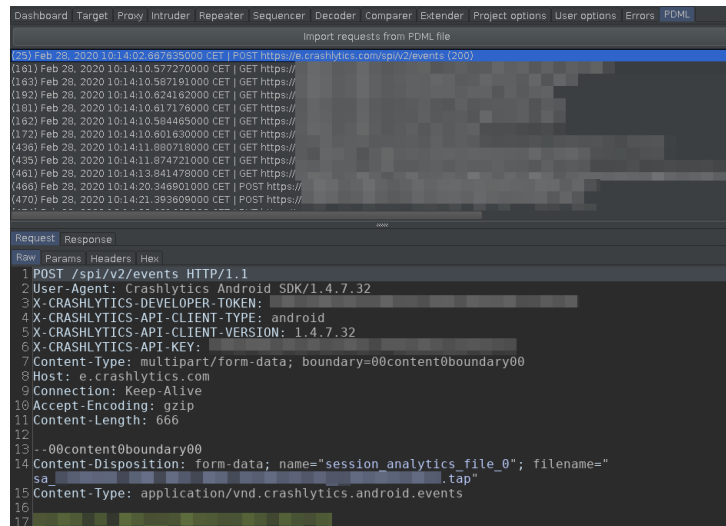




[HOME](#) [GITHUB](#) [TWITTER](#) [VIMEO](#)

[SILENTSIGNAL.EU](#)





# DECRYPTING AND ANALYZING HTTPS TRAFFIC WITHOUT MITM

May 4, 2020

android,api,burp suite,decryption,reverse\_engineering,tool,xml

Author: dnet

Sniffing plaintext network traffic between apps and their backend APIs is an important step for pentesters to learn about how they interact. In this blog post, we'll introduce a method to simplify getting our hands on plaintext messages sent between apps ran on our attacker-controlled devices and the API, and in case of HTTPS, shoveling these requests and responses into Burp for further analysis by combining existing tools and introducing a new plugin we developed. So our approach is less of a novel attack and more of an improvement on current techniques.

Of course, nowadays, most of these channels are secured using TLS, which provides encryption, integrity protection and authenticates one or both ends of the figurative tube. In many cases, the best method to overcome this limitation is man-in-the-middle (MITM), where a special program intercepts packets and acts as a server to the client and vice versa.

For well-written applications, this doesn't work out-of-the-box, and it all depends on the circumstances, how many steps must be taken to weaken the security of the testing environment for this attack to work. It started with adding MITM CA certificates to OS stores, recent operating systems require more and more obscure confirmations and certificate pinning is gaining momentum. Latter can get to a point, where there's a big cliff: either you can defeat it with automated tools like Objection or it becomes a daunting task, where you know that it's doable but it's frustratingly difficult to actually do it.

And then there are the cases from the second sentence of this post, where both ends perform authentication, and since the server is the one presenting the certificate most of the time, we usually refer to it as client certificate authentication, since that's the “exception” to the rule. In such scenarios, since the end-to-end TLS is split into two, the legitimate client authenticates to the MITM server, yet the MITM client also needs to present a certificate to the legitimate server. For this, the certificate and its private key must be extracted from the app, which, similarly to the aforementioned certificate pinning can be either done quick and easy (when it's in a PEM file) or it can lead to hours of frustrated reverse engineering.



```
lea     ecx, [eax+ebx*4]
movzx   eax, bh
jz       e_24_26

e_24_25:
805E0B4h] vfmaddsub132ps xmm0, xmm1, xmmword ptr cs:[edi+esi*4+805E0B4h]
shl     ebx, 12h
add     ebx, ecx
movzx   eax, bh
mov     al, 0
lea     eax, [ecx+edx*4]
lea     eax, [ecx+edx*4]
movzx   ecx, dh
dec     ecx
shr     eax, 1
mov     eax, 48A23077h
shl     edx, 8
test    edx, eax
lea     eax, [ecx+edx*4]
movzx   ecx, dh
lea     ebx, [ecx+ecx]
shr     eax, 9
mov     eax, 0C317A9Bh
or      eax, 0A5806DEFh
mul     eax
lea     edx, [ebp+0Ch]
mov     ch, 6
movzx   eax, bh
lea     edx, [eax+eax]
lea     edx, [ebp+0Ch]
dec     ecx
movzx   eax, bl
jz       e_25_26

mul
movzx
jz
e_24_
vfmad
lea
mul
movzx
sub
mov
movzx
shl
lea
sub
shl
lea
lea
add
movzx
mul
test
and
movzx
lea
movzx
mul
mul
shr
shl
lea
add
shr
jz
```



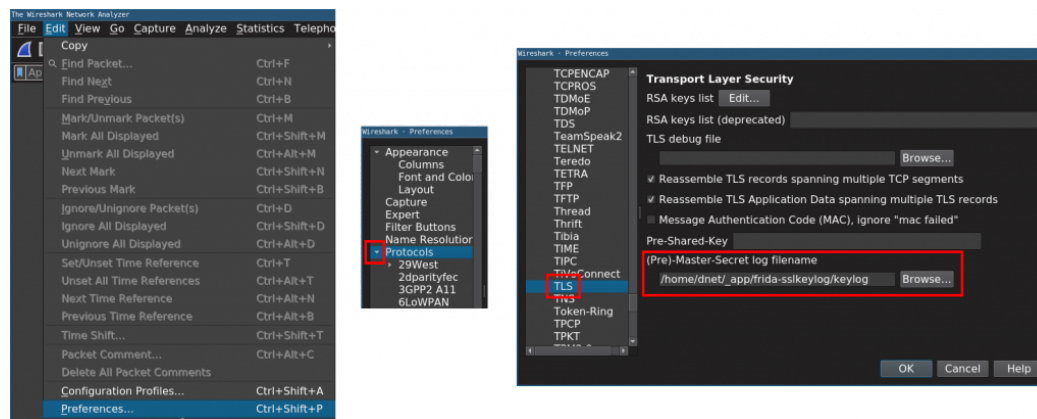
```
This gets worse and worse,  
until you give up or die.
```

Around this point is where some start thinking: do we really **need** to perform man-in-the-middle? Of course, MITM has its bright sides: modifying the plain text traffic on-the-fly is easy to implement, adding a match-and-replace rule to Burp to switch `X-Jailbroken: true` to `false` *just works*. On the other hand, if there are this many problems and all we need is *reading* the plaintext traffic, there are better solutions out there. It's just that MITM worked so well for so long, that it's the first thing that comes to mind; sometimes even if it's not the best fit for the problem.

It's obvious why we can read plaintext from TLS in a MITM scenario. But what do we really need to decrypt plaintext if we let the app speak directly to the API? Old-timers might remember Wireshark having the option to decrypt SSL/TLS when given the private key to the server certificate. This has two problems, one inherent to pentest projects, another related to the year being 2020. The former problem comes from the fact that in most pentest projects, you don't get the private key of the server, for multiple reasons. But even if you'd get it, the latter problem is that RSA-based key exchanges, where secrets are being encrypted using the public key from the server certificate are being phased out in favor of ephemeral solutions like

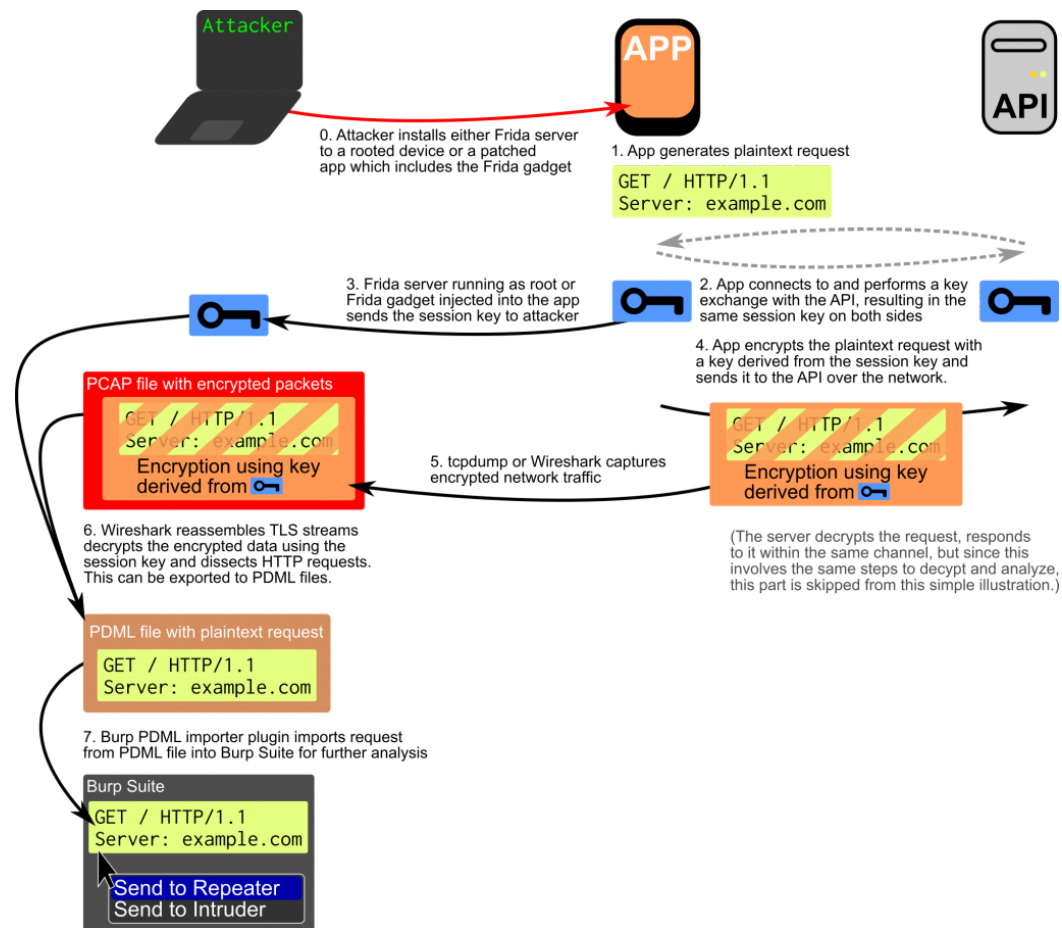
traditional (DLP-based) and elliptic curve (EC) Diffie-Hellman (DH). The reason, of course, is exactly why this approach against RSA key exchange would work, as real-world attackers could just as easily collect TLS traffic and compromise private keys later, hence modern implementations preferring key exchange algorithms with Perfect Forward Secrecy (PFS).

However, there's another way: cryptographic keys in a TLS session are derived from a (Pre-)Master Secret, which is present at both ends of the secure channel. And since we control one end of that, it can be used to get plaintext from TLS traffic. Fortunately, Wireshark supports this way as well, by specifying a file that has these secrets in a **specific format**. The best part is that this file is read continuously by Wireshark, so live network traffic can be decrypted in real-time if the file is also written as soon as those secrets are available. Since it's not a part everyone might visit every day in Wireshark, below is an illustrated guide that takes you to the precious input field that lets you specify the SSL key log file.



Browsers like Mozilla Firefox have a setting to dump such a file for debug purposes. In case of Android apps, there's a great project called [frida-sslkeylog](#) by [Saleem Rashid](#) that uses [Frida](#) to hook the appropriate OpenSSL (or BoringSSL) functions to extract the secrets real-time from the running app. For this, the only requirement on the device side is to have Frida available; this can be done by [installing it on a rooted Android following the official docs](#). If the app has issues running on rooted devices or there's another reason to avoid the above method, there's an alternative method of injecting the [Frida Gadget](#) into the app, thus it runs within the same process, avoiding the need for root. The easiest way to do this is by using the `patchapk` command of [Objection](#).

This gives us a nice decrypted traffic in Wireshark, which is already useful, but for us, this was only the beginning. If we wanted to use this information to test the API itself, we needed a way to get these request-response pairs into Burp to use its tools such as Repeater, Scanner, and Intruder. To sum up what I wanted (and in the end managed) to do, here's a diagram:



Learning from my mistakes and the wisdom of my colleague PZ, I searched for an existing solution, but the only reasonable one was **Pcap Importer**, which worked on the PCAP level, thus supported plain HTTP only. What I needed was a way to get the high-level, pre-processed information such as decrypted HTTP (ex-HTTPS) requests into Burp.



Fortunately, I remembered using PDML for other, previous network protocol reverse engineering projects of mine. PDML is an XML-based structured export format of Wireshark which contains all the parsed information that's also available within the GUI as part of the tree structure in the bottom panel. By consuming this format, most of the work is done by Wireshark, such as detecting HTTP messages and correlating requests and responses, parsing URLs.

Since I already wrote a [Burp extension to parse CFURL caches](#) (we have a [blog post about that](#) as well), most of the boring stuff could be copy-pasted from there, such as having a UI with a load button, an open file dialog, a message list, and the standard Message Editor controls of Burp, all tied together. Parsing seemed a bit difficult in the beginning, but as it turned out, the useful parts were found in two places:

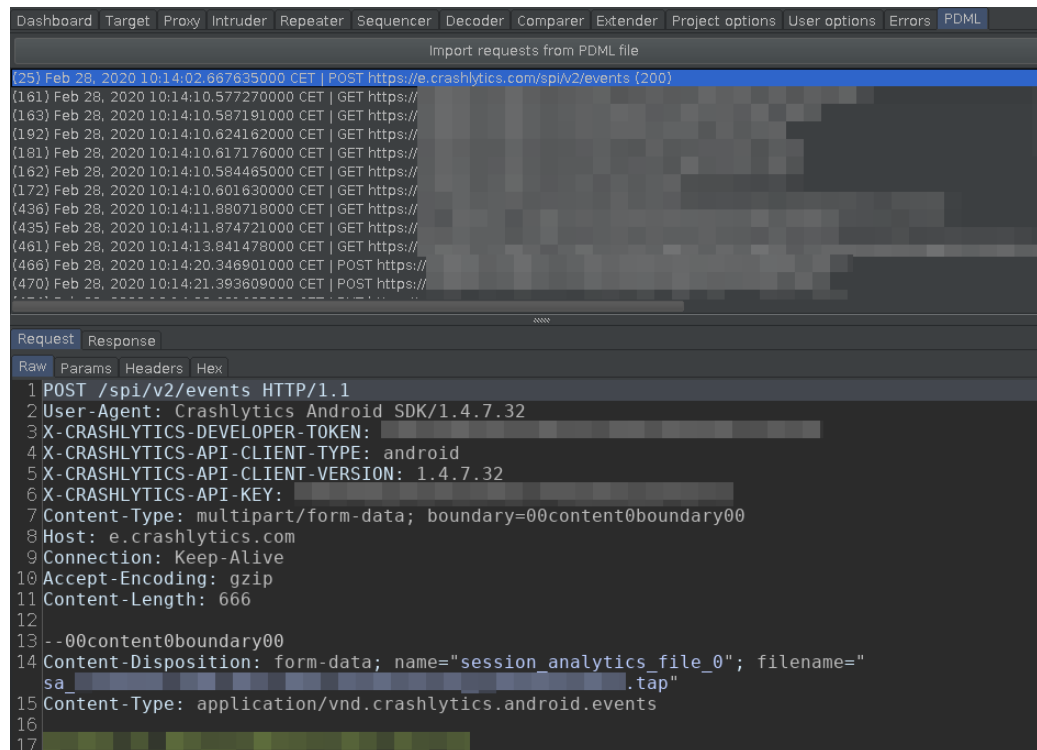
- Attributes called `show` contained parsed strings, such as integers like request-response correlation info, response code or processed strings like the URL or the HTTP method. These were used for metadata and UI strings.
- Attributes called `value` contained raw (but plaintext) bytes encoded using hexadecimal digits, request and response bodies were reconstructed from these.

The last big hurdle was performance; these PDML files got huge quickly since they contained every bit of information about all the packets. Most people when confronted with XML parsing choose DOM since it parses the whole document into

an in-memory tree that can be traversed in any random pattern, helped by tools such as XPath. However, in this case, this leads to huge latency for both the initial parsing and tree building phase and then traversing this structure. The logic was rewritten using a streaming (SAX) parser that fires events for every structural element, thus instead of an implicit tree, I could build a custom structure of my own that contained only the things I needed. In this case, this consisted of storing requests indexed by their packet number. When the response is found later, the request can be accessed in  $O(1)$  time, and added to the list of HTTP messages along with the response. This made things an order of magnitude faster.

Our new Burp plugin called PDML importer is available in [our GitHub repository](#) under MIT license, pull requests are welcome. Thanks to Saleem Rashid for developing frida-sslkeylog, and Balázs Goldschmidt for explaining SAX parsers to me at the university 10 years ago, which came very handy to make the Burp plugin much faster.

Below is a screenshot of the plugin in action, some values had been masked to protect the innocent. The number in parentheses on the left is the number of the packet in the original capture that the request was transmitted in so that it can be cross-referenced if needed. The exact timestamp also comes from packet capture metadata.



The screenshot shows a network traffic analysis tool interface. At the top, there are tabs for Dashboard, Target, Proxy, Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, Project options, User options, Errors, and PDML. Below these is a section titled 'Import requests from PDML file' containing a list of network requests. The first request is highlighted: (25) Feb 28, 2020 10:14:02.667635000 CET | POST https://e.crashlytics.com/spi/v2/events (200). Below this list, there are tabs for Request and Response. The 'Request' tab is selected, showing a detailed view of the POST request. The request details include: 1 POST /spi/v2/events HTTP/1.1, 2 User-Agent: Crashlytics Android SDK/1.4.7.32, 3 X-CRASHLYTICS-DEVELOPER-TOKEN: [REDACTED], 4 X-CRASHLYTICS-API-CLIENT-TYPE: android, 5 X-CRASHLYTICS-API-CLIENT-VERSION: 1.4.7.32, 6 X-CRASHLYTICS-API-KEY: [REDACTED], 7 Content-Type: multipart/form-data; boundary=00content0boundary00, 8 Host: e.crashlytics.com, 9 Connection: Keep-Alive, 10 Accept-Encoding: gzip, 11 Content-Length: 666, 12, 13 --00content0boundary00, 14 Content-Disposition: form-data; name="session\_analytics\_file\_0"; filename="sa\_[REDACTED].tap", 15 Content-Type: application/vnd.crashlytics.android.events, 16, 17 [REDACTED]

Of course, this is not the only possible approach, and certainly not the best solution for every scenario, as there is no one-size-fits-all method. Thus it's worth reading how others tackled similar issues:

- Péter Párkányi used tools built upon eBPF for decrypting Zoom traffic, which is a bit more low-level, requiring less resources, but limited to Linux,
- m1el injected a purpose-built DLL into the running process using `CreateRemoteThread` on Windows to reveal plaintext network traffic from Oculus, which is similar to what Frida does, but handcrafted for a single platform.

Share this, because you can!



.....