The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations

Jan 17, 2019 • David Wong

On May 15th, I approached Yuval Yarom with a few issues I had found in some TLS implementations. This led to a collaboration between Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, Yuval Yarom and I. Spearheaded by Eyal, the research has now been published here. And as you can see, the inventor of RSA himself is now recommending you to deprecate RSA in TLS.

We tested 9 different TLS implementations against cache attacks and 7 were found to be vulnerable: OpenSSL, Amazon s2n, MbedTLS, Apple CoreTLS, Mozilla NSS, WolfSSL, and GnuTLS. The cat is not dead yet, with two lives remaining thanks to BearSSL (developed by my colleague Thomas Pornin) and Google's BoringSSL.

The issues were disclosed back in August and the teams behind these projects were given time to resolve them. Everyone was very cooperative and CVEs have been dispatched (CVE-2018-12404, CVE-2018-19608, CVE-2018-16868, CVE-2018-16869, CVE-2018-16870).

The attack leverages a side-channel leak via cache access timings of these implementations in order to break the RSA key exchanges of TLS implementations. The attack is interesting from multiple points of view (besides the fact that it affects many major TLS implementations):

- **It affects all versions of TLS (including TLS 1.3) and QUIC**. Where the latter version of TLS does not even offer an RSA key exchange! This prowess is achieve because of the only known downgrade attack on TLS 1.3.
- **It uses state-of-the-art cache attack techniques**. Flush+Reload? Prime+Probe? Branch-Predition? We have it.
- **The attack is very efficient**. We found ways to ACTIVELY target any browsers, slow some of them down, or use the long tail distribution to repeatdly try to break a session. We even make use of lattices to speed up the problem.
- **Manger and Ben-Or on RSA PKCS#1 v1.5**. You heard of Bleichenbacher's million messages attack? Guess what, we found better. We use Manger's OAEP attack on RSA PKCS#1 v1.5 and even Ben-Or's algorithm which is more efficient than and was published BEFORE Bleichenbacher's work in 1998. I uploaded some of the code here.

To learn more about the research, you should read the white paper. I will talk specifically about protocol-level exploitation in this blog post.

Attacking RSA, The Origins

While Ben-Or et al. research was initially used to support the security proofs of RSA, it was none-the-less already enough to attack the protocol. But it is only in 1998 that Daniel Bleichenbacher discovers a padding oracle and devise his own practical attack on RSA. The consequences are severe, most TLS implementations could be broken, thus mitigations were designed to prevent Daniel's attack. Follows a series of "re-discovery" where the world realizes that it is not so easy to implement such mitigations:

- Bleichenbacher (CRYPTO 1998) also called the 1 million message attack, BB98, padding oracle attack on PKCS#1 v1.5, etc.
- Klima et al. (CHES 2003)
- Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 V1.5 In Xml Encryption (ESORICS 2012)
- Degabriele et al. (CT-RSA 2012)
- Bardou et al. (CRYPTO 2012)
- Cross-Tenant Side-Channel Attacks in PaaS Clouds (CCS 2014)
- Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks (USENIX 2014)
- On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption (CCS 2015)
- DROWN (USENIX 2016)
- Return Of Bleichenbacher's Oracle Threat (USENIX 2018)
- The Dangers of Key Reuse: Practical Attacks on IPsec IKE (USENIX 2018)

Let's be realistic, **the mitigations that developers had to implement were unrealistic**. Furthermore, an implementation that would attempt to log such attacks would actually help the attacks. Isn't that funny?

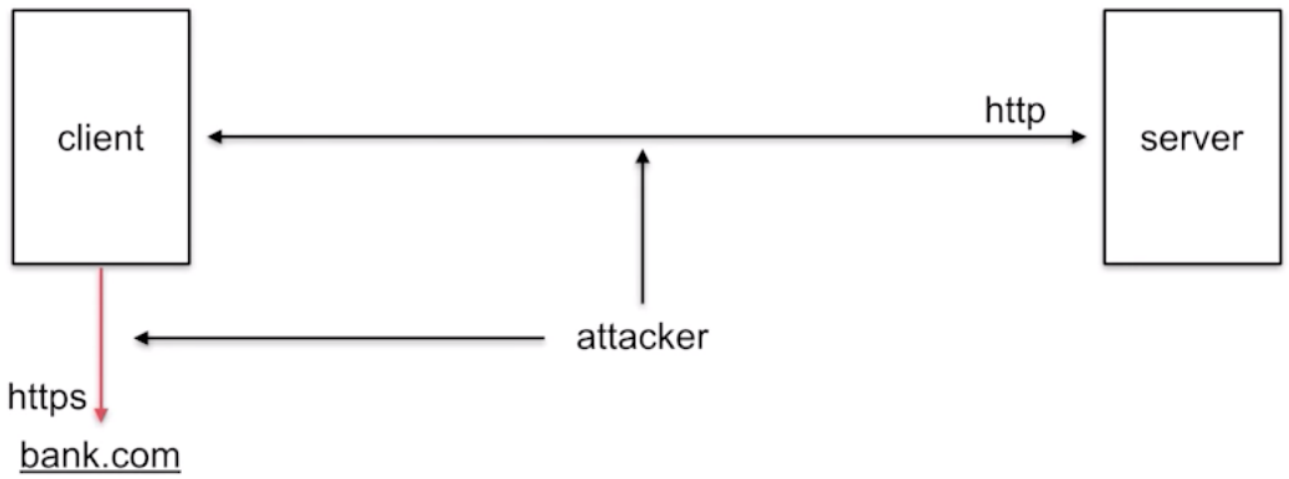
The research I'm talking about today can be seen as one more of these "re-discovery". My previous boss' boss (Scott Stender) once told me: "you can either be the first paper on the subject, or the best written one, or the last one". We're definitely not the first one, I don't know if we write that well, but we sure are hopping to be the last one :)

RSA in TLS?

Briefly, SSL/TLS (except TLS 1.3) can use an RSA key exchange during a handshake to negotiate a shared secret. An RSA key exchange is pretty straight forward: the client encrypts a shared secret under the server's RSA public key, then the server receives it and decrypts it. If we can use our attack to decrypt this value, we can then passively decrypt the session (and obtain a cookie for example) or we can actively impersonate one of the peer.

Attacking Browsers, In Practice.

We employ the BEAST-attack model (in addition to being colocated with the victim server for the cache attack) which I have previously explained in a video here.



With this position, we then attempt to decrypt the session between a victim client (Bob) and `bank.com`: we can serve him with some javascript content that will continuously attempt new connections on `bank.com`. (If it doesn't attempt a new connection, we can force it by making the current one fail since we're in the middle.)

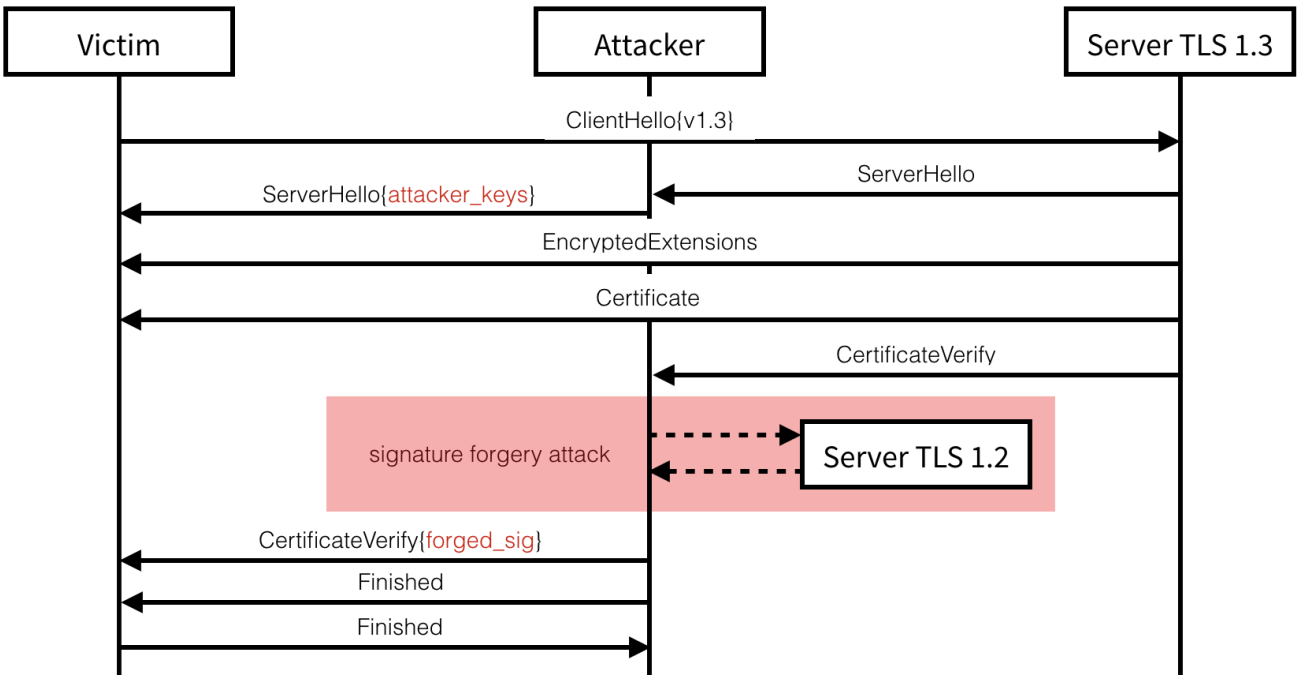
Why several connections instead of just one? Because most browsers (except Firefox which we can fool) will time out after some time (usually 30s). If an RSA key exchange is negotiated between the two peers: it's great, we have all the time in the world to passively attack the protocol. If an RSA key exchange is NOT negotiated: we need to actively attack the session to either downgrade or fake the server's RSA signature (more on that later). This takes time, because the attack requires us to send thousands of messages to the server. It will likely fail. But if we can try again, many times? It will likely succeed after a few trials. And that is why we continuously send connection attempts to `bank.com`.

Attacking TLS 1.3

There exist two ways to attack TLS 1.3. In each attack, the server needs to support an older version of the protocol as well.

1. The first technique relies on the fact that the current server's public key is an RSA public key, used to sign its ephemeral keys during the handshake, and that the older version of TLS that the server supports re-use the same keys.
2. The second one relies on the fact that both peers support an older version of TLS with a cipher suite supporting an RSA key exchange.

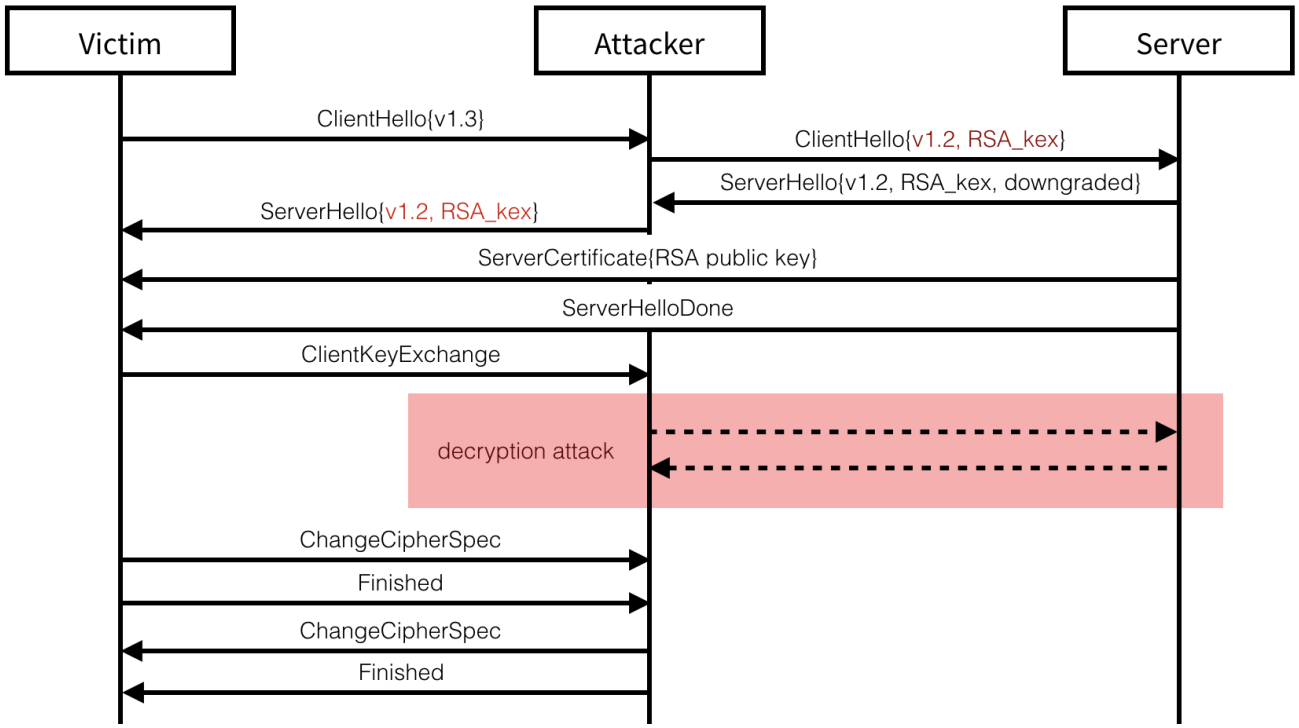
While TLS 1.3 does not use the RSA encryption algorithm for its key exchanges, it does use the signature algorithm of RSA for it; if the server's certificate contain an RSA public key, it will be used to sign its ephemeral public keys during the handshake. A TLS 1.3 client can advertise which RSA signature algorithm it wants to support (if any) between RSA and RSA-PSS. As most TLS 1.2 servers already provide support for RSA , most will re-use their certificates instead of updating to the more recent RSA-PSS. RSA digital signatures specified per the standard are really close to the RSA encryption algorithm specified by the same document, so close that Bleichenbacher's decryption attack on RSA encryption also works to forge RSA signatures. Intuitivelly, we have pms^e and the decryption attack allows us to find $(pms^e)^d = pms$, for forging signatures we can pretend that the content to be signed tbs (see RFC 8446) is $tbs = pms^e$ and obtain tbs^d via the attack, which is by definition the signature over the message tbs. However, this signature forgery requires an additional step (blinding) in the conventional Bleichenbacher attack (in practice this can lead to hundreds of thousands of additional messages).



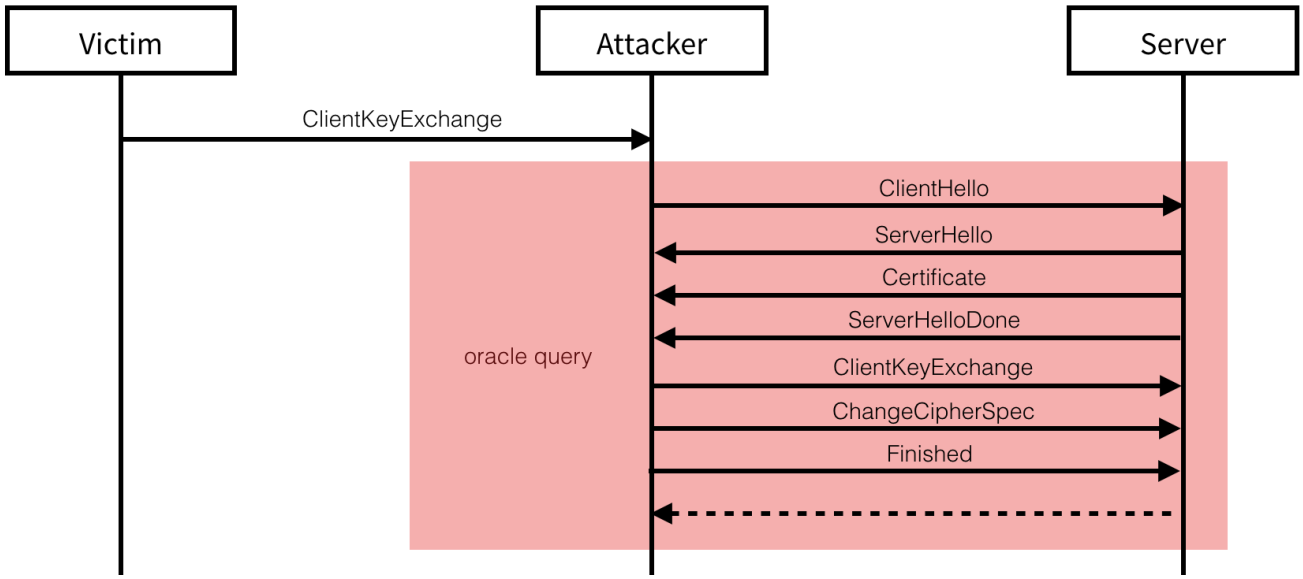
Key-Reuse has been shown in the past to allow for complex cross-protocol attacks on TLS. Indeed, we can successfully forge our own signature of the handshake transcript (contained in the `CertificateVerify` message) by negotiating a previous version of TLS with the same server. The attack can be carried if this new connection exposes a length or bleichenbacher oracle with a certificate using the same RSA key but for key exchanges.

Downgrading to TLS 1.2

Every TLS connection starts with a negotiation of the TLS version and other connection attributes. As the new version of TLS (1.3) does not offer an RSA key exchange, the exploitation of our attack must first begin with a downgrade to an older version of TLS. TLS 1.3 being relatively recent (August 2018), most servers supporting it will also support older versions of TLS (which all provide support for RSA key exchanges). A server not supporting TLS 1.3 would thus respond with an older TLS version's (TLS 1.2 in our example) server hello message. To downgrade a client's connection attempt, we can simply spoof this answer from the server. Besides protocol downgrades, other techniques exist to force browser clients to fallback onto older TLS versions: network glitches, a spoofed TCP RST packet, a lack of response, etc. (see POODLE)



Continuing with a spoofed TLS 1.2 handshake, we can simply present the server's RSA certificate in a `ServerCertificate` message and then end the handshake with a `ServerHelloDone` message. At this point, if the server does not have a trusted certificate allowing for RSA key exchanges, or if the client refuse to support RSA key exchanges or older versions than TLS 1.2, the attack is stopped. Otherwise, the client uses the RSA public key contained in the certificate to encrypt the TLS premaster secret, sends it in a `ClientKeyExchange` message and ends its part of the handshake with a `ChangeCipherSpec` and a `Finished` messages.



At this point, we need to perform our attack in order to decrypt the RSA encrypted premaster secret. The last Finished message that we send must contains an authentication tag (with HMAC) of the whole transcript, in addition of being encrypted with the transport keys derived from the premaster secret. While some clients will have no handshake timeouts, most serious applications like browsers will give up on the connection attempt if our response takes too much time to arrive. While the attack only takes a few thousand of messages, this might still be too much in practice. Fortunately, several techniques exist to slow down the handshake:

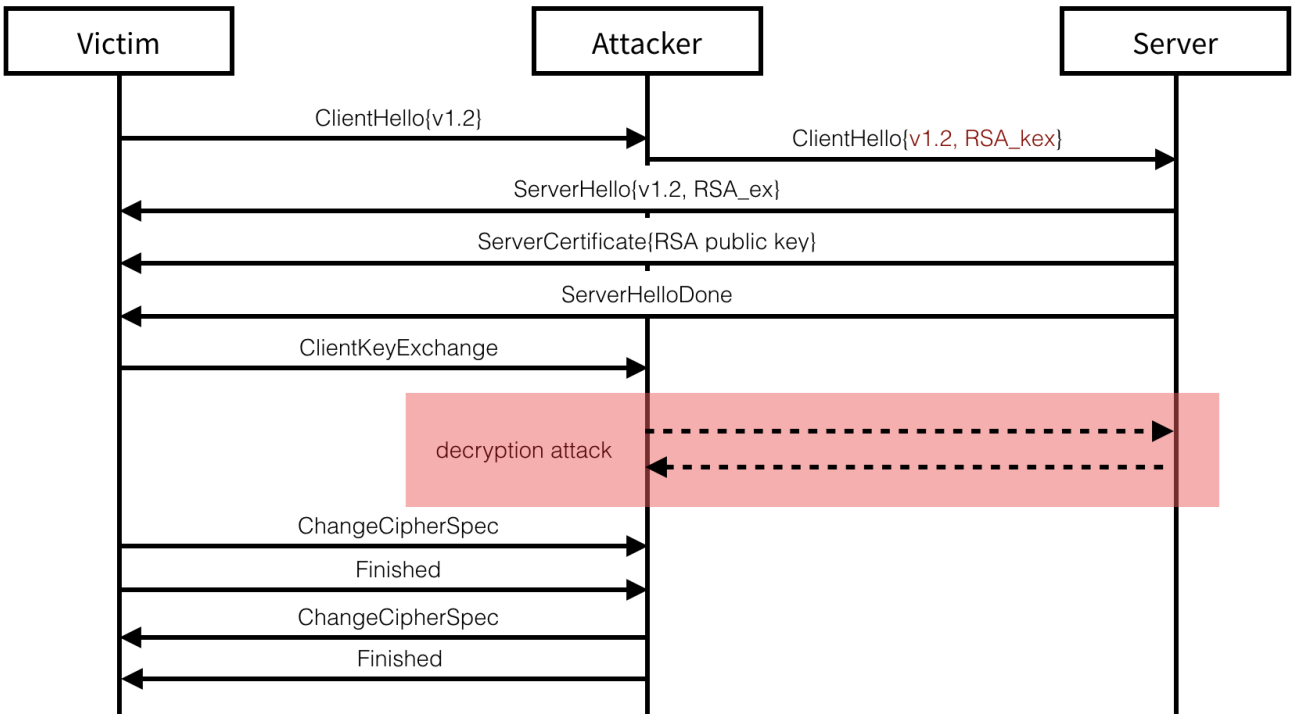
- we can send the ChangeCipherSpec message which might reset the client's timer
- we can send TLS warning alerts to reset the handshake timer

Once the decryption attack terminates, we can send the expected Finished message to the client and finalize the handshake. From there everything is possible, from passively relaying and observing messages to the impersonated server through to actively tampering requests made to it. This downgrade attack bypasses multiple downgrade mitigations: one server-side and two client-side. TLS 1.3 servers that negotiate older versions of TLS must advertise this information to their peers. This is done by setting a quarter of the bytes from the `server_random` field in the `ServerHello` message to a known value. TLS 1.3 clients that end up negotiating an older version of TLS must check for these values and abort the handshake if found. But as noted by the RFC, "It does not provide downgrade protection when static RSA is used." – Since we alter this value to remove the warning bytes, the client has no opportunity to detect our attack. On the other hand, a TLS 1.3 client that ends up falling back to an older version of TLS must advertise this information in their subsequent client hellos, since we impersonate the server we can simply ignore this warning. Furthermore, a client also includes the version used by the client hello inside of the encrypted premaster secret. For the same reason as previously, this mitigation has no effect on our attack. As it stands, RSA is the only known downgrade attack on TLS 1.3, which we are the first to successfuly exploit in this research.

Attacking TLS 1.2

As with the previous attack, both the client and the server targetted need to support RSA key exchanges. As this is a typical key exchange most known browsers and servers support them, although they will often prefer to negotiate a forward secret key exchange based on ephemeral

versions of the Elliptic Curve or Finite Field Diffie-Hellman key exchanges. This is done as part of the cipher suite negotiation during the first two handshake messages. To avoid this outcome, the `ClientHello` message can be intercepted and modified to strip it out of any non-RSA key exchanges advertised. The server will then only choose from a set of RSA-key-exchange-based cipher suites which will allow us to perform the same attack as previously discussed. Our modification of the `ClientHello` message can only be detected with the `Finished` message authenticating the correct handshake transcript, but since we are in control of this message we can forge the expected tag.



On the other side, if both peers end up negotiating an RSA key exchange on their own, we can passively observe the connection and take our time to break the session.

Cryptography Services

Cryptography Services is a dedicated team of consultants from NCC Group focused on cryptographic security assessments, protocol and design reviews, and tracking impactful developments in the space of academia and industry.