CSE 667, Assignment 3

Due: Friday, November 13, 2020, by 11:59 pm.

Note 1: The total mark for this assignment is 30.

Note 2: You should *NOT* directly copy anything from slides or other resources. You may get the ideas from slides but what you submit *must be in your own words*. Any help must be acknowledged.

1. Describe the Decrypting RSA using Obsolete and Weakened eNcryption (DROWN) attack. (6 points)

The DROWN attack makes use of a flaw in an old version of TLS 1.0, called SSL 2.0, that many servers support alongside modern TLS in order to enable old clients to connect. The DROWN attack comes in due to architects’ tendency to be lazy with updates. A single TLS server usually only carries a single public-key pair. Anytime TLS 1.0 is used, it makes use of the same public key that SSL 2.0 uses for clients of a different nature, assuming the architect in question has not changed their public keys or varied their use. This is problematic because SSL 2.0 is susceptible to Bleichenbacher’s attack.

To execute DROWN, an adversary records the ciphertext used in a session of the modern version of TLS, (the book notes 1.3), which is encrypted using the public key shared with the SSL 2.0 server. The adversary then uses the ciphertext to perform Bleichenbacher’s attack, described in a subsequent question, on the old SSL 2.0 server, and compromises the key. From here, the adversary has now compromised TLS due to them sharing the same key.

2. In order to reduce the decryption time of the RSA with a 4096-bits modulus for implementation in resource-constrained devices like IoT devices, smart cards, RFID tags, etc, it has been suggested to use a 1000-bit private exponent. Is the resulting scheme secure? Prove your claim. (6 points)

In class, we discussed Weiner’s attack developed in 1987. The algebra behind this attack is not necessary to explain the security of this proposed scheme, but it does result in the idea the private decryption key for RSA can be determined if a certain condition is met. This condition is that if d < N0.25, the RSA encryption scheme is guaranteed to be insecure.

The scheme proposed states that it uses a ‘4096-bits modulus’ and a ‘1000-bit private exponent’, but notice the wording here does not state values, but rather bits. Now, a 4096 bit modulus encapsulates every number from 0 to 24096, and a 1000 bit private exponent represents every number from 0 to 21000.

Let’s take the largest numbers from each set, and see if we can fit them into our known inequality: d < N0.25 . We can substitute 21000 in for d , since it is the largest possible quantity that could break this measure. For N, we can substitute 24096, but we also need to include the 0.25, so this yields 2(4096 \* 0.25).

Our new inequality reads: 21000 < 2(4096 \* 0.25)

Since we have equal bases, we can compare the exponents of both sides of the inequality yielding

1000 < (4096 \* 0.25)

Which simplifies to

1000 < 1024

Which is true. Since this inequality is true, the resulting scheme is insecure and should definitely not be used. Any value of d lower than 1024 will be susceptible to Weiner’s attack.

3. Describe Bleichenbacher's attack on RSA-PKCS1 v1.5 which applies to the SSL 3.0 protocol. (6 points)

Bleichenbacher’s attack on RSA-PKCS1 v1.5 is described in the paper, “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”. In addition, I used the resource described “What’s So Special About PKCS#1 v1.5? And The Attack That Just Won’t Go Away!”, another article published in order to help me understand the more difficult concepts, and Dr. Bibak’s lecture. In plaintext, the attack works as follows:

Recall that unlike normal RSA encryption, PKCS1 v1.5 adds some additional fields not present in stock RSA encryption, namely the idea of padding the cipher with some randomized information. Every PKCS1 padding must begin with the same two bytes: 0x00 0x02, in order to distinguish the start of the block from the remainder of the cipher. In addition, padding is provided as needed to ensure the overall plaintext matches the modulus value for the encryption.

We begin the attack by sending our first message to the oracle, or in this case, the service performing the encryption. This message is mostly a dummy message, and begins with our two start bytes: 0x00 0x02, and any padding necessary to suit the modulus value. The cipher returned will be used to figure out the key, and so we save this for later.

We then perform a mock creation of the ciphertext by doing (c \* (s ^ e)) mod n, where c is an arbitrary integer (in this case the output of our initial cipher run), s is a series of chosen integers, and e is the pre-shared public key. N represents the modulus of the RSA encryption, which if not provided, can be guessed ‘to be a multiple of eight’ through brute force. We then send this mock ciphertext to the oracle, and it will respond with an error if the decoded message does not begin with 0x00 0x02, or it will respond favorably if the message does start with this information. The author repeats the creation of the mock ciphertext as many times as necessary, substituting a new value for s in each time, until the server can perform a valid decode on our sent message.

From this point, we fix (c \* (s ^ e)) to be the same length as our modulus (n) using additional padding, matching the specification for RSA-PKCS1. After this is secured, we divide the message we were able to successfully send by the values of s, and through some wonderful number theory + algebra magic, the output of this division yields the key used to encrypt our dummy message.

4. Describe OAEP, OAEP+, and SAEP+. (6 points)

* OAEP, otherwise known as Optimal Asynchronous Encryption Padding, is a newer modification of PKCS1, called PKCS1 v2.0. The process to encrypt with OAEP begins by taking the message and appending 0x01 to it followed by a large number of 0s for padding. We take a string of random values, send them through a random hash function/oracle H, and XOR the output of that Hash function with the full message block including padding. That output becomes the first half of the message to encrypt. The output is also sent through a separate random hash function/oracle G, and this is XORed with the previous string of random values to derive the second half of the full message block. From here, the two parts of the whole message are concatenated into one, and the entire block is encrypted. In practice, using SHA-256 for H and G are recommended.
* OAEP+ follows the same exact decryption and encryption processes with one key exception. The previous padding is replaced with a new field called ‘W(m, r)’, which represents yet another random hash function/oracle. During the decryption phase, this field is validated before the message is decrypted to determine if the message is valid or not, saving time on decrypts and demonstrating additional integrity.
* SAEP+ is the same as OAEP+ except it removes the random hash function/oracle G, as well as the second XOR step that is done following the usage of G. This saves even more time, removing an entire hash function, on both encyrpts and decrypts.

5. a) How the implementation of TLS 1.0 made it secure against Bleichenbacher's attack? Explain in detail. (3 points)

TLS 1.0 developed a server side modification to the decryption method in RSA PKCS1. It modified the description process by prebuilding a string of 48 random bytes before the message was decrypted. If the decrypt was successful, a correct message was sent back to the client. If the message was not successful, the random string was sent back instead. This process prevents a given client from using the Bleichenbacher attack, and does not reveal any information on the desired key, e.

b) How the decryption in OAEP does work? (3 points)

First, we take the first half of the output of OAEP, known as ‘plaintext to encrypt’, and feed this through hash function G. We then take the second half of the output of OAEP, known as ‘with RSA’ and XOR that with the output of G. We take the output of this operation, known as ‘random’ and pipe it through hash function H, and XOR the result of that with ‘plaintext to encrypt’ in order to get the original message.