Everything Stays Local

Document malware, and disarming it without uploading anything

Toilet Duck Web

August 2026

Abstract

A document is not a picture of information. It is a container format with an execution surface accumulated over thirty years, and it is the one executable payload a person is expected to open, from a stranger, as part of their job. This paper sets out the mechanisms by which documents carry and trigger execution, argues that the standard controls fit the problem badly — in particular that submitting a document to a scanning service is a disclosure decision, not a diagnostic one — and describes a content disarm and reconstruction tool built under a single hard constraint: the file never leaves the machine. It covers the detection catalogue, the rewriting engines, the guarantees the design does and does not offer, the descriptive layer that turns a scan into something an analyst can write down, and several implementation mistakes that are more instructive than the design.

1 The shape of the problem

Users believe they are receiving text and pictures. What arrives is a ZIP archive or an OLE2 (Object Linking and Embedding) compound file, containing scripting engines, event hooks, outbound network references, and in many cases other whole files.

This is not an accident of format design so much as its consequence. Office formats were built to automate office work, which means running code, embedding objects from other applications, and referring to resources elsewhere. Every one of those is a feature with a documented specification, and every one is a delivery mechanism.

The reason documents remain the delivery vehicle of choice is social rather than technical. Refusing the attachment is refusing the work: the CV, the invoice, the court filing, the manuscript. An organisation that instructs staff to open nothing from strangers has instructed its recruiters, accounts payable, legal team and editors not to do their jobs.

2 A taxonomy of mechanisms

Five families cover essentially everything. The right-hand column is what matters for a defensive tool: detection sees structure, never intent.

Family	Mechanism	Artifact
Macro code	VBA or Basic runs on open or on event	`vbaProject.bin`, `Basic/`
Embedded object	A second file carried inside the first	`embeddings/`, magic bytes
Remote fetch	The document reaches out when opened	external relationship targets
Reader features	The viewer's own scripting and actions	`/OpenAction`, `/JS`, `/Launch`
Legacy container	Pre-XML formats with richer object models	OLE2 streams, `Ole10Native`

2.1 OOXML: it is a ZIP file

`.docx`, `.xlsx` and `.pptx` are ZIP archives of XML parts. Renaming one to `.zip` opens it, and so does everything inside.

Executable parts.
`word/vbaProject.bin` carries compiled VBA (Visual Basic for Applications) — the famous one. `word/activeX/` instantiates COM (Component Object Model) controls on load. `xl/queryTables/` and `xl/connections/` fetch data on refresh.

Carried parts.
`word/embeddings/` holds arbitrary files, frequently OLE2 droppers. `customXml/` and `docProps/custom.xml` are places to hide data that survives ordinary editing. A part under `*/media/*` whose magic bytes are not an image format is not a picture.

The macro is the one everybody knows. The other rows are the ones that still work when macro execution is disabled by policy, which is the usual state of a managed estate and therefore the usual state of the attacker's target.

2.2 The relationship graph

Every OOXML part is wired to others through `.rels` files, and those relationships may point outside the document:

- `TargetMode="External"` — fetched on open.
- `attachedTemplate` — *remote template injection*. The document is empty and harmless; the template it loads on open is not.
- `oleObject`, `frame`, `hyperlink` — each a documented route elsewhere.
- `docProps/app.xml` carrying a `<Template>` that points at a UNC (Universal Naming Convention) path. Opening it authenticates outward before any content is rendered.

The observation that shapes everything downstream

A file can be malicious while containing nothing malicious. Scanning the bytes of such a document tells you nothing: the payload is at the far end of a URL, fetched at open time, and it can differ per recipient and per hour. No signature, however current, is looking at the right object.

2.3 ODF: the same shape, different names

OpenDocument Format is also a ZIP archive, and the families map across almost exactly: `Scripts/` and `Basic/` carry StarBasic, with the same power as VBA; `META-INF/manifest.xml` declares what runs; `Configurations2/` holds toolbar and menu customisations.

One distinction matters more than it appears. `Scripts/` is a *part*, and can be deleted. A `<script>` element or an `onload=` attribute inside `content.xml` is *inside the document body* — delete that part and you have deleted the document. The two require different remedies, and conflating them is how sanitisers acquire a reputation for destroying files.

2.4 PDF: the reader is the attack surface

A PDF is a graph of objects, several of whose types mean *do something* rather than *draw something*.

Name	Meaning	Severity
`/OpenAction`	run this the moment the file opens	HIGH
`/AA`	additional actions: page open, close, focus	HIGH
`/JavaScript`, `/JS`	the reader's own scripting engine	HIGH
`/Launch`	execute an external program	HIGH
`/EmbeddedFile`	a whole file carried inside	HIGH
`/RichMedia`	Flash and relatives, still parsed by some readers	HIGH
`/XFA`, `/AcroForm`	form engines with their own scripting	REVIEW
`/GoToE`, `/GoToR`, `/SubmitForm`	navigate to embedded, remote, or post data	REVIEW
`/URI`, `/Annots`	links and comments	INFO

The last row is a design statement rather than a detection. A tool that flags `/Annots` on every PDF containing a hyperlink has taught its operators to ignore its output, which is worse than not having flagged it.

2.5 Legacy containers

OLE2 / CFB (Compound File Binary) — `.doc`, `.xls`, `.ppt` — is a small filesystem in a file: storages, streams, a FAT (File Allocation Table), and a mini-FAT for small streams. The names worth flagging are `_VBA_PROJECT`, `Macros`, `PROJECT`, `ObjectPool`, and `\x01Ole10Native`: the Packager stream, which carries an arbitrary file with its original name, extracted to a temporary directory and opened on double-click.

RTF (Rich Text Format) deserves separate mention because it sits between the two worlds. It carries embedded OLE objects like a `.doc`, but it is plain text like an `.xml` — which, as §5.3 argues, is precisely what makes it disarmable in a way OLE2 is not.

2.6 Format confusion and obfuscation

Magic bytes worth finding anywhere inside a document: `4D 5A` (PE, Portable Executable), `D0 CF 11 E0` (OLE2 inside something else), `50 4B 03 04` (nested ZIP), `25 50 44 46` (PDF inside a document), `FWS`/`CWS` (Flash).

A file that is a valid ZIP *and* a valid PDF is a polyglot. The reader picks a parse by extension, the scanner picks by magic bytes, and where they disagree the scanner scanned a different file from the one that executed. This is not an edge case; it is a standing technique, and §5.5 describes what happens to a tool that ignores it.

Two further obfuscations defeat naive scanning outright. PDF names accept hex escapes, so `/JavaScript`, `/J#61vaScript` and `/#4A#61#76#61#53#63#72#69#70#74` are the same name to a reader and different strings to a substring search — normalise before matching, never after. And a PDF may contain several appended revisions: the first `%%EOF` is not the end of the file, and the revision a reader resolves is the last one.

3 Why the standard controls fit badly

"Antivirus will catch it."
Signature detection is retrospective by construction, and remote template injection ships a document with no payload in it at all. There is nothing for a signature to match.

"Just don't enable macros."
Correct, and insufficient. ActiveX, external relationships, embedded objects, and every PDF action in the table above are untouched by that setting.

"Upload it to a scanning service."
This is the one that motivates the design.

> **Submitting a document is a disclosure decision**
>
> The file you are unsure about is a legal filing, a medical record, a source document from a confidential contact, an unannounced set of results. Sending it to a third party to ask whether it is dangerous *publishes it* — and multi-scanner services have historically made submitted samples available to their subscribers.
> You cannot ask that question about a document you are not permitted to share. For a large class of users — journalists, lawyers, clinicians, HR, anyone under NDA — that class is most of their documents, and the tooling silently assumes it does not exist.

4 The constraint is the product

The deliverable is a single `index.html`. It is opened from disk. Nothing leaves the machine.

Forbidden by design: `fetch` and `XMLHttpRequest`, WebSockets, external `<script>`, `<link>` or `<img>`, CDNs, web fonts, service workers, `navigator.sendBeacon`. A `<meta>` CSP (Content Security Policy) ships inside the file, and it would break the page if any of those were reintroduced by accident — the guard is not a policy document, it is a thing that fails loudly.

The consequences are accepted rather than worked around. No library may be used; where one would help, the minimal parser is written instead. There is no build step, no bundler, no package manager. The stated reason is that **one person must be able to audit the deliverable by reading one file**, and every convenience that would weaken that was declined.

5 Engines

5.1 The ZIP rewriter

Locate the end-of-central-directory record, walk the central directory, build an entry table. Entries that survive are *raw-copied* — the compressed bytes are moved across

without an inflate/deflate round trip, which is faster and bit-faithful. Only entries actually edited are decompressed, modified and recompressed.

Removing `vbaProject.bin` is not enough on its own. Its `<Override>` must also go from `[Content_Types].xml`, and its `<Relationship>` from the sibling `.rels`. A macro-free `.docm` that still declares a macro part is a file that no longer opens cleanly.

Unsupported compression methods, oversized archives and Zip64 all take the same path: say so plainly, preserve the bytes, do nothing clever. Reporting "too large to handle" beats emitting a corrupt archive.

5.2 The PDF disarm: length-preserving overwrite

A PDF's cross-reference table stores byte offsets. Change the length of anything and every subsequent offset is wrong, the table no longer resolves, and the file is broken rather than cleaned.

So change no lengths. Overwrite each action name in place with an inert one of exactly the same byte count:

```
/OpenAction   →   /XpenAction
/JavaScript   →   /XavaScript
   /Launch    →   /Xaunch
       /JS    →   /XS
       /AA    →   /XA
```

Every offset, stream length and structural invariant survives untouched. The reader no longer recognises the name, so the action is never dispatched. The file still opens; it simply does nothing.

Matching is done on normalised names but the overwrite happens at the original escaped bytes, so `/J#61vaScript` is neutralised without changing its length either. Names appearing inside content streams are left alone: a literal occurrence in page content is not an action.

5.3 RTF, and a replacement that must end in a digit

Because RTF is plain text, the same length-preserving trick applies to its control words. Eight are neutralised — `\objupdate`, `\objdata`, `\objocx`, `\objautlink` among them — each with a same-length inert replacement. `\objdata`'s hex payload is decoded far enough to identify what is embedded, so a carried executable is named as one rather than reported as an opaque blob.

Why every replacement ends in a digit

The first implementation padded with an underscore. A control word is alphabetic, so `\objdat_` *terminates* the word and Word renders the underscore as literal text in the document — visible to the recipient, and invisible in any test that only checked the payload was gone.

A digit is the control word's parameter. `\objdat0` parses as an unknown word with a parameter and is skipped silently, which is what disarming ought to look like. Three of the eight replacements were also the wrong length on the first pass, which the planner would have skipped without comment; the lengths are now asserted in the self-test.

5.4 OLE2 and MS-OVBA decompression

The CFB parser resolves the FAT and the mini-FAT and exposes a stream reader. The mini-FAT half is load-bearing: streams below the header's cutoff (normally 4096 bytes)

do not live in ordinary sectors but packed inside the root entry's mini stream, and short VBA modules and `Ole10Native` payloads are almost always below it. A sector-only reader returns *empty* for precisely the streams worth reading.

`vbaProject.bin` is not VBA source. It is itself a CFB container whose modules are compressed per MS-OVBA §2.4.1 — a signature byte, then chunks of literals and back-references. Reading raw bytes therefore recovers only the literal runs that happen to survive, which is exactly why `strings` on a malicious document shows scattered VBA keywords but never a whole procedure. The modules are now decompressed and read properly, in both the CFB path and the OOXML one.

Nothing recompresses or rewrites a stream, which keeps the non-goal on OLE2 rewriting intact. Module offsets live in the (also compressed) `dir` stream; rather than parse it, every `0x01` in a stream is tried as a candidate start and the most text-like result kept — simpler, and robust against a malformed directory.

5.5 Content dispatch

Routing was originally by file extension. A `.doc` renamed `.docx` therefore went to the ZIP parser, threw, and surfaced to the operator as *"could not scan this file"* — failing open on the oldest trick there is.

Routing is now by magic bytes, and an extension that disagrees with its content is itself a high-severity finding rather than an error message. A ZIP with an unhelpful name is classified as OOXML or ODF from the parts inside it.

The same class of bug, twice more

A whole macro technique had no rule. `xl/macrosheets/` was absent from the catalogue, so a spreadsheet whose only payload was an XLM (Excel 4.0 macro) sheet scanned clean — a false negative on the technique that dominated 2020–21, precisely because it is not VBA and lives nowhere near `vbaProject.bin`.
And every path rule could be evaded by pressing shift. Fixing the first exposed the second: path matching was case-sensitive, and Excel writes `macroSheets` with a capital S. `word/VbaProject.bin` opens identically in Word and matched nothing at all.
Both have the same shape as the `URLDownloadToFileA` bug in §5.7: a rule that describes the documentation rather than the file.

5.6 Container structure

Some properties belong to the archive rather than to any part, and no per-part rule can see them: data appended past the end-of-central-directory record, duplicate entry names, extreme compression ratios. Each is a way of making two readers disagree about what an archive contains.

Detecting the first required relaxing a check that demanded the record end exactly at end-of-file. That looked like strictness and was in fact a false negative — backwards-scanning readers, Word included, open such an archive happily, so refusing to parse meant an appended payload could ride inside a document the tool had declared unscannable. The strict pass still runs first; the fallback validates the central directory instead of the file position, preserving the original guard against a stray signature inside a comment.

5.7 Capability naming

Everything above answers *what active content is present*. That is a structural question and the only kind a sanitiser can answer honestly. It is also not the question an operator has, which is *what would happen if I opened this*.

Nine behaviours are named, as report-only findings:

Capability		Capability	
Runs automatically when opened	HIGH	Writes a file to disk	REVIEW
Starts another program	HIGH	Uses Windows Script Host or WMI	REVIEW
Runs PowerShell	HIGH	Contains obfuscated code	REVIEW
Downloads content from the network	HIGH	Uses DDE to run a command	HIGH
Writes to the registry or schedules a task	HIGH		

The patterns are keywords for behaviours that Office, WSH (Windows Script Host) and PDF define themselves, matched over text already parsed for other reasons. A capability is therefore never the only reason a file is flagged. Each finding carries its rationale and up to four matched excerpts, because an operator must be able to see why the tool said what it said.

It is deliberately not a signature corpus: nothing ships, nothing updates, nothing is fetched. It is also not intent — a backup macro and a dropper both "download from the network". And it is report-only in the strict sense, being excluded from the auto-select set: what removes the risk is deleting the part that hosts the macro, which the catalogue already offers. Naming behaviour adds no destructive action; it only tells the operator what they are deciding about.

Two bugs, both found by fixtures rather than review

A word boundary that never matched a real macro. The download rule ended in `\b`. VBA declares the ANSI (8-bit, non-Unicode) entry point `URLDownloadToFileA`, so the boundary failed against every realistic `Declare` statement. The rule matched the documentation and missed the malware. It now accepts `[AW]?`.

A long run of base64 characters is not base64. The obfuscation rule matched on character class alone, and fired on a fixture whose macro body is the literal string `FAKEVBA` repeated forty times — a 280-character run inside the class. Real encoded data uses most of the 64-symbol alphabet; a repeated marker uses a handful. The rule now requires at least 24 distinct characters in a match, which required a pattern form where a regular expression can be vetoed by a predicate.

5.8 The descriptive layer

Everything above answers whether a file is dangerous. It does not answer the question an analyst asks immediately afterwards — what do I write down, and what do I pivot on — and a triage tool that cannot answer it sends its user to fetch a second tool, which for a local-only tool is a real cost rather than an inconvenience.

So a separate, strictly read-only layer describes the file. Nothing in it can remove, alter or rewrite a byte.

Per item.
Every archive member: size, compressed size, method, CRC-32, DOS timestamp, com-

pression ratio, Shannon entropy, MD5, SHA-256, and what its magic bytes say it actually is.

Authorship.
OOXML `docProps`, ODF `meta.xml`, PDF `/Info` — author, last-modified-by, application, company, timestamps. The same `lastModifiedBy` across two unrelated-looking documents is frequently a stronger link than anything in either payload.

Indicators.
URLs, domains, IPv4, UNC paths, registry keys, file paths, environment variables, e-mail addresses.

Timeline.
Every timestamp the file admits to, oldest first.

Two decisions in that list are worth defending, because both look wrong.

MD5, in a tool written in 2026

It has been collision-broken since 2004, WebCrypto deliberately declines to implement it, and it decides nothing here — it is printed beside SHA-256, never instead of it.
It is printed because threat-intel sharing still runs on it. The ticket, the vendor portal, the colleague's IOC list: an analyst pasting a hash is pasting MD5 more often than not. Refusing to emit one on principle does not improve anyone's security posture; it just moves the work to a tool with fewer scruples about where the file goes.
Since the platform will not supply it, it is implemented here and checked against RFC 1321's own vectors, plus inputs of 55, 56 and 57 bytes — the padding boundary, which is where a hand-written MD5 usually breaks.

Indicators are matched on **syntax, not reputation**: what a URL looks like, never whether it is known bad. That keeps the non-goal on signature corpora intact — there is no list to ship, expire, or fetch — and it means the tool never claims an indicator is malicious, only that the document references it. They are emitted *defanged* (`hxxp`, `[.]`), because a report is pasted into tickets and chat clients, and one that turns a live sample's URL into a clickable link has manufactured an incident out of a write-up.

The rule that fired on every file

The first version reported a dozen indicators on a document with nothing wrong with it. Every OOXML and ODF file declares its schema namespaces as `http://` URLs, so every document produced a list of W3C and OASIS links. Acceptance test 1 — *clean input yields zero findings* — failed on the first run, which is exactly what it is for.
This is the `/Annots` mistake of §2.4 in a new place. A section that fires on every file has trained its reader to skip the section by the second week, and a skipped section is worse than an absent one: it still costs the space and it still looks like coverage.
The fix excludes *format boilerplate*, which is not the same thing as excluding *known-good*. Nothing is suppressed for being reputable, and a real URL sitting beside a schema namespace is still reported — with a test that fails if that stops being true. The distinction matters, because the second kind of list is the one that quietly becomes a blind spot.

A related trim: the first timeline listed fourteen rows carrying the same timestamp, because archive members are written in one pass. Identical stamps now collapse to a single row, and a timeline with only one distinct value is suppressed entirely — it says nothing the contents table does not already show. The outlier, an entry stamped years

before the rest, is the only thing anyone is reading a timeline for, and it now stands alone.

5.9 Verify after scrubbing

The removal plan states what it *intended* to remove. Only a re-scan states what the output actually contains. The scrubbed file is therefore re-scanned with the engine that produced the findings, and the report says what is confirmed gone.

Its failure paths are tested too: an unchanged file must not verify, and an unreadable output must report a parse error. A check that cannot fail is decoration.

6 Two rules that shaped everything else

Never damage a document silently.
Every destructive action is opt-in via a checkbox and an explicit button. Nothing is removed because the tool believed it knew better.

Doing nothing must return the original file, byte for byte.
With no findings selected, the rewriter emits output identical to its input: same entry order, same external attributes, same DOS timestamps, compressed bytes copied through untouched.

> **Why the second rule earns its keep**
>
> It is the canary for rewriter correctness. A round trip that changes bytes when asked to change nothing is a rewriter that is also corrupting the files it *was* asked to change — you have simply not noticed yet. It is the one test that fails loudly for an entire class of otherwise silent bugs, and it is re-run after every change.

7 Where the tool disagrees with its ancestor

This is a port of a Python sanitiser. Four of that tool's rules were changed, because each destroyed documents:

Original rule	Problem	Replacement
delete `word/theme/`	themes are benign styling	downgrade to informational
delete `word/_rels/`	removing all relationships corrupts OOXML	parse the XML, flag external targets, strip those elements
delete `content.xml`	that *is* the document	strip script elements and event attributes from within it
delete `Pictures/`	removes every image	check magic bytes, flag only non-images

A sanitiser that mangles documents gets switched off, and then it protects nobody. Precision is a security property, not a nicety.

8 Verification

Checked independently, and not only against itself:

- **ZIP output** — Python `zipfile.testzip()` for CRC (cyclic redundancy check) and structure, plus re-reading every surviving part to confirm the text is intact.

- **PDF output** — `pypdf` and `mutool`, deliberately not the tool's own parser, which would prove only that it agrees with itself.
- **CRC-32** — against the known vector `"123456789"` → `0xCBF43926`.
- **Byte-identical round trip** — acceptance test 4, re-run after every phase.
- **Capability rules** — 22 unit checks; a fixture with realistic VBA names all seven expected capabilities, and clean documents name zero.
- **In a real browser** — driven through Chromium, including the download navigation and the computed filename, rather than the engines in isolation.

At the most recent pass: 59 checks green across sniffing, RTF detect, disarm and re-scan, XLM, trailing data, MS-OVBA vectors and capability naming, plus the accumulated regression set. The network grep returns zero. Page weight is 123 KB against a 300 KB budget.

9 Limits

Stated plainly, because a security tool that overstates its scope is worse than one that does less:

- **It does not detect intent.** It finds structures that can execute or fetch. A benign macro and a malicious one are the same finding.
- **It does not rewrite OLE2.** Legacy `.doc` and `.xls` are reported, with advice to convert or print to PDF. Correct CFB rewriting is a larger project than this one.
- **It cannot see inside encryption.** An encrypted PDF is flagged; structural names outside the encrypted streams can still be disarmed, the content cannot be read.
- **Module offsets are found by scanning, not parsing.** The `dir` stream recording where each VBA module begins is itself compressed.
- **It is not a sandbox.** It never opens the document — which is the point, and which means it never observes behaviour either.
- **Zip64 is refused, not handled.**

10 Where it fits

Good fit: documents that cannot be uploaded anywhere; journalism, legal, medical and HR workflows; air-gapped or restricted machines; a first look before deciding whether to escalate to a sandbox.

Bad fit: as the only control in a mail path; as a replacement for detonation; as a guarantee.

> The honest claim is narrow. The tool removes the parts of a document that can act, states exactly what it removed, proves it did not damage the rest, and does all of it without the file leaving the machine.
> That is a smaller claim than "this document is safe". It is also one that can be kept.