Homework #4
CS468 Secure Programming and Systems Fall 2021

This homework is due on 12/3/2021. This is an individual homework assignment to be done by each student individually. The submission of your homework is your acknowledgement of the honor code statement

I have not copied any solution from anyone and not provided any solution to anyone on this assignment. The solution has been entirely worked out by me and represents my individual effort.

Please

- Use a word processor (MS-Word, LaTex, Framemaker, …) for your solutions. No hand writing submission will be accepted.
- Include your name and G-number at the beginning of your homework submission.
- Pack all your files with zip or tar and gzip and name your packed file as CS468_HW2_<your last name>_<your G#>.zip (or tar.gz)
- Submit your packed solution to the blackboard

This homework requires accessing, using and programming at zeus.vse.gmu.edu. Please create a directory named "CS468_HW4_<your last name>_<your G#> and do everything of this HW there.

1. Answer the following questions (20 points)

   1) Explain the concept of IPsec Security Association (SA) (5 points)

   2) What is IPsec SPI? Why does IPsec header need to include SPI? (5 points)

   3) g^{12345} mod 65537 has many multiplictive inverses in the form of g^x mod 65537 where $g>0$, $x>0$. Find one such x. (5 points)

   4) Explain the concept of firewall, list 3 limitations of firewall (5 points)

2. Implementing more secure authenticated remote shell client and server (50 points)

In HW3, you have been asked to develop two C programs: `RShellClient1.c` and `RShellServer1.c` with primitive authentication. However, the authentication protocol in HW3 has a number of serious vulnerabilities. Specifically, it does not provide data integrity and confidentiality, it is subject to MITM (man-in-the-middle) attack and replay attack

Now you are asked to improve the above client and server by implementing a more secure authentication based on challenge and response with nonce, user ID and password. Specifically, you need to develop two C programs: `RShellClient2.c` and `RShellServer2.c` (you can reuse any part of your `RShellClient1.c` and `RShellServer1.c`) such that

- `RShellServer2 <port number> <password file>` will listen on the specified <port number> and authenticate the remote shell command based on the SHA1 hash of the client's password and the random nonces from both client and server. If the authentication is successful, execute the shell command and return the execution result back to client. The <password file> should contain one or more line: <ID string>; <hex of SHA1(PW)>. The server will challenge the client and use the shared secret SHA1(PW) to authenticate client request. It will execute the shell command requested by the client once it has successfully authenticated the client.
- `RShellClient2 <server IP> <server port number> <ID> <password>` will read shell command from the standard input (i.e., the keyboard) and send the <ID> and the shell command to the server listening at the <server port number> on <server IP>. It will exchange nonce with the server and respond to the authentication challenge sent by server.

Protocol Specification

In order to have the functionalities described above, you need to implement the following protocol upon TCP between the client and server:

1. The client sends a `RSHELL_REQ` message that includes the client's ID and a random 32-bit `Nonce1`.
2. The server responds with an `AUTH_CHLG` message with a random 32-bit `Nonce2` to the client – asking the client to provide the appropriate response to the challenge.
3. The client responds with an `AUTH_RESP` message that contains the 256-bit AES CBC encryption of the server's `Nonce2+1` and the shell command: E_k`(Nonce2+1|Command)` where the encryption key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)` (i.e., `SHA256` of the concatenation of `SHA1(PW)`, `Nonce1` and `Nonce2`).
4. The server decrypts the received message with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)` and verifies if the decrypted 32-bit number equals to `Nonce2+1`. If not, the server sends an `AUTH_FAIL` message to the client. Otherwise, a) the server sends an `AUTH_SUCCESS` message (which contains the 256-bit AES CBC encryption of `Nonce1+1` with key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)`) to the client; b) the server executes the shell command from the client and sends the result back to the client via `RSHELL_RESULT` message(s).
5. If the client receives `AUTH_FAIL` message from the server, it should print out an error message "Authentication failed" and exit gracefully. If the client receives `AUTH_SUCESS` message, it should decrypt it with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)` and verify if the decrypt number equals to

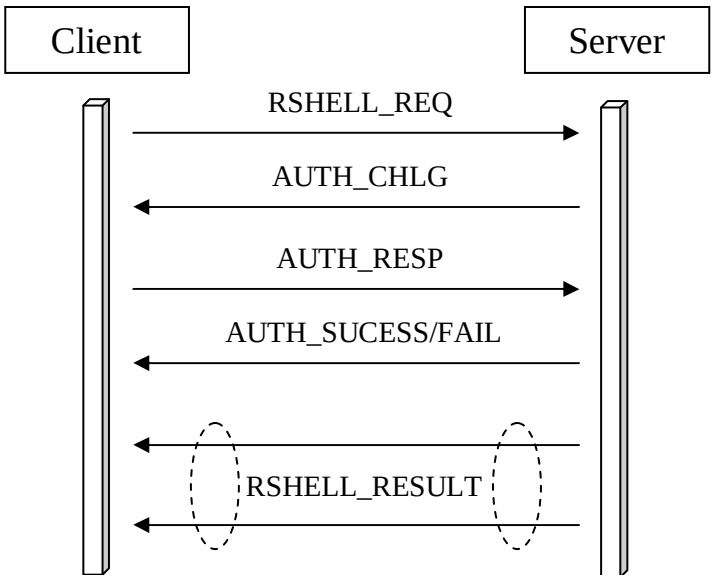

Figure 1: Message Interaction between RShellClient1 & RShellServer1

`Nonce+1` and print out a message indicating the validity of the received `AUTH_SUCESS` message. If the received `AUTH_SUCESS` message is valid, it should print out the shell command result received from `RSHELL_RESULT` message(s). Here the server may send one or multiple `RSHELL_RESULT` messages depending on the volume of the result. The client should be able to handle multiple `RSHELL_RESULT` messages. Again, the content (except the ID) of the `RSHELL_RESULT` message is encrypted with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)`.

Note:
The 256-bit AES encryption/decryption uses CBC mode with the first 128 bit of `SHA256(Nonce1|Nonce2)` as the IV.

The client and server could repeat the above steps. Unlike HW3, the authentication here is done for every shell command the client sends to the server.

Message Format

Figure 2 illustrates the format of all the messages of the protocol. Each message starts with the 1-byte type field, then the 2-byte payload length field that contains the number (in network byte order) of bytes of the payload that is right after. The payload is of variable length depending on the message type.

The ID field is of fixed length of 16 bytes. It contains a string of maximum length 15 bytes that denotes the user ID (e.g., Alice). The ID string is null terminated.

In message RSHELL_REQ, the “Nonce1” is a random 32-bit number generated by the client.

In message AUTH_CHLG, the “Nonce2” is a random 32-bit number generated by the server.

Message Format	Message Type 1 byte	Payload Length 2 bytes	Payload (variable length)		
RSHELL_REQ	0x11	Payload Length	ID	32-bit Nonce1	
AUTH_CHLG	0x12	Payload Length	ID	32-bit Nonce2	
AUTH_RESP	0x13	Payload Length	ID	E_k(Nonce2+1|Command)	
AUTH_SUCESS	0x14	Payload Length	ID	E_k(Nonce1+1)	
AUTH_FAIL	0x15	Payload Length	ID	32-bit Nonce1	
RSHELL_RESULT	0x16	Payload Length	ID	MR	E_k(Execution Result)

Figure 2: Message format of all the messages between RShellClient2 and RShellServer2

In message AUTH_RESP, the client increment the received Nonce2 by 1, then encrypt it and the shell command it wants the server to execute (e.g., pwd) with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)`.

In message AUTH_SUCCESS, the server encrypts Nonce1+1 (Nonce1 got from message RSHELL_REQ) with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)`.

In message AUTH_FAIL, the server simply include the Nonce1 got from message RSHELL_REQ.

In message RSHELL_RESULT, MR (more result) is a 1-byte field representing if there is any more execution result after the current RSHELL_RESULT message. If there is no more execution result, MR field should be zero. Otherwise, WR should be non-zero. The "Execution Result" is simply the execution result of the shell command at the remote host. Normally it should be printable ascii string. But it could contain unprintable char (e.g., if you trying to display a binary file). The server encrypt the execution result with 256-bit AES CBC using key k=`SHA256(SHA1(PW)|Nonce1|Nonce2)`.

Experiments:

1) create a text mode password file named `passwdfile.txt` that contains the following line:

```
Alice; <hex representation of SHA1(PW)>
```

where "Alice" is a recognized ID, and the PW is "SecretPW". Note you can obtain the SHA1 of "SecretPW" via appropriate OpenSSL command.

Take a screen shot or log (via script command) of the execution of the following commands:

2) run `./RShellServer2 <port num> passwdfile.txt` Here you want to choose some random port num between [1025, 65535]. If someone else is using the port num you picked, your server won't be able to bind it. You can try some other port number.

3) run `./RShellClient2 localhost <port num> InvalidID SecretPW` The server should reject any shell command it sends as InvalidID is not defined in the passwdfile.txt

4) run `./RShellClient2 localhost <port num> Alice WrongPW` The server should reject any shell command it sends as the SHA1 hash of WrongPW does not match that in passwdfile.txt

5) run `./RShellClient2 localhost <port num> Alice SecretPW` and type the following shell commands:

```
1) id
2) date
3) pwd
4) ls -l
```

the server should accept and execute any shell command it sends and return the execution result to the client. The client should print out the execution result on screen.

Name the screenshot or lof file "`CS468-HW4-1.*`" (here * depends on the format of your screenshot, e.g., jpg).

Submit:

- RShellClient2.c, RShellServer2.c, passwdfile.txt, corresponding makefiles and/or information about how to generate the executable from the source code. Note, you need to provide everything needed to generate the executable in zeus environment.
- `CS468-HW4-1.*`

3. Answer the following questions regarding the security protocol of problem 2 (30 points)

5) Is the authentication protocol subject to replay attack (e.g., the adversary replay captured packet in real-time)? Why?

6) How is client authenticated to the server?

7) Is the server authenticated to the client? Why?