Information Security:

Theory vs. Reality

Exercise 1

Tel Aviv University

0368-4474-01, Winter 2011-2012

Lecturer: Eran Tromer

Teaching Assistant: Roei Schuster (royschuster@mail.tau.ac.il)

Question 1

Consider the following scenarios in writing secure application on an Android-like mobile phone platform.

Imagine a third-party smartphone application called "SafeRing" – a ringtone app. It has the capability to access to the phone's contact list (to set custom ringers), reads e-mail (to sound an alarm when it is received) and reads incoming call and SMS information. The application contains code from the same advertisement company MalloryAds[1]. This code's official purpose is to display a banner on the applications' UI. MalloryAds' code is as privileged as the applications containing it, but cannot be trusted. Contact information should never leave the device.

1. SafeRing sometimes prompts the user for feedback, and wants to send this feedback to its developer. The user fills a feedback form, which has a constant predefined format. Recall that if SafeRing has access to the internet, MalloryAds can send your contacts outside, but again, contact information should never leave the device. (Hint: use an additional "helper" application.)

SmartMail is an e-mail responder that answers e-mails automatically (according to predefined rules, such as "I am on vacation…" to anyone with a company email address). SmartMail has the capability to send and receive e-mails. If SmartMail has responded to an e-mail, it wants to send SafeRing a special message about it, so that SafeRing will stop ringing. SmartMail also uses MalloryAds.

2. Explain how the developer of MalloryAds cause your contacts to be sent to her by e-mail, if the above is implemented in the straightforward way using inter-process communication between SafeRing and SmartMail.
3. How can you prevent this?

The developer of SmartMail wants to protect the set of rules defined by the user (determining when and how to automatically respond to e-mails) from being exposed[2]. For this purpose, he decided to split SmartMail into two parts: SmartMailBack and SmartMailFront. SmartMailBack contains the rules and responds to emails. SmartMailFront merely presents the user interface, which is a screen reporting the log of replied messages, and an advertisement by MalloryAds. Thus, SmartMailBack can be trusted, but SmartMailFront still contains MalloryAds. Note that in order for SmartMailFront to keep the log, SmartMailBack sends messages to SmartMailFront according to the sensitive rules.
A third application is installed – EmailLogger. This application backs up all of the user's incoming e-mails on an on-line server (has access to incoming e-mails and internet), and also contains MalloryAds.

4. Explain how the developer of MalloryAds can learn some information about the rules.
5. How can you prevent the above attack, using Distributed Information Flow Control (i.e., information flow control with the ability to define new tags)?

[1] As we have discussed in class, many free applications run code from advertisement companies who pay the developer per-click/per-download/etc. For simplicity, here we assume that MalloryAds does not itself require Internet access for displaying ads.

[2] Of course the set of rules is exposed when acting upon these rules. We wish to allow only this form of exposure (and nothing else). I.e., an automatic response should be indistinguishable from a manual one (except for the fast reply).

In items 1, 3 and 5, describe a possible solution using the tools we have learned in class. You may use the existing Android access control system, or suggest extensions. In your answer, address the following:

a) Describe the system needed to support these principles – what security mechanisms should it offer?
b) Describe how the applications will use these security mechanisms.
c) Explain why the desired security property is obtained.

Question2

A common way to encrypt data is using a *stream cipher.* A stream ciphers get a secret key as input, and generate 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.[3] The following code describes the RC4 stream generation algorithm:

```
// S is a 256-byte table initialized to a secret value
i := 0
j := 0
while GeneratingOutput:
        i := (i + 1) mod 256
        j := (j + S[i]) mod 256  // reads S[i]
        swap values of S[i] and S[j]  // reads S[j], then S[i]
        K := S[(S[i] + S[j]) mod 256] // reads S[i],then S[j], finally reads S[(S[i] + S[j]) mod 256]
        output K
endwhile
```

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: You can monitor the RC4 stream generation and observe every memory read access, learning the precise address accessed.

In this exercise we will assume knowledge of some part of the plaintext and indexes output by the attack, and gain the ability to find the rest of the plaintext.

"ciphertextOrd.txt" contains the ordinal values of about 23,000 ciphertext characters.[4]

"plaintext.txt" contains the first 10,000 characters of the corresponding plaintext.

[3] Actually, in the full RC4 cipher, the secret initial state of the table is generated from the *real* secret key using another algorithm; we disregard this detail here.

[4] the "ordinal value" refers to the ASCII table index. E.g. ordinal value of 'a' is 97, '#' is 35 etc. The C binary representation of the ordinal value of a character equals the binary representation of the character itself. Conversion in python is straightforward: ch(<ordinal>), and ord(<character>).

"indexes.txt" contains the output of the cache attack, i.e., the memory address of every read access to the RC4 internal state table S. Each line contains one address. Addresses are given in decimal. The location of S in memory is not known a priory, but you may assume that consecutive entries in S have consecutive addresses. Assume read accesses of the cipher are ordered as the red comments in the pseudo code indicate. For your convenience, attached is the script with which the attack was simulated – "rc4outputIndexes.py".

Given these, you would like to find the full state of the table S after 10,000 iterations. Knowing this state lets you compute the rest of stream by yourself, and thus decrypt the rest of the ciphertext.

1. Write a program which reads the above 3 files and outputs the file "secret.txt" - the full state of S after 10,000 iterations. Write into this file one value per line, ordered from index 0 to 255 (note that the order of the values matters). Values should be written in decimal (e.g., if S[5] contains the value 200, line 5 of the file should contain the three characters "200".).
   "plaintext.txt" only contains the beginning of the plaintext (first 10000 characters). Attached for your convenience, is a simple python program "rc4dec.py" which outputs the rest of the decrypted plaintext, given that the "secret.txt" file in its working directory. Use it to check your "secret.txt" correctness. To run this program with Python installed, put "rc4dec.py" in the same folder as "ciphertextOrd.txt" and "secret.txt", and then run the following commands:
   a. cd <folder in which the files are in>
   b. python rc4dec.py

   Your program should work for any 3 files as described above, found in its working directory and having the same names as the 3 ones above. Note that rc4dec.py will work only for the files which were attached to the exercise.
   You can use any programming language you wish. It is recommended to use Python. Submit your source code, a compiled version of it (if written in a compiled language like C) and your resulting "secret.txt". Write clear, quality code.
2. In reality, cache attacks don't disclose the full address of memory accesses, but just some bits of the address. Suppose that the cache line size is 4 bytes, so that you can't distinguish between accesses to the 4 table entries in each cache line, and thus learn only the 30 most significant bit of each read address (out of the 32 address bits). Suggest a heuristic approach for carrying out the above attack in this case. It suffices to describe it at a high level; no need for an implementation.

Good Luck!