Colliding X.509 Certificates

version 1.0, 1st March 2005

Arjen Lenstra[1,2], Xiaoyun Wang[3], and Benne de Weger[2]

[1] Lucent Technologies, Bell Laboratories, Room 2T-504
600 Mountain Avenue, P.O.Box 636, Murray Hill, NJ 07974-0636, USA
[2] Technische Universiteit Eindhoven
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
[3] The School of Mathematics and System Science, Shandong University
Jinan 250100, China

Announcement

We announce a method for the construction of pairs of valid X.509 certificates in which the “to be signed” parts form a collision for the MD5 hash function. As a result the issuer signatures in the certificates will be the same when the issuer uses MD5 as its hash function.

With this construction we show that MD5 collisions can be crafted easily in such a way that the principles underlying the trust in Public Key Infrastructure are violated. In particular we find it worrying that from one certificate alone it cannot be determined whether another, different certificate may exist with the same signature. For the second certificate the issuing Certification Authority may not have been able to verify “proof of possession” of the private key. Therefore, a relying party using a public key certificate based on MD5 can not be certain that the alleged certificate owner is indeed in possession of the corresponding private key.

Construction outline

Our method constructs X.509 certificates in which all fields except the public key can be taken arbitrary, apart from a minor byte length constraint which is easy to fulfill. We use specially crafted but secure public RSA keys.

The heart of our construction is that, starting from a specially crafted MD5-collision produced by the group of Xiaoyun Wang, we can construct a pair of different RSA moduli that yield a collision for the MD5 compression function. Due to the ability of Wang’s method to produce MD5 compression function collisions for any IV, and due to the iterative structure of MD5, we can append a collision to any block of data of our choice (provided that the bitlength is a multiple of the MD5 block length), while maintaining the collision property. Similarly we can then append data of our choice to the constructed collisions. In this way we can build colliding certificates.

The RSA moduli are secure in the sense that they are built up from two large primes. Due to our construction these primes have rather different sizes, but since the smallest still are around 512 bits in size while the moduli have 2048 bits, this does not constitute a realistic vulnerability, as far as we know.

Construction details

We provide a detailed description of our construction.

1. We first construct a template for the certificate, in which all fields are completely filled in, with the exception of the RSA public key modulus and the signature (apart from a first zero byte which is there to prevent the bitstring from representing a negative integer). We can easily meet the following three requirements:
   - the data structure should be compliant to the X.509 standard and the ASN.1 DER encoding rules[1];
   - the byte lengths of the modulus and the public exponent have to be fixed in advance;
   - the position where the public key modulus starts should be an exact multiple of 64 bytes after the beginning of the "to be signed" part.

   The third condition can e.g. be dealt with by adding some dummy information to the subject Distinguished Name. Note that the public key exponent bitlength has to be fixed in advance, it is just as easy to fix the entire public exponent. We take the usual "Fermat-4" number $e = 65537$. It is imperative to have the same e for both certificates.
2. We run the MD5 algorithm on the first portion of the "to be signed" part, truncated at the position where the modulus bytes start. This input to MD5 is an exact multiple of 512 bits. We suppress the padding normally used in MD5, and then get as output an IV that we use as input for the next step.
3. Using the techniques developed by Xiaoyun Wang et al.[2] we construct two different bitstrings b_1 and b_2, of 1024 bits each, for which the MD5 compression function with the IV from the previous step produces a collision.
4. The next step is to construct two RSA moduli from these bitstrings b_1 and b_2 respectively, by appending to each the same bitstring b, also of 1024 bits. This we do as follows.
   - generate random primes p_1 and p_2 of approximately 512 bits, such that e is coprime to $p_1 - 1$ and $p_2 - 1$;
   - compute b_0 between 0 and $p_1 p_2$ such that $p_1 | b_1 2^{1024} + b_0$ and $p_2 | b_2 2^{1024} + b_0$ (by the Chinese Remainder Theorem);
   - let k run through $0, 1, 2, \ldots$, and for each k compute $b = b_0 + k p_1 p_2$; check whether both $q_1 = (b_1 2^{1024} + b)/p_1$ and $q_2 = (b_2 2^{1024} + b)/p_2$ are primes, and whether e is coprime to both $q_1 - 1$ and $q_2 - 1$;
   - when k has become so large that $b \geq 2^{1024}$, restart with new random primes p_1, p_2;
   - when primes q_1 and q_2 have been found, stop, and output $n_1 = b_1 2^{1024} + b$ and $n_2 = b_2 2^{1024} + b$ (as well as p_1, p_2, q_1, q_2).

   It is reasonable to expect, based on the Prime Number Theorem, that this algorithm will produce in a feasible amount of computation time, two RSA moduli $n_1 = p_1 q_1$ and $n_2 = p_2 q_2$, that will form an MD5-collision with the specified IV. When the smaller primes p_1 and p_2 are around 500 bits in size, this algorithm usually returns a result in a few minutes of computation time. When this bitsize increases towards 512 the computation time grows considerably, because the search range for k then becomes almost empty. Nevertheless we have been able to find results with 512 bit p_1, p_2 and 1536 bit q_1, q_2 in a few days of computation time.

[1] See RFC 3280.

[2] See eprint archive article no. 2004/199, and the full paper that is to appear in the proceedings of EuroCrypt 2005.

5. We insert the modulus n_1 into the certificate. Now the “to be signed” part is complete, and we compute the MD5 hash of the entire “to be signed” part (including MD5-padding, and using the standard MD5-IV).
6. We apply standard PKCS#1v1.5-padding[3], and perform a modular exponentiation using the issuing Certification Authority’s private key. This gives the signature, which is added to the certificate. The first certificate now is complete.
7. To obtain the second valid certificate, all we have to do is to replace n_1 for n_2 as the public key modulus. The signature remains valid.

Note that the prime factors of each modulus have rather different sizes. Although this is unusual, for the parameter choices we make (smallest primes of around 500 bits for a modulus of 2048 bits) we see no reason to believe that these moduli are insecure, given the present state of factoring technology. Further note that the corresponding private keys can easily be computed from the public exponent and the prime factors of the moduli. Finding the MD5 collisions seems to be the computationally hardest part of our method, unless one insists on a bitsize for the smallest primes of at least 512.

Example

Below is an example pair of colliding certificates in full detail (byte dump).

The colliding certificates in binary form, as well as the CA certificate and some additional data, can be downloaded from
`http://www.win.tue.nl/~bdeweger/CollidingCertificates/`.

In the left column the exact bytes are presented in a form that clarifies the ASN.1 structure.

```
tag length data                             | comment
=== ====== ================================ | ====================================================
 30 820335                                  | ASN.1 header
------------------------------------------- | ----------------------------------------------------
 30 82021D                                  | ‘‘to be signed’’ part begins here
 A0 03                                      |
 02 01     02                               | X.509 version 3
 02 04     03507449                         | serial number (0x03507449)
 30 0D                                      |
 06 09     2A864886F70D010104               | signature algorithm identifier (md5withRSAEncryption)
 05 00                                      |
------------------------------------------- | ----------------------------------------------------
 30 3D                                      | issuer distinguished name starts here
 31 1A                                      |
 30 18                                      |
 06 03     550403                           |
 13 11     4861736820436F6C6C6973696F6E2043 | issuer common name (‘‘Hash Collision CA’’)
           41                               |
 31 12                                      |
 30 10                                      |
 06 03     550407                           |
 13 09     45696E64686F76656E               | issuer locality (‘‘Eindhoven’’)
 31 0B                                      |
 30 09                                      |
 06 03     550406                           |
 13 02     4E4C                             | issuer country code (‘‘NL’’)
------------------------------------------- | ----------------------------------------------------
```

[3] See PKCS#1 v2.1: RSA Cryptography Standard, Section 9.2.

```
 30 1E                                      |
 17 0D      3035303230313030303030315A       | not valid before (Feb. 1, 2005, 0h0m1s)
 17 0D      3037303230313030303030315A       | not valid after  (Feb. 1, 2007, 0h0m1s)
------------------------------------------- | -----------------------------------------------------
 30 60                                      | subject distinguished name starts here
 31 17                                      |
 30 15                                      |
 06 03      550403                          |
 13 0E      4861736820436F6C6C6973696F6E     | subject common name (‘‘Hash Collision’’)
 31 24                                      |
 30 22                                      |
 06 03      55040A                          |
 13 1B      77652075736564206120636F6C6C6973 | subject organization (‘‘we used a collision for MD5’’)
            696F6E20666F72204D4435           | (dummy text, used to fill up to multiple of 64 bytes)
 31 12                                      |
 30 10                                      |
 06 03      550407                          |
 13 09      45696E64686F76656E               | subject locality (‘‘Eindhoven’’)
 31 0B                                      |
 30 09                                      |
 06 03      550406                          |
 13 02      4E4C                            | subject country code (‘‘NL’’)
 30 820122                                  |
------------------------------------------- | -----------------------------------------------------
 30 0D                                      |
 06 09      2A864886F70D010101               | public key algorithm (rsaEncryption)
 05 00                                      |
 03 82010F 00                               | subject public key info
 30 82010A                                  |
 02 820101 00                               | public key modulus (2048 bits, 257 bytes)
                                            | ‘‘to be signed’’ part until here has a multiple of 64 bytes
                                            | different bytes are indicated by colors and underlining
                                            \----------------------------------\
            (certificate #1)                  (certificate #2)                 |
            CAB9E742C4B626871AB9A524846B05C1  CAB9E742C4B626871AB9A524846B05C1 |
            8895FB9365E9A69F480392FF2C3B3F79  8895FB1365E9A69F480392FF2C3B3F79 |
            41AD3406FFADB4034BDF847A4D37014F  41AD3406FFADB4034BDF847A4DB7014F |
            DB3283CB19D46FA8A765C6B3F016BF30  DB3283CB19D46FA8A765C633F016BF30 |
            6AFF7C2E5773689B3319B81564ABE7F5  6AFF7C2E5773689B3319B81564ABE7F5 |
            B9CF66C5E4FE790CEE047D36CC77B0AE  B9CF6645E4FE790CEE047D36CC77B0AE |
            5D087F30B560EB8872B34D406778662D  5D087F30B560EB8872B34D4067F8652D |
            D88464677DBD9B80989EF24FB82E0EA3  D88464677DBD9B80989EF2CFB82E0EA3 |
            2B5864AF33B8FE8659B094464699F477  2B5864AF33B8FE8659B094464699F477 |
            A6BFCA348C23CF681EC0A846A8B27A29  A6BFCA348C23CF681EC0A846A8B27A29 |
            071B563A1316B05F3827B82FB1F9DE1F  071B563A1316B05F3827B82FB1F9DE1F |
            238F3D12AD0DDAA97DDBCFCEEAD10939  238F3D12AD0DDAA97DDBCFCEEAD10939 |
            5E46E018AE237CE59355AC931872284C  5E46E018AE237CE59355AC931872284C |
            3A293FE9117941A1AD528364A0687AFF  3A293FE9117941A1AD528364A0687AFF |
            6083B14B009DD952C866CA43A0F41A7D  6083B14B009DD952C866CA43A0F41A7D |
            CE5876C16CB346E9A718091CEC3D57D9  CE5876C16CB346E9A718091CEC3D57D9 |
                                            /----------------------------------/
 02 03      010001                          | public exponent (65537)
------------------------------------------- | -----------------------------------------------------
 A3 1A                                      | version 3 extensions start here
 30 18                                      |
 30 09                                      |
 06 03      551D13                          | basic constraints
 04 02      3000                            |
 30 0B                                      |
 06 03      551D0F                          | key usage
 04 04                                      |
 03 02      05E0                            | ‘‘to be signed’’ part ends here
------------------------------------------- | -----------------------------------------------------
 30 0D                                      |
 06 09      2A864886F70D010104               | signature algorithm identifier (md5withRSAEncryption)
 05 00                                      |
------------------------------------------- | -----------------------------------------------------
 03 820101 00                               | signature (2048 bits, 257 bytes)
```

```
           1319E6FF66EF8621AEAE0CFBD2C067B9 |
           9C3834C00BE88E0A97E60205BC5ECD85 |
           646B6698BD2E91324826C8B10E2167EF |
           F264C5E45A234FDE5723A751EA2B7913 |
           06221B54B4C20E4CD16562D698ADE4D6 |
           33F053D653F8BE9C4D402EC9F92D3630 |
           98DD560596F7BF095AF3C9FED7EE2B49 |
           218018003F5C65F0511D454E6E522913 |
           2D0494B7B65EF9585AA9D433094FDB4F |
           9C994610AFE0F23FB26E5D246539AEFF |
           B6E0B0DF35B4D9AE3CF768C5AABC9355 |
           8DF87BF421288E79E9ADCBB8DA236452 |
           8E74F81348FFB9F5FAC43E974F3D79CC |
           A222FD675BFD3B808A3F66104232C806 |
           A25309A187D103D750893436D4A32909 |
           FE5C76B45495F52F29CF66A9E3DD473F |
=== ====== ================================ | =====================================================
```

How to verify

The certificates are valid in the sense that they comply with the relevant standards (RFC 3280, ASN.1 DER encoding), and also in the sense that their digital signature can be verified against the issuing Certification Authority's certificate. For manual verification of our claims we have provided the above byte dumps, as well as further technical data (such as the prime factors of the moduli and the CA public key) at the mentioned website. We would like to advise the interested reader about more convenient ways of verifying our claims. Tools that can be used are e.g. Peter Gutmann's `dumpasn1`[4], and Microsoft's standard Certificate Viewer as it comes with e.g. Windows XP. Unfortunately Microsoft's Certificate Viewer does not show the certificate's signature, but `dumpasn1` does, as the final byte string of length 257. Note that when the CA certificate is installed in the standard Windows (Internet Explorer) Certificate Store, the Certificate Viewer will automatically validate the certificate signatures against the CA certificate.

[4] See `http://www.cs.auckland.ac.nz/~pgut001/`