The University of North Carolina at Greensboro — Handout 3
CSC 495/680: Trusted Computing — October 4, 2010
Prof. Stephen R. Tate

CSC 495/680 Assignment 2

Due Wednesday, October 20

Objective: The purpose of this assignment is to get students some experience with programming using the Trusted Software Stack (TSS), and to experiment with keys and basic bind/unbind functionality.

System and Compilation/Linking Information: There are several sample programs on the SPAN systems in `/scratch/csc495` that contain useful bits of code that you can use in your assignment — be sure to read through that code and understand what is there before you start coding your programs for this assignment. There are also a lot of good examples of code in Chapters 7 and 8 of the book. For detailed documentation on specific TSS functions, refer either to the man pages on the SPAN lab systems or to the TSS specification (link on the class web page).

The standard TSS library in Linux is called "TrouSerS" and to write a program using this library, you must make sure you have the appropriate includes and library designations. For include files, you should always include `<tss/tspi.h>` and `<trousers/trousers.h>` — note that only the first of these is necessary for standard TSS programs, but the second one allows you to use the TrouSerS error code to printable string functions, which makes error messages much more understandable. For library linking, be sure to include "`-ltspi`" when you compile. For example, to compile the sample program `newkey-uuid.c` first copy it to a directory you can work in, such as your home directory, and then use the command

```
gcc -Wall -o newkey-uuid newkey-uuid.c -ltspi
```

Background Trusted Computing Information: There are two main ways to manage keys for use by the TPM: One way is to keep keys in the TSS-managed key ring (or "persistent storage"), registered by a 128-bit UUID, and then any program can load and use the keys by using this UUID. The second way is to store keys in files that are managed outside of the TSS library.

The TSS specification describes a standard key hierarchy — the pages of the specification describing this are distributed with this assignment in class, and if you're reading this electronically you can find this in Section 3.22 (KeyManagement) of the TSS specification (pages 143–145 of the "Errata A" version available on the class web page). The `labhost` systems in the SPAN Lab have the required "System Persistent" keys established on each system, although the optional "PK" key is not used (so "SK" and "RK" are directly below the SRK).

The UUIDs for these keys are defined in TrouSerS as initializers for a UUID object, and have the names "`TSS_UUID_SK`" and "`TSS_UUID_RK`". Note that the `labhost` systems require authentication on the SRK, but it is authentication with the "well-known secret" — you can see how this is handled in the `newkey-uuid.c` example.[1] The SK and RK are different — those keys do not require authorization at all.

The `newkey-uuid.c` example creates a non-migratable bind key underneath the SK, and registers the key using a random UUID that is hard-coded in the source code. Make sure you read and understand that example before moving on with the rest of this assignment.

Exercises: This section describes the programs you should write as part of this assignment. After you're sure that your program is running correctly, time how long it takes using the "`time`" command (e.g., to time the `newkey-uuid` program you'd type the command `time ./newkey-uuid` — it's the "real" time that you want to record). You'll need to use these times to answer the questions below. When you turn in your assignment, turn in a printout of the sourcecode, and in addition email all three programs to me (one email with three attachments, please!).

Program 1: The first program you should write for this assignment is to create a "bind" key and store it in a file. The section on "Key Objects" in Chapter 7 of the book describes how to extract a "key blob" from a key object, and this key blob can then be written out to a file. Your program can either get the name of the output file from the command line or can prompt the user to enter it, but it should not be hard-coded in your program (in other words, you should be able to run your program twice to create two keys stored in different files). The key you create should be a 2048-bit, non-migratable, bind key, with SK as its parent. No keys should require authorization for use.

Program 2: Next, you should write a program that uses TPM-generated keys to encrypt a small amount of data using a "bind" key. Your program should be able to handle using either the bind key generated by the sample `newkey-uuid.c` program (and registered in user persistent storage) or the key generated by your first program (and stored in a file). You can determine which key to use either through a command line switch or by prompting the user. The input data (i.e., the data you want to encrypt) should be read from a file, and the output ciphertext should be written to a file. As with other parameters, you are free to have your program get the filenames either from the command line or by prompting the user. The Chapter 7 section "Encrypted Data Objects" gives the basic information on how to perform the bind operation.

Program 3: The third program is the unbind (unencrypt) partner to Program 2 — it should use a bind key to decrypt ciphertext produced by the previous program. The structure is the same

[1] Note that this isn't actually standard, as the TSS spec says the SRK should not require authorization. However, it isn't quite as simple as that either — a TPM in "FIPS mode" requires authentication of the SRK, and the sample storage hierarchies described in Chapter 3 of the book talk about the SRK using the "well-known secret" for authorization. The SRK configuration in the lab follows the ideas from Chapter 3 of the book rather than the TSS specification, since that can be use in either FIPS mode or non-FIPS mode.

as the encryption program: you should read the ciphertext from a file and write the plaintext out to another file. The book does not have an example of this, but you should be able to figure this out from the techniques you used in the previous program. In particular, you will need to set the encrypted data blob attribute, and then call the Unbind function. Remember that you can find the exact arguments in the man pages ("`man Tspi_Data_Unbind`").

Final Activity: Once all three programs are written, compiled, and debugged, you should have all of the code (source and binary) in your home directory (or a subdirectory), along with some sample data files (original plaintext, ciphertext, and recovered plaintext). Now log in to a different SPAN `labhost` machine. Use the `showkeys` program from the samples directory to see what keys are registered on this machine. Try to use your "unbind" program to decrypt your ciphertext file on the new machine. You'll be asked to describe and interpret the results below in the questions.

Questions: Answer the following questions.

1. When using keys that are stored in the two ways explored here (by UUID in persistent storage vs. in an external file) the most obvious difference is in the use of the function `Tspi_Context_LoadKeyByUUID` versus `Tspi_Context_LoadKeyByBlob`. What other differences were there? Which was easier code to write?

2. One disadvantage of using UUIDs is that 128-binary values are not easy to remember or to specify from a user's standpoint. If the UUID hadn't been hardcoded into the sourcecode, you'd need some way for a user to specify the UUID of the key — clearly this is a challenge! Can you think of a solution that provides a "user-friendly" way of referring to keys, and yet uses the nice key management abilities of the TSS persistent storage?

3. Run unbind twice in a row on the same ciphertext, and measure the time of both runs. Was there any difference? Was this what you expected? After your tests, run `showkeys` from the sample programs, which lists all of the keys currently in persistent storage. What does the "isLoaded" column show? Can you describe any efficiency improvements to the way TrouSerS implements the functions that you're using?

4. Describe what happened (and why) when you tried to decrypt the ciphertext on another system (the "Final Activity" above)?

5. The "Roaming Key" (RK) is potentially promising for such situations: Keys below RK are migratable, and so could potentially be used on multiple systems. Here is one solution (call this "Solution 1"): a migratable key could be created under RK on one host, and then the migration process could be performed for each other host so that the key had a distinct re-wrapping underneath each system's RK. Therefore, each system would have its own version of the key that it was able to load. For "Solution 2" I give only a hint:

What if the RK itself were migrated among systems within an organization so that it was the same key on each system (although wrapped differently)? Carefully describe how such a solution would work — specifically describe how three things work: (1) how does the organization initialize a new system, (2) how does a user create a new key, and (3) how does a user use one of their keys. Be sure to address issues such as how keys are referred to (what UUID), how parent keys are located on each system, etc.

Finally, consider a case in which there are 100 systems on a corporate intranet, with 70 distinct users, each with 10 keys. How many total keys must be managed in "Solution 1"? What about in "Solution 2"?

6. One weakness of "Solution 2" in the previous question is that migration of RK must be very tightly controlled. What is the risk if an untrustworthy person has access to and can control the migration process for RK?

7. Consider a case in which I'm a user in an organization that has adopted "Solution 2" from above, but I don't trust the system administrator who controls the "RK" roaming key. Devise a solution (called "Solution 3") that the user controls, but which has many of the same properties as "Solution 2" (most importantly that the same key blob can be used on multiple systems). Using the numbers from above, if each of the 70 users manages their keys by this new solution, how many total keys are there now? What can or can't a dishonest system administrator do in this situation?

8. The Bind operation uses only the public part of the bind key that was generated by Program 1. Since this is public, it should be possible to do the Bind operation on any system — not just the system that generated the key. Test this out: try to run your bind program on a different host. Does it work? If it doesn't work, can you think of a way to make it work? (See also the optional exercises below)

Optional Exercises: The following exercises are optional, but will give you more experience programming for the TSS. They also can provide a small amount of extra credit (5 points each).

1. Adapt your "bind" program so that it works on any system (with bind keys that are stored in files).

2. Add support for attaching usage authorization to the keys you create (and then, of course, you'll need support in your other programs so that the authorization can be provided when the key is used).