminar, vas a poder:

- Nombrar y describir las vulnerabilidades EUTXO
- Explicar **por qué reentrancy/overflow casi no aplican** y qué los reemplaza
- Entender el panorama de herramientas de Cardano **con honestidad**
- Pensar **propiedades** para un validator y escribirlas como property tests
- Usar un **checklist de auditoría EUTXO** sobre el escrow de la Clase 5

SEGMENTO 1 · ~45 min

Modelo de amenazas EUTXO

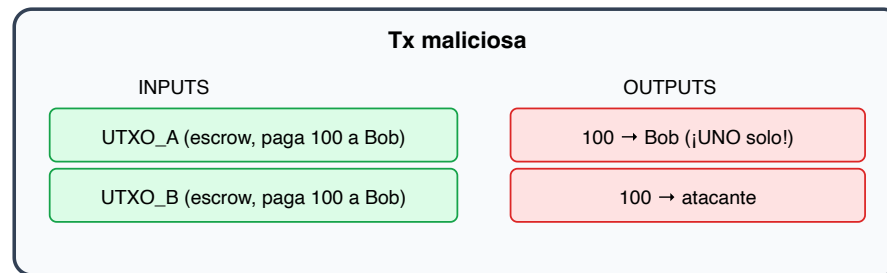
Los bugs vienen de lo que el validator se olvida de mirar

En EUTXO el validator es una **función pura** que mira la tx y dice sí/no.

Los bugs no vienen de estado mutable compartido (no lo hay) sino de **lo que el validator se olvida de verificar** sobre una transacción que él no construye.

1a. Double satisfaction — la estrella

Una sola acción de la tx satisface **dos validators a la vez**.



cada validator ve
"un output de 100
a Bob" y aprueba
roba 100 ADA

Es un bug de **composición**: cada validación es correcta en aislamiento.

En vivo: `./scripts/demo-bugs-cardano.sh --escena 2`, el validator decidiendo sobre esa tx. Y
`aiken check -m vuln_double`: el test que **pasa** demostrando que `escrow_vulnerable` la aprueba.

El caso canónico real: **BuyNFT** (Vacuumlabs)

Alice vende **dos** NFTs; cada validator exige " ≥ 100 ADA a la dirección de Alice".

Eve consume **los dos** UTXOs (los dos NFTs) pero crea **un solo** output de 100 a Alice.

Cada validator ve "un output de 100 a Alice" y aprueba → **Eve se lleva dos NFTs pagando uno.**

Mismo patrón que el escrow con Bob. Viene en distintos **matices** ("shades"):

- dos inputs del **mismo** script compartiendo un output
- inputs de **scripts distintos** que aceptan el mismo output
- combinación con **mint / withdrawals** que confunden la contabilidad

Las mitigaciones

Mitigación	Idea	Complejidad
1. Único input de script	filtrar <code>tx.inputs</code> por tu dirección, exigir <code>count == 1</code>	simple
2. Sumar el valor pagado	total pagado $\geq$ total de todos los inputs del script	media (permite batching)
3. Tagging de outputs	el datum ata cada output a su input	sofisticada

La primera, en Aiken. Todo lo que necesita está en el **contexto**: resuelve su propio `OutputReference` , saca de ahí su dirección de script y cuenta los inputs que la comparten.

```
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
}
```

1b. Validación insuficiente de outputs

El caso más común — la base del double satisfaction. El validator verifica **autorización** (firmas, deadlines) pero no **qué hace la tx con los fondos**.

```
// escrow.ak (Clase 5) — solo verifica la firma:
fn can_claim(d: Datum, tx: Transaction) -> Bool {
  let signed = list.has(tx.extra_signatories, d.beneficiary)
  let on_time = interval.before(d.deadline) |> interval.includes(tx.validity_range)
  signed && on_time
  // ✗ NO verifica que algún output vaya al beneficiario por el monto correcto
}
```

Con la firma presente, **cualquier** tx pasa — los fondos van a donde el armador quiera.

En vivo: `./scripts/demo-bugs-cardano.sh --escena 1` (Bob firma, la plata va a Eve). Y
`aiken check -m "escrow.{..}"` : 10 tests en verde, y **ninguno de los 7 del escrow mira un output**.

Double satisfaction $\subset$ validación insuficiente · el validator es un portero

double satisfaction $\subset$ validación insuficiente

la 1ª es la enfermedad (no mirar el efecto);
la 2ª es un síntoma agudo (con varios inputs)

el validator es un PORTERO

VE toda la tx, pero solo veta/aprueba
SU input — no mueve fondos ni dirige la tx

Todo su poder está en **verificar** el contexto antes de dar el sí. Si no verifica algo, ese algo simplemente **no está garantizado**.

1c. Datum no autorizado (*missing UTxO authentication*)

El datum lo provee quien **crea** el output, no el validator. Cualquiera puede crear un UTxO en la dirección del script.

- **Datum falso al bloquear:** alguien crea un UTxO en el script con un datum arbitrario (poniéndose como `beneficiary`). Si el validator no verifica su origen, lo trata como legítimo.
- **Continuación con datum alterado:** en contratos multi-paso, el validator debe verificar que el datum del output de continuación sea coherente.
- **Mitigación habitual:** un **token de validez** en el UTxO, acuñado por una policy que verificó el datum al crearlo. El validator exige el token; sin él, el UTxO no es suyo.

1d. Dust, min-ADA y agotamiento de recursos

Todo UTXO debe llevar un mínimo de ADA, y toda tx tiene un presupuesto de tamaño y de ejecución.

- **Token dust attack:** enviar muchos UTXOs minúsculos con tokens basura; gastarlos cuesta más que su valor → quedan atascados
- **Validators que asumen montos exactos:** verificar `value == 100 ADA` puede romperse por el min-ADA. Razonar con `>=`, no con `==`
- **Datum o valor sin cota** (*unbounded datum / value*): si crecen con el uso, un día el UTXO no entra en el presupuesto y queda **ingastable**, con los fondos adentro
- **UTXO compartido** (*UTxO contention*): un estado global en un solo UTXO lo disputan todos los usuarios, y con gastar un UTXO barato se bloquea al resto (*cheap spam*)

1e. Seguridad de minting policies

```
// Minting policy frágil: no limita cantidad ni distingue mint/burn
validator token_fragil(creador: ByteArray) {
  mint(_: Void, _policy: PolicyId, tx: Transaction) {
    list.has(tx.extra_signatories, creador)
    // ✗ no chequea cuánto se acuña ni si es mint o burn
  }
  else(_) { fail }
}
```

Errores típicos: no limitar la cantidad, no controlar quemado, no distinguir

`mint > 0` de `mint < 0`.

En vivo: `./scripts/demo-bugs-cardano.sh --escena 6`, un millón de tokens con esta policy.

1f. Asunciones sobre el contexto de la tx

El validator ve **toda** la tx. Bugs por asumir de más:

- que hay **un solo input** del script (puede haber varios → double satisfaction)
- que hay **un solo output** relevante
- un **orden** en las listas (pueden venir en cualquier orden)
- que no hay `mint`, `withdrawals` u otros inputs/outputs inesperados
- que todos los caminos verifican lo mismo: un segundo redeemer que saltea las verificaciones del primero (*other redeemer*)

1g. Tiempo: el validator ve un intervalo, no un reloj

La tx no lleva "la hora": lleva un `validity_range` que **elige quien la arma**, y el nodo sólo garantiza que el bloque cae adentro. El validator razona sobre ese intervalo.

```
// escrow.ak - Claim antes del deadline: (-∞, deadline] debe contener TODO el rango
interval.before(d.deadline) |> interval.includes(tx.validity_range)
```

- Un rango **sin cota superior** (el armador omitió `invalidHereafter`) no está contenido en `(-∞, deadline]` : el escrow lo rechaza. Correcto, y vale el test.
- El error típico es mirar **un solo borde**: `lower_bound < deadline` acepta `[1000, +∞)` , que incluye cualquier hora futura. Verificado: ese chequeo aprueba el rango abierto.
- El deadline es POSIX en milisegundos y la cadena cuenta slots: la conversión es del off-chain, y el validator no puede comprobarla.

En vivo: `./scripts/demo-bugs-cardano.sh --escena 4` (el escrow rechaza el rango abierto) y `--escena 5` (un chequeo de un solo borde lo acepta). Y `aiken check -m unlock_con_rango` : el test del vesting de la Clase 5 que cruza el deadline, en verde con su `can_unlock` .

SEGMENTO 2 · ~15 min

Contraste con la Clase 6

los dos catálogos

Dos catálogos, lado a lado

Dos listas, no una tabla de equivalencias: ningún bug de una columna "es" uno de la otra.

Ethereum (cuentas)

- Reentrancy
- Control de acceso (`msg.sender`)
- Overflow en `unchecked` / casts
- `delegatecall` / proxies
- Front-running / MEV
- Oráculos, DoS por gas

Cardano (EUTXO)

- Double satisfaction
- Validación de outputs
- Datum no autorizado
- Dust / min-ADA, agotamiento de recursos
- Seguridad de minting policies
- Asunciones del contexto de la tx; tiempo

¿Por qué casi no aplican reentrancy/overflow?

Reentrancy

el validator no muta estado
ni hace llamadas externas
no hay "durante" en el que re-entrar

Overflow

los Int de Aiken son bignum
(precisión arbitraria)
sin wraparound de tipos fijos

A cambio aparece **double satisfaction**, que en Ethereum **ni existe**.

Distinta arquitectura, distinto catálogo.

¿Entonces EUTXO es más seguro?

Elimina clases enteras de bugs por construcción: sin estado mutable no hay reentrancy, sin enteros de ancho fijo no hay overflow, y el validator es una función pura sobre un dato que se puede leer entero. Eso es una ventaja real, y es lo que lo hace **más fácil de razonar y de verificar**.

Lo que no elimina es la raíz del otro catálogo: **lo que el validator no verifica, no está garantizado**, y el que arma la tx es el adversario.

- Las auditoras reportan que double satisfaction y datum hijacking **ya casi no aparecen** en contratos nuevos; lo que queda son errores de diseño y de especificación.
- La evidencia es más chica que en Ethereum: menos valor en juego, menos ataques, menos herramientas. Un contrato no es seguro por estar en EUTXO: es seguro si verifica todo.

SEGMENTO 3 · ~30 min

El panorama de herramientas con honestidad

El ecosistema de Cardano, frente al de Ethereum (2026)

Dimensión	Ethereum	Cardano
Estático	Slither, Aderyn (maduros, 100+ detectores)	no hay. Proyectos individuales sin continuidad; IOG anunció uno en 2025, sin release
Fuzzing / property	Foundry, Echidna, Medusa	Aiken + fuzz : integrado al lenguaje, maduro. cooked-validators (Tweag, Haskell): ataques incorporados sobre trazas de txs
Simbólico / formal	Certora, SMTChecker, halmos	Blaster (IOG): hoy una biblioteca de Lean4; la integración con Aiken está en marcha
Práctica	Ethernaut, Damn Vulnerable DeFi	Cardano CTF (Invariant0): 25 niveles en Aiken

Menos comunidad, menos valor en juego, menos herramientas. El nivel automático que en Ethereum es **slither** . acá no existe: lo cubren el **property testing** y la **revisión con checklist**. Son pocas herramientas, pero las que hay alcanzan para este curso.

Qué se usa en Cardano: los tres niveles de garantía (CIP-52)

1. **Tooling automático:** `aiken check` + `fuzz`. Barato, se corre en cada cambio. Es el nivel **central y maduro**; lo que no tiene es un analizador estático.
2. **Auditoría manual con checklist:** **tan importante como en Ethereum**, y acá con **menos herramientas** que la asistan: no hay un estático que le prepare el terreno.
3. **Verificación formal:** la base existe (Blaster); para el dev de Aiken, en construcción.

Es el cuadro de capas de la Clase 6 ordenado por costo. Y hay una razón estructural para que lo formal avance: EUTXO es **más fácil de razonar** (determinismo: el validator es una función pura; localidad: todo está en el contexto). Lo que lo hace contraintuitivo al principio es lo que lo hace **verificable**.

Verificación formal en Cardano: Blaster (IOG)

Misma maquinaria que la Clase 6 (sistema de prueba + SMT), apuntada a Cardano.

	Hoy	En marcha
Qué es	Lean-blaster : una táctica de Lean4 que simplifica el objetivo y manda el resto a z3. PlutusCoreBlaster modela la máquina CEK: se razona sobre UPLC, el nivel que ejecutan los nodos	anotar el validator en su lenguaje (Aiken vía Midgard Labs; Scalus, Pebble, Futura) y obtener prueba o contraejemplo sin escribir Lean
Quién lo usa	quien escribe Lean4. IOG lo aplicó a Djed y a USDCx	propuesta Cardano High Assurance al tesoro de Cardano (2026), sin fecha pública

Clave: se verifica **directo sobre UPLC**, no sobre un modelo abstracto. En Ethereum el SMTChecker razona sobre Solidity; acá, sobre lo que corre en el nodo. Lo que falta es el puente desde Aiken.

SEGMENTO 4 · ~30 min

Property-based testing en profundidad

Cómo funciona el PBT (la técnica)

Un property test tiene un argumento que es un generador (`Fuzzer`), declarado con `via` :

```
test prop(x via fuzz.int()) { ... }
```

1. **Genera** un `x` con el generador, **corre** el cuerpo, exige `True` .
2. Lo repite **100 veces** (el default de `aiken check` ; se cambia con `--max-success`).
Cien `True` seguidos = `PASS [after 100 tests]` .
3. Un `False` corta: ese `x` es el **contraejemplo** y el test falla.
4. **Shrinking**: antes de mostrarlo, Aiken lo **achica** a uno más simple que siga fallando:
de `[12, 441, 0, -2]` a `[-1]` . Así se ve qué lo rompe. En QuickCheck lo programás; Aiken lo hace solo.

Es "para todo x..." aproximado por muestreo: 100 casos, no todos. Y para un validator no hace falta una blockchain: es una función pura `f(datum, redeemer, tx)` , así que el generador construye el **Transaction** como dato y se evalúa la función.

`aiken-lang/fuzz`: los generadores no vienen con Aiken

```
# cardano/aiken.toml
[[dependencies]]
name = "aiken-lang/fuzz"
version = "v2.2.0" # la v3.0.0 (sep 2026) pide Aiken 1.1.24 y stdlib v4: no compila acá
source = "github"
```

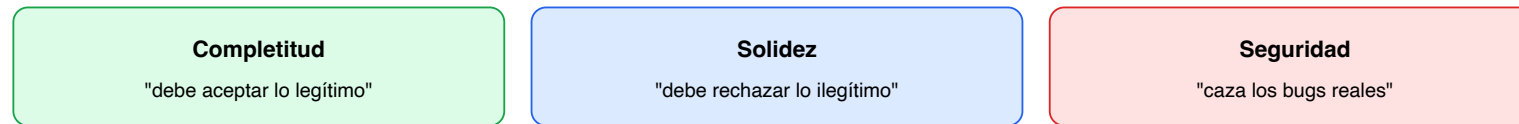
- El primer `aiken check` la baja de GitHub. El `use aiken/fuzz` va arriba, con los demás imports.
- Cada property test suma **sus corridas** al `Summary` (100 por defecto): con el proyecto en verde (12 checks, con el vesting resuelto) y un property test, `Summary 112 checks`.
- En el `aiken.toml` del repo ya está declarada. Los tests unitarios no la necesitan.

¿Cómo se piensa una propiedad de validator?

Un test unitario dice "para *esta* tx, aprueba/rechaza".

Una **propiedad** dice "para *toda* tx que cumpla X, hace Y".

Tres familias:



Ejemplos de propiedades

```
use aiken/fuzz

// Completitud: toda tx legítima es aprobada
test prop_claim_acepta_lo_legitimo(deadline via fuzz.int()) {
  let d = Datum { beneficiary: beneficiary_hash, owner: owner_hash, deadline }
  let tx = Transaction { ..placeholder,
    extra_signatories: [beneficiary_hash],
    validity_range: interval.before(deadline) }
  can_claim(d, tx)
}

// Solidez: ninguna tx sin firma es aprobada
test prop_claim_rechaza_sin_firma(deadline via fuzz.int()) {
  let d = Datum { beneficiary: beneficiary_hash, owner: owner_hash, deadline }
  let tx = Transaction { ..placeholder, extra_signatories: [],
    validity_range: interval.before(deadline) }
  !can_claim(d, tx)
}
```

Los comandos de la clase

```

cd cardano/
aiken check
aiken check -m "escrow_vulnerable.{..}"
aiken check -m "escrow.{..}"
aiken check -m vuln_double
aiken check -e -m "escrow.{claim_sin_firma_falla}"
aiken check -m prop_claim --max-success 1000
aiken check --seed=<n>
./scripts/demo-bugs-cardano.sh --escena 2
# todo el proyecto
# un módulo: el escrow double-satisfiable, 3 tests
# los módulos escrow*: el base y el vulnerable, 10 tests
# un test, por substring del nombre
# un test, por nombre exacto
# las dos propiedades de la slide anterior, mil corridas
# repetir la corrida que imprimió ese seed
# una tx en pantalla, y el validator decidiendo

```

`-m` filtra por **substring**, del nombre del módulo y del test: `"escrow.{..}"` también agarra `escrow_vulnerable`. `-e` exige el nombre exacto del test. Las propiedades corren 100 veces por defecto e imprimen el `--seed`.

Qué devuelve **aiken check**, según el tipo de test

```
$ aiken check -m "escrow_vulnerable.{..}"          # tests unitarios: ExUnits de cada uno
PASS [mem: 119.02 K, cpu: 40.11 M] vuln_claim_legitimo_un_input_un_output
PASS [mem: 92.07 K, cpu: 31.83 M] vuln_claim_sin_output_falla
PASS [mem: 212.81 K, cpu: 72.91 M] vuln_double_satisfaction_se_aprueba_indebidamente
└────────────────────────────────────────────────── 3 tests | 3 passed | 0 failed
```

```
$ aiken check -m prop_claim                        # property tests: cuántas corridas sobrevivieron
PASS [after 100 tests] prop_claim_acepta_lo_legitimo
PASS [after 100 tests] prop_claim_rechaza_sin_firma
└────────────────────────────────────────────────── with --seed=1884641325 → 2 tests | 2 passed | 0 failed
```

```
$ aiken check -m prop_claim_aprobado              # una propiedad que falla: el contraejemplo, achicado
Simplifying counterexample from 1 choices
FAIL [after 1 test] prop_claim_aprobado_paga_al_beneficiario
x counterexample
| #"deadbeef"
└────────────────────────────────────────────────── with --seed=3312377130 → 1 tests | 0 passed | 1 failed
```

Un unitario reporta **cuánto costó**; una propiedad, **cuántos casos aguantó**; y cuando falla, **el caso más chico que la rompe**. El tercero es la siguiente slide.

La propiedad de seguridad que el escrow FALLA

Para todo `Claim` aprobado, debe existir un output que pague al beneficiario.

```
// el destino del único output: Bob o Eve, al azar
test prop_claim_aprobado_paga_al_beneficiario(
  destino via fuzz.either(fuzz.constant(beneficiary_hash), fuzz.constant(eve_hash)),
) {
  let d = Datum { beneficiary: beneficiary_hash, owner: owner_hash, deadline: 10_000 }
  let tx = Transaction { ..placeholder,
    extra_signatories: [beneficiary_hash],
    validity_range: interval.between(1_000, 5_000),
    outputs: [pago_a(destino)] }
  !can_claim(d, tx) || algun_output_para(tx.outputs, beneficiary_hash)
}
```

```
| FAIL [after 1 test] prop_claim_aprobado_paga_al_beneficiario
| × counterexample
| | #"deadbeef"                                ← Eve: Bob firmó, la plata va a Eve, el escrow aprobó
```

Ese fallo es el descubrimiento de la vulnerabilidad. La habilidad del auditor EUTXO:
enumerar lo que el validator debería garantizar y escribir una propiedad por cada garantía.

SEGMENTO 5 · ~30 min

Checklist de auditoría EUTXO

(el entregable)

Checklist — Outputs y double satisfaction

Outputs (lo más importante):

- [] ¿verifica que los fondos van a la dirección correcta?
- [] ¿verifica el monto (con `>=`, considerando min-ADA)?
- [] ¿verifica el datum de los outputs de continuación?
- [] ¿es robusto si hay outputs extra inesperados?

Double satisfaction:

- [] ¿qué pasa si la tx gasta **varios** inputs de este script?
- [] ¿un solo output podría satisfacer dos validaciones?
- [] ¿ata su aprobación a *su* input (vía `OutputReference`)?

Checklist — Datum, contexto, tiempo, firmas

Datum:

- [] ¿se confía en el datum sin verificar su origen?
- [] ¿ `expect Some(d)` maneja el caso `None` ?

Contexto:

- [] ¿se asume un único input/output del script?
- [] ¿se asume un orden en las listas?

Tiempo y firmas:

- [] ¿ `validity_range` correcto (sin asumir un "ahora")?
- [] ¿mira **los dos bordes**? ¿rechaza un rango sin cota?
- [] ¿ `extra_signatories` contra el hash correcto?
- [] ¿la firma sola alcanza, o debería ir con validación de outputs?

Actividad

En grupos, aplicar el checklist completo a `escrow.ak` :

1. Recorrer cada ítem / con justificación
2. Para cada , escribir en una oración **cómo** se explotaría
3. Dibujar (en papel) una tx con dos inputs del escrow y un solo output que engañe a ambos validators
4. Priorizar por impacto: ¿cuál causa mayor pérdida?

El punto no es completar la grilla: es que cada  lo puedan **explicar como un ataque concreto**, con una transacción dibujada. *

SOLUCIÓN · ACTIVIDAD

El checklist sobre el escrow, resuelto

Ítem	Escrow (Clase 5)
Outputs (destino/monto)	✗ no verifica → vulnerabilidad
Double satisfaction	✗ no ata a su OutputReference
Datum maneja None	✓ con expect
Tiempo (validity_range)	✓ includes mira los dos bordes; rechaza el rango abierto
Firmas	✓ hash correcto

El validator **autoriza** bien pero no **constrata el efecto** de la tx. La propiedad de seguridad del Segmento 4 es la que lo destapa.

Lecturas

- **Cardano Developer Portal — "Smart Contract Security"** (Cardano Foundation, 2026): el catálogo oficial, 18 entradas, y el CTF
- **Vacuumlabs / Invariant0 — "Cardano Vulnerabilities" #1 a #6**: double satisfaction (#1, #2), *Trust no UTxO* (#3), *Time handling* (#4), *Token security* (#5, #6)
- **Cardano CTF (Invariant0)**: `github.com/Invariant-0/cardano-ctf` , 25 niveles en Aiken
- **CIP-52** — Cardano Audit Best Practice Guidelines (los tres niveles; estado *Proposed*)
- **Aiken** — tests y property-based: `aiken-lang.org` · `aiken-lang/fuzz` ·
Tweag `cooked-validators` y su retrospectiva de cinco años de auditorías (2026)
- **Plutonomicon (MLabs) — "Common Plutus Vulnerabilities"**; *Mastering Cardano*, cap. 8

Próxima clase

Clase 9 — Taller de análisis (Cardano)

Del checklist al código: tests que destapan, fix que cierra, y el **cierre del curso** comparando los dos mundos.

Traer: `aiken check` en verde y el checklist de hoy.