Increasingly mobile devices are making use of off-device computation for performance improvements or savings in battery life. For example, mobile browsers can make use of off-device website rendering to achieve both these benefits. However, in this model a malicious client could exploit these remote resources for what is in effect free computation. However, many of these services have no serious forms of authentication and furthermore don’t even identify individual clients. This combination makes it nearly impossible to filter malicious clients.

In this work, we reverse engineer a popular, cross platform cloud-based web browser (Puffin), and implement our own. This combination makes it nearly impossible to filter malicious clients. This work is left as future work.

To reverse engineering Puffin, we needed the traffic between the Puffin client and Puffin’s servers in the clear. Puffin uses TLS for end-to-end encryption, and the limited debugging capabilities of the Android platform made this a difficult task. We started by decompiling the Dalvik bytecode, and quickly realized it made all important calls through an included C library. By disassembling this library (libpuffin) we were able to begin to understand the workings of Puffin.

**Patching Libpuffin**

We found the SSL verification function inside Puffin and patched it to always verify the presented certificate. This allowed us to man-in-the-middle all Puffin traffic, and get the data in the clear. However, Puffin uses a binary protocol, and it is not immediately clear which messages do what.

**Reversing Traffic**

After decompiling the traffic and staring at it for hours, we were able to extract structure from the messages. Given enough time, messages like this make sense.

**Using Cookies**

Finally, Puffin’s servers send down a video stream containing the requested page. Rather than perform OCR on the stream, we found that cookies are sent in plain text. Using a combination of traffic in the clear and cookies, we can roll out a functional Lundi.

**Mitigations**

While fully preventing such attacks is impossible there are a few steps browser creators could take to mitigate the potential for attack. First, clients need to be uniquely identified, and second, the resources a single client can use should be capped. With these two changes in place, such an attack would likely not be worth pursuing.

If each client went through a registration process and were identified with a public-private keypair, then used this pair to sign requests, this would go a long way in preventing this attack. While this could still be spoofed, it would raise the barrier to entry and make the attack less desirable.

For further information contact Joe Pletcher (pletcher@cs.uoregon.edu)