CprE 308 Laboratory 12: Operating System Security

Department of Electrical and Computer Engineering
Iowa State University

Spring 2016

NOTE TO STUDENTS: If you do not show your lab report to your lab TA before you leave lab, you will not receive credit

1 Submission

Include the following in your lab report:

- Your full name and lab section in the upper right corner of the first page in large print.
- **(11pts)** A cohesive summary of what you learned through the various experiments performed in this lab. This should be no more than two paragraphs.
- A write-up of each experiment in the lab. Each experiment has a list of items you need to include. For output, cut and paste results from your terminal and summarize when necessary.

2 Resources

Read the manual pages about SSH and GnuPG by typing the following command in your terminal.

```
man ssh
man scp
man ssh-keygen
man ssh-agent
man gpg
```

3 Purpose

In this lab, you will become familiar with the use of ssh (secure shell) and gpg (GNU privacy guard). You will need a partner to finish some sections of this lab.

3.1 SSH

3.1.1 Definitions

SSH, short for Secure Shell, is a protocol used to authenticate, encrypt, and digest data transmitted over a network. SSH system is composed of two parts: the server and the client. The SSH server handles the incoming request from the SSH client. The SSH client sends the connection request to the server. SSH creates a secure channel between two computers. Any data transferred on the channel is encrypted and authenticated.

3.1.2 Secure remote logins

Suppose you are out of town on an interview in California. But you forget to bring the CprE 308 project documents which you want to show to the interviewer. You put the project documents on a machine which is a SSH server. You would like to retrieve these documents without using telnet because it is insecure. We will look at two scenarios.

Scenario I: Open Shell and so it is prompting you to enter a command.

2pts: Run the command: `ssh linux-1.ece.iastate.edu`. Copy the command line and output into the lab report. Run "ls" to check the output information. Which machine is the output information coming from?

Scenario II: Suppose your login name on the client machine does not match the login id on the server, the above will not work. This can happen for example, if you are using your partner's machine. There are two ways you can login from your partner's machine:

2pts: From your partner's machine, use the "-l" option with your NetID to ssh into `linux-1.ece.iastate.edu`. Copy your command line and output into the lab report.

2pts: Run the command `ssh <NetID>@linux-1.ece.iastate.edu` to log into `linux-1.ece.iastate.edu` with your partner's machine.

In either case, SSH establishes a secure channel between your machine (the client) and the server so that all transmissions between them are encrypted.

3.1.3 Secure file transfer

Suppose that while browsing your files you encounter a file you would like to print. Another ssh-like client program, `scp`, is used to copy the file across the network via a secure channel.

3pts: Log out of `linux-1.ece.iastate.edu` and return to the original shell. Fetch a file from `linux-1.ece.iastate.e` to your own machine using `scp`. Copy your command into the lab report.

3pts: Now use your partner's machine to fetch a file from `linux-1.ece.iastate.edu` to your partner's machine using `scp`. Copy your command into lab report. Hint: text after the symbol should indicate your NetID instead of using your partner's NetID.

Note that the destination filename need not be the same as the remote one. Please use `man scp` for details.

In the above procedure, we log out the ssh session then we use scp to fetch a file. But what if we do not want to log out? SSH supports an escape character. By default, the escape character is the tilde (~). The escape character must be the first character on the command line, i.e., following a newline (`Control-J`) or return (`Control-M`) character. If not, the client treats it literally, not as an escape character.

6pts: Log into `linux-1` using the command '`ssh linux-1.ece.iastate.edu`'. Next type the escape character '' followed by `CTRL+Z` to suspend the connection. Now use `scp` to fetch a file from `linux-1`. Then type '`fg`' to return to the SSH session. Copy the three commands you used into lab report.

3.1.4 Known Hosts

Two people who do not know each other generally will not trust each other. The same is true for two machines. When an SSH client connects to a server for the first time, it will realize that it has never connected to that machine before. Here is the question: how do you know after running command `ssh linux-1.ece.iastate.edu` successfully, the machine you logged in is really `linux-1.ece.iastate.edu`?

Each machine has a unique identity. The identity is a host key pair. One is the public key; the other is the private key. The public key is used by the machine to identify itself to other machines. When a client connects to a server, both machines will send their public key to each other. Likewise, the question still exists: how does the client know that the machine which sends the public key is the one it wants to connect? It is a good idea to record the server's public key before connecting to it. Here comes a role for the administrator. He can maintain a known-host list for all the machines in the system. But what if the system is too big, say all the machines in the world? Some protocols such as secure DNS may be the answer. In this lab, we assume that we can trust the public key which the machine sends for the first time, i.e. we assume that a man-in-the-middle attack is not possible the first time we connect to a server.

The first time a client connects to the server, the public key sent by the server is stored in your local account on the machine you used. After that, SSH will check the stored public key with the one sent by the server. Meanwhile, the server will also store the public key of the client. In the future, SSH will check the public key stored with the one sent by the client.

6pts: Under your home directory there is a directory called "`.ssh`". In this directory, you will find a file named known_hosts. Open the file and you will find that it records several machines' public keys. Copy the name of a machine and its IP address into lab report.

3.1.5 Cryptographic keys

As we stated above, each server and client has a key pair which is called a public/private key pair. The public key is used to identify the server or client itself. The private key is used to decode encrypted information or produce a digital signature. To use cryptographic authentication, you must first generate a key pair for yourself, consisting of a private key and a public key. To do this, use the `ssh-keygen` program. It requires you to specify `-t` option to produce a DSA or RSA key. Use `ssh-keygen` to generate an DSA key pair.

4pts: Under the .ssh directory, which file contains your private key? Which file contains your public key?

4pts: When you create the key pair, you are required to input the passphrase to encrypt a file. What is the difference between passphrase and password? What if you forget your passphrase?

4pts: Which file is the passphrase used to encrypt? Why isn't the other one encrypted with the passphrase? Use command `ls -l` to verify your answer.

3.1.6 Host authentication and User authentication

This part may seem confusing at first. Client host and server host have their own public keys and private keys. You generated another public/private key pairs by `ssh-keygen`. Is that necessary?

Until now there is only host authentication in your operation. The public and private keys you used are the keys of the host. These keys are generated when the SSH server is first started. The program `ssh-keygen` is used to generate the user key pairs which are used in user authentication. In the command `ssh linux-1.ece.iastate.edu`, the user authentication is accomplished by password authentication.

3.1.7 Installing a user public key on an SSH Server Machine

When passwords are used for authentication, the host operating system maintains the association between the username and the password. For user keys, you must set up a similar association manually. After creating the user key pair on the local host, you must install your public key in your account on the remote host. A remote account may have many public keys installed for accessing it in various ways.

The user public key must be placed in the file `~/.ssh/authorized_keys`. A typical authorized_keys file contains a list of public-key data, one key per line.

6pts: In the report, include all commands you used to copy the public key into the file authorized_keys in server `linux-1.ece.iastate.edu`.

Run the command `ssh linux-1.ece.iastate.edu` again. Do you need to input a password this time? No. You need only input the passphrase. On the surface, the only difference is that you provide the passphrase to your private key, instead of providing your login password. Underneath, however, something quite different is happening. In password authentication, the password is transmitted to the remote host.

5pts: In cryptographic authentication, what is the passphrase used for? Is it sent to the `linux-1.ece.iastate.edu`? Describe the whole authentication procedure briefly.

3.1.8 The SSH agent

Every time you use ssh to log in to a server, you have to retype the passphrase. That is a terrible thing. Life should be simpler. Wouldn't you like the machine to remember your passphrase? Whenever you come, it knows you and lets you in directly. The SSH agent will achieve this. It will render your key to the SSH client on your behalf.

The SSH agent is a process that remembers your private keys in memory and provides authentication services to SSH clients. When the agent is running, the SSH client process will interact with the agent directly instead of you. The agent program for OpenSSH is ssh-agent. The agent will run until it is killed, which happens for example, when you log out.

To run the agent, enter: `ssh-agent $SHELL`, where SHELL is the environment variable containing the name of your login shell. Once the agent is running, it is time to load the private key into it using the `ssh-add` program.

3pts: Copy all the commands you used into the report.

3pts: Now login to `linux-1.ece.iastate.edu` again. Copy the command and the output into the report.

3.2 GnuPG

In this part, we will learn the GNU Privacy Guard (GnuPG or GPG) which is widely used to encrypt, decrypt, sign and check messages on the Internet.

3.2.1 Generate a key pair

The first step to use GnuPG is to generate a key pair. For example:

```
$gpg --gen-key
gpg (GnuPG) 1.4.5; Copyright (C) 2006 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
```

```
under certain conditions. See the file COPYING for details.

Please select what kind of key you want:
    (1) DSA and ElGamal (default)
    (2) DSA (sign only)
    (5) RSA (sign only)
Your selection?
```

There are three kinds of options. Option 1 creates two key pairs. The DSA key pair is the primary key pair used only for making signatures. Option 2 only creates a DSA key pair. Option 5 only creates a RSA key pair. For us the default option is fine.

```
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

You must also choose a key size. The longer the key the more secure it is. For most applications, the default key size is adequate. However, this will change with time. For example, the size of 1024 is enough today. But it is possible that 2048 will be required 10 years from now. But the smaller size of the key, the higher speed for encryption and decryption is. A larger key size may affect signature length. Once these keys are selected, the key size can never be changed.

Finally, you must choose an expiration date. If Option 1 was chosen, the expiration date will be used for both the ElGamal and DSA keypairs.

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

For most users a key that does not expire is adequate. The expiration time should be chosen with care, however, since although it is possible to change the expiration date after the key is created, it may be difficult to communicate a change to users who have your public key. But by changing keys, it will be more difficult to attack your key. For example, it will take the hacker 1 month to decrypt your key according to the speed of computer nowadays. For this reason you may choose to change you key each week.

```
You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name:
```

You must provide a user ID in addition to the key parameters. The user ID is used to associate the key being created with a real person. After this, you are also required to input the email-address, comments and passphrase.

6pts: Use the GPG command-line option `--gen-key` to create a new key pair using the default selection. Copy the command and your selection into the lab report.

3.2.2 Exchanging keys

Listing keys: To communicate with others you must exchange public keys. To list the keys on your public key ring use the command-line option `--list-keys`.

```
$ gpg --list-keys
```

Exporting a public key: To send your public key to a correspondent you must first export it. The command-line option `--export` is used to do this. It takes an additional argument identifying the public key to export. The key is exported in a binary format.

```
$ gpg --output myname.gpg --export your_user_id
```

It can be inconvenient when the binary key is to be sent through email or published on a web page. GnuPG therefore supports a command-line option `--armor` that causes output to be generated in an ASCII-armored format similar to unencoded documents. In general, any output from GnuPG, e.g., keys, encrypted documents, and signatures, can be ASCIIarmored by adding the `--armor` option.

Importing a public key: Your partner's public key may be added to your public keyring with the `--import` option..

```
$ gpg --import your_partner.gpg
```

6pts: Exchange the public key with your partner. Include the command you used and the output of the `--list-keys`(after you have your partner's public key) into the report.(Hint: you can use `scp` or email to distribute your public key)

3.2.3 Encrypting and decrypting documents

To encrypt a document the option `--encrypt` is used. You must have the public keys of the intended recipients. The software expects the name of the document to encrypt as input; if omitted, it reads standard input. The encrypted result is placed on standard output or as specified using the option `--output`.

```
$ gpg --output doc.gpg --encrypt --recipient partner's_id doc.txt
```

The `-recipient` option is used once for each recipient and takes an extra argument specifying the public key to which the document should be encrypted.

To decrypt a message the option `--decrypt` is used. You need the private key to which the message was encrypted. Similar to the encryption process, the document to decrypt is input, and the decrypted result is output.

```
$ gpg --output doc.txt --decrypt doc.gpg
```

3pts: Encrypt your public key file and send it to your partner. Copy your command into the report.

3pts: Decrypt the public key file that your partner has sent to you. Copy your command into the report.

3.2.4 Making and verifying signatures

The command-line option `--sign` is used to make a digital signature. The document to sign is input, and the signed document is output.

```
$ gpg --output doc.gpg --sign doc.txt
```

Given a signed document, you can either check the signature or check the signature and recover the original document. To check the signature use the `--verify` option. To verify the signature and extract the document use the `--decrypt` option. The signed document to verify and recover is input and the recovered document is output.

```
$ gpg --output doc.txt --decrypt doc.gpg
```

A signed document has limited usefulness. Other users must recover the original document from the signed version, and even with clear-signed documents, the signed document must be edited to recover the original. Therefore, there is a third method for signing a document that creates a detached signature, which is a separate file. A detached signature is created using the `--detach-sig` option.

```
$ gpg --output doc.gpg --detach-sig doc.txt
```

3pts: Sign your public key file and send the signed file to your partner. Copy your command into the report.

3pts: Verify the signature of the public key file sent by your partner. Copy your command into the report.

3.3 Shellshock (Bash bug)

By some estimates the Shellshock bug has gone undiscovered for nearly 25 years. There are some good explanations of the bug online.

Shellshock Code and the Bash Bug - Computerphile: https://www.youtube.com/watch?v=MyldPMn95kk

3pts: Describe the Shellshock bug and how it works. The impact of the Shellshock bug was originally under estimated. The impact of the bug became much more clear as many researchers realized that several programs (including popular web servers such as Apache) make heavy use of environment variables.

Example: http://www.securitysift.com/shellshock-targeting-non-cgi-php/

3pts: What is an environment variable and how could it be used in conjunction with the Shellshock bug to remotely exploit a web server?

3.4 Buffer Overflows

The basic buffer overflow example we have looked at in class is shown below.

```
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[64];
  strcpy(buf, argv[1]);
}
```

Remember that it is vulnerable because:

- The program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Year after year, buffer overflows have ranked among the worst offenders when it comes to serious security issues. Programmers are slowly learning to code more securely and there are many tools that can analyze your code and point out security issues during development. With all the modern technology trying to eliminate buffer overflows, you might be wondering how they keep working their way into our code!

Take a look at the infamous Sendmail Crackaddr bug below. This code also contains a buffer overflow. Can you spot it? If you need help check out the extra resources below.

```
# define BUFFERSIZE 200
# define TRUE 1
# define FALSE 0
int copy_it ( char * input , unsigned int length ) {
  char c, localbuf [ BUFFERSIZE ];
  unsigned int upperlimit = BUFFERSIZE - 10;
  unsigned int quotation = roundquote = FALSE ;
  unsigned int inputIndex = outputIndex = 0;
  while ( inputIndex < length ) {
    c = input [ inputIndex ++];
    if ((c == '<') && (! quotation )) {
      quotation = TRUE ;
      upperlimit --;
    }
    if ((c == '>') && ( quotation )) {
      quotation = FALSE ;
      upperlimit ++;
    }
    if ((c == '(') && (! quotation ) && ! roundquote ) {
      roundquote = TRUE ;
      // upperlimit --; // decrementation was missing in bug
    }
    if ((c == ')') && (! quotation ) && roundquote ) {
      roundquote = FALSE ;
      upperlimit ++;
    }
    // if there is sufficient space in the buffer , write the character
    if ( outputIndex < upperlimit ) {
      localbuf [ outputIndex ] = c;
      outputIndex ++;
    }
  }
  if ( roundquote ) {
    localbuf [ outputIndex ] = ')';
    outputIndex ++;
  }
```

```
  if ( quotation ) {
    localbuf [ outputIndex ] = '>';
    outputIndex ++;
  }
}
```

The bug was discovered in 2003 by Mark Dowd. Later Thomas Dullien extracted a smaller toy example shown above. The original flaw was contained in about 500 lines of code and the toy example is about 50 lines of code. Finding the bug in this code is difficult enough with a single loop, but the original bug contains about 10 loops (with nested loops of depth of 4), GOTOs, lots of pointers, pointer arithmetic, and calls to string functions. Most program analysis tools a good at finding surface level bugs like the examples in class, but this tool is nested much "deeper" in the code. Finding these types of bugs is an ongoing research project. Resource: Sendmail Crackaddr Talk Resource: A Java version of the same bug (throws an Index out of Bounds Exception)

6pts: Explain how an attacker can trigger the buffer overflow in the Sendmail Crackaddr Toy example.

NOTE TO STUDENTS: If you do not show your lab report to your lab TA before you leave lab, you will not receive credit