TRAIL OF BITS

# Windows codesigning without Windows: taming the root of trust

**William Woodruff**

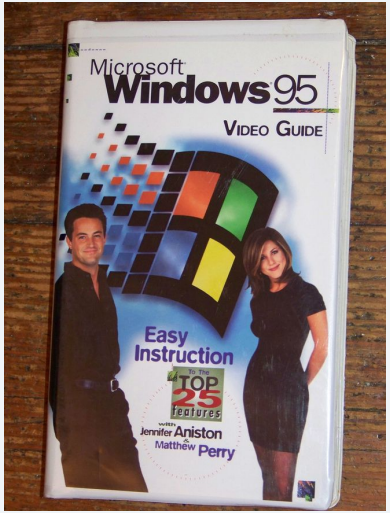


# agenda

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

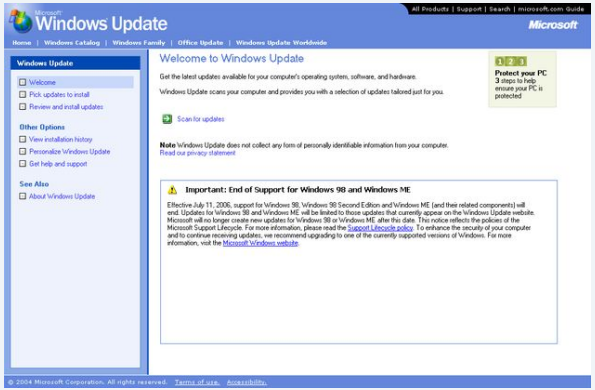
# codesigning?

- 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 ~~mans~~ 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. keysizes 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

# 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)));
```

# *μthenticode*: 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]

```
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
}
```

# *µthenticode*: 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.

# 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`

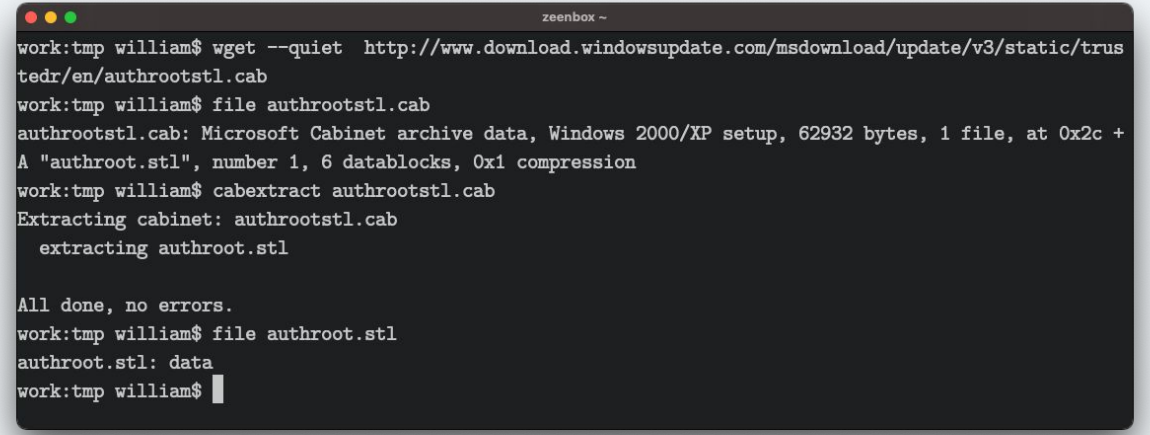
µthenticode is in C++ but we can do 1-3 in Rust because it's a separate output!
how hard could it be?



# 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



# untangling the root of trust

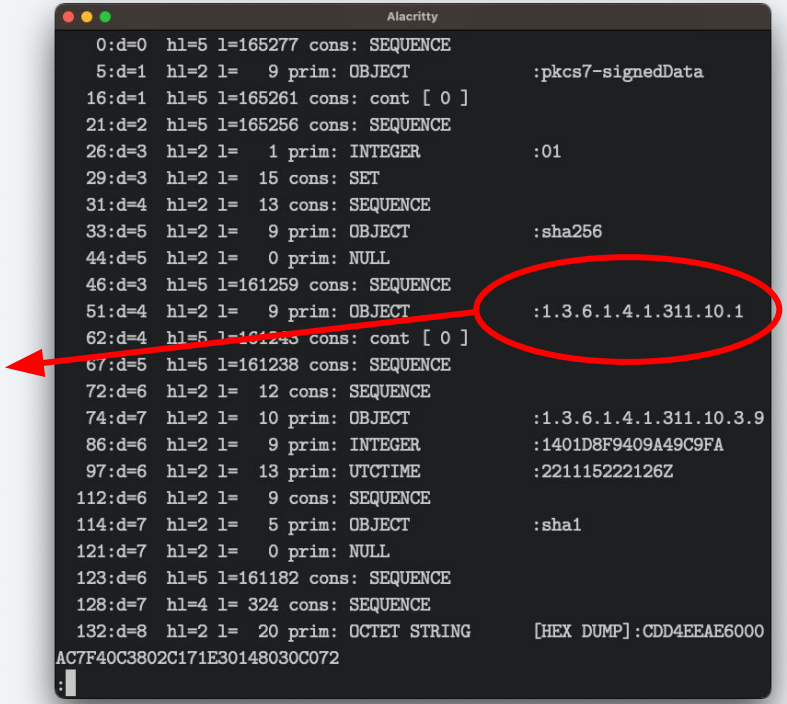
`authroot.stl` is *another* custom signed PKCS#7 blob, with a ContentType of `1.3.6.1.4.1.311.10.1`

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



# 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`…

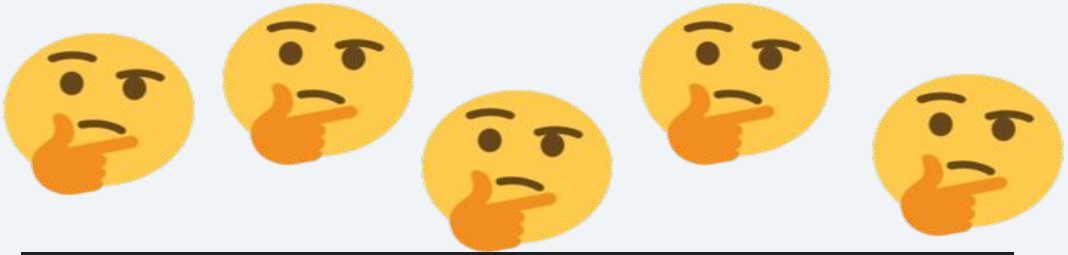
…but no actual certs anywhere to be seen??

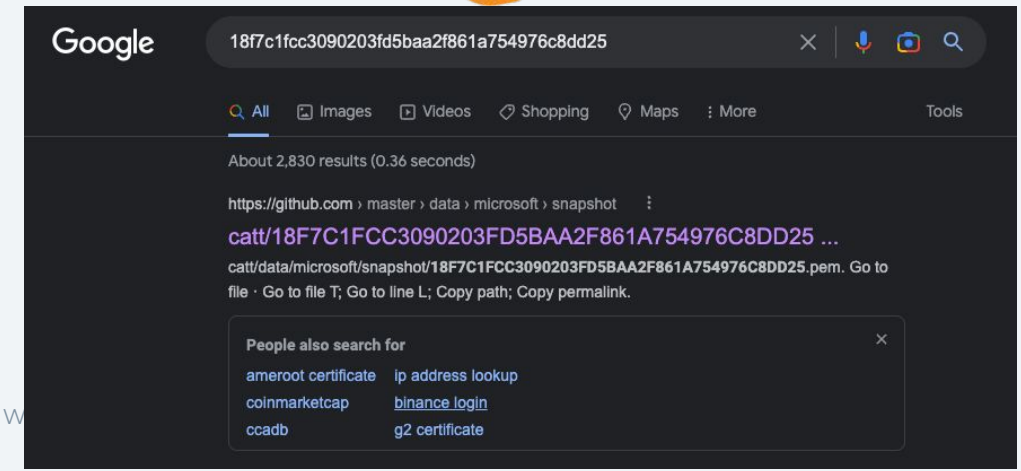
```
CertificateTrustList ::= SEQUENCE {
    version                 CTLVersion DEFAULT v1,
    subjectUsage            SubjectUsage,
    listIdentifier          ListIdentifier OPTIONAL,
    sequenceNumber          HUGEINTEGER OPTIONAL,
    ctlThisUpdate           ChoiceOfTime,
    ctlNextUpdate           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
```

```rust
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-CAESO](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>,
}
```

```rust
/// ```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-CAESO:
///
/// ```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"
  ]
}
```

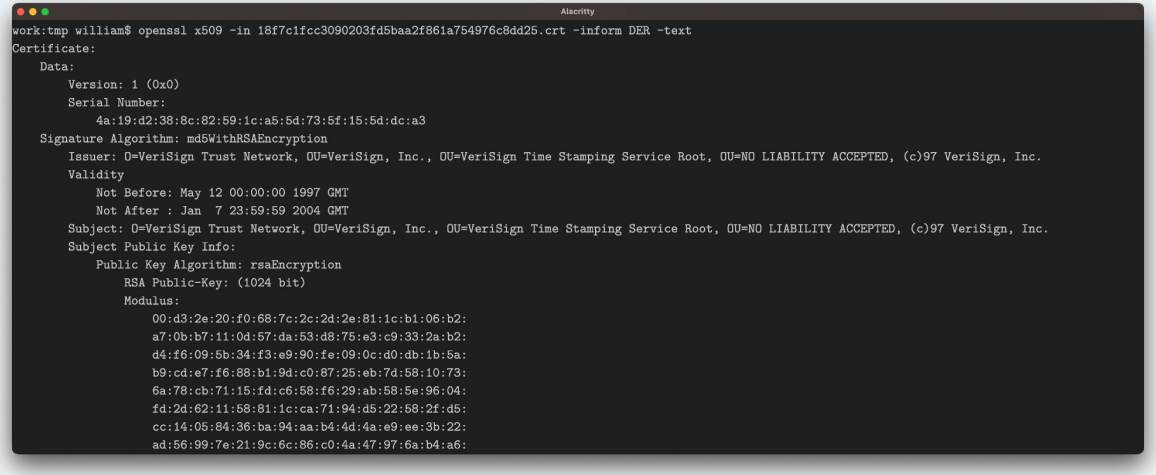




# untangling the root of trust

`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



# 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.)

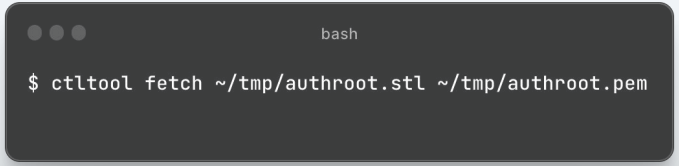
```
let tbs_cert = &cert.tbs_certificate;

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)




next steps:

- periodically re-build the trust bundle in CI;
- embed in µthenticode 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-CAESO] (2013 rev)

contact:


- william@trailofbits.com
- @yossarian@infosec.exchange