Tutorial 12 - Capturing and decrypting the entire traffic

What are we doing?

We are capturing the entire HTTP traffic. We will also be decrypting traffic where necessary.

Why are we doing this?

In daily life, when operating a web or reverse proxy server errors occur that can only be handled with difficultly come up again and again. In numerous cases there is a lack of clarity about what has just passed over the line or there is disagreement about exactly which end of communication was responsible for the error. In cases such as these it is important to be able to capture the entire traffic in order to narrow down the error to this basis.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).
- Understanding of the minimal configuration in Tutorial 2 (Configuring a minimal Apache server).
- An Apache web server with SSL/TLS support as in Tutorial 4 (Configuring an SSL server)
- An Apache web server with extended access log as in Tutorial 5 (Extending and analyzing the access log)
- An Apache web server with ModSecurity as in Tutorial 6 (Embedding ModSecurity)
- An OWASP ModSecurity Core Rules installation as in Tutorial 7 (Embedding ModSecurity Core Rules
- A reverse proxy as in Tutorial 9 (Setting up a reverse proxy)

Step 1: Using ModSecurity to capture the entire traffic

In Tutorial 6 we saw how we are able to configure ModSecurity to capture the entire traffic from a single client IP address. However, depending on the settings of the `SecAuditLogParts` directive, not all parts of the requests are recorded. Let's have a look at the different options in this directive: The ModSecurity audit engine labels different parts of the audit log using different letter abbreviations. They are as follows:

- Part A: The starting part of a single entry/request (required)
- Part B: The HTTP request headers
- Part C: The HTTP request body (including raw data for a file upload; only if body access was set via `SecRequestBodyAccess`)
- Part E: The HTTP response body (only if body access was enabled via `SecRequestBodyAccess`)
- Part F: The HTTP response headers (without the two date and server headers, set by Apache itself right before leaving the server)
- Part H: Further information from ModSecurity concerning additional information about the request, such as repeated entries in the Apache error log here, the `Action` taken, timing information, etc. It's worth taking a look.
- Part I: The HTTP request body in a space-saving version (uploaded files are not fully included, only individual key parameters for these files)
- Part J: Additional information about file uploads
- Part K: A list of all rules that returned a positive answer (the rules themselves are normalized; including all inherited declarations)
- Part Z: End of a single entry/request (required)

In Tutorial 6 we made the following selection for the individual headers:

```
SecAuditLogParts        ABIJEFHKZ
```

We have defined a very comprehensive log. This is the right approach in a lab-like setup. However, in a production environment this is only useful in exceptional cases. A typical variation of this directive in a production environment would thus be:

```
SecAuditLogParts            "ABFHKZ"
```

The request and response bodies are no longer being captured. This saves a lot of storage space, which is important on badly tuned systems. The parts of the body that violate individual rules are nonetheless written to the error log and in Part K. This is sufficient in most cases. However, from case to case, you will still want to capture the entire body. In cases such as these you can use a `ctl` directive for the action part of the `SecRule`. Multiple, additional parts can be selected via `auditLogParts`:

```
SecRule REMOTE_ADDR  "@streq 127.0.0.1"   \
    "id:10000,phase:1,pass,log,auditlog,msg:'Initializing full traffic log',ctl:auditLogParts=+EIJ"
```

Step 2: Using ModSecurity to write the entire traffic of a single session

The first step enables the dynamic modification of audit log parts for a known IP address. But what if we want to permanently enable dynamic logging for selected sessions and, as shown in the example above, expand it to the entire request?

In his ModSecurity Handbook Ivan Ristić describes an example in which a ModSecurity `collection` is employed to generate a separate session which remains enabled beyond an individual request. We'll use this idea as the starting point and write a somewhat more complex example:

```
SecRule TX:INBOUND_ANOMALY_SCORE  "@ge 5" \
  "phase:5,pass,id:10001,log,msg:'Logging enabled (High incoming anomaly score)', \
  expirevar:ip.logflag=600"

SecRule TX:OUTBOUND_ANOMALY_SCORE "@ge 5" \
  "phase:5,pass,id:10002,log,msg:'Logging enabled (High outgoing anomaly score)', \
  expirevar:ip.logflag=600"

SecRule &IP:LOGFLAG               "@eq 1" \
  "phase:5,pass,id:10003,log,msg:'Logging is enabled. Enforcing rich auditlog.', \
  ctl:auditEngine=On,ctl:auditLogParts=+EIJ"
```

In the integration of core rules proposed in the preceding tutorials we have already opened a persistent `collection` based on the IP address of the client making the request. The `collection` which is stored beyond an individual request is suitable for retaining data between different requests.

We'll use this ability to check its `core rules anomaly score` in the logging phase of the request. If it is 5 or higher (corresponding to an alarm or the `critical` level), we set the variable `ip.logflag` and via `expirevar` give it an expiration of 600 seconds. This means that this variable remains in the `IP collection` for ten minutes and then disappears on its own automatically. This mechanism is repeated for the `outgoing anomaly score` in the subsequent rule.

In the third rule we check whether this `Logflag` is set. We earlier saw a wondrous transformation of variable names depending on application in `ModSecurity`. We are seeing it again here, in which `ip.logflag` must be written in a `SecRule` as the variable `IP:LOGFLAG`. We have also become familiar with the `&` at the beginning: It denotes the number of variables of this name (0 or 1). This means we can check for the presence of `ip.logflag`. If the flag is previously set in both rules or at an earlier point in time in the past 10 minutes, then the audit engine is enabled and parts of the log that are not always set in the default configuration are added.

Forcing the audit log, which we have not yet become familiar with, is required, because we want to log requests that don't actually violate any of the rules. This means that the audit log is not yet enabled for the request. We make up for this with this rule.

Altogether, these three rules enable us to precisely monitor a conspicuous client beyond an individual suspicious request and to capture the entire client traffic in the audit log once suspicion has been aroused.

Step 3: Sniffing client traffic with the server/reverse proxy

Traffic between the client and the reverse proxy can normally be well documented using the methods described. In addition, we have the option of documenting traffic on the client. Modern browsers provide options for this and they all seem adequate to me. In practice however there are complications that can make capturing traffic difficult or impossible. Be it a fat client being used outside a browser, the client being used only on a mobile device with an interposed proxy modifying traffic in one direction or the other in such a way that traffic is being modified once more by another module after leaving ModSecurity or

that ModSecurity has no access to traffic whatsoever. In individual cases the latter is actually a problem, because an Apache module can abort the processing of a request and suppress access from ModSecurity by doing so.

In these cases one option is to interpose a separate proxy to capture the traffic. A number of tools are available. `mitmproxy`, in particular, appears to have some very interesting features and I use it to great effect. But because the development of this software is still very dynamic, installation of the current version can be quite demanding, which is why I won't be going into any of the details here. We'll select a somewhat rougher method.

It is therefore possible for entries in the audit log to not match what was received by the client or no longer match what the client originally sent. In these cases it preferable to selectively capture the actual traffic and decrypt the encrypted data. This suggestion, however, runs up against the strong encryption we configured in the fourth tutorial to secure it from snooping. The ciphers we prefer rely on `forward secrecy` for this. This means that a snooper is foiled in such a way that even possession of the encryption key no longer permits snooping. The means no capturing of the traffic of any kind is permitted between the client and the server. Unless we position a process in between that terminates the connection and presents a separate certificate to the client.

In all other cases in which we want to force decryption, but are unable to reconfigure the client, we have to employ a different, weaker type of encryption which is unaware of `forward secrecy`. Something like the `AES256-SHA` cipher which we defined as the only cipher on the client and use to connect to the server. If we are unable to use the cipher on the client side, then we have to weaken encryption on the entire server. It's immediately obvious that this is not desired and is only useful in a few instances. Be it us binding the client to a separate system or putting a time limit on reconfiguration.

As a test, Apache can also be configured using conditional `<if>` directives, presenting another cipher to an individual client. However, this will only work via `SSL renegotiate`. This means that an SSL handshake was carried out using `forward secrecy`, but it was then repeated with a weaker cipher. However, in my tests the commonly used decryption tools `wireshark` and `ssldump` were unable to handle this method. This means switching the server over to weaker encryption for the moment. In terms of security, I strong advise against relying on this variation until all other means have been exhausted.

In the fourth tutorial we operated the local Laboratory Service using the local `snake-oil` key. We are going to use this certificate once more and instruct the server to use the decryptable `AES256-SHA` cipher.

```
    ...

        SSLCertificateKeyFile  /etc/ssl/private/ssl-cert-snakeoil.key
        SSLCertificateFile   /etc/ssl/certs/ssl-cert-snakeoil.pem

        SSLProtocol             All -SSLv2 -SSLv3
        SSLCipherSuite          'AES256-SHA'
        SSLHonorCipherOrder     On

    ...
```

Step 4: Capturing encrypted traffic between the client and the server/reverse proxy

The explanations above set the stage for capturing and then decrypting traffic. We'll be doing it in two steps, first logging the traffic and then decrypting the log. Capturing is also called `pulling a PCAP`. This means we are providing a `PCAP` file, or a network traffic log in `PCAP` format. `PCAP` means `packet capture`. For this we'll either be using the most widespread tool, `tcpdump`, or `tshark` from the `Wireshark` suite. It is also possible to work right away in the `Wireshark` graphical interface.

```
$> sudo tcpdump -i lo -w /tmp/localhost-port443.pcap -s0 port 443
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
...
```

Alternatively:

```
$> sudo tshark -i lo -w /tmp/localhost-port443.pcap -s0 port 443
tshark: Lua: Error during loading:
 [string "/usr/share/wireshark/init.lua"]:46: dofile has been disabled due to running Wireshark as superuse
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback'
```

```
...
```

Here, the two commands that generate an identical log have been instructed to listen to the local `lo` interface and port 443 and to write to the file `localhost-port443.pcap`. The `-s0` option is important. This is referred to as the `snap length` or `capture size`. This indicates exactly how much data to capture from an IP packet. In our case we want the entire packet. The instruction for this is the value 0, which means everything.

These commands are used to start the log and we can now trigger the traffic in a second window. Let's give it a try using `curl`:

```
$> curl -v --ciphers AES256-SHA -k https://127.0.0.1:443/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 443 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
* SSLv3, TLS handshake, CERT (11):
* SSLv3, TLS handshake, Server finished (14):
* SSLv3, TLS handshake, Client key exchange (16):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSL connection using AES256-SHA
...
```

If the response we wanted was returned by the server, we can quit the log using `CTRL-c` in the `sniffing` window.

```
$> sudo tcpdump -i lo -w /tmp/localhost-port443.pcap -s0 port 443
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
^C15 packets captured
30 packets received by filter
0 packets dropped by kernel
```

Step 5: Decrypting traffic

Let's try to decrypt the `PCAP` file. We'll again be using `tshark` from the `Wireshark` suite. The `GUI` also works, but is less comfortable. What's important now is to pass the key we used on the server to the tool.

```
$> sudo tshark -r /tmp/localhost-port443.pcap -o "ssl.desegment_ssl_records: TRUE"\
-o "ssl.desegment_ssl_application_data: TRUE" \
-o "ssl.keys_list: 127.0.0.1,443,http,/etc/ssl/private/ssl-cert-snakeoil.key" \
-o "ssl.debug_file: /tmp/ssl-debug.log"
Running as user "root" and group "root". This could be dangerous.
  1   0.000000    127.0.0.1 -> 127.0.0.1    TCP 74 33517 > https [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK
  2   0.000040    127.0.0.1 -> 127.0.0.1    TCP 74 https > 33517 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS
  3   0.000088    127.0.0.1 -> 127.0.0.1    TCP 66 33517 > https [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=42
  4   0.001381    127.0.0.1 -> 127.0.0.1    SSL 161 Client Hello
  5   0.001470    127.0.0.1 -> 127.0.0.1    TCP 66 https > 33517 [ACK] Seq=1 Ack=96 Win=43776 Len=0 TSval=4
  6   0.002338    127.0.0.1 -> 127.0.0.1    TLSv1.2 865 Server Hello, Certificate, Server Hello Done
  7   0.002417    127.0.0.1 -> 127.0.0.1    TCP 66 33517 > https [ACK] Seq=96 Ack=800 Win=45312 Len=0 TSval
  8   0.004330    127.0.0.1 -> 127.0.0.1    TLSv1.2 408 Client Key Exchange, Change Cipher Spec, Finished
  9   0.018200    127.0.0.1 -> 127.0.0.1    TLSv1.2 141 Change Cipher Spec, Finished
 10   0.019624    127.0.0.1 -> 127.0.0.1    TLSv1.2 199 Application Data
 11   0.028515    127.0.0.1 -> 127.0.0.1    TLSv1.2 428 Application Data, Application Data
 12   0.029827    127.0.0.1 -> 127.0.0.1    TLSv1.2 119 Alert (Level: Warning, Description: Close Notify)
 13   0.030056    127.0.0.1 -> 127.0.0.1    TCP 66 33517 > https [FIN, ACK] Seq=624 Ack=1237 Win=46976 Len=
 14   0.037327    127.0.0.1 -> 127.0.0.1    TLSv1.2 119 Alert (Level: Warning, Description: Close Notify)
 15   0.037417    127.0.0.1 -> 127.0.0.1    TCP 54 33517 > https [RST] Seq=625 Win=0 Len=0
```

Not much is legible here yet. But when we apply the `debug file`, then we see the traffic in it.

```
$> cat /tmp/ssl-debug.log

Wireshark SSL debug log

Private key imported: KeyID bb:70:71:21:26:c6:6f:79:82:93:1a:08:ab:f9:db:1f:...
ssl_load_key: swapping p and q parameters and recomputing u
ssl_init IPv4 addr '127.0.0.1' (127.0.0.1) port '443' filename '/etc/ssl/private/ssl-cert-snakeoil.key' pas
ssl_init private key file /etc/ssl/private/ssl-cert-snakeoil.key successfully loaded.
association_add TCP port 443 protocol http handle 0x1af0f10

dissect_ssl enter frame #4 (first time)
ssl_session_init: initializing ptr 0x7f0044d42438 size 688
  conversation = 0x7f0044d41e98, ssl_session = 0x7f0044d42438
  record: offset = 0, reported_length_remaining = 95
dissect_ssl3_record: content_type 22 Handshake
decrypt_ssl3_record: app_data len 90, ssl state 0x00
association_find: TCP port 33517 found (nil)
packet_from_server: is from server - FALSE
decrypt_ssl3_record: using client decoder
decrypt_ssl3_record: no decoder available

...



ssl_generate_keyring_material ssl_create_decoder(client)
ssl_create_decoder CIPHER: AES256
decoder initialized (digest len 20)
ssl_generate_keyring_material ssl_create_decoder(server)
ssl_create_decoder CIPHER: AES256
decoder initialized (digest len 20)
ssl_generate_keyring_material: client seq 0, server seq 0
ssl_save_session stored session id[0]:
ssl_save_session stored master secret[48]:

...

ssl_decrypt_record: allocating 160 bytes for decrypt data (old len 96)
Plaintext[128]:
| db 2f 9e 70 d4 79 7e 51 18 a7 6e 32 1f 95 8f b6 |./.p.y~Q..n2....|
| 47 45 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c 20 |GET /index.html |
| 48 54 54 50 2f 31 2e 31 0d 0a 55 73 65 72 2d 41 |HTTP/1.1..User-A|
| 67 65 6e 74 3a 20 63 75 72 6c 2f 37 2e 33 35 2e |gent: curl/7.35.|
| 30 0d 0a 48 6f 73 74 3a 20 31 32 37 2e 30 2e 30 |0..Host: 127.0.0|
| 2e 31 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d |.1..Accept: */*.|
| 0a 0d 0a 96 42 bc 7a 70 a9 e1 8c b7 38 00 cc ca |....B.zp....8...|
| 6a 90 e9 08 9c d5 b9 08 08 08 08 08 08 08 08 08 |j...............|
ssl_decrypt_record found padding 8 final len 119
checking mac (len 83, version 303, ct 23 seq 1)
tls_check_mac mac type:SHA1 md 2

...

Plaintext[256]:
| f1 0b 2a 1a bc 28 29 32 cf 40 98 6b 65 7f f0 a4 |..*..()2.@.ke...|
| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
| 0a 44 61 74 65 3a 20 57 65 64 2c 20 30 32 20 4d |.Date: Wed, 02 M|
| 61 72 20 32 30 31 36 20 31 31 3a 31 35 3a 30 34 |ar 2016 11:15:04|
| 20 47 4d 54 0d 0a 53 65 72 76 65 72 3a 20 41 70 | GMT..Server: Ap|
| 61 63 68 65 0d 0a 4c 61 73 74 2d 4d 6f 64 69 66 |ache..Last-Modif|
| 69 65 64 3a 20 4d 6f 6e 2c 20 31 31 20 4a 75 6e |ied: Mon, 11 Jun|
| 20 32 30 30 37 20 31 38 3a 35 33 3a 31 34 20 47 | 2007 18:53:14 G|
| 4d 54 0d 0a 45 54 61 67 3a 20 22 32 64 2d 34 33 |MT..ETag: "2d-43|
| 32 61 35 65 34 61 37 33 61 38 30 22 0d 0a 41 63 |2a5e4a73a80"..Ac|
| 63 65 70 74 2d 52 61 6e 67 65 73 3a 20 62 79 74 |cept-Ranges: byt|
| 65 73 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 |es..Content-Leng|
| 74 68 3a 20 34 35 0d 0a 43 6f 6e 74 65 6e 74 2d |th: 45..Content-|
| 54 79 70 65 3a 20 74 65 78 74 2f 68 74 6d 6c 0d |Type: text/html.|
| 0a 0d 0a 48 d5 2d 0c 88 7a b8 8c 31 8a d1 97 cc |...H.-..z..1....|
| c9 5d cd a4 6b 88 e3 08 08 08 08 08 08 08 08 08 |.]..k...........|
ssl_decrypt_record found padding 8 final len 247
```

The HTTP traffic is now legible, even if in a somewhat difficult format.

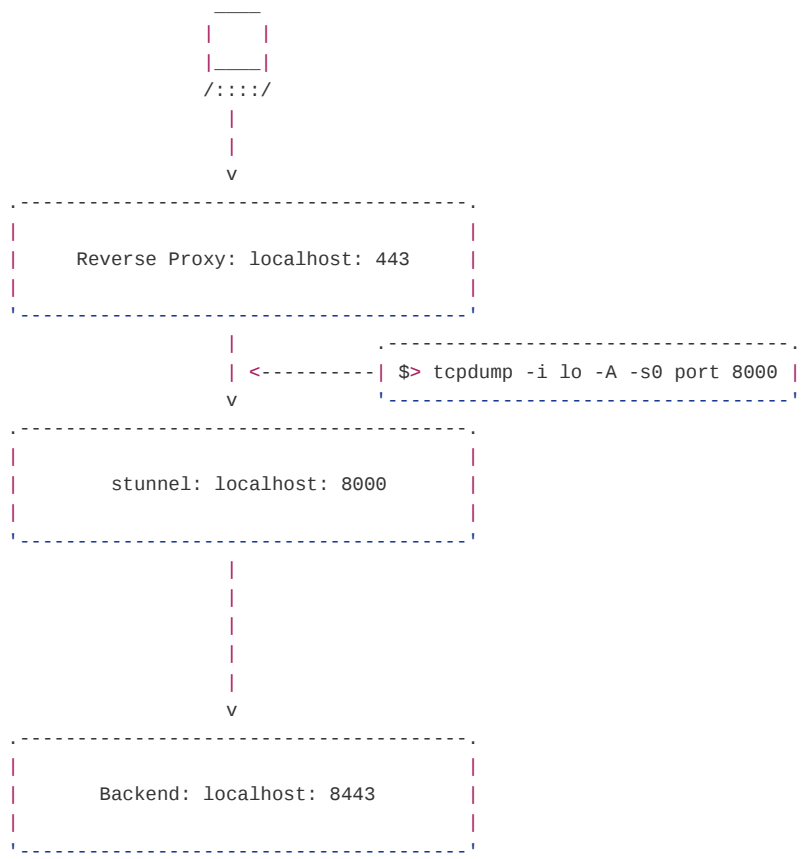
Step 6: Sniffing traffic between the reverse proxy and the application server

The ModSecurity audit log is written after the response to a request is sent. This already makes it clear that the audit log is primarily interesting for what may possibly be the final version of the response. On a reverse proxy this version of the request and above all the response do not necessarily match what the backend system actually sent, because the different Apache modules may have already intervened in the traffic. In order to capture this traffic we will be needing something else. The `mod_firehose` module is present in the development branch of the Apache web server. It can be used to capture and log virtually any place in the traffic. However, the developer community has decided not to include the module in Apache 2.4, but to wait until a later version.

This means that we are again confronted with the problem of having to decrypt network traffic. We can define the `cipher` being used on the reverse proxy side. This is done via the `SSLProxyCipherSuite` directive. But this will only work if we obtain the keys from the application server and client in order to convert the encryption back into plain text. If this is the case, the process is the one described above.

However, the application server key is normally not accessible, so we will have to turn to an alternative. We interpose a small `stunnel` tool between the reverse proxy and the backend. `stunnel` takes over the encryption of the backend for us. This enables the reverse proxy to talk to `stunnel` in plain text, giving us the opportunity to capture this connection 1:1. In order to disable all other snoopers we will be operating `stunnel` on the reverse proxy itself using a local IP address and a separate port. Afterwards, encryption then takes place between `stunnel` and the backend. For testing purposes on the local host network interface here. In practice this can certainly be done on a remote server.

A simple sketch of the setup for illustration:



First, the configuration of the reverse proxy:

```
...

    RewriteRule             /proxy/(.*)     http://localhost:8000/$1 [proxy,last]
    ProxyPassReverse        /               http://localhost:8000/


    <Proxy http://localhost:8000/>
```

```
        </Proxy>

    ...
```

And here's the configuration of the `stunnel daemon`:

```
$> cat /tmp/stunnel.conf

foreground = yes
pid = /tmp/stunnel.pid

debug = 5
socket = l:TCP_NODELAY=1
socket = r:TCP_NODELAY=1

[https]
client = yes
accept  = 8000
connect = localhost:8443
TIMEOUTclose = 0
```

The file is fairly self-explanatory, what's important is the `client` option: It instructs `stunnel` to accept plain text connections and to encrypt them to and from the backend. The default value is `no` here, which is the exact opposite behavior. The `TIMEOUTclose` option is an empirical value, which is sometimes found in `stunnel` instructions. The configuration of the backend sever still remains. Because we need a backend with SSL/TLS support, we can no longer use a `socat` backend like we did in Tutorial 9:

```
PidFile logs/httpd-backend.pid

Listen  127.0.0.1:8443

...

<VirtualHost *:8443>
        ServerName localhost
        ServerAlias ubuntu

        SSLEngine               On
        RewriteEngine           On


        SSLCertificateKeyFile   /etc/ssl/private/ssl-cert-snakeoil.key
        SSLCertificateFile      /etc/ssl/certs/ssl-cert-snakeoil.pem
        SSLProtocol             All -SSLv2 -SSLv3
        SSLHonorCipherOrder     On
        SSLCipherSuite          "AES256-SHA"

        <Directory /apache/htdocs>

        </Directory>

</VirtualHost>
```

Because this is the second Apache server being started in parallel, it's important that it doesn't come to loggerheads with the reverse proxy. We have already differentiated the ports. What's important is to also separate the `PidFile` file. We do not normally set it explicitly and are satisfied with the default value. But in our case, we have to set it manually. This is what is happening in the configuration above.

We now start the three different servers in sequence. If we use the `apachex` tool to control Apache, then we will suffer a bit each time `apachex` attempts to start the most recent configuration file. A quick `touch` on the desired configuration solves the problem. For `stunnel` it's important to use the more recent `stunnel4` version. In `Debian/Ubuntu` it is included in a package of the same name. The start is then very easy:

```
$> sudo stunnel4 /tmp/stunnel.conf
stunnel4 /tmp/stunnel.conf
```

```
2016.03.02 16:28:08 LOG5[8254:140331683964736]: stunnel 4.53 on x86_64-pc-linux-gnu platform
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Compiled with OpenSSL 1.0.1e 11 Feb 2013
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Running  with OpenSSL 1.0.1f 6 Jan 2014
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Update OpenSSL shared libraries or rebuild stunnel
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Threading:PTHREAD SSL:+ENGINE+OCSP Auth:LIBWRAP Sockets:POL
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Reading configuration from file /tmp/stunnel.conf
2016.03.02 16:28:08 LOG5[8254:140331683964736]: Configuration successful
```

The complete setup is now ready for our curl request. Let's test it in sequence. First directly on the backend, then via stunnel and finally via the reverse proxy:

```
$> curl -v -k https://localhost:8443/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8443 (#0)
...
> GET /index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8443
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 03 Mar 2016 10:00:04 GMT
* Server Apache is not blacklisted
< Server: Apache
< Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
< ETag: "2d-432a5e4a73a80"
< Accept-Ranges: bytes
< Content-Length: 45
< Content-Type: text/html
<
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host localhost left intact
$> curl -v -k http://localhost:8000/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET /index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8000
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 03 Mar 2016 10:01:04 GMT
* Server Apache is not blacklisted
< Server: Apache
< Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
< ETag: "2d-432a5e4a73a80"
< Accept-Ranges: bytes
< Content-Length: 45
< Content-Type: text/html
<
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host localhost left intact
$> curl -v -k https://localhost:443/proxy/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 443 (#0)
...
> GET /proxy/index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 03 Mar 2016 10:01:29 GMT
* Server Apache is not blacklisted
< Server: Apache
< Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
< ETag: "2d-432a5e4a73a80"
< Accept-Ranges: bytes
```

```
< Content-Length: 45
< Content-Type: text/html
<
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host localhost left intact
```

This worked pretty well. We see the following output in the `stunnel` window:

```
2016.03.03 11:03:49 LOG5[5667:140363675346688]: Service [https] accepted connection from 127.0.0.1:47818
2016.03.03 11:03:49 LOG5[5667:140363675346688]: connect_blocking: connected 127.0.0.1:8443
2016.03.03 11:03:49 LOG5[5667:140363675346688]: Service [https] connected remote server from 127.0.0.1:5459
2016.03.03 11:03:49 LOG3[5667:140363675346688]: transfer: s_poll_wait: TIMEOUTclose exceeded: closing
2016.03.03 11:03:49 LOG5[5667:140363675346688]: Connection closed: 190 byte(s) sent to SSL, 275 byte(s) sen
```

Thus, here `stunnel` reports the incoming connection on `source port` 47818 and that it has established a connection to the backend host on port 8443 with `source port` 54593; then come two numbers about the transfer rate. Overall, we can conclude that the setup works and we are ready to sniff the connection. Let's enable `tcpdump` or `tshark`. Decryption is no longer necessary now, because the connection we are sniffing can be read in plain text between the two localhost `sockets`. That's why it's important that we enable `snap length` and ASCII mode via `-A`.

```
$> sudo tcpdump -i lo -A -s0 port 8000
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
11:07:40.016067 IP localhost.47884 > localhost.8000: Flags [S], seq 2684270112, win 43690, options [mss 654
E..<..@.@.\.............@... .........0.........
..V4........
11:07:40.016103 IP localhost.8000 > localhost.47884: Flags [S.], seq 3592202505, ack 2684270113, win 43690,
E..<..@.@.<...........@.....     ...!.....0.........
..V4..V4....
11:07:40.016154 IP localhost.47884 > localhost.8000: Flags [.], ack 1, win 342, options [nop,nop,TS val 631
E..4..@.@.\.............@...!...
...V.(.....
..V4..V4
11:07:40.016647 IP localhost.47884 > localhost.8000: Flags [P.], seq 1:191, ack 1, win 342, options [nop,no
E.....@.@.[.............@...!...
...V........
..V4..V4GET /index.html HTTP/1.1
Host: localhost
User-Agent: curl/7.35.0
Accept: */*
X-Forwarded-For: 127.0.0.1
X-Forwarded-Host: localhost
X-Forwarded-Server: localhost
Connection: close


11:07:40.016738 IP localhost.8000 > localhost.47884: Flags [.], ack 191, win 350, options [nop,nop,TS val 6
E..4.>@.@.=...........@.....
.......^.(.....
..V4..V4
11:07:40.041573 IP localhost.8000 > localhost.47884: Flags [P.], seq 1:231, ack 191, win 350, options [nop,
E....?@.@.<...........@.....
.......^........
..V:..V4HTTP/1.1 200 OK
Date: Thu, 03 Mar 2016 10:07:40 GMT
Server: Apache
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Connection: close
Content-Type: text/html


11:07:40.041627 IP localhost.47884 > localhost.8000: Flags [.], ack 231, win 350, options [nop,nop,TS val 6
E..4..@.@.\.............@...........^.(.....
..V:..V:
11:07:40.041711 IP localhost.8000 > localhost.47884: Flags [P.], seq 231:276, ack 191, win 350, options [no
E..a.@@.@.=T.........@.............^.U.....
```

```
..V:..V:<html><body><h1>It works!</h1></body></html>

11:07:40.041745 IP localhost.47884 > localhost.8000: Flags [.], ack 276, win 350, options [nop,nop,TS val 6
E..4..@.@.\.............@...........^.(.....
..V:..V:
11:07:40.042044 IP localhost.47884 > localhost.8000: Flags [F.], seq 191, ack 276, win 350, options [nop,no
E..4..@.@.\.............@...........^.(.....
..V:..V:
11:07:40.047226 IP localhost.8000 > localhost.47884: Flags [F.], seq 276, ack 192, win 350, options [nop,no
E..4.A@.@.=...........@.............^.(.....
..V;..V:
11:07:40.047296 IP localhost.47884 > localhost.8000: Flags [.], ack 277, win 350, options [nop,nop,TS val 6
E..4..@.@.\.............@...........^.(.....
..V;..V;
```

We've done it! We are capturing the connection to the backend and are now sure about the traffic being exchanged between the two servers. In practice it is often unclear whether an error is actually being caused on the application server or perhaps on the reverse proxy after all. Using this construct that does not touch the SSL configuration of the backend server, we have a tool giving us a final answer in these relatively frequent cases.

References

- Ivan Ristić: ModSecurity Handbook
- Mod_firehose
- mitmproxy
- Wireshark SSL Howto including a Step by Step guide
- Stunnel Howto

License / Copying / Further use