Exercise 3: The one where we exploit faults and side channels

Security of Real-World Systems

November 25, 2024

This assignment contributes 30% to your CA grade.
Submission deadline (via Canvas): 06.12.2024, 17:00

1 Fault attack on CRT-RSA (8%)

You are given the Python 3 script `sign.py`, which computes an RSA-CRT signature on a (random) value x. With some probability, a fault will occur in this computation. You can runs this script with `python3 sign.py` on any Linux/Mac/Windows machine/VM with Python 3 installed.
Your job is to extend this script to factor the secret modulus if a fault occurs. To do that, add your code after the line

```
#################### ADD YOUR SOLUTION FROM HERE ##################
```

Any changes you make to or above this line will be discarded and will lead to points being deducted! You must use Python 3 syntax only (e.g. use `print("...")` instead of `print "..."`)!

The code that you add has to do *exactly* the following:

1. Check if the signature is valid (no fault) or invalid (faulty). If the signature is valid, your script should print "no fault" and exit.
2. If the signature is faulty, your script has to use the Lenstra attack (described in the lecture) to recover one prime factor of the modulus n. If the attack succeeds, the script has to print "p = [recovered value]", where the recovered value has to be in hex. Please note that the factors p and q are interchangeable, so your script only has to print the found factor (it does not matter if it matches up with what is called p or q in the script).
3. If the attack fails for some reason, i.e., returns n, your script has to print "useless fault".

You obviously *cannot* use the p/q (or other internal values)—your attack must only use what is printed to the screen (i.e., n, e, x, and the signature).
Submission on Canvas: Your extended Python 3 script named `sign.py`. Submitting more files or using a different filename will lead to points being deducted. Make sure that the script runs on a standard Linux system without additional Python packages etc. Expect your code to be tested with different keys as well. Your script must not print anything beyond what is specified above.

2 Timing attack on AES (22%)

You are given a Linux binary `aes` and a Python 3 script named `timing.py`. Both have to be in the same folder. Make sure that the `aes` binary is executable. When executed with a 32-character hex string, it will encrypt that buffer with AES-128 under a secret key, and output the plaintext, ciphertext, and some timing measurement (see below). For example:

```
./aes 00112233445566778899aabbccddeeff
00112233445566778899aabbccddeeff, 52b4721e41c7ec902906b8aab7049406, 69
```

This binary should run under any reasonable Linux, if not, please get in touch with us. The Python 3 script `timing.py` can be run with `python3 timing.py`. If `aes` is in the same folder and is executable (you might have to do `chmod o+x aes`), it will invoke it N times and store the plaintexts and the respective timing measurements in the variables `plaintexts` and `timings`. We tested that this can be done on the School's server (ssh to `tw.cs.bham.ac.uk`) if you do not have access to an x86 machine with Linux.
The used AES implementation used a variable-time `xtime()` in MixColumns, as explained in detail in the lectures. The timing that the program outputs is the *total* number of `xtime()` invocations (in the whole AES-128) where the "MSBit was one" branch is taken, as also shown in the demo in the lecture. In more detail, the `xtime()` implementation is as follows:

```
static inline uint8_t xtime(uint8_t b, int* t) {
   uint8_t r = b << 1;
   if(b & 0x80) {
      r ^= 0x1B;
      // Increment the total timing returned by the binary
      (*t)++;
   }
   return r;
}
```

Your job is to extend this script so that it performs a differential timing attack on `xtime()` as explained in the lecture, and prints the full AES key in hex. To do that, add your code after the line

```
#### ADD YOUR SOLUTION BELOW THIS LINE ########
```

Hints and remarks:

- To simplify your development, it might make sense to store the measurements into a file first, and then load them when you develop the attack. But make sure that your final submission does *not* do this, it has to work with the standard code in `timing.py`. You can also reduce N at the beginning to find syntax issues etc. without waiting for some time.
- Develop the attack for byte 0 first, and then extend it to all 16 key bytes. Due to the way Python 3 handles integers, this byte can be extracted e.g. as
  `pt_byte = (plaintexts[i] >> (15*8)) & 0xff`
  for measurement i. The correct key byte 0 is `0x29`. Going for other bytes is a fairly minimal change once your attack works for key byte 0.
- While efficiency is not the main goal of this exercise, your code should be "reasonable", i.e. given 3500 measurements it should not take more than 2 min. to recover the key from that.
- Comment your code to explain what's going on. You don't need to comment every single line, but you should explain the overall structure of your program.

Submission on Canvas: Your extended Python 3 script named `timing.py`. Submitting more files or using a different filename will lead to points being deducted. Make sure that the script runs on a standard Linux system without additional packages etc. Expect your code to be tested with different keys as well.