http://www.coincoin.fr.eu.org/?Attack-of-the-week-DROWN

Attack of the week : DROWN

- 6- Webographie -

Date de mise en ligne : mardi 1er mars 2016

https://3.bp.blogspot.com/-lS6YPu_k...
[PNG - 32.4Â kio]

To every thing there is a season. And in the world of cryptography, today we have the first signs of the season of TLS vulnerabilities.

This year's season is off to a roaring start with not one, but two serious bugs announcements by the OpenSSL project, each of which guarantees that your TLS connections are much less than private than you'd like them to be. I can't talk about both vulnerabilities and keep my sanity, so today I'm going to confine myself to the more dramatic of the two vulnerabilities : a new cross-protocol attack on TLS named "DROWN" .

Technically DROWN stands for "**D**ecrypting **R**SA using **O**bsolete and **W**eakened e**N**cryption", but honestly feel free to forget that because the name itself is plenty descriptive. In short, due to a series of dumb mistakes on the part of a vast number of people, DROWN means that TLS connections to a *depressingly huge slice of the web* (and mail servers, VPNs etc.) are essentially open to attack by fairly modest adversaries.

So that's bad news. The worse news â€” as I'll explain below â€” is that this whole mess was mostly avoidable.

For a detailed technical explanation of DROWN, you should go read the complete technical paper by Aviram *et al*. Or visit the DROWN team's excellent website . If that doesn't appeal to you, read on for a high level explanation of what DROWN is all about, and what it means for the security of the web. Here's the TL ;DR :

*If you're running a web server configured to use SSLv2, and particularly one that's running OpenSSL (even with all SSLv2 ciphers disabled !), you may be vulnerable to a fast attack that decrypts many recorded TLS connections made to that box. Most worryingly, the attack does not require the client to ever make an SSLv2 connection itself, and it isn't a downgrade attack. Instead, it relies on the fact that SSLv2 â€” and particularly the legacy "export" ciphersuites it incorporates â€” are pure poison, and simply having these active on a server is enough to invalidate the security of all connections made to that device.*For the rest of this post I'll use the "fun" question and answer format I save for this kind of attack. First, some background.

What are TLS and SSLv2, and why should I care ?Transport Layer Security (TLS) is the most important security protocol on the Internet. You should care about it because nearly every transaction you conduct on the Internet relies on TLS (v1.0, 1.1 or 1.2) to some degree, and failures in TLS can flat out ruin your day.

But TLS wasn't always TLS. The protocol began its life at Netscape Communications under the name "Secure Sockets Layer", or SSL. Rumor has it that the first version of SSL was so awful that the protocol designers collected every printed copy and buried them in a secret New Mexico landfill site. As a consequence, the first *public* version of SSL is actually SSL version 2 . It's pretty terrible as well â€” but not (entirely) for the reasons you might think.

Let me explain.

| https://3.bp.blogspot.com/-lSCPG7g2...
[JPEG - 24.5Â kio]

| Working group last call, SSL version 2.The reason you might *think* SSLv2 is terrible is because it was a product of the mid-1990s, which modern cryptographers view as the "dark ages of cryptography" . Many of the nastier cryptographic attacks we know about today had not yet been discovered. As a result, the SSLv2 protocol designers

were forced to essentially grope their way in the dark, and so were frequently devoured by grues â€” to their chagrin and our benefit, since the attacks on SSLv2 offered priceless lessons for the next generation of protocols.

And yet, these honest mistakes are *not* worst thing about SSLv2. The most truly awful bits stem from the fact that the SSLv2 designers were forced to *ruin their own protocol.* This was the result of needing to satisfy the U.S. government's misguided attempt to control the export of cryptography . Rather than using only secure encryption, the designers were forced to build in a series of "export-grade ciphersuites" that offered abysmal 40-bit session keys and other nonsense. I've previously written about the effect of export crypto on today's security . Today we'll have another lesson.

Wait, isn't SSLv2 ancient history ?For some time in the early 2000s, SSLv2 was still supported by browsers as a fallback protocol, which meant that active attackers could *downgrade* an SSLv3 or TLS connection by tricking a browser into using the older protocol. Fortunately those attacks are long gone now : modern web browsers have banished SSLv2 entirely â€” along with export cryptography in general. If you're using a recent version of Chrome, IE or Safari, you should never have to worry about accidentally making an SSLv2 connection.

The problem is that while *clients* (such as browsers) have done away with SSLv2, many *servers* still support the protocol. In most cases this is the result of careless server configuration. In others, the blame lies with crummy and obsolete embedded devices that haven't seen a software update in years â€” and probably never will. (You can see if your server is vulnerable here .)

And then there's the special case of OpenSSL, which helpfully provides a configuration option that's intended to disable SSLv2 ciphersuites â€” but which, unfortunately, does no such thing. In the course of their work, the DROWN researchers discovered that even when this option is set, clients may still request arbitrary SSLv2 ciphersuites. (This issue was quietly patched in January. Upgrade.)

The reason this matters is that SSL/TLS servers do a very silly thing. You see, since people don't like to buy multiple certificates, a server that's configured to use both TLS and SSLv2 will generally use *the same RSA private key to support both protocols*. This means any bugs in the way SSLv2 handles that private key could very well affect the security of TLS.

And this is where DROWN comes in.

So what is DROWN ?DROWN is a classic example of a "cross protocol attack". This type of attack makes use of bugs in one protocol implementation (SSLv2) to attack the security of connections made under a different protocol entirely â€” in this case, TLS. More concretely, DROWN is based on the critical observation that while SSLv2 and TLS both support RSA encryption , TLS properly defends against certain well-known attacks on this encryption â€” *while SSLv2's export suites emphatically do not*.

I will try to make this as painless as possible, but here we need to dive briefly into the weeds.

You see, both SSLv2 and TLS use a form of RSA encryption padding known as RSA-PKCS#1v1.5. In the late 1990s, a man named Daniel Bleichenbacher proposed an amazing attack on this encryption scheme that allows an attacker to decrypt an RSA ciphertext efficiently â€” under the sole condition that they can ask an online server to decrypt many *related* ciphertexts, and give back only one bit of information for each one â€” namely, the bit representing *whether decryption was successful or not*.

Bleichenbacher's attack proved particularly devastating for SSL servers, since the standard SSL RSA-based handshake involves the client encrypting a secret (called the Pre-Master Secret, or PMS) under the server's RSA public key, and then sending this value over the wire. An attacker who eavesdrops the encrypted PMS can run the Bleichenbacher attack against the server, sending it thousands of related values (in the guise of new SSL connections), and using the server's error responses to gradually decrypt the PMS itself. With this value in hand, the attacker can compute SSL session keys and decrypt the recorded SSL session.

| https://1.bp.blogspot.com/-EAqqU1Sm...
[JPEG - 21.3Â kio]

| A nice diagram of the SSL RSA handshake, courtesy Cloudflare (who don't know I'm using it, thanks guys !)The main SSL/TLS countermeasure against Bleichenbacher's attack is basically a hack. When the server detects that an RSA ciphertext has decrypted improperly, *it lies.* Instead of returning an error, which the attacker could use to implement the attack, it generates a *random* pre-master secret and continues with the rest of the protocol as though this bogus value was what it actually decrypted. This causes the protocol to break down later on down the line, since the server will compute essentially a random session key. But it's sufficient to prevent the attacker from learning whether the RSA decryption succeeded or not, and that kills the attack dead.

| https://4.bp.blogspot.com/-RdERzk3b...
[PNG - 26Â kio]

| Anti-Bleichenbacher countermeasure from the TLS 1.2 spec. Now let's take a moment to reflect and make an observation.

If the attacker sends a valid RSA ciphertext to be decrypted, the server will decrypt it and obtain some PMS value. If the attacker sends *the same* valid ciphertext a second time, the server will decrypt and obtain *the same* PMS value again. Indeed, the server will always get the same PMS even if the attacker sends the same valid ciphertext a hundred times in a row.

On the other hand, if the attacker repeatedly sends the same *invalid* ciphertext, the server will choose a different PMS every time. This observation is crucial.

In theory, if the attacker holds a ciphertext that might be valid or invalid â€” and the attacker would like to know which is true â€” they can send the same ciphertext to be decrypted repeatedly. This will lead to two possible conditions. In condition (1) where the ciphertext is valid, decryption will produce the "same PMS every time". Condition (2) for an *invalid* ciphertext will produce a "different PMS each time". If the attacker could somehow tell the difference between condition (1) and condition (2), they could determine whether the ciphertext was valid. That determination alone would be enough to resurrect the Bleichenbacher attack. Fortunately in TLS, the PMS is never used directly ; it's first passed through a strong hash function and combined with a bunch of random nonces to obtain a Master Secret. This result then used in further strong ciphers and hash functions. Thanks to the strength of the hash function and ciphers, the resulting keys are so garbled that the attacker literally cannot tell whether she's observing condition (1) or (2).

And here we finally we run into the problem of SSLv2.

You see, SSLv2 implementations include a similar anti-Bleichenbacher countermeasure. Only here there are some key differences. In SSLv2 there is no PMS â€” the encrypted value is used as the Master Secret and employed directly to derive the encryption session key. Moreover, in export modes, the *Master Secret* may be as short as 40 bits, and used with correspondingly weak export ciphers. This means an attacker can send multiple ciphertexts, then *brute-force the resulting short keys*. After recovering these keys for a tiny number of sessions, they will be able to determine whether they're in condition (1) or (2). This would effectively resurrect the Bleichenbacher attack.**This still**

sounds like an attack on SSLv2, not on TLS. What am I missing ?

| https://4.bp.blogspot.com/-Pk_xdh59...
[JPEG - 10.9Â kio]

| SSLv2 export ciphers.And now we come to the full horror of SSLv2.

Since most servers configured with both SSLv2 and TLS support will use the same RSA private key for decrypting sessions from either protocol, a Bleichenbacher attack on the SSLv2 implementation â€" with its vulnerable crappy export ciphersuites â€" can be used to decrypt the contents of a normal TLS-based RSA ciphertext. After all, both protocols are using the same darned secret key. Due to formatting differences in the RSA ciphertext between the two protocols, this attack doesn't work all the time â€" but it does work for approximately one out of a thousand TLS handshakes.

To put things succinctly : *with access to a whole hell of a lot of computation*, an attacker can intercept a TLS connection, then at their leisure make many thousands of queries to the SSLv2-enabled server, and decrypt that connection. The "general DROWN" attack actually requires watching about 1,000 TLS handshakes to find a vulnerable RSA ciphertext, about 40,000 queries to the server, and about 2^50 offline operations.

LOL. That doesn't sound practical at all. You cryptographers suck. First off, that isn't really a question, it's more of a rude statement. But since this is exactly the sort of reaction cryptographers often get when they point out *perfectly practical theoretical attacks* on real protocols, I'd like to take a moment to push back.

While the attack described above seems costly, it can be conducted in several hours and $440 on Amazon EC2. Are your banking credentials worth $440 ? Probably not. But someone else's probably are. Given all the things we have riding on TLS, it's better for it not to be broken *at all.*

More urgently, the reason cryptographers spend time on "impractical attacks" is that *attacks always get better*. And sometimes they get better fast.

The attack described above is called "General DROWN" and yes, it's a bit impractical. But in the course of writing *just this single paper*, the DROWN researchers discovered a second variant of their attack that's many orders of magnitude faster than the general one described above. *This* attack, which they call "Special DROWN" can decrypt a TLS RSA ciphertext in about *one minute* on a single CPU core.

This attack relies on a bug in the way OpenSSL handles SSLv2 key processing, a bug that was (inadvertently) fixed in March 2015 , but remains open across the Internet. The Special DROWN bug puts DROWN squarely in the domain of script kiddies, for thousands of websites across the Internet.

So how many sites are vulnerable ?This is probably the most depressing part of the entire research project. According to wide-scale Internet scans run by the DROWN researchers, more than 2.3 million HTTPS servers with browser-trusted certificates are vulnerable to special DROWN, and 3.5 million HTTPS servers are vulnerable to General DROWN. That's a sizeable chunk of the encrypted web, including a surprising chunk of the Chinese and Colombian Internet.

And while I've focused on the main attacks in this post, it's worth pointing out that DROWN also affects other protocol suites, like TLS with ephemeral Diffie-Hellman and even Google's QUIC . So these vulnerabilities should not be taken lightly.

If you want to know whether your favorite site is vulnerable, you can use the DROWN researchers' handy test .

What happens now ?In January, OpenSSL patched the bug that allows the SSLv2 ciphersuites to remain alive. Last March, the project inadvertently fixed the bug that makes Special DROWN possible. But that's hardly the end. The patch they're announcing today is much more direct : hopefully it will make it impossible to turn on SSLv2 altogether. This will solve the problem for everyone... at least for everyone willing to patch. Which, sadly, is unlikely to be anywhere near enough.

More broadly, attacks like DROWN illustrate the cost of having old, vulnerable protocols on the Internet. And they show the terrible cost that we're still paying for *export cryptography* systems that introduced deliberate vulnerabilities in encryption so that intelligence agencies could pursue a small short-term advantage â€” at the cost of long-term security.

Given that we're currently in the midst of a very important discussion about the balance of short- and long-term security , let's hope that we won't make the same mistake again.
[HTML - 1.6Â kio]