TECHDOM & CODE · CHAPTER 3

~$ open code-tutorials

Code tutorials

Five short programs that show how to make a tool that touches money safe. Each one runs on fake payments, and this page only shows what it really printed.

ENTRY T3 CATEGORY TECHDOM & CODE CHECKED 3 NOVEMBER 2026
READING 9 MIN SECTIONS 8



ls cover/ T3.webp

EVIDENCE ◆ Verified ◆ Reported ◆ Vivian's field notes

IN 30 SECONDS

01 Every example uses fake payments and runs on your own computer with Node or Python, nothing else installed.

02 A payment is a saved state, so a page reload goes back to waiting and never starts again.

03 The server sets the price and the total, and counts only what the payment provider confirms.

04 A draw is fair when its measured odds match the table shown, and the file that ships must be the file that was tested.

05 Clean an image export, then read it again before you trust it.

YOU WILL LEARN

- ✓ How to run each example and read its real output.
- ✓ Why a saved state, a server-side price and a held unknown outcome protect the person who pays.
- ✓ How to test a draw, a build and an image export instead of trusting them.
- ✓ What a code check can't tell you.

> *This is for people who build tools and for people deciding whether to trust one. Every output below came from a real run, so you can ask any maker for the same proof.*
>
> — VIVIAN

~/01-how-to-use-these-examples

How to use these examples

Techdom tools are usually tested on the maker's own machine, with the maker's own card, and shipped in a hurry. What goes wrong is simple: money

charged wrongly, a live session that freezes, a tool that keeps running after a payment. ◆ The habits below protect the person who plays and the person who builds.

Each example is a folder with a few files and a one-line command. The programs use only what comes with Node or Python. They never open a network connection, a checkout or another person's account, and every amount in them is made up. The rules for a real tool are in Safe design.

```
# Run any example on your own computer
```

run-any-example-on-your-own.md · 5 lines copy

```
1 Copy one example folder to a new folder.
2 Open run.txt. It holds one line, the command to run.
3 Run that command from inside the folder.
4 Compare what prints with the output described on this page.
5 Change one number, run it again, and see which line changes.
```

~/02-1-a-payment-is-a-saved-state

1. A payment is a saved state

A bank confirmation step often sends the person back to the page, and the page reloads. A script that starts from the top on every load then begins the payment again, even when the order message has already appeared. The fix is to keep a record outside the page: the attempt, its state and the last thing the provider said. When the page loads, it reads that record and goes back to waiting. ◆

A timer that unlocks the page after one minute doesn't fix it. It hides the fault and invites a second attempt at the worst moment. ◆

A payment attempt with a fake provider

confirm_state.js

```
'use strict';
// One payment attempt as a small state machine.
// The provider is a fake that lives in memory. No money, no
network, no real checkout.
//
// States of the saved record: awaiting -> confirmed | failed |
held
// Everything the page must remember lives in `store`, which
stands for storage
// that survives a reload. Nothing important is kept in a plain
variable.

const usd = (cents) => 'USD ' + (cents / 100).toFixed(2);

function fakeProvider(answers) {
  const attempts = new Map(); // attempt id -> amount in cents
  let polls = 0;
  return {
    create(id, cents) { if (!attempts.has(id)) attempts.set(id,
cents); },
    status(id) {
      if (!attempts.has(id)) return 'none';
      return answers[Math.min(polls++, answers.length - 1)];
    },
    attemptCount: () => attempts.size,
  };
}

// What runs every time the page loads or reloads.
function loadPage({ provider, store, limitCents }) {
  return {
    pay(cents) {
      if (store.stop) return 'refused: stopped';
// the stop is read first
      const rec = store.record;
      if (rec && (rec.state === 'awaiting' || rec.state ===
```

```
  'held')) {
        return 'refused: already ' + rec.state;
  // a reload never starts a second attempt
      }
      if (store.confirmed + cents > limitCents) return
  'refused: limit ' + usd(limitCents);
      const id = 'attempt-' + (++store.seq);
      store.record = { id, cents, state: 'awaiting' };
      provider.create(id, cents);
      return 'started ' + usd(cents);
    },
    poll() {
      const rec = store.record;
      if (!rec || rec.state !== 'awaiting') return rec ?
  rec.state : 'idle';
      const answer = provider.status(rec.id);
      if (answer === 'pending') return 'pending';
      if (answer === 'confirmed') { rec.state = 'confirmed';
  store.confirmed += rec.cents; }
      else if (answer === 'failed') rec.state = 'failed';
      else rec.state = 'held';
  // any other answer is unknown: hold
      return rec.state;
    },
    resolveHeld(didHappen) {
  // only a person calls this, after checking
      const rec = store.record;
      if (!rec || rec.state !== 'held') return false;
      rec.state = didHappen ? 'confirmed' : 'failed';
      if (didHappen) store.confirmed += rec.cents;
      return true;
    },
  };
  }

  const newStore = () => ({ record: null, confirmed: 0, seq: 0,
  stop: false });
  const say = (label, value) => console.log(label.padEnd(24) + ':
  ' + value);

```

```
  // The fault: a page that starts a fresh attempt every time it
  loads.
  {
    const provider = fakeProvider(['pending']);
    let seq = 0;
    const naiveLoad = () => provider.create('naive-' + (++seq),
  500);
    naiveLoad();   // first load
    naiveLoad();   // the bank page returns and the page reloads
    say('naive page, 1 click', provider.attemptCount() + '
  attempts at the provider');
  }

  // The fix: the record is saved, and a reload resumes waiting.
  {
    const provider = fakeProvider(['pending', 'confirmed']);
    const store = newStore();
    let page = loadPage({ provider, store, limitCents: 800 });
    say('pay USD 5.00', page.pay(500));
    page = loadPage({ provider, store, limitCents: 800 });
  // the reload
    say('pay again after reload', page.pay(500));
    say('attempts at provider', provider.attemptCount());
    say('poll 1', page.poll());
    say('poll 2', page.poll());
    say('confirmed so far', usd(store.confirmed));
    say('pay USD 5.00 again', page.pay(500));
  // 5 + 5 would pass the limit of 8
    store.stop = true;
    say('pay USD 1.00 after stop', page.pay(100));
  }

  // An unknown answer is never repeated. It waits for a person.
  {
    const provider = fakeProvider(['timeout']);
    const store = newStore();
    const page = loadPage({ provider, store, limitCents: 800 });
    page.pay(300);
    say('poll, odd answer', page.poll());
    say('pay while held', page.pay(300));
```

```
  page.resolveHeld(false);
// the person checked: nothing was taken
  say('after checking', store.record.state + ', counted ' +
usd(store.confirmed));
}
```

```
output $ node confirm_state.js
naive page, 1 click     : 2 attempts at the provider
pay USD 5.00            : started USD 5.00
pay again after reload  : refused: already awaiting
attempts at provider    : 1
poll 1                  : pending
poll 2                  : confirmed
confirmed so far        : USD 5.00
pay USD 5.00 again      : refused: limit USD 8.00
pay USD 1.00 after stop : refused: stopped
poll, odd answer        : held
pay while held          : refused: already held
after checking          : failed, counted USD 0.00
```

In the output, the naive page makes two attempts for one click and the saved-record page makes one. The second click is refused while the attempt is awaiting, the total counts only after `confirmed`, a send that would go over the limit of USD 8.00 is refused, a stop refuses the next send, and an odd answer becomes `held`.

Three rules come out of it. The stop is read before anything else, and it blocks new sends; it never hides money that was already confirmed. Don't treat an unknown answer as a failure: nothing repeats until a person has looked at the real account. A read can safely be asked again, but a call that creates a payment must never be sent twice. ◆

---

2. The server decides

Anything a person's own device says about money, time or identity can be changed on that device. So the server holds the rules that matter:

- Prices come from a catalogue on the server. The amount the client claims is never read.
- The amount counted is the amount the provider reports, and a mismatch is held, never rounded away.
- A payment counts only when the provider confirms it, and an order still in doubt counts against the limit at its full amount.
- The person is stored as a keyed hash, never as a name or an email address.
- Time is judged by the server clock, never the phone clock.

A catalogue, a limit and a ledger

server_decides.py

```
"""The server decides the price, the total and who is who. The
client only asks.

Everything here is in memory and fake: there is no payment
provider, no network and no real key.
Amounts are whole cents so that no rounding surprise can hide
in them.
"""
import hashlib
import hmac

CATALOGUE = {"small-gift": 500, "medium-gift": 1000}   # the
only place a price exists
LIMIT_CENTS = 1500                                       # set
by the person who pays, in advance
DEMO_KEY = b"made-up-demo-key"                           # a real
key would never sit in source


def usd(cents):
    return "USD %d.%02d" % divmod(cents, 100)


def pseudonym(identifier):
    """A keyed hash: the ledger can match a person to a past
send without storing who they are."""
    return hmac.new(DEMO_KEY, identifier.encode(),
hashlib.sha256).hexdigest()[:12]


class Ledger:
    def __init__(self):
        self.confirmed = {}   # pseudonym -> cents the provider
confirmed
        self.pending = {}     # order id -> (pseudonym, cents)
still waiting
```

```
        self.next_id = 1

    def counted(self, who):
        """What counts against the limit: confirmed money plus
every order still in doubt."""
        waiting = sum(c for p, c in self.pending.values() if p
== who)
        return self.confirmed.get(who, 0) + waiting

    def order(self, identifier, request):
        who = pseudonym(identifier)
        item = request.get("item")
        if item not in CATALOGUE:
            return "refused: unknown item"
        price = CATALOGUE[item]                       # the
client's own "amount" field is never read
        if self.counted(who) + price > LIMIT_CENTS:
            return "refused: limit %s" % usd(LIMIT_CENTS)
        order_id = "order-%d" % self.next_id
        self.next_id += 1
        self.pending[order_id] = (who, price)
        return "%s created for %s" % (order_id, usd(price))

    def provider_event(self, order_id, status, received_cents):
        """The fake provider reports what really happened. Only
this event can add to the total."""
        if order_id not in self.pending:
            return "ignored: unknown order"
        who, expected = self.pending[order_id]
        if status != "paid":
            del self.pending[order_id]
            return "%s closed, nothing counted" % order_id
        if received_cents != expected:
            return "%s held: expected %s, provider says %s" %
(order_id, usd(expected), usd(received_cents))
        del self.pending[order_id]
        self.confirmed[who] = self.confirmed.get(who, 0) +
received_cents
        return "%s counted %s" % (order_id,
usd(received_cents))
```

```
    ledger = Ledger()
    who = "person@example.invalid"

    print("client claims 1 cent  :", ledger.order(who, {"item":
    "small-gift", "amount": 1}))
    print("unknown item          :", ledger.order(who, {"item":
    "gold-gift"}))
    print("count before the event:",
    usd(ledger.counted(pseudonym(who))))
    print("provider, wrong sum   :", ledger.provider_event("order-
    1", "paid", 100))
    print("provider, right sum   :", ledger.provider_event("order-
    1", "paid", 500))
    print("order 2               :", ledger.order(who, {"item":
    "medium-gift"}))
    print("order 3 over the limit  :", ledger.order(who, {"item":
    "medium-gift"}))
    print("provider, cancelled   :", ledger.provider_event("order-
    2", "cancelled", 0))
    print("order 3 after cancel  :", ledger.order(who, {"item":
    "medium-gift"}))
    print("confirmed total       :",
    usd(ledger.confirmed[pseudonym(who)]))
    print("raw identifier stored :", any(who in str(k) for k in
    ledger.confirmed))
    print("same person, same tag :", pseudonym(who) ==
    pseudonym("person@example.invalid"))
```

```
output $ python server_decides.py
client claims 1 cent  : order-1 created for USD 5.00
unknown item          : refused: unknown item
count before the event: USD 5.00
provider, wrong sum   : order-1 held: expected USD 5.00, provider
says USD 1.00
provider, right sum   : order-1 counted USD 5.00
order 2               : order-2 created for USD 10.00
order 3 over the limit  : refused: limit USD 15.00
```

```
provider, cancelled   : order-2 closed, nothing counted
order 3 after cancel  : order-3 created for USD 10.00
confirmed total       : USD 5.00
raw identifier stored : False
same person, same tag : True
```

The client claims one cent and is still told USD 5.00, because only the catalogue is read. An unknown item is refused. A provider report of the wrong sum is held, a report of the right sum is counted, and a third order is refused because USD 5.00 confirmed plus USD 10.00 in doubt already reaches the limit of USD 15.00. After a cancellation the same order is accepted. The ledger never contains the raw identifier.

The person who pays sets the limit while calm, before the session. The server holds it so it can't be moved mid-session. A test once found an amount to debit that quietly came back as zero, so test zero, a negative number and a very large one. ◆

~/04-3-a-draw-you-can-check

3. A draw you can check

A game with a draw shows a prize table. The only fair promise is that the table is what the code does. ◆ Three habits make that true: check that the table adds up, draw without bias, and measure the real draw against the table.

Mapping a random number with the remainder operator gives some results more chances than others. ◆ The example shows it on a small range, where the effect is easy to count. Then it throws away the top of the range that would favour small tickets, and runs the real draw many times.

A draw measured against its shown odds

fair_draw.js

```
'use strict';
// A draw whose odds match the table shown to the player, with
fictional chips only.
// Three habits: check the table, draw without bias, and
measure the draw against the table.
const { webcrypto } = require('node:crypto');

const SHOWN = [
  ['5 chips', 25], ['10 chips', 25], ['20 chips', 20], ['50
chips', 12],
  ['100 chips', 9], ['250 chips', 6], ['1000 chips', 3],
];
const TICKETS = 10000; // one ticket is 0.01 of a percentage
point

function checkTable(table) {
  const total = table.reduce((sum, [, percent]) => sum +
percent, 0);
  if (total !== 100) throw new Error('table adds up to ' +
total + ', not 100');
  return true;
}

// Habit 1. Mapping a random number with % gives some outcomes
one more chance than others.
function showModuloBias() {
  const counts = new Array(10).fill(0);
  for (let value = 0; value < 256; value++) counts[value %
10]++;
  console.log('bytes 0..255 mapped with % 10 land on 0..9 this
many times:');
  console.log('  ' + counts.join(' '));
}

// Habit 2. Throw away the top of the range that would favour
small tickets.
```

```
const LIMIT = 2 ** 32 - (2 ** 32 % TICKETS);
function randomTicket() {
  const buffer = new Uint32Array(1);
  do { webcrypto.getRandomValues(buffer); } while (buffer[0] >=
LIMIT);
  return buffer[0] % TICKETS;
}

function pickByTable(table) {
  return (ticket) => {
    let edge = 0;
    for (const [name, percent] of table) {
      edge += percent * 100;
      if (ticket < edge) return name;
    }
    throw new Error('ticket out of range');
  };
}

// Habit 3. Run the real draw many times and compare each share
with the shown table.
function matchesShown(pick, shown, rounds) {
  const counts = new Map(shown.map(([name]) => [name, 0]));
  for (let i = 0; i < rounds; i++) {
    const name = pick(randomTicket());
    counts.set(name, counts.get(name) + 1);
  }
  return shown.every(([name, percent]) =>
Math.abs(counts.get(name) / rounds - percent / 100) < 0.005);
}

checkTable(SHOWN);
console.log('shown table adds up to 100:', true);
try {
  checkTable([['5 chips', 50], ['10 chips', 49]]);
} catch (error) {
  console.log('wrong table refused:', error.message);
}

showModuloBias();
```

```
    const draw = pickByTable(SHOWN);
    const edges = [0, 2499, 2500, 4999, 5000, 7000, 8200, 9100,
    9700, 9999];
    console.log('edge tickets give:', edges.map(draw).join(' | '));

    const ROUNDS = 200000;
    console.log('draw matches the shown odds over ' + ROUNDS + '
    rounds:', matchesShown(draw, SHOWN, ROUNDS));

    // A build step that resets every weight to an equal share. The
    source was fine, the shipped draw is not.
    const drawEqual = (ticket) => SHOWN[Math.floor(ticket /
    (TICKETS / SHOWN.length))][0];
    console.log('equal-weights build matches the shown odds:',
    matchesShown(drawEqual, SHOWN, ROUNDS));
```

```
output  $ node fair_draw.js
shown table adds up to 100: true
wrong table refused: table adds up to 99, not 100
bytes 0..255 mapped with % 10 land on 0..9 this many times:
  26 26 26 26 26 26 25 25 25 25
edge tickets give: 5 chips | 5 chips | 10 chips | 10 chips | 20
chips | 50 chips | 100 chips | 250 chips | 1000 chips | 1000 chips
draw matches the shown odds over 200000 rounds: true
equal-weights build matches the shown odds: false
```

A table that adds up to 99 is refused. Counting the bytes 0 to 255 under the remainder operator gives six results 26 chances and four results 25, which is the bias. The edge tickets land on the right prizes, and over 200000 rounds the real draw stays within half a point of every shown share. The build with equal weights doesn't.

The prizes in the example are fictional chips. A checked draw makes a game honest. It doesn't make it safe. A fair game with a house edge still pays back less than it takes, so the limit is what protects the player. ◆ The pressure

designs no draw check can excuse, such as hidden totals and near-misses, are named in Safe design.

> // Vivian's field notes
>
> *An honest prize table costs the maker some income, because it gives up the near-miss and the hidden total. Safe design often earns less. That's just how it is.*
>
> ◆

~/05-4-verify-what-ships

4. Verify what ships

The file you tested isn't always the file you ship. A build step can change behaviour: an obfuscation step can reset the option weights, so the shipped wheel isn't the one that was tested. A wrong version can charge an amount different from the one it shows. A test switch that skips payment can stay on in a build handed to someone else. ◆

Three builds, one checker

ship_check.py

```
"""Check the file that ships, not the file you wrote.

Three small files are built in a temporary folder, so nothing
outside it is touched.
The checker compares the shipped file with the one that was
tested, the odds with the table shown
to the player, the amount shown with the amount charged, and
looks for a leftover test switch.
"""
import hashlib
import json
import pathlib
import tempfile

TESTED = {
    "version": "1.4",
    "odds": {"5 chips": 25, "10 chips": 25, "20 chips": 20, "50
chips": 12,
             "100 chips": 9, "250 chips": 6, "1000 chips": 3},
    "options": [{"label": "small", "shown_cents": 500,
"charged_cents": 500},
                {"label": "large", "shown_cents": 1000,
"charged_cents": 1000}],
    "switches": {"skip_payment": False},
}

# What three different builds might produce from that source.
BUILDS = {"build-a.json": json.loads(json.dumps(TESTED))}

reset = json.loads(json.dumps(TESTED))                # a
build step that flattened the odds
reset["odds"] = {name: 14 for name in reset["odds"]}
BUILDS["build-b.json"] = reset

leftover = json.loads(json.dumps(TESTED))             # a
wrong amount and a test switch left on
```

```
leftover["options"][1]["charged_cents"] = 100
leftover["switches"]["skip_payment"] = True
BUILDS["build-c.json"] = leftover


def sha256(path):
    return hashlib.sha256(path.read_bytes()).hexdigest()


def usd(cents):
    return "USD %d.%02d" % divmod(cents, 100)


def check(shipped_path, tested_hash, shown_odds):
    problems = []
    if sha256(shipped_path) == tested_hash:
        return problems                                 # the
same bytes as the tested file: nothing else to prove
    data = json.loads(shipped_path.read_text(encoding="utf-8"))
    if data["odds"] != shown_odds:
        problems.append("odds differ from the table shown")
    if sum(data["odds"].values()) != 100:
        problems.append("odds add up to %d, not 100" %
sum(data["odds"].values()))
    for option in data["options"]:
        if option["shown_cents"] != option["charged_cents"]:
            problems.append("%s shows %s but charges %s" % (
                option["label"], usd(option["shown_cents"]),
usd(option["charged_cents"])))
    for name, value in data["switches"].items():
        if value:
            problems.append("test switch %s is on" % name)
    return problems


with tempfile.TemporaryDirectory() as folder:
    root = pathlib.Path(folder)
    for name, content in BUILDS.items():
        (root / name).write_text(json.dumps(content, indent=1,
sort_keys=True), encoding="utf-8")
```

```
        tested_hash = sha256(root / "build-a.json")
        print("tested file hash starts with", tested_hash[:12])
        for name in BUILDS:
            path = root / name
            found = check(path, tested_hash, TESTED["odds"])
            print("%s hash %s" % (name, sha256(path)[:12]))
            if found:
                print("  REFUSED")
                for line in found:
                    print("  - " + line)
            else:
                print("  accepted")
```

```
output $ python ship_check.py
tested file hash starts with 148447d35d7d
build-a.json hash 148447d35d7d
  accepted
build-b.json hash 9f7e9766f724
  REFUSED
  - odds differ from the table shown
  - odds add up to 98, not 100
build-c.json hash 63ac7b010e79
  REFUSED
  - large shows USD 10.00 but charges USD 1.00
  - test switch skip_payment is on
```

Build A has the same hash as the tested file and is accepted. Build B has flattened odds that add up to 98 and is refused. Build C shows USD 10.00 but charges USD 1.00, and has a test switch left on, so it's refused too.

A matching hash only proves you hold the tested file. It says nothing about whether that file is good. If a maker publishes no hash, file size is a weak check, and the amount shown against the amount charged is the check that matters.

- The person installing can use the same idea; see Check a tool before you install it.

After any change made by an AI assistant, retest the whole tool. Asked to improve one visual feature, an assistant can also break parts that worked. The safe-design rules apply whichever model wrote the code, and unreviewed code that handles money is a risk to the people who run it. ◆

> // Vivian's field notes
>
> *I don't trust an assistant's first answer on anything that touches money. After retesting everything, I get a second reader, a person or another model, to read the part that holds the money.*
>
> ◆

~/06-5-break-your-guards-on-purpose

5. Break your guards on purpose

A guard that's never been attacked is a guess. A script that breaks each protection in turn on a throwaway copy, and reports which ones held, finds what reading never will. A first version of a guard can let a stop slip through when two stops arrive at the same moment. Testing at the layer that stores the totals shows the gap, and that layer must then be made to refuse. ◆

Never run a test like this on real data. Work on a copy, with fake money.

TRY IT

Which test finds the flaw?

A guard checks that the total plus the new amount stays under the limit, then records the send. In review it looks correct. Which test is most likely to find a flaw?

A Send two requests at the same instant to a throwaway copy and check the stored total

The right answer. Correct. Checking and then recording leaves a gap where both requests pass the check. Test at the layer that stores the totals, and make that layer refuse.

B Read the code once more and confirm the order of the checks

Not this one. Reading already found nothing. The flaw is in the timing, and reading rarely shows timing.

C Send one request at a time, many times

Not this one. Sequential tests pass because the gap never opens.

D Try one real payment of a small amount

Not this one. Real money proves little here and puts a person at risk. A copy with fake money lets you repeat the test as often as you like.

~/07-6-clean-an-export-then-read-it

6. Clean an export, then read it again

Images from generators and editors can carry hidden text: a program name, a time, a place, sometimes the whole prompt. For a persona who stays anonymous, that's a leak. ◆ See Stay anonymous.

The safe method lists what a plain picture needs and drops everything else, instead of listing what to remove. Then the cleaned file is read again, and the export is refused if anything else is left or if the file doesn’t parse. A browser can add a colour profile on its own when it saves an image, so the check has to read what was really written. ◆

An allow-list strip with a second look

scrub_check.js

```
'use strict';
// Clean an image export and then check the result, using a
tiny PNG-shaped file built in memory.
// The rule: name what a plain picture needs, drop every other
chunk, re-read the result,
// and refuse the export if anything else is still there or if
the file does not parse.

const SIGNATURE = Buffer.from([0x89, 0x50, 0x4e, 0x47, 0x0d,
0x0a, 0x1a, 0x0a]);
const KEEP = new Set(['IHDR', 'PLTE', 'tRNS', 'IDAT',
'IEND']);

const TABLE = Array.from({ length: 256 }, (_, n) => {
  let c = n;
  for (let k = 0; k < 8; k++) c = c & 1 ? 0xedb88320 ^ (c >>>
1) : c >>> 1;
  return c >>> 0;
});

function crc32(buffer) {
  let c = 0xffffffff;
  for (const byte of buffer) c = TABLE[(c ^ byte) & 255] ^ (c
>>> 8);
  return (c ^ 0xffffffff) >>> 0;
}

function makeChunk(type, data) {
  const body = Buffer.concat([Buffer.from(type, 'latin1'),
data]);
  const head = Buffer.alloc(4);
  head.writeUInt32BE(data.length);
  const tail = Buffer.alloc(4);
  tail.writeUInt32BE(crc32(body));
  return Buffer.concat([head, body, tail]);
}
```

```
function readChunks(buffer) {
  if (buffer.length < 12 || !buffer.subarray(0,
8).equals(SIGNATURE)) throw new Error('not a PNG file');
  const chunks = [];
  let at = 8;
  while (at < buffer.length) {
    if (at + 12 > buffer.length) throw new Error('truncated
chunk');
    const length = buffer.readUInt32BE(at);
    const end = at + 12 + length;
    if (end > buffer.length) throw new Error('truncated
chunk');
    const type = buffer.toString('latin1', at + 4, at + 8);
    if (crc32(buffer.subarray(at + 4, at + 8 + length)) !==
buffer.readUInt32BE(end - 4)) {
      throw new Error('bad checksum in ' + type);
    }
    chunks.push({ type, start: at, end });
    at = end;
  }
  if (!chunks.length || chunks[0].type !== 'IHDR' ||
chunks.at(-1).type !== 'IEND') {
    throw new Error('missing header or end');
  }
  return chunks;
}

function strip(buffer) {
  const kept = [SIGNATURE];
  const dropped = [];
  for (const chunk of readChunks(buffer)) {
    if (KEEP.has(chunk.type))
kept.push(buffer.subarray(chunk.start, chunk.end));
    else dropped.push(chunk.type);
  }
  return { bytes: Buffer.concat(kept), dropped };
}

// The second look: read the cleaned file again and list
```

```
  anything that is not on the allow-list.
function leftovers(buffer) {
  return readChunks(buffer).map((c) => c.type).filter((type)
=> !KEEP.has(type));
}

function exportSafely(buffer) {
  const { bytes, dropped } = strip(buffer);
  const left = leftovers(bytes);
  if (left.length) throw new Error('refused, still holds ' +
left.join(','));
  return { bytes, dropped };
}

const text = (keyword, value) => makeChunk('tEXt',
Buffer.from(keyword + '\0' + value, 'latin1'));
const original = Buffer.concat([
  SIGNATURE,
  makeChunk('IHDR', Buffer.alloc(13)),
  text('Software', 'example editor 1.0'),
  makeChunk('tIME', Buffer.alloc(7)),
  makeChunk('eXIf', Buffer.from('placeholder camera and place
data')),
  text('prompt', 'a long hidden prompt'),
  makeChunk('IDAT', Buffer.from([1, 2, 3])),
  makeChunk('IEND', Buffer.alloc(0)),
]);

console.log('before  :', readChunks(original).map((c) =>
c.type).join(','));
const result = exportSafely(original);
console.log('dropped :', result.dropped.join(','));
console.log('after   :', readChunks(result.bytes).map((c) =>
c.type).join(','));
console.log('bytes   :', original.length + ' -> ' +
result.bytes.length);

try {
  exportSafely(original.subarray(0, original.length - 5));
} catch (error) {
```

```
    console.log('cut-off file:', 'refused, ' + error.message);
  }

  // Prove that the second look works: a strip that forgets one
  chunk type is caught by the re-read.
  const careless = Buffer.concat([SIGNATURE,
  ...readChunks(original)
    .filter((c) => KEEP.has(c.type) || c.type === 'tEXt')
    .map((c) => original.subarray(c.start, c.end))]);
  console.log('careless strip, second look:',
  leftovers(careless).join(','));
```

```
output $ node scrub_check.js
before  : IHDR,tEXt,tIME,eXIf,tEXt,IDAT,IEND
dropped : tEXt,tIME,eXIf,tEXt
after   : IHDR,IDAT,IEND
bytes   : 202 -> 60
cut-off file: refused, truncated chunk
careless strip, second look: tEXt,tEXt
```

The made-up file holds two text chunks, a time and a camera-style data chunk besides the picture. The strip drops those four, and the file shrinks from 202 to 60 bytes. A file cut short is refused, and a careless strip that forgot text chunks is caught by the re-read.

Video needs the same care: date, program, encoder and hidden notes. Anything handed out should carry no source, comments, handles or other identifying metadata. ◆

~/08-7-what-code-can-t-check-for-you

7. What code can’t check for you

Some things have to be measured. Web accessibility guidance asks that a page doesn’t flash more than three times in any one-second period, unless the flash stays below set thresholds. ◆ Flashing content can cause migraines, dizziness, nausea and seizures. ◆ A tool that makes spirals and loops needs a measured flash check that reads every frame, including the seam where the loop restarts; see Hypno and edits. A page can read the visitor’s request for less motion with the CSS feature `prefers-reduced-motion`. ◆

A working payment path doesn’t prove the sensory layer works. ◆ Flicker, a chaotic spiral or a hum can pass every payment test.

What a maker says, and the test that answers it

WHAT THEY SAY OR DO

01 It works on my machine

THE COUNTER

Test the way the person meets it: signed out, on a phone, and with a fake provider first. ◆

WHAT THEY SAY OR DO

02 One small real payment won't hurt

THE COUNTER

Fake provider first, the provider test mode next, and a real payment only at the very end, tiny, with the amount written down in advance and a recording. Never open-ended. ◆

WHAT THEY SAY OR DO

03 The page unlocks itself after a minute, so a hang can't matter

THE COUNTER

A timer hides a stuck state. Find out why it's stuck and give it a waiting state. ◆

WHAT THEY SAY OR DO

04 The option names are the same, so the links are fine

THE COUNTER

Store a direct link or identifier per option, and retest every option after anything is recreated or renamed. ◆

A tool that handles money is only as safe as the last thing nobody tested.

Small pitfalls

A modal dialog

It can hide the bank window that asks the person to confirm. Test with the real window layout. ◆

A strict content policy

A rule written as “none” next to other sources can break the payment page. ◆

Capitalising styles

They change the text a person sees, and a test that reads the page text then fails. ◆

A preview server

It can keep old modules, so a fix looks broken or a fault looks fixed. Restart it. ◆

TRY IT

Check yourself

QUESTION 1 OF 4

Which value decides what a send costs?

- ◯ The amount the page sends
- ◯ The price in the server catalogue
- ◯ The amount the person types

Answer. The price in the server catalogue. Whoever holds the device can change what the client sends. The server reads its own catalogue.

QUESTION 2 OF 4

A bank confirmation returns and the page reloads. What should the code do?

- ◯ Read the saved record and go back to waiting
- ◯ Start a new attempt to be safe
- ◯ Unlock the page after a minute

Answer. Read the saved record and go back to waiting. A reload must never start a second attempt. A timer only hides the fault.

QUESTION 3 OF 4

The provider sends an answer the code doesn't recognise. What's the safest next state?

- ◯ Held, until a person checks the real account
- ◯ Failed, then retry at once
- ◯ Confirmed, so the player isn't annoyed

Answer. Held, until a person checks the real account. An unknown outcome may be a payment that went through. Repeating it could take the money twice.

QUESTION 4 OF 4

A shipped file has the same hash as the tested file. What does that prove?

- ◯ You hold the file that was tested, nothing about whether it's good
- ◯ The tool is safe to use
- ◯ The odds are fair

Answer. You hold the file that was tested, nothing about whether it's good. A hash identifies a file. Safety and fairness need their own tests.

REMEMBER IT

- Save the payment state outside the page, and let a reload go back to waiting.
- The server sets the price, reads the amount from the provider and counts only confirmed payments.
- Hold an unknown outcome until a person checks. Never repeat it.
- Measure a draw against the table shown, and test the file that ships.
- Strip an export by allow-list, then read it again.
- Test on copies with fake money, and retest the whole tool after every change.

SOURCES AND EVIDENCE

Check it

Every claim with its evidence level and its sources.

EVIDENCE LEVELS

Verified	Checked against the original source.
Reported	Stated by a serious secondary source read in full.
Community	Widely said in the scene; no document proves it.
Vivian's field notes	First-hand experience, not a study.

◆ Unverified	Plausible, not proven yet.

01 Techdom tools are usually tested on the maker's own machine, with the maker's own card, and shipped in a hurry.

◆

02 What goes wrong is simple: money charged wrongly, a live session that freezes, a tool that keeps running after a payment.

◆

03 A bank confirmation step often sends the person back to the page, and the page reloads.

◆

04 A script that starts from the top on every load then begins the payment again, even when the order message has already appeared.

◆

05 The fix is to keep a record outside the page: the attempt, its state and the last thing the provider said.

◆

06 When the page loads, it reads that record and goes back to waiting.

◆

07 A timer that unlocks the page after one minute doesn't fix it.

◆

08 It hides the fault and invites a second attempt at the worst moment.

◆

09 The stop is read before anything else, and it blocks new sends; it never hides money that was already confirmed.

◆

10 Don’t treat an unknown answer as a failure: nothing repeats until a person has looked at the real account.

◆

11 A read can safely be asked again, but a call that creates a payment must never be sent twice.

◆

12 Anything a person’s own device says about money, time or identity can be changed on that device.

◆

13 The amount the client claims is never read.

◆

14 The amount counted is the amount the provider reports, and a mismatch is held, never rounded away.

◆

15 A payment counts only when the provider confirms it, and an order still in doubt counts against the limit at its full amount.

◆

16 The person is stored as a keyed hash, never as a name or an email address.

◆

17 Time is judged by the server clock, never the phone clock.

◆

18 The person who pays sets the limit while calm, before the session.

◆

19 The server holds it so it can’t be moved mid-session.

◆

20 A test once found an amount to debit that quietly came back as zero, so test zero, a negative number and a very large one.

◆

21 The only fair promise is that the table is what the code does.

◆

22 Mapping a random number with the remainder operator gives some results more chances than others.

◆

23 A fair game with a house edge still pays back less than it takes, so the limit is what protects the player.

◆

24 An honest prize table costs the maker some income, because it gives up the near-miss and the hidden total. Safe design often earns less. That's just how it is.

◆

25 A build step can change behaviour: an obfuscation step can reset the option weights, so the shipped wheel isn't the one that was tested.

◆

26 A wrong version can charge an amount different from the one it shows.

◆

27 A test switch that skips payment can stay on in a build handed to someone else.

◆

28 It says nothing about whether that file is good.

◆

29 If a maker publishes no hash, file size is a weak check, and the amount shown against the amount charged is the check that matters.

◆

30 Asked to improve one visual feature, an assistant can also break parts that worked.

◆

31 The safe-design rules apply whichever model wrote the code, and unreviewed code that handles money is a risk to the people who run it.

◆

32 I don’t trust an assistant’s first answer on anything that touches money. After retesting everything, I get a second reader, a person or another model, to read the part that holds the money.

◆

33 A script that breaks each protection in turn on a throwaway copy, and reports which ones held, finds what reading never will.

◆

34 A first version of a guard can let a stop slip through when two stops arrive at the same moment.

◆

35 Testing at the layer that stores the totals shows the gap, and that layer must then be made to refuse.

◆

36 Images from generators and editors can carry hidden text: a program name, a time, a place, sometimes the whole prompt.

◆

37 For a persona who stays anonymous, that’s a leak.

◆

38 The safe method lists what a plain picture needs and drops everything else, instead of listing what to remove.

◆

39 Then the cleaned file is read again, and the export is refused if anything else is left or if the file doesn't parse.

◆

40 A browser can add a colour profile on its own when it saves an image, so the check has to read what was really written.

◆

41 Video needs the same care: date, program, encoder and hidden notes.

◆

42 Anything handed out should carry no source, comments, handles or other identifying metadata.

◆

43 Web accessibility guidance asks that a page doesn't flash more than three times in any one-second period, unless the flash stays below set thresholds.

◆

Web Content Accessibility Guidelines (WCAG) 2.2 (https://www.w3.org/TR/WCAG22/)

Archive (https://web.archive.org/web/20261029121709/https://www.w3.org/TR/WCAG22/)

Accessed 3 November 2026

44 Flashing content can cause migraines, dizziness, nausea and seizures.

◆

Understanding Success Criterion 2.3.1: Three Flashes or Below Threshold | WAI | W3C (https://www.w3.org/WAI/WCAG22/Understanding/three-flashes-or-below-threshold.html)

Archive (https://web.archive.org/web/20261001165853/https://www.w3.org/WAI/WCAG22/Understanding/three-flashes-or-below-threshold.html)

Accessed 3 November 2026

45 A page can read the visitor's request for less motion with the CSS feature prefers-reduced-motion.

◆

prefers-reduced-motion CSS media feature - CSS | MDN (https://developer.mozilla.org/en-US/docs/Web/CSS/Reference/At-rules/@media/prefers-reduced-motion)

Archive (https://web.archive.org/web/20261031201800/https://developer.mozilla.org/en-US/docs/Web/CSS/Reference/At-rules/%40media/prefers-reduced-motion)

Accessed 3 November 2026

46 A working payment path doesn't prove the sensory layer works.

◆

47 Test the way the person meets it: signed out, on a phone, and with a fake provider first.

◆

48 Never open-ended.

◆

49 Find out why it's stuck and give it a waiting state.

◆

50 Store a direct link or identifier per option, and retest every option after anything is recreated or renamed.

◆

51 Test with the real window layout.

◆

52 A rule written as "none" next to other sources can break the payment page.

◆

53 They change the text a person sees, and a test that reads the page text then fails.

◆

54 Restart it.

◆