Problem Set 8–Fake a Signature

CSCI3381-Cryptography

Due December 9, 2014

Just a single problem about RSA signatures. There was a bug in some browsers' implementations of the RSA signature verification algorithm that allowed an attacker to create a fake signature on a server certificate if the certificate authority used $e = 3$ as its public verification exponent. The problem, detailed below, is that the verification algorithm did not verify the entire signature! You will use this to create a fake signature on a certificate.

1 Background

The standard for generating/verifying RSA signatures using the SHA-1 hash function is described in detail in the notes illustrating the start of a secure Web session. Here it is again: To create the signature, the SHA-1 hash $h(m)$ of the message is computed, and then padded to fill out 2048 bits (512 hex digits) as follows:

```
00 01 || FF...FF || Z
```

where `Z` is

```
00 30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 || h(m)
```

We then take this value K and raise it to the d power mod N, where N is the RSA modulus and d the secret signing key. The result is the signature S. To verify the signature, we compute $K = S^e \bmod N$, where e is the public verification exponent, check that the result has the form above, and that the last 160 bits match $h(m)$.

The magic numbers `30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14` are a code that informs the verifier that this is an RSA signature using the SHA-1 hash function. Since RSA signatures might be used with hash functions with different lengths, the signature verification will not be able to predict where in K this block will be found, and instead will have to search for the start of this block, parse it to determine which hash function was used, and then proceed to verify the hash. Thus the verification algorithm can be implemented this way: Check that the first two bytes are 00 01. Check that the subsequent bytes are a bunch of FFs, followed by a 00. Check the fifteen bytes after this to determine the hash function used, and then, if it is SHA-1, check that the next 20 bytes equal the SHA-1 hash of the message. This algorithm is implemented in the `badVerify` function in the accompanying `.py` file posted on the website.

Why is this bad? Because the verification algorithm does not check to see that the SHA-1 hash is the *last* 20 bytes of K. As a result, if the public verification exponent is 3, an attacker can place the crucial 15+20 bytes somewhere other than the right-and end of K, and then play with the low-order bits so that the signature is verified. This bug was pointed out 8 years ago, and variants of the bug have been identified in a number of browsers as recently as this year. (On the other hand, signatures with verification exponent 3 are much rarer.)

Here you are asked to implement an attack based on this weakness. The underlying math is pretty simple, and is similar to the low-exponent attack on RSA encryption that we saw earlier.

2 The attack

One of the accompanying files contains a 'certificate'—not the usual certificates presented by secure websites, but rather an ASCII text file attesting to my many fine qualities. Another file contains the public key information of the signer, as well as the signature, here encoded in decimal as a long integer, that I will present along with my certificate. The public verification exponent is 3.

In addition, I have provided two different verification functions for checking RSA-SHA1 signatures on ASCII strings. The good one is completely correct, the second incorporates the bug described above. You should check that the certificate and signature I have provided are verified as correct by *both* of these signature verification functions.

(a) Your job is to replace my name by yours on the certificate, and produce a signature that will pass the bad verification algorithm with the given public key parameters. Feel free to play around with the text of the certificate, but make sure it contains your name as the grantee.

The idea is to form the 2048-bit padded hash of the certificate as follows (in hex):

```
00 01 || FF...FF || Z || 00 .. 00
```

where `Z` is

```
00 30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 || h(cert)
```

with 180 zero bytes (360 hex digits 0) at the right. Let Y be the integer whose hex representation is the 2048-bit string given above. The ceiling of the cube root of this value,

$$X = \lceil \sqrt[3]{Y} \rceil$$

is a signature on `cert` that will pass the bad verification algorithm. Of course, it should not pass the *good* verification algorithm.

Submit:

- a text file containing your certificate
- the signature X represented as a decimal integer

- $X^3 \bmod N$ represented as a 512-digit hex string—this is the hex string that the verification algorithms work on

(b) Prove that this works: That is, prove that $X^3 \bmod N$ looks to the bad verification algorithm like a properly formated padding of the hash of the certificate. You should be able to see this in the last of the items that you hand in, but your proof should show that the scheme works in general (and should show the reason for using 180 zero bytes at the right).