Information Security:

Theory vs. Reality

Exercise 1

corrected 17.11.15

Tel Aviv University

0368-4474-01, Winter 2015-2016

Lecturer: Eran Tromer

Submit a ZIP file containing all your work to istvr1516.course@gmail.com by 1 December 2015.

Include your name and student ID in the subject.

Submissions are individual and must be done independently of other students and any course-specific reference material. Using reference material that is not course-specific is OK, if you state so explicitly in your code and the body of the submission email; provide a pointer to its origin; and adapt it to precisely and elegantly answer the actual question.

A common way to encrypt data is using a *stream cipher.* A stream cipher gets a secret key as its input, and generates a stream of pseudorandom bits. To encrypt a plaintext message, one XORs it with the pseudorandom stream. The security of stream ciphers relies on the fact that to someone who doesn't know the key, the stream looks completely unpredictable, and in particular: even if a prefix of the stream is exposed, the rest should remain unpredictable.

RC4 is a widely used stream cipher, used for WiFi security and many other applications. The internal state of the RC4 algorithm is a table S of 256 bytes. The initial state of this table is some permutation of the values 0,…,255, and this is the secret key.[1] The following code describes the RC4 stream generation algorithm:

```
1     def RC4gen(S):  # S is a 256-byte table initialized to a secret value
2           i = 0
3           j = 0
4           while moreNeeded():
5                 i = (i + 1) % 256
6                 j = (j + S[i]) % 256
7                 S[i],S[j] = S[j],S[i] #swap
8                 Si = S[i]
9                 Sj = S[j]
10                K = S[(Si + Sj) % 256]
11                output(K)
```

Suppose you are given a device which performs RC4 encryption using an unknown secret key, and you would like to recover this key. By analyzing the architecture of the device, you found that it is vulnerable to a very strong cache attack: an attacker, running his malicious code on the same machine, can monitor the RC4 stream generation and observe some of its memory accesses. Specifically, you can learn the precise address accessed in line 9 (but no other information).

In this exercise we will assume knowledge of the addresses learned by the above monitoring, and exploit it to decrypt ciphertext. You are given the following files (attached to the exercise):

> "`ciphertext.bin`" contains the (unknown to you) secret plaintext. During the RC4 encryption which generated this ciphertext, the cache attack was applied.
>
> "`indices.txt`" contains the output of the cache measurement. Each time the read access in line 9 of the above algorithm was executed, the accessed address was recorded to this file. Each line contains one address in decimal. The location of `S` in memory is not known a priori, but you may assume that consecutive entries in `S` have consecutive addresses. For your convenience, attached is the script with which the attack was simulated (`rc4outputIndices.py`).

Given these, you would like to find the full initial state of the table `S`. Knowing this state lets you decrypt the entire ciphertext by yourselves.

A. Write a program which reads "`indices.txt`" and outputs the file "`secret.txt`" – the full initial state of the table `S`, in the following format: 256 lines, one per entry of `S`, ordered by index, in

[1] In the full RC4 cipher, the secret initial state of the table is generated from the *real* secret key using another deterministic algorithm, as shown in `rc4outputIndices.py`.

decimal (for example, if `S[5]` contains the value 200, then line 6 of "`secret.txt`" should contain the three characters "`200`" and a newline).

Attached for your convenience, is a simple Python program "`rc4dec.py`". Use it to check your "`secret.txt`" correctness. To run this program with Python installed, put "`rc4dec.py`" in the same folder as "`ciphertext.bin`" and "`secret.txt`", and then run the following commands:

a. cd *<folder in which the files are in>*
b. python rc4dec.py

Given the correct "`secret.txt`", the plaintext (an image) should – following the script execution – appear in the working directory (where the rest of the files are).
Your program should work for the general attack scenario, i.e. it will be tested given other "`indices.txt`" files. For testing purposes, you may assume that "`indices.txt`" will be in the same directory as your program.

Implement your solution in Python (or request the teaching assistant's permission, in advance, to submit in another language). Submit your source code and your resulting "`secret.txt`". Write clear, high-quality code and include a brief explanation of your program.

B. In reality, the cache attack can fail to determine memory addresses with enough certainty for some iterations of the stream generation. Suppose that every certain number of iterations there is a single iteration for which cache measurements are unreliable and therefore not included in the data. Assume that it is known exactly for which iteration the measurements are not available. For the iterations in which the attack works successfully, you can assume that all the read accesses are disclosed fully (i.e. the whole 32 bit addresses for reads in line 6-10 are available). Explain (at a high level) how to modify your solution above in order to carry out the attack. Implementation is not required.

It turns out that the compiler implements the modular reduction operator "%" using this internal function:

```
M1    unsigned int mod_int(unsigned int i, unsigned int m) {
M2          while (i>=m)
M3                i -= m;
M4          return i;
M5    }
```

C. Describe how an attacker can exploit a (micro)architectural side channel to detect information about the loop iterations in `mod_int`. Explicitly specify your assumptions and what you learn.

D. The initial (secret) value of the `S` array is generated, by some deterministic algorithm, from a secret password. Suppose that this password is very short (4 characters). Using only the side-channel leakage about loop iterations in `mod_int`, how can you find the password?