Signature Forgery Attack on RSA with small e

November 8, 2023

1 Introduction

Ok, boss move: finally a challenge that requires everyone's favorite language `C` (and thus a library for working with big integers, since unlike `Python`, `C` does not support them natively)! We require `C` because it has a specific way of comparing strings: the `strcmp` function compares two strings character by character *until* either a pair of differing characters is found *or* a terminating null-character is found in one of the two strings. If all characters were equal up until that null-character, then `C` will consider the two strings equal even though we as humans probably wouldn't consider e.g., "smorgasbord\0" and "smorg\0" to be identical. Another quick reminder about `C`: there are no booleans, so 0 means `false` and 1 means `true`. Therefore, the statement

```
if !  strcmp(char* str1, char* str2) (...)
```

asks whether `strcmp` returned 0, which it does when the two strings under comparison indeed are equal.

Ok, that was the `C`-specifics! What do we actually want to do? Our task is to sign a random message specified by the server. We have access to the public key (and thus e as well as the modulus n), but not the private exponent d, which is (ideally) required for signing!

2 The Solution

It now is probably a good time to admit that I do not fully understand why this attack works, and digging around in the WWW didn't deliver much - maybe Henning came up with this himself? Anywho, algorithm 2 describes how we go about solving it.

Since the server checks the correctness of the signature we proffer by raising it to the public exponent $e = 3$ and comparing the result to the original message via *strcmp*, we keep appending zero-bytes to the original message until we have found a perfect cube. Apparently we are very likely to succeed in this, and this is the point where I lack understanding - why are we bound to find a perfect

```
1: while ||msg|| < log n do                          ▷ log n = length of n in binary
2:     msg += ´b\x00´
3:     m = int(msg)
4:     s = ⌈∛m⌉
5:     if not strcmp(msg, str(s^e)) then
6:         return s
7:     end if
8: end while
```

cube when we keep appending zeroes? Are perfect cubes just so common? That would be an explanation. Once we have managed to append a certain number of zero-bytes to the original message so that the result (in integer representation, of course) is a perfect cube, we know that when we present the server with a "signature" corresponding to the third root of this perfect cube we will succeed: raising the "signature" to the power of $e = 3$ produces the original message with appended zeroes, which will pass the *strcmp*-check when compared to the original message, and we're done.