



**TRAIL
OF
BITS**

Windows codesigning without Windows: taming the root of trust

William Woodruff



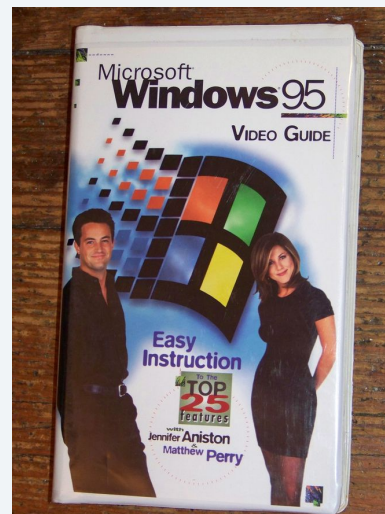
agenda

- quick introduction to codesigning
- codesigning on windows
- codesigning on windows...without windows
 - `oops, no trust`
- taming the root of trust
 - `introducing windows-ctl`



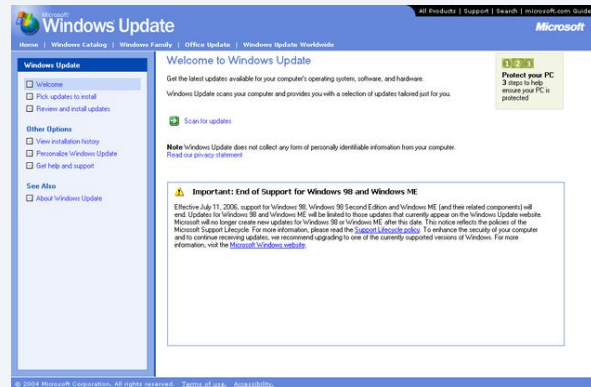
codesigning?

- in the bad old days, software came on physical media w/ holographic stickers
- “legitimate” copies were sold in big box stores and had holographic stickers, tamper-evident packaging, proofs of purchase, etc.
- only a few brave souls dared to distribute (much less charge for) non-trivial software over the internet



codesigning?

- distributing code over the internet is way cheaper, easier, and less stressful
- ...but introduces security problems:
 - unencrypted networks (HTTP, FTP)
 - untrustworthy websites (fake vendors, ad/spyware)
 - policy enforcement (\$corp employees may not use \$app)
 - IR and triage (whose ~~man's~~ binary is this?)
- OS vendors recognized a need (and business line) for codesigning



codesigning?

- every piece of software comes with a *digital signature*
 - **cryptographic proof** that someone (the *private key* holder) signed for the software
 - anybody with the *public key* can verify the signature
- the public (verifying) key **must** be available for use on the client machine
 - how do we get the public key to the client without using the same untrusted channel?



“just add more keys until it works”

- chicken-and-egg problem: we can't distribute the public keys with the signature, because anybody can strip them off and add their own
 - ...and we also can't bake them *directly* into the OS or platform, because not all software vendors are known ahead of time
- ...we need a *public key infrastructure* (PKI): an ecosystem of *policies* and *procedures* that allows us to:
 - *verify and rotate* public keys
 - *revoke* keys that are compromised or insecure (e.g. key sizes too small for modern crypto)
 - *audit and control* who issues valid signatures



codesigning on Windows: authenticode

- [authenticode](#) is Microsoft's code PKI for Windows, with:
 - a digital signature container format, built on [PKCS#7](#);
 - an [X.509](#) certificate and issuing Certificate Authority ecosystem, made up of vendors (Comodo, etc.);
 - an ultimate root Certificate Authority for the issuing CAs, held offline by Microsoft themselves;
- Microsoft distributes a *trust bundle* with Windows that contains the root CA certificate and some issuing CA certs
 - Windows Update periodically updates the trust bundle

PKCS#7
<p>contentInfo</p> <p>Set to SPCIndirectDataContent, and contains:</p> <ul style="list-style-type: none">• PE file hash value• Legacy structures
<p>certificates</p> <p>Includes:</p> <ul style="list-style-type: none">• X.509 certificates for software publisher's signature• X.509 certificates for timestamp signature (optional)
<p>SignerInfos</p> <p>SignerInfo</p> <p>Includes:</p> <ul style="list-style-type: none">• Signed hash of contentInfo• Publisher description and URL (optional)• Timestamp (optional) <p>Timestamp (optional)</p> <p>A PKCS#9 counter-signature, stored as an unauthenticated attribute, which includes:</p> <ul style="list-style-type: none">• Hash value of the SignerInfos signature• UTC timestamp creation time• Timestamping authority signature

authenticode: implementation details

- authenticode digital signatures are baked into Windows executables
 - referenced in the “optional” data directory table, under the “attribute certificate table”
 - certificate table can contain multiple entries, each of which has a length, revision, type, and the actual “certificate” body
 - the “certificate” is actually a custom PKCS#7 SignedData, containing a custom detached signature, certificates (needed to chain against the trust root), and an optional PKCS#9 counter-signature from an RFC 3161 time stamping authority

```
struct win_certificate {
    uint32_t length;           /* dwLength */
    uint16_t revision;        /* wRevision */
    uint16_t certificate_type; /* wCertificateType */
    uint8_t certificate[/* length */]; /* bCertificate */
} __attribute__((aligned (8)));
```



uthenticode: authenticode without windows

- authenticode is 99% standard PKCS#7; the only nonstandard bit is the signed material
 - stored in `SpcIndirectDataContent`, which is a MS-custom PKCS#7 ContentInfo payload
 - boils down to a digest of **most** of the executable being signed (minus the parts that are modified by signature inclusion)
- GitHub: [trailofbits/uthenticode](https://github.com/trailofbits/uthenticode)

```
SpcIndirectDataContent ::= SEQUENCE {
    data          SpcAttributeTypeAndOptionalValue,
    messageDigest DigestInfo
}

SpcAttributeTypeAndOptionalValue ::= SEQUENCE {
    type          OBJECT IDENTIFIER,
    value         [0] EXPLICIT ANY OPTIONAL
}

DigestInfo ::= SEQUENCE {
    digestAlgorithm AlgorithmIdentifier,
    digest          OCTETSTRING
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm      OBJECT IDENTIFIER,
    parameters    [0] EXPLICIT ANY OPTIONAL
}
```



*μ*thenticode: authenticode without windows

- to verify, we extract the body of the `SpcIndirectDataContent` and pass it into `PKCS7_verify` (ugh) as the signed data
- ...we also have to cross-check the digest in `SpcIndirectDataContent` against our own computed digest for the PE, to make sure someone hasn't put a valid signature in an unrelated file.

```
/* Our actual verification happens here.
 *
 * We explicitly ... experimentally ... we to -- the ... directly
 * from the ... p7_'.
 *
 * We ... for the ... since we ... -chain verif...
 * (we ... since we don't ... to Windows ... Publishers stor... non-Windows).
 */
auto status = PKCS7_verify(p7_, certs, nullptr, signed_data.get(), nullptr, PKCS7_NOVERIFY);
```



trust is everything

- µthenticode has a major deficiency: it doesn't have access to the system trust store, so any signature verification it does isn't chained back to an authoritative root of trust
- rephrased: an attacker can put any signature + certificate they control in the authenticode payload, and µthenticode will happily verify it
- arguably a non-issue since the binary *as run* will still go through Windows' own verification (and fail), but it isn't ideal from a completeness perspective
 - can we do better?



untangling the root of trust

to do better, we need to:

1. retrieve the Windows trust store for ourselves;
2. parse whatever format it's in;
3. re-emit it as a "standard" bundle of PEM'd X.509 certs;
4. load the bundle back into an OpenSSL X509_STORE (ugh)
5. use the store during PKCS7_Verify

µthenticode is in C++ but we can do 1-3 in Rust because it's a separate output!
how hard could it be?



untangling the root of trust

Windows Update has a hardcoded URL for fetching the root of trust:

<http://www.download.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab>

```
zeenbox ~
work:tmp william$ wget --quiet http://www.download.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab
work:tmp william$ file authrootstl.cab
authrootstl.cab: Microsoft Cabinet archive data, Windows 2000/XP setup, 62932 bytes, 1 file, at 0x2c +
A "authroot.stl", number 1, 6 datablocks, 0x1 compression
work:tmp william$ cabextract authrootstl.cab
Extracting cabinet: authrootstl.cab
  extracting authroot.stl

All done, no errors.
work:tmp william$ file authroot.stl
authroot.stl: data
work:tmp william$
```



untangling the root of trust

authroot.stl is *another* custom signed PKCS#7 blob, with a ContentType of 1.3.6.1.4.1.311.10.1

unlike the executable format, not really documented anywhere...

from [oidref](#):

Description by oid_info

PKCS #7 ContentType Object Identifier for Certificate Trust List (CTL) szOID_CTL
[View at oid-info.com](#)

```
Alacrity
0:d=0 hl=5 l=165277 cons: SEQUENCE
5:d=1 hl=2 l= 9 prim: OBJECT          :pkcs7-signedData
16:d=1 hl=5 l=165261 cons: cont [ 0 ]
21:d=2 hl=5 l=165256 cons: SEQUENCE
26:d=3 hl=2 l= 1 prim: INTEGER       :01
29:d=3 hl=2 l= 15 cons: SET
31:d=4 hl=2 l= 13 cons: SEQUENCE
33:d=5 hl=2 l= 9 prim: OBJECT         :sha256
44:d=5 hl=2 l= 0 prim: NULL
46:d=3 hl=5 l=161259 cons: SEQUENCE
51:d=4 hl=2 l= 9 prim: OBJECT         :1.3.6.1.4.1.311.10.1
62:d=4 hl=5 l=161245 cons: cont [ 0 ]
67:d=5 hl=5 l=161238 cons: SEQUENCE
72:d=6 hl=2 l= 12 cons: SEQUENCE
74:d=7 hl=2 l= 10 prim: OBJECT        :1.3.6.1.4.1.311.10.3.9
86:d=6 hl=2 l= 9 prim: INTEGER        :1401D8F9409A49C9FA
97:d=6 hl=2 l= 13 prim: UTCTIME      :221115222126Z
112:d=6 hl=2 l= 9 cons: SEQUENCE
114:d=7 hl=2 l= 5 prim: OBJECT        :sha1
121:d=7 hl=2 l= 0 prim: NULL
123:d=6 hl=5 l=161182 cons: SEQUENCE
128:d=7 hl=4 l= 324 cons: SEQUENCE
132:d=8 hl=2 l= 20 prim: OCTET STRING [HEX DUMP]:CDD4EEAE6000
AC7F40C3802C171E30148030C072
:
```

untangling the root of trust

several days later...

[“\[MS-CAESO\]: Certificate Autoenrollment System Overview,”](#) page 52

full ASN.1 definitions! no need to bushwack through DER!!

boils down to a “Certificate Trust List,” in which each TrustedSubject has a SubjectIdentifier...

...but no actual certs anywhere to be seen??

```
CertificateTrustList ::= SEQUENCE {
    version                CTLVersion DEFAULT v1,
    subjectUsage            SubjectUsage,
    listIdentifier          ListIdentifier OPTIONAL,
    sequenceNumber         HUGEINTEGER OPTIONAL,
    ctlThisUpdate          ChoiceOfTime,
    ctlNextUpdate          ChoiceOfTime OPTIONAL,
    subjectAlgorithm        AlgorithmIdentifier,
    trustedSubjects        TrustedSubjects OPTIONAL,
    ctlExtensions          [0] EXPLICIT Extensions OPTIONAL
}

CTLVersion ::= INTEGER {v1(0)}

SubjectUsage ::= EnhancedKeyUsage

ListIdentifier ::= OCTETSTRING

TrustedSubjects ::= SEQUENCE OF TrustedSubject

TrustedSubject ::= SEQUENCE{
    subjectIdentifier      SubjectIdentifier,
    subjectAttributes      Attributes OPTIONAL
}

SubjectIdentifier ::= OCTETSTRING
```




```

pub struct CertificateTrustList {
    /// This trust list's version. The default version is 1.
    #[asn1(default = "Default::default")]
    pub version: CtlVersion,

    /// X.509-style usage.
    pub subject_usage: SubjectUsage,

    /// See [MS-CAES0](https://yossarian.net/junk/hard_to_find/ms-caeso-v20090709.pdf) page 48.
    pub list_identifier: Option<ListIdentifier>,

    /// Some kind of sequence number; purpose unknown.
    pub sequence_number: Option<Uint>,

    /// NOTE: MS doesn't bother to document `ChoiceOfTime`, but experimentally
    /// it's the same thing as an X.509 `Time` (See <https://www.rfc-editor.org/rfc/rfc5280#section-4.1>)
    /// X.509-style time for when this CTL was produced/released.
    pub this_update: Time,

    /// X.509-style time for when the next CTL will be produced/released.
    pub next_update: Option<Time>,

    /// Presumably the digest algorithm used to compute each [`TrustedSubjects`]s identifier.
    pub subject_algorithm: AlgorithmIdentifier<Any>,

    /// The list of trusted subjects in this CTL.
    pub trusted_subjects: Option<TrustedSubjects>,

    // TODO: this should really be `x509_cert::ext::Extensions`
    // but that's a borrowed type and this struct is owning.
    /// Any X.509 style extensions.
    #[asn1(context_specific = "0", optional = "true", tag_mode = "EXPLICIT")]
    pub ctl_extensions: Option<Any>,
}

```

```

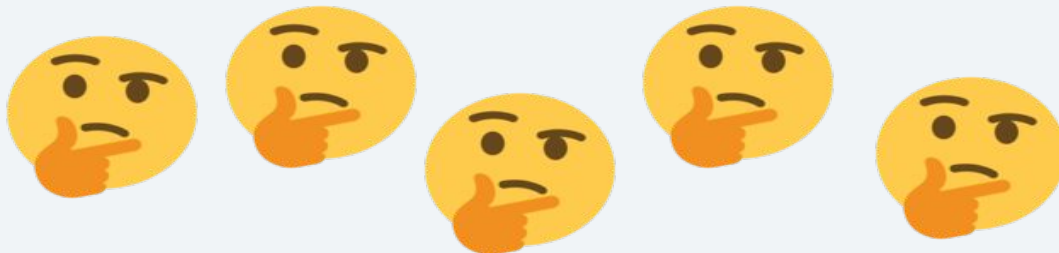
/// ``asn1
/// SubjectIdentifier ::= OCTETSTRING
/// ``
pub type SubjectIdentifier = OctetString;

/// Completely undocumented by MS.
///
/// As best I can tell this is:
///
/// ``asn1
/// MetaEku ::= SEQUENCE OF OBJECT IDENTIFIER
/// ``
pub type MetaEku = Vec<ObjectIdentifier>;

/// Represents a single entry in the certificate trust list.
///
/// From MS-CAES0:
///
/// ``asn1
/// TrustedSubject ::= SEQUENCE {
///     subjectIdentifier SubjectIdentifier,
///     subjectAttributes Attributes OPTIONAL
/// }
/// ``
#[derive(Clone, Debug, Eq, PartialEq, Sequence)]
9 implementations
pub struct TrustedSubject {
    identifier: SubjectIdentifier,
    /// Any X.509 attributes attached to this [`TrustedSubject`].
    pub attributes: Option<Attributes>,
}

```

```
work:windows-ctl william$ ./target/debug/ctltool dump ~/tmp/authroot.stl | jq '.[1]'
{
  "identifier": "18f7c1fcc3090203fd5baa2f861a754976c8dd25",
  "ekus": [
    "1.3.6.1.5.5.7.3.8"
  ]
}
```



Google

18f7c1fcc3090203fd5baa2f861a754976c8dd25

All Images Videos Shopping Maps More Tools

About 2,830 results (0.36 seconds)

<https://github.com> > master > data > microsoft > snapshot

[catt/18F7C1FCC3090203FD5BAA2F861A754976C8DD25 ...](#)

[catt/data/microsoft/snapshot/18F7C1FCC3090203FD5BAA2F861A754976C8DD25.pem](#). Go to file · Go to file T; Go to line L; Copy path; Copy permalink.

People also search for

- [ameroot certificate](#)
- [ip address lookup](#)
- [coinmarketcap](#)
- [binance login](#)
- [ccadb](#)
- [g2 certificate](#)



untangling the root of trust

SubjectIdentifier is just SHA1(cert_der), and uniquely identifies each trust root member on MS's update servers:

<http://www.download.windowsupdate.com/msdownload/update/v3/static/trustedr/en/HASH.crt>

```
Alacritty
work:tmp william$ openssl x509 -in 18f7c1fcc3090203fd5baa2f861a754976c8dd25.crt -inform DER -text
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number:
      4a:19:d2:38:8c:82:59:1c:a5:5d:73:5f:15:5d:dc:a3
    Signature Algorithm: md5WithRSAEncryption
    Issuer: O=VeriSign Trust Network, OU=VeriSign, Inc., OU=VeriSign Time Stamping Service Root, OU=NO LIABILITY ACCEPTED, (c)97 VeriSign, Inc.
    Validity
      Not Before: May 12 00:00:00 1997 GMT
      Not After : Jan  7 23:59:59 2004 GMT
    Subject: O=VeriSign Trust Network, OU=VeriSign, Inc., OU=VeriSign Time Stamping Service Root, OU=NO LIABILITY ACCEPTED, (c)97 VeriSign, Inc.
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (1024 bit)
      Modulus:
        00:d3:2e:20:f0:68:7c:2c:2d:2e:81:1c:b1:06:b2:
        a7:0b:b7:11:0d:57:da:53:d8:75:e3:c9:33:2a:b2:
        d4:f6:09:5b:34:f3:e9:90:fe:09:0c:d0:4b:1b:5a:
        b9:cd:e7:f6:88:b1:9d:c0:87:25:eb:7d:58:10:73:
        6a:78:cb:71:15:fd:c6:58:f6:29:ab:58:5e:96:04:
        fd:2d:62:11:58:81:1c:ca:71:94:d5:22:58:2f:d5:
        cc:14:05:84:36:ba:94:aa:b4:4d:4a:e9:ee:3b:22:
        ad:56:99:7e:21:9c:6c:86:c0:4a:47:97:6a:b4:a6:
```



untangling the root of trust

final step is transformation from individual DER certs to a PEM bundle

turned out to be an excellent stress test for Rust's `x509-cert` due to all kinds of garbage in the MS trust root:

- negative serial numbers (s/o to Agencia Catalana de Certificacio)
- oversized serial numbers (s/o to Krajowa Izba Rozliczeniowa S.A.)

```
let tbs_cert = &cert.tbs_certificate;

writeln!(output, "Serial: {}", tbs_cert.serial_number)?;
writeln!(output, "Issuer: {}", tbs_cert.issuer)?;
writeln!(output, "Subject: {}", tbs_cert.subject)?;
writeln!(output, "Not Before: {}", tbs_cert.validity.not_before)?;
writeln!(output, "Not After: {}", tbs_cert.validity.not_after)?;
writeln!(output, "{}", cert.to_pem(LineEnding::LF)?)?;
```

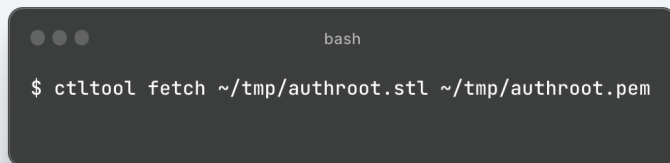


wrapup

we turned this entire adventure into a reusable Rust library + CLI:

<https://github.com/trailofbits/windows-ctl>

(not yet available on crates due to unversioned RustCrypto patches)



```
bash
$ ctltool fetch ~/tmp/authroot.stl ~/tmp/authroot.pem
```

next steps:

- periodically re-build the trust bundle in CI;
- embed in μ thenticode for full chain signature verification!



thank you!

resources:

- [Verifying Windows binaries, without Windows](#) (ToB blog, 2020)
- [trailofbits/uthenticode](#) (OSS Authenticode implementation)
- [trailofbits/windows-ctl](#) (OSS Windows trust root generation)
- [RustCrypto/formats](#) (link to closed PRs for DER, X.509 patches)
- [\[MS-CAES0\]](#) (2013 rev)

contact:

- william@trailofbits.com
- [@yossarian@infosec.exchange](#)

