432 Information Security
Homework 2

Ed Felten

September 15, 2008

Contact Bill Zeller with comments or questions at wzeller@princeton.edu

1 Setup

You will be using the same setup that you used for Assignment 1. You should download and use the new version of our code, as some parts of it have changed. You should not be reusing files from Assignment 1. However, you may replace the **MySSHShell.py** file we provide with the file you edited in Assignment 1.

2 Code Structure

- PythonSSH
  - Client (client specific code)
    - ClientCommandTransport.py
    - **MyClientCommandTransport.py**
    - MyClient.py (fully functional SSH client. You can run this directly)
  - Server (server specific code)
    - ServerCommandTransport.py
    - **MyServerCommandTransport.py**
    - MyServer.py (fully functional SSH server. You can run this directly)
    - SSHShell.py
    - MySSHShell.py (You can replace this file with your submission from assignment 1)
  - Fun (random code you can use when creating your shell)
  - DiffieHellmanFake.py (a fake implementation of DiffieHellman)
  - SSHUtil.py (Utility functions that facilitate conversion between Python data types and SSH data types)

  - **PacketUtil.py** (Utility functions that facilitate packet encryption and MACing)

(You will be submitting the files in **bold**)

3 The Assignment

The goal of this assignment is to implement part of the SSH Transport Layer[1]. Specifically, you will be focusing on the encryption of packets and the computation of hashes and MACs related to these packets.

An SSH session (at the transport level) consists of the following steps:

- A list of supported algorithms is exchanged. This ensures the client and server can communicate with each other.
- A key exchange takes place. Diffie-Hellman is used to provide each side with a secret key. In addition, an exchange hash is computed that is used to verify the key exchange and also as a session identifier. You will be using the classes we provide (DiffieHellmanServer and DiffieHellmanClient) to access the correct data to send and you will be computing the exchange hashes on both the client and server side.
- The secret key is used along with the exchange hash (now considered the session_id), to generate six keys that will be used during the actual communication between the client and server. These keys are the IV, the encryption key and the integrity key. (There are 6 because, e.g., there's one encryption key for the server and a different one for the client). These keys are generated for you. There is no requirement that the client and server use the same encryption or MAC algorithms, although in this case they do. (See 5.2 of RFC4253)
- Sending and receiving data requires use of the binary packet protocol (See 4 of RFC4253) used in the transport layer. This packet consists of the packet length, the padding length, the payload (the data), some random padding, and the value:
  `MAC(key, sequence_number || unencrypted_packet)`
  where *key* is the integrity key from the key negotiation step and *unencrypted_packet* is the entire unencrypted packet (except for the MAC). The rest of the packet (not the mac) is then encrypted and concatenated with the MAC to produce the packet that will be sent to the other party. You will be encrypting these packets (and computing the MAC) as well as decrypting and verifying the MAC.

We have provided a "fake" implementation of Diffie-Hellman in DiffieHellmanFake.py. This is an insecure implementation that always uses the same shared secret. Because of this, MyServer.py and MyClient.py only work together and are not currently interoperable with general purpose SSH clients or servers. You will be implementing Diffie-Hellman in a later assignment which will allow you to use your favorite SSH client to connect to the server you created.

---

[1] The SSH Transport Layer is defined in RFC4253

3.1 A note on SSH strings and integers

The SSH protocol uses a different format for integers and strings[2] than what is used by Python. This means when operating on strings and integers in SSH, you'll need to use utility functions to convert them. These functions are `SSHUtil.Int2SSHNum` and `SSHUtil.Str2SSHStr`. This document will specify `SSH_NUM` and `SSH_STR` when you need to use these functions, respectively.

3.2 What To Do

You will be implementing a variety of functions in a variety of different files. To make this easier, we've created unit tests for each file you'll be editing. For example, when editing `PacketUtil.py` you can run `python PacketUtil.py` directly, which will run the unit tests in the file. If the message "no errors" is printed, your code has passed the unit tests. This means you *do not* need to start the server and client each time you run your code. If your code passes all of the unit tests in all of the files, you should be able to connect your client to the server. The assignment is considered complete if you can log into your SSH server and enter commands in the shell.

- PacketUtil.py
  - CreateAESObject(key, keySize, iv, blocksize)

    Return a new PyCrypto AES[3] object. This object should be initialized with the first `keySize` bytes of `key`[4], the constant telling the object to use CBC mode (`AES.MODE_CBC`) and the first `blockSize` bytes of `iv`.
  - EncryptAndMACPacket(seqid, packetData, aesObj, macKey)

    Return an encrypted packet concatentated with the MACed packet (the digest).

    - `packetData` The data to be encrypted
    - `aesObj` The object you created in `CreateAESObject`
    - `macKey` The key to use in the MAC function.
    - You can use the utility function `PacketUtil.GetMacData` to compute the data to MAC from `seqid` and `packetData`.
    - You should use HMAC-SHA1[5]
  - DecryptPacket(data, aesObj)

    Return decrypted `data` using `aesObj` (created by `CreateAESObject`)
  - VerifyMAC(macData, seqid, packetData, macKey)

    Return true if MAC is correct, else false.

    - `macData` MAC value from received packet, to be verified
    - `seqid` The sequence id

---

[2] See Section 5 of RFC 4251 for details

[3] See `http://trevp.net/tlslite/docs/public/tlslite.utils.Cryptlib_AES.Cryptlib_AES-class.html`

[4] See *slice notation*: `http://docs.python.org/tut/node5.html`

[5] Python provides MAC functions `http://docs.python.org/lib/module-hashlib.html`

- `packetData` The packet data
- `macKey` The key to use in the MAC function
- Again, you can use the utility function `PacketUtil.GetMacData` to compute the data to MAC from `seqid` and `packetData`.

- MyClientCommandTransport
  - KeyExchangeInit(self, generator, prime)

    Initialize a DiffieHellmanClient object and then send a packet with data `e` (`SSH_NUM`) (from DiffieHellman) of type `self.SSH_MSG_KEXDH_INIT`

    You can send packets with `self.sendPacket(TYPE, DATA)`
  - KeyExchangeReply(self, ignored, pubKey, f, signature)

    Create a serverKey (this is returned from calling `keys.Key.fromString(pubKey)`). Compute the Diffie-Hellman shared secret using `f` and set the shared secret (using `self.SetSharedSecret`). Compute the exchange hash by calling `self.ComputeExchangeHash` and set this value (using `self.SetExchangeHash`).

    Return True if you the signature matches the exchange hash (`server.verify(sig,hash)` will test this for you).
  - ComputeExchangeHash(self, pubKey, f)

    Return a SHA1 hash digest of the following, in order:
    - Our version (`SSH_STR`)
    - The other version (`SSH_STR`)
    - Our key initialization payload (`SSH_STR`)
    - The other key initialization payload (`SSH_STR`)
    - The public key blob (pubKey) (`SSH_STR`)
    - Our public Diffie Hellman key (e) (`SSH_NUM`)
    - The server's public Diffie Hellman Key (`SSH_NUM`)
    - The Diffie Hellman shared secret (`SSH_NUM`)

    These values will be arguments to the function or made available by the superclass (**ClientCommandTransport.py**).

- MyServerCommandTransport
  - ComputeExchangeHash(self, clientDHpublicKey, serverDHpublicKey, sharedSecret, pubKey)

    Return a SHA1 hash digest of the following, in order:
    - The other version (`SSH_STR`)
    - Our version (`SSH_STR`)
    - The other key initialization payload (`SSH_STR`)
    - Our key initialization payload (`SSH_STR`)
    - The public key blob (pubKey) (`SSH_STR`)
    - The client's public Diffie Hellman Key (clientDHpublicKey) (`SSH_NUM`)
    - Our public Diffie Hellman key (serverDHpublicKey) (`SSH_NUM`)

- The Diffie Hellman shared secret (sharedSecret) (`SSH_NUM`)

  These values will be arguments to the function or made available by the superclass (**ServerCommandTransport.py**).

- GetDHSharedSecret(self, generator, prime, clientDHpublicKey)
  Initialize a DiffieHellmanServer object and compute the shared secret using `clientDHpublicKey`. Return a tuple of (Diffie-Hellman's) `F` and the shared secret.