pcollins
pmcollins2

1.) This scheme is not secure at all. Given what we know is public which includes:

g^x *(mod G)= X,* g^y *(mod G) = Y, g* and *G*

and what is private which includes:

x and *y*

The value $X \cdot Y = g^x \times g^y \pmod G = g^{x+y} \pmod G$ is easily calculable with the public information, which means the shared secret is publically calculable. A adversary trying to create it would not actually care what the private *x* or *y* is since it isn't necessary. In normal Diffie-Hellman key exchange each party must either know *x* or *y* in order to calculate the shared secret $g^{xy} \pmod G$. The adversary must solve the Discrete Log Problem in either $g^x \pmod G = X$ or $g^y \pmod G = Y$ for *x* or *y* which given the Baby-step, Giant-step algorithm takes $O(\sqrt{|G|})$ amount of work which is very time consuming and inhibitive for large Groups *G*.

2.1) The password is "145219".

2.2) My code is a basic for-loop that tries different "passwords" that are numbers starting at 0 and continuing to some max value. In the loop, I calculate the hash by concatenating the username salt and password and running the sha256 function. I then compare this hash to the given password hash. If it is the same I break out of the loop because I've found the password or else I continue on. Something like:

```
for (int password from 0 to infinity and beyond):
    if(sha256_hash("ristenpart,"+password+",134153169")==givenHash):
        print "Found password:"+password
        break;
```

The worst case running time for this algorithm is *O(N)* given that the password is a number. However, I wrote this in Go and multithreaded it, so it calculates it in less than a second. Check it out in **hasher.go** ☺.

2.3) This attack might be made intractable by using a Password-Based Key-Derivation Function such that essentially runs the hashing function thousands of times, so that brute-forcing a single password hash would be orders of magnitude more work.

3.1) The pseudo-code for the bad encryption goes as follows:

Open and read the key for the MAC and AES Cipher
Grab plaintext message from command arguments
Create the header of the encryption containing the length of the message and prepend
MAC the header and message and append it
Pad the end with zeros, so that the plaintext length is a multiple of the block size
Generate a random initialization vector the CBC mode of the AES 256 cipher
Encrypt the plaintext with header, message, hash, and padding with AES 256 CBC
Print out the result with the initialization vector prepended

The pseudo-code for the bad decryption is as follows:

- Open and read the key for the MAC and AES Cipher
- Grab ciphertext message from command arguments
- Slice off the initialization vector from the ciphertext
- AES 256 CBC decrypt the ciphertext
- Read off the header of the plaintext
  - Validate the version of the header
  - Validate the subversion of the header
  - Validate the reserved byte of the header
  - Validate the second reserved byte of the header
  - Validate the length of the plaintext given the message length
- Validate the MAC hash comparing the plaintext MAC and the hashed plaintext
- If all validation passes, print that the message was received

3.2) The script that implements the attack is called **oracle.py**. It discovers the first four bytes of the first plaintext block via a header oracle. A paper that it is based on a similar SSH flaw that helped me understand it is referenced here http://www.isg.rhul.ac.uk/~kp/SandPfinal.pdf

3.3) The new implementation should be a Encrypt-then-MAC scheme and likewise a MAC-then-decrypt scheme. The pseudo-code for the good encryption goes as follows:

- Open and read the key for the MAC and AES Cipher
- Grab plaintext message from command arguments
- Create the header of the encryption containing the length of the message and prepend
- Pad the end with zeros, so that the plaintext length is a multiple of the block size
- Generate a random initialization vector the CBC mode of the AES 256 cipher
- Encrypt the plaintext with header, message, and padding with AES 256 CBC
- MAC the ciphertext and append it
- Print out the result with the initialization vector prepended

The pseudo-code for the good decryption is as follows:

- Open and read the key for the MAC and AES Cipher
- Grab ciphertext message from command arguments
- Validate the MAC hash comparing the ciphertext MAC and the hashed cipertext
  - Stop execution and return if it is incorrect
- Slice off the initialization vector from the ciphertext
- AES 256 CBC decrypt the ciphertext
- Read off the header of the plaintext
  - Validate the version of the header
  - Validate the subversion of the header
  - Validate the reserved byte of the header
  - Validate the second reserved byte of the header
  - Validate the length of the plaintext given the message length
- If all validation passes, print that the message was received