TeLeScope - real-time peering into the depths of TLS traffic from the hypervisor

Radu Caragea, Bitdefender
rcaragea@bitdefender.com

May 25, 2016

1 Introduction

Analyzing network traffic is a task that comes up often in the context of threat analysis. However, with the advent of free SSL/TLS certificates, any attacker can deploy encrypted channels for the purpose of delivering malware or receiving exfiltrated data. Thus, a need arises for the analysis of such communications as efficiently and covertly as possible. Real-world cases involve:

- investigating honeypot traffic: malware, backdoors etc. that have been downloaded using TLS and planted on machines with (intentionally) weak credentials
- pre-infection: malware delivery from sites (e.g. malvertising)
- post-infection: the communication with the C&C servers.

Having this information is vital for collecting the samples and then carrying out dynamic analysis on them. However, it is not currently trivial to decapsulate the encrypted traffic without major and thus noticeable modifications of the target VM.

Current solutions to this issue involve:

- adding a root CA (certificate authority) to the machine and proxying traffic in order to split the connection and re-sign certificates on-the-fly
- modifying/recompiling crypto libraries to log extra information (a solution deemed non-portable)
- using mechanisms already present that log such information (such as the SSLKEYLOGFILE environment variable present only in browsers relying on libNSS and boringSSL).

All these methods rely, in the end, on modifications in the guest VM; modifications that are visible and can be ultimately detected by the malware itself which can then choose to deactivate itself, leading to False Negatives.

An ingenious approach to this problem is to exfiltrate the key (called Master Secret in TLS terminology) of a conversation using an out-of-guest solution such as the one described in "Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection". The author

would use PANDA to instrument an emulated machine and set predefined "tap points" at various moments of execution and dump relevant memory. Although elegant, the approach has several drawbacks: both in terms of speed (the machine is emulated, not virtualized) and in terms of setup; the tap points need to be predefined and will break if the underlying system goes through a system update (unintended by the analyst) that targets the crypto libraries where the tap points are set or an explicit modification to the system files done by the attacker or the malware.

In this paper we first do away with the performance overhead of the previous approach by showing that the process can be replicated using modern memory introspection techniques similar to the ones employed in DRAKVUF. We then present a novel technique that not only works for virtualized machines with a minimal overhead but is actually OS-agnostic and crypto-library-agnostic: no assumptions about these are currently required to obtain the TLS keys. We achieve this performance by using some key X86 architecture features and by operating only under the assumption that the TLS communication that is to be decrypted respects the appropriate RFCs, an assumption which obviously holds if the communication is to be working.

Instead of pausing the machine (which would introduce noticeable latency) and doing a full memory dump, we develop a memory diffing technique using primitives already present in hypervisor technologies. Then, although this allows reducing the dump from gigabytes to megabytes, the time taken to write this quantity to a storage is still non-negligible (on the order of a few milliseconds) and thus we show how to further "disguise" the process in network latency, without having to pause the machine at all. Finally, we discuss the issue that the TLS context has multiple parameters: encryption keys, IVs or nonces, MAC keys and would imply that searching for them in the "micro memory dump" takes quadratic or even cubic time. However, we develop techniques for the most commonly used ciphers that require only linear time.

Because the primitives we require are mainly used in the cloud we then go further and raise the question of whether the proverbial tables have turned and we ourselves might be the ones in somebody else's sandbox being "dynamically analyzed".

In the following sections we first describe TLS as presented in the RFCs and pinpoint the features that allow us to leverage our technique. Afterwards, we employ out-of-VM breakpoints to exfiltrate TLS keys from known contexts: template VMs, offered by default by cloud providers. Then, we generalize the approach, making it more robust and allowing it to work without knowing any details about the underlying VM, useful when an attacker has full access to the VM, a scenario present in honeypots. As some brute forcing is still required to identify the correct keys we then assess the time needed and offer optimizations to usual approaches for the ciphers currently implemented in HTTPS enabled software. Finally, we provide benchmarks that show perceived latency and scalability.

2 Transport-Layer Security handshake

As described in RFC5246[1], the client and server need to first negotiate connection parameters and establish an encrypted channel in order to communicate securely.

[1] `https://tools.ietf.org/html/rfc5246#section-7.3`

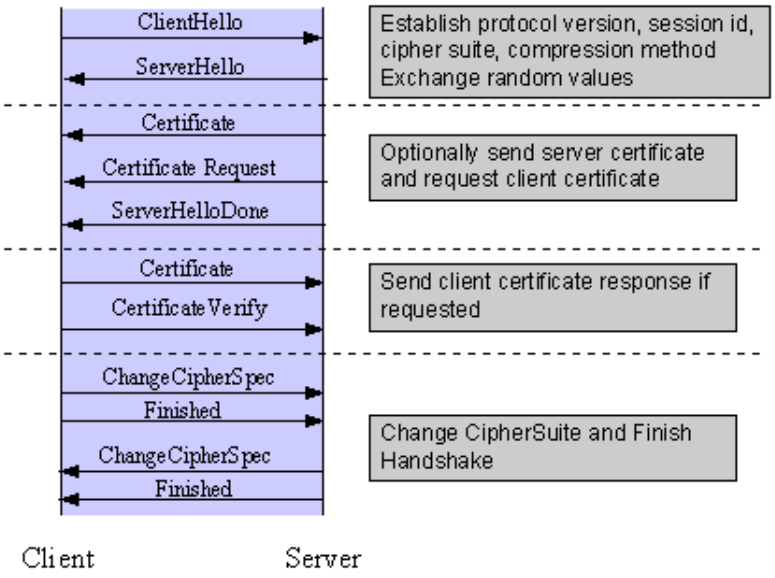

Figure 1: TLS handshake; `httpd.apache.org/docs/2.4/images/ssl_intro_fig1.gif`

This process starts off with an initial Handshake with the client sending a Client Hello containing (among many others):

- the list of supported cipher suites
- client random

The server responds with a Server Hello deciding further information on the cipher context:

- the chosen cipher suite from the ones enumerated by the client
- server random
- public key for key exchange (in the Server Certificate)
- parameters for key generation

If key exchange is selected, the client will send a 48-byte Premaster Secret encrypted with RSA. If key generation is selected then the Premaster Secret is mutually agreed upon using a Diffie-Hellman variant. The Premaster Secret will then be converted to the Master Secret using a PRF and truncated to 48 bytes as described in RFC5246[2]

This Master Secret will form part of the key block used to derive the key material containing the encryption keys, MAC keys and additional parameters to be used in the selected cipher suite according to the algorithm in listing 1.

After the encryption context has been completely established, both parties will send a Change Cipher Spec message indicating that the next message from the originating end will be encrypted with the corresponding keys. The handshake ends when both the client and the server have sent encrypted Client Finished (CF) and Server Finished (SF) messages.

Given the data flow we can conclude the following:

[2] `https://tools.ietf.org/html/rfc5246#section-8.1`

```
To generate the key material, compute

   key_block = PRF(SecurityParameters.master_secret,
                   "key expansion",
                   SecurityParameters.server_random +
                   SecurityParameters.client_random);

until enough output has been generated.  Then, the key_block is
partitioned as follows:

   client_write_MAC_key[SecurityParameters.mac_key_length]
   server_write_MAC_key[SecurityParameters.mac_key_length]
   client_write_key[SecurityParameters.enc_key_length]
   server_write_key[SecurityParameters.enc_key_length]
   client_write_IV[SecurityParameters.fixed_iv_length]
   server_write_IV[SecurityParameters.fixed_iv_length]
```

Listing 1: excerpt from `https://tools.ietf.org/html/rfc5246#section-6.3`

- The key material is dependent upon both Client Random and Server Random.
- How the key material is used depends on the chosen cipher suite.
- Thus, the Client Finished and Server Finished messages depend on the key material being computed and a cipher suite being chosen.

It follows that the encryption keys (along with IV/nonce, MAC keys) will not be present in memory at the moment immediately before Server Hello reception by the client. We will come back to this key observation when generalizing our technique to work without knowledge of the underlying OS.

3 Extracting keys from a template VM

Knowing how the TLS handshake works and having all the information about the virtual machine (VM) under scrutiny and its address space layout makes it significantly easier to extract TLS secrets. Most of the memory layout is already known, including processes that carry out encryption and all the shared libraries. Given just a memory dump, a simple inspection using the Volatility framework can provide the location of the keys with moderate effort. However, we would like to do this on the fly, without the need to copy the whole guest physical memory to storage and then doing post-mortem analysis.

Next we discuss the particular case of a Windows 7 machine with Internet Explorer. The approach can be adjusted for any Virtual Machine coming from a template, that the analyst has access to.

Internet Explorer delegates all TLS-related tasks to the Microsoft Schannel security support provider through the `ncrypt.dll` library. The key material is in fact generated and stored in the address space of the `lsass.exe` process (Local Security Authority Subsystem Service). To decrypt or encrypt content in the TLS stream, processes go through RPC

and communicate with `lsass.exe`, making this process a prime candidate for snooping on TLS Master Secrets.

Obtaining the Master Secrets can be done by using introspection techniques like the ones employed in DRAKVUF to set out-of-VM breakpoints on the functions that compute the Master Secret. Since the key material is calculated from this Master Secret and elements present in the traffic, it suffices to obtain either the Premaster Secret or the Master Secret.

Determining where to set a trap and what memory to dump at the trap event requires some reverse engineering, however. Using the Microsoft Symbol Server we obtain the public debugging symbols for `ncrypt.dll` and determine that the TLS Master Secret is generated in the function called `Tls1ComputeMasterKey` disassembled in IDA as presented in figure 2

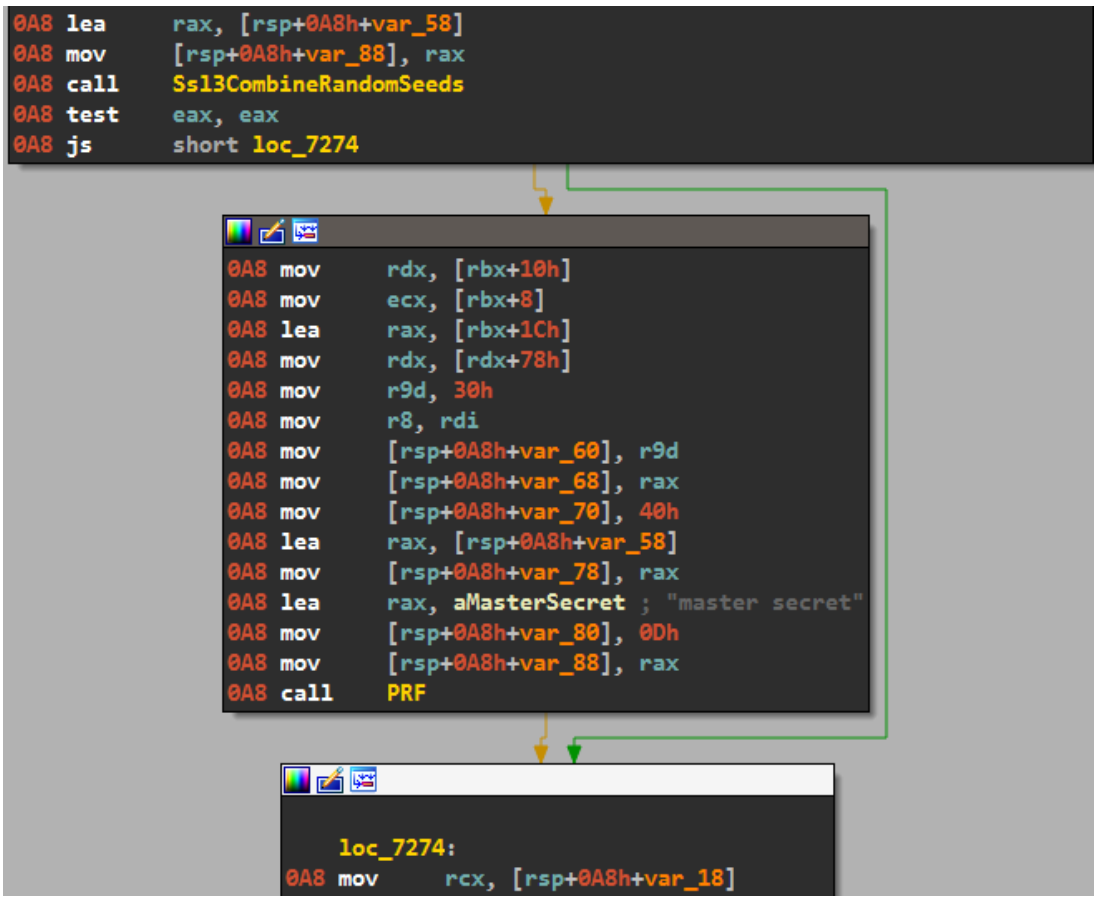

Figure 2: ncrypt.dll!Tls1ComputeMasterKey disassembly

Using the RFC2246[3] it is easier to identify what happens in the assembly code. A relevant excerpt is present in listing 2

```
master_secret = PRF(pre_master_secret, "master secret",
                            ClientHello.random + ServerHello.random)
```

Listing 2: Excerpt from RFC 2246 illustrating master secret derivation

The `Ssl3CombineRandomSeeds` concatenates the Client Hello and Server Hello and stores

[3] `http://tools.ietf.org/html/rfc2246#page-47`

the result on the stack. Next, this concatenated string, the "master secret" fixed string and the Premaster Secret are fed to the PRF function. This means that if we have the symbols available to the `ncrypt.dll` and we can set an out-of-VM breakpoint on the PRF function, it suffices to know the calling convention and check whether a parameter is the "master secret" fixed string. Then we only need to dump the memory pointed to by the corresponding parameter passed to the PRF function.

However, this solution does not scale: any change in the `ncrypt.dll` will invalidate the attack and require reanalysis of the binary for appropriate breakpoints.

4 Generalized technique for arbitrary VMs

4.1 Optimized memory dump

The general case that we would like to solve is when there is no information about the underlying VM or crypto library used for TLS communication, in which case key exfiltration becomes significantly harder. The intuitive way would be to determine when a target TLS connection is in progress and the keys for the encryption context should be in memory. As we discussed previously, this should be after both Client/Server Finished messages have been exchanged and, of course, before the connection has ended.

A naive approach would be to determine such a moment and pause the VM in order to create a memory dump of the whole guest physical memory (pausing is needed to ensure that the connection does not end until the memory dump completes; should that happen, the keys might be overwritten/erased from memory). The problem with this method is its noticeability: dumping gigabytes of memory to disk does not happen instantly; it depends on both RAM speed and permanent storage write speed, in this case, the disk write speed being the bottleneck. A possible optimization would be to copy the VM RAM in another memory zone in the hypervisor. However, this approach does not scale as browsers usually issue around 5-10 TLS connections at a time as observed in typical browsing sessions.

The method we propose is to reuse part of the Live Migration mechanisms present in most hypervisors and in particular we focus on the Xen hypervisor. To achieve Live Migration, the Xen hypervisor takes the following steps [4]

- enable_logdirty: a `XEN_DOMCTL_SHADOW_OP_ENABLE_LOGDIRTY` hypercall is issued so that the hypervisor starts tracking pages that have been written to starting from that moment
- send_memory_live: the guest physical memory is sent on the network and the dirty pages are iteratively tracked using the `XEN_DOMCTL_SHADOW_OP_CLEAN` hypercall to find a moment when the number of dirty pages is below a threshold while transmitting them on the network at each iteration
- suspend_and_send_dirty: this is the last stage of the live migration in which the VM is finally put on pause, the dirty page set is retrieved and sent to the migration destination and execution is continued at that endpoint.

[4] `http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=tools/libxc/xc_sr_save.c;h=7dc3a48ccb170b34f291ccc4e7cc78542be05e2d;hb=HEAD#l614`

Using the log-dirty primitives we can narrow down the needed number of pages for the memory dump from the whole RAM to only the ones that get modified during the TLS handshake.

We first show how to track the pages for a single connection and then generalize for multiple overlapping connections.

4.2 Single connection tracking

As we mentioned before, throughout the lifetime of the TLS connection some key packets on the network can signal whether the keys are present in memory or not (yet). However, obtaining these signals passively (using tcpdump or similar) in the hypervisor does not provide the opportunity to act on this information in time. We need to attach inline with the network stream and obtain packet information. To this end there are multiple methods that could be put to use such as:

- netfilter queue
- xen events
- ebtables

For our proof-of-concept we split the process into two components:

- `Telescope`: a component that attaches to the guest VM from the hypervisor and processes network packets to and from that VM. According to network events it also toggles the logdirty mechanism and dumps memory pages for each detected TLS handshake
- `Telesync`: a component that processes the memory dumps and produces a wireshark-compatible output that can be used to decrypt TLS traffic for that specific connection

The `Telescope` uses `libnetfilter_queue` to track packets pertaining to the TLS handshake by applying a few "dumb" rules to iptables. Each VM has its own netfilter queue and all TCP packets exchanged with the VM that:

- start with the TLS Handshake Protocol signature bytes are sent to a netfilter queue (unique per VM)
- start with the TLS Change Cipher Spec Protocol signature bytes are also sent to the corresponding VM queue

Of course, this approach might produce some False Positives when other protocols exchange packets starting with those bytes but these are dealt with later when validating TLS Record Layers.

When a Server Hello packet is detected to be in transit towards the VM, the context is updated such that:

- the TLS version is extracted from the packet for later use in `Telesync`
- the cipher suite chosen by the server is also saved for later use in `Telesync`

- the log-dirty mechanism is enabled (as the key can only be present after the client has received this packet as we have proved)

When a Client Finished packet is detected to originate from the VM, the context is updated such that:

- a sanity check is put in place to ensure that the corresponding Server Hello has been captured and logging has been previously enabled
- the VM is put on pause
- the log-dirty mechanism is disabled and the dirty pages are saved to disk
- the VM is unpaused
- the Client Finished encrypted payload is saved for later use.

For the Server Finished packet we only need to save the encrypted payload; the key material is generated once for both encryption directions such that the keys are already present in the memory dump saved from the Client Finished packet.

Actually, the VM does not need to be paused at all as we do not need a consistent state of the whole pages, only the keys, which do not "move around" in the physical memory as our experiments have shown. To ensure that the connection does not end before we have saved the dirty pages to disk, we can keep the packets from the server in the netfilter queue until the process completes, effectively disguising the process in network latency.

4.3 Multiple connection tracking

The previous section only discusses non-overlapping connections, a simplified scenario from what happens in real-world usage. To be able to track multiple connections efficiently we need to solve the following problem: given a logging primitive that returns the current dirty page set (since the last logging), work out the pages that have been modified between any two arbitrary events (in our case Server Hello and Client Finished). Example:

Events \ Page Index	0	1	2	3	4	5	6
e1	0	1	0	1	1	0	1
e2	1	0	0	1	0	0	1
e3	0	1	1	1	0	0	1
e4	1	0	0	0	1	1	1

Assuming there are 2 connections and thus 2 pairs of Server Hello/Client Finished let's first assume that:

- e1 is Server Hello 1
- e2 is Client Finished 1
- e3 is Server Hello 2
- e4 is Client Finished 2

This scenario would correspond to the serial case that we have already discussed: the only pages that need to be saved are those that have been modified (marked 1) at the

corresponding Client Finished event (pages 0, 3, 6 for connection 1 and 0, 4, 5, 6 for connection 2).

Now let's assume that:

- e1 is Server Hello 1
- e2 is Server Hello 2
- e3 is Client Finished 1
- e4 is Client Finished 2

This is the kind of scenario that we would like to capture correctly. Notice that just returning the dirty set of pages corresponding to the Client Finished is not enough (for connection 1 the correct dirty set is 0, 1, 2, 3, 6 as opposed to 1, 2, 3, 6), we will also need to keep a history of the dirty pages of events in between. To solve this problem efficiently space-wise first notice that all columns are independent, we only need to solve the problem for one page and the generalization is trivial.

The problem reduces to the following: given a list of events and their corresponding bit return the bitwise OR between any event and all subsequent events f(e_i, e_k). Of course, if the bit from the last event is 1 the final result will be 1.

Events	0
e1	0
e2	1
e3	0
e4	0
e5	0
e6	1
e7	0
e8	0
e9	0

Notice that f(e1, e9) = 1 because of event 2 and event 6, f(e6, e9) = 0 because the last modification occurred before event 6. It follows that we are only interested in the last modification time of a page. Having the start event timestamp and last modification timestamp we can decide if the last modification occurred after the start event (and thus the final result is 1) or before the start event (and the result will be 0).

The only thing remaining is to be able to pair events. To this end we keep a global list of currently active TLS connections with the following data:

- Source IP and port
- Destination IP and port
- Server Hello timestamp

The final and generalized solution is the following:

- Create a global array of timestamps (equal to the number of pages in the guest VM)
- For each Server Hello get the current dirty set and update the timestamps of the modified pages to the current timestamp.

- For each Client Finished get the current dirty set and dump the dirty pages. Afterwards, obtain the timestamp of the corresponding Server Hello and iterate over the timestamp array, dumping any page that has been modified after the Server Hello event.

4.4 Obtaining the encryption context from memory dumps

Even though we obtained encrypted payloads in each direction (the Client and Server Finished messages are encrypted) and now we could assume the keys are in plain format and theoretically try all the keys, it is not trivial to pinpoint which would be correct ones. A first approach would be to try all consecutive key_sz bytes (key_sz being the encryption key size) and apply a simple heuristic such as measuring the entropy of the decrypted message. Selecting the decrypted message with the least entropy should provide the correct key but it may also produce False Negatives (the TLS payload itself may be encrypted or compressed) and would add another layer of complexity to the running time of the process.

To solve this problem we now describe the internals of the `Telesync` component. One of our chosen approaches is to use the TLS message format itself to mount a KPA (Known Plaintext Attack). In general, Client/Server Finished messages start with the following pattern:

```
14 00 00 0c [12 random bytes]
```

As observed from browser traffic, this pattern might vary in cases where an Encrypted Extensions message is sent first:

```
43 00 00 [SZ] [SZ bytes] 14 00 00 0C [12 random bytes]
```

However, the first ciphertext block cannot always be decrypted efficiently and thus we discuss each case separately in the following sections regarding the ciphers selected by current browsers in the real-world.

5 Customized KPA for current ciphers

5.1 Ciphers used in Alexa top 1000

We conducted a survey on the Alexa top 1000 sites to see which symmetric encryption algorithms are selected in the Server Hello and thus we need to focus our attention on. Out of the first top 1000 sites, only 70% used TLS/SSL. Running once with an openSSL client and then with a libNSS based client results in the breakdown illustrated in figure 3

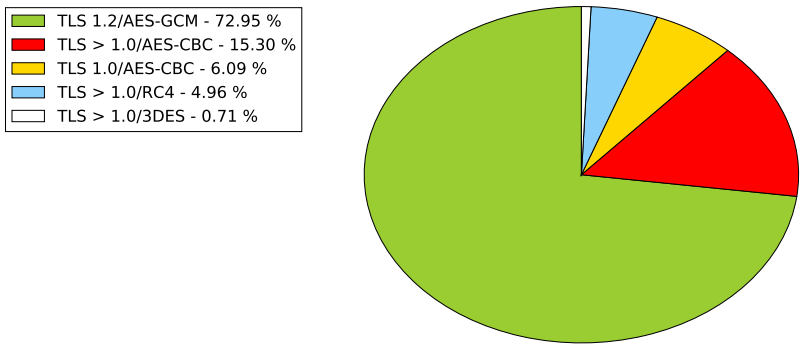

Figure 3: Alexa Top 1000 server-selected symmetric ciphers

5.2 Stream ciphers

5.2.1 RC4

RC4 is a legacy stream cipher not common at present because of its security risks. However, browsers will still add it to their cipher suite preference list towards the end. TLS uses a 16 byte key for RC4 without any other parameters and padding the message is not necessary as this is a stream cipher. Bruteforcing the CF/SF messages requires knowing which of the two possible known plaintexts appear at the beginning of the decrypted messages. Since there is no padding, the packet length provides a side-channel: the minimum packet length is $4 + 12 + MAC\_output\_size$. If the packet size is equal to this value then the first 4 bytes are `14 00 00 0C` otherwise the first 4 bytes are `43 00 00 [SZ]` where SZ can be easily calculated.

The False Positive rate of this method is 2^{-32} which is easily acceptable given that per connection we need to try on average less than 2^{25} keys. Should there be more than one key, a further validation can be imposed by checking the MAC.

5.2.2 Chacha20

Chacha20 is a relatively new stream cipher which is used in a few cipher suites mainly in Google Chrome and by crypto libraries such as BoringSSL but is not commonly selected.

```
/* CRYPTO_chacha_20 encrypts |in_len| bytes from |in| with the given key and
 * nonce and writes the result to |out|, which may be equal to |in|. The
 * initial block counter is specified by |counter|. */
OPENSSL_EXPORT void CRYPTO_chacha_20(uint8_t *out, const uint8_t *in,
                                     size_t in_len, const uint8_t key[32],
                                     const uint8_t nonce[8], size_t counter);
```

Initially[5], the nonce was set to the TLS sequence number making the bruteforce process similar to RC4. The only difference was that the MAC function is replaced with the associated POLY1305 algorithm used with Chacha20. However, in recent[6] implementa-

[5] https://www.ietf.org/proceedings/88/slides/slides-88-tls-1.pdf
[6] https://www.ietf.org/proceedings/90/slides/slides-90-cfrg-0.pdf

tions, the nonce is also taken from the key material, making it the only cipher that needs quadratic time to be bruteforced.

5.3 Block ciphers

5.3.1 AES

The Advanced Encryption Standard algorithm is the standard block cipher used in TLS and is the only mandatory[7] cipher. The commonly used block cipher modes supported for AES are CBC and GCM. The CBC mode is slightly different in TLS 1.0 versus higher versions and will be discussed separately.

5.3.2 AES CBC for TLS 1.0

The cipher block chaining mode is the second most commonly used mode in TLS connections and works as in the following diagram:

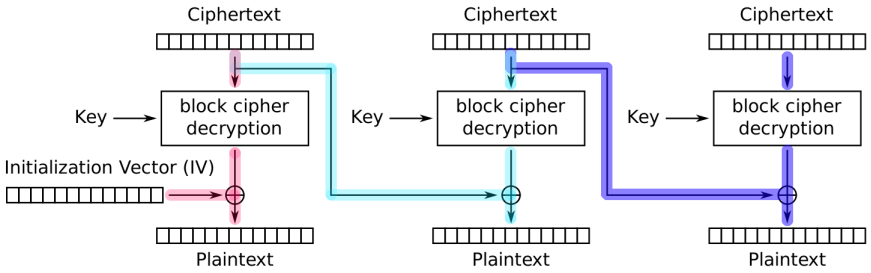

Figure 4: CBC mode

Decrypting each block requires the previous block to be used as a XOR mask after standard decryption. Because of this, the IV (initialization vector) can be viewed as an artificial block that is used for the decryption (and encryption) of the very first block. By randomizing the IV, a plaintext, although encrypted with the same key, will map to different ciphertexts.

Since our KPA depends on the first 4 bytes of the first block we also need the IV. This raises the time needed given N candidates of keys/IVs to $N \times (aes\_decrypt\_time + N \times xor\_mask\_time)$ thus $O(N^2)$.

To narrow down the search we use another feature of block ciphers: padding. In TLS, block cipher padding consists of a byte B indicating padding size, followed by B bytes of value B as in figure 5.

```
0000  14 00 00 0c 7f a6 87 08 ab 2e 6c 32 fd ba f7 c9  ..........l2....
0010  9c c5 76 20 da 83 6e b5 27 af ac ce e0 ac 1a e4  ..v ..n.'.......
0020  8c 86 55 fc 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b  ..U.............
```

Figure 5: TLS padding

[7] https://tools.ietf.org/html/rfc5246#section-9

Thus, possible paddings are:

- `01 01`
- `02 02 02`
- `03 03 03 03`
- etc.

Testing the block where padding is included does not require the IV and can be done in $N \times aes\_decrypt\_time$. The probability that a random block decrypts to a valid padding is $\frac{1}{256^2}$. This reduces the time taken from quadratic to linear and by trying the next packet in the stream for valid padding upon decryption reduces False Positive rate even further.

5.3.3 AES CBC for TLS 1.1 and higher

For TLS versions 1.1 and higher, the IV is included before the encrypted payload such that the number of operations is reduced to $N \times (aes\_decrypt\_time + xor\_mask\_time)$ for a FP probability of 2^{32} similar to the one in RC4. Reducing any duplicate candidates can be done using the padding like before.

5.3.4 AES GCM

Galois Counter Mode is used as an Authenticated Encryption with Associated Data (AEAD) mode in modern TLS implementations as it has speed advantages. Basically, it can be viewed as AES in Counter Mode with a built-in MAC primitive:

- the Counter Mode part requires the AES key and a nonce
- the authentication part requires authentication data and the encryption of a null-byte block with the AES key (resulting H).

The AES key and 4 bytes from the nonce are taken from the keying material. The rest of the nonce precedes the encrypted payload as the explicit nonce. The additional data [8] depends on the packet sequence number.

Thus, applying the same known plaintext attack technique as before depends on both the key and the nonce making the running time once again quadratic. To reduce it we need further insight into the GCM mode in order to use the information given by the tag.

Since we can mimic the authentication algorithm, it is easy to obtain a value H for each key candidate and follow a modified flow. First multiply the ciphertext blocks with H and instead of combining the first encrypted counter to get the tag, use the tag to get to the encrypted counter. Given that we know the value of the final tag we obtain the encryption of the first counter-mode plaintext block with the key. Decrypting it should reveal the explicit nonce, the secret salt and the counter equal to zero. The probability of False Positives is 2^{-96}. Moreover, let us observe that no plaintext must be known!

[8] `https://tools.ietf.org/html/rfc5246#section-6.2.3.3`

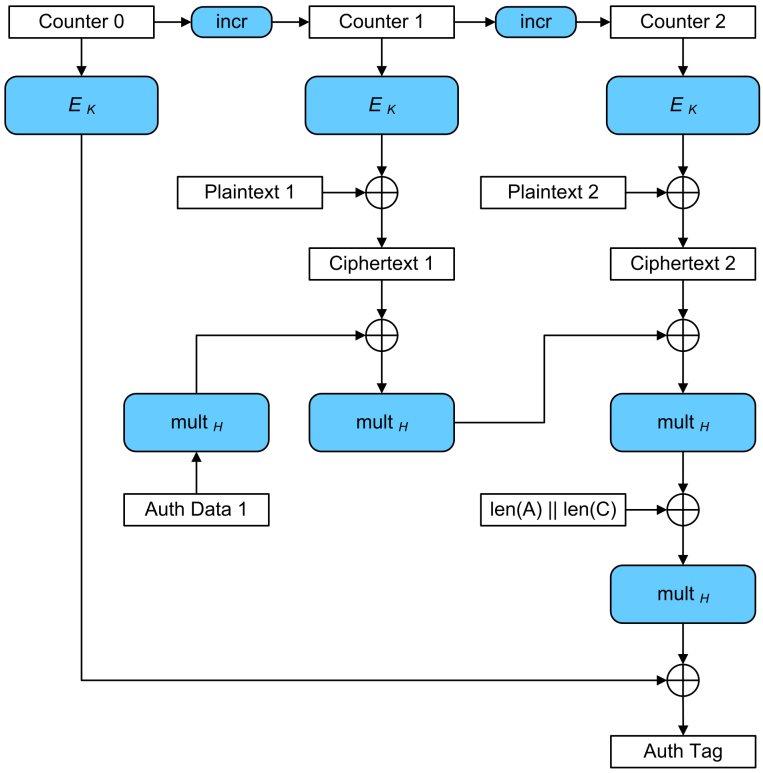

Figure 6: AES GCM operation

6 Benchmarks

In the following, we will present the performance numbers we have obtained in the two main components: Telescope and Telesync.

6.1 Telescope benchmarks

To assess the performance penalties imposed, a couple of tests have been devised for the two main platforms used in the cloud: Linux and Windows. The tests benchmark the disk space used, the VM pause time and the network delay caused by the page dump.

6.1.1 Linux serial connections

In the first scenario, we test 100 connections in serial from the Linux command line. To avoid the delay imposed by serving the actual HTTPS content we only measure the handshake time:

```
#!/bin/bash
echo > test
for i in `seq 1 100`; do
        openssl s_client -connect nimbus.bitdefender.net:443 < test &>/dev/null
done
```

We get the following time measurements first without the Telescope attached to the VM and then with:

```
# time bash serial.sh        # without Telescope
real        0m2.920s
user        0m1.405s
sys         0m0.444s
# time bash serial.sh        # with Telescope
real        0m3.493s
user        0m1.006s
sys         0m0.746s
```

The overhead imposed is of 500 ms, thus 5 ms on average per connection. Regarding disk space, the memdumps vary between 512K an 2500K in this test run as can be seen in figure 7.

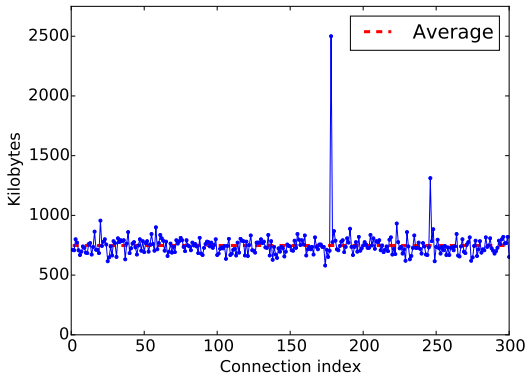

Figure 7: Memory dump size for serial connections on Linux

The VM pause time varies between 0.012 ms and 0.291 ms with an average of 0.058 ms and page dump time (packet delay) varies between 0.181 ms and 2.980 ms with an average of 0.471 ms as can be seen in figure 8.

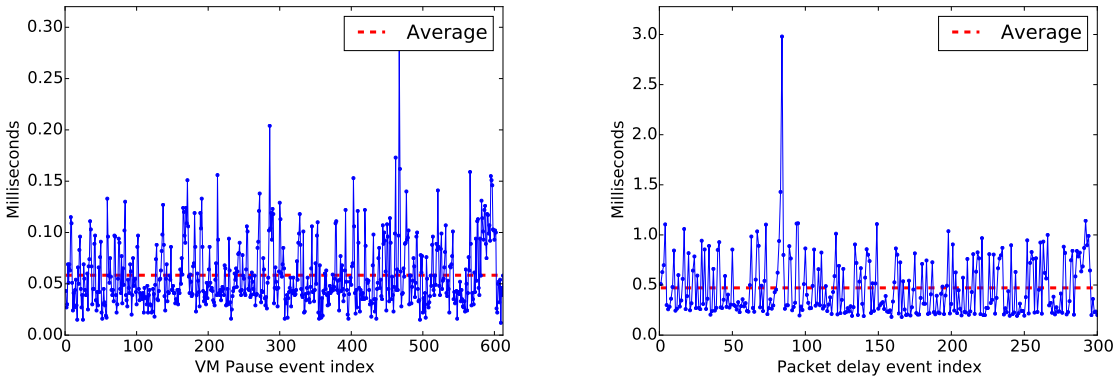

Figure 8: VM pause time and delay for serial connections on Linux

6.1.2 Linux parallel connections

A more plausible case is with connections happening in parallel. To this end, we modify our script in the following way:

```
#!/bin/bash
for i in `seq 1 100`; do
        openssl s_client -connect nimbus.bitdefender.net:443 < test &>/dev/null &
done
while [ 1 ] ; do
        jobs | grep "Running" &>/dev/null
        if [ $? -eq 1 ] ; then
                break;
        fi
done;
```

As before, we present the results with and without Telescope:

```
$ time bash parallel.sh          #without Telescope
real        0m0.539s
user        0m0.278s
sys         0m0.257s
$ time bash parallel.sh          #with Telescope
real        0m1.139s
user        0m0.511s
sys         0m0.582s
```

The overhead imposed is about the same: 600 ms meaning 6 ms per connection.

In this case, the memory dump size has increased because of background activity, the average size being 6968 KB as can be seen in figure 9.

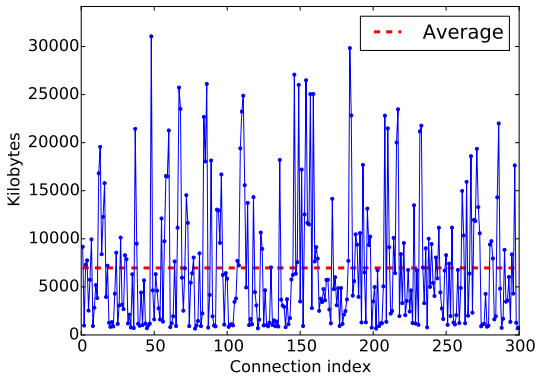

Figure 9: Memory dump size for parallel connections on Linux

The VM pause time varies between 0.016 ms and 0.169 ms with an average of 0.035 ms and page dump time (packet delay) varies between 0.210 ms and 9.684 ms with an average of 2.29 ms as can be seen in figure 10.

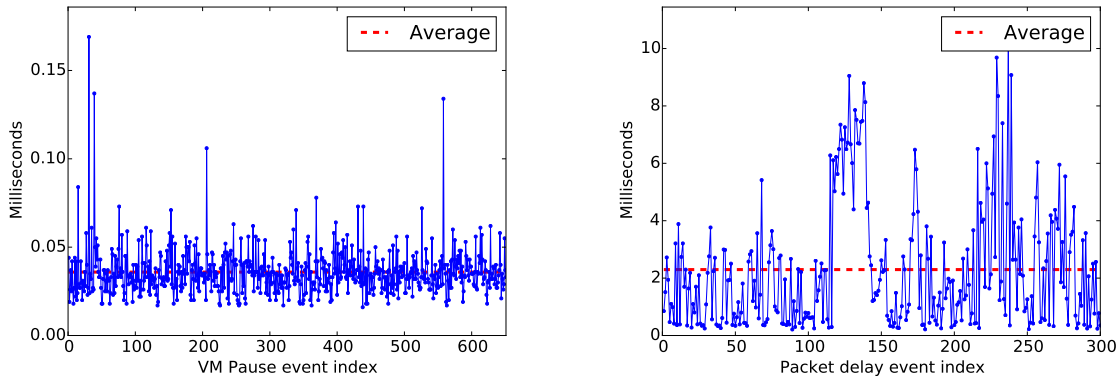

Figure 10: VM pause time and delay for parallel connections on Linux

6.1.3 Windows Firefox browsing session

On Windows, we assess the performance a bit differently: we shall visit 5 HTTPS enabled sites as part of a routine browsing session:

- https://conference.hitb.org
- https://mail.google.com
- https://twitter.com
- https://facebook.com
- https://slack.com

This results in 90 connections logged and as it was expected, the Windows environment modifies more memory pages than its Linux counterpart: on average 14541 KB in our session but with some heavy outliers, such as the 155 MB memdump as can be seen in figure 11.

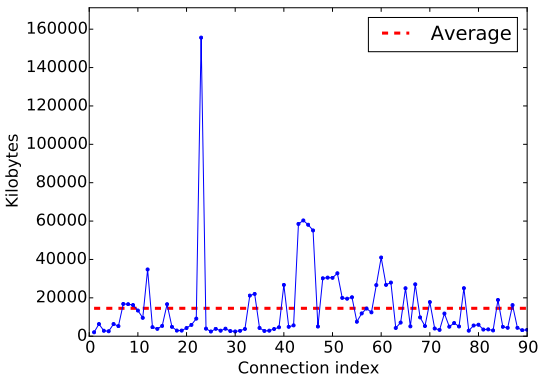

Figure 11: Memory dump size for a browsing session on Windows

The VM pause time varies between 0.031 ms and 0.215 ms with an average of 0.05 ms and page dump time (packet delay) varies between 0.776 ms and 9.997 ms with an average of 5.22 ms as can be seen in figure 12.

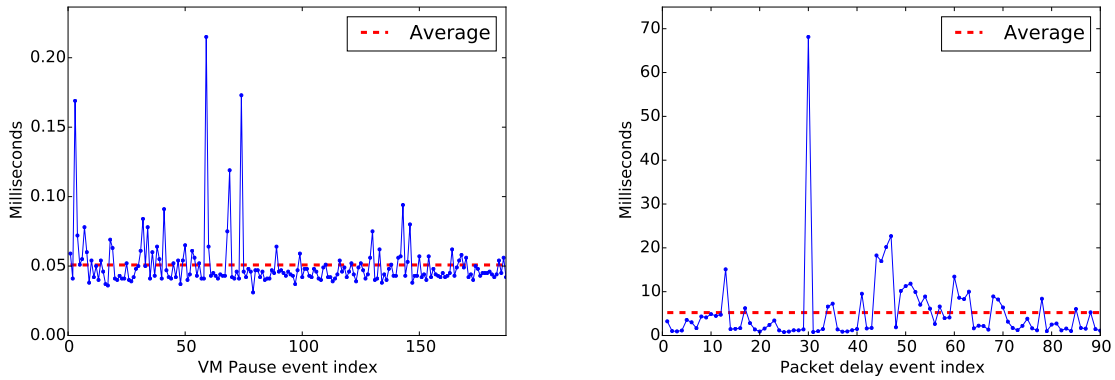

Figure 12: VM pause time and delay for a browsing session on Windows

6.2 Telesync benchmarks

For the crypto benchmarks we will investigate the performance on the most common cipher: AES-GCM, which also requires the most operations to be done in the brute-force process, and specifically we focus on the larger memory dumps from the Windows experiments.

Firstly, we check the raw performance, without heuristics and running on only one core versus six cores:

```
$ du -k
155608          ./telescope-10.10.15.33:49818=104.244.43.39:443-pmEaYq

$ time bash -c "TLS_HEURISTICS=0 OMP_NUM_THREADS=1 /opt/weapons/telescope/telesync
./telescope-10.10.15.33:49818=104.244.43.39:443-pmEaYq"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 6657.724]
[telesync.c]:916 Client write key is: D49A9B4EEFDE497BC88E1B4792555988
[telesync.c]:917 Server write key is: 449E186FE18729257CDAAB7B4C26DC79
$ time bash -c "TLS_HEURISTICS=0 OMP_NUM_THREADS=6 /opt/weapons/telescope/telesync
./telescope-10.10.15.33:49818=104.244.43.39:443-pmEaYq"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 645.495]
[telesync.c]:916 Client write key is: D49A9B4EEFDE497BC88E1B4792555988
[telesync.c]:917 Server write key is: 449E186FE18729257CDAAB7B4C26DC79
```

As expected, the time taken drops considerably when using multiple cores, in this case the speedup is 10X as one thread finds the correct key before the end of its share and forces the other threads to abandon their search too.

But this is not the only optimization that we can employ: the memory dumps are not simply random but structured data and as we have observed, consist of repeating patterns, low entropy segments and null bytes comprise anywhere from 40% to 75%. Thus, we can employ some heuristics given that the Key Material generated from the PRF has pseudo-random statistical properties. Using these heuristics the numbers drop down even further:

```
$ time bash -c "TLS_HEURISTICS=1 OMP_NUM_THREADS=1 /opt/weapons/telescope/telesync
./telescope-10.10.15.33:49818=104.244.43.39:443-pmEaYq"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 3844.105]
[telesync.c]:916 Client write key is: D49A9B4EEFDE497BC88E1B4792555988
[telesync.c]:917 Server write key is: 449E186FE18729257CDAAB7B4C26DC79
$ time bash -c "TLS_HEURISTICS=1 OMP_NUM_THREADS=6 /opt/weapons/telescope/telesync
./telescope-10.10.15.33:49818=104.244.43.39:443-pmEaYq"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 424.872]
[telesync.c]:916 Client write key is: D49A9B4EEFDE497BC88E1B4792555988
[telesync.c]:917 Server write key is: 449E186FE18729257CDAAB7B4C26DC79
```

However, by using heuristics getting a definitive measurement of the speedup is significantly more complicated. In another browsing session, a memory dump 53% bigger (155 MB versus 237 MB) than the previously tested could be processed 3.6 times faster:

```
$ du -k
243436        ./telescope-10.10.15.33:49794<=>54.247.125.40:443-Dym4t0

$ time bash -c "TLS_HEURISTICS=0 OMP_NUM_THREADS=6 /opt/weapons/telescope/telesync
./telescope-10.10.15.33\:49794\<\=\>54.247.125.40\:443-Dym4t0"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 719.757]
[telesync.c]:916 Client write key is: B4767DDEA269402A6325FBF34EDD087A
[telesync.c]:917 Server write key is: 79F4EFFEEB4F7FA1BCF653E2A5D519D7
$ time bash -c "TLS_HEURISTICS=1 OMP_NUM_THREADS=6 /opt/weapons/telescope/telesync
./telescope-10.10.15.33\:49794\<\=\>54.247.125.40\:443-Dym4t0"
Cipher suite is C02F => TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
[Decryption time in ms: 117.322]
[telesync.c]:916 Client write key is: B4767DDEA269402A6325FBF34EDD087A
[telesync.c]:917 Server write key is: 79F4EFFEEB4F7FA1BCF653E2A5D519D7
```

7 Conclusion

As we have shown, TLS decryption is feasible given access to the hypervisor. If a known template VM is employed, key exfiltration can be done instantly because the memory layout is predictable and breakpoints can be easily set from outside the guest machine.

If there is no guarantee on the VM layout, the `Telescope` can be used to create a differential memory dump, without pausing the machine for more than 0.1 ms on average. The resulting dump can be processed offline later in an overwhelming majority of cases in linear time.

By using the `Telescope`, pause time is barely noticeable such that the SLA will not be affected and unless the tenant is actively looking, it is virtually impossible to tell that the techniques we have described are being applied to their VM.