2IC80 Laboratory on Offensive Computer Security
Lab Project Assignment

Abdel K. Bokharouss[1] and Adriaan Knapen[1]

[1] *Eindhoven University of Technology, The Netherlands*

June 10, 2018

fHTTP is an application which has been developed in relation to the lab assignment of the course Laboratory on Offensive Computer Security as taught at Eindhoven University of Technology. The incentive behind fHTTP is to develop (and release) an application capable of a fully-fledged ARP spoofing tool. The spoofing capabilities can then be used for other attacks which target both the confidentiality and integrity of the users and/or systems on the same (local) network. That is, we provide our users to filter traffic on the network and we facilitate the option to inject malicious code into traffic which is read, manipulated and passed on.

In particular, we provide our users to read insecure cookies (which can be exploited for session-hijacking), to filter (unencrypted) traffic passing by based on self-specified regular-expressions and we provide our users the ability to inject code into the packets that pass by. The users can inject into an `img`-tag and the user can change the accept-encoding HTTP header to change (or disable) the content encoding algorithm. Lastly, `Nmap`-like functionality is incorporated into the application to provide the users the ability to scan the local network for potential targets. This is in relation to the *main-objective* of this application: the application should be an easy-to-use tool usable in different scenarios and settings. We do not want to bother the users with continuous interaction with a command-line interface. The capabilities of the applications and a general idea of how to employ the application should be clear after one glance at `fHTTP`'s GitHub page.

1 Description of the Attack

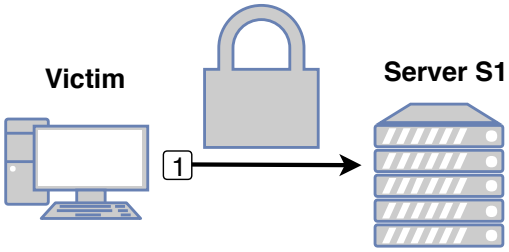
The attack we implemented is aimed at intercepting HTTP cookies on a network the attacker has access to. In addition, cookies might have been set before the attack was actually started. Albeit it is required that either the cookie does not have the *Secure* flag enabled or is send over an insecure connection during the attack.

A graphical representation of the attack can be seen in Figure 1. Below will the corresponding steps in the attack be elaborated:

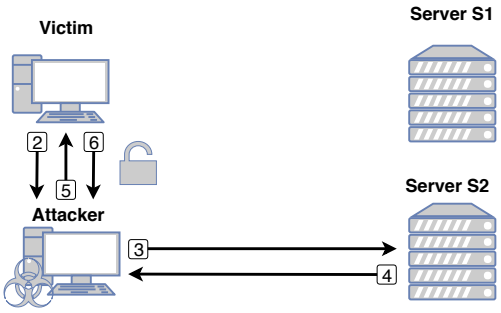
1. The victim connects to `S1` before the attack starts, and sets a cookie. Possibly using a secure connection.

The attacker acquires a man-in-the-middle (MitM) position, for example using ARP poisoning.

2. `Victim` requests a web page over HTTP to another server, since the attacker is in a MitM position will it instead be redirected to the attacker.
3. `Attacker` redirects the request to the actual recipient in order to obtain a legitimate request.
4. Again as a result of the MitM position will the response of the request be redirected back to the `Attacker`.
5. The `Attacker` injects an image into the HTTP traffic with a reference to `S1` over an insecure connection.
6. `Victim` tries to retrieve the image by sending an insecure request to `S1`, which contains the cookie set



(a) *Before the attack.*



(b) *During the attack.*

Figure 1: *Graphical representation of the communication flow during the attack.*

in (1). This request is intercepted by the `Attacker` whom now has the desired cookie.

An example use case of this attack could be that `S1` is the web server of an online banking service. The victim logged in earlies onto this service on a network which he or she actually trusts. Later on, the network changes and the victim does not trust the new environment enough and stops interacting with `S1`. Now the victim would expect that, without explicitly connecting to `S1`, his/her confidentiality with `S1` is untouched when interacting with other servers. Still does this attack manage to trick the victim into breaking the confidentiality, without raising any alert either by the victim or the banking service.

2 Technical set-up

This section is going to touch upon various technical aspects of the setup that is used to develop and test the application. In particular, the set-up which is used for the reproduction of attacks is going to be described and references to tools and other extensions that were employed to analyze the attacks are going to be given. Initially, three virtual machines were used to reproduce the attacks and to test various aspects of the application. This is depicted in Figure 2. This set-up has been used to integrate ARP-spoofing and network-scanning capabilities. The local network-scanning is an integral part of the application. Since the objective is to develop an easy-to-use tool, we do not want to bother the user to find out who or what is connected to the local network. The user should be able to simply 'plug-and-play' in whatever environment the application is

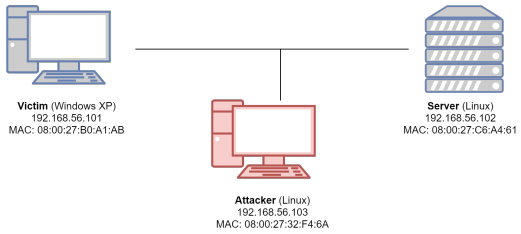


Figure 2: *Set-up which is employed for application-development & attack-reproduction and -analysis*

used. This is a feature similar to the list of capabilities of tools such as *Ettercap* (a suite for man-in-the-middle attacks) and *Nmap* (network mapper). Once ARP-spoofing and local-network scans were correctly implemented and assessed to be employable in different network settings (e.g. by changing network interfaces by exploiting `ifconfig` commands: `ifconfig iface down/up`), the set-up was extended with other functionalities for different testing purposes.

For starters, a simple web-page was hosted on the Linux server depicted on the right in Figure 2. This web-page has been developed to test various aspects of the applications such as the filtering of `HTTP` requests, Cookies et cetera. The dummy page is depicted in Figure 3. One can simply set and retrieve payloads according to the tests on would like to employ.

Figure 3: *The dummy page that was hosted on the server for testing purposes*

Once these tests were conducted and were deemed to be successful, a fourth entity was added to the set-up depicted in Figure 3. In particular, a second server (a copy of the first server) was added to the environment to assess the injection capabilities (e.g. `IMG`-tag injection). The victim sets a cookie in communications with this new (second) server. Subsequently, the cookie can be retrieved through an `IMG`-tag injection which contains a reference to this second server when traffic is send from the victim to the original server (as is explained in the previous section).

Aside from this set-up, *Wireshark* (`version: 2.2.6`) was used both in the initial development and testing phase (e.g. ARP-spoofing, local-network scan) and later phases aimed at packet modifications (e.g. injections etc.). In addition, various scripts/automated tests have been written which eased the development and testing process of the application. These tests can be found under `/test/` and are (primarily) designed for the aforementioned network setting.

As explained before, the tool has to be easy-to-use since we both want to reach a large (tech-savvy) audience and want to showcase to less technical people how vulnerable one can be in various everyday settings by demonstrating how quick one can learn the relevant concepts (considering that we have little to no prior experience) and techniques and process them into a fully-fledged application. Hopefully, this will be a warning to them what Cyber-security researchers and engineers (or 'hackers') can do when you are their 'bullseye'. The afore-stated incentive behind the development of this easy-to-use application goes, therefore, hand-in-hand with the development of a graphical user-interface (GUI). Python's de-facto GUI package *TkInter* has, therefore, been used to develop the GUI. This does, however, introduce (yet another) dependency. The applications has been developed in- and tested with `TkInter 8.5`. In addition, `Scapy` was, of course, used. The application has been extensively tested using `version: 2.3.2`. An update to `2.4.0` was considered and has been evaluated, but this affected the performance of the local network scanner.

3 Attack Analysis

3.1 Attack characteristics

The attack is highly stable and reliable, it involves no probabilistic aspects - e.g. requirements to win a race condition or the like. In case the environment fulfills all requirements, then the attack is almost guaranteed to work. This is due to the fact that almost all aspects of the attack uses tricks which are also used for legitimate reasons. Hence the environment inherently supports the attack. The only exception is the modification of the HTTP payload, the obvious counter measure is using HTTPS. But the transition from the web using HTTP to HTTPS only is still ongoing, and is far from finished.

Getting into the right environment to deploy the attack will be the most challenging obstacle into employing the tool successfully. The following requirements apply:

Position in the network The attacker has to be able to achieve a MitM position in between, this could either mean that the attacker should be on the same network as the victim, or that it takes over some gateway or router which the victim uses.

Insecure cookies The cookie which the attacker wants to intercept may not have the secure flag set, since then the browser of the victim would not include this into insecure requests. Hence making it impossible to intercept it.

HTTP traffic The attacker needs insecure HTTP traffic in which the image tag can be injected.

The impact of the attack can be severe, since cookies can be extremely sensitive information like tokens of web sessions. Which, in case the web service is public, gives the attacker to take over the session and hence impersonate the victim. In practice would every online service onto which the victim logs on be an interesting target.

3.2 Defense mechanisms

It will be quite hard for most victims to detect or mitigate this attack, since it does not generate errors on both client and server side. The attack could either be detected by noticing the vast amount of ARP packets, although this is highly impractical for most users, since it is low level and hence requires technical knowledge in order to detect the attack. The other option is to detect the changes made to the HTTP requests, in order to make this work should the victim or server notice that either the request header or response body was tampered with. Although both are still completely legitimate according to the standards do they often contain newly introduced padding. Obviously is this not definitive proof that someone is tampering with the connection, but it does make a strong indicator.

4 Attack Engineering

The easy-to-use nature of the application originates from its linear flow of instructions and capabilities. The user can, for example, not start to try to sniff packets or inject something in an `img`-tag when the ARP-cache poisoning thread has not been configured and started yet (i.e. no man-in-the-middle position). In particular, the user is lead through the following chain of exploitations, configurations and (possible) attacks: Local Network Scan $\rightarrow$ ARP Cache Poisoning $\rightarrow$ Injections and Extractions.

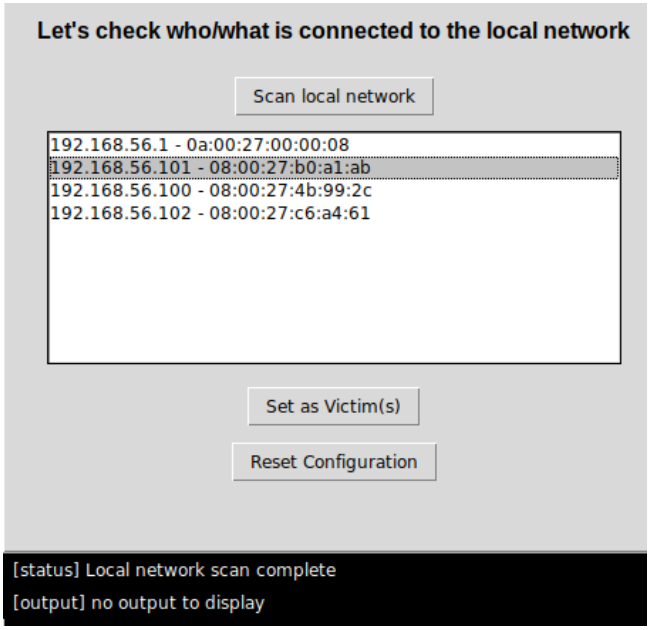


Figure 4: *The network scan as is facilitated to the end-user*

A user can perform the ARP Cache Poisoning without a local network scan (i.e. specifying the target/victim IP's himself), but execution of ARP Cache poisoning and a Local Network Scan are preconditions for the injections and extractions. The first condition is rather trivial (mitm). The latter condition is needed to have a full map of all the IP-addresses to their respective MAC-addresses. The ARP Cache poisoning is possible on its own since it only requires the user's own mac-address (automatically retrieved in the application) and the target and victim IP'. We need to know the real mac-addresses of our victims for the injections and extractions (e.g. for packet sniffing, modification and -forwarding).

A snippet of the source code (see comments in the source code) of one of the core functionalities of our network discoverer is depicted in Figure 5. The code is rather self-explanatory. The ARP-requests can also be observed in Wireshark. This should gives you an easy to digest visualization of what is being done in the iterations of the source code (see incremental IP-addresses). The next step is the actual spoofing. If the user has selected the target and victim(s) in the list, they will be automatically loaded in the next tab Figure 6. ARP spoofing is done on a separate thread which keeps poisoning (10 second interval) as long the thread is alive as can be seen in the source code.

```
def scan_local_network(self):
    ip_mac_pairs = {}  # to be stored in a ip_to_mac_mapper object
    for network, netmask, NA, iface, address in scapy.config.conf.route.routes:  # can give issues w/ scapy > 2.3.2
        if network == 0 or iface == 'lo' or address == self.local_host or address == self.non_routable:
            continue  # skip default gateway and loop-back network
        if netmask == self.def_mask or netmask <= 0:  # where def_mask = 0xFFFFFFFF
            continue
        net = self.to_CIDR_notation(network, netmask)  # convert
        if iface != scapy.config.conf.iface:  # check if it is the primary network interface
            continue  # scapy does not support arp-ing on non-primary network interfaces
        if net:
            try:
                ans, unans = scapy.layers.l2.arping(net, iface=iface, timeout=1, verbose=False)
                for s, r in ans.res:
                    try:
                        ip_mac_pairs[r.psrc] = r.hwsrc
                    except socket.herror:
                        pass  # did not resolve
```

(a) *A snippet from the source-code of the network discoverer*

```
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.87? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.88? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.89? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.90? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.91? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.92? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.93? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.94? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.95? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.96? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.97? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.98? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.99? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.100? Tell 192.168.56.103
PcsCompu_4b:99:2c   PcsCompu_32:f4:6a   ARP     60 192.168.56.100 is at 08:00:27:4b:99:2c
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.101? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.102? Tell 192.168.56.103
PcsCompu_b0:a1:ab   PcsCompu_32:f4:6a   ARP     60 192.168.56.101 is at 08:00:27:b0:a1:ab
PcsCompu_c6:a4:61   PcsCompu_32:f4:6a   ARP     60 192.168.56.102 is at 08:00:27:c6:a4:61
00:00:00_00:00:00   Broadcast           ARP     42 Gratuitous ARP for 192.168.56.103 (Request)
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.104? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.105? Tell 192.168.56.103
PcsCompu_32:f4:6a   Broadcast           ARP     42 Who has 192.168.56.106? Tell 192.168.56.103
```

(b) *The ARP requests from our local network scanner in Wireshark*

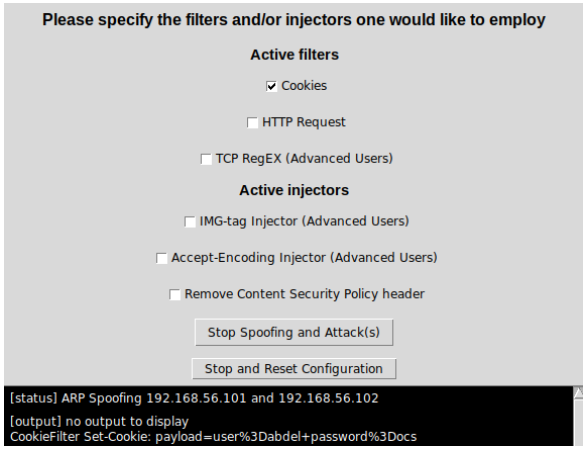
Figure 5: *Network scan in the GUI and Wireshark*

(a) *The ARP spoofing tab in the GUI*

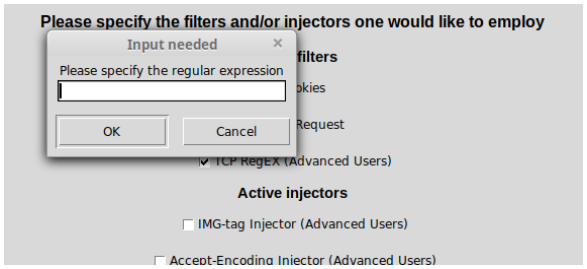
```
def _spoof_arp(self):
    packets = []

    # For each combination between two distinct victims, poison their cache to pass their traffic through us.
    for v1 in self._victims:
        for v2 in self._victims:
            if v1 is not v2:
                packets.append(Ether() / ARP(op=ARP.who_has, hwsrc=self.own_mac_address, psrc=v1, pdst=v2))

    if packets:
        sendp(packets)

def run(self):
    while self.keep_alive:
        self._spoof_arp()
        sleep(10)
```

(b) *The core functionality of the ARP-spoofing class (inherits threading.Thread)*

Figure 6: *ARP spoofing in the GUI and source code*



(a) *The tab where all the injectors and filters are facilitated. Note that this is just a snippet. The output and status frame are significantly larger in the application. In addition, one can go to a previous step in the tab-navigation based GUI*



(b) *User input has to be given for some functionalities*

Figure 7: *Extractors and Injectors*

Note that in the previous step the button that leads to the injection and extraction tab is still disabled. Once the 'Start ARP spoofing' button is clicked one is redirected to the most fun and resourceful tab. We can see this tab in Figure 7 where the cookie-filter has been turned on and a dummy attack has been displayed. For some extractors/filters and injectors, like the TCP regular-expression filter and `img`-tag injector, user input is needed which the user can specify in the input box that pops-up in the GUI.

The code for filters/extractors and injectors can be found in our repository (`fHTTP`'s GitHub page). A lot emphasis and effort is put on modularity, abstraction, adhering to overall design principles and minimization of dependencies and various other (core) software engineering principles. Introducing new filters/extractors, by other contributors since it is an open-source project, is for example made easy by the facilitation of abstract classes.

5 Discussion

`fHTTP` manages to offer a fairly easy to use tool to pull of a quite powerful attack. Still is there a lot left to improve, which mainly fall into three categories:

Required technical knowledge The graphical interface and strict flow of the application was mostly aimed at making the tool also usable for users without too much technical knowledge on networking. The current application is full of technical terms, in most cases are they irrelevant since the default values will suffice. Replacing these technical terms or explaining their consequences would be a good addition into making the tool even more easy to use.

Usability Currently the tool only offers a method into obtaining cookies, but in most cases is this not extremely useful, since only rarely the cookie values itself are interesting. Thus the attacker would need to find a way to use this newly obtained information by itself. Hence a great addition would be a way to actually use this information, one way would be to directly load them into a browser, which would allow the attacker to use the session of the victim.

Features The current setup of the tool would allow for many more features to be added, for now we present a tool which is mostly the minimum viable product for the attack we wanted to present. But many more aspects of the tool could be extended. Some suggestions would be different methods into obtaining the MitM position, more filters to also find other sensitive data like for example passwords, or other attacks like SSL stripping or redirecting victims to different web sites.

To get in conclusion, the tool offers its users everything required to successfully pull of an attack, but has even more potential for growth. The fact that the code is released open source under the MIT license allows everyone, including ourselves, to improve, to reuse, and - most importantly - to learn.