Man in the Middle Attacks

Network Security Lab – University of Trento – 2016-04-27

Ali Davanian – Amit Kumar Gupta – Jan Helge Wolf

Introduction

Man in the Middle (MitM) attacks are attacks in which an adversary is able to intercept, manipulate, and/or forge network traffic between two communication partners due to his location between the parties. The attacker transparently relays the traffic, such that both partners believe to be talking to each other directly, while in reality both communicate with the adversary. In this sense, an active MitM attack relies on the inability (or lack of willingness) of the two parties to authenticate each other; to prevent passive MitM attacks, additional encryption of the communication channel is necessary.

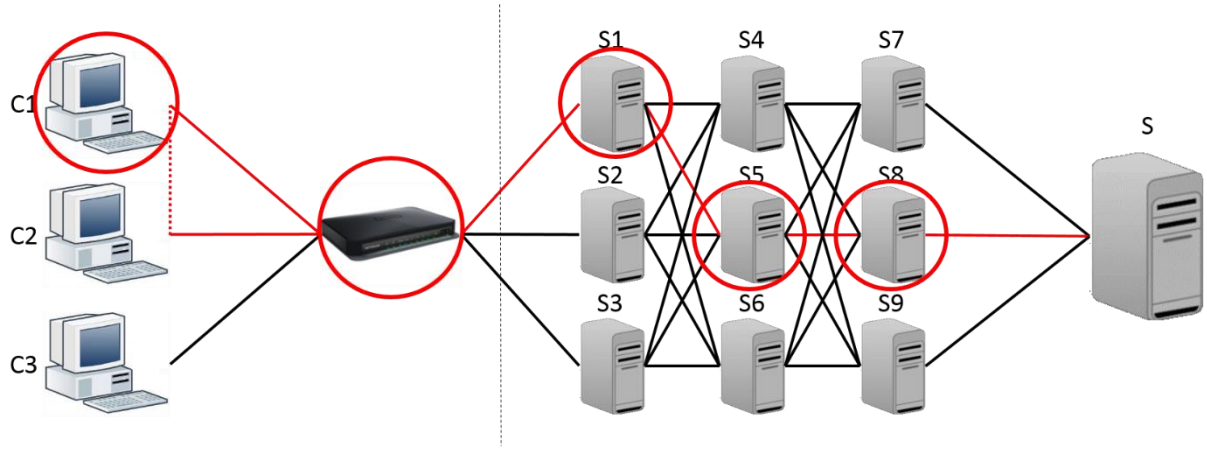

Figure 1: Schematic depiction of a typical network route between a client (C2) and a server (S)

As Figure 1 shows, all devices between the two communication partners (here *C2* and *S*) could perform a MitM attack. This typically includes the provider of the local network infrastructure (routers etc.) as well as the Internet Service Provider (ISP) of the client, as well as the backbones routing the internet traffic and possibly the ISP of the server. To the University of Trento audiences of this article, Opera Universitaria is the best example. Opera provides student wired and wireless access to the internet and at any time it can act as a MitM.

Additionally, any other device connected to the same local network as the client could perform an Address Resolution Protocol (ARP) spoofing attack, imitating the router to all other machines. This way, the client will believe, for example, C1 to be the router and send all its traffic to it. C1 will then forward the traffic to the actual router and act as a MitM on the connection. Again in the context of University of Trento, all students living on the same floor can be a MitM at their will.

In our lab, we demonstrate MitM attacks on the Hypertext Transfer Protocol (HTTP) and HTTP over Transport Layer Security (TLS), which is commonly referred to as HTTPS. Since HTTP is an unencrypted, unauthenticated protocol by design, performing a MitM attack on it is trivial given the necessary physical position between the communication partners as described above. Due to the time constraint of this lab, we assume that the attacker already managed to place herself between Alice and Bob. To simulate this, we manually use proxy capability of browsers to put the attacker in the “Middle”.

TLS, however, employs a two-step process to ensure the confidentiality of the communication between the two partners. In the first step, asymmetric cryptography is used to negotiate a session

key, which is later used to symmetrically encrypt the communication content. For this reason, a MitM cannot simply read or manipulate HTTPS network traffic passing through it.

In the rest of this report, we first present our lab configuration. After that, we provide a brief description of MitM attacka on HTTP given its simplicity. Next we focus on HTTPS MitM attacks, focusing on how an attacker could try to attack the key negotiating step to perform a MitM attacks, and how the TLS protocol defends against this type of attack.

Lab configuration

To demonstrate the attacks, we set up two Virtualbox virtual machines, referred to as *victim* and *attacker*. To simplify the setup, the victim also acts as a web server hosting a faux online banking web application. The two virtual machines are connected by an internal network using static IP addresses (no DHCP server). In order to emulate a MitM attack, the victim machine connects to the online banking application using the attacker machine as a proxy (in a real-world scenario, the attacker might be any of the players described in the previous chapter). The attacker will then try to eavesdrop on the communication between the victim and the server (in this case, itself), as well as actively manipulate network traffic (see Figure 2).

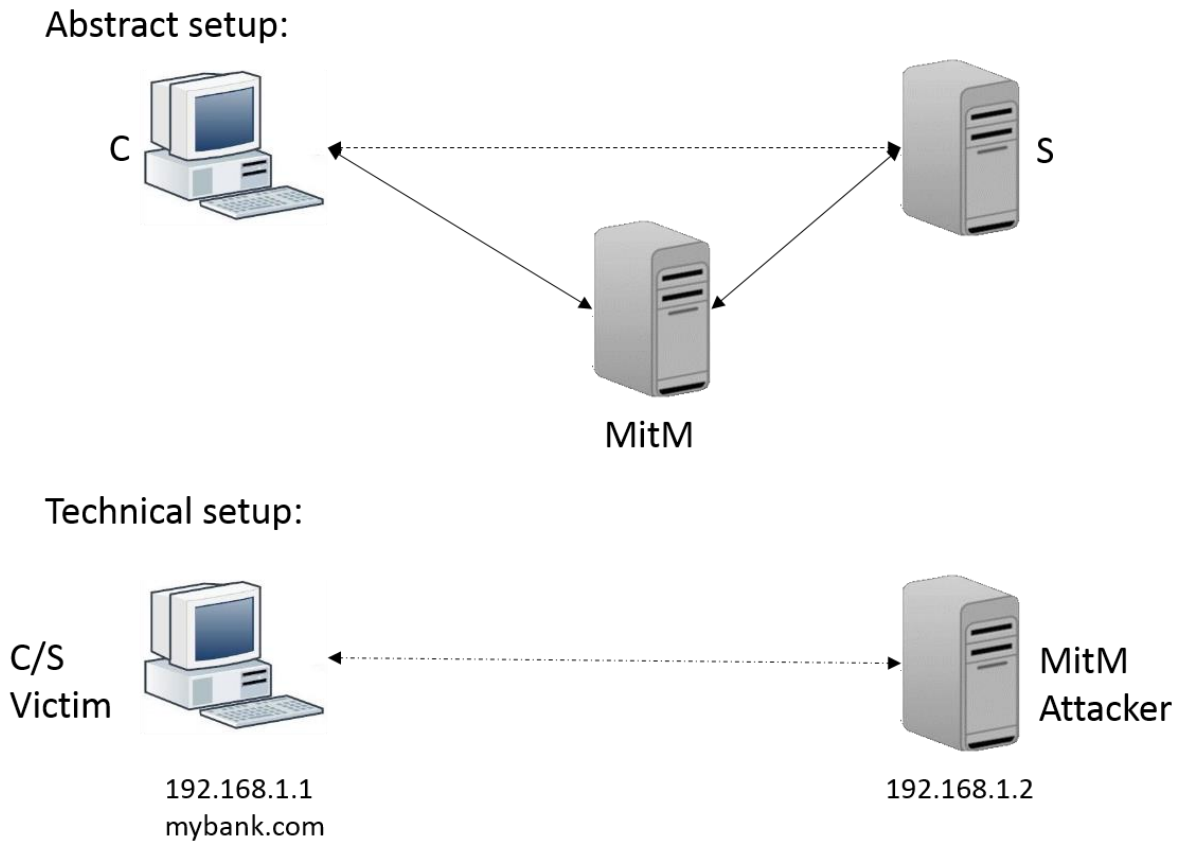

Figure 2: Network setup of the virtual machines used during the lab

The victim VM (IP address 192.168.1.1) is an Ubuntu 14.04 Desktop machine. To simplify the management of the proxy settings in the browser, we installed the *FoxyProxy Basic* plugin for Firefox[1]. Additionally, to serve the online banking application, we installed a classic LAMP (Linux, Apache, MySQL, PHP) stack:

[1] See https://addons.mozilla.org/en-US/firefox/addon/foxyproxy-basic/.

```
$ sudo apt-get install apache2

$ sudo apt-get install mysql-server libapache2-mod-auth-mysql
    php5-mysql

$ sudo apt-get install php5 libapache2-mod-php5
```

After installing Apache, you also need to install the following modules:

```
$ sudo a2enmod headers

$ sudo a2enmod ssl
```

After generating a self-signed X.509 certificate for the domains `mybank.com`, `ssl.mybank.com` and `secure.mybank.com`[2], we copied the crt file into `/etc/ssl/certs` and the key file into `/etc/ssl/private`.

Additionally, we added the hosts file (`/etc/hosts`) as to make sure these domains are resolved to localhost. Following entries must be added in the hosts file of both attacker and the victim: `mybank.com`, `ssl.mybank.com` and `secure.mybank.com`. The corresponding IP addresses are:

On Victim's hosts file: 127.0.0.1

On Attacker's hosts file: 192.168.1.1

We added virtual host files for the three domains mentioned[3] to the `/etc/apache/sites-enabled` directory and deployed the web applications to the `/var/www/` folder[4]. We also added security exceptions so that Firefox would accept the self-signed certificate. To do that, open `https://secure.mybank.com` and accept the security exception.

Finally, we set up the web application in the `/var/www/mybank.com`, `/var/www/ssl.mybank.com`, and `/var/www/secure.mybank.com` directories.

You also need to setup mysql[5]:

```
$ mysql –u root [-p password]

$ CREATE DATABASE mybank;

$ USE mybank;

$ SOURCE mybank.sql;

$ CREATE USER 'mybank'@'localhost' IDENTIFIED BY
    'DC9VYaixgtQcYiLu9jSd';

$ GRANT ALL PRIVILEGES ON mybank.* TO 'mybank'@'localhost';
```

The attacker VM (IP address 192.168.1.2) is an Ubuntu 14.04 Server machine. In order for it to serve as a proxy, we installed mitmproxy[6] following the official installation instructions[7]:

---

[2] The cert and key files are attached to this report as `mybank.com.crt` and `mybank.com.key`.

[3] The necessary Apache configuration files are attached to this report as `apache2.zip`.

[4] The web application files are attached to this report as `mybank.zip`.

[5] mybank.sql is attached to this report.

[6] See https://mitmproxy.org/.

[7] See http://docs.mitmproxy.org/en/stable/install.html.

```
$ sudo apt-get install python-pip python-dev libffi-dev libssl-
    dev libxml2-dev libxslt1-dev libjpeg8-dev zlib1g-dev

$ sudo pip install mitmproxy
```

The credentials of the Ubuntu users used during the lab are `victim`/`victim` and `attacker`/`attacker`, respectively.

To set up the internal network, right-click the virtual machine in Virtualbox and enter the virtual machine settings. In the Network tab, set the first network adapter to “Internal network” and the network name to “intnet". Repeat these steps for both virtual machines. Then, inside the virtual machines, edit the `/etc/network/interfaces` file as such:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 192.168.100.<victim:1|attacker:2>
netmask 255.255.255.0
gateway 0.0.0.0
```

MitM attacks on HTTP

To perform a passive attack on HTTP, we start the attacker machine, log in and start mitmproxy:

```
$ mitmproxy
```

Subsequently, we start the victim machine, log in, open Firefox, and set the attacker as a proxy using FoxyProxy:

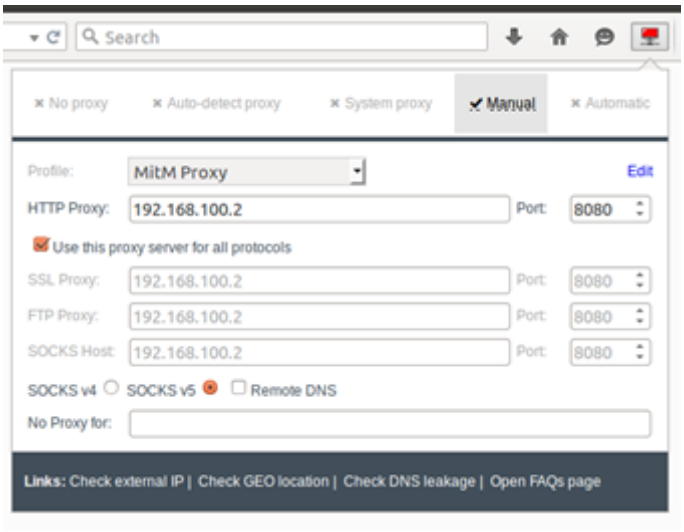

Figure 3: Proxy settings in FoxyProxy

Then we visit `http://mybank.com` and log in with `user`/`user` as credentials.

mitmproxy is an interactive console application that can be used to monitor, intercept, and manipulate HTTP traffic. By looking at the attacker machine, we should now be able to see at least three requests to the mybank.com domain, with one of them being a POST request. Using the arrow keys, we navigate to this request and open the flow detail view[8] via the Enter key.

[8] mitmproxy refers to one request-response pair as a *flow*.

In the Request view, which is opened by default, we can see the details of the POST request made to log in to the banking application, including the URL-encoded form fields which contain the credentials of the user (see Figure 4). Using the Tab key, we could also inspect the details of the response, which in this case consists of a redirect to `http://mybank.com/overview.php`. This example shows that any MitM can eavesdrop on unencrypted HTTP connections without any further complication, reading all information sent via this channel.

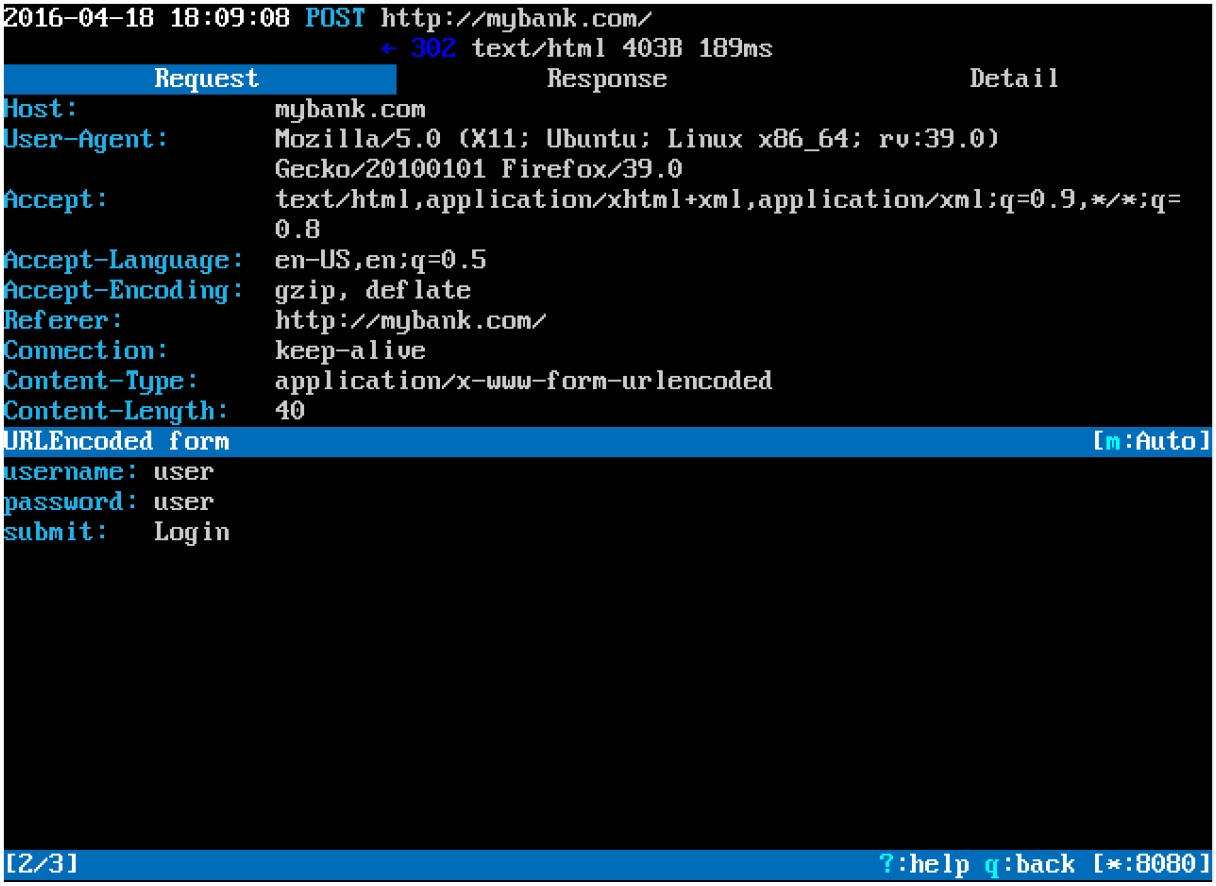

Figure 4: Flow detail view of mitmproxy showing user credentials POSTed to a login form

MitM attacks on HTTPS

As described earlier, HTTPS avoids simple MitM attacks by employing strong encryption protecting the content of the network traffic. Therefore, it is not possible to simply eavesdrop on the channel. This is why mitmproxy (and other proxy solutions) try a different technique when challenged with HTTPS connections: *TLS termination*. When mitmproxy notices that a client tries to establish an HTTPS connection to a server, it imitates this server to the client, while at the same time building up an encrypted connection to the actual server. If this endeavor is successful, we end up with not one encrypted connection between the client and the server, but two separate links: one between the client and the proxy, and another one between the proxy and the server (see Figure 5).

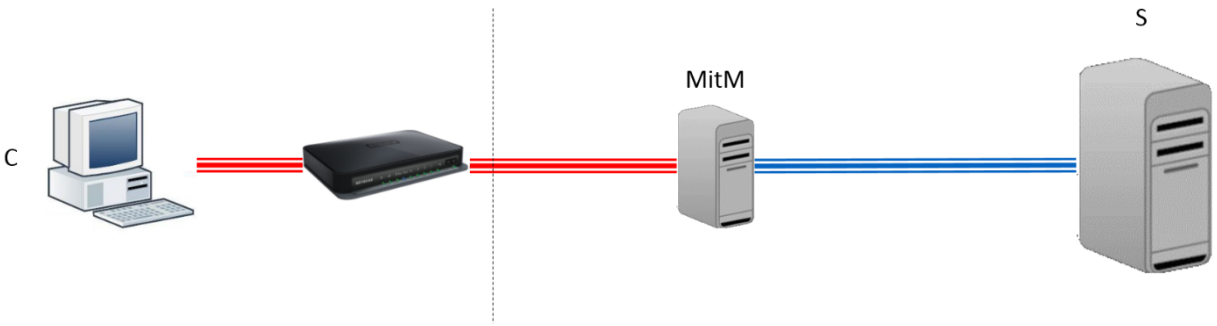

Figure 5: TLS termination

Technically, this is done as follows: mitmproxy possesses a self-signed certificate, which can be used to sign other certificates. A certificate is basically a public-private key pair with an additional "stamp" on it, certifying certain properties of the keypair – most notably, who owns the private key. Self-signed means that in this case, the owner of the certificate himself certified these properties.

When the HTTPS connection is established, mitmproxy on the fly generates a new certificate for the visited page and signs it with its root certificate, thereby pretending to be the legitimate server of the visited domain. Thus, TLS termination is not an attack on the encryption, but on the authentication part of the protocol.

To prevent this kind of attack – in this model, anybody can pretend to be any server in the world – TLS employs a hierarchy-based trust model. This means that any client (e.g., any browser) trusts a number of organizations, called *Certification Authorities* (CAs), with certifying other people's key pairs. Since the mitmproxy certificate is not in this list of trusted CAs, the browser will not trust the connection and warn the user. In our lab setup, this can be replicated by accessing `https://ssl.mybank.com` both with and without the proxy enabled (see Figure 6).

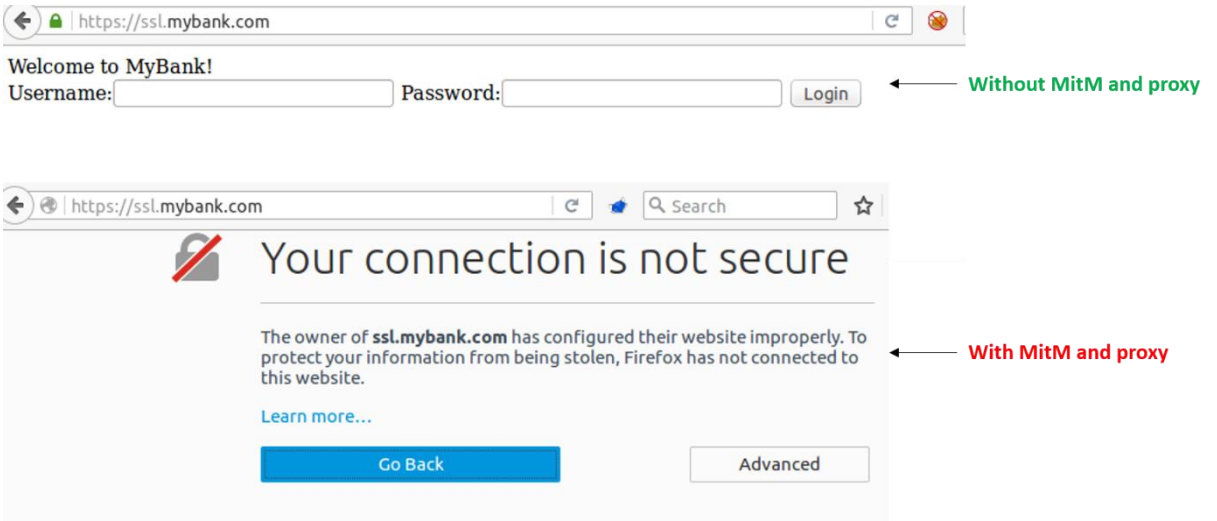

Figure 6: Trusted (top) and rejected (bottom) TLS connections

sslstrip

At the moment, no effective attacks are publicly known against the current TLS version 1.2. However, a MitM can employ a number of measures to prevent a TLS connection being established in the first

place. Moxie Marlinspike at Black Hat 2009 first presented this type of attack and dubbed it *sslstrip*[9]. The effectiveness of sslstrip is based on the fact that the vast majority of users do not explicitly open an encrypted connection by typing `https://domain.com`. Instead, encrypted connections are regularly established using HTTP to HTTPS redirects (using the HTTP Location header or client-side), by links, or form action fields pointing to HTTPS locations.

What is common to all these techniques is that the establishment of an HTTPS connection is negotiated via unencrypted HTTP. This allows a MitM to modify the content sent from the server to the client, removing all references to HTTPS. This way, the client never even tries to initiate an HTTPS connection, allowing the proxy to read and modify traffic at its will. On the other side of the connection, the proxy opens an HTTPS connection to the server, since the server is expecting the client to communicate to it in encrypted form.

In our lab, we use the scripting capabilities of mitmproxy to run an sslstrip attack. To this end, we stop mitmproxy (`q`, then `y` to confirm; another `q` might be necessary to leave the flow details view in the beginning), then change into the mitmproxy directory (`cd ~/mitmproxy`) and then start it again with the `sslstrip.py` script (`mitmproxy -s sslstrip.py`).

The sslstrip script (see) contains three functions. The `start` function is called in the very beginning of every flow, independent of it being a request or a response. In this case, it initializes a set of domains we know to support HTTPS and possibly expect HTTPS connections. mitmproxy will establish HTTPS connections to all of those servers.

The `request` function is called whenever a request is intercepted. Its `flow` parameter allows us to tamper with all relevant parts of the request. In this case, we drop any caching-related headers to enforce a new response by the server, and force the connection to be HTTP unless it is headed to one of the domains in the HTTPS set.

The most interesting function, however, is the `response` function, which handles the manipulation of the server response to the client. After decoding the response content (in case compression or similar measures are in use), we first drop all HTTPS security-related headers. Then we search for new domains to be added to the HTTPS set (in form actions and HTTP Location headers), and strip all references to the HTTPS protocol, replacing them `https://` by `http://`. Finally, we mark all cookies sent by the server as unsecure to force the browser to send them even on unencrypted connections.

The effect of this script can be observed by visiting `http://ssl.mybank.com` both with the proxy disabled and enabled. `ssl.mybank.com`'s index page is served via HTTP, however, its login form points to an HTTPS location – with the proxy enabled, this link is replaced by a simple HTTP location. This way, the client does not even know it is supposed to initiate a secure connection and will, thus, fail to do so. Unless the user specifically makes sure an HTTPS connection is established, they will not notice any difference to normal operations. Obviously, the unencrypted traffic can be inspected, intercepted, and manipulated in mitmproxy as usual.

[9] See https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf for the slides of his presentation, https://www.youtube.com/watch?v=MFol6IMbZ7Y for a full recording of the presentation, and https://moxie.org/software/sslstrip/ for the software implementing the attack.

```
from netlib.http import decoded
import re
from six.moves import urllib

def start(context, argv) :

    #set of SSL/TLS capable hosts
    context.secure_hosts = set()

def request(context, flow) :

    flow.request.headers.pop('If-Modified-Since', None)
    flow.request.headers.pop('Cache-Control', None)

    flow.request.scheme = 'http'
    flow.request.port = 80

    #proxy connections to SSL-enabled hosts
    if flow.request.pretty_host in context.secure_hosts :
        flow.request.scheme = 'https'
        flow.request.port = 443

def response(context, flow) :

    with decoded(flow.response) :
        flow.request.headers.pop('Strict-Transport-Security', None)
        flow.request.headers.pop('Public-Key-Pins', None)

        #strip links in form actions
        pattern = re.compile(r"action=.{0,3}https://([^/?\"']+).*",
     re.MULTILINE)
        matches = pattern.search(flow.response.content)

        if (matches) :
            context.secure_hosts.add(matches.group(1))

        #strip links in response body
        flow.response.content =
     flow.response.content.replace('https://', 'http://')

        #strip links in 'Location' header
        if
     flow.response.headers.get('Location','').startswith('https://'):
            location = flow.response.headers['Location']
            hostname = urllib.parse.urlparse(location).hostname
            if hostname:
                context.secure_hosts.add(hostname)
            flow.response.headers['Location'] =
     location.replace('https://', 'http://', 1)

        #strip secure flag from 'Set-Cookie' headers
        cookies = flow.response.headers.get_all('Set-Cookie')
        cookies = [re.sub(r';\s*secure\s*', '', s) for s in cookies]
        flow.response.headers.set_all('Set-Cookie', cookies)
```

Listing 1: sslstrip.py

Active attacks

So far, we have used the attacker's position as a MitM only to eavesdrop on the connection between the user and their online banking platform. Now, we want to perform an active attack, redirecting a wire transfer performed by the user to our own account. Therefore, on the victim machine, we click the "Wire transfer" link, which leads us to the respective form.

On the attacker machine, we activate a so-called *interception filter*, allowing us to "freeze" the connection between the client and the server, to manipulate it, and then release it once we are done. To this end, in the mitmproxy main screen, we type `i` and then `~q | ~s` (the ~ sign can be typed using the Alt Gr and the + key next to Enter). This filter matches both requests and responses, granting us full control over the communication. The interception filter should now be displayed on the blue status line at the bottom of the mitmproxy main screen.

Now we can proceed to perform a wire transfer of an arbitrary amount from account IT9999999999 to IT0000000000. After submitting the request, it is displayed in the mitmproxy in red, indicating that it is being intercepted. When opening the flow details view of the respective request, we should see a screen similar to the one in Figure 7.

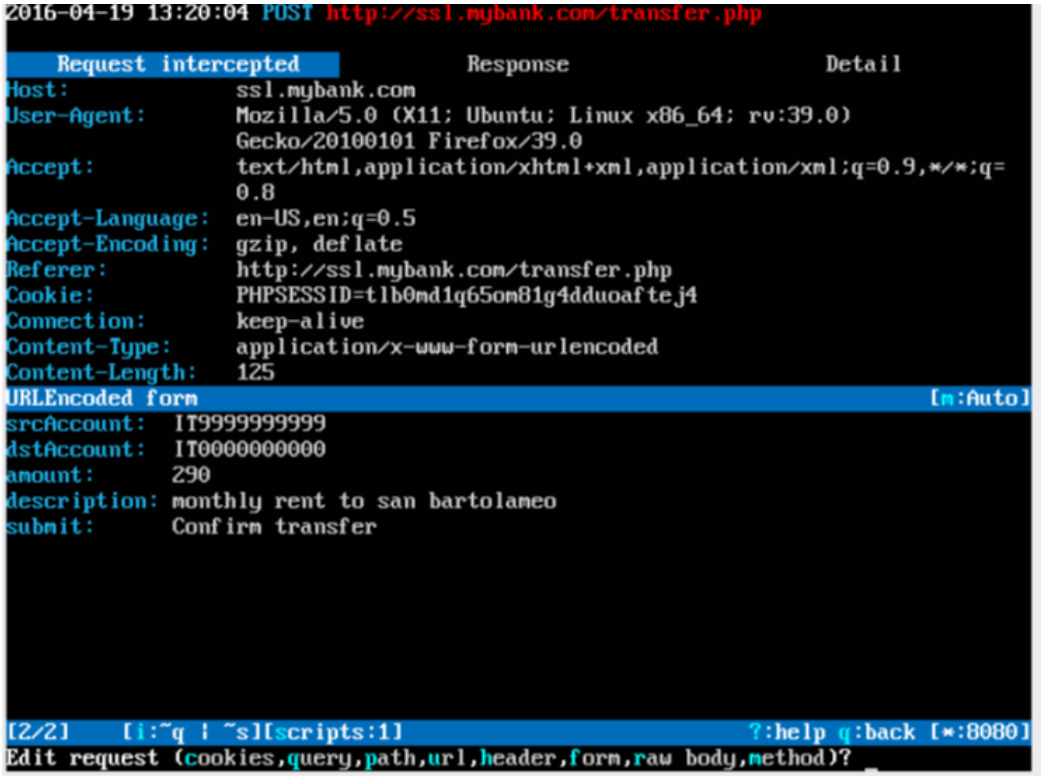

Figure 7: An intercepted request in mitmproxy

Now we cannot only observe the flow details, but can also edit them to our wishes. For example, by pressing `e` followed by `f`, we can edit the values POSTed through the wire transfer form (a field can be edited by pressing Enter, editing is stopped by pressing Esc). By changing the `dstAccount` value to IT5555555555, we can redirect the transfer to our own account. Additionally, we can of course change the amount to be transferred (see Figure 8). By pressing `q` to leave the editing view and then `a`, we accept the request to be sent to the server.

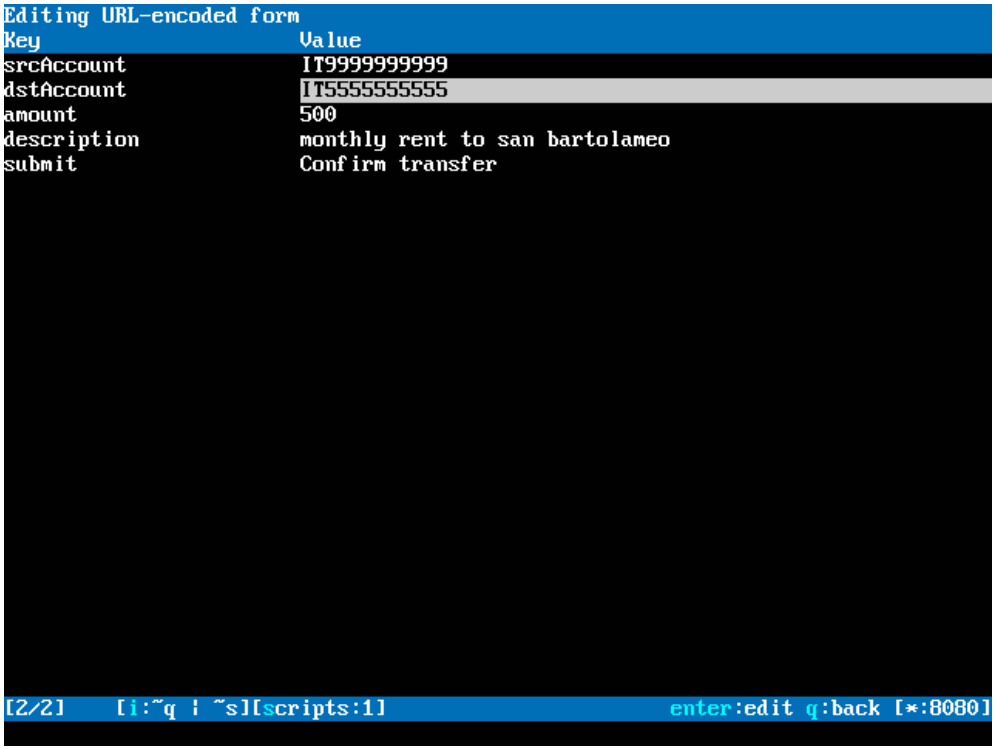

Figure 8: A manipulated request in mitmproxy

The manipulated request has been sent to the server and the money is now on our account. However, the server presents a confirmation message to the user, detailing the amount of the transfer as well as the accounts involved. Therefore, we need to manipulate the response sent by the server, too. To this end, we press Tab to change to the "Response intercepted" view, and press `e` and then `r` to edit the raw HTML response body. After manipulating the body, we close the editor (`Ctrl X`), save our changes to the default location (`y`, Enter), and accept the manipulated response (`a`). By changing the respective values in the confirmation message back to their original values, we deceive the user, letting them believe their transaction was executed as intended (see Figure 9).

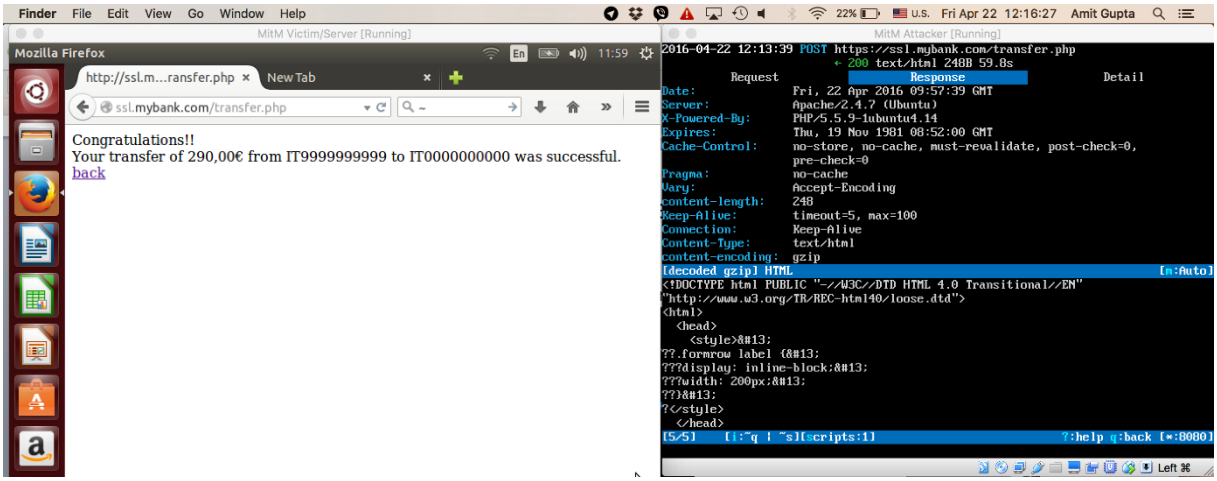

Figure 9: Manipulated response in Firefox (victim) and mitmproxy (attacker)

Finally, to stop the sslstrip script, we exit mitmproxy by pressing `q` and `y`, and then restart it by executing `mitmproxy`.

Certificate forgery

The active attack performed in the previous chapter was made possible by sslstrip, e.g., by preventing encrypted communication from taking place. Even though most users will probably not notice the difference between an encrypted and an unencrypted website due to lack of awareness and the reduction of positive feedback for HTTPS websites on part of the browsers, this attack can be spotted by an advertent user.

Another approach to breaking HTTPS connections is to trick the user into accepting a rogue root certificate. As detailed in the previous chapters, a TLS certificate's authenticity is established by the valid signature of a trusted CA. If the MitM's root certificate is included in the client's list of trusted CAs, the attacker can issue valid-seeming certificates for any domain. Real-life examples of such incidents include the Lenovo Superfish scandal[10] as well as proxy solutions employed by enterprises to monitor the internet traffic of their employees[11].

In this lab, we are going to mimic these scenarios by manually installing the mitmproxy root certificate into Firefox's trust store. To recall the standard behavior, we can visit `https://ssl.mybank.com` while the proxy is activated: Firefox confronts us with a security warning concerning the invalidity of the certificate presented by mitmproxy.

Now, we visit the "magical URL" `mitm.it`, which allows us to install the mitmproxy root certificate by pressing "Other". In the following popup, we check the first checkbox reading "Trust this CA to identify websites" and confirm by clicking "OK". If we now visit `https://ssl.mybank.com`, we can see that Firefox now accepts the certificate presented by mitmproxy and the page is displayed without any error messages. From now on, we can now inspect, intercept and manipulate all HTTPS traffic in mitmproxy as if it was in plain HTTP.

Defenses against MitM attacks

Partially as a response to the sslstrip attacks developed and presented by Marlinspike, two extensions to the HTTP protocol were developed: HTTP Strict Transport Security (HSTS) and Public Key Pinning Extension for HTTP (HPKP). Apart from those mechanism, which require both a client conforming to the respective standards and support by the websites/servers visited, clients can take a number of precautions.

HTTP Strict Transport Security

HSTS[12] is a mechanism to prevent non-encrypted connections to happen between a client and a particular domain, which has declared to accept HTTPS connections only. To this end, the browser internally stores a combination of the respective domain and an expiration time. As long as this entry is valid (current time < expiration time), the browser will automatically switch to HTTPS whenever the user tries to access the domain. If this, for some reason, is not possible, the browser will refuse to establish HTTP connections to the server, presenting the user a non-overridable error message. This way, sslstrip attacks are not feasible against domains supporting HSTS.

For this mechanism to work, the browser obviously has to be aware of the server accepting HTTPS connections and committing to keep accepting HTTPS connections for a given timeframe. There are ways to convey this information: First, the server can add a `Strict-Transport-Security` HTTP

[10] See for example http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/.

[11] See https://www.bluecoat.com/sites/default/files/documents/files/How_to_Gain_Visibility_and_Control_of_Encrypted_SSL_Web_Sessions.a.pdf for a marketing brochure of one commercial TLS termination proxy provider.

[12] See RFC 6797 (https://tools.ietf.org/html/rfc6797) for details.

header to its responses; this header includes the time the server promises to support HTTPS (in seconds from the current time). However, this implies that the first connection made between the client and the server is not tampered with (Trust-on-First-Use model), since otherwise a MitM can strip out the header (as the sslstrip script shown above does).

To prevent this, the major browser vendors have agreed on a common website[13] on which website administrators can register their website for inclusion in an HSTS preload list hardcoded into the Chrome, Firefox, Safari, Internet Explorer 11, and Edge browsers. This way, the browser will employ all restrictions described above even on the first connection attempt. Using the `Strict-Transport-Security` header, websites can update the expiration field stored in the browser. For example, if the website promises to support HTTPS for at least six months, and the user visits the respective website at least every six months, continuous protection against sslstrip attacks is ensured.

HTTP Public Key Pinning

HPKP[14], on the other hand, is a measure to prevent certificate forgery attacks. Website administrators can use the `Public-Key-Pins` HTTP header to send a SHA-256 hash of their public key with every response they send to clients, as well as a timeframe in the format also used by HSTS. A browser supporting this feature will store this hash and, during the validity of the entry, only accept HTTPS connections to the respective website if the hash of the certificate presented by the websites matches the stored hash. In case of a mismatch, the connection establishment is aborted and a non-overridable error message is shown to the user. This mechanism, similar to HSTS, relies on a Trust-on-First-Use model; due to the larger amount of information that needs to be stored, no browser preload is offered to common website administrators. Some browsers, however, possess a hardcoded list of a few major websites and their HPKP entries.

If the browser possesses a valid HPKP entry for a domain visited by the user, it will reject any unknown TLS certificate, even if it is issued by a trusted CA. This way, interception by a MitM is not possible even if the attacker managed to have his root certificate installed in the user's browser. This mechanism has also been proposed to counter nation-state or similarly powerful attackers, which might be able to issue themselves valid certificates by stealing the root certificates of trusted CAs or by legally forcing them to issue rogue certificates.

Client-side precautions

As a simple user, none of the measures described above can be implemented, since they require support by the respective websites and web servers. However, every user is advised not to trust unknown networks, such as open hotspots, since the providers of these networks can easily perform any kind of MitM attacks on their users. In case of unprotected networks, even other users can perform this kind of attacks without major problems.

To prevent sslstrip attacks, users should always verify that their communication is actually transmitted in a secure fashion when they perform sensitive operations.

Similarly, they should not add security exceptions for self-signed or otherwise invalid certificates unless they are completely sure they are actually communicating with the server they intend to communicate with, for example by out-of-band transmission of the public key fingerprint. Otherwise, they risk falling victim to MitM attacks using TLS termination.

Finally, users should never add root certificates to their browser trust store unless they completely trust the owner of the certificate, since by adding the certificate, they allow that person to perform

[13] See https://hstspreload.appspot.com/.

[14] See RFC 7496 (https://tools.ietf.org/html/rfc7469) for details.

arbitrary TLS termination attacks on every HTTPS connection established in the future. When considering the number of root certificates trusted by major browsers by default, it is however questionable whether most users actually trust each and every one of these CAs in the first place…(see Figure 10).

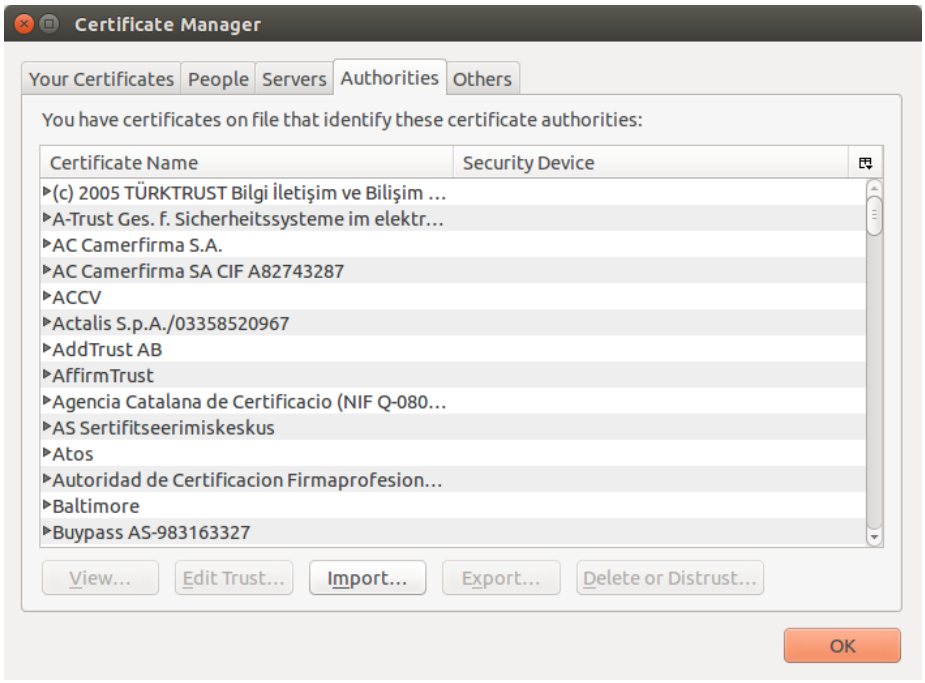

Figure 10: CAs trusted by Firefox

Attachments

`apache2.zip`

`mybank.com.crt`

`mybank.com.key`

`mybank.sql`

`mybank.zip`

`sslstrip.py`