Zhexin Qiu
zhexinq

Homework 1

- Using PGP

1. First, I check whether the fingerprint of Mahmood's public key I found matches the one provided in the write-up. Since a PGP public key's fingerprint is almost unique (is resulted from a hash function where collision is extremely rare). As long as that one provided in the handout is authenticated and matched with the one I found, I can be pretty sure it's valid. Another way is to look for certification: if the user information matches those provided in write-up and it is certified by some CA who signed on this public key or in the PGP's web of trust model, some users that I trust attested to this certification, then the public key's validity holds.
2. My email address: zhexin.qiu@sv.cmu.edu fingerprint: 308D AD4D 6D13 D8B8 058A 2C82 0E45 6D8A 1B9B 75F5
3. I will choose using PGP/MIME. Basically because PGP/Inline doesn't support attachments. Since I know Mahmood has several mail clients supporting PGP, PGP/MIME is a better choice for combining message body with attachments to encrypt HTML mails or special character set.

- Padding Attack

1. See attached .py script
2. See attached .py script

- Finding MD5 collisions

1. Assuming Wang and Yu's results are well know, two almost identical files with only 128 bytes in the middle are different can be created to have the same MD5 hash. For attacking executables, one can get two pairs of 64-byte blocks having the same MD5 hash with an arbitrary initialization vector from collision generator implemented by Wang and Yu's algorithm. For collision attacks on two executables, one can divide the binary into 64-byte chuncks and decide at which two chunks he wants to make the two files different. Then he can compute the MD5 hash of the chunks before those two he want to modify, which result as a 16-byte initial vector. Using this IV with the Wang and Yu's collision generator, he can find a pair of different 128-byte numbers, which can be placed in two executables that evaluated to same MD5 hash.

2. Please use chmod 777 to change the executables to be runnable if permission is not allowed

1.
   a. Since the MD5 hashing always pad the input file to multiple of 64-byte blocks and Wu and Yu's method guarantees to produce a pair of 128 byte different values evaluated to same MD5 hash for arbitrary initial vector. So long as this pair of 128 byte values are placed in symmetric 64-byte slots in two files, the file data before and after them can be of arbitrary length with padding to multiple of 64 bytes. That is, as long as two files are more than 128 bytes, one can reserve a 128 byte space in the middle, pad the bytes before the reserved space to be multiple of 64 bytes, compute the initial vector, use this value to get a pair of 128-byte MD5 collision numbers, and place them into reserved space. The bytes after the reserved space can be arbitrary length, since the MD5 algorithm will pad them anyway.
   b. In order to implement a collision attack on a document, I'll use the same principles: find a 128-byte slot where I want to inject different characters, compute the initial vector up to the point where I want to inject, use that IV to compute two MD5 collision 128-byte strings, and inject these two in the location chosen before. In a document that supports condition constructs (e.g. a presentation language such HTML/CSS/JavaScript), a similar attack can be used to produce drastically different web contents presented by a browser just like the ones used in modifying executables. However, I think it's very difficult to attack an ASCII document in a stealth way, because ASCII document doesn't provide condition constructs. Its values only range from (0~127), it's really hard to find a pair of 128-byte MD5 collision values such that all bytes are within ASCII range. It's even harder to have both of them have semantics in human language. Even if they have, the output difference is not very effective.

- Public Key Cryptography

1. According to RSA, given p = 23, q = 17, e = 3, we have: $\varphi(n) = \varphi(pq) = (p-1)(q-1) = 22 \times 16 = 352$. Since we choose e = 3, to satisfy the requirement $3 \times d \equiv 1 \ mod \ 352$:
   Euclidean algorithm:
   $$352 = 3 \times (117) + 1 \qquad (I)$$
   Using I to rewrite:
   $$352 + 3 \times (-117) \equiv 1 \ mod \ 352$$
   $$352 + 3 \times (235) \equiv 1 \ mod \ 352$$
   $$3 \times (235) \equiv 1 \ mod \ 352$$
   => d = 235
2. The keyboard has the public key (pq, e) = (391, 3) and the SecApp has private key (391, 235) and also public key. The encryption/decryption scheme for a keyboard input 'B' goes as follows:
   The keyboard first encrypt the message using the public key:
   'B' = 66 (in decimal) -> $c = m^e \ mod \ n$: 66^3 mod 391 = 111
   The SecApp receives the encrypted message and use the its private key to decrypt:
   $m = c^d \ mod \ n$: $111^{235} \ mod \ 391 = 66$, which result in character 'B'
3. In my opinion, one advantage to have a low e parameter as 3 is that it's fast to compute a private key corresponding to it, because any large number (coprime with 3) divides by 3 will have remainder either 1, 2, which only requires two substitutions of equations to compute corresponding private key d. A disadvantage of choosing 3 as exponent is when RSA is used in message validation with not suitable padding. Cryptographer Bleichenbacher have found the vulnerability in RSA with PKCS1.5 padding scheme such that he was able to fake an RSA signature of any certificate.
4. The purpose of XYZ Corp is to prevent malware from sniffing/manipulating keyboard inputs to SecApp.
   a. First of all, I don't think the RSA scheme can prevent other applications from manipulating inputs to SecApp. Since the public key is known, any applications can grab the public key and generate any messages that want to SecApp, there is not way SecApp could know whether this message comes from the keyboard or somewhere else. Unless this scheme also provides authentication.
   b. RSA as public-key encryption is not as efficient as private-key encryption. So XYZ's design could undermine the performance of the keyboard when typing in messages into SecApp. Instead, a private key design can speed up performance if a secure channel for key exchange is provided
   c. Also the prime parameters p, q chosen for this design is too small. It would be very easy for attacker to factor n = pxq, since is a very small number. Hence, any applications know p,q can compute the private key d, and gain access to all messages typed on the keyboard.