CS255: Introduction to Cryptography **Winter 2009**

Programming Project #1

Milestone #1 Due Date: Friday, January 21st, 2009.
Milestone #2 Due Date: Wednesday, February 11th, 2009.

1 Overview

For the first programming assignment you will be implementing a password manager, similar (but inferior in many ways) to your operating system's keychain software. The password manager must be able to operate securely either locally or over a network. It must be protected against both an adversary who takes over the network, and to a lesser degree against one who takes over the server.

The project is divided into two milestones in order to encourage you to start early and work out some of the more difficult parts before starting on your implementation. Significant parts of the implementation are provided to you, so that you can focus on a small number of essential aspects of the problem.

The first milestone requires you to design the secure protocol between the client and the server. Your deliverable is a short (e.g. one or two page) specification of the protocol you are planning to use in your implementation. The specification should be concise, clear, and sufficiently detailed. The deliverable includes both diagrams of the allowed message exchanges in the system, and textual description of how the protocol works, and why it is secure.

The second (and final) milestone requires you to employ your existing protocol design and complete the implementation of the password manager. Similar to your protocol specification, your implementation should be as concise and clear as possible, and you have to follow some more specific conventions outlined in this text in order to enable us to grade your implementation efficiently.

2 Features

Any password manager's essential function is a secure map from the names of resources to their passwords, protected by a highly secret *master password*. For simplicity, we will assume that these are all strings. In particular, there are no additional data such as user name, site URL or notes. The software is divided up into a client, which manages the user interface, and a remote server, which stores the database of passwords. For simplicity, our server only supports a single user.

- The user must be able to add, remove and change (resource name, password) pairs. These pairs should be stored on the server, and should persist across restarts of both the client and server.
- The user must be able to change the master password at any time, without re-encrypting all of his stored passwords.
- The client software should not need to save any state between sessions. That way, it can be used on multiple machines at the same time without synchronizing state between the clients.

3 Threat Model

The password manager should be secure against the following attackers:

- Attackers who connect to the server as password manager clients should only be able to mount an *online* attack against the user's master password. That is, they shouldn't be able to retrieve anything useful for an offline dictionary attack. For example, this means the server must not blindly send password-encrypted known text that can be analyzed by the attacker offline by trying different, plausible master passwords. On the other hand, your key exchange protocol could involve messages that contain password-encrypted *random bits*, since those bits can not be analyzed by the attacker without successfully completing the secure handshake. In other words, encryption of random bits with a relatively weak password-derived key still satisfies the perfect secrecy criterion mentioned in class.
- Active network attackers should not be able to read any of the user's passwords. Nor should they be able to tamper with either the client or the server: they should't be able to convince the client to accept the wrong password (even an old one for the same resource), or to convince the server to change or delete a password. Of course, a network attacker can deny service; you do not need to make any effort to prevent this.
- Attackers who break into the server by some external means should not be able to read any of the passwords stored there, though they might tamper with them or delete them.

You are *not* required to protect the secrecy of the user's resource names or operations; that is, an eavesdropper may be able to determine whether the user is adding a new password, modifying one, or looking one up, and for what site. Nor are you required to protect the *length* of the stored passwords.

4 Cryptographic Requirements

4.1 Counter Mode (CTR)

You should use AES encryption in counter mode to protect the secrecy of the stored passwords (and, if you're doing the extra credit, the resource names).

Counter mode encryption generates a pseudorandom sequence by encrypting successive values of a counter. Formally, encryption of a message $(m_0, m_1, \ldots, m_n)$ is $(IV, E(k, IV) \oplus m_0, E(k, IV+1) \oplus m_1, \ldots, E(k, IV+n) \oplus m_n)$. As in other modes of encryption, new IV should be chosen randomly each time. Unlike most other modes of operation, counter mode does not require padding: the ciphertext length can be truncated to the length of the actual message without losing information.

4.2 Integrity Check using MACs

You will need to protect the integrity of messages on the network in order to prevent an attacker from modifying them while in transit. To prevent replay attacks without storing persistent state on the client, you should use a different MAC key in each session. You will still need a unique nonce on each message, but because of the per-session MAC, they can be unique-per-session instead of globally unique.

5 Protocol Specification

The protocol specification for all message exchanges between the client and server is your first deliverable. One good example of a security protocol description is given in RFC 4763, available on the Internet. Note the protocol description in section 3.2, and figures 1, 2, and 3.

For an example of another, somewhat informal and less relevant for now security protocol specification, you can look at the TLS protocol page in Wikipedia:

http://en.wikipedia.org/wiki/Secure_Sockets_Layer

Note again the structure of the diagram, showing the different message exchanges between a client and a server, the payload of the messages, and the local processing that occurs on the client and server. Also note the specific, unambiguous format defined for each message.

Before starting on the protocol design, please proceed through the end of this text. The supplied code dictates some of the protocol format and semantics. You will save time by adhering to the choices that have already been made in the code provided, and focusing on the parts of the implementation that are actually missing.

6 Components

6.1 Big Picture

On the client, **EncryptedMap** provides to the **Client** class the interface to the password map, making sure to encrypt values before sending them to the server via **NetworkedMap** (and conversely, decrypt them after retrieval). **NetworkedMap** uses **SecureBlobIO** to communicate with the server.

On the server, **NetworkedMapServer** acts as glue between **SecureBlobIO** and **FileMap**.

Not all (but certainly some) traffic over the network needs to be encrypted, while all of it will likely have to be authenticated: this is relevant when you consider the network protocol design, and the SecureBlobIO implementation. Additionally, the structure of the provided code implies that password value encryption and decryption happens at the client only (in other words, **EncryptedMap** is only used on the client). This has implications both for what traffic you can afford not to encrypt, and for what information you need (and don't need) to store persistently at the server.

Also note that the steady-state protocol between the client and the server is largely defined (look at **NetworkMapServerThread**'s run() method). In essence, the important protocol piece you are defining is the *session setup*, during which the client and server agree on a key they are going to use while the connection lasts.

6.2 Map

The primary functionality of the password manager is a secure, persistent, networked map from strings to strings. This functionality is developed in layers: **FileMap** provides a persistent map; **NetworkedMap** (along with **NetworkedMapServer** and **NetworkedMapServerThread**) exports it over the network; **EncryptedMap** provides secrecy and authentication; and **StringMap** translates to and from Strings using the UTF-8 character set.

You don't need to implement any of these maps: the only security-related one is EncryptedMap, and it's fairly trivial.

The maps included in this package do not quite conform to the Java map specifications in that they treat **byte** arrays as immutable objects. It is important to realize that because arrays are actually mutable, two arrays with the same elements are not considered equal by the Java standard libraries. As a result, **FileMap** and the like behave differently from, say, a **HashMap<byte[],byte[]>**.

6.3 Aes and Hmac

Aes and Hmac provide convenience classes over the Java cryptographic library. Hmac wraps the system implementation of HMAC/SHA1, and Aes uses HMAC and the system implementation of AES to implement authenticated AES counter-mode encryption.

You need to implement the Aes class; the Hmac class is provided for you.

6.4 BlobIO

BlobIO handles input and output using arrays of bytes (*blobs*), and, for convenience, arrays of arrays of bytes. Its instance FileBlobIO implements atomic file operations using temporary files and renaming semanatics. Its instance IOBlobIO uses the standard input/output libraries to send over pipes and network sockets.

You need to implement the SecureBlobIO instance, which will provide a channel whose integrity is protected from network attackers.

6.5 Client

The Client class implements the password manager's GUI. The current client is fairly limited; for instance, it cannot connect to any server other than localhost. You're welcome to improve this class, but it's not really the point of the project.

6.6 NetworkedMapServer

This class implements the network server, saving files in a directory called net_test. You don't need to modify it.

6.7 Test

The test class will conduct a simple series of tests over a virtualized network. It won't involve the GUI, and it can be built and run even if you don't have SWT installed.

7 Implementation

You will be using the JCE (Java Cryptographic Extensions) while programming for this assignment. You should spend some time getting familiar with the provided framework.

7.1 Getting the code

Download the *pp1.tar.gz* file linked on site to a directory in your account. Untar and unzip using the following command:

tar xvzf pp1.tar.gz

This should create the source tree for the project under the *pp1/* directory.

7.2 Description of the code

Here is a brief description of the files we provide. The files you need to change are in **bold**

Makefile	Makefile for the project
pwman/Aes.java	The implementation of AES modes
pwman/SecureBlobIO.java	Cryptographic network protocol
pwman/Hmac.java	Wrapper class around HMAC/SHA1
pwman/BlobIO.java	Binary Input/Output module
pwman/Client.java	GUI client
pwman/NetworkedMapServer.java	Main server
pwman/Test.java	Test harness
pwman/EncryptedMap.java	Encrypted implementation of binary map
pwman/NetworkedMap.java	Network map protocol

7.3 Running the code

To build the project, enter the *pp1* directory and type *make*. To test the system, type *make run-test*. To use the client and server, type *make all run-server &* followed by *make run-client*. The client will only compile and run on a machine which has the SWT graphics library installed; this can be obtained on Ubuntu by typing *sudo apt-get install libswt3.2-gtk-java*. To erase created class files along with cores, emacs temporary files and the test and net_test directories, type *make clean*.

Note: Your solution will be tested on the myth machines. So, please test your code on one of

the myths before submitting. Note that SWT may not be directly available on the cluster workstations, in which case the client GUI will neither compile nor run.

7.4 Crypto Libraries and Documentation

Java's security and cryptography classes are divided into two main packages: java.security.* and javax.crypto.*. They have been integrated into Java 2 Platform Standard Edition v 1.5. Classes for cryptographic hashing and digital signatures (not required for project 1) can be found in security, whereas ciphers and MACs are located in the JCE.

The following are some links to useful documentation :

- Java API
  http://java.sun.com/j2se/1.5.0/docs/api
- JCE Reference Guide
  http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html
- Java Tutorial
  http://java.sun.com/docs/books/tutorial/
- Chapter 6 from Java Cryptography by Jonathon Knudsen
  http://www.oreilly.com/catalog/javacrypt/chapter/ch06.html

Some classes/interfaces you may want to take a look at:

- javax.crypto.KeyGenerator
- javax.crypto.SecretKey
- javax.crypto.Mac
- javax.crypto.Cipher
- javax.crypto.SecretKeyFactory
- java.security.SecureRandom

8 Deliverables and Grading

8.1 Milestone #1

In your specification, you have to:

- describe (using high-level pseudocode) how each side (client, server) of the SecureBlobIO connection behaves upon connection establishment;
- describe (using struct-like statements and pseudocode) what each subsequent message exchange looks like (including on-the-wire format); what behavior is triggered at the recipient; and how the response eventually gets processed at the origin;

- include a diagram for the above items.

Your write-up will be graded as follows: correctness, security, and completeness (about 70%); conciseness and clarity (about 30%). This milestone is worth 5 points (meaning, 5% of your final grade for the course).

8.2 Milestone #2 (Final Submission)

In addition to your well-commented solution to the assignment, you should submit a README containing the names, leland usernames and SUIDs of the people in your group, a description of the design choices you made in implementing each of the required security features, and the protocol specification as submitted for the first milestone (with any corrections you have made during your implementation).

When you are ready to submit, make sure you are in your *pp1* directory and type:

make clean; /usr/class/cs255/bin/submit

Your submission will be graded as follows: correctness, security, and completeness of the implementation (about 70%); clean and well-documented code (about 30%). This milestone is worth 10 points (meaning, 10% of your final grade for the course). Note that an insecure protocol from the first milestone, used as-is for the second milestone will automatically yield an insecure implementation which can not receive maximum credit (this is why you should analyze and fix any problems from your milestone #1 write-up for your milestone #2 work).

9 Miscellaneous

- We strongly encourage you to use the class newsgroup (su.class.cs255) as your first line of defense for the programming projects. TAs will be monitoring the newsgroup daily and, who knows, maybe someone else has already answered your question.
- You can also email the staff at cs255ta@cs.stanford.edu