Portion of a sample chapter — DO NOT DISTRIBUTE

A sample chapter by Jedidiah R. Crandall

0.1 Background

For background, see `https://arxiv.org/abs/1804.03367`. Basically, Lab 2 and QQ Browser have the same basic structure to how a client encrypts a message for the server: generate a random 128-bit AES session key, use that as plaintext for an RSA scheme and encrypt it with the server's public key, then send that RSA ciphertext followed by the AES encryption (using that AES session key) of the message over the wire.

0.2 How textbook RSA works

RSA is an asymmetric cryptography algorithm, also called a public key algorithm. This means that there is both a public key, which the entire world can know for all we care, and a private key that the decrypting party can keep to themselves and never share with anyone (not even the parties sending them encrypted messages). This is in contrast to a symmetric algorithm, like AES, where both the sender and receiver need a copy of the exact same key that is used for encryption and decryption. Here we'll repeat the version of RSA encryption that you'll see on Wikipedia or in any textbook, but instead of the usual suspects (Alice and Bob) we'll use QQ Browser clients and the QQ server hosted by Tencent to make the context more clear.

The problem we (putting ourselves for the moment in the heads of QQ Browser's developers) want to solve is that we want hundreds of millions of clients to be able to each send many messages a day to a server, but we don't want any of those clients (who might be NSA spies or bored graduate students) to be able to decrypt messages sent by other clients if they're somehow able to record them off the network. Baidu Browser and UC Browser failed horribly at this, because the secret key their servers used to decrypt messages was also hard-coded into the software for every client in the world, and every client used the same key.

One solution to this problem would be for every client to generate a random key to use for every message, and then somehow send that random key to the server. But how to send this key? You can't send it in plaintext, and you can't encrypt it with another secret AES key that is hardcoded into every client and the server because that just gets you back to your original problem. The clients need some way to encrypt messages to the server in a way that even they themselves can't decrypt.

RSA, an algorithm published by Rivest, Shamir, and Adleman in 1978, solves this problem. This is the technique that QQ Browser's developers used, pulling the algorithm right out of a textbook (and thus ignoring about four decades of research in cryptography). Before releasing the client software, the developers simply created an RSA key pair where the public key can be hardcoded into every client and the private key is given only to the server. Any message encrypted with the public key can only be decrypted with the private key, and it's computationally infeasible for anyone with the public key to guess what the private key is. Thus, the NSA (or Jeff) can download the QQ Browser client and extract the public key *via* reverse engineering, but it won't help them decrypt the communications that other clients are having with the server. In theory.

So, let’s take a look at what a textbook version of RSA looks like ala QQ Browser. First, we have to generate a key pair using the following steps:

1. Choose two large primes, p and q
2. Multiply them together to get $n = pq$
3. Calculate the totient, which happens to be $(p-1)(q-1)$
4. Using the totient and the Extended Euclidean Algorithm, find a matching e and d such that $ed \equiv 1 (\text{mod } n)$

e is coprime to the totient and d is e’s multiplicative inverse modulo the totient. Essentially, e and d are a carefully chosen pair that make the encryption (using e) and decryption (using d) work.

The encryption key, e, and the product of the two primes, n, can be made known to the whole world. Based on the underlying assumption that factoring large numbers is very hard for classical computers to do, telling the whole world the value of n doesn’t reveal to them what p and q were, so they can’t compute what d is even if they know e because they don’t know the totient.

So, QQ Browser does exactly that, they hardcode e and n into the client software that the whole world can see, and d is kept only on the server (the server will also need n, but it’s not part of the secret). So, we can say that the tuple (e, n) is the public key and the tuple (d, n) is the matching private key.

To use RSA public key cryptography to send their randomly chosen AES key to the server (a secret that they don’t want anybody but the server to know), a client encrypts it using the public key. So if m, or the message, is the 128-bit AES session key that a client wants to use to encrypt things to send to the server, they send the server m by encrypting it like this:

$$c \equiv m^e (\text{mod } n)$$

The funny-looking equals sign with three lines means the two numbers are equivalent, but “equals” is good enough to understand the computation. The “mod” simply means you take the remainder after dividing by n. All of the computations so far for generating the key, and encrypting, as well as the following computation for decrypting, are easy for computers to do with very large numbers because of various number theory tricks.

So c is the message that the client sends to the server, which is the encrypted copy of the 128-bit ephemeral AES key. The server can decrypt it using the private key:

$$m \equiv c^d (\text{mod } n)$$

This is because in RSA $ed \equiv 1 (\text{mod } n)$ by definition, so when you apply both the public and private key to something you are basically raising it to the power of 1 and getting back what you started with:

$$m \equiv (m^e)^d (\text{mod } n)$$
$$m \equiv m^{ed} (\text{mod } n)$$
$$m \equiv m^1 (\text{mod } n)$$
$$m \equiv m (\text{mod } n)$$

The beauty of it is that someone who knows the encryption key e can’t possibly decrypt the message in any practical way using e. You need the secret decryption key d to decrypt it, even though it was encrypted with e. This asymmetric property of RSA is why we can give the same encryption key to every client in the world and even spare them the labor of reverse engineering the code to find that key by publicizing it, but still maintain the property that only the server can decrypt the messages that the clients encrypt. In theory.

As it turns out, there are two little facts about textbook RSA that have to be dealt with in practical implementations of it. One is that RSA is *malleable*, meaning that we can change the plaintext in meaningful ways (without knowing what the plaintext is) by performing operations on the ciphertext. For example, let's say we want to multiply the secret message m (which we don't know) by 2. We can calculate a new ciphertext based on the encrypted ciphertext c that corresponds to m like this:

$$c' \equiv c \times 2^e \equiv (\text{mod } n)$$

We know c and e (recall that e is the public key). Now we have an encrypted ciphertext for $2m$:

$$c' \equiv c \times 2^e \equiv m^e \times 2^e \equiv (2m)^e (\text{mod } n)$$

The new ciphertext c' is the same as if we had encrypted $2m$.

The second fact that has to be dealt with in practice when implementing RSA is that information can leak if the server's observable behavior changes depending on the value of m that it decrypts. For example, a server might abort with an error message if the plaintext that it decrypts is greater than $2^{128} - 1$, or the maximum value of a 128-bit AES key. If we can send c' to the server after we recorded c from another client, we just learned whether $2m$ is greater than $2^{128} - 1$, which gives us the most significant bit of the original client's secret m that they encrypted into c. In other words, we just learned a bit of the AES key.

For these kinds of reasons it is not advisable to pick up an average crypto textbook, flip to the chapter about RSA, and start writing code. Techniques such as Optimal Assymetric Encryption Padding (OAEP) [2] and many best practices about how to handle error cases have to be implemented to avoid problems like those above. For a good book about how you're supposed to really engineer cryptography, see *Cryptography Engineering* [6].

0.3 QQ Browser's "fixed" cryptography implementation

So now let's look at how QQ Browser version 6.5.0.2170, which was released in response to Jeff's findings about the weak 128-bit RSA key, handles client encryption for messages sent to the server.

The major change was that the RSA implementation (and therefore also the client's public key and server's private key) were switched to a 1024-bit scheme. This puts factoring n and calculating the private key (based on reverse engineering the client binary) outside the resources of a graduate student. But, as mentioned before, the attack we'll carry out would work just as well for even a 4096-bit or larger RSA key.

QQ Browser's implementation of 1024-bit RSA was still textbook RSA. Basically, if a client wants to send an encrypted message to the server, it would first generate a random 128-bit AES key to be used as the session key. Then it would encrypt that 128-bit message using the 1024-bit RSA public key. After sending the RSA ciphtertext to the server, the client would also encrypt some data using that AES session key and also send that to the server. The server would decrypt the RSA ciphertext to recover the 128-bit AES session key, use that to decrypt the data from the client, and then send its own response to the client encrypted with the 128-bit session key.

Our attack model is simple: The NSA, or anyone with the ability to record encrypted messages that any client sends to the server, wants to decrypt those messages that they recorded. For example, the attacker could be sitting at the network gateway of a particular target whose data they'd like to record and decrypt. We know from the Snowden revelations that the NSA was doing exactly that for UC Browser [7].

0.4 The attack

Without the private key, d, that only the server has, we can't decrypt any of the messages, c, that clients encrypt and send to the server. And without a quantum computer there's not really a practical way to find out what d is. In fact, even in the attack that I'm about to describe we never find out what d is. But, if we can find a way to trick the server into decrypting c *or ciphertexts related to* c and then leaking to us bits of information about m *or plaintexts related to* m, then we don't need the private key. We're like a puppetmaster, pulling the strings of the server to get it to do things for us.

As happens in virtually any hack, we are going to program somebody else's code, based on our malicious inputs, to do things for us. Since the server will have access to decrypted plaintexts, we want to program the server to tell us something about those plaintexts. Using the malleability of RSA, it's possible for us to create plaintexts that are related to the original plaintext from the client and then learn something about our created plaintexts (which will tell us something about the original plaintext if we choose the related plaintexts carefully). Specifically, we can record a ciphertext that any client sends to the server and then double the plaintext that that ciphertext decrypts to, without really even knowing what that plaintext was. Recall that multiplying the ciphertext by 2^e doubles the plaintext. This is possible because QQ Browser uses the textbook version of RSA.

Suppose the client whose communications we want to decrypt encrypts the following 128-bit AES key with 1024-bit RSA and sends it to the server:

```
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

The server will decrypt the encrypted RSA ciphertext, c, that it receives into m. Since m is a 1024-bit number, the server is going to chop off the last 128-bits of m to use as the AES key. So the server will decrypt the following, where red are bits it will throw away and green are bits it will use as the AES key for encrypting its communications back to the client (and also for decrypting communications it receives from the client):

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

This shows what the server sees as the RSA plaintext *after* **it decrypts the ciphertext we sent it,** which we are trying to trick the server into leaking bits of to us. So, we as the attacker have recorded c by eavesdropping on the client and server's communications over the Internet. But without the private key, d, we don't know what m (the green part, which is the AES key to decrypt the rest of the message) is. Let's explore what happens if we open our own connection to the server, and as our ciphertext we send $c \times 2^e$. The server will decrypt that into the following plaintext:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
0110000100101100111011110111011001000101111111001110101011100110
0000000111001001011110010010101001000111011010100001011101110110
```

The above happens to be $2m$. The server will then throw out the red bits (which includes a bit from the original AES key) and then use the green part for AES encryption and decryption with us. So far, this doesn't help us that much, since we'd still need to brute force 2^{127} possibilities. Since m is a multiple of 2, we know that the last bit of the AES key (in green) is a zero since all even binary numbers end in 0, but there are still 127 bits we'd need to guess.

So, we know how to double plaintexts by manipulating ciphertexts. What if we double it more than once? What if we do it 16 times, by sending $c \times 2^{16e}$ as out ciphertext. When we multiplied the plaintext by 2 above, we effectively bit shifted the AES key by 1. Now we're multiplying the plaintext by 2^{16}, which is the equivalent of bit-shifting to the left 16 times:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000001011000010010110
0111011110111011001000101111111001110101011100110000000011100100
1011110010010101001000111011010100001011101110110000000000000000
```

Now there are $128 - 16 = 112$ bits in the AES key that we don't know for sure are zeroes, so 2^{112} possibilities that we'd need to brute force. You may be wondering at this point, how would we brute force the AES key? It's simple, the server will encrypt and decrypt with that key, so we just try all the possibilities until we decrypt a valid server response. To try a key we act as a new client and encrypt and decrypt with that key to see if the server can understand us and vice versa. But 2^{112} is still too many possibilities, we'd have to connect to the server to try a key 5 decillion 192 nonillion 296 octillion 858 septillion 534 sextillion 827 quintillion 628 quadrillion 530 trillion 496 billion 329 million 220 thousand and 96 times. So, let's instead send as our ciphertext $c \times 2^{127e}$, then the server will decrypt the following plaintext, and use the green part as the AES key (again, throwing away the red part):

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0101100001001011001110111101110110010001011111100111010101111001
1000000011100100101111001001010100100011101101010000101110111101
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Now, we've finally figured out **step one** of our attack. By sending $c \times 2^{127e}$ as our ciphertext, we've forced the server to encrypt what it sends back to us with one of two AES keys, either...

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

...or...

```
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Here's the insight: Which of those two keys the server encrypted with depends on the AES key chosen by the original client, specifically the most significant bit of our related plaintext is going to be the least significant bit of the original plaintext, m. So we just learned one bit of the original client's AES key! All we have to do is try both possibilities for encrypting a test message to the server and decrypting the server's response. That's step 1 out of 128 in cracking the AES session key and decrypting what that client sent to the server!

So, what's step 2? Well, we'll simply shift left by 126 instead of 127, by sending as our ciphertext $c \times 2^{126e}$. Then the server will decrypt the following for the session AES key (again, the red part is thrown out):

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0010110000100101100111011110111011001000101111110011101010111100
1100000000111001001011110010010101001000111011010100001011101110
1100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

So now there are two bits in our AES key with the server that we don't know for sure are zeroes, but we don't want to try all four possibilities because that's not going to be efficient going forward. We already know the bit in blue from step 1, so really there are only two possibilities of keys to try: Either...

```
0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

...or...

```
1100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

So then, for step 3, we would send $c \times 2^{125e}$ as our ciphertext, and the server would decrypt:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0001011000010010110011101111011101100100010111111001110101011110
0110000000011100100101111001001010100100011101101010000101110111
0110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Then the two possibilities we have to try for the AES key are either:

```
0110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

...or...

```
1110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

By step 128, we are sending c as our plaintext, but since we've inferred 127 bits by then (in steps 1 through 127) there are only two possibilities we need to try. The server will be using the same AES key as it used with the original client:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

Then we can try two keys, either:

```
0011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

...or...

```
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

The second possibility will be the one that decrypts the server response correctly, so now we know the original client's AES session key and can decrypt their communications with the server for that session.

To recap, let's go over a hypothetical attack scenario. Suppose the Thai equivalent of the NSA wants to spy on a reporter in Thailand who uses QQ Browser on a day-to-day basis. For any message between that user's QQ Browser client and the QQ Browser server that collects the records of their wherabouts, what websites they go to, *etc.*, the Thai government can record it at the country's Internet borders. Then they can decrypt the message by making 128 of their own connections to the QQ Browser server, each time learning one bit of the ephemeral AES key that was used for the original message. Once they know that 128-bit AES key, they can simply use it to decrypt the message.

0.5 Putting it all together

This type of attack has plagued SSL/TLS, the protocols that web browsers and web servers use for encryption on the Internet, for years (see, *e.g.*, [1, 4]). Many other protocols have fallen prey to Bleichenbacher-style attacks (see, *e.g.*, [3, 5, 8]). QQ Browser's use of textbook RSA made the example attack in this chapter easy to understand, but much more subtle bugs in well-implemented crypto implementations can be exploited using more sophisticated methods. These attacks are particularly insidious because the server's only fault is interacting with clients in seemingly normal ways, such as using the key the client chose to use or sending error messages when decryption doesn't work out. The attacker is taking normal things that normal servers do, and manipulating them into a carefully sequenced saraband where the strings the attacker is pulling are the strings that have always been there by design, the same strings that all clients pull, but the actions the "puppet" (*i.e.*, the server) are taking are extremely malicious and not at all what the server's designers intended.

References

[1] Aviram *et al.* DROWN: Breaking TLS using SSLv2. Available at `https://drownattack.com/drown-attack-paper.pdf`.

[2] M. Bellare and P. Rogaway. Optimal asymmetric encryption–How to encrypt with RSA. Advances in Cryptology–EUROCRYPT'94, 1994.

[3] D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 1–12, London, UK, UK, 1998. Springer-Verlag.

[4] H. Bck, J. Somorovsky, and C. Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). Cryptology ePrint Archive, Report 2017/1189, 2017. `https://eprint.iacr.org/2017/1189`.

[5] T. Duong and J. Rizzo. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 481–489, Washington, DC, USA, 2011. IEEE Computer Society.

[6] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[7] A. Hildebrandt and D. Seglins. Spy agencies target mobile phones, app stores to implant spyware. CBC News, `http://www.cbc.ca/news/canada/spy-agencies-target-mobile-phones-app-stores-to-implant-spyware-1.3076546`.

[8] T. Jager and J. Somorovsky. How to break XML encryption. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 413–422, 2011.