-Anand Mudgerikar

Introduction:

In many real-world applications, sensitive information must be kept in log files on an untrusted machine. In the event that an attacker captures this machine, we would like to guarantee that he/she will gain little or no information from the log files and to limit his/her ability to corrupt the log files. To address this issue, a protocol – named Secure Audit Log protocol – has been presented in the paper “Secure Audit Logs to Support Computer Forensics”, by Bruce Schneier and John Kelsey. The paper is available at http://www.schneier.com/paper-auditlogs.html.
This project requires you to implement a simulation of the secure audit log protocol presented in Section 3 of the paper. As part of the project, you will use basic cryptographic algorithms such as public-key encryption (RSA), symmetric encryption (AES), and message authentication code (HMAC).

Cryptographic Protocols used:

1. RSA 2048 and RSA 512

The initial key exchange between the trusted party and the untrusted party is done using RSA 2048 for encryption. I use another RSA key pair(512) for authentication. Self signed X509 certificates generated using these RSA keys are used for signing.

2. AES 256

We use AES 256 for encryption of data in the protocol.

3. SHA 256

For hashing, we have used SHA256 protocol.

4. HMAC

We use HMAC for verifying the hash values.

5. Envelope Functions:

All the communication between the machines is done using EVP functions of Openssl. We use RSA-2048 for key transfer and 256-bit AES for communication in the EVP functions.

S**ome Design Decisions:**

1. RSA Key Generation: The RSA keys are generated each time the protocol is run, so it is not required to explicitely call the genkeys.sh script. But if you require changes in the keys and certificates during the operation of the protocol, you can call the genkeys.sh script.

2. Use of AES-256: Even though it is a general consensus that AES-128 is better than AES-256 in terms of security. I do not quite agree because the issue with the AES 256-bit key schedule only opens up the possibility of a related key attack. Related key attacks depend on things being encrypted with keys that are related to each other in specific ways. When cryptographic systems are properly designed like our system is, related key attacks should not be relevant because good crypto systems shouldn’t use or create related keys.

Also using AES-256 in our application helps us as we use a SHA256 hash as the key for AES. If we use AES-128 we would have to use only half the bits of the hash. Even though this technique(Hash Truncation) is approved by NIST, doing does incease the chance of collisions. In any case, both methods are practically secure and I had a personal preference of using AES256.

3. Simulation of Trusted, Untrusted and Verifier Machines

In my application you require to login into different machines before executing any of the commands. This basically disallows different machines from running commands they are not entitled to use.
* Passing a string to the input asking for integer in the first prompt causes the program to go in an infinite loop. I have not figured out why that happens even when I sanitized the output.

Trusted Party can run the verifylog command.
Untrusted Party can run append,newlog and closelog commands.
Verifier can run verify command.

All the machines can run the exit command.

4. Log File Types: In this application only two types of logs besides the control messages are maintained:

“00005”	- accessible to verifier
“00006”	- not accessible to verifier
“00001”	- Initiate new log type
“00002”	- Message received from Trusted party
“00003”	- Successful Verification
“00004”	- Unsuccessful verification or timeout

This feature which allows verifier access or not has been written in code and if you wish to use it then you need to modify the set flag_v bit in code. This feature is turned off by default as our append command only gives a message as input and we have only a single verifier.

5. Verifylog command

The verifylog command does not print the control messages present in the log.

6. Encrypted Log Separation: Each of the logs in the log file are separated by a string “LOGENTRYSEPERATOR\n”.

7. Log Entry numbers are mainained from 1 and not 0. So, the actual log messages start from 3 with log entries 1 and 2 are control messages.

8. I have assumed that there exists a secure channel between the verifier and trusted party. It has not been implemented in this application.

Usage:

```
cd Debug
sudo make clean
sudo make all
./logger
```