# CS548 Term Project *”Side Channel Leaks over HTTPS”*

Antoine RONDELET, Enoch LEE, Alexis GACEL, Felix BOSCHE

ABSTRACT

***Abstract* – *Nowadays web security is becoming a major concern for all the internet users and companies as there has been several security breaches in famous web sites. Therefore what users do on the internet must remain private. There still remains ways to know what users are doing on web sites by exploiting side-channel leaks. Our attack, which exploits a side-channel leak consists in performing network analysis and then infer the user’s actions based on the packet size. In this paper we will detail how we exploit such side-channel leaks over HTTPS to guess which information the user entered on the travel research engine KAYAK***

## I. PROTOCOLS OVERVIEW

### A. TCP/IP

The TCP/IP stack is a complete set of networking and communications protocol. It is a 4-layer concrete application of the OSI Model meant to standardize communication between devices over the Internet by specifying how data should be packetized, addressed, transmitted, routed, and received. It is called TCP/IP stack because of the two foundation protocols *Transmission Control Protocol* (TCP) and the *Internet Protocol* (IP).

### B. HTTP

*Hypertext Transfer Protocol* (HTTP) is a client-server communication protocol which was developed for the World Wide Web.It is a 7-layer concrete application of the OSI Model which aims at identifying the partners. The most famous HTTP clients are the web browsers which we use in the every day life. HTTPS is the secured version of HTTP which uses TLS/SSL.

### C. TLS/SSL

*Transport Layer Security* (TLS), and its predecessor *Secure Sockets Layer* (SSL) are cryptographic protocols for securing communications over the internet. The current version of TLS as of 2008 is TLSv1.2 but a first draft for TLSv1.3 was published in 2014. It satisfies the three objectives of security as depicted in the triangle Figure 1.

SSL/TLS layer sits just between the application layer (HTTP) and transport layer (TCP) thus, it does not require major overhaul of these two protocols and operate transparently on the user point-of-view. Its combination with HTTP is called HTTPS and will be the focus of this paper.

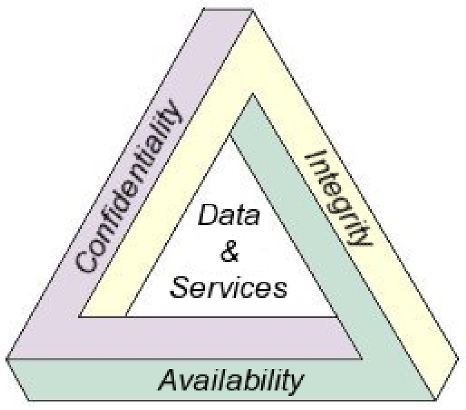


Fig. 1: The triangle of security

The first when establishing a secure channel of communication with HTTPS is the TLS handshake. It allows the peers to authenticate each other and to negotiate a cipher suite and other parameters of the connection. This part bears no interest for our project.

Once a secure channel has been setup, client and server starts exhcnahging data through TLS Records. The figure 2. below illustrates how these TLS records are built.

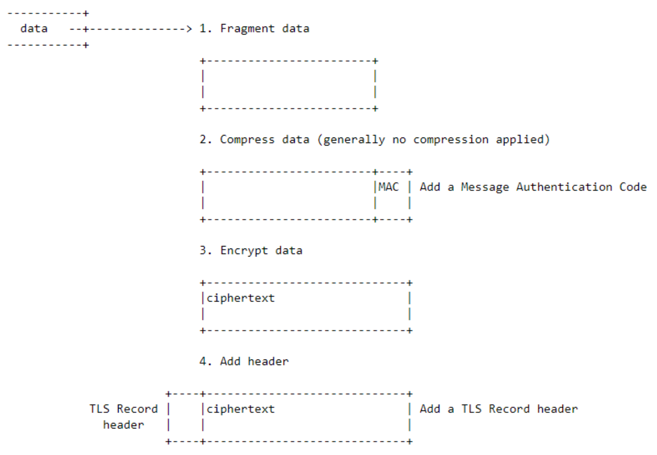


Fig. 2: Building of a TLS Record

## II. SIDE CHANNEL LEAKS

Historicaly side channels where used to describe attacks on physical crypto hardware systems.Secure web connections exchange informations that should only be understandable by the client and the server. An eavesdropper cannot understand what the two partners are exchanging but with network analysis it can infer informations such as which page the user is actually on. There are many types of possible attacks which exploit SCL. A first example is timing attacks on SSH[4]: this attack exploits the fact that when you connect using SSH all the keystrokes are send in separate IP packets, thus giving the inter-keystroke timing information. Using a statistical study on keystroke pattern one could know with a certain propability the key that was typed. An other example of SCL is Torben [6]. This is an application which can deanonimyze a Tor user. The attack exploits the interplay to know what pages the user visited using web page markers. This application shows that even Tor which is supposed to keep the anonimity of all its users is vulnerable to SCL. These examples show that SCL are a reality today and very hard to detect as they are passive and non-invasive.

## III. Content of our research

### A. Goals

As we said earlier in this paper, side-channel leaks have already been studied for quite a long time now. However, despite being well-known to developers, it seems that web applications are still leaking many pieces of information. While some researchers have already carried out studies on side-channel leaks on web applications, we couldn't find any paper presenting tools that automated the exploit of such information to infer the user input. Thus, we saw in this project, the opportunity to extend previous work, and build such an automated tool.

Nonetheless, since all web application present its very own specificities, designing a tool that would work for every web application seemed to be unfeasible at first glance. That is the reason why we decided to carry our project on a chosen target web application.

With this in mind, our project could then be defined as building a packet-analyzing tool that could give out the user inputs by analyzing TLS traffic of the targeted web application.

### B. Hypothesis

In order for us project to be successful, we decided to adopt several assumptions based on [1]:

- **Successfully mounted MiTM[1] attack**: This could be made possible in the case where the attacker manages to make his victim connect to a rogue wireless hotspot for instance. In such a scenario, network security technologies used on Wi-Fi wireless networks[2] do not hold, and the attacker can focus his attack on the HTTPS traffic of his victim. Note that, in our case, the attacker is passive. This means that he does not manipulate any data, but only sits between the victim and the web application, and observe the traffic.
- **Highly interactive target web application**: This hypothesis we make here, is a fairly weak hypothesis. Indeed, today, plenty of websites try to improve the user experience by implementing features such as auto-completion. Such functionality ensures low entropy inputs and aim to improve interactions with the user. From an attacker's point of view, such interactions between the client and server side of the web application are a bargain. They produce a lot of network traffic observable by the eavesdropper.
- *Significant traffic distinction*: In order for our attack to be successful, the attacker should be able to differentiate two requests based on observable attributes. While, this does not mean that the attacker is able to decrypt the encrypted data, this signify that some observable attributes should provide the attacker with enough information to differentiate the network traces of different requests.

### C. Environment

As aforementioned, the first part of our project has been to look for an appropriate web application to target. After a few days of research, we found that airlines website and, in a larger extent, journey websites were good candidates. In fact, they tend to offer many auto-completion features to facilitate the booking of their customers and make their experience as good as possible. Moreover, while some of them do not have protected *home pages*[3], a majority of them shield their customers' activity with encrypted channels. Furthermore, we agreed in the group that customers' journey data were pretty sensitive and could help an attacker acquire valuable knowledge on his victim's habits or his location during a trip.

At this point, since many candidates seemed to be viable for our project, we decided to carry out our attack on the web application *www.kayak.com*. This website is a fare aggregator and travel metasearch engine operated by The Priceline Group, based in the United States. Since KAYAK is available in 18 languages and used world wild, being able to achieve our attack on this website would have tremendous impact.

Our attack has been performed on a MacBook Air running macOS Sierra version 10.12.6. However, as we will see in the next section, we needed to do a precomputation phase in order to infer the user input with some satisfiable accuracy. In order to carry out this precomputation, we used 10 machines running Ubuntu[4] 16.04, with 521Mb of RAM and a single core CPU.

## IV. Description of the attack

The purpose of our project was to develop a tool that would infer the user inputs on a web pages using HTTPS. In order for us to implement such a tool, and implement the inference process, we needed to build a database to confront the sniffed data size to, in order to retrieve the most probable payload it could correspond to. The idea was to test all possible entries in the search bar of the website, and observe in the meantime the

[1]Man in The Middle. See more on https://en.wikipedia.org/wiki/Man-in-the-middle_attack
[2]More details: https://www.lifewire.com/what-is-wpa2-818352
[3]By protected we mean that they do not use HTTPS
[4]https://www.ubuntu.com

network traffic we generated. If we could manage to associate our actions to our network traffic trace, then we would be able to sniff the victim's network and compare the leaked information (packet size in our case) with our own network trace, and see to which payload it corresponds.

The following section presents the precomputation step of our attack.

## A. Attack precomputation

In order to infer the victim's input, we needed to test all possible entries in the search bar of *www.kayak.com*. While this could be done manually for a few payloads, this task is burdensome and error prone. In such a manipulation, the attacker has to observe his own network traffic[5], and map the request and response's packets to the payload he entered.

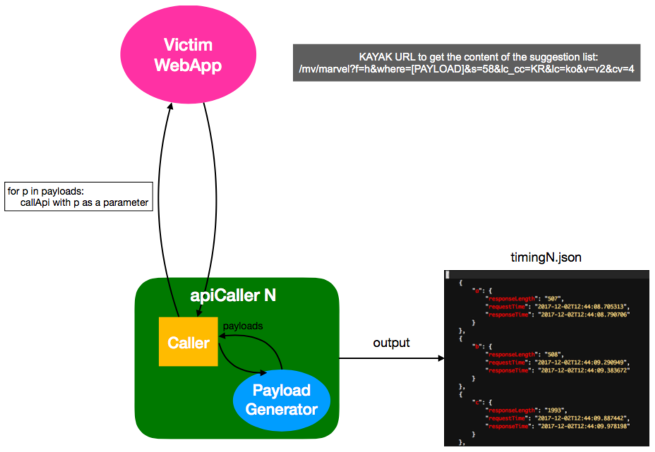
Our idea was to build a set of software components dedicated to test all possible user input in the search bar of the target web application. By doing so, we could record the generated network traffic and come up with an automated way to map each payload with the corresponding network traffic. However, since we wanted to have the best "dataset" possible, to carry out our attack, we decided to iterate on this "mapping" step, and do it many times and with different machines, in order to eliminate potential noise or inconsistencies from our records.

*The "API Caller":* In order to ease the work of the attacker, we implemented the *API Caller*, a component dedicated to test all possible input in the search bar of *www.kayak.com*. This module contains a *payload generator* submodule which generates all possible letter strings up to a certain length. Once all these payloads generated, they were given to the part of the *API Caller* dedicated to execute requests on the target web app.

By examining the web application with the developer console of our browsers, we were able to know the exact URL used by KAYAK to fetch the entries of the suggestion list. Thus, we could identify the location of the user input, split the URL into different parts and include our generated payloads, as if it were the result of a "human-machine" interaction.

As shown on the figure 3, the caller module receives the list of generated payloads from the generator, and sends a request to KAYAK for each payload in this list. However, in order to prevent the web server hosting KAYAK, to believe that we were DoS[6] attackers, we took the precaution to impose a few microseconds of sleeping time between each request. Although, this made our precomputation slower, this helped us make sure that we were mimicking actual user input.

Furthermore, as shown on 3, the output of the *API Caller* is a *timing.json*. This file is a JSON[7] array whose entries are JSON objects that associate each payload with the time the request has been made, the time the response has been received


Victim WebApp
KAYAK URL to get the content of the suggestion list:
/mv/marvel?f=h&where=[PAYLOAD]&s=58&lc_cc=KR&lc=ko&v=v2&cv=4
for p in payloads:
callApi with p as a parameter
apiCaller N
Caller
payloads
Payload Generator
output
timingN.json

```
{
    "a": {
        "responseLength": "507",
        "requestTime": "2017-12-02T12:44:08.705313",
        "responseTime": "2017-12-02T12:44:08.790706"
    }
},
{
    "b": {
        "responseLength": "508",
        "requestTime": "2017-12-02T12:44:09.290949",
        "responseTime": "2017-12-02T12:44:09.383672"
    }
},
{
    "c": {
        "responseLength": "1993",
        "requestTime": "2017-12-02T12:44:09.887442",
        "responseTime": "2017-12-02T12:44:09.978198"
    }
},
```

Fig. 3: The apiCaller

and the value of the *Content-Length* HTTP header. In fact, when the attacker precomputes the database necessary to carry out the attack, the requests are initiated from his machine, and the responses are received on his machine too. Thus, at point of time, the attacker's HTTP client decrypts the traffic. Then, he can know the "real" size of the response body. Such information is always to know from an attacker's perspective and might help differentiate the victim's encrypted traffic in the next steps.
In addition of knowing the size of the response body, we also stored the time when we initiated the request and received the response from the server. This timing information was used to help us split the network trace. With these times, we were able to extract the packets corresponding to the sending of each payload to the server (and the corresponding requests. Indeed, since HTTP dictates that responses must arrive in the order they were requested, and since our *API Caller* sends a new request after receiving the previous one, we had an ordered and easy-to-analyze network trace. Using the request and response time allowed us to know exactly which TCP traffic was generated after the request and response corresponding to a specific payload. As a consequence, the timing information provided as an output of the *API Caller* was paramount for the rest of the precomputation.

*The "Trace Builder Manager":* As aforementioned, the purpose of the *API Caller* was to lessen the burden of the attacker, by trying all possible inputs in the search bar of our target web application. While the requests and responses were sent and received, we needed to keep track of the network activity. This was the task of the *Trace Builder Worker*. Each of these workers were bash[8] scripts. They launched one tcpdump[9] daemon and an *API Caller*. While the *API Caller* was in charge of trying all possible user inputs, the tcpdump daemon recorded all the network trace in a *.pcap* file.

In order to maximize the output of our attack, we wanted to observe as many network traces as possible. That way we

[5]This can be done with software like Wireshark (https://www.wireshark.org)
[6]Denial of Service (https://en.wikipedia.org/wiki/Denial-of-service_attack)
[7]https://www.json.org

[8]https://www.gnu.org/software/bash/
[9]

would be able to get rid of any noise in our dataset. This noise could come from the network itself, or just from the fact that HTTP headers can differ slightly from one connection to another. Indeed, we know that HTTP headers are within the data encapsulated in TCP packets. Thus, a different change in headers could add some noise in our dataset, in the sense that, for a given payload, requests and responses would not have the exact same byte size. Since *User-Agent*, *Referrer* and some other HTTP headers tend to change frequently from one machine/connection to another, it is likely that such changes can be felt in our data. In the case where 2 payloads generate a very different and distinguishable traffic, such changes would not necessarily be a problem. However, we observed in our study that some web-traffics were pretty similar for some pairs of different payloads. In this case, having only a few bytes of difference between 2 traffics related to the same payload could fool the attacker and threaten the viability of the inference.

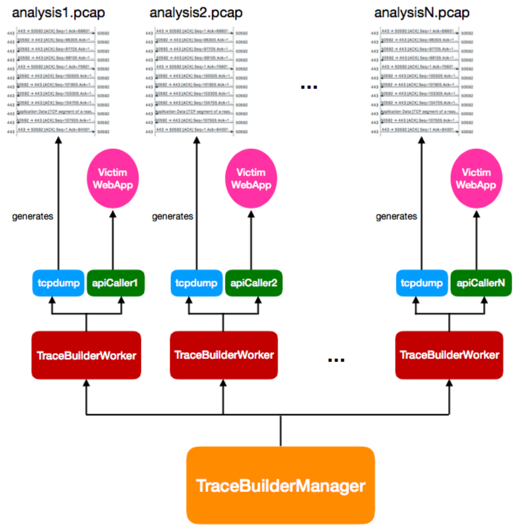


Fig. 4: The "TraceBuilderManager" launches many "TraceBuilderWorkers" to records multiple traffic traces

The figure 4 shows the *TraceBuilderManager*, which is the solution we came up with, in order to deal with the uncertainty described above. This component starts many *TraceBuilderWorkers* which launches one *tcpdump* daemon and an *API Caller*. The output of each worker is composed of 2 files: One network trace file (*.pcap*) and timing file which is used to analyze the network trace as explained earlier.

*The "dbBuilder":* The last component involved in the pre-computation step of our attack is the *dbBuilder*. This module launches as many *TraceAnalyzers* as the *TraceBuilderManager* launched *TraceBuilderWorkers*. Each *TraceAnalyzer* take the network trace file (*.pcap*) and timing file generated by the *TraceBuilderWorkers* at the previous step and are in charge of analyzing the network trace, and attribute to each payload tested by the *API Caller*, the network traffic they generated and which was captured by the *tcpdump* daemon. At the end of the analysis, each *TraceAnalyzer*, outputs a *resultTrace.json* file, which is a JSON file containing an array where each entry are payload mapped to the size, in bytes, of their associated request and response.

At this point of the precomputation, one could directly begin to attack the victim. Nonetheless, as we want to remove as much noise as possible from the database, the *dbBuilder* waits for each of its *TraceAnalyzers* to terminate in their respective threads, and collects all the resulting *resultTrace.json* files. In order for all our data to be represented in final database file, the *dbBuilder* computes for each payload in the *resultTrace.json* files, the mean of their request and response size.

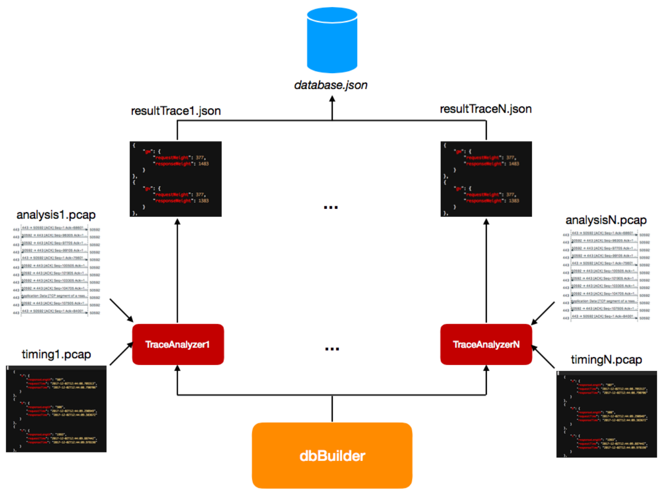
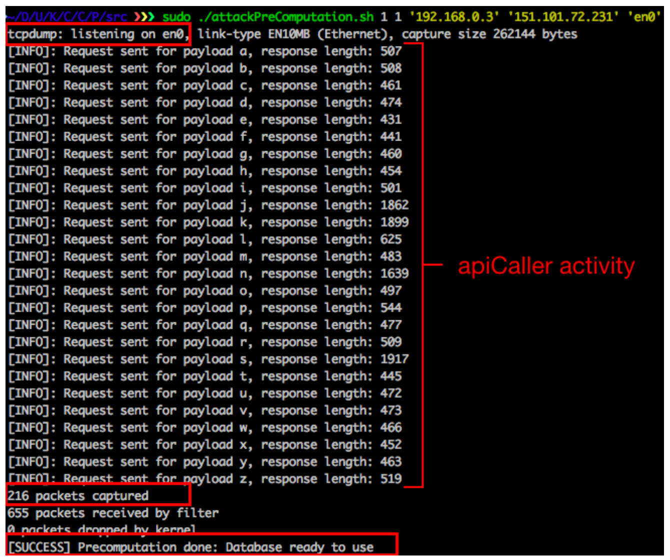


Fig. 5: The dbBuilder

The output of the *dbBuilder* represents the end of the precomputation stage. All the steps and components mentioned in this part have been wrapped up in a bash script, *attackPre-Computation.sh*, which takes the number of desired network traces, the payload max length, the attacker's IP address, the target webapp's IP address and the network interface to listen/sniff on, as arguments (see figure 6)



```
~/D/U/K/C/C/P/src ❯❯❯ sudo ./attackPreComputation.sh 1 1 '192.168.0.3' '151.101.72.231' 'en0'
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
[INFO]: Request sent for payload a, response length: 507
[INFO]: Request sent for payload b, response length: 508
[INFO]: Request sent for payload c, response length: 461
[INFO]: Request sent for payload d, response length: 474
[INFO]: Request sent for payload e, response length: 431
[INFO]: Request sent for payload f, response length: 441
[INFO]: Request sent for payload g, response length: 460
[INFO]: Request sent for payload h, response length: 454
[INFO]: Request sent for payload i, response length: 501
[INFO]: Request sent for payload j, response length: 1862
[INFO]: Request sent for payload k, response length: 1899
[INFO]: Request sent for payload l, response length: 625
[INFO]: Request sent for payload m, response length: 483
[INFO]: Request sent for payload n, response length: 1639
[INFO]: Request sent for payload o, response length: 497
[INFO]: Request sent for payload p, response length: 544
[INFO]: Request sent for payload q, response length: 477
[INFO]: Request sent for payload r, response length: 509
[INFO]: Request sent for payload s, response length: 1917
[INFO]: Request sent for payload t, response length: 445
[INFO]: Request sent for payload u, response length: 472
[INFO]: Request sent for payload v, response length: 473
[INFO]: Request sent for payload w, response length: 466
[INFO]: Request sent for payload x, response length: 452
[INFO]: Request sent for payload y, response length: 463
[INFO]: Request sent for payload z, response length: 519
216 packets captured
655 packets received by filter
0 packets dropped by kernel
[SUCCESS] Precomputation done: Database ready to use
```

Fig. 6: The database precomputation with a single network trace and payloads of length 1

Now that we have talked about the precomputation, the successfully mounted MiTM attacker, should be able to carry out the attack on his victim.

```
[
 {
   "a": {
    "requestWeight": 374,
    "responseWeight": 541
   }
 },
 {
   "b": {
    "requestWeight": 374,
    "responseWeight": 548
   }
 },
    ...
]
```

Listing 1: The JSON structure of the precomputed database

## B. Victim's network eavesdropping

After precomputing the database, the attacker should now be ready to eavesdrop the victim's network and infer the user's activity on the targeted web application.

The component in charge of the attack is called: the *networkAnalyzer*. This module is in charge of sniffing the victim's network traffic and analyze the data in order to return the most probable payload entered by the user. However, since we wanted to keep good performances during our attack, we don't wanted to slow down the sniffing by doing the inference on the same process. That is the reason why we split these 2 tasks in separate threads (see figure 7)

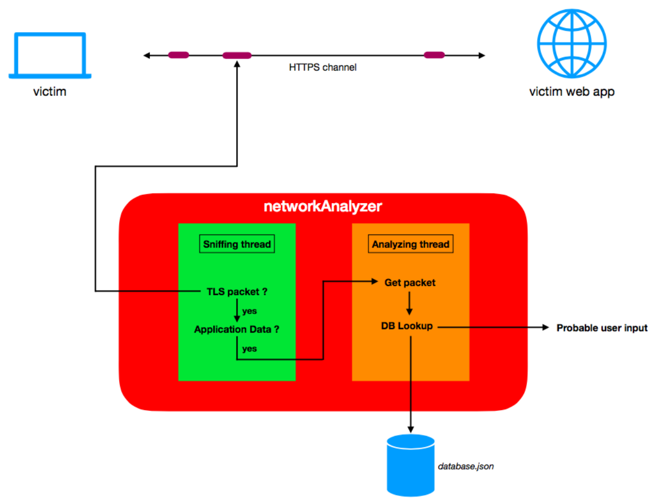


Fig. 7: The networkAnalyzer

*The Sniffing thread::* The *sniffing thread* aimed to do a *LiveCapture* of the victim's network and determine the nature of the captured packet (whether it is a TLS packet or a TCP ACK for instance). Once the *"sniffer"* was sure that the captured packet was a TLS packet, the next step was to make sure that this packet was actually carrying application data (and not a TLS Alert or other kind of TLS messages). If these 2 conditions were met, then the packet's information were written in a queue[10] whose data was consumed by the *Analyzing thread*.

*Note:* In order to analyze and dissect TLS and TCP packets sniffed on the network, we used the python module *Pyshark*[11] that is a wrapper around Tshark (Wireshark's CLI tool). This module was pretty handy since it enabled us to access the fields of each packet pretty easily. Nevertheless, we felt that this module was pretty slow to capture live traffic. Therefore, we often sniffed a victim's packet a few seconds after it was actually emitted by the victim user. Our attack can, thus, be qualified as a "near real time attack".

*The Analyzing thread::* While the thread dedicated to sniff the victim's network filtered TLS packets, carrying application data, before writing them into a queue; the thread devoted to analyze the packets, consumed the data from the queue. Each object read from the queue was a python object whose attributes were a *direction* and the *size* of the packet in bytes. The *direction* attribute specified whether the packet was sent as part of a request (client to server communication) or received as a response from the server. The *size* field contained the size in bytes of the application data encapsulated in the TLS packet.
Based on this information, the *analyzing thread* was able to do a lookup in the database and output the most probable payload. The database lookup is different is the case of requests and responses. However, in both cases, this can be split into 2 parts. Let's focus on requests first:

- The analyzer consumes a piece of data from the queue. The *direction* attribute is set to "Request". Thus, the *analyzer* only compares the value of the *size* attribute to the stored payload's *requestWeight* JSON field (see listing 1 for the DB structure).
- The comparison between "observed" request size and request size of values stored in the database has been implemented with a percentage of difference. The pseudo algorithm is described below:

```
// Step 1- Read data sent by the "sniffer"
sniffedData <- queue.get()
sniffedDataSize <- sniffedData.size
sniffedDataDirection <- sniffedData.direction

// Step 2- Find the most probable entries that
    match the observed Request traffic
minimumDifferencePercentage = 100
potentialPayloads = []
for each entry in the database, do:
 // Compute the percentage of difference
    absDiff <- |sniffedDataSize - entry.
        requestWeight|
    average <- (sniffedDataSize + entry.
        requestWeight) / 2
    percentageOfDiff <- (absDiff/average) * 100
    if percentageOfDiff <
        minimumDifferencePercentage:
     append entry to potentialPayloads
return potentialPayloads
```

Listing 2: Pseudo algorithm to infer user input during a Request

[10]https://docs.python.org/2/library/queue.html
[11]https://kiminewt.github.io/pyshark/

Using the percentage of difference to infer the probable user input allowed us to have more flexibility. Indeed, exact matching of request/response traffic size would not have been a good fit since we saw earlier in this paper that the actual TLS packet size, for a same payload, differed slightly from one request to another. Thus, doing exact matching would have produced very bad results in this case. Instead, we decided to measure the percentage of difference between the sniffed packets' size and the entries in our database. The result was a list of potential payloads. This list contained all the payloads for which the percentage of difference with the sniffed value was minimal. Returning such a list allowed us not to lose any information, and provide the attacker with the best information as possible.

We can see in the listing 2 that when the *"analyzer"* tries to find the payload corresponding to a sniffed request packet, it does a lookup in the entire database. This lookup can hardly be avoided because when the user inputs some data in the search bar, we do not have any clue of what this data can actually be. However, in the case of a response, the situation is different. Indeed, we know that the web applications responds what the user requested. Thus, in the case of a response analysis, we can optimize the lookup in the database. Instead of iterating on the entire list of records of the database to find the payloads with the closest response sizes, we can use the fact that we already observed the user request. Since we observed the user request and returned the list of potential payloads, we already eliminated a lot of entries of our database, and narrowed down our research to a subset of all payloads. The idea is to use this subset of the database in order to find the user payload by using the information we got from the traffic of the response.

The pseudo code corresponding to the inference of the user payload becomes:

```
// Step 1- Read data sent by the "sniffer"
sniffedData <- queue.get()
sniffedDataSize <- sniffedData.size
sniffedDataDirection <- sniffedData.direction
// [Response Only] We use the result of the
    observation of the request
potentialPayloadsFromRequestObservation

// Step 2- Find the most probable entries that match
     the observed Response traffic (Based on what we
     observed during the request)
minimumDifferencePercentage = 100
probablePayloads = []
for each entry in the
    potentialPayloadsFromRequestObservation, do:
 // Compute the percentage of difference
    absDiff <- |sniffedDataSize - entry.
        responseWeight|
    average <- (sniffedDataSize + entry.
        responseWeight) / 2
    percentageOfDiff <- (absDiff/average) * 100
    if percentageOfDiff <
        minimumDifferencePercentage:
     append entry to probablePayloads
return probablePayloads
```

Listing 3: Pseudo algorithm to infer user input during a Response (using the result of the inference on the request)

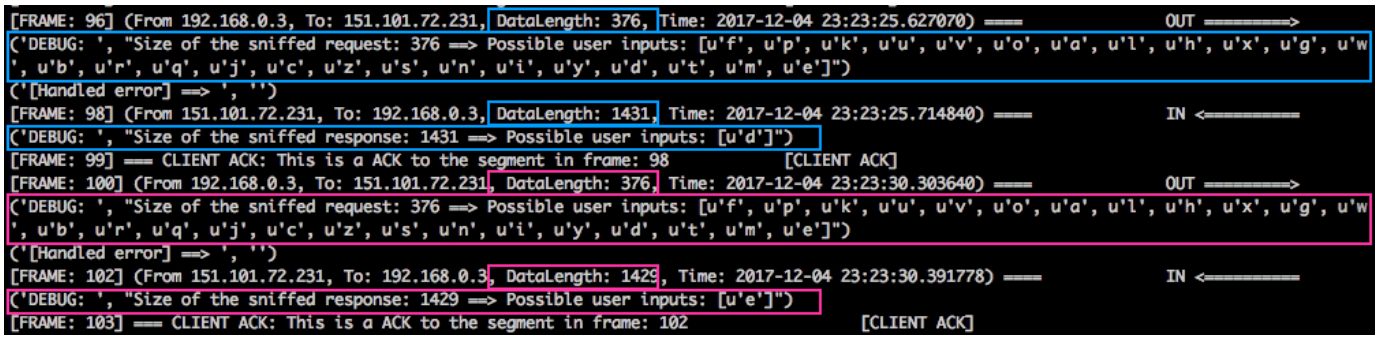
So here, we used the fact that the content of the response was tied to the content of a request, by trying to find the user input in the potential payloads returned during the analysis of the request. This made our program faster. Furthermore, since we used information from requests and responses, our program became more robust.

## V. Results

Looking back to our project proposal a few weeks ago, we proposed to "Build a packet-analyzing tool that could give out the user keypresses on a highly interactive web application". After reading some papers on Side Channel Leaks, we didn't find any work proposing a tool dedicated to automate side channel leaks exploit. We decided to build our own tool from scratch (see details in previous sections), in order to automate the different actions taken by an attacker during a network analysis attack.

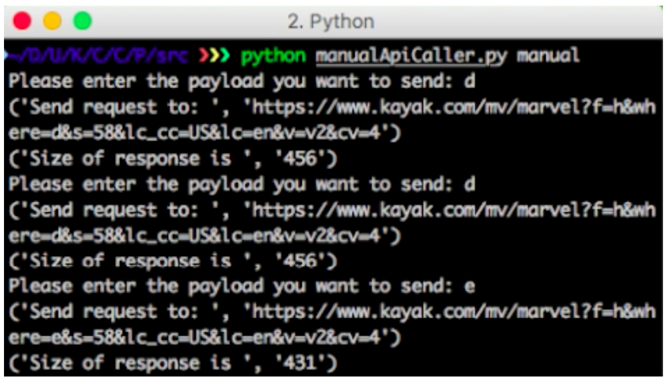
In order to demonstrate the mechanisms described in the previous sections, we did some simulations from which we extracted the figures 8 and 9. As we did simulations of our attack on *www.kayak.com*, we encountered some hindrance due to the fact that the web application generated a lot of traffic (see Section VI for more details). This made our analysis pretty hard to execute. That is the reason why we decided to use a *manualApiCaller* as represented on figure 9. This simple python module creates a HTTPS session with the web application and asks the user to enter the payload he wants to send to the application, before executing the request on the API. This helped us simulate user input in the search bar of the website, while isolating the network we wanted to analyze (no traffic due to advertisement and other parts of the web application).
While figure 9 shows the simulation of user input in the search bar of *www.kayak.com*, we launched in parallel, our attack to try to infer the payloads we were typing, based on information leaks over in TLS packets. Figure 8 shows the output of the *networkAnalyzer*. As we can observe, after a request (represented by the "OUT" arrow in the figure) is being executed further to a user input, the *networkAnalyzer* outputs the possible payloads corresponding to the observed packet size. In this case, a request of size 376 seems to correspond to the input of one letter in the search bar. Based on this first result, the analyzer will return the letter of this list whose response size is the closest to the response size observed when the server sends data back to the user. In this case, we didn't consider any payloads of size 2 to compare with the response size. We only considered the result of the request analysis. Figure 8 represents the TLS traffic analysis of the 2nd and 3rd requests of the figure 9. In these cases, the inference is successful, and the attacker is able to know the user inputs of the search bar of *www.kayak.com* despite the use of a secure channel for communications.

While figure 8 shows successful inference, we should not forget to mention that ut tool does mistakes in the case where network traffic is hard to distinguish. In the case where payloads' traffic size are very close to one another, a slight change in the server's response (a change in a HTTP header



```
[FRAME: 96] (From 192.168.0.3, To: 151.101.72.231, DataLength: 376, Time: 2017-12-04 23:23:25.627070) ====          OUT =========>
('DEBUG: ', "Size of the sniffed request: 376 ==> Possible user inputs: [u'f', u'p', u'k', u'u', u'v', u'o', u'a', u'l', u'h', u'x', u'g', u'w
', u'b', u'r', u'q', u'j', u'c', u'z', u's', u'n', u'i', u'y', u'd', u't', u'm', u'e']")
('[Handled error] ==> ', '')
[FRAME: 98] (From 151.101.72.231, To: 192.168.0.3, DataLength: 1431, Time: 2017-12-04 23:23:25.714840) ====          IN <==========
('DEBUG: ', "Size of the sniffed response: 1431 ==> Possible user inputs: [u'd']")
[FRAME: 99] === CLIENT ACK: This is a ACK to the segment in frame: 98           [CLIENT ACK]
[FRAME: 100] (From 192.168.0.3, To: 151.101.72.231, DataLength: 376, Time: 2017-12-04 23:23:30.303640) ====          OUT =========>
('DEBUG: ', "Size of the sniffed request: 376 ==> Possible user inputs: [u'f', u'p', u'k', u'u', u'v', u'o', u'a', u'l', u'h', u'x', u'g', u'w
', u'b', u'r', u'q', u'j', u'c', u'z', u's', u'n', u'i', u'y', u'd', u't', u'm', u'e']")
('[Handled error] ==> ', '')
[FRAME: 102] (From 151.101.72.231, To: 192.168.0.3, DataLength: 1429, Time: 2017-12-04 23:23:30.391778) ====          IN <==========
('DEBUG: ', "Size of the sniffed response: 1429 ==> Possible user inputs: [u'e']")
[FRAME: 103] === CLIENT ACK: This is a ACK to the segment in frame: 102           [CLIENT ACK]
```

Fig. 8: The network analyzer inferring correctly the user input over an HTTPS connection



```
~/D/U/K/C/C/P/src ››› python manualApiCaller.py manual
Please enter the payload you want to send: d
('Send request to: ', 'https://www.kayak.com/mv/marvel?f=h&wh
ere=d&s=58&lc_cc=US&lc=en&v=v2&cv=4')
('Size of response is ', '456')
Please enter the payload you want to send: d
('Send request to: ', 'https://www.kayak.com/mv/marvel?f=h&wh
ere=d&s=58&lc_cc=US&lc=en&v=v2&cv=4')
('Size of response is ', '456')
Please enter the payload you want to send: e
('Send request to: ', 'https://www.kayak.com/mv/marvel?f=h&wh
ere=e&s=58&lc_cc=US&lc=en&v=v2&cv=4')
('Size of response is ', '431')
```

Fig. 9: The manual API Caller we used to simulate user input in the search bar

for instance) might fool our tool and make it infer a wrong payload. However, despite such errors, our tool is working as expecting. In fact, our goal was to automate side channel leaks exploit, not to develop a perfect inference over TLS. In the case where the traffic is barely distinguishable, a ”real” attacker would also have some troubles in inferring the underlying user input. For now, our results are ”empirical” and based on the numerous simulations we made. We encountered some difficulties when we tried to launch our final precomputation (see Section VI). We believe, however, that our tool could be extended and improved in order to make it less error prone and more accurate (see Section VII).

## VI. Limitations

- First of all in order to eliminate the noise from the data that we recovered from sniffing the network, we needed a lot of network traces. This meant to do a lot of queries thus making the precomputation very long. in the short time we had to realise the project we were enable to perform a full precomputation with which we would have had better results.
- An other problem we encountered while doing the precomputation was that the server we used to do the precomputation was blacklisted as DigitalOcean believed we were doing a DDOS. Therefore we could not do the full computation we desired in order to have results as accurate as possible. A solution to this problem was to increase the time between two queries but we lacked time to do it.
- Kayak is a website which generates a lot of traffic which we had to filter. This was a complicated and time consuming task as we had to anaze a lot of traces

## VII. Improvements

We started this project from scratch not using any already existing libraries from python. Therefore this was a long process as we had to design all the architecture ourselves. Neththeless we are satisfied with our work and have thought of some improvements to our project. First we think that we could use Machine Learning or Neural Networks as they have been proven to be very efficient. This Learning agent would therefore replace our database and we would train it with a set of traces. We do not know if using such an Agent would improve our results but we would like to try as they have been proven to be very efficient in similar environments.

As mentionned previously in section II,timig attacks are very powerful and we wish to implement a timing attack as our case is similar to the case of the SSH timing attack. Indeed every time the user type a letter an AJAX request is send thus we could improve the accuraty by adding this feature. Additionally we need to guess the user’s keyboard layout by infering his location based on its IP adress. Indeed a user in France would most likely have a AZERTY keyboard and not a QWERTY which is more common.

Finally we could fingerprint several vulnerable websites that we would have identified before hand. We would then use those fingerprints to know which page the user is consulting. The prerequisites would be that we can see the network traffic generated when we fetch the homepage of these websites. This would then be stored in a database so we can then infer which page the user is. We would also need to perform a precomputation on all those websites. Then we would be able to know on which website the user is and use the precomputated database associated with the website.

## VIII. Conclusion

During this project we learned a lot about HTTPS flows and how to bypass the encryption to guess what informations are send. This was very interesting as HTTPS is the most widely used protocol for applications and we proved that it has some flaws. Additionally this project showed only one way to exploit SCL but there are plenty of other ways to exploit these leaks. As said before attacks exploiting SCL are very hard to detect, this is why those attacks are very powerful and should be not be easy to exploit.

## References

[1] *"Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow"*, Shuo Chen et. al, 2010

[2] *"Traffic analysis of ssl encrypted web browsing"*, Heyning Cheng, Heyning Cheng, and Ron Avnur, 1998.

[3] *"Traffic Analysis of the HTTP Protocol over TLS"*, George Danezis

[4] *"Timing Analysis of Keystrokes and SSH Timing Attacks"*, Dawn Song, David Wagner, and Xuqing Tian, 10th USENIX Security Symposium, 2001

[5] *"I Still Know what You Visited Last Summer"*, Zachary Weinberg, 2011

[6] " *Torben: A practical side-channel attack for deanonymizing tor communication*" D. Arp, F. Yamaguchi, and K. Rieck, ACM, 2015, pp. 597602