



DKIM, Explained

How a message proves it really came from you
and arrived exactly as you sent it

A plain-English guide for people who run businesses, not mail servers. The second in a series — the first covers SPF. No prior knowledge assumed, and no cryptography beyond what changes a decision you make.

01

The problem DKIM solves

SPF checks who handed the message over. It says nothing about what is inside.

SPF answers one question well: was this message sent by a computer you authorised? That is genuinely useful, and it is not the only question worth asking.

A message passes through several machines between you and your recipient. Any of them can alter it. So can anyone who manages to sit in the middle of one of those hops. Nothing in SPF would notice.

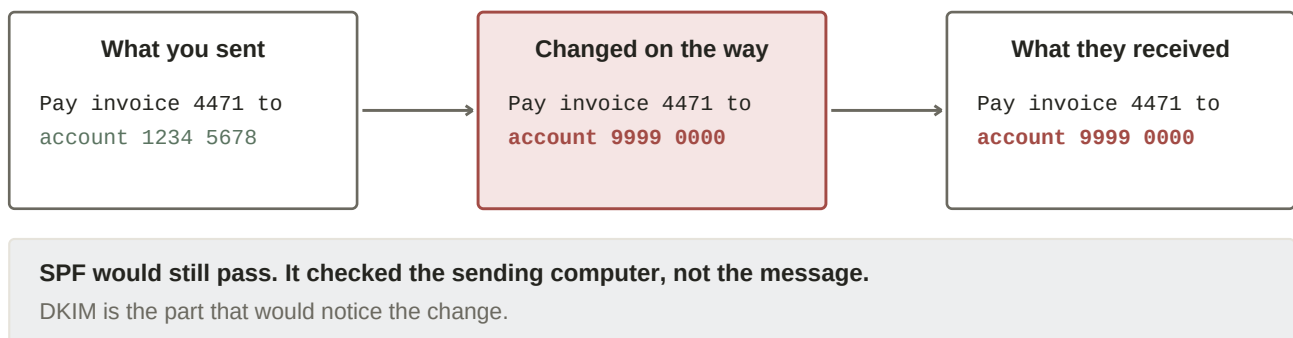


Figure 1 — Same sender, same route, different bank details.

DKIM closes this by attaching something to the message that only you could have produced, and that stops matching if a single character changes. The recipient can check it without ever contacting you.

What DKIM actually is

A signature attached to each message as it leaves. It proves two things at once: that the message was signed by someone holding your domain's secret key, and that nothing has been altered since. Anyone can check it. Nobody else can produce it.

The two keys

The part that sounds wrong the first time: half of it is published deliberately.

DKIM uses a matched pair of keys, generated together. They are not interchangeable, and that asymmetry is what makes the whole thing work.

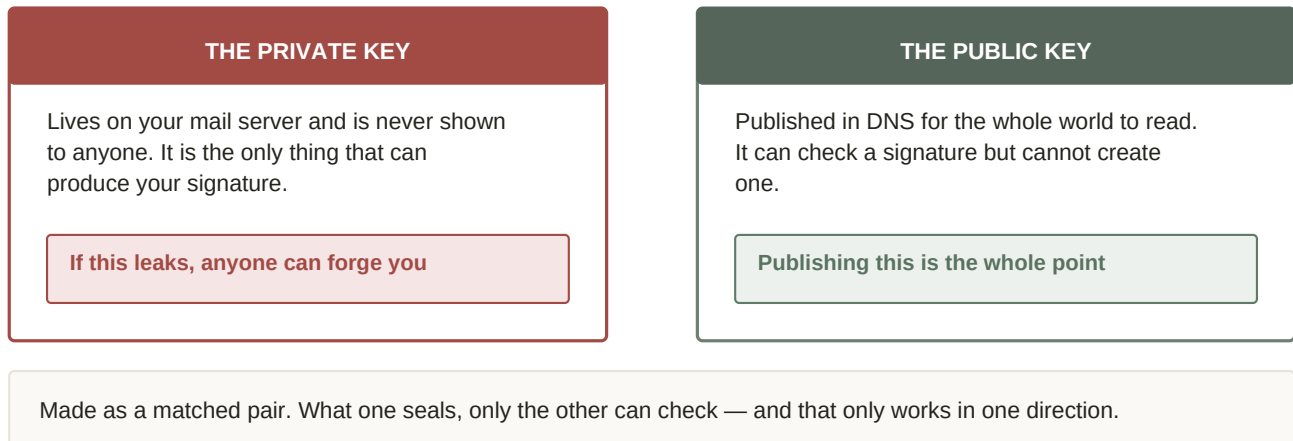


Figure 2 — One key seals, the other checks. Neither can do the other's job.

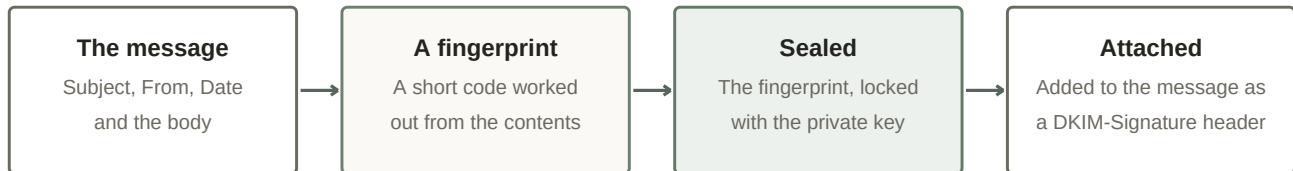
The private key seals; it cannot check. The public key checks; it cannot seal. So publishing the public half gives away nothing: a forger who downloads it can verify your messages all day and still cannot produce one.

Who actually handles the keys

In practice, your mail provider. Google Workspace, Microsoft 365 and every reputable bulk sender generate the pair for you, keep the private half, and hand you a line of text to paste into your DNS. You will almost certainly never see a private key, and if a provider ever emails you one, something is wrong.

What happens when you send

Two steps, both automatic, both on your mail provider's server.



Why a fingerprint and not the message itself?

Because the fingerprint changes completely if even one character of the message changes. Sealing the fingerprint therefore seals the whole message, without having to seal every word of it.

Figure 3 — A fingerprint of the message, sealed so only you could have sealed it.

Why the fingerprint step exists

Sealing a whole message would be slow and would produce something enormous. Instead the server first reduces the message to a short code — a few dozen characters — worked out from every byte of it. Change a comma and that code changes completely and unpredictably.

Sealing that short code therefore seals the entire message. This is the single idea that makes DKIM practical, and the proper name for the code is a **hash**.

What the recipient receives

An ordinary-looking message with one extra header attached. It is not secret and not hidden — it is the instructions for checking:

```
DKIM-Signature: v=1; a=rsa-sha256; d=yourdomain.com; s=mc1;  
h=from:to:subject:date; bh=2jU8...; b=AuyN...
```

What happens when it arrives

The same work, done backwards, by a server that has never heard of you.

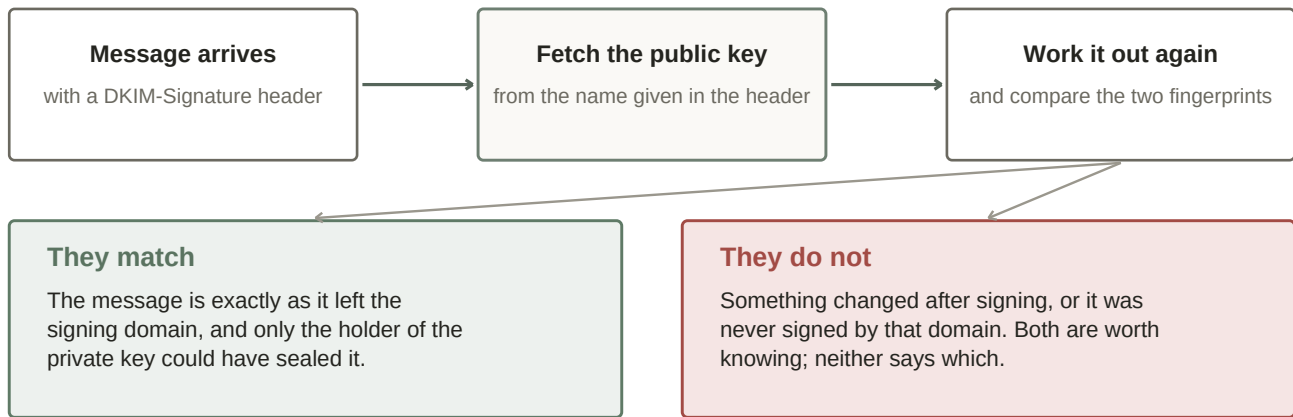


Figure 4 — Fetch the public key, redo the fingerprint, compare.

Notice what is *not* involved: you. The receiving server does not contact your company, and you never learn that the check happened. Everything it needs is either attached to the message or published in your DNS.

A failure does not mean an attack

A signature that does not verify means the message is not bit-for-bit what was signed, or was not signed by that domain at all. Tampering causes that. So does a mailing list adding a footer, a gateway rewriting a subject line, or a key that was rotated too quickly. Section 7 covers which is which.

05

Selectors: why one domain has many keys

The piece with no equivalent anywhere else in email, and the one that confuses everybody.

You probably send mail through several services — your email provider, a newsletter tool, an invoicing system, a helpdesk. Each needs to sign as you, and handing them all the same private key would mean any one of them being breached compromises all of them.

So each gets its own key pair, and each key gets a short name. That name is the **selector**.

One domain can have many keys. Each has a short name — the selector.

google	Google Workspace signs your day-to-day mail
mc1	Mailchimp signs your newsletters
zendesk1	Zendesk signs your support replies

The signature says which one it used, and the receiver builds the DNS name from it:

s=mc1 d=yourdomain.com

-> look up mc1._domainkey.yourdomain.com

This is why you cannot simply "look up a domain's DKIM key". Without the selector there is nothing to look up, and selectors are not guessable.

Figure 5 — The signature names the key it used, and the receiver builds the lookup from it.

A practical consequence

Rotating a key, or removing a service, is a per-selector job. You can retire Mailchimp's key without touching the one Google Workspace uses, and mail signed with the retired key simply stops verifying — which is why you publish the new selector, switch signing over, and only then remove the old one.

Reading a DKIM signature

Seven pieces. Two of them are the ones to look at.

**DKIM-Signature: v=1; a=rsa-sha256;
d=yourdomain.com; s=mc1; h=from:to:subject:date; bh=...; b=...**

v=1

The version. Always 1.

a=rsa-sha256

Which maths was used. rsa-sha256 is the current standard; the older rsa-sha1 must no longer be used.

d=yourdomain.com

The domain taking responsibility for this message. This is the name that matters, and the one DMARC compares against.

s=mc1

Which of that domain's keys was used. Combined with d= it gives the DNS name to look up.

h=from:to:subject:date

Which headers the signature covers. Anything not listed here is not protected.

bh=...

The fingerprint of the body.

b=...

The signature itself — the sealed value everything else is checked against.

The header travels with the message. Nothing in it is secret — it is the instructions for checking.

Figure 6 — The header, pulled apart.

If you only ever read two, read d= and h=. The first says which domain is taking responsibility — and DMARC cares whether that matches the From address your reader sees. The second says how much of the message the signature actually covers.

Also seen	What it means
t=y	Testing mode. Receivers must treat the message as unsigned, however perfect the signature. Useful for a day; harmful if forgotten.
l=	Limits how much of the body is signed. Should never be used — see Section 7.
c=	How whitespace is tidied before hashing. The default tolerates the minor reformatting servers do in transit. Nothing to decide.
x=	An expiry on the signature itself. Rare, and easily set in a way that breaks delayed mail.

What the signature does and does not cover

A signature covers what it says it covers. The gap is where the surprises live.

COVERED	NOT COVERED
The headers named in h= Usually From, To, Subject, Date	Any header not listed in h= It can be added or altered freely
The message body Any change to these breaks it	Attachments added after signing if the signature used an l= limit

The l= tag should never be used.
It limits how much of the body is signed. Anything appended past that point is unsigned, verifies perfectly, and can say whatever the attacker likes. RFC 8301 forbids it outright.

Figure 7 — Anything outside the left column can be changed without breaking anything.

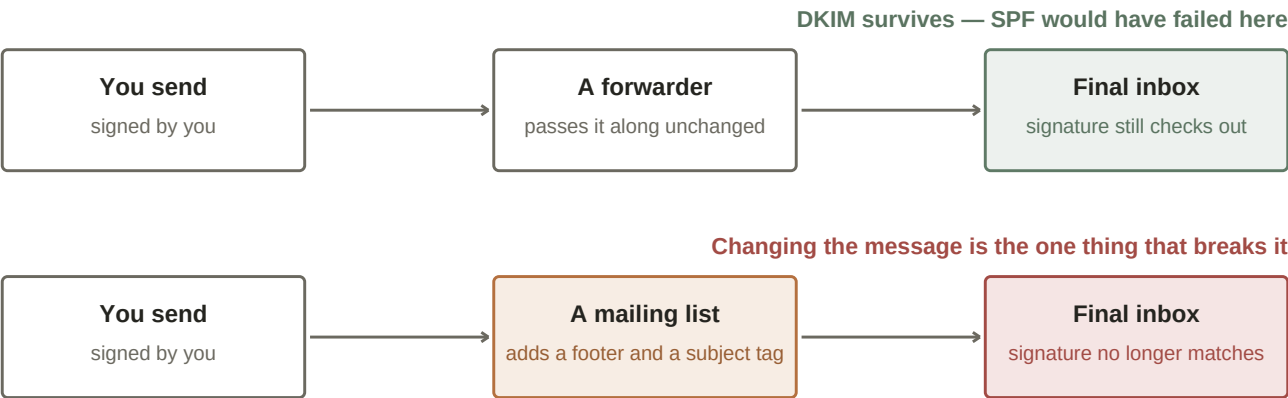
Headers not named in h= are unprotected. Anyone along the path can add one, and the signature still verifies perfectly. This is not a flaw being exploited — it is the signature doing exactly what it was told to do.

The one setting that is genuinely dangerous

The l= tag tells the verifier to sign only the first so-many characters of the body. Everything after that point is unsigned, and an attacker can append whatever they like — new payment instructions, a different phone number — and the signature will still verify. RFC 8301 forbids its use outright. If you see it in your own signatures, ask your provider to remove it.

Forwarding, and the one thing that breaks DKIM

Where SPF fails, DKIM holds. Where DKIM fails, it is usually a mailing list.



This is not a fault in either system. DKIM signs the content, so it travels with the message and notices edits. SPF checks the connection, so it notices a new sender. Between them they cover both, which is why neither is enough alone.

Figure 8 — DKIM travels with the message; SPF is tied to the connection.

This is the clearest illustration of why the two are complementary rather than alternatives. A forwarded message fails SPF and passes DKIM. A message sent directly from an unauthorised server passes neither. A message altered in transit passes SPF and fails DKIM.

Situation	SPF	DKIM
Sent normally from your provider	Passes	Passes
Forwarded, unchanged	Fails	Passes
Altered in transit	Passes	Fails
Sent by a forger from their own server	Fails	Fails
Through a mailing list that adds a footer	Fails	Fails

Why the last row is not a disaster

DMARC requires only that SPF *or* DKIM passes and aligns — never both. A mailing list breaking both at once is the known hard case, and it is why well-run lists rewrite the From address to their own domain and sign as themselves.

Key strength and rotation

Two numbers and one habit.

How big the key should be

RFC 8301 requires at least 1024 bits and recommends 2048. Use 2048 unless something in your DNS genuinely cannot hold a record that long, which is rare and worth fixing rather than working around. Anything below 1024 should be replaced today — it is not merely weak, receivers are entitled to reject it outright.

How often to change it

A private key is a secret that sits on a server for years. Rotating it every six to twelve months limits how long an undetected leak stays useful. Most providers will do this for you; the question to ask is whether they do it at all.

How to rotate without breaking anything

Publish the new selector's public key first, and leave the old one in place. Switch signing to the new key. Wait — a week is generous — for mail signed with the old key to finish being delivered and checked. Only then remove the old record. Removing it first is how a rotation turns into an outage.

A short checklist

What to ask your provider, and how to tell it was done.

Getting it right

- **Sign every stream, not just the main one.** Your newsletter tool, invoicing system and helpdesk each need their own selector. Mail that is not signed at all cannot benefit from any of this.
- **Use 2048-bit keys.** Ask; do not assume.
- **Check that d= matches your domain.** Some providers sign with their own domain by default, which authenticates them rather than you and does nothing for DMARC.
- **Remove t=y once testing is done.** A key left in testing mode provides no protection at all.
- **Rotate on a schedule,** publishing the new key before retiring the old.

Signs something is wrong

- Your signatures contain `l=`. Anything appended to the body is unsigned and will still verify.
- The `d=` value is your provider's domain rather than yours. DMARC will not count it.
- `h=` does not include `f r o m`. The one header that matters most is unprotected.
- A selector's DNS record has been removed while mail is still being signed with it. Everything signed by that key now fails.
- Nobody can say how many selectors your domain has. That is the same problem as not knowing who sends mail as you.

Check your own domain

ReaperCloud's DKIM tool finds your keys without needing you to know the selectors, reports the key size, and flags testing mode and body-length limits. reapercloud.com/dkim.php, or reapercloud.com/report.php for SPF, DKIM and DMARC together.

This document describes DKIM as specified in RFC 6376, with the algorithm and key-size requirements of RFC 8301. It is an explanation, not a specification: where they differ, the RFCs are correct and this is simplified.