Lab 3: Attacking RSA

This lab is due on **February 28** at **11:59PM**, following the submission checklist below. Late submissions will be penalized according to course policy. Your writeup MUST include the following information:

1. List of collaborators (on all parts of the project, not just the writeup)
2. List of references used (online material, course nodes, textbooks, wikipedia,...)
3. Number of late days used on this assignment
4. Total number of late days used thus far in the entire semester

If any of this information is missing, at least 20% of the points for the assignment will automatically be deducted from your assignment. See also discussion on plagiarism and the collaboration policy on the course syllabus.

While we provide you with the tools to run this lab at any platform (Windows, Linux and Mac OS), I strongly recommend working at a Linux or Mac OS machine as it will make things considerably easier for you. The instructions given for the rest of the description of the lab are therefore explicitly for this case (unless otherwise noted).

Administration: This lab will be administered by Sophia Yakoubov.

Introduction

In this lab, you will investigate vulnerabilities in RSA, when it is used incorrectly.

Objectives:

- Understand how to apply RSA encryption and digital signatures.
- Investigate how using RSA incorrectly can compromise confidentiality and integrity.

1 Getting Familiar with RSA

1.1 Textbook RSA

Here, we summarize a naive version of the RSA ("textbook RSA") encryption and digital signatures schemes. This version of RSA should *not* be used; you will show in this lab several ways in which it can be vulnerable. Consult `https://en.wikipedia.org/wiki/RSA_(cryptosystem)` for more detail.

1.1.1 Encryption

Below we describe the key generation (KeyGen), encryption (Enc), and decryption (Dec) algorithms of RSA.

1. KeyGen:
   (a) Choose two random primes p and q.
   (b) Let $n = pq$ be the modulus.
   (c) Select a random encryption exponent e such that the greatest common denominator of e and $(p-1)(q-1)$ is 1 (that is, e and $(p-1)(q-1)$ are relatively prime).
   (d) Let the decryption exponent d be $e^{-1} \bmod (p-1)(q-1)$.
   (e) Return the public key $pk = (n,e)$ and the secret key $sk = d$.
2. $\mathsf{Enc}(pk = (n,e), m)$, where m is a message to be encrypted:
   (a) Return the ciphertext $c = m^e \bmod n$.
3. $\mathsf{Dec}(sk = d, c)$, where c is a ciphertext to be decrypted:
   (a) Return the recovered message $m = c^d \bmod n$.

As an example, say that during KeyGen, I choose $p = 5$ and $q = 11$. Therefore, I have $n = 55$. If I choose $e = 3$, I need to find a d such that $3d \bmod (5-1)(11-1) = 3d \bmod 40 = 1$. I can try a few numbers, and see that $3 \times 27 \bmod 40 = 81 \bmod 40 = 1$, so I set $d = 27$.[1]
Now, say that you know only $n = 55$ and $e = 3$. Encryption is just raising the message m to the power of e; you can encrypt $m = 51$ as $c = 51^3 \bmod 55 = 132651 \bmod 55 = 46$.
Decryption is the inverse of encryption; it takes the e-th *root* of the ciphertext. e and d are chosen in such a way that raising to the power of d is the same as taking the eth root. Therefore, I, who also know $d = 27$, can decrypt $c = 46$ by taking $m = 46^{27} \bmod 55 = 51$.
Taking the eth root of a value modulo an n whose factorization you don't know is considered to be unrealistic for large numbers. However, in our example, someone else, who doesn't know $d = 27$, can also figure out what m is, just by trying each number modulo 55 and raising it to $e = 3$. (If they

[1] In reality, there are way more efficient ways to do this than just trying a few numbers!

get 46, they know they've found the right m!) This is only possible because we chose small p and q. If p and q are over 1000 bits long, instead of just 5 bits long, decryption without knowledge of d (or the factorization of n) becomes unrealistic.

1.1.2 Digital Signatures

In "textbook RSA" digital signatures, key generation (KeyGen) is exactly the same as it is in the encryption scheme described above. Signing (Sig) is reminiscent of decryption, and verification (Ver) is reminiscent of encryption.

1. $\mathsf{Sig}(sk = d, m)$, where m is a message to be signed:
   (a) Return the signature $\sigma = m^d \bmod n$.
2. $\mathsf{Ver}(m, \sigma, pk = (n, e))$, where m is a message and σ is a signature to be verified:
   (a) Verify that $\sigma^e \bmod n = m \bmod n$.

1.2 Using RSA Securely

In order to avoid vulnerabilities such as the ones explored in this lab, instead of "textbook RSA", RSA PKCS #1 (`ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc`) is used.

1.2.1 Encryption

To encrypt a message m using RSA PKCS #1 version 1.5, the message is padded as follows:

$$\texttt{00}\ \ \texttt{02}\ \ \underbrace{\texttt{RR}\cdots\texttt{RR}}_{\text{at least 8 random non-zero bytes}}\ \ \texttt{00}\ \ \underbrace{m}_{\text{the message itself}}$$

The result is encrypted using the "textbook RSA" encryption function.

1.2.2 Digital Signatures

To sign a message m using RSA PKCS #1 version 1.5, the message is padded as follows:

$$\texttt{00}\ \ \texttt{01}\ \ \underbrace{\texttt{FF}\cdots\texttt{FF}}_{k/8-38\text{ bytes wide}}\ \ \texttt{00}\ \underbrace{\texttt{3021300906052B0E03021A05000414}}_{\text{ASN.1 "magic" bytes}}\ \ \underbrace{\texttt{XX}\cdots\texttt{XX}}_{\text{20-byte SHA-1 digest}}$$

The result is signed using the "textbook RSA" signing function. The algorithms that check the digital signatures need to be implemented very carefully. In Part 5, you will show that if the implementation of the verification algorithm is slightly off, PKCS1.5 signatures can be forged.

NOTE: The implementation vulnerabilities that are explored in this lab have occurred often enough in the wild that many cryptographers and practitioners have concluded that using RSA PKCS

v1.5 is not a good idea. RSA-OAEP is the suggested choice for RSA encryption. See `https://tools.ietf.org/html/rfc3560`.)
Avoiding the attacks discussed in this lab has also been cited as motivation for using elliptic curve cryptographic signatures (e.g., ECDSA) instead of RSA.
Nevertheless, RSA is still widely used in practice.

1.3 Textbook RSA in Python

You can experiment with RSA yourself, using Python. In `tools.py`, we provide several functions you can use in this exploration. For instance, we provide text to integer (and integer to text) conversion, because the RSA algorithms assume that the message m is an integer modulo n.

- `text_to_int`: takes in a string and returns an integer which can be used in RSA.
- `int_to_text`: takes in an integer and returns the corresponding string.

We also provide a few other functions:

- `find_root`: takes in integers x and r, and returns the integer component of the rth root of x. (Python's built in `pow` function can be used to do this, but it doesn't work for very large 'long'-type values.)
  For instance, `find_root`$(x = 50, r = 2) = 7$. $\sqrt{50} \sim 7.07106$, and 7 is the integer part of 7.07106.
- `combine_moduli`: takes in two integer moduli n_1 and n_2 and two integers x_1 and x_2, and returns an integer x mod $n_1 n_2$ such that $x \bmod n_1 = x_1$ and $x \bmod n_2 = x_2$.
  For instance, `combine_moduli`$(n_1 = 5, n_2 = 7, x_1 = 2, x_2 = 3) = 17$, since $17 \bmod 5 = 2$, and $17 \bmod 7 = 3$.

The following shows how to use Python to do some basic RSA operations:

```
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
import tools

# set the security parameter:
secparam = 2048

# generate an RSA key:
key = RSA.generate(secparam)
public_key = key.publickey()

# save the public and private keys to files and read them back:
open('key.pub', 'w').write(public_key.exportKey())
open('key.priv', 'w').write(key.exportKey())
```

```
# read the public key in:
recovered_public_key = RSA.importKey(open('key.pub', 'r').read())
assert recovered_public_key == public_key
# read the private key in:
recovered_key = RSA.importKey(open('key.priv', 'r').read())
assert recovered_key == key

# use the PUBLIC KEY to encrypt a message:
message = "I ATE SOME PIE"
message_int = tools.text_to_int(message)
ciphertext = public_key.encrypt(message_int, None)

# save the ciphertext to a file and read it back:
open('ciphertext', 'w').write(str(ciphertext))
recovered_ciphertext = eval(open('ciphertext', 'r').read())
assert recovered_ciphertext == ciphertext

# use the PRIVATE KEY to decrypt the message:
recovered_message_int = int(key.decrypt(ciphertext))
assert recovered_message_int == message_int

# use the PRIVATE KEY to sign a hash of the message:
hash = SHA256.new(message).digest()
signature = key.sign(hash, None)

# save the signature to a file and read it back:
open('signature', 'w').write(str(signature))
recovered_signature = eval(open('signature', 'r').read())
assert recovered_signature == signature

# use the PUBLIC KEY to verify the signature:
hash = SHA256.new(message).digest()
assert public_key.verify(hash, signature)
```

We can also use math operations in Python to explore RSA ciphertexts and signatures manually. For instance, you can manually verify the signature. The modulus can be accessed as public_key.n, and the public exponent can be accessed as public_key.e.

```
print "%0128x" % pow(signature[0], public_key.e, public_key.n)
# The suffix of the signature^e mod n should be:
print SHA256.new(message).hexdigest()
```

2 Small Message Space Attack

Imagine that you are a government agent trying to determine when your neighboring country, Malland, will attack. Open `part2_ctext` to find a “textbook RSA" ciphertext sent by Malland to its ally, Horridland. You know that the Mallanders are likely to be saying one of three things:

1. "attack tomorrow at dawn",
2. "attack just before dusk", or
3. "attack early next week".

The file `part2_key.pub` contains Horridland’s public key. Use this knowledge to figure out what the message is.
What would you do to prevent Malland from executing the same attack on ciphertexts *you* send?

What to submit

1. A Python 2.x script named `part2.py` that:
   (a) Takes in three parameters:
      i. The path to a file with a public key,
      ii. The path to a file with a ciphertext, and
      iii. The path to a file enumerating the possible messages, each on a new line.
   (b) Prints the identified message.

   This script should be callable as follows:

   ```
   goldbe$ python part2.py part2_key.pub part2_ctext part2_messagespace
   The message is 'xxxxx'.
   ```

2. A file named `part2.txt` with your suggestion for how to prevent the attack you executed in `part2.py`.

3 Small Plaintext and Encryption Exponent Attack

Open `part3_ctext` to find another “textbook RSA" ciphertext, sent by Malland to Horridland. However, this time, you don’t really know what Malland might be saying. You do, however, know that the encryption exponent is $e = 3$, and that Mallanders tend to be very brief and to the point, so you think the message in question is probably very short. Use this knowledge to figure out what the message is.

Defense: Padding A technique that prevents this attack is *padding*, which is used in PKCS #1. Padding artificially increases the size of the message so that it is almost as big as the modulus n by adding extra characters or bits to the message. Why would padding help prevent this attack?

What to submit

1. A Python 2.x script named `part3.py` that:

   (a) Takes in one parameter: the path to a file with a ciphertext.

   (b) Prints the identified message.

   This script should be callable as follows:

   ```
   goldbe$ python part3.py part3_ctext
   The message is 'xxxxx'.
   ```

2. A file named `part3.txt` that explains why padding prevents the attack you executed in `part3.py`.

4 Broadcast Attack

Since you thwarted Malland's plots with Horridland, Malland enlisted the help of two more allies - Awfuland and Badland. Horridland, Awfuland and and Badland made the mistake of all having the same low encryption exponent, $e = 3$. Their public keys are in `part4_key1.pub`, `part4_key2.pub`, and `part4_key3.pub`, repectively. Malland sends Horridland, Awfuland and Badland a plan of attack, still using "textbook RSA" encryption. You get your hands on all three ciphertexts:

1. In `part4_ctext1`, find the ciphertext Malland sent Awfuland, encrypted as $c_A = m^3 \bmod n_A$ (using Awfuland's modulus n_A).

2. In `part4_ctext2`, find the ciphertext Malland sent Badland, encrypted as $c_B = m^3 \bmod n_B$ (using Badland's modulus n_B).

3. In `part4_ctext3`, find the ciphertext Malland sent Horridland, encrypted as $c_H = m^3 \bmod n_H$ (using Horridland's modulus n_H).

Use the fact that all three ciphertexts use the same low encryption exponent to figure out what their plan of attack is.
Hint: Use the provided function `combine_moduli` in `tools.py`.

If Malland *padded* the message by adding enough '1' characters to the end to make it almost as long as the modulus, would that prevent you from recovering their message? If not, how *could* Malland go about thwarting you?

What to submit

1. A Python 2.x script named `part4.py` that:

   (a) Takes in six parameters:

      i. three paths to files with public keys (all of whose public exponents are $e = 3$), and

      ii. three paths to files with ciphertexts.

   (b) Prints the identified message.

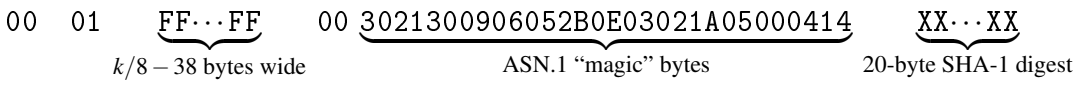
   This script should be callable as follows:

```
goldbe$ python part4.py part4_key1.pub part4_key2.pub part4_key3.pub
part4_ctext1 part4_ctext2 part4_ctext3
The message is 'xxxxx'.
```

2. A file named `part4.txt` that explains whether padding the message with '1' characters prevents the attack you executed in `part4.py`, and if not, how that attack could be prevented.

5 Breaking PKCS #1 v1.5

Up until now, we have been talking about encryption. Now, we'll take a brief detour and talk about RSA signatures. Signatures are frequently generated on *hashes* of messages, instead of on messages themselves. PKSC #1 v1.5 is a signing standard widely used on the internet. It involves adding padding to a message hash before signing it using RSA, as described in Part 1.2.2. When PKSC #1 v1.5 is used, an RSA signature on the following padded message hash is generated:



When PKCS padding is used, it is important for implementations to verify that every bit of the padded, signed message is exactly as it should be. It is tempting for an implementor to validate the signature by first stripping off the `00 01` bytes, then some number of padding `FF` bytes, then `00`, and then parse the ASN.1 and verify the hash. If the implementation does not check the length of the `FF` bytes and that the hash is in the least significant bits of the message, then it the public verification exponent e is low, is possible for an attacker to forge values that pass this validation check.

Say that $e = 3$, as in Parts 3 and 4. If the length of the required padding, ASN.1 bytes, and hash value is significantly less than $n^{1/3}$ then an attacker can construct a cube root over the integers whose most significant bits will validate as a correct signature, ignoring the actual key. To construct a forged signature that will validate against such implementations, an attacker simply needs to construct an integer whose most significant bytes have the correct format, including the hashed message, pad the remainder of this value with zeros or other garbage that will be ignored by the vulnerable implementation, and then take a cube root over the integers, rounding as appropriate.

Historical fact: This attack was discovered by Daniel Bleichenbacher, who presented it in a lightning talk at the rump session at the Crypto 2006 conference. At the time, many important implementations of RSA signatures were discovered to be vulnerable to this attack, including OpenSSL. In 2014, the Mozilla library NSS was found to be vulnerable to this type of attack: `https://www.mozilla.org/security/advisories/mfsa2014-73/`.

Horridland is still using a broken implementation of signature verification; in fact, below is the code they are using.

```
ASN1_MAGIC = "003021300906052b0e03021a05000414"

def verify_rsa(sig_hex, message, public_key):
    """
    Verifies an RSA signature.
    inputs:
        sig_hex: a digital signature, in hexadecimal
        message: the message string to which sig_hex corresponds
        public_key: the public verification key
            (a Crypto.PublicKey.RSA._RSAobj value)
    output: true or false
    """
    sig_int = int(sig_hex , 16)
    m_int = pow(sig_int, public_key.e, public_key.n)
    m_hex = "%0512x" % m_int
    h = SHA.new(message).hexdigest()
    return re.match('0001f*' + ASN1_MAGIC + h, m_hex) is not None
```

Open `part5_key.pub` to find Malland's public verification key. You want to tell Horridland to retreat, and make them believe that it's a message from their ally Malland. Using the signature forgery technique described above, produce an RSA signature on "retreat immediately" that Horridland would accept.

Since 2010, NIST has specified that RSA public exponents must be at least $2^{16} + 1$. Briefly explain why Bleichenbacher's attack would not work for these keys.

What to submit

1. A Python 2.x script named `part5.py` that:
   (a) Takes in two parameters:
      i. the path to a file with a public key, and
      ii. a message string.

(b) Prints a valid signature on the message under the given public key.

This script should be callable as follows:

```
goldbe$ python part5.py part5_key.pub 'retreat immediately!'
Successfully forged a signature on 'retreat immediately!':
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
2853cc9cf8c785651a3f5dcdea7c4ff005af90887d7c0fbaba09a4cf1
5276fe10d4cffe86db8fcc7efa92ab0138eff20b6c1eb175728afa269
36a4f00aabad04939057c5f363409ba0a28b3e1b5716d088f673e161
```

2. A file named `part5.txt` containing your answer to the question above (why Bleichenbacher's attack wouldn't work with a high public verification exponent).

6 RSA in the Wild

Find one network protocol that uses RSA on the internet. Make sure to cite your source, and to specify the version of the protocol that you are talking about!

1. A file named `part6.txt` containing the name and version of the network protocol that uses RSA on the internet, and a citation to your source.

Submission Checklist

- Part 2:
  1. `part2.py`
  2. `part2.txt`
- Part 3:
  1. `part3.py`
  2. `part3.txt`
- Part 4:
  1. `part4.py`

    2. `part4.txt`

- Part 5:
    1. `part5.py`
    2. `part5.txt`
- Section 6:
    1. `part6.txt`