HTTP Header Analysis

*Author:*Roland Zegers
Master System and Network Engineering
University of Amsterdam
roland.zegers@os3.nl

August 31, 2015

Abstract

Many companies are busy finding new solutions for detecting the growing amount of malware that is propagated over the Internet. A lot of anti-malware developers use the unique signature of the payload as a detection mechanism. Others look at the communication channels. Research has been done to see what information HTTP headers can provide. Different aspects of headers have been investigated in order to track malware: header sizes, type errors or the presence or absence of certain headers. This information is then used to create a fingerprint or signature.

The goal of this research was to look at order of HTTP request headers to see if it is possible to determine if malware is present. Although the header order of malware is very irregular, it does not stand out when compared to HTTP headers from regular traffic. Websites have their own header order as well as programs that communicate over HTTP like Windows updates and anti-virus solutions. Some websites request special services or offer specific content. This leads to the insertion of extra headers, like security-, SOAP- or experimental headers, which create inconsistency in the order of the headers. As a result of this, it is unfeasible to use header order to reliably identify systems or malware.

Contents

1 Introduction **3**
- 1.1 Rationale . . . 3
- 1.2 Related research . . . 3

2 Research Questions **4**
- 2.1 Problem definition . . . 4
- 2.2 Research Questions . . . 4

3 Request headers **4**
- 3.1 HTTP header structure . . . 4
- 3.2 Retrieved request headers . . . 5

4 Fingerprinting **7**
- 4.1 Active fingerprinting . . . 7
- 4.2 Passive fingerprinting . . . 7
- 4.3 Fingerprinting method . . . 8
- 4.4 Request header order fingerprinting . . . 8

5 Approach **9**
- 5.1 Method . . . 9
- 5.2 Header selection . . . 10
- 5.3 Process . . . 10

6 Results **11**
- 6.1 Analysis of HTTP headers of uninfected systems . . . 11
- 6.2 Analysis of HTTP headers of the exploit kits . . . 12
- 6.3 Header probabilities of uninfected systems . . . 13
- 6.4 Header ordering of systems with an infection overlaid . . . 15
- 6.5 Calculating header uncertainty . . . 17
- 6.6 Comparison with data provided by Fox-IT . . . 18

7 Conclusions **21**
- 7.1 Influences on header ordering . . . 21
- 7.2 Determining the presence of malware . . . 21
- 7.3 Fingerprinting malware . . . 21
- 7.4 Fingerprinting PC's . . . 21

8 Future work **22**
- 8.1 Limitations . . . 22
- 8.2 Alternative parsing method . . . 22

References **23**

A Script listings **25**

B Consistency checking **34**

C Header examples **36**

D Request header listings **40**

E Graphs **43**

1 Introduction

1.1 Rationale

HTTP traffic is probably the most well-known traffic type used throughout the Internet. [27] Less well-known to the broad audience is that the HTTP communication is done using HTTP headers. These headers provide a lot of information regarding the system, the browser and applications that run on this system. The intention of these headers is to provide web pages to the user in the best possible manner, using for example the correct language, format or screen dimensions. The HTTP headers however, can also be abused for malicious purposes. A botnet can pretend to be a regular system and malware can manipulate headers to get information or to execute unwanted commands. In order to detect these malicious activities, fingerprints are made which represent a blueprint of the characteristics of the system. If the normal behavior of the system is to put the first letter in uppercase characters in the HTTP header fields, then this is a characteristic of which a fingerprint could be made to determine the validity of this system. In this report, the feasibility of using the header order to detect malicious behavior is explored.

1.2 Related research

HTTP headers is a topic which has been researched before, but often the research concentrated on HTTP response headers or the value of the header. An interesting article is *"Web application fingerprinting"*, which clearly describes different tools to analyze response headers. [8] A case where request headers are the topic of research is when looking at Request Smuggling were the incomplete parsing of request headers by intermediate systems is exploited. [5] The content of the responses is often used to fingerprint servers and applications running on those servers. An example of this is the research done by Dustin Lee, Jef Rowe, Calvin Ko and Karl Levitt. In their paper *"Detecting and Defending against Web-Server Fingerprinting"* they discuss techniques for remote identification of web servers using the response headers and also propose some defenses against server probing [24]. Author Saumil Shah gives a good introduction of HTTP fingerprinting in his paper *"An introduction to HTTP fingerprinting"*. [28] There are other researchers that also investigated request headers. A good example is the research done by Ralph Broenink from the University of Twente. [21]. He used the HTTP request headers to track the browsers used by the clients. A research performed by Juan M. Estevez-Tapiador is closely related to part of this research [26]. In his paper *"Measuring normality in HTTP traffic for anomaly-based intrusion detection"* he describes his findings on attacks carried out over HTTP traffic. He did this by trying to distinguish between normal traffic and malicious traffic. Another research that is related is a work of researcher Peter Eckersley. He did research on browser uniqueness using request headers. In his research, he set up a website to collect HTTP headers and created a database with browser fingerprints. In this research it was proven that it was possible to correctly guess more than 99 percent of the browsers using HTTP headers. [25]

2 Research Questions

This section describes the research questions that are the starting point for this research. A problem definition is presented first. From this, the research questions are derived.

2.1 Problem definition

The order of HTTP headers is dependent on several factors; the operating system, the browsing behavior, the websites and the installed software all influence which headers are sent in what order. Systems with similar operating systems and browsers behave in the same way. These similarities enable us to make a profile (=fingerprint) of these systems. As to our best knowledge, there's no existing research which addresses the influence of the HTTP header order when analyzing malware.

2.2 Research Questions

There are a lot of different categories of HTTP headers. An overview of all different types is given in section 3.1. Response headers are in general the most interesting because servers generate these responses. Servers are interesting for hackers because they often contain valuable data like financial information, trade secrets, marketing plans etc. Hackers who gain this information can use it to sell it to competitors or bribe the company by threatening to expose the data. In this research, however, the focus is on the request headers. Several malware types make use of request headers to fetch a malicious payload from a site and use that to infect the client. This research is conducted to see if it can be determined that a client is infected with malware. This is done by specifically looking at the header ordering.

The main research question is:

- *Is it possible to determine from which source certain HTTP traffic comes, when analyzing and correlating HTTP header ordering?*

  Sub-questions derived from main question are:

- Is it possible to create reliable fingerprints from the analyzed results?
- Is it possible to determine if malware is present by analyzing outliers in the HTTP header ordering?
- Can fingerprints be created that match on the outliers?

3 Request headers

This chapter provides an overview of HTTP request headers. Section one discusses the structure of HTTP traffic. Then, the second section describes the different header categories there are. The last section gives a short overview of all the headers that were retrieved from the test data during the research.

3.1 HTTP header structure

An HTTP request always starts with a start line, followed by a block of headers and then, optionally, followed by an HTTP body. For a request header, the start line consists of a method (GET, HEAD, POST etc.), a request URL and a version. The headers consist of a name, followed by a colon (:), then optional whitespace, a value and are closed with a CRLF. The header section is terminated with a blank line (an extra CRLF after the last header). This blank line must always be present, even if there are no headers or body. [22]

There are several categories of headers: [13]

- General headers; this category of headers is not related to a particular message or message component. The general headers only contain information about the message itself, not about the content.

- Request headers; these headers have several functions. They provide the server with more details about the request that the client makes. They also tell the server details about the client itself and provide the server with information about how the response should be given.
- Response headers; these headers contain information about the response that is sent. The response is sent by the server answering the request but a response can by edited by an intermediate system, like a proxy server.
- Entity headers; these headers provide information about a resource that is sent in the body of a message.
- Extension headers; headers created by individual application developers. They are not part of the HTTP specification, but must be forwarded. These headers have names that are often preceded by an "x". They are also referred to as experimental headers. This convention is no longer used [1] but these headers are still seen a lot in network traffic.

3.2 Retrieved request headers

Prior to processing the pcap files, an inventory was made of all headers that were present in the pcap files. The company Fox-IT in the Netherlands provided a lot of pcap files with HTTP traffic for the research. Next to that, three newly configured systems were used to capture traffic that would be used as a baseline for uninfected traffic. From both data-sets a different amount of unique headers were retrieved. Besides those files, some malware pcap files were collected from Malware-traffic-analysis.net [7]. From these pcaps, the headers were also gathered. The following list an overview of the headers that were found in all data-sets.

1. **Host**; this header points to the server where the request is being sent to.
2. **Referer**; this field contains the URL of the current webpage. A referer (no type error) is only present if the user clicked a link on a previous website. A manually typed URL will not provide a referer header.
3. **User-Agent**; this header contains information about the originating user-agent that sent the request.
4. **Accept**; the Accept header field is used to specify which media types are acceptable for the client.
5. **Accept-Encoding**; this header field restricts the content encodings that are in the response.
6. **Accept-Language**; this field restricts the languages that are acceptable for the client.
7. **Content-Length**; the Content-Length header contains the length of content in bytes.
8. **Content-Type**; this header field indicates the media type that should be sent.
9. **Cookie**; a Cookie contains one or more name/value pairs. It is used to send a token to the server. It is often used for stateful communication.
10. **Connection**; this header specifies options for the request header.
11. **Cache-Control**; Cache-Control passes caching directions along with the message.
12. **If-Modified-Since**; conditional request. Restricts the request unless the document has been modified since a specified date.
13. **If-Unmodified-Since**; conditional request. Restricts the request unless the document has not changed since a specified date.
14. **If-Match**; conditional request. Get the document if the entity tags supplied do match those of the current document.
15. **If-None-Match**; conditional request. Get the document if the entity tags supplied do not match those of the current document.

16. **Pragma**; this header specifies an alternative way to pass directions along with the message. It is not specific to caching.

17. **Range**; requests a specific range from a resource.

18. **If-Range**; conditional request. This header allows a conditional request for a certain range.

19. **DNT**; do not track option. It is a request to the web application to disable tracking of the user.

20. **Origin**; the Origin header defines the security contexts that were the cause to initiate a HTTP request by the User-Agent.

21. **SOAPAction**; used for appropriately filtering SOAP messages in HTTP.

22. **Upgrade-Insecure-Requests**; used by the client to let the server know it prefers an encrypted and authenticated response.

23. **UA-CPU**; allows a website to determine what cpu a client is using.

24. **UA-Java-Version**; used by a website to determine the java version used on the client.

25. **__RequestVerificationToken**; cross-site Forgery request verification token.

26. **Rest-Authorization-Code**; header used with AJAX REST pass-through authentication.

27. **Sec-WebSocket-Key,Sec-WebSocket-Protocol, Sec-WebSocket-Version**; headers used with the websocket protocol.

28. **Content-Disposition**; header sent to the origin server to suggest a default filename if the user requests that the content is saved to a file.

29. **Ajax-Request**; general purpose class AJAX http request.

30. **X-Requested-With, X-Moz, x-Akamai-Streaming-SessionID, X-APP-VERSION, X-Booking-AID, X-Booking-Exp, X-Booking-Pageview-Id, X-Booking-Session-Id, X-CorrelationId, X-CSRFToken, X-Disqus-Publisher-API-Key, X-DNT-Version, x-flash-version, X-IDCRL_ACCEPTED, X-Last-HR, X-Last-HTTP-Status-Code, X-Moz, X-NEW-APP, X-NewRelic-ID, X-Office-Version, X-Old-UID, x-prototype-version, X-Prototype-Version, x-requested-with, X-Requested-With, X-Retry-Count, X-Signature, X-Verify**; experimental headers, mainly used when requesting specific content or services or when certain programs are used.

31. **X-TeaLeaf, X-TeaLeafType, X-TeaLeafSubType, X-TeaLeaf-Page-Url, XTeaLeaf-UIEventsCapture-Version, X-TeaLeaf-Screen-Res, X-TeaLeaf-Browser-Res, X-TeaLeaf-Page-Render, X-TeaLeaf-Page-Img-Fail, X-TeaLeaf-Page-CUI-Events, X-TeaLeaf-Page-CUI-Bytes, X-TeaLeaf-Page-Dwell,X-TeaLeaf-Visit-Order**; experimental headers. Legacy headers used by the IBM Tealeaf program. [2]

The picture shows all the different header categories. On the left is the complete HTTP message, including the HTML body. The middle part shows the different header categories. The right part of the picture only shows the specific request header categories.

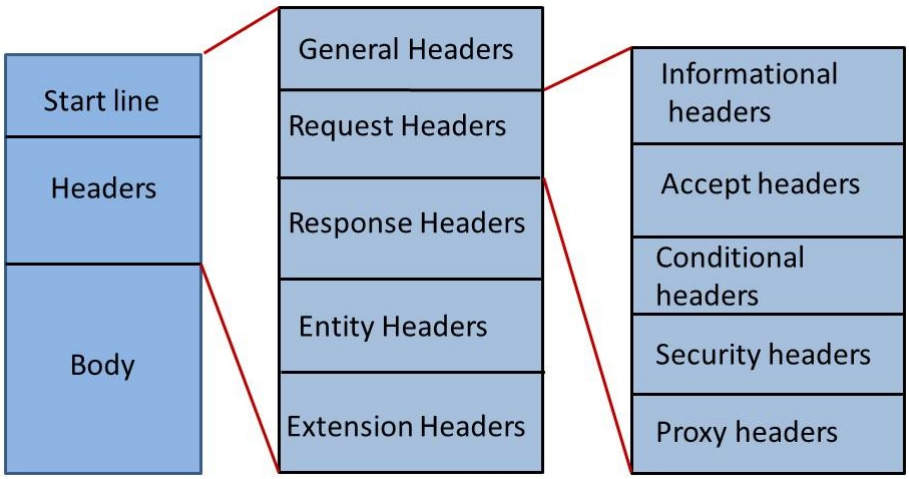


Figure 1: HTTP headers

4 Fingerprinting

A lot of tools that can do system or application fingerprinting exist. As Wikipedia states, fingerprinting means: *"fingerprinting maps arbitrarily large data to a much shorter bit string, 'the fingerprint' to uniquely identify the original data for practical purposes"* [12]. Example programs are are Nmap, Ettercap or P0F [14]. Often a confidence rating is used, as it is not always possible to relate the outcome with absolute certainty to a specific system. An example is TCP/IP fingerprinting were eight different tests are performed on TCP/IP traffic to determine the type of operating system used that sent this traffic. [14] The outcome of these tests do not always specifically point to one operating system, but to confidence levels which are probability ratings. [15]. An example could be a confidence level of 60 percent that the operating system is Windows 7 and a 40 percent confidence level that the operating system is Windows 95.

The first section of this chapter discusses active fingerprinting, followed by an explanation of passive fingerprinting in the second section. The third section elaborates more on the reliability and the last section zooms in on the header order regarding fingerprinting.

4.1 Active fingerprinting

Active fingerprinting is done by executing a series of tests against a system and then analyzing the results to determine the operating system. [16] This is an example of active fingerprinting, were the results can be influenced based on the traffic sent to the system. This form of fingerprinting is the most efficient because of the possibility of influencing the results. By sending specific packets, responses can be generated that clearly distinct one system from another. The disadvantage of this option is that it is often hard to do while remaining unnoticed. The reason for this is that the packets that are sent are often 'special' packets. They have special flags enabled or data is put in specific fields that are not commonly used. This is done to get a response that is unique for an operating system. These packets however, are quickly noticed by Intrusion Detection Systems as anomalies.

4.2 Passive fingerprinting

The opposite of active fingerprinting is passive fingerprinting, where one cannot influence the response from the system and can only analyze the traffic that comes from the system. [16] The disadvantage

is that one has to wait until the system sends traffic for some reason. Another disadvantage is that access to the traffic is necessary. An advantage is that this type of fingerprinting remains unnoticed because no traffic is generated. This option is mostly slower than active fingerprinting.

4.3 Fingerprinting method

A reliable fingerprint can only be created if the object that is fingerprinted has distinct characteristics that are unique for that object. Once there are more objects with similar characteristics, the fingerprint is less reliable (because then it can also match other objects) and thus, will generate more false positives. More than one fingerprint can be created of an object. With TCP/IP fingerprinting for example, eight different fingerprints are created [14]. The more fingerprints match, the more reliable the outcome is.

4.4 Request header order fingerprinting

Evaluating the order of the request headers can only be done using passive fingerprinting. It is hard or even impossible to force a system to send a request. The option is therefore to listen to the requests that are sent, and then analyze the header order. A distinct header order can be used as a fingerprint. If headers show up on different positions, then still a fingerprint can be made; Based on probabilities of a header on a certain position, a lower or higher confidence level can be used. Besides probability ratings, also uncertainty or entropy can be used as a form of fingerprinting [17, 23]. A lower entropy value for a system means less headers showing up on different positions. If used as a baseline or fingerprint a system can be monitored for this value. If the entropy of a system suddenly increases with a reasonable amount, then that could be an indication that there is something wrong. This is discussed in more detail in the chapter 6.

5 Approach

This chapter gives an overview of the approach that was taken to solve the research questions. The first section describes the method that was used to obtain the results. The second section discusses the way in which the headers were selected. In the third section the process used is explained.

5.1 Method

The data used for this research were pcap files with captured HTTP traffic. Using pcap files gives complete control over the type of traffic that will be analysed (as opposed to for example, live traffic). Three data-sets of pcap files with HTTP network traffic were analysed.

- A data-set with traffic coming from uninfected systems, used as a baseline.
- A data-set with traffic provided by Fox-IT. It was unknown if this data-set contained malware or not.
- A data-set, containing traffic produced by three exploit kits.

The traffic in pcaps provided by Fox-IT was gathered during a red team - blue team hacking contest [4]. The goal was to find out if the header order of these systems could give an indication of the presence of malware. To determine this, a baseline of uninfected HTTP traffic needed to be collected, because the difference in header ordering between good traffic and bad traffic had to become clear. The table 1 shows the operating systems and browser types used by Fox-IT. All systems used were Vmware images.

Name	**Platform**	**Browser**
System 1	Windows 8.1	Internet Explorer 11
System 2	Ubuntu	Firefox v.31
System 3	Windows 7	Internet Explorer 9

Table 1: Fox-IT systems

To provide a baseline of uninfected traffic, three newly installed systems were used. Several thousands of headers were captured per system. All these systems were Vmware images also. Table 2 shows the specifications of these systems.

Name	**Platform**	**Browser**
System 1	Ubuntu 14.04	Firefox v40
System 2	Windows 7	Chrome v44
System 3	Windows 8.1	Opera v31

Table 2: Baseline test systems

All systems were updated with the latest patches. The Windows systems also had an Office suite installed, Acrobat Reader and an anti-virus solution to resemble a normal system. Then three pcap files, each containing a different exploit kit were analyzed also. [7]

An exploit kit is a prepackaged web application that can have any kind of payload: ransomware, banking Trojans or whatever payload the hacker desires. All exploit kits have in common that they operate using HTTP and use this protocol to fetch the payload to infect the computer. [6] The following three virus infections were used:

1. Fiesta Exploit Kit: This is a hacker toolkit that can be used to download and install different exploits. [9]
2. Nuclear EK: This is an Exploit kit that uses a flash vulnerability. [10, 20]

3. Sweet Orange EK: Again an Exploit Kit capable of delivering different payloads, using a database backend for statistics on successful infections. [11]

All uninfected systems were overlaid with the headers of the exploit kits to see what effect the exploit kits have on the header order of the systems. From the results, several statistics were retrieved, as will be explained in chapter 6.

5.2 Header selection

The headers that are mentioned in the section 3.2 are all the headers that were collected during the research. In order to be able to make a comparison between the traffic from baseline systems and the traffic from the Fox-IT systems the headers were compared that were commonly present in the data-sets from Fox-IT and the uninfected systems and all the headers that were used by the malware. This ensured that all the relevant headers were analyzed. The researched collection of headers is defined as follows:

- A = all unique headers retrieved from the Fox-IT traffic
- B = all unique headers retrieved from the baseline (uninfected) systems
- C = all unique headers found in the pcaps containing malware
- χ = The headers used for comparison

$$Then : \chi = (A \cap B) \cup C \tag{1}$$

For retrieving the headers, a script was run that collects unique headers and stores them in a file. The script named searchhdr.sh is added to the appendix D on page 40. The headers found matching the equation 1 were used and added to the procflow.sh script which was used for processing. The script can be found in Appendix A on page 27.

5.3 Process

For obtaining the results of the research, the pcap files will be parsed to get the HTTP request headers out. Then, the request headers are made countable by removing all start lines and header values. Using line numbers, the position of the header is determined. All the position indications are summed for every header. An alternative way of doing this is described in 8.2 on page 22. Next, all the data is collected and probability and uncertainty calculations are made. The calculations are described in chapter 6. More details about the process can be found in the Appendix A.

6 Results

This chapter discusses the results regarding the analysis of the request headers. The way headers are ordered with uninfected systems is explained in the first section. In section two the analysis turns to the headers of the exploit kits. The third section shows the probability tables of the uninfected systems. In section four, comparisons are made between uninfected and infected systems using bar charts. Next, section five adds uncertainty values to the results. The last section of this chapter shows what impact the Sweet Orange exploit kit has on the header order of the traffic from the systems from Fox-IT.

6.1 Analysis of HTTP headers of uninfected systems

Vmware images with three different operating systems and browsers were used to observe the header ordering. Using different images was done to see if different browsers and/or operating systems would have any impact on the header order. The images were newly installed so it is assumed they do not contain any malware. Several thousands of headers were collected per system and analyzed. Analysis shows that the header order of every uninfected system is very dispersed over a lot of positions. Header positions in general are fairly consistent but get mixed up in a number of cases:

- In some cases were different websites are visited (some websites show a similar header order, others are completely different).
- When special programs are run that use HTTP communication (Windows update, anti-virus).
- When specific content is requested (flash, XML).
- When a website requests specific services (authentication, authorization, encryption, proxy services).

What is meant with 'mixed up' is that in most of the cases a header occurs on the same position, or roughly the same position. When headers are sent regarding the special cases mentioned above, headers can suddenly appear on a totally different position. As an example, the following output from the countnum.sh script is a representation of the User-Agent header distribution of uninfected system 2:

```
The total number of 1s are: 2
The total number of 2s are: 0
The total number of 3s are: 145
The total number of 4s are: 3445
The total number of 5s are: 136
The total number of 6s are: 32
The total number of 7s are: 0
The total number of 8s are: 1
The total number of 9s are: 0
The total number of 10s are: 0
```

Most of the headers appear on the position 4, with some scattering to position 3 and 5. Position 6 is already a doubtful position. There are outliers on position 1 and 8 (and maybe also position 6). These outliers are probably the result of the special cases discussed earlier.

Several examples of headers that are ordered in a consistent way and headers that have inconsistent ordering can be seen in the Appendix C on page 36. An explicit example can be seen on the pictures in Appendix C concerning the 'Referer' header. In the consistent header examples (15, 16, 17), the 'Referer' header is on position 5 or 6. When looking at example 20 of one of the Fox-IT systems on page 39, this header suddenly appears on position 19. The use of the IBM tealeaf program creating a lot of specific headers was responsible for this. [3]

6.2 Analysis of HTTP headers of the exploit kits

The exploit kit traffic did not show a consistent header order. Also, all kits observed communicate with only a small amount of HTTP headers. These factors create a profile that is not distinct enough to stand out over normal traffic. The pictures 2, 3, 4, 5 show four examples of headers used by the Fiesta Exploit Kit. All samples comes from the same infection. The samples show that the header order is different every time.

```
GET /ai_qkvu2/0652c44ba3f882425144540960f05520405050a580056520b03010b5255055554
HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25
Host: nrkuktxvn.myftp.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Figure 2: Fiesta Exploit header 1

```
GET /yzzzpiehxpvij8ps46znskyaqfa5ijkduakhxwcbj9 HTTP/1.1
Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/
pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, */*
Referer:
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET
CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729)
Accept-Encoding: gzip, deflate
Host: nrkuktxvn.myftp.org
Connection: Keep-Alive
```

Figure 3: Fiesta Exploit header 2

```
GET /
ai_qkvu2/4a374fcc5b4966050058040c015d52530052030f0f5201530f54070e0507525450;118800;94
HTTP/1.1
Accept: */*
Accept-Language: en-US
Referer: http://nrkuktxvn.myftp.org/yzzzpiehxpvij8ps46znskyaqfa5ijkduakhxwcbj9
x-flash-version: 11,8,800,94
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET
CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729)
Host: nrkuktxvn.myftp.org
Connection: Keep-Alive
```

Figure 4: Fiesta Exploit header 3

```
GET /ai_qkvu2/453db7e738f4f53d574d565f570c54070006035c590307070f00075d5356540050;1;2;1
HTTP/1.1
User-Agent: Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25
Host: nrkuktxvn.myftp.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Figure 5: Fiesta Exploit header 4

6.3 Header probabilities of uninfected systems

The probability of all different positions a header can appear on are calculated. From that the consistency of a header is determined. Using this information, a possible fingerprint can be created. With a certain system, how often do headers appear on the same position? These most often recurring header positions could be a unique fingerprint for that system.

The probability is calculated as:

$$P(x) = \frac{n(x)}{n} \tag{2}$$

What was observed however, is that a clean system already shows a lot of different header positions. This was irrelevant of the operating system or browser used. All three uninfected systems showed the same behavior.

The probability tables 3, 4, 5 of the three uninfected systems are shown. The probability tables show in percentages the probability of the presence of a header on one or more positions. This means for example that the header 'Content-Type' in table 3 appeared on two different positions. 20.45 percent of these headers were found on one position and 79.55 percent of these headers were found on another position. Independent of the operating system or browser used, most headers are distributed over multiple positions, albeit that in most cases the biggest number of headers reside on one position.

Header item	**1st pos.**	**2nd pos.**	**3rd pos.**	**4th pos.**	**5th pos.**
Accept	100				
Accept-Encode	100				
Accept-Language	100				
Cache-Control	100				
Connection	37.72	61.04	1.06	0.04	0.15
Content-Length	68.63	5.88	15.69	9.80	
Content-Type	20.45	79.55			
Cookie	0.61	97.87	1.46	0.06	
Host	100				
If-Modified-Since	27.71	68.67	3.01	0.60	
If-None-Match	13.42	22.15	63.76	0.67	
If-Range	0				
If-Unmodified-Since	0				
Origin	80.95	11.90	7.15		
Pragma	100				
Range	100				
Referer	99	0.97	0.03		
User-Agent	100				
X-Requested-With	94.74	5.26			
SOAPAction	0				
Upgrade-Insecure-Req	0				
X-Moz	100				

Table 3: HTTP Header probabilities system 1

The observation that the Ubuntu system (System1) has the least dispersion of headers probably has a relation with the fact that the Ubuntu system has no anti-virus installed and no or less automated program update checking mechanisms enabled or other services running that communicate over HTTP to the outside world. As explained in 6.1 these services contribute to more dispersion of headers.

Header item	**1st pos.**	**2nd pos.**	**3rd pos.**	**4th pos.**	**5th pos.**	**6th pos.**	**7th pos.**
Accept	0.85	90.30	0.42	7.92	0.50	0.03	
Accept-Encode	0.98	0.08	87.19	11.14	0.08	0.50	0.03
Accept-Language	88.16	11.23	0.08	0.51	0.03		
Cache-Control	89.19	10.81					
Connection	0.85	99.10	0.05				
Content-Length	100						
Content-Type	95	5					
Cookie	89.40	9.73					
Host	99.10	0.05	0.11	0.05	0.69		
If-Modified-Since	4.35	73.91	21.74				
If-None-Match	66.67	11.11	22.22				
If-Range	71.43	28.57					
If-Unmodified-Since	100						
Origin	51.22	46.34	2.44				
Pragma	0						
Range	59.09	20.45	15.91	4.55			
Referer	87.90	11.48	0.08	0.51	0.03		
User-Agent	0.05	3.84	91.21	3.60	1.27	0.03	
X-Requested-With	100						
SOAPAction	0						
Upgrade-Insecure-Req	96.92	2.31	0.77				
X-Moz	0						

Table 4: HTTP Header probabilities system 2

The second system, which is the Windows7 system, shows the headers more distributed over different places. Still, most headers occupy mainly one or two positions. Where the Connection header was the most scattered on system 1 and occupied five different positions, here the Accept-Encode header occupies seven different positions.

Header item	**1st pos.**	**2nd pos.**	**3rd pos.**	**4th pos.**	**5th pos.**	**6th pos.**	**7th pos.**	**8th pos.**
Accept	0.20	0.98	93.12	0.40	3.98	1.32		
Accept-Encode	0.75	0.07	0.14	87.56	9.32	0.81	1.36	
Accept-Language	0.03	88.39	9.39	0.82	1.37			
Cache-Control	2.88	92.31	4.81					
Connection	0.98	99.02						
Content-Length	86.96	6.52	6.52					
Content-Type	11.76	5.88	3.92	76.47	1.96			
Cookie	0.06	97.61	0.06	2.27				
Host	98.26	0.47	0.10	0.40	0.17	0.03	0.47	0.10
If-Modified-Since	14.89	46.81	34.04	2.13	2.13			
If-None-Match	67.44	23.26	6.98	2.33				
If-Range								
If-Unmodified-Since	100							
Origin	100							
Pragma	100							
Range	5	70	25					
Referer	88.23	9.58	0.82	1.37				
User-Agent	0.47	1.88	89.31	5.68	2.62	0.03		
X-Requested-With	40.68	57.63	1.69					
SOAPAction	100							
Upgrade-Insecure-Req	100							
X-Moz								

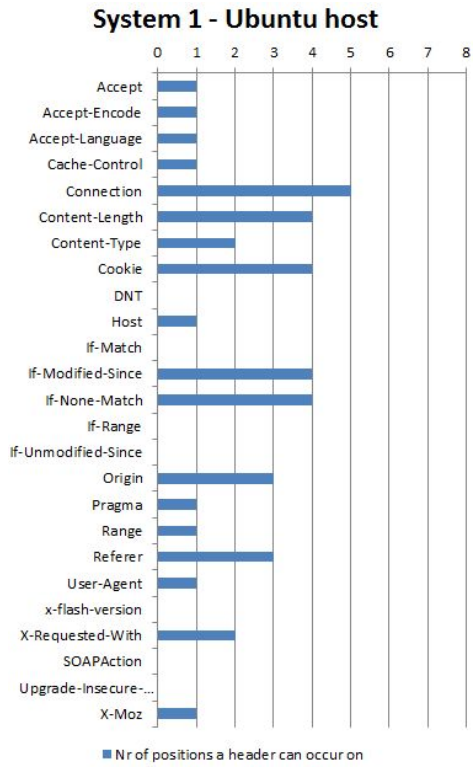
Table 5: HTTP Header probabilities system 3

The third system, which is the Windows 8.1 system, shows the most amount of dispersion. The Host header shows up on eight different positions, although 98 percent of this traffic occupied the same position. Most of the other headers also show that more than two-thirds of all headers appear on only one position. Only the 'If-Modified-Since' and the 'X-Requested-With' header show a somewhat evenly distribution over two places.

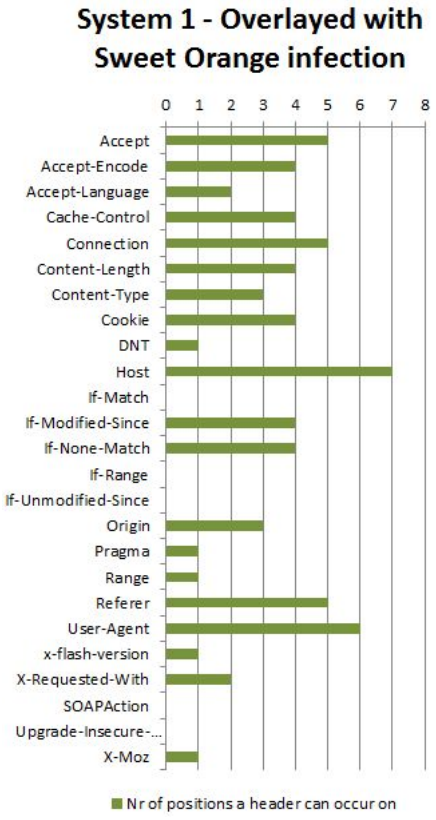
6.4 Header ordering of systems with an infection overlaid

Using the same systems, infection headers were overlaid. It was not possible to actually infect the systems in the test environment. Three exploit kit pcap files were used because these exploit kits generate their own request headers to communicate with web servers containing the malicious payload. The header order used by these exploit kits was observed and then this order was added to the existing set of headers from the uninfected systems. From that probability, occurrence and uncertainty ratings were calculated.

The graphs 6a, 6b, 7a, 7b and 8a and 8b show the number of different positions a header can have. Both graphs with and without infection headers overlaid are presented. The infection used was the Sweet Orange Exploit Kit. [11]. From the three exploit kits, the Sweet Orange kit creates the most disturbance regarding header ordering. No probability tables are shown here for two reasons: the first reason is that there was no actual infection done, it is therefore not clear what the real amount of traffic would be. The second reason is that assumed is that the probabilities will not change much: it is expected that an exploit kit will generate only a small amount of traffic; The pcap files from the exploit kits used contained a complete infection run and showed only very little HTTP header communication. [6, 19, 20].



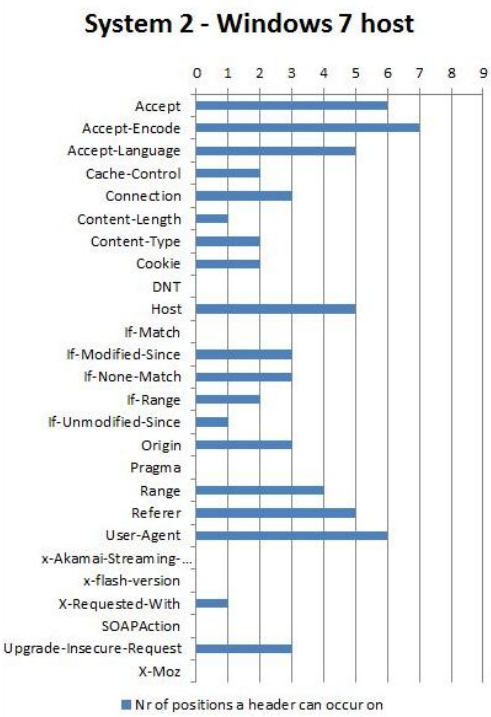
(a) System 1 - uninfected



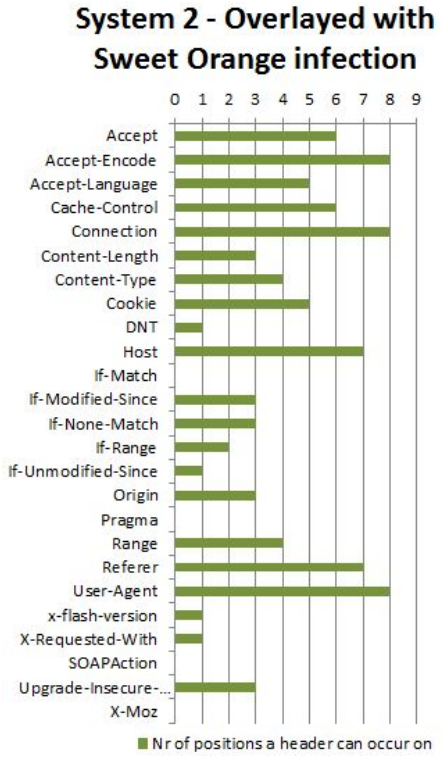
(b) System 1 - with Sweet Orange infection overlaid

Overlaying the Sweet Orange exploit kit leads to an increased number of positions being used by some headers. Notably both Host and User-Agent headers appear on six more different places. Also a previously unseen DNT header shows up. The test system is however, a clean system on which

only a few thousand headers were used as a baseline. In a real world situation, the differences will probably be much less.

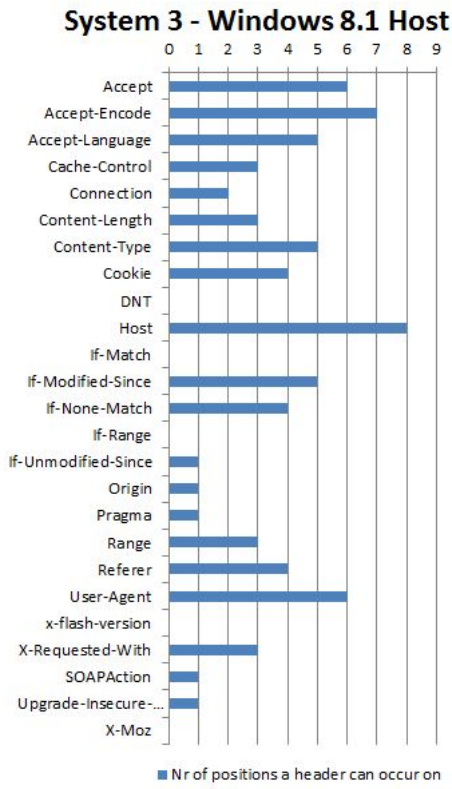


(a) System 2 - uninfected

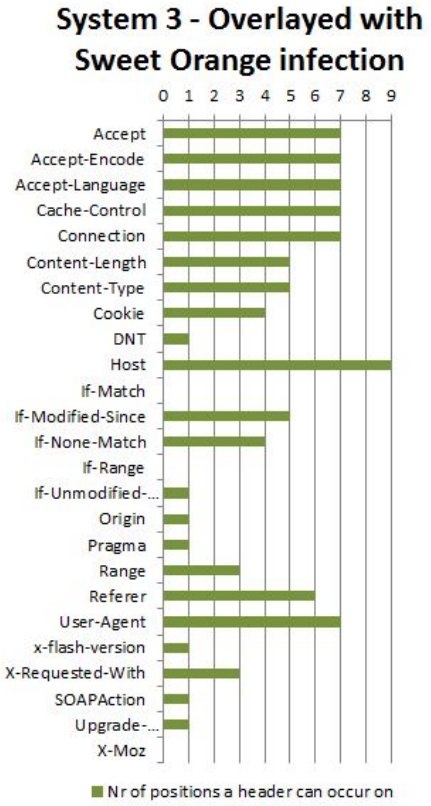


(b) System 2 - with Sweet Orange infection overlaid

With the Windows 7 host the differences between the uninfected and infected system are less visible. Most notable are the Connection header that appears on five more different positions after infection. A Cache-Control header that shows up on four more different places. With the infection two new headers are introduced which are the DNT and the x-flash-version header.



(a) System 3 - uninfected



(b) System 3 - with Sweet Orange infection overlaid

With the Windows 8.1 host, the differences are also limited to a few headers. Most notable again here is the increase in positions of the Connection header and the appearance of the DNT and x-flash-version header.

Only the comparison between the uninfected systems and the Sweet Orange infection is shown here. Comparisons with the other infections can be found in appendix E.

6.5 Calculating header uncertainty

The previous calculated probabilities using the formula 2 are used as input for calculating the header uncertainty. Doing this gives a profile or fingerprint of the system, expressed in entropy bits. For this, Shannon's formula on entropy calculation is used: [17, 23]

$$H(X) = -\sum_{i=1}^{n} p_i log_2(p_i) \tag{3}$$

The table shows the entropy change before and after three different infections:

Systems	Entropy before infection	Fiesta Exploit infection	Sweet Orange infection	Nuclear EK infection
System1	5.15	5.85	7.50	6.36
System2	6.98	7.09	7.56	7.45
System3	7.38	7.54	8.20	7.62

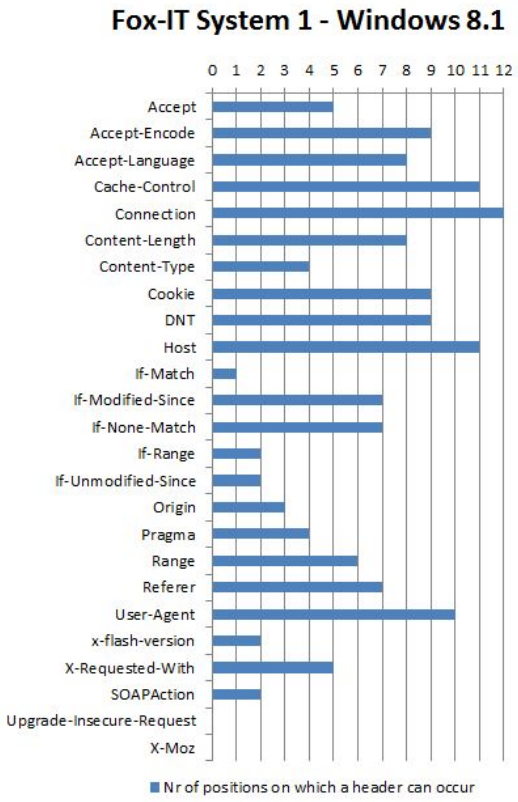
Table 6: Entropy calculations before and after infections

Although the systems already have a high entropy value, after overlaying an infection on them, the entropy rises even more. The Sweet Orange exploit kit, which has the biggest dispersion of headers, shows as expected, the biggest increase in entropy.

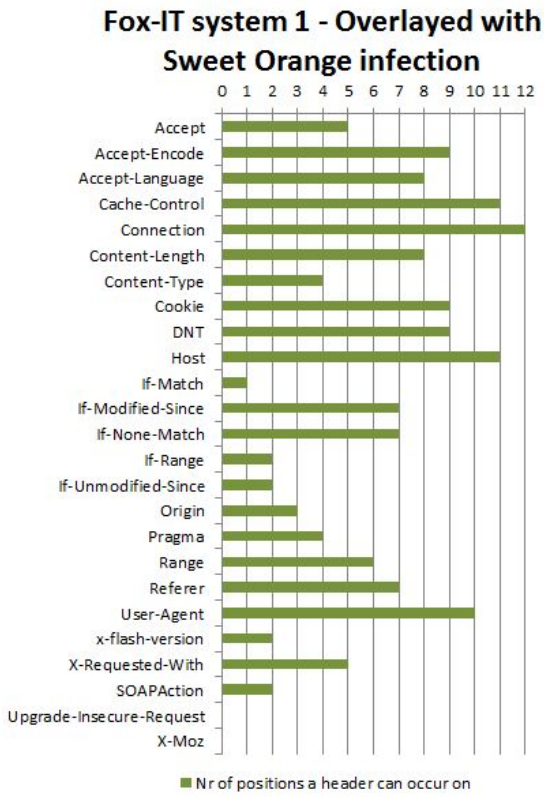
6.6 Comparison with data provided by Fox-IT

The data provided by Fox-IT was meant to be compared to the results of the uninfected systems in order to determine if malware was present by looking at the header order. [18]. After analyzing the data of Fox-IT much more headers were found when compared to the traffic of the uninfected systems. In total there were seventy-five different headers discovered in the test data-set provided by Fox-IT and twenty three headers in the uninfected systems. Lists of all headers of the uninfected systems and Fox-IT systems are added in Appendix D. In order to be able to compare these systems the formula 1 on page 10 was used.

The figures 9a, 10a and 11a are charts from the three Fox-IT systems with their header order occurrences. Next to these charts, the charts 9b, 10b and 11b with the Sweet Orange exploit kit headers overlaid, are shown.

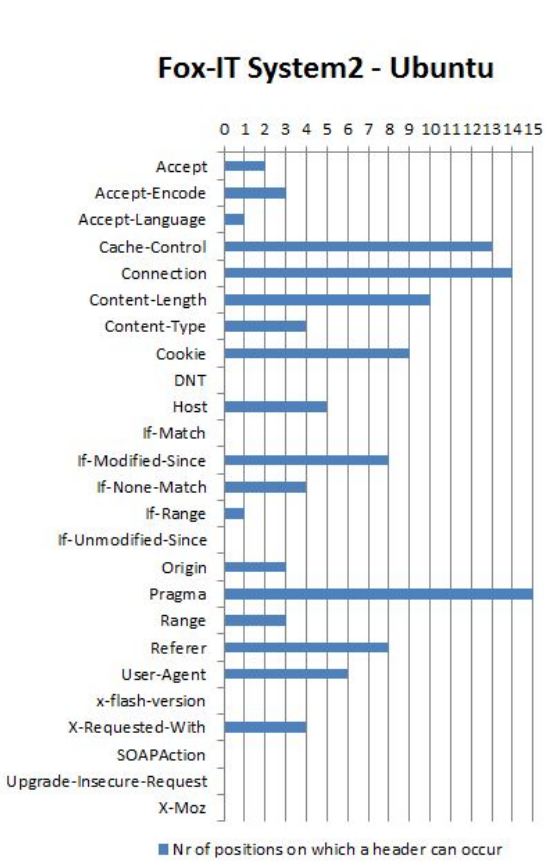


(a) Fox-IT System 1 - Uninfected

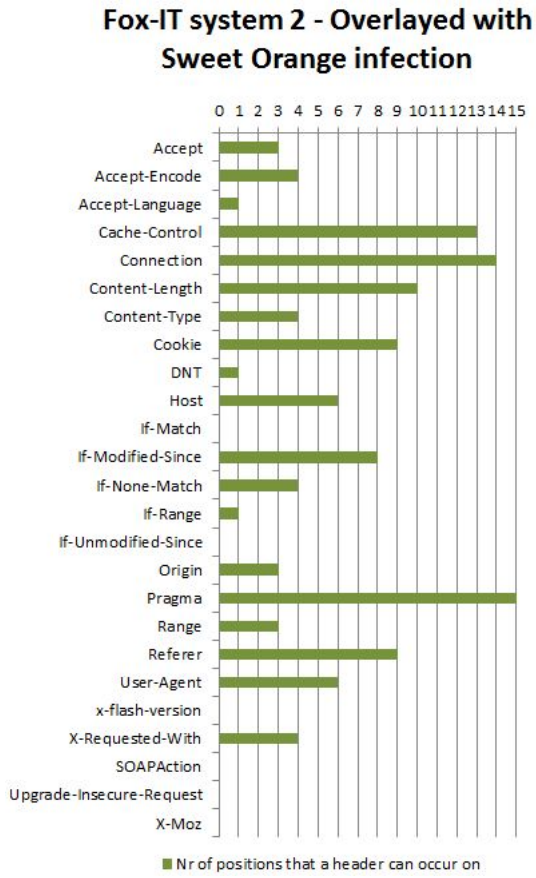


(b) Fox-IT System 1 - with Sweet Orange infection overlaid

The header order of system 1 is exactly the same after the infection is overlaid as it was before. The infection has no impact on the header order. The headers are already so much dispersed that the headers of the Sweet Orange exploit kit do not change anything.

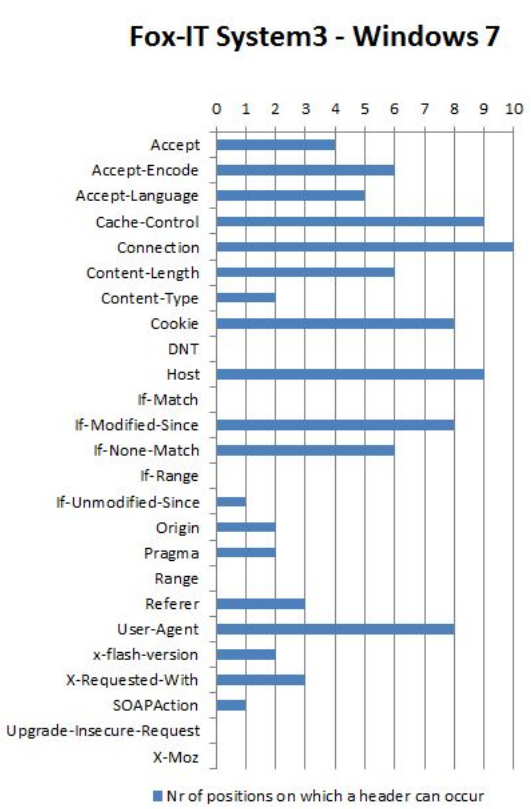


(a) Fox-IT System 2 - Uninfected

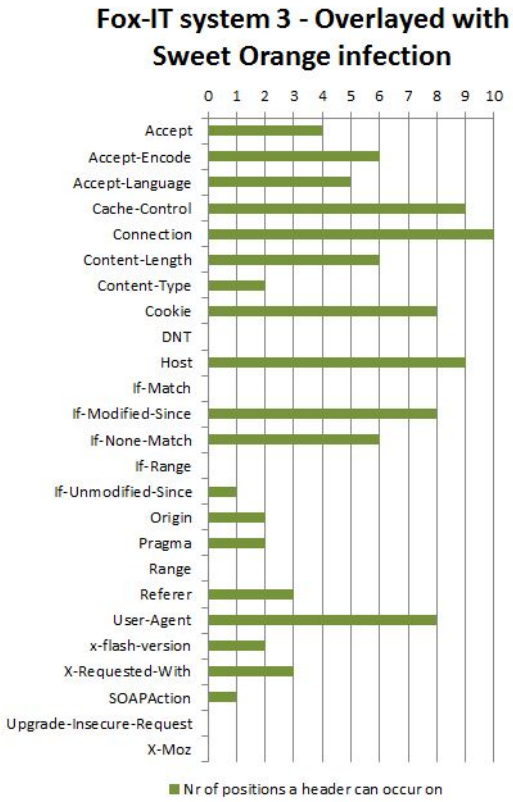


(b) Fox-IT System 2 - with Sweet Orange infection overlaid

Although hardly visible, minimal change is done to the ordering of system 2 after the infection. Besides the increase in the Accept and Accept-Encode headers also the appearance of a DNT and x-flash-version header is shown after the infection. The reason that some headers appear on fourteen or fifteen different positions is because of the use of the IBM Tealeaf program on this system, which creates a lot of custom (experimental) headers.



(a) Fox-IT System 3 - Uninfected



(b) Fox-IT System 3 - with Sweet Orange infection overlaid

With system 3, again no changes are made to the order of the headers after the infection.

Overlaying the headers of the exploit kit that has the most dispersed header order has minimal effect on number of header order positions. When looking at the entropy of the systems, for system 1 and 3 the entropy will remain the same before and after the infection. Only for system 2 the entropy values change. This is shown in table 7.

Systems	**Entropy before infection**	**After Sweet Orange infection**
System1	10.25	10.25
System2	7.50	8.15
System3	7.81	7.81

Table 7: Entropy calculations before and after Sweet Orange infection

7 Conclusions

After analyzing all the traffic, several conclusions could be drawn from the results. Section one discusses all components that have an influence on the header order. The second section concludes on the usability of the header order for determining the presence of malware. This is followed by section three which discusses fingerprinting malware and finally section four that concludes on fingerprinting PC's.

7.1 Influences on header ordering

Observations during this research showed that the header order is fairly consistent when a single website is visited. Slight changes occur when specific content is requested. For example, an 'x-flash-version' header is put in between when flash content is requested. Also when a program is run like Windows Update, the headers this program uses are all on the same position. This is the big difference with the exploit kits that were observed. They send for example the first request with the 'Accept' header on position 1. In the next request the header is on position 5 and the next request it is on position 3. Things change when a system is used to browse to different websites, request different kinds of content and use different programs that communicate over HTTP (which is normally done on every system). Then, clearly the number of positions a header can appear on, increases. Therefore, the more websites are visited, the more programs are installed that use HTTP communication, the more scattered the header order will become on that system.

7.2 Determining the presence of malware

The results show that malware is not very consistent with header ordering as was explained in the previous section. The baseline systems that were used in this research were newly installed virtual images with only a few programs running on these them. The amount of headers collected on these systems equals a few hours or days of browsing, depending on the user. Analyzing the traffic, an increase of entropy could be observed when exploit kit traffic was overlaid. When the same was done on the intensively used systems provided by Fox-IT, no changes were visible in most cases. For the traffic of the Fox-IT systems it can therefore not be concluded that they do, or do not contain malware.

Although our baseline systems showed an increase in entropy, this will not be distinct enough to create a reliable fingerprint for it. Furthermore it will likely be the case that real world systems have a much more distributed header order. Using the header order as an indication for the presence of malware will therefore not be a viable solution.

7.3 Fingerprinting malware

From the exploit kits that were analyzed, the unstructured header order is a characteristic that could be used as a trigger to use for fingerprinting malware in general. But, as mentioned in the section 7.2, the distributed header order profile does not stand out enough against the already dispersed header order of normal systems that are used on a regular base. Creating reliable fingerprints using the header order will most likely be impossible.

7.4 Fingerprinting PC's

The header order of computer systems in some cases creates a distinct profile that can be used as a fingerprint. On one of the systems, a lot of experimental headers were retrieved, due to the use of a specific program. This gave this system a distinct header order profile compared to the other systems. But the characteristics will not stand out between two systems that use the same program. Using a header order fingerprint to identify a unique system will therefore also be unpractical to apply.

8 Future work

In section one, the limitations that applied to this research are discussed. The next section shows a possible improved method of parsing HTTP header traffic.

8.1 Limitations

During this research, there were some limitations. First, it was not possible to work with actual infected systems in the experimentation environment. Also due to time restrictions it was not possible to apply a probably better parsing method, which is discussed in the next section.

8.2 Alternative parsing method

The parsing of headers in this research was done in a very straightforward way. A better way of doing it would probably be by parsing headers based on the HTTP header convention. The advantage of this is that it works irrelevant of the type and size of the header values. With this method, it is not necessary to analyze the data beforehand to find all the headers and take into account strange header values, as was done in this research. The alternative parsing method should look something like the following: Jump to the first CRLF (=omit the start line), store everything following this CRLF up to the first colon (the first header). Create a counter, set it to one and store this value with the first header. Jump to the next CRLF, store again everything up until the next colon and increase the counter by one and store the value with the header. Repeat this until a CRLF is followed by another CRLF (the blank line that separates the headers from the body). Then stop and repeat the process again. This should parse all traffic using the formal HTTP convention in a correct way.

References

[1] Deprecating the "x-" prefix and similar constructs in application protocols. http://tools.ietf.org/html/rfc6648. [Online; accessed 23 -August-2015].

[2] Ibm tealeaf. http://www-01.ibm.com/support/knowledgecenter/SS2MBL_9.0.2/UICj2Guide/UIC/UICj2InstandImpl/SupportForLegacyHeaders_86.dita. [Online; accessed 23 -August-2015].

[3] Ibm tealeaf. http://www-01.ibm.com/software/info/tealeaf/. [Online; accessed 28 -August-2015].

[4] Red team. https://en.wikipedia.org/wiki/Red_team. [Online; accessed 25 -August-2015].

[5] Http request smuggling. https://www.owasp.org/index.php/HTTP_Request_Smuggling, 2009. [Online; accessed 5-June-2015].

[6] Forbidden fruit: The sweet orange exploit kit. https://www.bluecoat.com/security-blog/2012-12-17/forbidden-fruit-sweet-orange-exploit-kit, 2012. [Online; accessed 21-June-2015].

[7] Malware-traffic-analysis.net. http://www.malware-traffic-analysis.net/, 2012. [Online; accessed 21-June-2015].

[8] Web application fingerprinting. https://pentestlab.wordpress.com/tag/http-response-header/, 2012. [Online; accessed 04 -July-2015].

[9] Fiesta exploit kit. http://www.malware-traffic-analysis.net/2013/11/29/index.html), 2013. [Online; accessed 21-June-2015].

[10] Nuclear ek from 95.211.128.101 - babyserr.ru. http://malware-traffic-analysis.net/2014/04/27/index.html), 2013. [Online; accessed 21-June-2015].

[11] Sweet orange ek from 95.163.121.188 - google.chagwichita.com:16122. http://malware-traffic-analysis.net/2014/08/18/index.html, 2013. [Online; accessed 21-June-2015].

[12] Fingerprint(computing). https://en.wikipedia.org/wiki/Fingerprint_(computing), 2014. [Online; accessed 24-June-2015].

[13] Http message headers. http://www.tcpipguide.com/free/t_HTTPMessageHeaders.htm, 2014. [Online; accessed 21-June-2015].

[14] Tcp/ip stack fingerprinting. https://en.wikipedia.org/wiki/TCP/IP_stack_fingerprinting, 2014. [Online; accessed 21-June-2015].

[15] Enhanced operating system identification with nessus. http://www.tenable.com/blog/enhanced-operating-system-identification-with-nessus, 2015. [Online; accessed 30-June-2015].

[16] Enhanced operating system identification with nessus. http://resources.infosecinstitute.com/passive-fingerprinting-os/, 2015. [Online; accessed 30 -June-2015].

[17] Entropy (information theory). https://en.wikipedia.org/wiki/Entropy_%28information_theory%29, 2015. [Online; accessed 6-June-2015].

[18] Fox-it. https://www.fox-it.com/nl/, 2015. [Online; accessed 30 -June-2015].

[19] An in-depth analysis of the fiesta exploit kit. http://blog.0x3a.com/post/110052845124/an-in-depth-analysis-of-the-fiesta-exploit-kit-an, 2015. [Online; accessed 29-June-2015].

[20] Nuclear ek leverages recently patched flash vulnerability. https://blog.malwarebytes.org/exploits-2/2015/03/nuclear-ek-leverages-recently-patched-flash-vulnerability/, 2015. [Online; accessed 21-June-2015].

[21] Ralph Broenink. User browser properties for fingerprinting. http://letmetrackyou.org/paper.pdf, 2011. [Presented on the 16th biannual Twente Student Conference on IT].

[22] Marjorie Sayer Anshu Aggarwal-Sailu Reddy David Gourley, Brian Totty. Http, the definitive guide. http://shop.oreilly.com/product/9781565925090.do, 2002. [Online; accessed 8-June-2015].

[23] Dolors. Claude shannon's theory on entropy. https://www.youtube.com/watch?v=JnJq3Py0dyM.

[24] Calvin Ko Karl Levitt Dustin Lee, Jeff Rowe. Detecting and defending against web-server fingerprinting. https://acsac.org/2002/papers/96.pdf. [Published on the 18th annual Computer Security Applications Conference, 2002].

[25] Peter Eckersley. How unique is your webbrowser? https://panopticlick.eff.org/browser-uniqueness.pdf, 2014. [Online; accessed 24-June-2015].

[26] J Diaz-Verdejo J. Estevez-Tapiador, P. Garcia-Teodoro. Measuring normality in http traffic for anomaly based intrusion detection. http://www.sciencedirect.com/science/article/pii/S1389128604000064, 2003. [Online; accessed 5-June-2015].

[27] J. Gettys R. Fielding. Rfc 2616 - hypertext transfer protocol http1/1. https://www.ietf.org/rfc/rfc2616.txt, 1999. [Online; accessed 5-June-2015].

[28] Saumil Shah. An introduction to fingerprinting. http://www.net-square.com/httprint_paper.html.

A Script listings

parseflows.sh script to parse pcap files

This script iterates through all the pcap files and uses the tcpflow program to split the files in ASCII files containing request headers per IP address. Within this script, the Procflow.sh script is started that processes all the flow files.

```
#!/bin/bash
# pcap files are parsed using tcpflow, only HTTP request header
    flows.
# flows are then divided per IP address to sub-folders
# Then, the procflow.sh script is executed per directory to create
    text files containing header item positions
# The folders containing the flow files are cleaned for the next
    iteration
# The end number in the FOR loop resembles the number of pcaps that
    will be processed
# Registering begin time to display at the end of the process
DATUM=`date`
        # Looping through all the pcap files
        for NUM in {1..56}
        do
        # Split the pcap file into separate request flows
        tcpflow -e http -r ${NUM}.pcap  dst port 80
        # Put the flows in separate folders per system
        # Virus files IPs
        #mv /home/roland/Flows/192.168.204.157* /home/roland/Flows/
            Host1
        #mv /home/roland/Flows/192.168.204.229* /home/roland/Flows/
            Host2
        #mv /home/roland/Flows/172.016.165.133* /home/roland/Flows/
            Host3
        # Foxit IT files IPs
        mv /home/roland/Flows/192.168.010.002* /home/roland/Flows/
            Host1
        mv /home/roland/Flows/192.168.010.015* /home/roland/Flows/
            Host2
        mv /home/roland/Flows/192.168.010.057* /home/roland/Flows/
            Host3
        mv /home/roland/Flows/192.168.010.075* /home/roland/Flows/
            Host4
        mv /home/roland/Flows/192.168.010.076* /home/roland/Flows/
            Host5
        mv /home/roland/Flows/192.168.010.083* /home/roland/Flows/
            Host6
        # Good systems IPs
        #mv /home/roland/Flows/010.000.002.015* /home/roland/Flows/
            Host1
        #mv /home/roland/Flows/192.168.000.103* /home/roland/Flows/
            Host2
        #mv /home/roland/Flows/192.168.001.211* /home/roland/Flows/
            Host3
        # Check to make sure no clients are omitted
        mv /home/roland/Flows/*00080  /home/roland/Flows/Check
                #Create folder listings as input for the procflow
                    script. The end number resembles the number of
```

```
                hosts
        for NR in {1..6}
        do
                # Export variable so it can be picked up by
                    procflow.sh
                export NR
                # check if files are present in folder. If
                    not, skip.
                if test -n "$(find /home/roland/Flows/Host${
                    NR}/ -maxdepth 1 -name '*00080' -print -
                    quit)"
                then
                ls /home/roland/Flows/Host${NR}/*00080 > /
                    home/roland/files${NR}.txt
                # remove path from filenames
                cat /home/roland/files${NR}.txt | sed 's
                    /.*\///'
                # Start procflow.sh for counting headers
                /home/roland/Flows/Host${NR}/procflow.sh /
                    home/roland/files${NR}.txt
                # clean Host folder and files file for next
                    iteration
                rm /home/roland/Flows/Host${NR}/*00080*
                rm /home/roland/Flows/Host${NR}/*.txt
                rm /home/roland/files${NR}.txt
                #reset global defined variable
                export NR=
                else
                echo "no files present in /home/roland/Flows/
                    Host${NR}"
                fi
        done
    echo ">>>>>>>>>>>>>>  FILE $NUM is processed <<<<<<<<<<<<<"
    done
# showing begin- and end time of the script
echo "Script started: "
echo $DATUM
echo "Script ended: "
date
echo "end of program"
```

procflow.sh script to split flows and count headers

The procflow.sh script is started from within parseflows.sh. Every time parseflows.sh splits a pcap file into separate flows, procflow.sh processes those flows. What procflow.sh does is split the flow files further so every single request is in a separate file. Within every file the script searches for the request labels that are defined in the script. If a match is found, the line number of the request is copied to an output file with a similar name as the request header. This is continually repeated for all files. All hits are appended in the output files.

```
#!/bin/bash
# This program substitutes the ref, cookie and uri headers with empty
    content, because the content sometimes occupies more than one
   line
# Script picks op global variable NR, defined in parseflows.sh.
# Script uses files.txt as parameter, defined in  parseflows.sh.
FILENAME=$1
while read -r line
        do
        NAME=$line
        # reset file counter
        COUNT=0
        # Split files so every flow header is in a separate file
        csplit -n 1 -f ${NAME}xx -zk ${NAME} '/^\s*$/' {*}
        # delete all split files smaller than 2 bytes
        find -name "*80xx*" -size -2c -delete
        # Incremental counter to process all xx files
                while [ -f ${NAME}xx${COUNT} ]
                do
                # remove blank lines
                sed -i '/^\s*$/d' ${NAME}xx${COUNT}
                # Sanitize flow files, remove request content;remove
                    unconventional headers
                cat ${NAME}xx${COUNT} |tail -n +2 |sed -e 's/:.*$/:/g
                    ' -e '/GET/,$d' -e '/POST/,$d' -e '/HEAD/,$d' > ${
                   NAME}xx${COUNT}-out
                # count the headers
                cat ${NAME}xx${COUNT}-out | grep -an "Referer:"|cut -
                    f1 -d: >> /home/roland/Flows/Host${NR}/referer.txt
                     2> /dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "Host:"|cut -f1
                   -d: >> /home/roland/Flows/Host${NR}/host.txt 2> /
                   dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "[uU]ser-[aA]
                    gent:"|cut -f1 -d: >> /home/roland/Flows/Host${NR
                    }/useragent.txt 2> /dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "Accept:"|cut -
                    f1 -d: >> /home/roland/Flows/Host${NR}/accept.txt
                    2> /dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "Accept-Encoding
                    :"|cut -f1 -d: >> /home/roland/Flows/Host${NR}/
                    acceptencode.txt 2> /dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "Accept-Language
                    :"|cut -f1 -d: >> /home/roland/Flows/Host${NR}/
                    acceptlang.txt 2> /dev/null
                cat ${NAME}xx${COUNT}-out | grep -an "Cookie:"|cut -
                    f1 -d: >> /home/roland/Flows/Host${NR}/cookie.txt
```

```
            2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Connection:" |
            cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            connection.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Content-Type:" |
            cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            contentype.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Content-Length
            :" | cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            contentlength.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Cache-Control
            :" | cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            cachecontrol.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Pragma:" | cut -
            f1 -d: >> /home/roland/Flows/Host${NR}/pragma.txt
            2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Range:" | cut -f1
             -d: >> /home/roland/Flows/Host${NR}/range.txt 2>
            /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "If-Unmodified-
            Since:" | cut -f1 -d: >> /home/roland/Flows/Host${NR
            }/ifunmodisince.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "DNT:" | cut -f1 -
            d: >> /home/roland/Flows/Host${NR}/dnt.txt 2> /dev
            /null
        cat ${NAME}xx${COUNT}-out | grep -an "If-Match:" | cut
            -f1 -d: >> /home/roland/Flows/Host${NR}/ifmatch.
            txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "If-None-Match
            :" | cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            ifnonematch.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "If-Modified-
            Since:" | cut -f1 -d: >> /home/roland/Flows/Host${NR
            }/ifmodisince.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Origin:" | cut -
            f1 -d: >> /home/roland/Flows/Host${NR}/origin.txt
            2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "If-Range:" | cut
            -f1 -d: >> /home/roland/Flows/Host${NR}/ifrange.
            txt 2> /dev/null
             cat ${NAME}xx${COUNT}-out | grep -an "x-flash
                -version:" | cut -f1 -d: >> /home/roland/
                Flows/Host${NR}/x-flver.txt 2> /dev/null
cat ${NAME}xx${COUNT}-out | grep -an "X-[rR]equested-[wW]ith
    :" | cut -f1 -d: >> /home/roland/Flows/Host${NR}/x-requested
    -with.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "SOAPAction:" |
            cut -f1 -d: >> /home/roland/Flows/Host${NR}/
            soapaction.txt 2> /dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "Upgrade-
            Insecure-Requests:" | cut -f1 -d: >> /home/roland/
            Flows/Host${NR}/upgrade-insecure-requests.txt 2> /
            dev/null
        cat ${NAME}xx${COUNT}-out | grep -an "X-Moz:" | cut -f1
             -d: >> /home/roland/Flows/Host${NR}/x-moz.txt 2>
            /dev/null
```

```
                # Increase counter
                COUNT=$((COUNT+1))
                done
        done < "$FILENAME"
replace newlines with commas, for better readability and easier
    counting
cat /home/roland/Flows/Host${NR}/referer.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/referer.txt
cat /home/roland/Flows/Host${NR}/host.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/host.txt
cat /home/roland/Flows/Host${NR}/useragent.txt |tr "\\n" "," >> /home
    /roland/Flows/Host${NR}/Txtfiles/useragent.txt
cat /home/roland/Flows/Host${NR}/accept.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/accept.txt
cat /home/roland/Flows/Host${NR}/acceptencode.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/acceptencode.txt
cat /home/roland/Flows/Host${NR}/acceptlang.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/acceptlang.txt
cat /home/roland/Flows/Host${NR}/cookie.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/cookie.txt
cat /home/roland/Flows/Host${NR}/connection.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/connection.txt
cat /home/roland/Flows/Host${NR}/contentype.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/contentype.txt
cat /home/roland/Flows/Host${NR}/contentlength.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/contentlength.txt
cat /home/roland/Flows/Host${NR}/cachecontrol.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/cachecontrol.txt
cat /home/roland/Flows/Host${NR}/pragma.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/pragma.txt
cat /home/roland/Flows/Host${NR}/range.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/range.txt
cat /home/roland/Flows/Host${NR}/ifunmodisince.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/ifunmodisince.txt
cat /home/roland/Flows/Host${NR}/dnt.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/dnt.txt
cat /home/roland/Flows/Host${NR}/ifmatch.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/ifmatch.txt
cat /home/roland/Flows/Host${NR}/ifnonematch.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/ifnonematch.txt
cat /home/roland/Flows/Host${NR}/ifmodisince.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/ifmodisince.txt
cat /home/roland/Flows/Host${NR}/origin.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/origin.txt
cat /home/roland/Flows/Host${NR}/ifrange.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/ifrange.txt
cat /home/roland/Flows/Host${NR}/x-flver.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/x-flver.txt
cat /home/roland/Flows/Host${NR}/x-requested-with.txt |tr "\\n" ","
   >> /home/roland/Flows/Host${NR}/Txtfiles/x-requested-with.txt
cat /home/roland/Flows/Host${NR}/soapaction.txt |tr "\\n" "," >> /
    home/roland/Flows/Host${NR}/Txtfiles/soapaction.txt
cat /home/roland/Flows/Host${NR}/upgrade-insecure-requests.txt |tr
    "\\n" "," >> /home/roland/Flows/Host${NR}/Txtfiles/upgrade-
    insecure-requests.txt
```

```
cat /home/roland/Flows/Host${NR}/x-moz.txt |tr "\\n" "," >> /home/
    roland/Flows/Host${NR}/Txtfiles/x-moz.txt

echo "end of program"
echo $COUNT
```

countnum.sh script count the linenumbers the headers are positioned on

This script uses the ouputfiles from the procflow.sh script as input. All the collected linenumbers are counted and totals per line number are presented in a file that also has a similar name as the request header.

```
#!/bin/bash
# this script counts the number of recurring numbers in a file and
    outputs this to the screen and a result file
# input is the filename of the file that needs to be counted
# Script needs to run only once to count the totals
for INPUT1 in *.txt
do
#Check presence of a certain number in the file (b1, b2 etc) and
    count them. Put all results in a variable.
NR1=`grep -o '\b1\b' ${INPUT1} |wc -l`
NR2=`grep -o '\b2\b' ${INPUT1} |wc -l`
NR3=`grep -o '\b3\b' ${INPUT1} |wc -l`
NR4=`grep -o '\b4\b' ${INPUT1} |wc -l`
NR5=`grep -o '\b5\b' ${INPUT1} |wc -l`
NR6=`grep -o '\b6\b' ${INPUT1} |wc -l`
NR7=`grep -o '\b7\b' ${INPUT1} |wc -l`
NR8=`grep -o '\b8\b' ${INPUT1} |wc -l`
NR9=`grep -o '\b9\b' ${INPUT1} |wc -l`
NR10=`grep -o '\b10\b' ${INPUT1} |wc -l`
NR11=`grep -o '\b11\b' ${INPUT1} |wc -l`
NR12=`grep -o '\b12\b' ${INPUT1} |wc -l`
NR13=`grep -o '\b13\b' ${INPUT1} |wc -l`
NR14=`grep -o '\b14\b' ${INPUT1} |wc -l`
NR15=`grep -o '\b15\b' ${INPUT1} |wc -l`
NR16=`grep -o '\b16\b' ${INPUT1} |wc -l`
NR17=`grep -o '\b17\b' ${INPUT1} |wc -l`
NR18=`grep -o '\b18\b' ${INPUT1} |wc -l`
NR19=`grep -o '\b19\b' ${INPUT1} |wc -l`
NR20=`grep -o '\b20\b' ${INPUT1} |wc -l`
NR21=`grep -o '\b21\b' ${INPUT1} |wc -l`
NR22=`grep -o '\b22\b' ${INPUT1} |wc -l`
NR23=`grep -o '\b23\b' ${INPUT1} |wc -l`
NR24=`grep -o '\b24\b' ${INPUT1} |wc -l`
NR25=`grep -o '\b25\b' ${INPUT1} |wc -l`
NR26=`grep -o '\b26\b' ${INPUT1} |wc -l`
NR27=`grep -o '\b27\b' ${INPUT1} |wc -l`
NR28=`grep -o '\b28\b' ${INPUT1} |wc -l`
NR29=`grep -o '\b29\b' ${INPUT1} |wc -l`
NR30=`grep -o '\b30\b' ${INPUT1} |wc -l`
NR31=`grep -o '\b31\b' ${INPUT1} |wc -l`
NR32=`grep -o '\b32\b' ${INPUT1} |wc -l`
NR33=`grep -o '\b33\b' ${INPUT1} |wc -l`
NR34=`grep -o '\b34\b' ${INPUT1} |wc -l`
NR35=`grep -o '\b35\b' ${INPUT1} |wc -l`
#Print the result on the screen, but also put it in a file
#Add .doc extension so th files are not picked up by the for loop
echo "The total number of 1s are: ${NR1}" |tee counted-${INPUT1}.doc
echo "The total number of 2s are: ${NR2}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 3s are: ${NR3}" |tee -a counted-${INPUT1}.
```

```
    doc
echo "The total number of 4s are: ${NR4}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 5s are: ${NR5}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 6s are: ${NR6}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 7s are: ${NR7}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 8s are: ${NR8}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 9s are: ${NR9}" |tee -a counted-${INPUT1}.
    doc
echo "The total number of 10s are: ${NR10}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 11s are: ${NR11}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 12s are: ${NR12}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 13s are: ${NR13}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 14s are: ${NR14}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 15s are: ${NR15}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 16s are: ${NR16}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 17s are: ${NR17}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 18s are: ${NR18}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 19s are: ${NR19}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 20s are: ${NR20}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 21s are: ${NR21}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 22s are: ${NR22}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 23s are: ${NR23}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 24s are: ${NR24}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 25s are: ${NR25}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 26s are: ${NR26}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 27s are: ${NR27}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 28s are: ${NR28}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 29s are: ${NR29}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 30s are: ${NR30}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 31s are: ${NR31}" |tee -a counted-${INPUT1
```

```
    }.doc
echo "The total number of 32s are: ${NR32}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 33s are: ${NR33}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 34s are: ${NR34}" |tee -a counted-${INPUT1
    }.doc
echo "The total number of 35s are: ${NR35}" |tee -a counted-${INPUT1
    }.doc
done
# also print the total amount of numbers present in the file
TOTAL=$(($NR1+$NR2+$NR3+$NR4+$NR5+$NR6+$NR7+$NR8+$NR9+$NR10+$NR11+
    $NR12+$NR13+$NR14+$NR15+$NR16+$NR17+$NR18+$NR19+$NR20+$NR21+$NR22+
    $NR23+$NR24+$NR25+$NR26+$NR27+$NR28+$NR29+$NR30+$NR31+$NR32+$NR33+
    $NR34+$NR35))
echo "The total amount of numbers is: $TOTAL" |tee -a counted-${
    INPUT1}.doc
echo "end of program"
```

B Consistency checking

This research relied on several scripts and third party programs to obtain the desired results. In order to be sure that the processing of these programs did not influence the results, some tests were done prior to using the programs for the research.

The goal of these tests was to be certain that the header order is not changed during processing. It was tested by looking with Wireshark at a dozen of small pcap files, with only a few headers and writing down the header order (it is assumed that Wireshark shows the correct header order). Next, the files were processed by the different programs (Tcpflow, Csplit) and after that, the header order was checked again. After individually checking the programs, the small pcaps were also completely parsed and processed and the end result was again compared with the header orders that were written down. These tests did not show any inconsistencies. Below are three pictures 12, 13 and 14 of a test file. On picture 12 the header order is shown of the test file in Wireshark. The startline will be removed during processing, so counting starts with the first header, in this case the 'Host:' header.

Picture 13 shows the header after processing by tcpflow. The header order is not changed. The number files that are the end result of the process show on what position a header was found. The picture 14 shows the position of the accept header, which is three. This is the correct position.

```
GET /data/ocs/section/index.html:intl_homepage1-zone-2/views/zones/common/zone-
manager.html HTTP/1.1
Host: edition.cnn.com
Connection: keep-alive
Accept: text/html, */*; q=0.01
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/44.0.2403.155 Safari/537.36
Referer: http://edition.cnn.com/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: optimizelyEndUserId=oeu1433673421230r0.3960122938733548;
grvinsights=ecf2697ff0ca68084c2787aff8ed4675;
__gads=ID=1532e5c6c844296b:T=1433673427:S=ALNI_MZ7sRPmsmb5rz1GDTIp0x8GTXGO0A;
notice_preferences=2:; s_fid=3322FF3717786476-0088B8C7EB230E61; s_vi=[CS]v1|
2ABA0F68051D3356-400001408008F420[CE]; __CT_Data=gpv=6&apv_49_www04=5; WRUID=0;
_chartbeat2=C8GJOtCxQGM6B8wpDQ.1433673434433.1437028245880.0000000000000001;
ug=55741ecc0d9d210a3c8f78728b01439a; ugs=1; optimizelySegments=%7B%22171825878%22%3A%
22direct%22%2C%22172244722%22%3A%22gc%22%2C%22172343962%22%3A%22none%22%2C%22172468417%22%
3A%22false%22%2C%22179260998%22%3A%22true%22%7D; optimizelyBuckets=%7B%7D;
optimizelyPendingLogEvents=%5B%5D
```

Figure 12: HTTP headers in Wireshark

```
GET /data/ocs/section/index.html:intl_homepage1-zone-
Host: edition.cnn.com
Connection: keep-alive
Accept: text/html, */*; q=0.01
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebK
Referer: http://edition.cnn.com/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: optimizelyEndUserId=oeu1433673421230r0.396012
673427:S=ALNI_MZ7sRPmsmb5rz1GDTIp0x8GTXGO0A; notice_p
408008F420[CE]; __CT_Data=gpv=6&apv_49_www04=5; WRUID
c0d9d210a3c8f78728b01439a; ugs=1; optimizelySegments=
22%2C%22172468417%22%3A%22false%22%2C%22179260998%22%
```

Figure 13: HTTP headers after processing by tcpflow

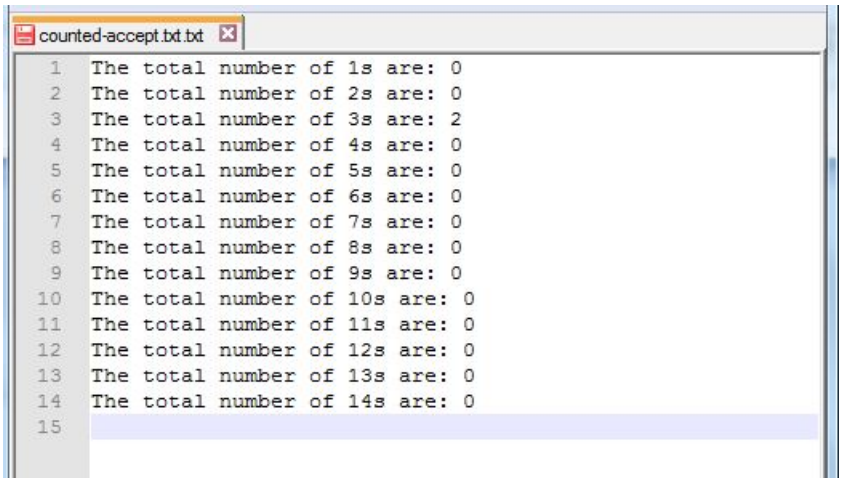


Figure 14: HTTP header number as endresult

C Header examples

This section shows several examples of headers that are used by specific programs or by the browser to retrieve webpages. Requests to the same webpage are mostly consistent. The nu.nl website shows consistency with different headers:

```
GET /sat/sat.gif?log=1&evt=con&cid=4108984&c_ver=2015.03.2(
artikel&c_vmspot=95133757_71000335&c_screen=1600x900&c_viev
terug-niveau-van-crisis.html&c_hash=&rid=1439985001740&cb=1
Host: sat.sanoma.fi
Connection: keep-alive
Accept: image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit
Referer: http://www.nu.nl/wonen/4108984/verkoop-nieuwbouwwo
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: SanomaWeb=06de7b2f-2263-42c6-a083-9fd49509bfa3

GET /sat/sat.gif?log=1&evt=con&cid=&c_ver=2015.03.20.10.15.
2036b6ded700f&c_events=0&c_session=sck2vyt7ua&c_id=ia9pgfik
_currenturl=http%3A%2F%2Fwww.nu.nl%2F&c_hash=&rid=14399850(
Host: sat.sanoma.fi
Connection: keep-alive
Accept: image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit
Referer: http://www.nu.nl/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: SanomaWeb=06de7b2f-2263-42c6-a083-9fd49509bfa3

GET /sat/sat.gif?log=1&evt=click&cid=&c_ver=2015.03.20.10.1
2036b6ded700f&c_events=1&c_session=sck2vyt7ua&c_id=ia9pgfik
editie-sail-amsterdam-van-start.html&c_selector=%5Bdata-sac
20start&c_custom_label=undefined&c_currentPage=http%3A%2F%2
20Time)&rid=1439985009074&cb=1439985013905&ctz=120 HTTP/1.1
Host: sat.sanoma.fi
Connection: keep-alive
Accept: image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit
Referer: http://www.nu.nl/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: SanomaWeb=06de7b2f-2263-42c6-a083-9fd49509bfa3

GET /js/sat.js HTTP/1.1
Host: sat.sanoma.fi
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit
Referer: http://www.nu.nl/amsterdam/4108813/negende-editie-
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: SanomaWeb=06de7b2f-2263-42c6-a083-9fd49509bfa3

GET /sat/sat.gif?log=1&evt=con&cid=4108813&c_ver=2015.03.2(
2036b6ded700f&c_events=0&c_session=sck2vyt7ua&c_id=ia9pgfik
=http%3A%2F%2Fwww.nu.nl%2F&c_currenturl=http%3A%2F%2Fwww.nu
Host: sat.sanoma.fi
```

Figure 15: Consistent HTTP header order of nu.nl

Also CNN.com shows a consistent header ordering:

Figure 16: Consistent HTTP header order of cnn.com

The Tweakers.net headers shows a fairly consistent ordering. Consistency is broken when probably a shockwaveflash item is requested (fourth header in the picture):

```
GET /downloads/35278/tor-browser-501.html HTTP/1.1
Host: tweakers.net
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Appl
Referer: http://tweakers.net/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: TnetID=1lR6VmaIPex_UOb6Y_y3MmfYIkwen377; pl=
_ga=GA1.2.1168111167.1439985218; wt3_eid=%3B31881670

GET /nieuws/104844/partijen-schikken-zaak-rond-recht
Host: tweakers.net
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Appl
Referer: http://tweakers.net/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: TnetID=1lR6VmaIPex_UOb6Y_y3MmfYIkwen377; pl=
_ga=GA1.2.1168111167.1439985218; wt3_eid=%3B31881670

GET /video/player/10545/3d-realms-geeft-e3-trailer-b
Host: tweakers.net
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Appl
Referer: http://tweakers.net/nieuws/104844/partijen-
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: TnetID=1lR6VmaIPex_UOb6Y_y3MmfYIkwen377; pl=
_ga=GA1.2.1168111167.1439985218; wt3_eid=%3B31881670

GET /video/s1playlist/10545/620/349/playlist.xspf?zo
Host: tweakers.net
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Appl
X-Requested-With: ShockwaveFlash/18.0.0.232
Accept: */*
Referer: http://tweakers.net/video/player/10545/3d-r
Accept-Encoding: gzip, deflate, sdch
Accept-Language: nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: TnetID=1lR6VmaIPex_UOb6Y_y3MmfYIkwen377; pl=
3B31881670584598%7C2143998521800136214%232143998523

GET /crossdomain.xml HTTP/1.1
Host: tweakers.net
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Appl
```

Figure 17: Fairly consistent HTTP header order of tweakers.net

The header order between the different websites is also reasonably consistent. Header order drastically changes when certain programs are run. The pictures 18, 19 and 20 are some examples of this.

Windows update uses only a few headers:

Figure 18: Windows update

Avast anti-virus update checking also uses some request headers:

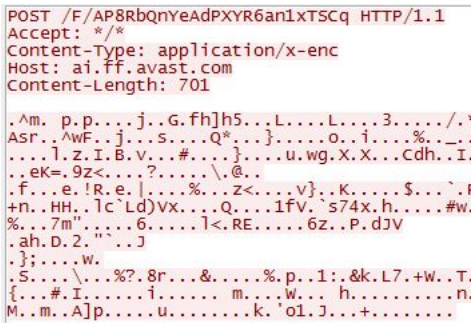


Figure 19: Avast anti-virus

IBM's Tealeaf program using a lot of experimental headers:

Figure 20: IBM Tealeaf

D Request header listings

In this section, the listings of headers that were retrieved from the pcap files are presented. This script was used as input for building the other scripts. The script uses the name of the pcap file as an input parameter. It creates HTTP request flows with the Tcpflow program and processes these. All header content is removed and the headers are saved in an output file. Then a 'cat' command using sort, unique and grep options sorts this file and removes duplicate entries.

searchhdr.sh script for making an inventory of the headers

```
#!/bin/bash
# search different type of headers present in pcap files
tcpflow -e http  -o /home/roland/output -r /home/roland/$1.pcap dst
    port 80
for file in /home/roland/output/*00080;
do
cat $file | sed -e 's/:.*$/: /g' -e 's/GET.*$/ /g' -e 's/HEAD.*$/ /g'
    -e 's/POST.*$/ /g' -e 's/^  <.*$/ /g' -e 's/^<.*$/ /g'  >> /home/
    roland/outfile.txt
done
rm /home/roland/output/*
echo "end of program"
```

The command used to view the header inventory file is:

```
cat outfile.txt |sort|uniq|cut -c-45|grep -a ': '
This line was used to clear readable overview and filter out possible
    unwanted matches in the output file.
```

Twenty-three headers were found in the pcap files from the baseline systems.

```
Accept:
Accept-Encoding:
Accept-Language:
Cache-Control:
Connection:
Content-Length:
Content-Type:
Cookie:
Host:
If-Match
If-Modified-Since:
If-None-Match:
If-Range:
If-Unmodified-Since:
Origin:
Pragma:
Range:
Referer:
SOAPAction:
Upgrade-Insecure-Requests:
User-Agent:
X-Moz:
X-Requested-With:
```

After analyzing the pcap files from the Fox-IT systems, seventy-five headers were retrieved. This includes four headers that were written both with upper- and lowercase characters.

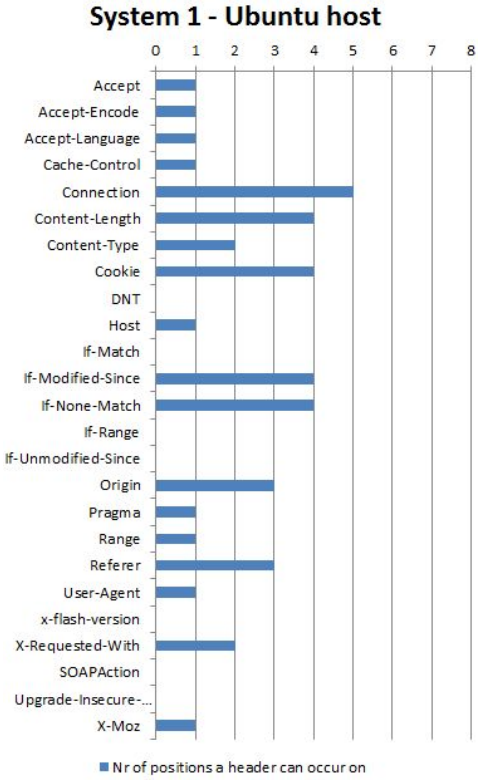
```
Accept:
accept-encoding:
Accept-Encoding:
Accept-Language:
Ajax-Request:
Authorization-Code:
Cache-Control:
Connection:
Content-Disposition:
Content-Length:
Content-Type:
Cookie:
Cookie:
DNT:
Host:
If-Match:
If-Modified-Since:
If-None-Match:
If-Range:
If-Unmodified-Since:
Origin:
Pragma:
Range:
Referer:
__RequestVerificationToken:
Rest-Authorization-Code:
Sec-WebSocket-Key:
Sec-WebSocket-Protocol:
Sec-WebSocket-Version:
SOAPAction:
UA-CPU:
UA-Java-Version:
Upgrade:
user-agent:
User-Agent:
x-Akamai-Streaming-SessionID:
X-APP-VERSION:
X-Booking-AID:
X-Booking-Exp:
X-Booking-Pageview-Id:
X-Booking-Session-Id:
X-CorrelationId:
X-CSRFToken:
X-Disqus-Publisher-API-Key:
X-DNT-Version:
x-flash-version:
X-IDCRL_ACCEPTED:
X-Last-HR:
X-Last-HTTP-Status-Code:
X-Moz:
X-NEW-APP:
X-NewRelic-ID:
```

```
X-Office-Version:
X-Old-UID:
x-prototype-version:
X-Prototype-Version:
x-requested-with:
X-Requested-With:
X-Retry-Count:
X-Signature:
X-TeaLeaf:
X-TeaLeaf-Browser-Res:
X-TeaLeaf-Page-Cui-Bytes:
X-TeaLeaf-Page-Cui-Events:
X-TeaLeaf-Page-Dwell:
X-TeaLeaf-Page-Img-Fail:
X-TeaLeaf-Page-Last-Field:
X-TeaLeaf-Page-Render:
X-TeaLeaf-Page-Url:
X-TeaLeaf-Screen-Res:
X-TeaLeafSubType:
X-TeaLeafType:
X-TeaLeaf-UIEventCapture-Version:
X-TeaLeaf-Visit-Order:
X-Verify:
```

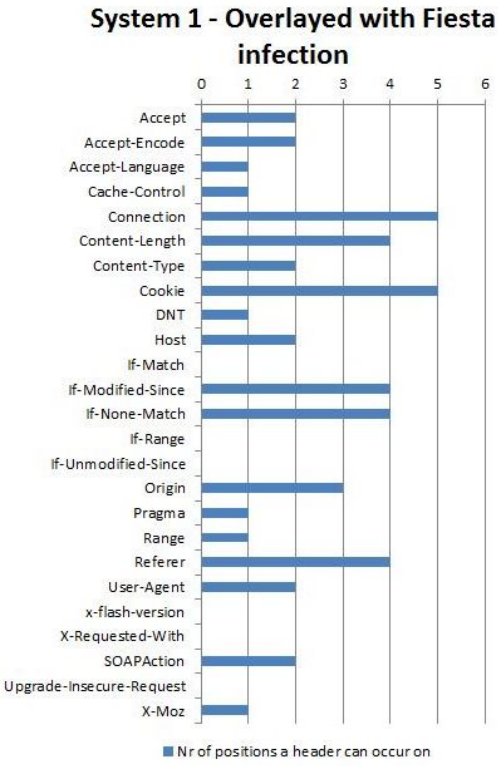
E Graphs

This section contains the header occurrence comparisons between the baseline systems en the Fiesta and Nuclear EK infections. Furthermore it also contains the graphs regarding these infections overlaid on the Fox-IT systems.

The graphs show on how many different positions a header was seen. When a bar of the graph reaches to the number four, then this means that the header was seen on four different positions on this system.

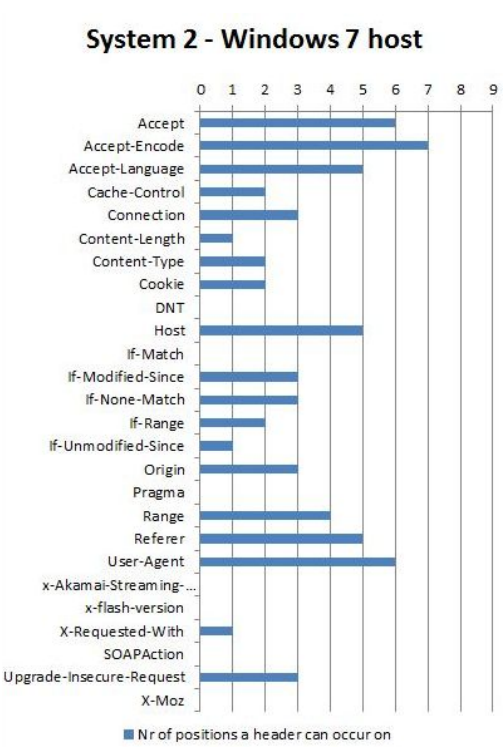


(a) System 1 - uninfected

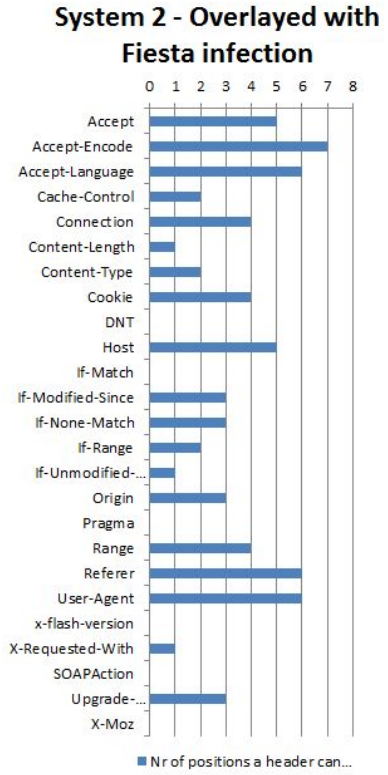


(b) System 1 - with Fiesta infection overlayed

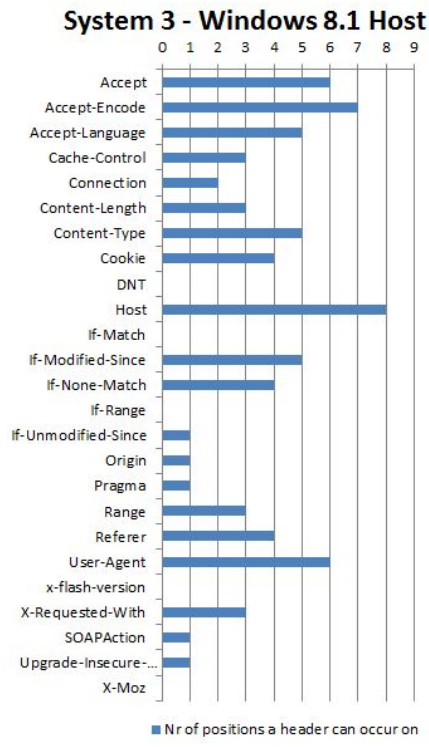
The change in header order after the Fiesta infection is overlaid is clearly visible. For the two Windows hosts in the pictures 22a, 22b, 23a en 23b, this distinction is less clear. From all three used exploit kits, the Fiesta exploit kit has the least impact on header order changes.



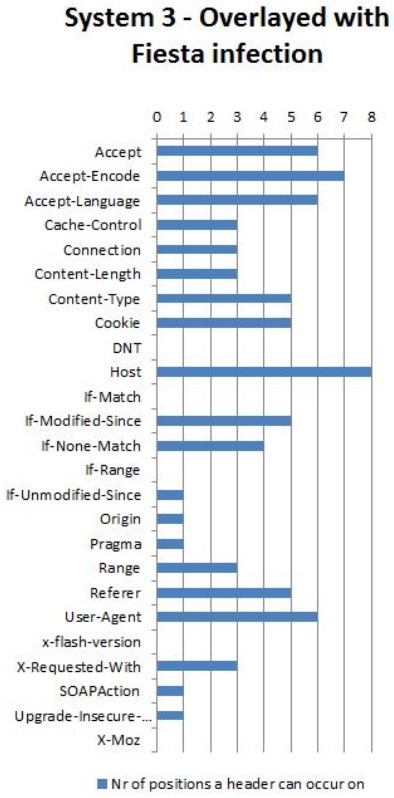
(a) System 2 - uninfected



(b) System 2 - with Fiesta infection overlayed



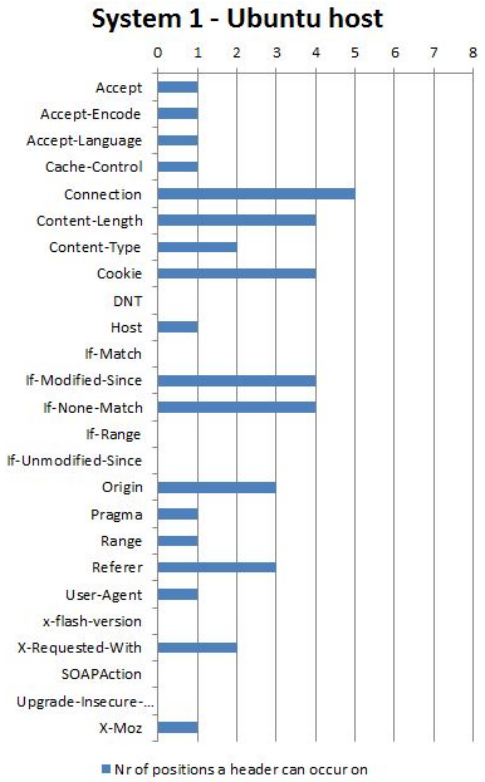
(a) System 3 - uninfected



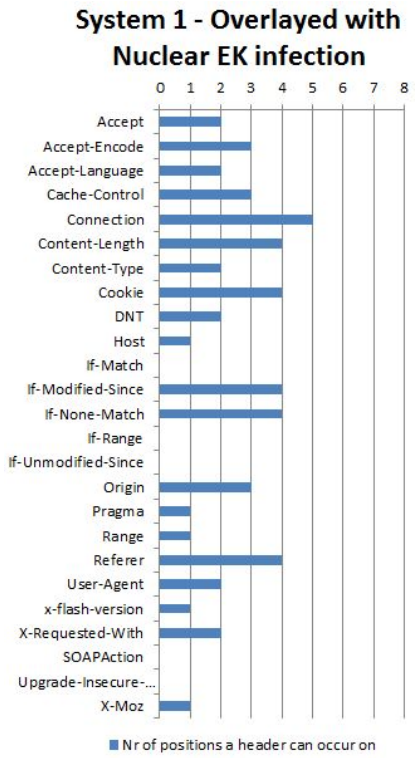
(b) System 3 - with Fiesta infection overlayed

The Nuclear EK infection also has a minor impact on header order change. This is also the case

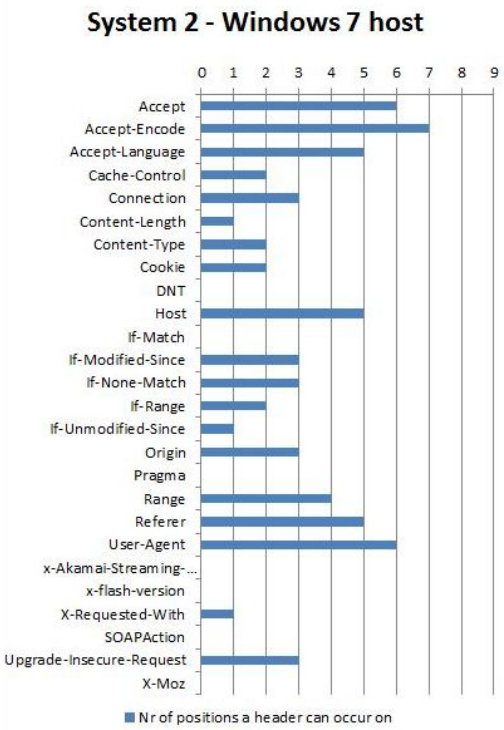
for the Windows hosts.



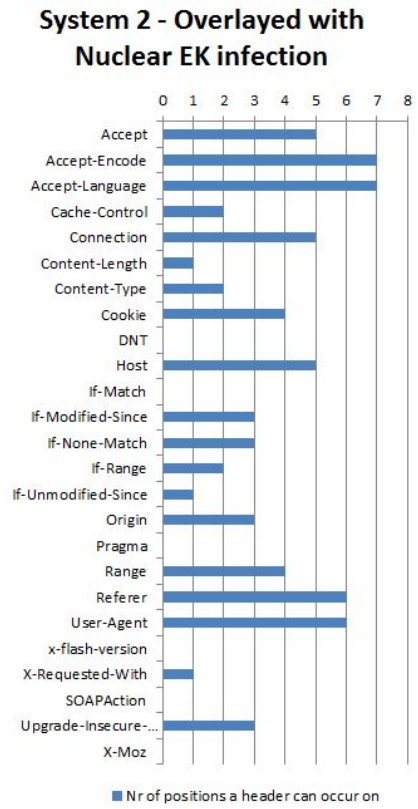
(a) System 1 - uninfected



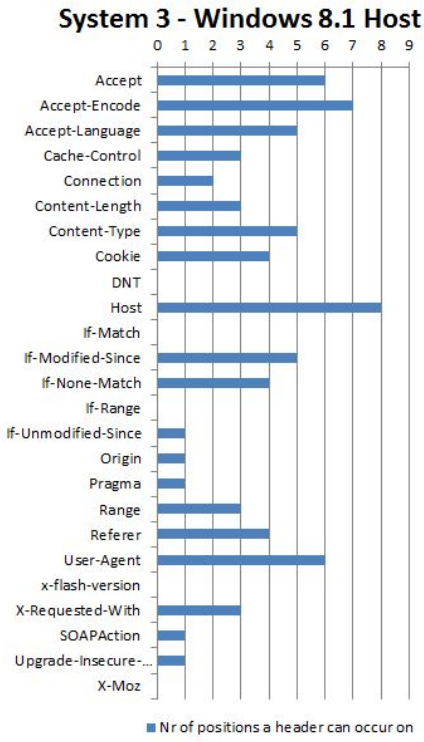
(b) System 1 - with Nuclear EK infection overlayed



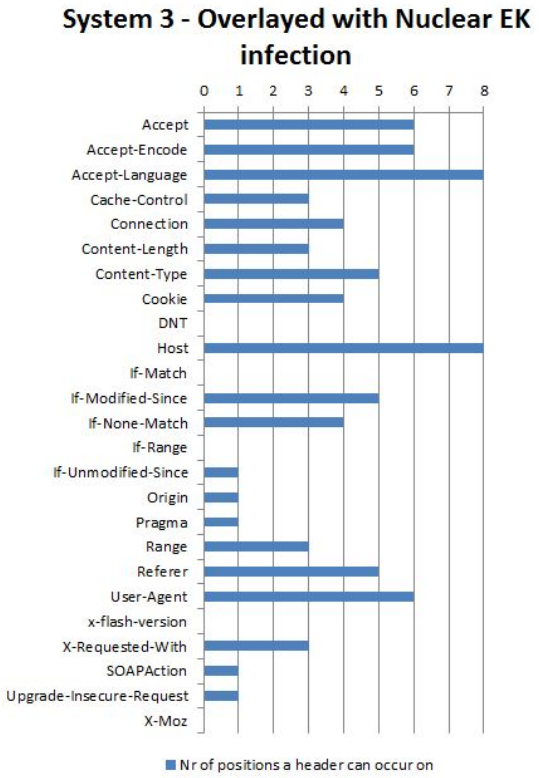
(a) System 2 - uninfected



(b) System 2 - with Nuclear EK infection overlayed



(a) System 2 - uninfected



(b) System 3 - with Nuclear infection overlayed