Cryptographic Attacks[1]

This project is due on **Tuesday October 30th** at **11:59 p.m.**. **This is an individual project.**

The code and other answers you submit must be entirely your own work. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone else. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via Blackboard, following the submission checklist at the end of this file.

Introduction

In this project, you will investigate vulnerabilities in poorly implemented RSA signature schemes.

RSA Signature Forgery

A secure implementation of RSA encryption or digital signatures requires a proper padding scheme. RSA without padding, also known as *textbook RSA*, has several undesirable properties. For example, it is trivial for an attacker with only an RSA public key pair (n, e) to produce a mathematically valid message, signature pair by choosing an s and returning (s^e, s).

In order to prevent an attacker from being able to forge valid signatures in this way, RSA implementations use a padding scheme to provide structure to the messages that are encrypted or signed. The most commonly used padding scheme in practice is defined by the PKCS #1 v1.5 standard, which can be found at `https://tools.ietf.org/html/rfc2313`. The standard defines, among other things, the format of RSA keys and signatures and the procedures for generating and validating RSA signatures.

1.1 Validating RSA Signatures

You can experiment with validating RSA signatures yourself. Create a file called `key.pub` that contains the following RSA public key:

[1] This project is taken from a project designed by Nadia Heninger for her CIS 331 course at UPenn, and I am very grateful to her for letting me borrow it.

```
-----BEGIN PUBLIC KEY-----
MFowDQYJKoZIhvcNAQEBBQADSQAwRgJBALB8X0rLPrfgAfXMW73LjKYb5V9QG5LU
DrmsA9CAittsLvh2c082wHwVyCIiWQ8S3AA/jfW839sFN4zAZkW2S3cCAQM=
-----END PUBLIC KEY-----
```

You can view the modulus and public exponent of this key by running:

```
$ openssl rsa -in key.pub -pubin -text -noout
```

Create a file containing only the text `CIS 331 rul3z!`.

```
$ echo -n CIS 331 rul3z! > message
```

The following is a base64-encoded signature of the file `message` using the public key above.

```
C+XuJ3pAF0p496uGTnqtMaCUTClnKHGsyoK5WjLBfnivIeGQjK1e6KabqdjLKJQ8
WsFrf0Wf/auH3KOSprg2QQ==
```

Convert this signature into a binary file (on Linux: `base64 -d -i sig.b64 > sig`.)

```
$ base64 -D -i sig.b64 > sig
```

Now verify the signature against the file you created.

```
$ openssl dgst -sha1 -verify key.pub -signature sig message
```

We can also use basic math operations in Python to explore this signature further. Remember, RSA ciphertexts, plaintexts, exponents, moduli, and signatures are actually all integers.

Open a Python shell and run the following commands to import the signature as an integer:

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Hash import SHA
>>> signature = int(open('sig').read().encode('hex'),16)
```

Next, import the public key file that you created earlier:

```
>>> pubkey = RSA.importKey(open('key.pub').read())
```

The modulus and exponent are then accessible as `pubkey.n` and `pubkey.e`, respectively.

Now reverse the signing operation and examine the resulting value in hex:

```
>>> "%0128x" % pow(signature, pubkey.e, pubkey.n)
```

You should see something like `'0001fffff ... f8c6ee82f9d0bca80b80f72a5337375c3d99695e'`.

Verify that the last 20 bytes of this value match the SHA-1 hash of your file:

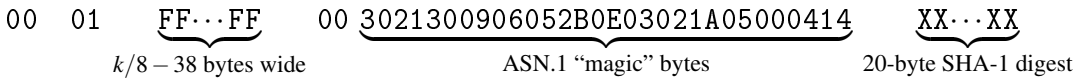
```
>>> SHA.new("CIS 331 rul3z!").hexdigest()
```

This component is intended to introduce you to RSA signatures; you will not submit anything for it.

1.2 PKCS #1 v1.5 Signature Padding

The signed value you examined in the previous section had been padded using the PKCS #1 v1.5 signature scheme. PKCS #1 v1.5 padding for RSA signatures is structured as follows: one `00` byte, one `01` byte, some `FF` bytes, another `00` byte, some special ASN.1 bytes denoting which hash algorithm was used to compute the hash digest, then the bytes of the hash digest itself. The number of `FF` bytes varies such that the size of m is equal to the size of the RSA key.

A k-bit RSA key used to sign a SHA-1 hash digest will generate the following padded value of m:



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 implementer 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 is possible for an attacker to forge values that pass this validation check.

This possibility is particularly troubling for signatures generated with $e = 3$. 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 "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.

1.3 Constructing Forged Signatures

The National Bank of CS 590 has a website at `http://cs590-f18.cs.purdue.edu/project4/bank` that its employees use to initiate wire transfers between bank accounts. To authenticate each transfer request, the control panel requires a signature from a particular 2048-bit RSA key that is listed on the website's home page. Unfortunately, this control panel is running old software that has not been patched to fix the signature forgery vulnerability.

Using the signature forgery technique described above, produce an RSA signature that validates against the National Bank of CS 590 site.

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. His talk is described in this mailing list posting: `https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html`. 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/`.

What to submit A Python 2.x script called `bleichenbacher.py` that:

1. Accepts a double-quoted string as command-line argument.
2. Prints a base64-encoded forged signature of the input string.

You have our permission to use the control panel at `http://cs590-f18.cs.purdue.edu/project4/bank`[2] to test your signatures. We have provided a Python library, `roots.py`, that provides several useful functions that you may wish to use when implementing your solution. You can download `roots.py` at `https://www.cs.purdue.edu/homes/clg/CS590/projects/roots.py`[3]. Your program should assume that PyCrypto and `roots.py` are available, and may use standard Python libraries, but should otherwise be self-contained.

In order to use these functions, you will have to import `roots.py`.You may wish to use the following template:

```
from roots import *
from Crypto.Hash import SHA
import sys

message = sys.argv[1]

# Your code to forge a signature goes here.

root, is_exact = integer_nthroot(27, 3)
print integer_to_base64(root)
```

1.4 [Bonus] Implement the Functionality in `roots.py`

For extra credit, you may implement the cube root functionality (as well as a few additional helper functions that might be useful) found in `roots.py` yourself! This means that you can implement the entire attack however you want, so long as you still achieve the desired goals. If you do not wish to do this extra credit, email me and I will provide you access to the completed `roots.py` file.

[2]Due to the way the server has been setup, it is only accessible in a special manner. You must be inside the Purdue CS network in order to access the webpage. One way to do this is using an RDP client (like Microsoft Remote Desktop) to access *mc18.cs.purdue.edu* using your normal Purdue login credentials. This will allow you to have a graphical interface within the CS network to access the website. But of course, there are most likely other ways that you can achieve the same thing on your local machine, and you are free to do whatever you would like.

[3]See Section 1.4 for more details about this

You can download `roots-skeleton.py` at `https://www.cs.purdue.edu/homes/clg/CS590/projects/roots-skeleton.py`. The skeleton code is also below.

```
# CS 590 Project 1 helper functions

def integer_nthroot(y, n):
    "‘"’"
    Return a tuple containing x = floor(y**(1/n))
    and a boolean indicating whether the result is exact (that is,
    whether x**n == y).

    >>> from sympy import integer_nthroot
    >>> integer_nthroot(16,2)
    (4, True)
    >>> integer_nthroot(26,2)
    (5, False)

    "‘"’"

def integer_to_base64(z):
    '''Converts an arbitrarily long integer to a big-endian base64 encoding'''

def integer_to_bytes(n):
    '''Converts an arbitrarily long integer to bytes, big-endian'''

def bytes_to_integer(b):
    '''Converts big-endian bytes to an arbitrarily long integer'''
```

Part 2. Writeup

1. 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.

2. If you completed Section 1.4, explain how your program works and anything that I might need to know to use it.

What to submit A text file named `writeup.txt` containing your answers.

Submission Checklist

Upload to Blackboard a gzipped tarball (`.tar.gz`) named `project1.`*`purdueid`*`.tar.gz` The tarball should contain only the following files. Do not make your files dependent on local files or esoteric libraries.

Section 1.3

`bleichenbacher.py`: A Python 2.x script that takes a string argument and outputs a signature forged using Bleichenbacher's attack.

[Optional] Section 1.4

`roots.py`: A Python 2.x script that implements the necessary cube root functionality.

Writeup

`writeup.txt`: A text file containing your answers to the wrap-up questions.

Bonus Challenge [Extra Credit]

Generate a digital signature for the sentence "My name is <your name>. My voice is my passport." that verifies correctly using OpenSSL with the following 1024-bit RSA public key. (Hint: The modulus might not have been generated like a normal RSA modulus.):

```
-----BEGIN PUBLIC KEY-----
MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQCgF35rHhOWi9+r4n9xM/ejvMEs
Q8h6lams962k4U0WSdfySUevhyI1bd3FRIb5fFqSBt6qPTiiiIw0KXte5dANB6lP
e6HdUPTA/U4xHWi2FB/BfAyPsOlUBfFp6dtkEEcEKt+Z8KTJYJEerRie24y+nsfZ
MnLBst6tsEBfx/U75wIBAw==
-----END PUBLIC KEY-----
```