University of Amsterdam
System & Network Engineering

RP2 Project

HTTP Session Identification

Authors:
Kevin de Kok
kevin.dekok@os3.nl

Marcus Bakker
marcus.bakker@os3.nl

Coordinators:
Bart Roos
roos@fox-it.com

Sander Peters
peters@fox-it.com

Abstract

Previous research from two former students state the importance of identifying HTTP sessions[1]. The identification of HTTP sessions is not trivial, because HTTP is a sessionless protocol. In the header of the HTTP protocol[2] is no field defined that can identify a session.

During this research project the behaviour of the HTTP protocol was analysed and the results were used to define methods to identify HTTP sessions. The methods are placed into two categories. One category defines the starting point of a HTTP session. The other category can correlate HTTP messages that belong to a specific HTTP session.

The defined methods are based on web 1.0. Some methods are more successful then others.

July 5, 2010

Contents

1 Introduction

Two former OS3 students T. Kinkhorst and M. van Kleij did a research about detection of drive-by downloads[1]. This has developed the need to identify HTTP sessions.

A HTTP session can be defined as all HTTP traffic that is generated from visiting one single web-page. The content of the main web-page, but also all content retrieved from other sources (e.g. images and advertisements) can be a part of a HTTP session.

The possibility to identify HTTP sessions can be used to analyse HTTP traffic. Since HTTP is a stateless protocol it is hard to distinguish sessions from each other. The HTTP protocol does not have a field in the header that identifies a HTTP session.

A web browser however does have this possibility to distinguish HTTP sessions from each other. The browser maintains TCP sockets on which it sends en receives HTTP messages. It also knows which TCP socket belongs to which browser tab. Every browser tab contains one single HTTP session. By knowing which TCP sockets belongs to a specific browser tab, a browser is perfect capable of knowing which HTTP messages belong to which HTTP session. The knowledge on which TCP socket belongs to which browser tab is not known for this research.

This research project will look into methods to identify HTTP sessions by analysing HTTP messages from a distant view, not by using any form of host based analysis. By analysing the HTTP protocol and their behaviour there are possibilities to identify a HTTP session.

Research questions

The project has the following main research question:

- How can HTTP sessions be distinguished from each other?

The main research question can be divided in the following sub-questions:

- What is the definition of a HTTP session?
- Which characteristics of HTTP traffic can be used to develop a method to identify HTTP sessions?

Scope

- HTTP session identification will be based on the HTTP protocol (RFC: 2616 [2]);
- The methods to identify HTTP sessions will be developed for web 1.0;
- HTTP session identification will be executed from a central point in the network (no host-based detection);
- The research project will mainly focus on the development of HTTP session identification methods and not on a possible PoC.

2 Project approach

This research project was done in multiple stages. Every stage needed to be completed to continue with the next stage. The following stages were defined:

Stage 1: Theoretical research

In the first week of the research project a literature review was executed to gather information about the subject. The literature review consisted of: reading previous research and reading RFC2616 (Hypertext Transfer Protocol – HTTP/1.1)[2].

Stage 2: Defining identification methods

Based on the theoretical research of the previous stage, methods to identify HTTP sessions were defined (chapter 3). These methods were verified at stage 5 (chapter 4). The defined identification methods were not adjusted during the verification of the methods (stage 5). By this one can prove their theory in practice according to the predefined identification methods.

Stage 3: Creating the test setup

To get familiar with the HTTP protocol and the behaviour a lab set-up was created (figure 1). The set-up consisted of two servers and multiple clients. Server A (Psyduck) was the gateway for the clients and had a web server[3] running. Server B (Bulbasaur) was only running a web server. All the HTTP network traffic from the clients to the web servers was captured on server A's eth1 network interface. The actual capturing of the network traffic was performed at stage 4.

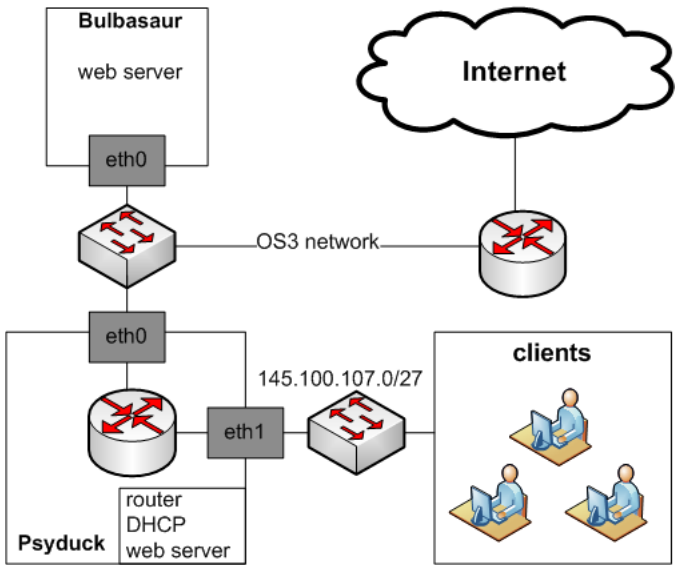

Figure 1: Network diagram of the test setup

Two different websites were used for the test set-up, namely:

- Website lab environment (labsite)
  - *Main web page*
    - ◇ image;
    - ◇ text;
    - ◇ iframe;
    - ◇ *hyperlink 1* to *local web page 1*;
    - ◇ *hyperlink 2* to *local web page 2*;
    - ◇ *hyperlink 3* to *remote website 2*.
  - *local web page 1* (hyperlink 1)
    - ◇ remote image;
    - ◇ text.
  - *local web page 2* (hyperlink 2)
    - ◇ Redirection to *remote website 1*.
  - *Remote web page 1*
    - ◇ image;
    - ◇ text.
  - *Remote web page 2*
    - ◇ image;
    - ◇ text.
- Dutch website security.nl

The lab website (labsite) was created such that it contains all elements needed to verify the identification methods. The website security.nl was added to the dataset to see until what extend the developed methods also worked on a more complex website.

Stage 4: Creating the dataset

The two websites from stage 3 were used to create a dataset. The dataset contains the traffic that is generated from visiting both websites. The dataset was created by using predefined steps:

1. Start capture;
2. Open bookmark labsite;
3. Open bookmark security.nl;
4. Open *hyperlink 1* from labsite;
5. Open *hyperlink 2* from labsite;
6. Open *hyperlink 3* from labsite;
7. Open news item security.nl;
8. Open news item security.nl;
9. Open news item security.nl;
10. End capture.

Stage 5: Method verification

The defined identification methods of stage 2 were verified by making use of the created dataset from stage 4. The dataset was analysed by means of Wireshark and some analysis made use of the PoC (appendix A). The theoretical methods were applied on this dataset to verify if the methods worked as expected.

Stage 6: Document findings

At this stage the results of the research project were documented. Documentation was done during the entire period of the research project.

3 Identifying HTTP sessions

In this chapter the characteristics that can identify a HTTP session within HTTP traffic are defined. The definition of a HTTP session is presented in section 3.1.

At section 3.2 the characteristics of HTTP traffic that can be used to identify HTTP session are presented. At section 3.3 the methods to distinguish HTTP sessions are defined. In the last section (section 3.4) the limitations of the identification methods are being discussed.

Important in every section of this chapter are the terms: HTTP request message(s), HTTP response message(s) and HTTP session. These can be defined using set theory as shown below. These abbreviations will be used in the pseudocode (section 3.3) and if necessary also in other parts of this chapter.

$A = Rq \cup Rs$	A	$=$	*All HTTP messages*
$S \subset A$	S	$=$	*HTTP session*
$Rq \in A$	Rq	$=$	*all HTTP request messages*
$Rs \in A$	Rs	$=$	*all HTTP response messages*
$Rq^n \in Rq$	Rq^n	$=$	*one sinlge HTTP request message*
$Rs^n \in Rs$	Rs^n	$=$	*one sinlge HTTP response message*

3.1 Definition of a HTTP session

A HTTP session can be defined as all HTTP request and response messages that are involved in getting the content of one single web page. The content exists of the main web page and can contain embedded objects from other web sources (e.g. images and video). A session starts when an user types in an URL into the browser and hits ”enter”. At his point TCP connection(s) will be established between the client (web browser) and server(s) (web server(s)). The content will be retrieved by means of HTTP request messages from client to server. The server will respond with HTTP response messages containing the content. A session is said to be finished, when all content is received by the client.

The user will now take his time to read the web page. After reading the web page the user may click on a hyperlink. The action of clicking on this hyperlink, will establish a new HTTP session. A HTTP session can be started by typing the URL into the browser, by clicking on a hyperlink and another possibility is the usage of a bookmark. All these actions will open a new web page.

3.2 HTTP traffic characteristics

In this section the characteristics of HTTP traffic that can be used to identify a HTTP session are presented. In the next section (section 3.3) these characteristics are used to define the methods to identify a HTTP session.

3.2.1 Referrer

The referrer[4] of a HTTP request message contains the URI from where the request originated. This can be used to follow a request to a website outside of the visited website. Embedded objects are a good an example of that. This section shows an overview of different usages of the referrer.

Embedded objects

An embedded object can be included into a website, embedded object are: Flash applets, Java applets, advertisement banners, images, stylesheets and other web related objects. If for example a Flash applet is included on a website. The HTTP request to fetch that Flash applet will contain a referrer that is pointing to the location of the website that included the Flash applet. This also holds for other embedded objects.

Iframe

An iframe[5] is a inline frame that contains another document. In the case of an iframe the referrer works as follows:
A client does a HTTP request to a web page on server A, and that web page contains an iframe which is located on server B. The client does a new HTTP request to fetch the iframe on server B. The referrer in that HTTP request contains the source URI from where the HTTP request originated (the web page that includes the iframe). Below an overview is shown of the HTTP messages that are send and received during this process:

1. REQUEST to server A;
   http_request(`'http://serverA/rp2/'`)
2. RESPONSE from server A;
   http_response(`'<body> HTML code`
   `iframe src = "http://serverB/iframe/index.html"`
   `</body>'`)
3. REQUEST to server B;
   http_request(`'http://serverB/iframe/index.html'`)
   http_request.referrer(`'http://serverA/rp2/'`)
4. RESPONSE from server B;
   http_response(`'<body> HTML code </body>'`)

3.2.2 Time between successive fetches

Besides plain HTML code, a web page can also contain embedded objects. These objects can be anything from plain HTML code (iframe) to a built-in Flash video player.

Embedded objects can be fetched in parallel by means of multiple TCP connections to different web servers that serve embedded objects. Clients (an user agent like Mozilla Firefox) can further choose to pipeline the requests for the embedded objects over one single TCP connection. The server must send the responses in the same order it received the requests from the client[2]. These two techniques will speed up the process to fetch a whole web page, including the embedded objects.

It will take however a certain amount of time before an embedded object is successfully fetched. Y. Bhole and A.Popescu[6] describe in their paper[6] that there is a time gap between successive fetches of embedded objects. They define this as the inter-arrival time or Active Off time (AOT). The Active Off Time lies between 10 and 600 ms.

3.2.3 Location header

The location header is used in HTTP response messages when the requested content is moved permanently (301) or when the requested content is moved temporarily (302). The client is forced to load the different web page present at the location header.

Below a part from the RFC2616[2] (Hypertext Transfer Protocol – HTTP/1.1) about the Location header:

```
The Location response-header field is used to redirect the recipient
to a location other than the Request-URI for completion of the
request or identification of a new resource. For 201 (Created)
responses, the Location is that of the new resource which was created
by the request. For 3xx responses, the location SHOULD indicate the
server's preferred URI for automatic redirection to the resource. The
field value consists of a single absolute URI.
   Location       = "Location" ":" absoluteURI
```

A moved permanently (301) works according to the following predefined steps:

1. REQUEST to server B;
   http_request(`'http://serverB/remote'`)

2. RESPONSE from server B;
   http_response(`'<title>301 Moved Permanently</title> <body> HTML code <p>The document has moved <a href="http://serverB/remote/">here</a>.</p> </body>'`)

3. REQUEST to server B;
   http_request(`'http://serverB/remote/'`)

4. RESPONSE from server B;
   http_response(`'<body> HTML code </body>'`)

The minor difference is in the missing `'/'` at the end of the request from step 1.

3.3 Distinguish HTTP sessions

In this section all methods that are used to identify a HTTP session are being presented. The methods are placed into three categories:

- 3.3.2 - Start of a HTTP session
- 3.3.3 - HTTP message correlation
- 3.3.4 - Not suitable

The methods in the first two categories will contain pseudocode to further define the methods. The pseudocode will include set theory as described at beginning of section 3.

Before the methods of the different categories are being presented. One method is first discussed. This method does not fit into any of the predefined categories. This method is being used to separate clients and is described in section 3.3.1.

3.3.1 Client separation

Identifying a HTTP session consists of separating the clients. This step is needed to distinguish the different clients that are present on a network. Every client will start their own HTTP sessions with a different source IP.

The method will first look at the source IP-address from where HTTP request messages originated. Different sets of HTTP request messages will be created that are having the same source IP-address. The corresponding HTTP response messages are added to the correct set by looking at the destination IP-address of the HTTP response messages. The destination IP-address of such a message has to match the source IP-address of the HTTP request message to be added to that set.

Method 1 Client separation

```
{Create separate list$X$ for every client containing all $Rq^n$ and $Rs^n$ of that client}
for all $Rq^n$ in $Rq$ do
  if $Rq^n.srcIP$ not in list$X$ then
    new list$X$ {create new list for client $X$}
  end if
  list$X$.append($Rq^n$) {Add $Rq^n$ to correct list$X$}
end for

{Add all $Rs^n$ to correct list$X$}
for all $Rs^n$ in $Rs$ do
  list$X$.append($Rs^n$) {Add $Rs^n$ to correct list$X$}
end for
```

3.3.2 Start of a HTTP session

This section will describe all methods that can be used to find the start of a HTTP session. If the HTTP messages are taken into account, than the start of a HTTP session can be defined as a pair consisting of the first HTTP request message and corresponding HTTP resonse message of a HTTP session.

The functions `storePair(HTTPMessage)` and `getPair(HTTPMessage)` are used in the pseudocode to get the corresponding HTTP request message or HTTP response message of the given `HTTPMessage`. The function `storePair(HTTPMessage)` stores the pair to define the start of a HTTP session S, and will later be used for the HTTP message correlation (section 3.3.3). The function `getPair(HTTPMessage)` will only return the pair.

3.3.2.1 Time between successive fetches

As described at section 3.1 a new HTTP session is established by an user action. Typing in a URL, clicking on a hyperlink or opening a bookmark. If one can detect these actions, the start of a new HTTP session can be identified.

After the web page is fetched from the server by the client, the user will take time to read the content on the web page. As described by Y. Bhole and A.Popescu[6] the average read time, or Passive Off time as they call it is 18 seconds. This implies, that there is a period of silence on the wire. The user is reading the web page but no web content is fetched any more.

This period of silence on HTTP traffic can be used to identity the start of a HTTP session. As discussed earlier at section 3.2.2 the time between successive fetches mostly lies between 10 and 600 ms. Which is called the Active Off time. When the period of silence is greater then a defined Active Off Time, the assumption can be made that a new HTTP session is started.

Method 2 Time between successive fetches

```
$AOT = 0.6$ {The Active Off Time is set to 600ms}
for all $Rs^n$ in $Rs$ do
  if $(Rs^n.time - Rs^{n-1}.time) > AOT$ then
    print A new HTTP session was detected
    $S$ = storePair($Rs^n$)
  end if
end for
```

3.3.2.2 No referrer

The referrer as discussed at section 3.2.1 can be used to find the starting point of a HTTP session. When the user-agent (web browser) is fetching a website, it will send along the referrer with every request when retrieving the embedded objects. But, the first request to the website when an user types in an URL in the address bar or when an user opens a bookmark does not contain any referrer. Such user action defines the start of a new HTTP session, as discussed at section 3.1.

The absence of a referrer can be related to an user action. The user action can than further be related to the start of a new HTTP session.

Method 3 No referrer

```
for all $Rq^n$ in $Rq$ do
  if $Rq^n.referrer == Null$ then
    print A new HTTP session was detected
    $S$ = storePair($Rq^n$)
  end if
end for
```

3.3.2.3 Hyperlink present at GET header

The body of a website contains hyperlinks, those hyperlinks are used to navigate to other parts of a website. When an user clicks on a hyperlink then a new HTTP session is started (see section 3.1). A characteristic of these hyperlinks is that they can only be opened by an user action. This characteristic can be used to define the start of a new HTTP session.

Two conditions have to be met before the assumption can be made that a new HTTP session is started. First, the URI in the GET header has to match the hyperlink in the body of a HTTP response message. Match means in this matter that their is a match between the hyperlink and the end of the GET header. For example:

- Hyperlink: `1/index.html`
- GET header: `/rp2/new_website/1/index.html`

This is a valid match because the hyperlink is present at the end of the GET header. This match identifies the start of a new HTTP session.

The second condition is that only HTTP response messages (containing the hyperlinks) that where received before a HTTP request message (containing the GET header) can be used to find such a match. This constraint is needed to make sure that it was the user that clicked on the hyperlink. The user can only have clicked on this hyperlink (which will appear at a GET header) if the HTML body was already received by a HTTP response message.

Method 4 Hyperlink present at GET header

```
for all $Rs^n$ in $Rs$ do
  for all $Rq^n$ in $Rq$ where $Rq^n.ID > Rs^n.ID$ do
    if $Rq^n.GET$ in $Rs^n.hyperlinks$ then
      print A new HTTP session was detected
      $S$ = storePair($Rq^n$)
    end if
  end for
end for
```

3.3.3 HTTP message correlation

This section will describe all methods that are being used to correlate HTTP messages that belong to one HTTP session. The starting point of the first two methods will be a HTTP session defined as S. S is defined by the function `storePair(HTTPMessage)` by one of the methods in the previous section (section 3.3.2).

Until now, S only contains two HTTP messages (Rq^1 and Rs^1). The methods below will add other HTTP messages that belong to a specific HTTP session S. The function `storePair(HTTPMessage)` is also used by these methods, but is now called `getPair(HTTPMessage)`. This function will not store the pair but will only return a pair.

3.3.3.1 Link the referrers

As discussed at section 3.2.1, it is possible with the referrer to see where a HTTP request message originated from. By making use of this property, it is possible to link all HTTP messages that are involved in fetching a web page. Lets have an example of this usage.

Picture 2 shows a simplified overview of fetching a web page. The HTTP response messages are not included in the picture, only the HTTP request messages are shown. In the first step the client will request the main web page of `www.example.com` and will not send along any referrer. In the next step the iframe that is emedded at `www.example.com` will be requested from `www.iframe.com`. The referrer for this request will have the value `www.example.com`. The iframe located at `www.iframe.com` contains a picture that is located at `www.img.com`. The client will send a request to `www.img.com` to retrieve this picture. The referrer of this request will contain the value `www.img.com`, that is the website which hosts the image.

By making use of the referrer it is possible to see where a HTTP request message originated from and thereby linking HTTP request messages to each other. These linked HTTP messages are belonging to the same HTTP session. In the case of the situation above, all HTTP messages can be linked, and are belonging to the same HTTP session.

The method can be explained in a more explicit way, by making use of

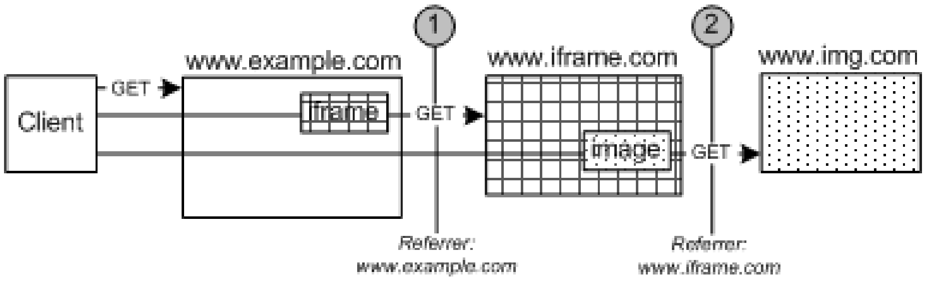

Figure 2: Diagram that shows the referrers of fetching a web page.

pseudocode (method 5).

The linking of HTTP message will be done by comparing the value of a referrer with an URI. The URI is composed of the combined value Host header + GET header of a HTTP request message that contains a HTML body. The first HTTP message of S is just such a message. The value for the referrer can come from any HTTP request message, but only if it also meets the following two conditions:

- A HTTP request message can not belong to more than one HTTP session. A HTTP request message already belonging to a session S, can thus not be placed in any other session S (Rq^n not in any S).
- A HTTP request message can only be matched if that message was received after the HTTP request message from which the URI was derived ($Rq^n.ID > ID$).

A match is valid if the above to conditions are met, and there is a match between the referrer of that HTTP request message and the earlier defined URI. The function call getPair(Rq^n) will than be used to find the corresponding HTTP response message of that HTTP request message. This pair will than be added to the correct HTTP session S.

This process will continue for every defined HTTP session S. The method can recursively link to the whole chain of HTTP messages that are linked by referrers. This is for example the case with the iframe from figure 2. This iframe is embedded at `www.example.com` but also contains an image which uses a different referrer. The pseudocode for this method can be found on the next page.

Method 5 Link the referrers

```
for all $S$ in $A$ do
  link($S$, $S.Rq^n$)
end for

function link($S$, $Rq^n$)
$URI = Rq^n.Host + Rq^n.GET$
$ID = Rq^n.ID$
for all $Rq^n$ in $Rq$ do
  if ($Rq^n.referrer == URI$) and ($Rq^n.ID > ID$) and ($Rq^n$ not in any $S$)
  then
    pair = getPair($Rq^n$)
    $S$.append(pair) {add the HTTP message pair to the HTTP session $S$}
    if pair.$Rs^n.hasHTMLBody$ then
      link($S$, $Rq^n$)
    end if
  end if
end for
end function
```

3.3.3.2 HTML body HTTP GET correlation

The HTML code of a web page contains URIs to embedded objects. These embedded objects can be located on the same domain, or located on a different domain as the main web page. Which embedded objects are being requested by a client can be determined by looking at the HTTP GET headers of the HTTP request messages. By looking at the URIs for embedded object that are present in the main web page's HTML code and the HTTP GET headers one can assume which HTTP messages do or do not belong to a HTTP session.

When the main web page's HTML code does not contain a URI that matches a HTTP GET header, then that HTTP message belongs to another session. But all HTTP GET headers that do have a URI which correlates to the HTML code belong to the same HTTP session.

It will get more difficult if a website contains an iframe. The HTML code of the iframe has also be taken into account. If the iframe contains embedded objects, then these are also considered to be part of the main web page. All HTTP GET headers that have a URI present at the HTML code of the iframe, do belong to the same HTTP session as the main web page.

The pseudocode for this method (method 6), gives a more explicit definition on the inner workings. Just as the previous method, it starts with a list of HTTP sessions S and the function link(S, Rs^n) will be used to link HTTP messages and add them to the correct HTTP session S.

Linking HTTP messages will be done by matching the URIs of embedded objects to the GET header of HTTP request messages. The URI of a embedded object has to match to the end of a GET header. The example below is a valid match:

- URI embedded object: `lokaal_plaatje.png`
- GET header: `/rp2/new_website/lokaal_plaatje.png`

The value for the GET header can come from any HTTP request message that meets the following two conditions:

- A HTTP request message can not belong to more then one HTTP session. A HTTP request message already belonging to a session S, can thus not be placed in any other session S (Rq^n not in any S).
- A HTTP request message can only be matched if that message was received after the HTTP response message from where the URIs of the embedded objects were derived ($Rq^n.ID > Rs^n.ID$).

A match is valid if the above two condition are met, and there is match between the GET header and URI of an embedded object, as described earlier. The function getPair(Rq^n) will be used to find the corresponding HTTP response message of the matched HTTP request message. This pair will be added to the correct HTTP session S.

Just like the previous method, this process will continue for every defined HTTP session S. Tthe method is also recursive, like the previous one, to be able to also link all HTTP messages of a HTTP session.

Method 6 HTML body HTTP GET correlation

```
for all $S$ in $A$ do
  link($S$, $Rs^n$)
end for

function link($S$, $Rs^n$)
$URI$ = $S.Rs^n.embeddedOcjets$ {return all embedded objects URIs of the HTML body}
for all $Rq^n$ in $Rq$ do
  if ($Rq^n.GET$ in $URI$) and ($Rq^n.ID > Rs^n.ID$) and ($Rq^n$ not in any $S$) then
    pair = getPair($Rq^n$)
    $S$.append(pair) {add the HTTP message pair to the HTTP session $S$}
    if pair.$Rs^n.hasHTMLBody$ then
      link($S$, $Rs^n$)
    end if
  end if
end for
end function
```

3.3.3.3 Use the Location header to link HTTP messages

The Location header as discussed at section 3.2.3 can be used to link HTTP messages to each other. If the URI present at the Location headers is found at a HTTP GET header of a HTTP request message then the HTTP request message containing the URI in the GET header can be linked to a previous HTTP response message containing that URI in the Location header.

Method 7 Use the Location header to link HTTP messages

```
for all $Rs^n$ in $Rs$ were $Rs^n.location$ not Null do
    location = $Rs^n.location$
    for all $Rq^n$ in $Rq$ do
        if ($location$ in $Rq^n.GET$) and ($Rs^n.ID < Rq^n.ID$) then
            print $Rq^n$ and $Rs^n$ belong to the same HTTP session
        end if
    end for
end for
```

3.3.4 Not suitable

This section will describe two possible methods that can not be used for HTTP session identification. The explanation will this time not contain any pseudocode.

3.3.4.1 Cookies

Lets have an example on the usage of an cookie to see why it can not be used for HTTP session identification.

An user will go to `www.example.com` where to web server of that website will send the user a cookie to store in its browser. Every time the user will go to this website or a particle part of this website (e.g. `www.example.com\article1.html`), it will send the cookie along with every HTTP request message. The cookie will not change while it is in the browser of the user (until it expires and the browser may receive a new one). It is static peace of information.

`www.example.com` also contains pictures that are hosted on `www.pics.com`. If an user visits `www.example.com`, the picture of this website are part of the same HTTP session. `www.pics.com` will probably also use a cookie. This cookie is however different than the one from `www.example.com`, and can not be related to each other. The HTTP request messages (containing a cookie) that will fetch the HTML body of `www.example.com` and the pictures at `www.pics.com` can thus also not be related to each other. These HTTP request messages are however part of the same HTTP session.

When looked at the situation that an user will open several articles from the same website, than these user actions will start new HTTP sessions. The HTTP request messages to fetch these articles will however all contain the same cookie. Again the cookie can not be used to separate these different HTTP request messages.

3.3.4.2 TCP port numbers

A simplified explanation of using TCP port numbers for HTTP session identification is as follows:

> An user visits the website `www.example.com`. The website contains embedded pictures hosted on `www.pics.com`. First a TCP connection to fetch the content of `www.example.com` will be established. This TCP connection will have a certain source port x. Shortly after the HTML body of `www.example.com` is fetched, the browser will set up a new TCP connection to fetch the embedded pictures from `www.pics.com`. This TCP connection will than have source port $x + 1$.
>
> Because the source port of this second TCP connection is increment with one, it can be related to that first TCP connection. If they can be related to each other, all the content that is fetched across the two TCP connection can also be related to each other. This states that all HTTP message going across those two TCP connection are belonging to the same HTTP session.

The problem of this idea is that source port numbers are not increment by one or any other number. The browser will ask the Operating System (OS) to open a TCP socket, the OS will return a TCP socket with a randomly chosen source port.

3.4 Limitations

In this section the limitations of the identification method are being discussed.

3.4.1 Javascript

When Javascript is used there might be a chance that the referrers are invalid. Invalid means that the referrer does not contain the URI from the originating site. It is very easy to change the referrer within Javascript[7]. Javascript may break or change the referrer and the referrer can not be trusted from that point any more.

3.4.2 Web 1.0

During time constrains the research is focused on web 1.0 [8] (static content) related objects and not into web 2.0 [9] (user generated content). Techniques as: Ajax, XML, Json, Flash can generate HTTP traffic even if the website is already full fetched, to update the content of the website (e.g. Google Docs). This makes it harder to identify HTTP sessions. The research is limited to web 1.0 as already mentioned at the scope of the introduction.

4 Method verification

This section describes the verification of the methods. The methods are classified in two categories, namely:

- Start of a HTTP session - section 4.1
- HTTP message correlation - section 4.2

With the methods from the category “Start of HTTP session” one can identify the start of a HTTP session. With the methods from “HTTP message correlation” one can correlate HTTP messages that belong to the same HTTP session. The dataset that was created at stage 4: “Creating the dataset” (section 2) is used for the verification.

4.1 Start of a HTTP session

The base time line (figure 3) presents the start time in seconds of all HTTP sessions from the dataset. The dataset contains a total of 8 HTTP sessions. The actions that start a HTTP session are displayed in table: 1. The following methods are represented in the category “Start of HTTP session”;

- Time between successive fetches;
- No referrer;
- Hyperlink present at GET header.

The verification of methods are also displayed on a timeline. If the time is presented in bold, it means that is in line with the base time line and counted as a true positive.

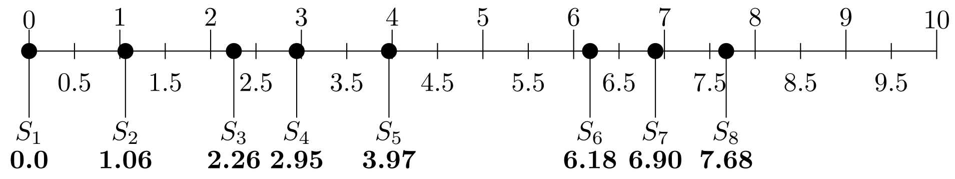

Figure 3: Start time of the HTTP session in seconds.

Request for	**from source**
S_1 - labsite	bookmark
S_2 - security.nl	bookmark
S_3 - Hyperlink web page 1	labsite
S_4 - Hyperlink web page 2	labsite
S_5 - Hyperlink web page 3	labsite
S_6 - Hyperlink article 1	security.nl
S_7 - Hyperlink article 2	security.nl
S_8 - Hyperlink article 3	security.nl

Table 1: "request to" and originating source.

4.1.1 Time between successive fetches

This method was verified by making use of the PoC (appendix A.3).

Findings

Five out of eight sessions are detected. Four out of five detected sessions correspond with the timestamps from the base time line. Session 1 is the first session in the dataset, thus always detected. Session 2 is a false positive.

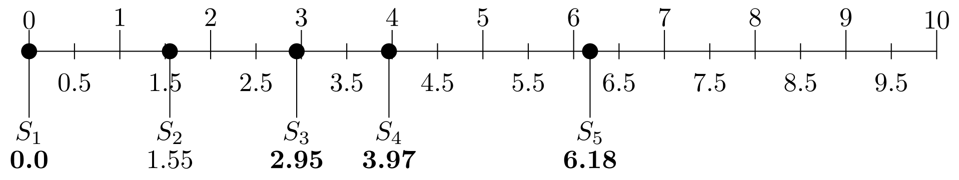

Figure 4: Start time of the HTTP session in seconds.

Request for	**from source**	**false/true positive**
S_1 - labsite	bookmark	true
S_2 - /favicon.ico	security.nl	false
S_3 - Hyperlink web page 2	labsite	true
S_4 - Hyperlink web page 3	labsite	true
S_5 - Hyperlink article 1	security.nl	true

Table 2: "request to" and originating source.

Defining the starting point of a HTTP session by using a timeframe is very hard. This approach is most useful by using "slow" browsing. When opening a lot of websites (e.g. tabs) in a short amount of time then the timeframe wont exceed the AOT of 600ms. These user actions of opening a lot amount of websites wont be detected as new HTTP sessions. These can be called true negatives. It can however be very useful to detect HTTP session on mobile phones. With a mobile phone if is very hard or impossible to open a lot amount of website in a short amount of time. This is de cause of the restrictive user interface.

By applying web 2.0 technologies a lot of false positives can be generated. Sessions remain open for longer amount of time. If the AOT is passed, it can still happen that a web 2.0 website (e.g. Google Docs) loads content from within an existing HTTP session. But due the fact that the AOT time is passed, it will be detected as a new HTTP session.

Conclusion

The method works well with ”slow” browsing. But a problem occurs at the moment an user opens a lot of websites in a short amount of time (”fast” browsing). Or when the user visits a web 2.0 website like Google docs.

”Fast” browsing can generate true negatives and web 2.0 website can generate false positives. Web 2.0 may also have other implications on this method, but these were not taken into account for this research.

Sessions:	
True positives	4
False positives	1
Real sessions	8

Table 3: Results “Time between successive fetches”.

4.1.2 No referrer

This method was verified by making use of the PoC (appendix A.4).

Findings

The first two HTTP sessions are identified due to an user action of opening a bookmark and are thereby true positives. The other three HTTP sessions (S_3 till S_5) are generated from Javascript code from Google and are detected as false positives.

The results show that method works as expected if an user opens a website from a bookmark. The result will be the same if the user opens the website not by means of a bookmark, but by typing in the URL into the address bar. In this case the same HTTP request message would have been send.

But the last three sessions are false positives, because the Javascript code from Google does not send any referrer with the HTTP request message. This problem could be solved, if HTTP request messages originating from Javascript are ignored for this method. This may be achieved by interpreting the Javascript

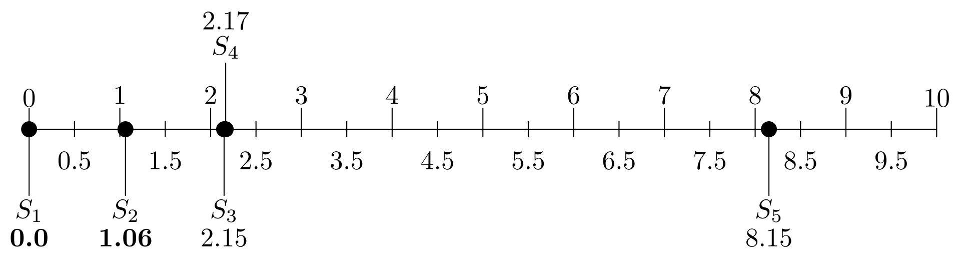

Figure 5: Sessions start time in seconds.

Request for	**from source**	**false/true positive**
S_1 - labsite	bookmark	true
S_2 - security.nl	bookmark	true
S_3 - /pagead/js/graphics.js	pagead2.googlesyndication.com	false
S_4 - /pagead/abglogo/abg-nl-100c-ffffff.png	pagead2.googlesyndication.com	false
S_5 - /pagead/sma8.js	pagead2.googlesyndication.com	false

Table 4: “request to” and originating source.

code inside the HTML body, and see what HTTP request messages it will send. Those HTTP request messages can than be ignored for this method.

Conclusion

The results are in line with the method as explained at section: 3.3.2.2. But if there is any Javascript code present at the website that clears the referrer, the method could generate false positives. This problem can be solved if all HTTP request message originated from Javascript are ignored for this method.

Sessions:	
True positives	2
False positives	3
Real sessions	8

Table 5: Results “No referrer”.

4.1.3 Hyperlink present at GET header

This method was verified by making use of a packet analysis tool and from the PoC (A.5).

Findings

All HTTP sessions, except for S_3 and S_5 are true positives. The user actions of clicking on a hyperlink from the labsite and security.nl are correctly identified.

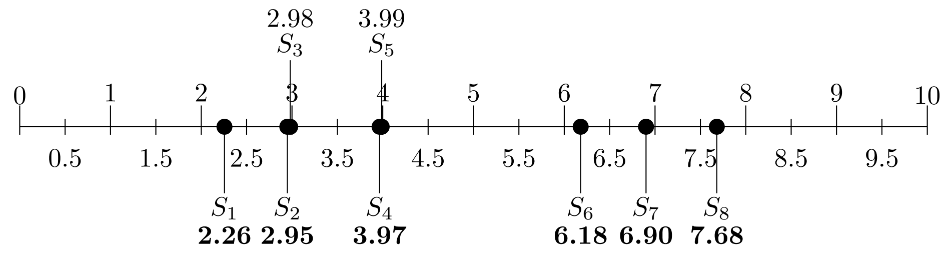

Figure 6: Start time of the HTTP session in seconds.

Request for	**from source**	**false/true positive**
S_1 - Hyperlink web page 1	labsite	true
S_2 - Hyperlink web page 2	labsite	true
S_3 - Hyperlink web page 2 (301)	labsite	false
S_4 - Hyperlink web page 3	labsite	true
S_5 - Hyperlink web page 3 (301)	labsite	false
S_6 - Hyperlink article 1	security.nl	true
S_7 - Hyperlink article 2	security.nl	true
S_8 - Hyperlink article 3	security.nl	true

Table 6: “request to” and originating source.

The results show that method works as expected if an user opens a web page from a hyperlink

The two false positives are the cause of a *301 Moved Permanently*. The body of the *301* HTTP response message contains a hyperlink that is followed by the client. The client will send a HTTP request message containing the URI of this hyperlink in the GET header. This will cause a false match between the hyperlink and the GET header.

Conclusion

The results are in line with the method as explained at section: 3.3.2.3. All user actions by means of clicking on a hyperlink are detected. A HTTP response message with *response code: 301* will generate a false positive (S_1 S_3). This problem can be solved, by ignoring all HTTP response messages with *response code: 301* for this method. And thereby generate only true positives.

Sessions:	
True positives	6
False positives	2
Real sessions	8

Table 7: Results “Hyperlink presence at GET header”.

4.1.4 Conclusion

The first method: “Time between successive fetches” is not reliable in some situations due to the fact that in can only give correct results with ”slow” browsing behaviour. When a user opens multiple websites in a short amount of time, this method will generate true negatives. But to detect the start of a HTTP session on mobile phones it may be very effective. The cause is the restrictive user interface, which results into ”slow” browsing. Web 2.0 can also cause problems if the website starts sending traffic after the AOT has elapsed. This will generate false positives.

The other two methods: “No referrer” and “Hyperlink present at GET header” work really well. The second method, has a problem that it can generate false positives when a request is generated by a HTTP response message containing a *301 Moved permanently.* This problem can easily be solved by ignoring these kind of HTTP messages. Then it will only detect true positives.

The method: “No referrer” also has a problem of generating false positives, when a HTTP request has no referrer due to the use of Javascript. This problem can be solved by ignoring any HTTP message originated from Javascript.

If the problems of the last two methods are solved, the combination of those two methods will give a perfect view of when a HTTP session is started. “No referrer” will identify all user actions of typing in a URL or opening a bookmark. The method: “Hyperlink present at GET header” will identify every user action of opening a hyperlink. These are all the action that can start a new HTTP session, as defined at section 3.1.

4.2 HTTP message correlation

This section will describe the verification of the methods from the category "HTTP message correlation":

- Link the referrers;
- HTML body HTTP GET correlation;
- Use the Location header to link HTTP messages.

4.2.1 Link the referrers

This method was verified by making use of a packet analysis tool and from the PoC (A.5).

Findings

Using the referrers to correlate HTTP messages to a HTTP session works very well. While verifying two problems did appear. One is the cause of Javascript and the other one has to do with a HTTP response message with *response code 301.*

Javascript can change the value of the referrer. In the dataset the referrer value: `http://pagead2.googlesyndication.com/pagead/ads?client=<VERY LONG STRING>` showed up. The problem is that this referrer value can not be related to any other HTTP message. The results is that the HTTP request message containing this referrer value and the corresponding HTTP response message can not me assigned to any HTTP session.

This problem may be solved by interpreting the Javascript code in the HTML body. The HTTP request message containing the referrer value, will belong to the same HTTP session as the HTML body (including the Javascript code), from which this referrer value was generated. A problem to this approach is however, that the referrer value may not be static. In this case the referrer value of the dataset that is being analysed, will be different as the referrer value this is generated at the HTTP session identification process.

The second problem has to do with a HTTP response message 301. An example:

1. `GET /remote/1`
2. `301 Moved Permanently`
3. `GET /remote/1/`
4. `200 OK`

Message 1 is defined as the start of HTTP session. It will be immediately moved to another location (message 2) to get the requested content. The client will fetch this content from the new location at message 3. The server answers with the requested content at message 4.

Normally message 3 will contain a referrer that points to the location of message one (`/remote/1`). But due to the *301 Moved Permanently* it will not point to this message. Because the referrer value does not point to the first message, it can also not be linked to that message, and thereby not assigned to that HTTP session. Section 4.2.3 will present a solution to this problem.

Conclusion

The method works really well to correlate HTTP messages and assign them to the correct HTTP session. A problem occurs when Javascript is used to change the referrer value. This problem may be solved by interpreting the Javascript code at the HTML body. Another problem occurs when an HTTP response message with *response code 301* is received. A solution is presented for this last problem at section 4.2.3.

4.2.2 HTML body HTTP GET correlation

This method was verified by making use of a packet analysis tool and from the PoC (A.5).

Findings

Using the HTML body and the GET headers of HTTP request messages to correlate HTTP message gives good results. The method is able to find almost all HTTP message that belong to a specific HTTP session.

The case where it does not work has to do with Javascript. Javascript may be used to generate the URI of an embedded object in real-time. An example from the dataset is as follows:

```
document.write(unescape("%3Cscript src='" + gaJsHost +
"google-analytics.com/ga.js' type='text/javascript'%3E %3C/script%3E"));
```

The URI of the embedded object is not available as plaintext in the HTML body. It will be generated by the Javascript code:

```
+ gaJsHost + "google-analytics.com/ga.js
```

Because the URI value is not known it can not be matched with any GET header of a HTTP request message.

The solution to this problem may be to interpret the Javascript code, and see which URI value will be used. The problem of this solution is that the value of the URI may not be static. In this case the generated URI will not show up in the dataset that is being analysed. The URI present at the dataset (at a GET header) will be older as the URI that is generated at the process of analysing the dataset.

Conclusion

The results from this method are positive by means of correlating HTTP messages and assign them to the correct HTTP session. A problem occurs when

Javascript is used to generate the URI of an embedded object in real-time. This problem may be solved by interpreting the Javascript code at the process of HTTP session identification.

4.2.3 Use the Location header to link HTTP messages

This method was verified by making use of a packet analysis tool.

Findings

This method performs really poor in correlating HTTP messages. It is only able to correlate a HTTP response message with *response code 301*, with a HTTP request message. This HTTP request message will request the content from the new location as mentioned at the HTTP response message 301.

It can however be used to solve a problem that the method: "Link the referrers" had. That problem occurs with a HTTP response message 301.

An example of the usage:

1. `GET /remote/1`
2. `301 Moved Permanently`
3. `GET /remote/1/`
4. `200 OK`

This method can correlate message 2 with message 3 as belonging to the same HTTP session. The method: "Link the referrers" could only correlate message 1 and 2 as belonging to the same HTTP session. This method will add message 3 to the same session as message 1 belongs to. The method: "Link the referrers" can thereby continue its process of correlating HTTP messages.

Because message 4 is the response of message 3, it can also be added to the same HTTP session.

Conclusion

The method is not that useful to correlate a lot of HTTP message to a specific HTTP session. It can be used to solve the problem with the HTTP response message with *response code 301* that the method: "Link the referrers" has. Thereby it is very useful if used in combination with the method: "Link the referrers".

5 Recommendations

This section describes two recommendations that can help to identify HTTP sessions, but fall out of the scope of this research project.

5.1 Host based identification

It would be interesting to research the possibilities of HTTP session identification performed at the host itself. Hosts can maintain their own state, and pinpoint HTTP sessions to a browser tab. The techniques that are used in browsers could be adapted for this purpose. A plug-in in a browser, for example in Firefox, could extend the HTTP header with a extra field that binds a HTTP message to a browser tab by using a random ID for every browser tab.

Another way to extend the HTTP header is by adding an extra header field with javascript. Javascript could be injected in the HTTP response messages to accomplish this.

5.2 Web 2.0

Web 2.0 websites make use of new web technologies to provide the content to their end users. An example of such a technology is Ajax. The methods of this research may not be that effective if these kind of techniques are being used.

It would however be interesting to research web 2.0 techniques and see if these techniques add or leave any extra information, that can be used to identify HTTP sessions.

6 Conclusion

In this chapter the research question as mentioned in the introduction is answered and the future work is being discussed.

6.1 Research question

- How can HTTP sessions be distinguished from each other?

To distinguish HTTP sessions two categories were created with different methods. The first category “Start of HTTP session” contains three methods and are being used to define the start of a HTTP session:

- Time between successive fetches - section 3.3.2.1;
- No referrer- section 3.3.2.2;
- Hyperlink present at GET header - section 3.3.2.3.

The second category “HTTP message correlation” is used to correlate HTTP messages and assign them to a HTTP session. The starting point of each of those methods will be a HTTP session as defined by one of the methods from the previous category:

- Link the referrers - section 3.3.3.1;
- HTML body HTTP GET correlation - section 3.3.3.2;
- Use the Location header to link HTTP messages - section 3.3.3.3.

The description of the methods can be found at section 3.3.2 and section 3.3.3. The methods were verified at chapter 4. The results of the method verification showed that every method has its pros and cons.

By combining the methods it is possible to identify HTTP sessions within the limits of the defined scope. The see how the methods perform in practice, outside the controlled lab environment, large scale testing is needed.

6.2 Future work

The following topics are defined for future work.

Large scale testing

As mentioned above, large scale testing is needed to see how the methods perform in practise, outside the controlled lab environment. It is recommended to perform large scale testing in an automated fashion. As this project has shown, it takes a lot of time to analyse HTTP traffic by hand.

Mobile phones

It would be interesting to see how effective the the method: “Time between successive fetches” is on mobile phones. Mobile phones use slow interaction to open a website. One cannot open multiple websites on a mobile phone below a predefined AOT, because of the way of providing input on a mobile phone. This characteristic could be very useful for the method: “Time between successive fetches”.

Web 2.0

Web 2.0 was out of the scope for this project. Web 2.0 can introduce new problems in identifying HTTP sessions. It is necessary to see to which extend the developed methods of this research also work on web 2.0.

A Proof Of Concept

This appendix describes the different parts of the PoC. Except for A.6, which describes a limitation of the PoC. The source code of the PoC is also included.

A.1 HTTParser

The HTTParser provides an API to retrieve the information from the HTTP messages inside a pcap file in a simple manner. The HTTParser was later used in other stages of the PoC to implement three identification methods.

Figure 7 shows a diagram of the HTTParser. In step 1 Thsark (a command line version of Wireshark[10]) decodes the pcap file and takes care of things like reassembling chunked HTTP messages[11]. The output is a text file that is used at step 2 by the HTTParser's Python code. HTTParser.py parses this text file and puts the necessary information for the identification methods into Python Class Instances (HTTP message objects). The HTTP message object are later being used to implement the identification methods.

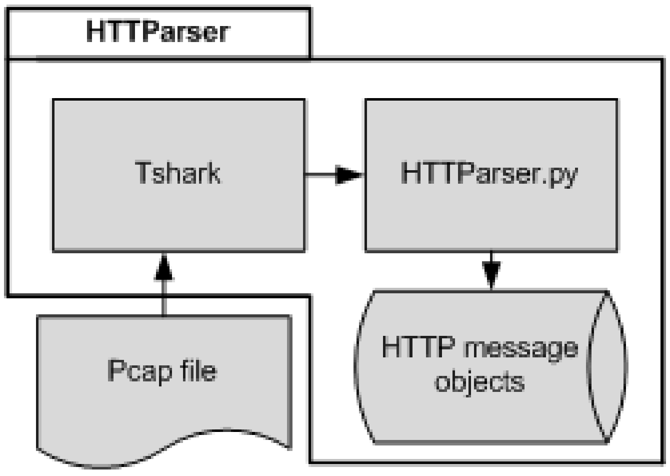

Figure 7: Diagram of the HTTParser

```
import re, hashlib, os

HOUR = 3600
MIN  = 60
id_packet = 1
id_stream = 1
time_base = []
list_packets = []   # List of all HTTPacket object instances
list_streams = []   # List of TCP streams
file_pcap = ""

# HTTPacket class that is used to save the necessary information of
    HTTP response and request messages.
class HTTPacket:
    def __init__(self):
        self.id_packet = 0
        self.id_frame = 0
        self.id_session = 0
        self.id_stream = 0
        self.time = []
        self.time_diff = 0
        self.srcIP = ""
        self.dstIP = ""
        self.srcPort = 0
        self.dstPort = 0
        self.responseCode = ""
        self.userAgent = ""
        self.host = ""
        self.date = ""
        self.location = ""
        self.get = ""
        self.referrer = ""
        self.contentType = ""
        self.contentLength = 0
        self.body = ""
        self.request = 0
        self.response = 0
        self.hyperlinks = []

    # Print all available packet information
    def printall(self):
        if self.response == 1:
            print " <<< RESPONSE <<<"
        else:
            print " >>> REQUEST >>>"
        print "Packet ID:........",self.id_packet
        print "Frame ID:.........",self.id_frame
        print "Stream ID:........",self.id_stream
        if self.id_session != 0:
            print "Session ID:.......",self.id_session
        print "Time:.............",self.time_diff
        text = self.srcIP+":"+self.srcPort
        print "SRC:..............",text
        text = self.dstIP+":"+self.dstPort
        print "DST:..............",text
        if self.host != "":
            print "Host:.............",self.host
        if self.get != "":
            print "GET:..............",self.get
        if self.userAgent != "":
            print "User-Agent:.......",self.userAgent
        if self.responseCode != "":
            print "Response Code:....",self.responseCode
        if self.date != "":
            print "Date:.............",self.date
        if self.location != "":
            print "Location:.........",self.location
        if self.referrer != "":
            print "Referrer:.........",self.referrer
        if self.contentType != "":
            print "Content-Type:.....",self.contentType
        if self.contentLength != 0:
            print "Content-Length:...",self.contentLength
        if self.body != "":
            print "Body:\n ",re.sub("\n", "\n  ", self.body)
        print "
            ----------------------------------------------------------------\
--------------------------------"

# TCPStream class is used to save the properties of different TCP
    streams in the pcap file
class TCPStream:
    def __init__(self):
        self.id_stream = 0
        self.srcIP = ""
        self.dstIP = ""
        self.srcPort = 0
        self.dstPort = 0

    def printall(self):
        print "ID:.........",  self.id_stream
        print "dstIP:......",  self.dstIP
        print "srcIP:......",  self.srcIP
        print "dstPort:....",  self.dstPort
        print "srcPort:....",  self.srcPort
        print "------------------------------------------------"

# Removes any whitespace at the end of the string
def removeWhitespace(string):
    return re.sub(r"(.*)\s", r"\1", string)

# Calculate difference in the arrival time of two HTTP messages
def timeDiff(t1, t2):
    diff =  int(t2[0])   - t1[0]
    diff += int(t2[1])   - t1[1]
    diff += float(t2[2]) - t1[2]
    return round(diff, 6)

# Print a HTTPacket instance with a specific ID
def printPacket(id_packet):
    for p in list_packets:
        if p.id_packet == id_packet:
            p.printall()

```

```
# Print all HTTPacket instances
def printAllPackets():
    for p in list_packets:
        p.printall()

# Print all packet from the given client IP-address
def printPacketsFromClient(ip):
    list = getPacketsFromClient(ip)
    for p in list:
        p.printall()

# Print all the packets from the given list
def printPacketsFromList(list):
    for l in list:
        l.printall()

# Set the pcap file to use
def setPcap(pcap):
    global file_pcap
    file_pcap = pcap

# Get a list of clients
def getAllClients():
    list = []
    for p in list_packets:
        if p.request == 1 and p.srcIP not in list:
            list.append(p.srcIP)
    return list

# Get all HTTP message from a specific client
def getPacketsFromClient(ip):
    list = []
    for p in list_packets:
        if p.request == 1 and p.srcIP == ip:
            list.append(p)
        elif p.response == 1 and p.dstIP == ip:
            list.append(p)
    return list

# Create a list of TCP streams
def getTCPStreams():
    global list_streams, id_stream
    for p in list_packets:
        if p.request == 1:
            if streamExist(p) == 0:
                stream = TCPStream()
                stream.dstIP = p.dstIP
                stream.srcIP = p.srcIP
                stream.dstPort = p.dstPort
                stream.srcPort = p.srcPort
                stream.id_stream = id_stream
                list_streams.append(stream)
                id_stream += 1

# Check if the TCP stream already exists
def streamExist(packet):
    global list_streams
    for s in list_streams:
        if packet.request == 1:
            if packet.dstIP   == s.dstIP   and packet.srcIP   == s.srcIP and \
               packet.dstPort == s.dstPort and packet.srcPort == s.srcPort:
                return 1
    return 0

# Set the TCP stream IDs of all packets in list_packets
def setStreamID():
    getTCPStreams()
    for p in list_packets:
        for s in list_streams:
            if p.request == 1 and \
            p.dstIP   == s.dstIP   and p.srcIP   == s.srcIP and\
            p.dstPort == s.dstPort and p.srcPort == s.srcPort:
                p.id_stream = s.id_stream
                break
            elif p.srcIP   == s.dstIP   and p.dstIP   == s.srcIP and\
                 p.srcPort == s.dstPort and p.dstPort == s.srcPort:
                p.id_stream = s.id_stream
                break

# Return a list containing a pair
def getPair(packet):
    list = getStream(packet.id_stream)
    pair = []
    id = packet.id_packet
    pos = 0

    #define pos in list
    for l in list:
        if l.id_packet == packet.id_packet:
            break
        pos += 1

    if packet.request == 1:
        for l in list[pos:]:
            if l.response == 1:
                pair.append(packet)
                pair.append(l)
                break
    else:
        for l in reversed(list[:pos]):
            if l.request == 1:
                pair.append(l)
                pair.append(packet)
                break
    return pair

# Return a list of packets from stream with the given id
def getStream(id):
    list_pkts = []
    for p in list_packets:
        if p.id_stream == id:
```

```python
            list_pkts.append(p)
    return list_pkts

# Walk through all lines of the Tshark text file and create class instances of the HTTPacket class
def parse():
    # Only generate the Tshark file if it does not exists
    m = hashlib.md5()
    m.update(file_pcap)
    file_tshark = m.hexdigest()[0:10] + ".tshark"
    if not os.path.exists(file_tshark):
        command = "tshark -r " + file_pcap + " -R http -V > " + file_tshark
        os.system(command)

    global id_packet
    global list_packets
    global time_base
    fd_tshark = open(file_tshark, "r")
    lines = fd_tshark.readlines()
    length = len(lines)
    x = 0

    while x < length:
        line = lines[x]
        x+=1
        #find beginning of packet
        if re.match("Frame [0-9]+ \(.*\)", line):
            http_obj = HTTPacket()
            http_obj.id_packet = id_packet
            http_obj.id_frame = re.sub(r"Frame ([0-9]*) \(.*", r"\1", line)
            http_obj.id_frame = removeWhitespace(http_obj.id_frame)
            list_packets.append(http_obj)
            id_packet += 1
        elif re.match("    Arrival Time: .*", line):
            time = re.sub(r".*([0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]+).*", r"\1", line)

            http_obj.time =  re.split(":", time)
            http_obj.time[0] = int(http_obj.time[0]) * HOUR
            http_obj.time[1] = int(http_obj.time[1]) * MIN
            http_obj.time[2] = float(http_obj.time[2])

            #If this is the first packet set the time_base variable
            if id_packet == 2:
                time_base = http_obj.time
                http_obj.time_diff = 0.000000
            else:
                http_obj.time_diff = timeDiff(time_base, http_obj.time)
        elif re.match("    Source: [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}", line):
            http_obj.srcIP = re.sub(r"    Source: ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}).*", r"\1", line)
            http_obj.srcIP = removeWhitespace(http_obj.srcIP)
        elif re.match("    Destination: [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}", line):
            http_obj.dstIP = re.sub(r"    Destination: ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}).*", r"\1", line)
            http_obj.dstIP = removeWhitespace(http_obj.dstIP)
        elif re.match("    Source port: .*", line):
            http_obj.srcPort = re.sub(r"    Source port: .*\((.*)\)", r"\1", line)
            http_obj.srcPort = removeWhitespace(http_obj.srcPort)
        elif re.match("    Destination port: .*", line):
            http_obj.dstPort = re.sub(r"    Destination port: .*\((.*)\)", r"\1", line)
            http_obj.dstPort = removeWhitespace(http_obj.dstPort)
        elif re.match(r"    HTTP/1.1 [0-9]{3} [a-zA-Z ]+\\r\\n", line):
            http_obj.responseCode = re.sub(r"    HTTP/1.1 ([0-9]{3} [a-zA-Z ]+).*", r"\1", line)
            http_obj.responseCode = removeWhitespace(http_obj.responseCode)
        elif re.match("        Request URI: .*", line):
            http_obj.get = re.sub(r"        Request URI: (.*).*", r"\1", line)
            http_obj.get = removeWhitespace(http_obj.get)
        elif re.match(r"    Date: .*\\r\\n", line):
            http_obj.date = re.sub(r"    Date: (.*)\\r\\n", r"\1", line)
            http_obj.date = removeWhitespace(http_obj.date)
        elif re.match(r"    Server: .*\\r\\n", line):
            http_obj.response = 1
        elif re.match(r"    Location: .*\\r\\n", line):
            http_obj.location = re.sub(r"    Location: (.*)\\r\\n", r"\1", line)
            http_obj.location = removeWhitespace(http_obj.location)
        elif re.match(r"    User-Agent: .*\\r\\n", line):
            http_obj.request = 1
            http_obj.userAgent = re.sub(r"    User-Agent: (.*)\\r\\n", r"\1", line)
            http_obj.userAgent = removeWhitespace(http_obj.userAgent)
        elif re.match(r"    Host: .*\\r\\n", line):
            http_obj.host = re.sub(r"    Host: (.*)\\r\\n", r"\1", line)
            http_obj.host = removeWhitespace(http_obj.host)
        elif re.match(r"    Referer: .*\\r\\n", line):
            http_obj.referrer = re.sub(r"    Referer: (.*)\\r\\n", r"\1", line)
            http_obj.referrer = removeWhitespace(http_obj.referrer)
        elif re.match(r"    Content-Type: .*\\r\\n", line):
            http_obj.contentType = re.sub(r"    Content-Type: (.*)\\r\\n", r"\1", line)
            http_obj.contentType = removeWhitespace(http_obj.contentType)
            if re.match("^image/*.", http_obj.contentType):
                http_obj.body =  "  -------------\n"
                http_obj.body += "  |             |\n"
                http_obj.body += "  |   PICTURE   |\n"
                http_obj.body += "  |-------------|\n"
                http_obj.body += "  "
        elif re.match(r"    Content-Length: .*\\r\\n", line):
            l = re.sub(r"    Content-Length: ([0-9]+).*", r"\1", line)
            l = removeWhitespace(l)
            http_obj.contentLength = int(l)
        elif re.match("^Line-based text data: text/.*", line):
            http_obj.body = ""
            # Greb all lines until a new frame and at them to the body
            if id_packet == 119:
                print "test"
            while not re.match("Frame [0-9]+ \(.*\)", lines[x]) and x+1 < length:
                http_obj.body += re.sub(r"(\\n)|(\\t)", r"", lines[x])#\[truncated\]
                x += 1;
    fd_tshark.close()
    setStreamID()
```

A.2 Client separation

The client separation as discussed at chapter 3.3.1 is actually implemented in the HTTParser from above. HTTParser contains two function for this purpose, nameley:

`getAllClients()`	This function can be used to get a list of all client IP-address present in the pcap file.
`getPacketsFromClient(ip)`	This function can be used to retrieve a list of `HTTP message objects` from a client with a specific IP-address.

A.3 Time between successive fetches

The method “Time between successive fetches” is implemented by making use of the HTTP message objects provided by HTTParser. It will print out the time in microseconds when the start of a new HTTP session was detected. Along with the time it will also print the first HTTP request and response message of that HTTP session.

```
#!/usr/bin/python
import HTTParser as ht, sys

DEFAULT_PCAP = "" # Has to be set to a valid file location
TIME_GAP = 0.6
http_sessions = []

argvlen = len(sys.argv)
if argvlen > 1 :
    file_pcap = sys.argv[1]
else:
    print "No pcap file is specified. Set to default: ",
        DEFAULT_PCAP, "\n"
    file_pcap = DEFAULT_PCAP

class HTTPSession:
    def __init__(self):
        self.id_session = 0
        self.http_pkts = []

    def addPacket(self, p):
        self.http_pkts.append(p)

    def getSize(self):
        return len(self.http_pkts)

# Initialize HTTParser
ht.setPcap(file_pcap)
ht.parse()
pkts = ht.list_packets

# Walk through all the packets and define the start of the HTTP
    sessions
time_1 = []
time_2 = []
session = []
first = 1
id_session = 1
for p in pkts:
    if p.response == 1 and first == 1: # Has only to be execute
        once to set time_1
        time_1 = p.time
        first = 0
        http_session = HTTPSession()
        http_session.http_pkts = ht.getPair(p)
        http_session.id_session = id_session
        http_sessions.append(http_session)
```

```
            id_session += 1
            continue
        elif p.response == 1:
            time_2 = p.time
            if ht.timeDiff(time_1, time_2) > TIME_GAP:
                http_session = HTTPSession()
                http_session.http_pkts = ht.getPair(p)
                http_session.id_session = id_session
                http_sessions.append(http_session)
                id_session += 1
            time_1 = time_2

print "TOTAL SESSION: ", len(http_sessions)
for s in http_sessions:
    print "# #### New HTTP session:", s.id_session, "#### #"
    print " Time:", s.http_pkts[0].time_diff
    print ""
    ht.printPacketsFromList(s.http_pkts)
    print ""
    print "# #### ................ #### #"
```

A.4 No referrer

The method "No referrer" is implemented by making use of the HTTP message objects provided by HTTPParser. It will print out the time in microseconds when the start of a new HTTP session was detected. Along with the time it will also print the first HTTP request and response message of that HTTP session.

```
#!/usr/bin/python
import HTTPParser as ht, sys

DEFAULT_PCAP = "" # Has to be set to a valid file location
http_sessions = []
id_session = 1

argvlen = len(sys.argv)
if argvlen > 1 :
    file_pcap = sys.argv[1]
else:
    print "No pcap file is specified. Set to default: ",
        DEFAULT_PCAP, "\n"
    file_pcap = DEFAULT_PCAP

class HTTPSession:
    def __init__(self):
        self.id_session = 0
        self.http_pkts = []

    def addPacket(self, p):
        self.http_pkts.append(p)

    def getSize(self):
        return len(self.http_pkts)

# Initialize HTTPParser
ht.setPcap(file_pcap)
ht.parse()
pkts = ht.list_packets

for p in pkts:
    if p.request == 1 and p.referrer == "":
        http_session = HTTPSession()
        http_session.http_pkts = ht.getPair(p)
        http_session.id_session = id_session
        http_sessions.append(http_session)
        id_session += 1

print "TOTAL SESSION: ", len(http_sessions)
for s in http_sessions:
    print "# #### New HTTP session:", s.id_session, "#### #"
    print " Time:", s.http_pkts[0].time_diff
    print ""
    ht.printPacketsFromList(s.http_pkts)
    print ""
    print "# #### ................ #### #"
```

A.5 View HTML messages

View HTML messages is a quite simple Python executable file that calls functions of HTTParser to print out all packets of the given pcap file.

An example of the output (some parts of the output are omitted with `x`):

```
 >>> REQUEST >>>
Packet ID:....... 1
Frame ID:........ 1
TCP stream ID:... 1
Time:............ 0.0
SRC:............. xxx.xxx.xxx.xxx:45467
DST:............. xxx.xxx.xxx.xxx:80
Host:............ xxxxxxx.xxx.nl
GET:............. /rp2/new_website/
User-Agent:...... Opera/9.80 (X11; Linux i686; U; en-GB)
Presto/2.2.15 Version/10.10


--------------------------------------------------------------------
 <<< RESPONSE <<<
Packet ID:....... 2
Frame ID:........ 2
TCP stream ID:... 1
Time:............ 0.000828
SRC:............. xxx.xxx.xxx.xxx:80
DST:............. xxx.xxx.xxx.xxx:45467
Response Code:... 200 OK
Date:............ Thu, 17 Jun 2010 19:20:51 GMT
Content-Type:.... text/html
Content-Length:.. 855
Body:
      <html>
      <body>
      xxx
      HTML code
      xxx
      </body>
      </html>
```

```
#!/usr/bin/python
import HTTParser as ht, sys

DEFAULT_PCAP = "" # Has to be set to a valid file location

argvlen = len(sys.argv)
if argvlen > 1 :
    file_pcap = sys.argv[1]
else:
    print "No pcap file is specified. Set to default: ",
        DEFAULT_PCAP, "\n"
    file_pcap = DEFAULT_PCAP

# Initialize HTTParser
ht.setPcap(file_pcap)
ht.parse()
# Print HTTP messages
ht.printAllPackets()
```

A.6 Truncated HTML code

The output of Tshark will truncate very long lines in the HTML body. This problem could not be solved by making use of any options of Thsark. Wireshark gives the same result.

These truncated lines of the HTML body caused problems when proper parsing of the HTML body was needed. That in its turn prevented proper parsing of the URIs of hyperlinks and embedded objects. The latter was a requirement to implement any other identification methods besides A.3 and A.4.

This problem can be solved, by not relaying on Tshark for parsing the pcap file. Due to time constrains there was no time to rewrite the PoC and solve this problem.

List of Tables

List of Figures

References

[1] T. Kinkhorst and M. van Kleij. Busting the ghost on the web: real time detection of drive-by-infections, June 2009. URL `http://www.delaat.net/~cees/sne-2008-2009/p46/report.pdf`.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. URL `http://www.rfc.net/rfc2616.html`.

[3] Apache web server. URL `http://www.apache.org`.

[4] Http referrer. URL `http://en.wikipedia.org/wiki/HTTP_referrer`.

[5] W3C. Iframe. URL `http://www.w3schools.com/tags/tag_iframe.asp`.

[6] Y. Bhole and A.Popescu. Measurement and analysis of http traffic, December 2005.

[7] Fun with the referrer property, July 2001. URL `http://www.netmechanic.com/news/vol4/javascript_no14.htm`.

[8] Wikipedia. Web 1.0, . URL `http://en.wikipedia.org/wiki/Web_1.0`.

[9] Wikipedia. Web 2.0, . URL `http://en.wikipedia.org/wiki/Web_2.0`.

[10] Wireshark, . URL `http://www.wireshark.org`.

[11] Packet reassembling, . URL `http://www.wireshark.org/docs/wsug_html_chunked/ChAdvReassemblySection.html`.