Hey ! I'm **David**, a security consultant at Cryptography Services (https://cryptoservices.github.io/), the crypto team of **NCC Group** . This is my blog about **cryptography** and **security** and other related topics that I find interesting.

Latest from the Links section (all links (/links))

Why iOS (/links/link/1092#disqus_thread)

Verifiable Random Functions (/links/link/1091#disqus_thread)

Write and Alice and Bob protocol, and get it translated into a Tamarin's input (/links/link/1090#disqus_thread)

Improving by simplifying the GnuTLS PRNG (/links/link/1089#disqus_thread)

X25519 and zero outputs (/links/link/1088#disqus_thread)

Popular articles (most read articles (/home/most))

Key Compromise Impersonation attacks (KCI) (/article/372/key-compromise-impersonation-attacks-kci/)

Babun, Cmder and Tmux (/article/223/babun-cmder-and-tmux/)

TLS, Pre-Master Secrets and Master Secrets (/article/340/tls-pre-master-secrets-and-master-secrets/)

The Logjam Attack (/article/270/the-logjam-attack/)

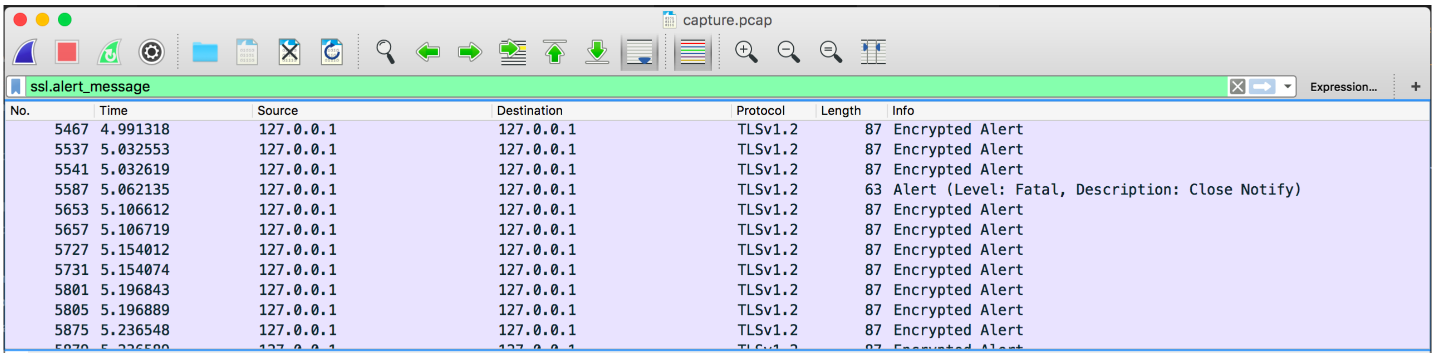
Schnorr's Signature and non-interactive Protocols (/article/193/schnorrs-signature-and-non-interactive-protocols/)

Fault attacks on RSA's signatures September 2016

Facebook was organizing a CTF last week (http://gsec.hitb.org/sg2016/facebook-capture-the-flag/) and they needed some crypto challenge. I obliged, missed a connecting flight in Phoenix while building it, and eventually provided them with one idea I had wanted to try for quite some time. Unfortunately, as with the last challenge I wrote for a CTF (https://www.cryptologie.net/article/271/hacking-week-2015-crypto-4-write-up/), someone solved it with a tool (https://github.com/zack-lau/WriteUps/blob/master/HITB-Facebook-CTF-2016/capture_Mexico-tls/README.md) instead of doing it by hand (like in the good ol' days). You can read the quick write up there, or you can read a more involved one here.

The challenge

The challenge was just a file named `capture.pcap`. Opening it with Wireshark would reveal hundreds of TLS handshakes. One clever way to find a clue here would be to filter them with `ssl.alert_message`.



```
ssl.alert_message

No.   Time      Source     Destination  Protocol  Length  Info
 5467 4.991318  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5537 5.032553  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5541 5.032619  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5587 5.062135  127.0.0.1  127.0.0.1    TLSv1.2       63  Alert (Level: Fatal, Description: Close Notify)
 5653 5.106612  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5657 5.106719  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5727 5.154012  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5731 5.154074  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5801 5.196843  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5805 5.196889  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
 5875 5.236548  127.0.0.1  127.0.0.1    TLSv1.2       87  Encrypted Alert
```

From that we could observe a **fatal alert** being sent from the client to the server, right after the `server Hello Done`.

Mmmm

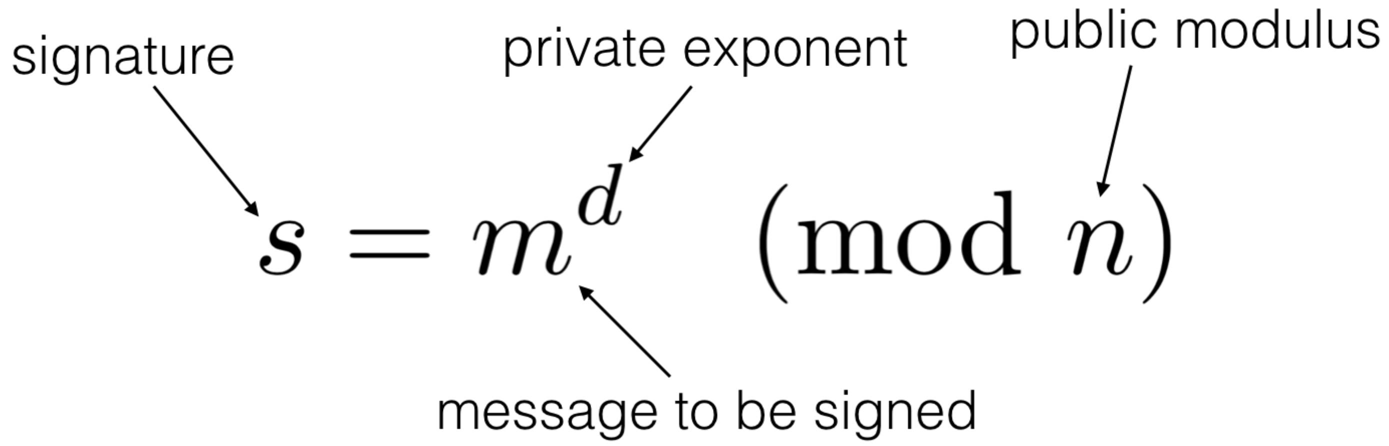
Several hypothesis exist. One way of guessing what went wrong could be to run these packets to `openssl s_client` with a `-debug` option and see why the client decides to terminate the connection at this point of the handshake. Or if you had good intuition, **you could have directly verified the signature** :)

After realizing the **signature was incorrect**, and that it was done with **RSA**, one of the obvious attack here is the **RSA-CRT attack**! Faults happen in RSA, sometimes because of malicious reasons (lasers!) or just because of random errors that can happen in different parts of the hardware. One random bit shifting and you have a fault. If it happens at the wrong place, at the wrong time, you have a cryptographic failure!

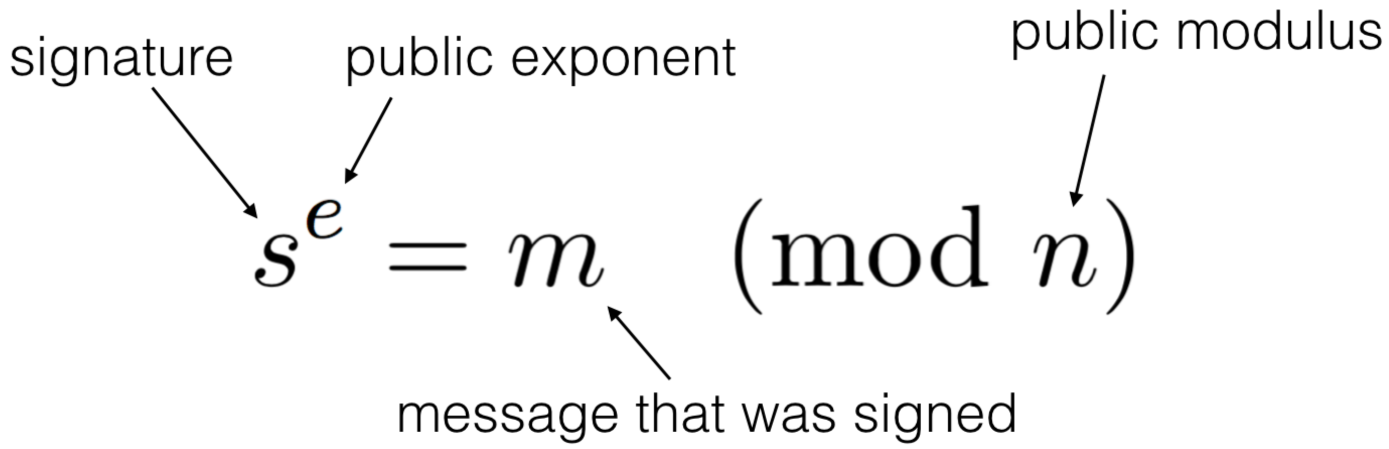
RSA-CRT

RSA is slow-ish, as in not as fast as symmetric crypto: I can still do 414 signatures per second and verify 15775 signatures per second (according to `openssl speed rsa2048`).

Let's remember a RSA signature. It's basically the inverse of an encryption with RSA: you decrypt your message and use the decrypted part as a signature.



To verify a signature over a message, you do the same kind of computation on the signature using the **public exponent**, which gives you back the message:



We remember here that N is the public modulus used in both the signing and verifying operations. Let $N = pq$ with p, q two large primes.

This is the basis of RSA. Its security relies on **the hardness to factor N**.

I won't talk more about RSA here, so check Wikipedia (https://en.wikipedia.org/wiki/RSA_(cryptosystem)) if you need a recap =)

It's also obvious for a lot of you reading this that **you do not sign the message directly**. You first hash it (this is good especially for large files) and pad it according to some specifications. I will talk about that in the later sections, as this distinction is not immediately important to us. We have now enough background to talk about *The Chinese Remainder Theorem* (CRT), which is a theorem we use to **speed up** the above equation.

So what if we could do the calculation mod p and q instead of this huge number N (usually 2048 bits)? Let's stop with the what ifs because this is exactly what we will do:

$$\left.\begin{array}{ll} s_1 = m^{d_p} & \pmod{p} \\ s_2 = m^{d_q} & \pmod{q} \end{array}\right\} \implies s \pmod{pq}$$

Here we compute two partial signatures, one mod p, one mod q. With $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$. After that, we can use CRT to stich these partial signatures together to obtain the complete one.

I won't go further, if you want to know how CRT works you can check an explanation in my latest paper (http://eprint.iacr.org/2016/644).

RSA-CRT fault attack

Now imagine that a fault happens in one of the equation mod p or q:

$$\left.\begin{array}{ll} s_1 = m^{d_p} & \pmod{p} \\ \widetilde{s_2} = m^{d_q} & \pmod{q} \end{array}\right\} \implies \widetilde{s} \pmod{pq}$$

Here, because one of the operation failed ($\widetilde{s_2}$) we obtain a faulty signature $\tilde{s}$. What can we do with a faulty signature you may ask? We first observe the following facts on the faulty signature.

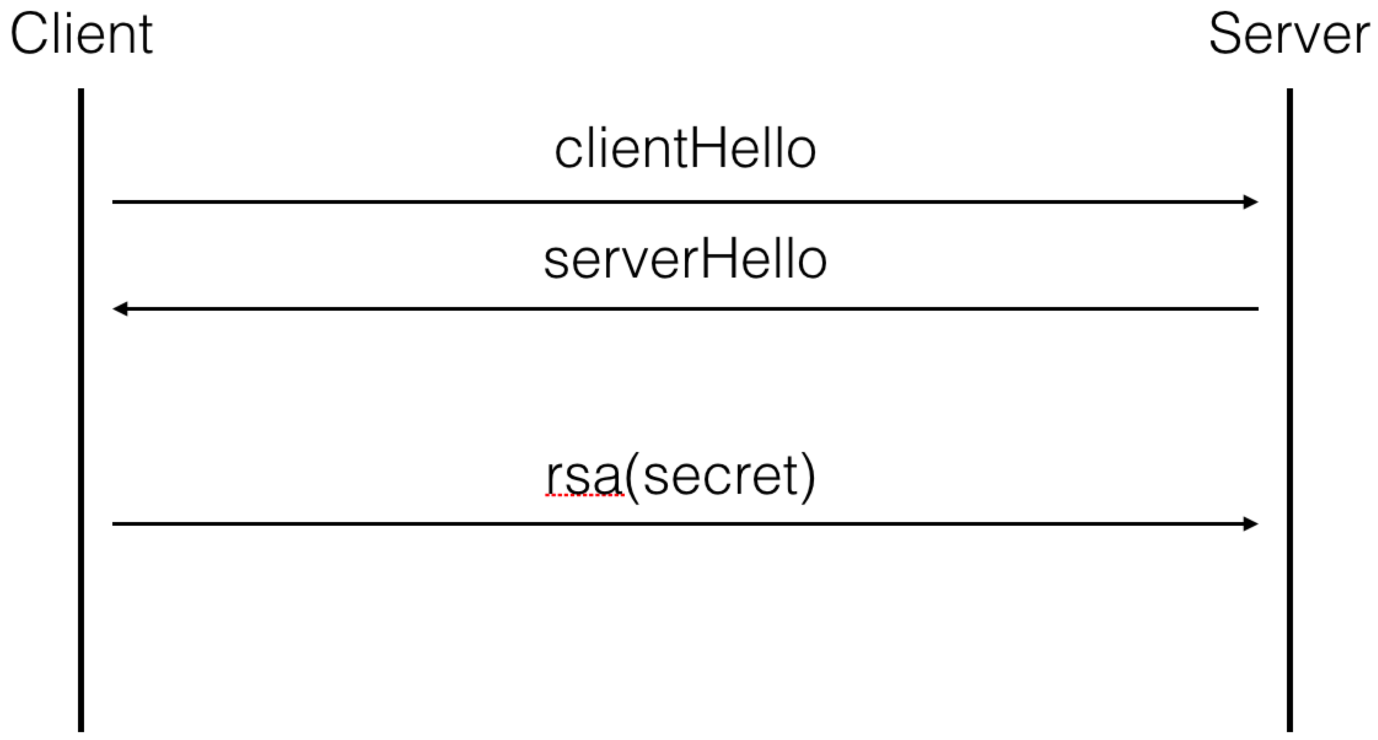
$$\begin{cases} \widetilde{s}^e = m & \pmod{p} \\ \widetilde{s}^e \neq m & \pmod{q} \end{cases} \implies \begin{cases} \widetilde{s}^e - m = 0 & \pmod{p} \\ \widetilde{s}^e - m \neq 0 & \pmod{q} \end{cases} \implies \begin{cases} p \mid \widetilde{s}^e - m \\ q \nmid \widetilde{s}^e - m \end{cases}$$

See that? p divides this value (that we can calculate since we know both the faulty signature, the public exponent e, and the message). But q does not divide this value. This means that $\tilde{s}^e - m$ is of the form pk with some integer k. What follows is naturally that $gcd(\tilde{s}^e - m, N)$ will give out p! And as I said earlier, if you know the factorization of the public modulus N then it is **game over**.

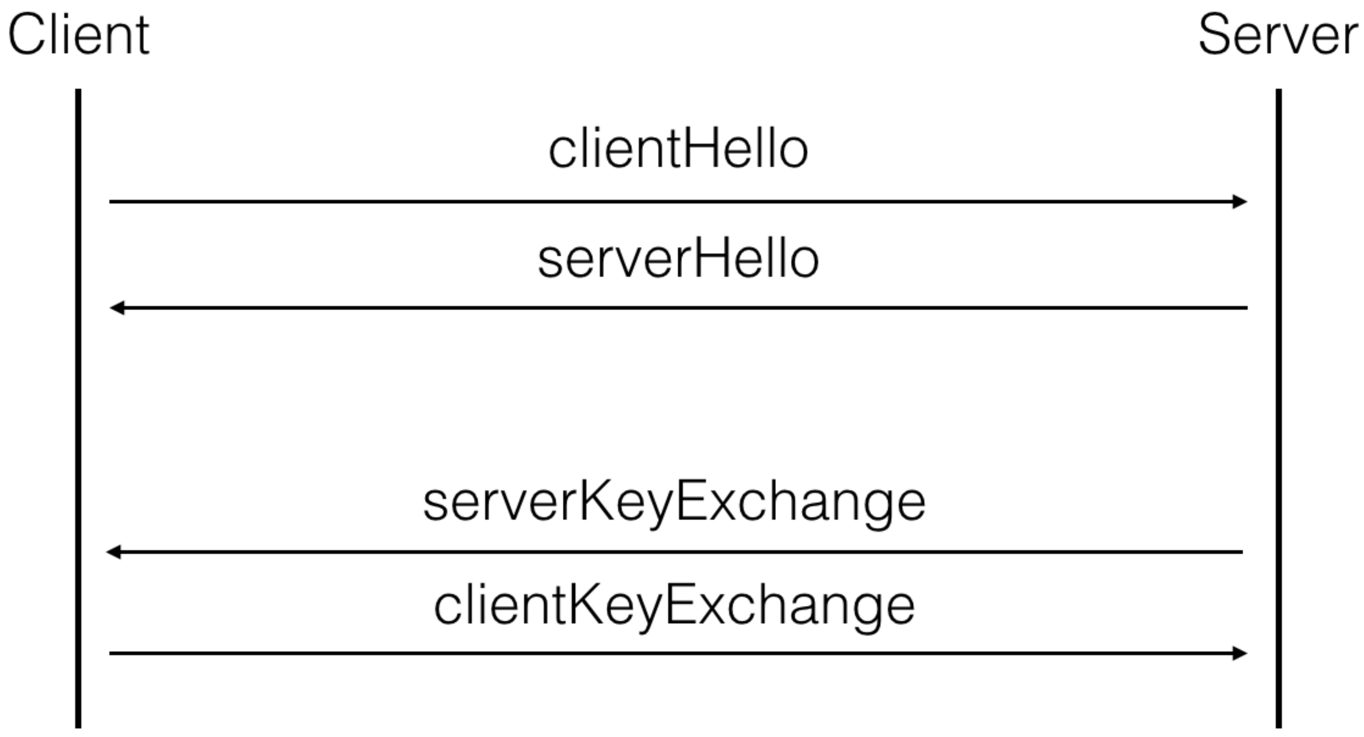
Applying the RSA-CRT attack

Now that we got that out of the way, how do we apply the attack on TLS?

TLS has different kind of **key exchanges**, some basic ones and some ephemeral (forward secure) ones. The **basic key exchange** we've used a lot in the past is pretty straight forward: **the client uses the RSA public key found in the server's certificate to encrypt the shared secret with it.** Then both parties derive the session keys out of that shared secret



Now, if the client and the server agree to do a **forward-secure key exchange**, they will use something like Diffie-Hellman or Elliptic Curve Diffie-Hellman and **the server will sign his ephemeral (EC)DH public key with his long term key.** In our case, his public key is a RSA key, and the fault happens during that particular signature.



Now what's the message being signed? We need to check TLS 1.2's RFC (https://tools.ietf.org/html/rfc5246#section-7.4.3):

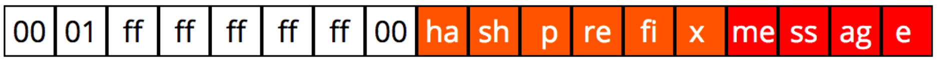
```
struct {
    select (KeyExchangeAlgorithm) {
        ...
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHParams params;
            } signed_params;
            ...
} ServerKeyExchange;
```

- the `client_random` can be found in the *client hello* message
- the `server_random` in the *server hello* message
- the `ServerDHParams` are the different parameters of the server's ephemeral public key, from the same TLS 1.2 RFC (https://tools.ietf.org/html/rfc5246#section-7.4.3):

```
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;     /* Ephemeral DH parameters */

dh_p
   The prime modulus used for the Diffie-Hellman operation.

dh_g
   The generator used for the Diffie-Hellman operation.

dh_Ys
   The server's Diffie-Hellman public value (g^X mod p).
```

TLS is old, they use a non provably secure scheme to sign: **PKCS#1 v1.5**. Instead they should be using RSA-PSS (https://www.emc.com/emc-plus/rsa-labs/historical/raising-standard-rsa-signatures-rsa-pss.htm) but it's a whole different story :)

PKCS#1 v1.5's padding is pretty straight forward:



- The `ff` part shall be long enough to make the bitsize of that padded message as long as the bitsize of N
- The hash prefix part is a hexstring representing the hash function being used to sign

And here's the sage code!

```
# hash the signed_params

h = hashlib.sha384()
h.update(client_nonce.decode('hex'))
h.update(server_nonce.decode('hex'))
h.update(server_params.decode('hex'))
hashed_m = h.hexdigest()

# PKCS#1 v1.5 padding
prefix_sha384 = "3041300d060960864801650304020205000430"

modulus_len = (len(bin(modulus)) - 2 + 7) // 8
pad_len = len(hex(modulus))//2 - 3 - len(hashed_m)//2 - len(prefix_sha384)//2

padded_m = "0001" + "ff" * pad_len + "00" + prefix_sha384 + hashed_m

# Attack to recover p
p = gcd(signature^public_exponent - int(padded_m, 16), modulus)

# recover private key
q = modulus // p
phi = (p-1) * (q-1)

privkey = inverse_mod(public_exponent, phi)
```

What now?

Now what? You have a private key, but that's not the flag we're looking for... After a bit of inspection you realize that the last handshake made in our `capture.pcap` file has a different key exchange: a **RSA key exchange**!!!

What follows is then pretty simple, Wireshark can decrypt conversations for you, you just need a private key file. From the previous section we retrieved the private key, to make a `.pem` file out of it (see this article to know what a .pem is (https://www.cryptologie.net/article/260/asn1-vs-der-vs-pem-vs-x509-vs-pkcs7-vs/)) you can use rsatool (https://github.com/ius/rsatool).

Tada! Hope you enjoyed the write up =)

Well done! You've reached the end of my post. Now you can leave me a comment :)

ddddavidee

Let's remember a RSA signature. It's basically the inverse of an encryption with RSA: you decrypt your message and use the *encrypted* part as a signature.
Should read "decrypted".

david

thanks :D

thang

Can you upload the pcap file?

Your name

Enter name

Do you have a homepage?

This information will be displayed

Capital of France? (antispam)

Enter antispam