ImperialViolet

POODLE attacks on SSLv3 (14 Oct 2014)

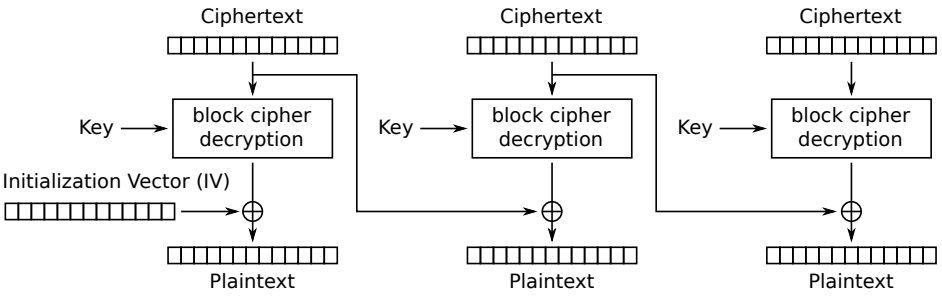
My colleague, Bodo Möller, in collaboration with Thai Duong and Krzysztof Kotowicz (also Googlers), just posted details about a padding oracle attack against CBC-mode ciphers in SSLv3. This attack, called POODLE, is similar to the BEAST attack and also allows a network attacker to extract the plaintext of targeted parts of an SSL connection, usually cookie data. Unlike the BEAST attack, it doesn't require such extensive control of the format of the plaintext and thus is more practical.

Fundamentally, the design flaw in SSL/TLS that allows this is the same as with Lucky13 and Vaudenay's two attacks: SSL got encryption and authentication the wrong way around – it authenticates before encrypting.

Consider the following plaintext HTTP request, which I've broken into 8-byte blocks (as in 3DES), but the same idea works for 16-byte blocks (as in AES) just as well:

The last block contains seven bytes of padding (represented as •) and the final byte is the length of the padding. (And I've used a fictional, 8-byte MAC, but that doesn't matter.) Before transmission, those blocks would be encrypted with 3DES or AES in CBC mode to provide confidentiality.

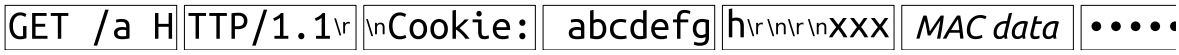
Now consider how CBC decryption works at the receiver, thanks to this public domain diagram from Wikipedia:



An attacker can't see the plaintext contents like we can in the diagram, above. They only see the CBC-encrypted ciphertext blocks. But what happens if the attacker duplicates the block containing the cookie data and overwrites the last block with it? When the receiver decrypts the last block it XORs in the contents of the previous ciphertext (which the attacker knows) and checks the authenticity of the data. Critically, since SSLv3 doesn't specify the contents of the padding (•) bytes, the receiver cannot check them. Thus the record will be accepted if, and only if, the last byte ends up as a seven.

An attacker can run Javascript in any origin in a browser and cause the browser to make requests (with cookies) to any other origin. If the attacker does this block duplication trick they have a 1-in-256 chance that the receiver won't reject the record and close the connection. If the receiver accepts the record then the attacker knows that the decryption of the cookie block that they duplicated, XORed with the ciphertext of the previous block, equals seven. Thus they've found the last byte of the cookie using (on average) 256 requests.

Now the attacker can increase the length of the requested URL and decrease the length of something after the cookies and make the request look like this:



Note that the Cookie data has been shifted so that the second to last byte of the data is now at the end of the block. So, with another 256 requests the attacker can expect to have decrypted that byte and so on.

Thus, with an average of 256×n requests and a little control of the layout of those requests, an attacker can decrypt *n* bytes of plaintext from SSLv3. The critical part of this attack is that SSLv3 doesn't specify the contents of padding bytes (the •s). TLS does and so this attack doesn't work because the attacker only has a 2^{-64} or 2^{-128} chance of a duplicated block being a valid padding block.

This *should* be an academic curiosity because SSLv3 was deprecated very nearly 15 years ago. However, the Internet is vast and full of bugs. The vastness means that a non-trivial

number of SSLv3 servers still exist and workarounds for the bugs mean that an attacker can convince a browser to use SSLv3 even when both the browser and server support a more recent version. Thus, this attack is widely applicable.

SSL/TLS has a perfectly good version negotiation mechanism that should prevent a browser and server that support a modern TLS version from using anything less. However, because some servers are buggy and don't implement version negotiation correctly, browsers break this mechanism by retrying connections with lesser SSL/TLS versions when TLS handshaking fails. These days we're more aware of the fact that fallback behaviour like this is a landmine for the future (as demonstrated today) but this TLS fallback behaviour was enshrined long ago and we're stuck with it. It means that, by injecting some trivial errors on the network, an attacker can cause a browser to speak SSLv3 to any server and then run the above attack.

What's to be done?

It's no revelation that this fallback behaviour is bad news. In fact, Bodo and I have a draft out for a mechanism to add a second, less bug-rusted mechanism to prevent it called `TLS_FALLBACK_SCSV`. Chrome and Google have implemented it since February this year and so connections from Chrome to Google are already protected. We are urging server operators and other browsers to implement it too. It doesn't just protect against this specific attack, it solves the fallback problem in general. For example, it stops attackers from downgrading TLS 1.2 to 1.1 and 1.0 and thus removing modern, AEAD ciphers from a connection. (Remember, everything less than TLS 1.2 with an AEAD mode is cryptographically broken.) There should soon be an updated OpenSSL version that supports it.

Even with `TLS_FALLBACK_SCSV`, there will be a long tail of servers that don't update. Because of that, I've just landed a patch on Chrome trunk that disables fallback to SSLv3 for all servers. This change will break things and so we don't feel that we can jump it straight to Chrome's stable channel. But we do hope to get it there within weeks and so buggy servers that currently function only because of SSLv3 fallback will need to be updated.

Chrome users that just want to get rid of SSLv3 can use the command line flag `--ssl-version-min=tls1` to do so. (We used to have an entry in the preferences for that but people thought that "SSL 3.0" was a higher version than "TLS 1.0" and would mistakenly disable the latter.)

In Firefox you can go into `about:config` and set `security.tls.version.min` to 1. I expect that other browser vendors will publish similar instructions over the coming days.

As a server operator, it is possible to stop this attack by disabling SSLv3, or by disabling CBC-mode ciphers in SSLv3. However, the compatibility impact of this is unclear. Certainly, disabling SSLv3 completely is likely to break IE6. Some sites will be happy doing that, some will not.

A little further down the line, perhaps in about three months, we hope to disable SSLv3 completely. The changes that I've just landed in Chrome only disable fallback to SSLv3 – a server that correctly negotiates SSLv3 can still use it. Disabling SSLv3 completely will break even more than just disabling the fallback but SSLv3 is now completely broken with CBC-mode ciphers and the only other option is RC4, which is hardly that attractive. Any servers depending on SSLv3 are thus on notice that they need to address that now.

We hardened SSLv3 and TLS 1.0 against the BEAST attack with 1/n-1 record splitting and, based on an idea by Håvard Molland, it is possible to do something called anti-POODLE record splitting this time. I'll omit the details, but one can ensure that the last block in a CBC record contains at least fixed six bytes using only one split for AES and two for 3DES. With this, CBC is probably less bad than RC4. However, while anti-POODLE record splitting *should* be easy to deploy because it's valid according to the spec, so was 1/n-1 and deploying that was very painful. Thus there's a high risk that this would also cause compatibility problems. Therefore I'm not proceeding with anti-POODLE record splitting and concentrating on removing SSLv3 fallback instead. (There's also the possibility that an attacker could run the attack on the server to client direction. If we assume that both the client and the server get patched then we might as well assume that they are patched for `TLS_FALLBACK_SCSV`, which makes record splitting moot.)