A JOURNEY INTO MALWARE HTTP COMMUNICATION CHANNELS SPECTACLES

An exposé of multiple malware families that fell into the trap of making mistakes in the construction of the HTTP request and response headers

Mohamad Mokbel
Trend Micro TippingPoint – DVLabs
mohamad_mokbel@trendmicro.com

Table of Contents

INTRODUCTION

Over the years, malware authors have used different communication protocols that sit at various layers in the OSI model to establish an exchange link with its command and control server(s), be it either for registering the infection, exfiltrating data, performing DDoS attacks against specific targets, relaying traffic, or any other specific control commands functionality. Such protocols include standard and custom ones, operating mainly over TCP and UDP transport layers, in binary or plain-text formats. List of standard protocols used by different malware families include IRC, FTP, SMTP, HTTP(S), SMB, TLS/SSL, and SSH among others. Alternatively, sometime malware authors design their own custom communication protocol, with varying levels of complexities, mainly in binary format. Perhaps, the most common binary custom protocol is the one used by the PCRat families of malware, which consists of a 5-byte identifier set at the time of compilation, a dword value indicating the size of the packet, followed by another dword value holding the size of the payload before compression, and finally the payload containing exfiltrated data, zlib compressed.

In certain cases, the same malware might support more than one protocol, one for receiving control commands (e.g. HTTP) while another (e.g. FTP) for exfiltrating data to a drop zone.

At the beginning, the Internet Relay Chat (IRC) application layer - text-based protocol, was the most commonly used by malware authors, for C&C communication and building large botnets. This was due to the fact that the protocol was easy to use and integrate, and remote control was already centralized via a ready-made IRC client. However, it wasn't long before enterprises started to block this protocol at the perimeter level due to its non compliance with the company's policies, either via the firewall by blocking associated default port numbers, or via an IDS/IPS signature. This gave attackers the reason to adopt something more mainstream, that forms the underlying mechanism by which the WWW functions, that is the HTTP - application layer protocol. Nowadays, a huge number of malware families use the HTTP protocol as its primary channel for C&C communications, and finding a malware that communicates over the IRC protocol is a rare occurrence.

There is a common misconception among inexperienced practitioners in the field, thinking that all it takes to block an application layer protocol, is to block its default number in the firewall, and that's it. A protocol is never defined by its port number, it is the semantics and architectural design and implementation that defines it. Thus, you are at will to change the default port number, and choose another one, depending on your deployment and configurations.

Over the last decade, as malware C&C communications shifted its focus into HTTP, certain peculiarities, intentional or nonintentional, blunders, and obvious errors in the usage of the protocol were spotted. For example, using specific headers in a GET request that only make sense in a POST request, or using wrong Content-Length value that doesn't match the actual payload's size, and the use of a unique non-standard header in a non-standard compliant way among others. Those "mistakes" could be leveraged by researchers to their advantage when writing IDS/IPS signatures for detecting malware C&C traffic, even if the core of the traffic is not filterable due to a high chance of false-positive (FP) or potential performance impact.

Since the HTTP protocol follows the client-server model, malware has full control over the client and server setup and communications, unlike a regular client such as a web browser that sends HTTP requests to different web servers in the most possible RFC standard conformant way. Malware need not use the default port number (80) for the HTTP protocol, nor honor the semantics of the protocol; it could be merely emulating HTTP request-response traffic without actually using the protocol at all. So, why would an attacker do that? It is simply done so that C&C beaconing slips through network defences inconspicuously, be it a firewall, a proxy or an IDS/IPS device.

Another shared misconception among non-developers is the assumption that just because a malware is using either of the Windows libraries, WinHTTP or WinINet to communicate with its C&C server, then it's suggested that it must be HTTP RFC standard compliant. Or, the other way, just because a malware is using the Windows Sockets Winsock library for sending and receiving HTTP requests, then, it is assumed that it is not HTTP.

In this paper we will document and explore different malware families that fell into the trap of making mistakes in the construction of the HTTP request and response headers, be it intentional or nonintentional. This is not primarily about typographical errors, but rather an endeavour at documenting semantic- and syntax-level type errors. Moreover, we'll attempt to answer certain posited scenarios that emerge as a by-product of some of the deliberate mistakes made by the attacker that could lead to breaking automated systems or other devices that handle HTTP traffic. Furthermore, we will show how those mistakes could be used to our advantage for detecting such anomalous traffic that deviates from normal and standard compliant HTTP requests and responses. To our knowledge, this is the first paper that attempts to survey, document and perform root-cause analysis on such cases.

USE-CASES ANALYSIS

In this section we'll research several malware cases that have certain types of errors in their HTTP C&C communications. Also, a technical breakdown of how those errors are committed at the code level will be provided for better understanding of the suspected reason(s) behind them. Whenever possible, on a case-by-case basis, a detection guidance will be outlined that helps in tracking and spotting those anomalies as they traverse the network, or when data mining HTTP traffic captures looking for suspicious indicators.

Note: the value of the Host header-field in all of the packet decodes has been replaced with "example.com", or in case of an IP address, it is replaced with "10.10.10.1", unless noted otherwise. Moreover, Appendix A lists the hashes of some of the malware families referenced in this paper.

1. The mysterious case of PKEY (also known as Adelinoq)
    - Wrong Content-Length value that doesn't match the actual payload's size

This case is particularly interesting because of the way the malware computes the total size of the POST request and the error it commits when sending it. For example, it sends the following request (domain name is replaced with a dummy one with the same length):

```
POST /link.php HTTP/1.1
Host: abcdefghij[.]com
Accept: text/html, */*
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Connection: close

// payload is shown in hexadecimal
000000A0  50 4b 45 59 00 00 00 00  00 00 00 00 00 00 00 00 PKEY.... ........
000000B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000001A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00    ........ .......
```

Note the discrepancy between the value ("4") of the Content-Length header and the actual payload size ("271"). The intended payload size is actually 4, that is the payload string "PKEY" only. So, how did the malware end up constructing above request as such? While calculating the total length of the POST request, including the payload, so that it copies it to a new buffer on the heap (initialized to zero) before sending it to the server, it performs the following computations:

```
length_of("uri_file_name") + length_of("post_req_format_pattern") +
length_of("payload_str") + length_of("C&C_server_addr") + 0x104 = packet size
```

For example,

```
length_of("/link.php") + 0x90 + length_of("PKEY") + length_of("abcdefghij[.]com") +
0x104
= 0x09 + 0x90 + 0x04 + 0x0E + 0x104
= 0x1AF (431) -> packet size
```

Right before sending above payload, the malware copies the data to a new buffer on the heap, initialized to zero, with a size of 431 bytes, and then sends the packet to the C&C server. Thus, why the rest of the POST payload consists of zeros. The valid size of the packet should be 164 bytes only.

The malware uses the Winsock library ws2_32.dll, using the *send* function to communicate above packet to the server.

The following code snippets illustrate how the malware calculates the size of the packet and sends it:

```
00409FA9    push    99h
00409FAE    push    offset post_req_format_pattern
00409FB3    call    decrypt_data
00409FB8    pop     ecx
00409FB9    pop     ecx
00409FBA    mov     ebx, eax
00409FBC    push    ebx
00409FBD    call    get_str_len
00409FC2    push    [ebp+buf]              ; C&C server address
00409FC5    mov     esi, eax               ; POST request headers length
00409FC7    call    get_str_len            ; 0xE
00409FCC    cmp     [ebp+PKEY_length], 0
00409FD0    push    [ebp+s]                ; URI file name
...
0040A01E    mov     edi, eax
0040A020    add     edi, [ebp+PKEY_length] ; Add the length of "PKEY" and the C&C
                                             address "abcdefghij[.]com"
0040A023    call    get_str_len            ; URI filename len ("/link.php"); EAX = 0x9
0040A028    add     edi, esi               ; 0x12 + 0x90 (length of the POST request
                                                          headers) = 0xA2
0040A02A    lea     esi, [eax+edi+104h]    ; 0x9 + 0xA2 + 0x104 = 0x1AF
0040A031    push    esi                    ; 0x1AF (431)
0040A032    call    heap_alloc
...
0040A131    push    0                      ; flags
0040A133    push    esi                    ; len (431 bytes)
0040A134    push    edi                    ; buf
0040A135    push    ebx                    ; s
0040A136    call    ds:send
```

It is not clear why the malware would go to such length just to calculate the size of the packet, instead of using the *strlen* function and then creates the buffer on the heap with the correct size. Two hypotheses could be attributed to this observation, it is either a nondeliberate mistake, or a deliberate one that's meant to thwart automated systems or analysts from communicating with the server, should the payload not meet that pattern. Unfortunately, the server wasn't alive at the time of testing to confirm the deliberate mistake hypothesis. With everything taken together, this is highly likely a nondeliberate mistake, committed by the author of this malware.

Another anomaly in the request is that it is missing a User-Agent header!

Depending on the IDS/IPS engine you're using, detecting above traffic could be captured with this Snort rule:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"Adelinoq Detection";
flow:to_server; content:"POST "; offset:0; depth:5; content:"|0A|Content-Length|3A|
4|0D|"; offset:16; depth:384; content:"|0A 0D 0A|PKEY|00 00 00|"; fast_pattern;
distance:0; within:256;)
```

Detection could be generalized in case of data mining network traffic captures, by checking the value of the Content-Length header against the payload's size, and if a mismatch is detected, then the traffic should be deemed suspicious. Such detection would have a great performance impact if to be deployed inline.

2. The mysterious case of 'reverse gear'
    - Direction of HTTP Request and Response is inverted

This case is very unusual since after the malware establishes a successful TCP-handshake with the server, it sends the following HTTP response-like to the C&C server, before receiving any request:

```
HTTP/1.1 200 OK
Server: sffe
Content-Length:53
Connection:Keep-Alive

// payload is shown in hexadecimal
0000004B  9f 98 85 84 92 88 96 84  84 92 85 83 f6 00 00 00   ........ ........
0000005B  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0000006B  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0000007B  00 00 00 00 00                                     .....
```

And the C&C server responds with:

```
GET /index.html HTTP/1.1
Accept:*/*
Host:127.0.0.1
Content-Length:30
Connection:Keep-Alive

// payload is shown in hexadecimal (A payload associated with a GET request!)
00000062  96 84 84 92 85 83 88 98  9c f6 00 00 00 00 00 00   ........ ........
00000072  00 00 00 00 00 00 00 00  00 00 00 00 00 00         ........ ......
```

Despite the structure of the traffic shown above, the malware is actually not using the HTTP protocol at all (despite what Wireshark tells you!), those headers are used only for camouflaging the real communication protocol that is the payload data in both, the response and the request. The payload data contains XOR encrypted (key: 0xd7) string(s). The malware doesn't parse nor construct the response headers in a dynamic way, since it is stored fixed, as it is. For decrypting the server response (GET request), it XORs the entire packet (including the headers!), searching for the string "ASSERT_OK!" to validate its communication with the server. In the request (response) packet, the malware sends the string "HORSE_ASSERT!" XOR encrypted, in the payload. All of the malware communications carry this disguised traffic structure.

The malware uses the Winsock library ws2_32.dll, using the *send* function to communicate above packet to the server, and the *recv* function to receive packets from the server.

Some observations about anomalies in this traffic include:

1. The HTTP headers even attempt to mimic the response as if it's coming from Google web server (sffe)
2. Although it doesn't violate the RFC, but zero space between the header name and its value is less common in legitimate traffic
3. Host header pointing to loopback address, yet the request's direction is ingress towards a particular host
4. GET request with a body!

So why would the malware use this method to cloak the real communication protocol in a reversed HTTP type request? We don't know the exact reasons, but we can speculate on that. Attempting to blend with normal traffic and bypass certain proxy server rules, or getting caught by the proxy! Additionally, without netflow context, the traffic might appear to the untrained eye as local to the network.

How to detect this irregular traffic? Detecting the actual malware traffic is easy, but generalizing detection to cover anomalies requires some thought process and careful considerations.

First, let's look at the case where a payload/body is attached to a GET request. Although it is not strictly forbidden by the RFC7231, however, it is not recommended, and allowing or disallowing it is server dependent. Thus, depending on your IPS/IDS engine, you could monitor for such traffic by checking for the method GET AND (the presence of a (body OR Content-Length header)). Moreover, the GET request doesn't even come with a Content-Type header, informing the server about the type of data to expect (will come to this point later on)!

Second, zero space character between the header name and the value.

Third, checking the Host header for localhost IP address or any other local IP in the context of a remote request.

Fourth, taking the directionality of the request/response into consideration, that is, a Google web server (or any other similar web servers' names) traffic originating from your network, outbound.

Another malware that falls into the same trap is the Volgmer malware, which communicates with its C&C server by sending GET requests with a body, and the server responding with a HEAD request with a body. In both directions, the body payload is the actual malware communication protocol.

3. The mysterious case of ProtonBot GET request
    - Content-Type header in a GET request

This case confirms the misleading use of the Content-Type header in a GET request. In a POST or PUT request, the Content-Type header is used to signal to the server what type of data is being sent. If used in the server response, it informs the agent what type of data the server is sending. However, this is not how the malware family known as ProtonBot uses it, and for example, it instead sends a GET request similar to the following:

```
GET /page.php?id=1A28798B-9001-51CF-710A-89AF207D10F2&clip=get HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Proton Browser
Host: example.com
Cache-Control: no-cache
```

Such request won't be even parsed by Wireshark HTTP decoder. We can say with confidence that using said header in a GET request similar to the one shown above, doesn't make sense. Therefore, it is an anomaly in its own right when seen in network traffic and should be inspected accordingly.

Searching network packet captures for such incongruity can be done simply by checking for the presence of the Content-Type header in non-POST/PUT requests.

4. The mysterious case of Crosswalk
    - Content-Length header value is too large

This case illustrates couple of interesting observations in relation to the malware's request and the server response. For example, the malware sent:

```
GET http://10.10.10.1/QUERY/en-us/msdn/ HTTP/1.1
dCy: RjFDRDJGSkcta2N0YTJOMFlUSk9NRmxVUw==1
Connection: Keep-Alive
Host: 10.10.10.1
Content-Length: 0
```

And the server responded with:

```
HTTP/1.1 200 OK
Content-Length: 524288000
Connection: keep-alive

// payload is omitted for readability purpose
// payload is of binary type and actual size is 256 bytes
```

Following is a list of key observations of the traffic shown above:

- The custom header "dCy". Although it is no longer mandatory that a custom HTTP header has to start with the characters "X-", it should still be treated with suspicion.
- The presence of the Content-Length header in a GET request. This doesn't make sense. However, you'd be surprised to realize that even some legitimate applications use it in this way too. This is either due to copy-and-paste code snippets, or the developer not being diligent enough to understand the requirements of the request.

- The thing that stands out the most in the above stream is the Content-Length header with the value 524288000 bytes (524.288 MB). Considering that the actual size of the payload is 256 bytes, this is surely not true!
    - Why would the server respond with such a value that doesn't represent the actual payload size? It is either to flood an automation system that attempts to interact with the C&C server in order to milk server responses, or it is a genuine mistake by the malware's author. In case the automation system sends multiple requests to the C&C, all in parallel, and submissively parses the server response HTTP headers in order to allocate buffer for the payload on the heap, then the system might end up exhausting its memory resources in a short period of time. Note that the malware uses the Winsock *recv* function to read the server response, thus, the malware itself is not vulnerable to such behavior.

Depending on how the server response is parsed, such request could be detected by comparing the Content-Length header value against the actual payload size; this could be done with the WinHTTP library, or, manually parsing and inspecting the header's value against the payload's size, and noting any discrepancy.

5. The mysterious case of Socnet
    - Illegal characters in header name

This case reveals how an attacker could end up making a blatant mistake in the construction of the request headers. The mistake in this case is the presence of illegal characters in the Connection header name, as follows:

```
GET /ab/setup.php?act=tw_get HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/532.5 (KHTML,
like Gecko) Chrome/4.1.249.1064 Safari/532.5
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q
=0.5
(null)Connection: close
```

As evidenced in the request shown above, the Connection header is preceded by the illegal string "`(null)`". Accepting or rejecting such request depends on how the HTTP server parses and implements the standard specifications. In this case, the malware's C&C server was running Apache/2, and it did respond with the 200 OK status code! Testing such malformed request against the latest version of Apache gets it rejected with the 400 Bad Request status code. It is surprising that the C&C server accepted it, since the header-field name is not allowed to contain the separators "(" and ")". This request will also pass on Nginx web server with the 200 OK status code. Nginx doesn't inspect any header for such anomalies.

The string "`(null)`" is not hardcoded in the code. The entire request layout is stored in a C string with couple of format specifiers as shown below. Note the format specifier "%s" preceding the Connection header. There is no reason for this specifier to exist at such location in the string. The value of this particular specifier is coming from a structure (allocated on the heap) of size 68 bytes, initialized to zero at the time of creation. In particular, it is the data member at offset 8 in the structure that is referenced

```
0042B378 db 'GET %s HTTP/1.1',0Dh,0Ah
              db 'Host: %s',0Dh,0Ah
              db 'User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) Apple'
              db 'WebKit/532.5 (KHTML, like Gecko) Chrome/4.1.249.1064 Safari/532.5'
              db 0Dh,0Ah
              db 'Accept: application/xml,application/xhtml+xml,text/html;q=0.9,tex'
              db 't/plain;q=0.8,image/png,*/*;q=0.5',0Dh,0Ah
              db '%sConnection: close',0Dh,0Ah
              db 0Dh,0Ah,0
              align 4
```

with respect to this specifier. However, this data member is never set anywhere in the code, and the only contextual reference to it is when used as a time-out interval for suspending the execution of the current threat using the Sleep function. Irrespective of whether this data member could be set remotely or not, this specifier in the request header is a mistake, and accordingly we end up with the string “(null)” instead.

Generalizing detection for such type of mistake is not simple, since it might collide with custom headers, and the recommended way to do it is by searching for any illegal character in the header-field name, as documented in the specification.

6. The mysterious case of Cobra, Aytoke and Klokla
    - Whitespace characters at the end and beginning of the headers

This is another similar case to Socnet whereby a whitespace character is present in the format of the request as well as in the values of every header. The malware sends a request similar to the following (the whitespace character is shown highlighted in dark green with the imposed character ‘**SP**’):

```
GET http://example.com/index01.html HTTP/1.1SP
Accept-Language: en-usSP
Accept: */*SP
User-Agent: MozillaSP
Proxy-Connection: Keep-AliveSP
Host: example.comSP
Cache-Control: no-store, no-cacheSP
Pragma: no-cacheSP
Cookie: <exfiltrated data (encoded)>SP
```

Those whitespaces are hardcoded in the malware format string pattern. The malware uses the function *send* from the Winsock library to send above request to the C&C server. At the time of testing the sample the server wasn’t responding. It is unlikely that the malware C&C server is HTTP compliant or that it uses the HTTP protocol for that matter; it is likely to consist of a custom parser.

Testing such request against Nginx server returns the 200 OK status code, whereas in the case of Apache server, the server responds with the 400 Bad Request status code, exclusively when the ‘**SP**’ is after the protocol version number.

In the Aytoke case, the ‘**SP**’ is at the beginning of the Connection header. This is an illegal character in the header-field name. However, the server accepted the request, and returned the expected response. The server could be simply accepting the request instead of rejecting it, and ignoring this header. The request is as follows:

```
GET / HTTP/1.1
Host: checkup[.]dyndns[.]org
SPConnection: close
```

```
|00|
```

The only header that's required in an HTTP 1.1 request is the Host header as per RFC2616 "*A client MUST include a Host header field in all HTTP/1.1 request messages.*".

Accepting or rejecting a request usually follows the robustness principle as clarified in the RFC2145, original RFC791 (section 3.2), "*In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior. That is, it must be careful to send well-formed datagrams, but must accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).*". Of course, this is implementation and interpretation dependent, whereby different HTTP servers need not behave the same way.

Another anomaly in the request is the one-byte (\x00) body payload, similar to the "reverse gear" case.

Those mistakes are non-deliberate, and this is a perfect example of how we could leverage such blunders to our advantage. A similar case was outlined by Fox-IT InTELL team, where an extraneous whitespace character was present in Cobalt Strike server (NanoHTTPD) response, at the end of the status code description, which was used for detecting active Cobalt Strike team servers.

Detecting spaces at the end or beginning of the headers could be matched with a content payload, or preferably using a built-in HTTP primitive such as the *http_header* in Snort.

The case of Klokla is a clear violation of the RFC, whereby the malware constructs some of the header-field names with a space before the colon. For example, the malware sends a request similar to the following:

```
POST /test/add.php HTTP/1.1
HostSP: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept(-Encoding)SP: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------84732825325733
Content-Length: 1234
```

As shown above, the Host and Accept-Encoding headers field-name end with a space before the colon, and according to the RFC7230 for HTTP/1.1, section Field Parsing, "*No whitespace is allowed between the header field-name and colon*". The server must reject such request with a response code of 400 (Bad Request). Such restriction is required for security purposes. And the vulnerability identified by the CVE-2019-16276 in the Go's HTTP library net/http, is a clear demonstration of what could go wrong when such violation in the standard happens.

Notice the wrong header-field name "Accept(-Encoding)", with the parentheses. The correct header is "Accept-Encoding."

Klokla constructs the entire request's headers manually, and uses the Socket library for sending above request, instead of using an HTTP library. Surprisingly, the malware C&C server accepted above request with the status code 200. The server was running Apache with PHP/5.3.28.

7. Accept header shenanigans
    - The many cases of malware Accept header blunders

The Accept request header is meant to signal to the server what content type(s) the client is able to understand. The syntax of this header consists primarily of a <mime_type>/<sub_type>, <mime_type>/* or */*. The MIME type is standardized in RFC6838, and a list of all official types are defined by IANA. Since the syntax of the header is known in advance, mining traffic captures looking for aberrations could be implemented with a simple regular expression; for example, checking for the absence of the "/" character in the header-field value, or checking it against a whitelist. Performing the former check against 2769 of HTTP malware traffic capture files reveals 18 families that do not adhere to the standard's defined syntax. Such aberrations in the value of the Accept header-field include, empty value, one ascii character, one or two non-ascii characters, the values: "xml", "*.*", a domain name value or a nonsense value (e.g., dots and consecutive commas).

8. Cookie header oddities and other peculiarities
    - The many faces of malware Cookie header

The HTTP Cookie header is used for tracking, session management or customization. It is requested by the server via the Set-Cookie header to store whatever Cookie on the client side. The Cookie header has a standardized syntax that consists of a list of name-value pairs, where the name is separated from the value by the equal '=' sign, and the pair by a semicolon and a space "; ". Searching the same set of packet captures used in case 7, looking for the absence of the separator '=' in the Cookie name-value pair(s) exposes 13 families that do not adhere to the standard's defined syntax. Results include, 9 of the families have the header field value empty, 1 has a hardcoded string, and 3 used it for data exfiltration. Checking for the pair separator ';' such that it is not followed by a space, reveals the malware family Noobot (not in the results of the first check), exfiltrating data via the Cookie header. More fine-grained checks on the syntax could be performed, for example, checking for non-allowed characters in the header's value.

For example, a variant of the Sefnit malware family sends a request similar to the following:

```
GET favicon.ico HTTP/1.1
Host: example.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg
Accept-Language: en-us
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.0.1)
Cookie: 5b1ac9eff0694a3dbd7add2cd8ecad580cec7db8.
Content-Length: 10038
Max-Forwards: 107719
|00|
```

The Cookie header-field value is clearly doesn't follow the standard syntax specification. However, above request is full of errors for it to be a valid HTTP GET request.

Some observations about anomalies in this traffic include:

1. The URL points to the filename "favicon.ico" without a forward slash
2. The Content-Length header does not represent the length of the payload being sent; it is actually a dynamically constructed value which represents the port number the malware will listen on for inbound communications

3. The Max-Forwards header field is used with either the TRACE or OPTIONS request methods and not GET, for limiting the number of forwards the proxies have to carry
    a. The value “107719”is hardcoded and doesn’t make much sense in this context!
4. The body payload \x00 is not separated from the request headers by CR LF characters
5. Body payload with a GET request, as in case 2

Wireshark HTTP decoder won’t recognize above request as a valid GET request. It is unlikely that the malware C&C server is HTTP compliant or that it uses the HTTP protocol for that matter at all; it is highly likely to consist of a custom parser. The HTTP headers are mainly used as a camouflage for data exfiltration, and the request will pass as a valid HTTP GET request for the untrained eye.

The malware uses the Winsock library ws2_32.dll, using the *send* function to communicate above packet to the server, and the *recv* function to receive packets from the server.

Generalizing the detection of the anomalies in the traffic shown above align with what have already been discussed in the previous cases.

9. Silencing Silence (the mysterious case of Silence/TrueBot)
    - Multiple new lines between headers

This case clearly exemplifies how a developer could commit a blatant mistake, and yet still go unnoticed. The backconnect module of the Silence/Truebot malware sends a request over port 80 similar to the following:

```
GET /index.php?xy=1 HTTP/1.1
User-Agent:


Host: 10.10.10.1
Connection: Keep-Alive
```
TCP Stream - ASCII

```
00000000  47 45 54 20 2f 69 6e 64  65 78 2e 70 68 70 3f 78   GET /ind ex.php?x
00000010  79 3d 31 20 48 54 54 50  2f 31 2e 31 0d 0a 55 73   y=1 HTTP /1.1..Us
00000020  65 72 2d 41 67 65 6e 74  3a 20 0d 0a 0d 0a 0d 0a   er-Agent : ......
00000030  48 6f 73 74 3a 20 31 30  2e 31 30 2e 31 30 2e 31   Host: 10 .10.10.1
00000040  0d 0a 43 6f 6e 6e 65 63  74 69 6f 6e 3a 20 4b 65   ..Connec tion: Ke
00000050  65 70 2d 41 6c 69 76 65  0d 0a 0d 0a               ep-Alive ....
```
TCP Stream – Hex Dump

As evidenced in the request shown above, the User-Agent header holds the value of 2 CRLF characters. This is simply not correct in the context of this request, as per the standard specification of the message format. Said request is to be interpreted as a GET request with the only header being User-Agent with an empty value, and with a body payload consisting of all the bytes highlighted in gray. Even Wireshark HTTP decoder errs on this traffic with the message “*Expert Info (Error/Malformed): Leading CRLF previous message in the stream may have extra CRLF*”. Technically speaking, above request is invalid and the server should reject it, since the Host header wouldn’t be interpreted as part of the request headers. Note that at the time of testing the sample, the server didn’t respond with anything!

Surprisingly, such error is committed at the code level as demonstrated in the following code snippet:

```
004084D4       push    0                    ; dwFlags
004084D6       push    0                    ; pszProxyBypassW
004084D8       push    0                    ; pszProxyW
004084DA       push    0                    ; dwAccessType
004084DC       push    offset pszAgentW     ; "\r\n\r\n"
004084E1       call    eax ; WinHttpOpen
```

The malware uses the Windows HTTP Services (WinHTTP) library for sending above request. To initialize the use of WinHTTP functions and return WinHTTP-session handle, it calls the function *WinHttpOpen*, which takes as the first argument the name of the application calling the WinHTTP functions, that is the User-Agent header field value in the HTTP protocol parlance.

Generalizing detection for this request is not easy, other then checking for a request without a Host header, and an empty User-Agent.

OTHER CASES AND DISCUSSION

Other similar cases include but not limited to, exfiltrating data in the user-agent header field, typographical errors in the construction of the HTTP header-field name and value, and occurrence of special characters that are usually not present in benign traffic, among others.

For example, mining the same set of malicious pcaps used in case 7, searching for the special character '+' in the User-Agent header-field value, yields 15 malware families, however, one of the user-agents is not malicious; the Bestafera family uses the Googolebot (Desktop) user-agent value "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)". Most of the results consist of legitimate User-Agent values where spaces are replaced with '+', or data for data exfiltration (for example, sending base64 encoded data, or used as a separator for grouping infected system identifying information, …).

Searching for the character '&' exposes the old malware family known as Alureon, where the malware sends the HTTP URI parameters in the User-Agent field (ex., User-Agent: id=5247819786&smtp=ok&ver=102). Similarly, SecureWorks Counter Threat Unit Research Team details a similar case of DanBot committing a typographical error with the character '&' being present somewhere in the user-agent.

Searching for the character '=" reveals another 8 unique malware families (disjoint from the above 2 checks). 7 of them include the malware exfiltrating data through the User-Agent and one where the malware author making a blatant mistake in constructing the header-field value (User-Agent: User-Agent=Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.0.1) Gecko/2008070208 Firefox/3.0.1), and the same mistake, that is repeating the header field name followed by an equal sign is repeated in most of the headers in the request.

Searching for the character '\'unravels another 7 unique malware families.

Scanning HTTP headers for special or not commonly used characters, while taken into their proper context, will undoubtedly reveals strange traffic. Moreover, scanning a given header field-value for other than predefined values, might also uncover suspicious traffic. For example, searching the Content-Disposition header for values other than the expected values: inline, attachment, form-data, name or filename, a variant of the malware family Nalodew is revealed, whereby it sent a POST request similar to the following:

```
POST /cm.php?do=3&cf=1|681957093953-1712835101-2-3116a03e| HTTP/1.1
Accept: */*
Content-Disposition: ###01504290527-0-2(v30000)
User-Agent: 30000!0*:---:<computer_name>-<user_name>:---:LAN Connection:---:www:---:NEW
Host: example.com
Content-Length: 3555
Connection: Keep-Alive
Cache-Control: no-cache

// payload is omitted for readability purpose
```

Notice how the malware sends infected system information via the User-Agent header field-value.

And the most anticipated corrective mistake in the request headers is the Referer one, whereby the header-field name is written not misspelled, as in "Referrer". Note that the correct header name is the misspelled version, that is "Referer". Interested reader could refer to this Wikipedia HTTP Referer page for more information on the etymology of the header name. The malware family known as Ultralocker sends a request similar to the following:

```
POST / HTTP/1.1
User-Agent: agent
Referrer: http://www.yahoo.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 51
Accept: */*
Host: 10.10.10.1
Connection: Keep-Alive

pcname=<computer_name>&hwid=<id_value>&version=Locker
```

This is not too surprising, since only 8 out of 2769 malware families have the Referer header field in their requests.

Another easy method for detecting doubtful HTTP requests is by checking for the case-sensitivity of the known header-field names. Although, per specification, the header-field name is case insensitive, legitimate traffic tend to follow proper casing on the headers, and malware tend to deviate from such norm. The following table shows number of malware families that fall into such category with respect to different headers, albeit not mutually exclusive.

Header field-name (lowercase)	**#**	
pragam	6	
cookie	1	
user-agent	2	one of the families used the library libsfml to generate network traffic, which has the header, all lower case, with the hardcoded value "libsfml-network/2.x".
connection	3	
cache-control	9	
host	2	
content-length	6	
content-type	6	
accept-encoding	3	
accept	0	

content-disposition	10	one family was found sending control commands via the content-disposition request and response headers.
proxy-connection	0	

In addition to checking the headers, we could also check the case sensitivity of the HTTP request's method, for example, checking for "get" (lowercase), reveals 2 unique malware families, whereas checking for "post" (lowercase), reveals nothing.

Another interesting finding is when scanning the request version string "http/1.[01]" (lowercase), which to our surprise detected the malware family know as Bluesummer, which sends a request similar to the following:

```
GET /index.html http/1.1
Accept:*/*
Accept-Language: hs-uk
Accept-Encoding:gzip ,deflate
Proxy-Connection: Keep-Alive
Pragma:no-cache

// this GET request has a payload
// payload is omitted for readability purpose
```

There are other anomalies as well in the above request, and in particular, the Accept-Language header-field value "hs-uk", which is not a standard one. Additionally, notice the zero-spacing between the separator ':' and the header value; while not an indicator of a malicious traffic on its own, it nonetheless calls for attention and investigation compared to normal network traffic.

To revisit empty header(s) and other anomalies, the following case highlights and documents yet another malware known as Beerish that committed such errors, but this time the malware is used in a limited targeted attacks, delivered via the exploitation of the vulnerability CVE-2019-1367:

```
GET /ixx/u3/scrobi.32 HTTP/1.1
Accept: */*
Accept-Language: en
Accept-Encoding:
User-Agent: IE7
Host:www.example.com
```

A case in point, where the traffic is not detectable from an IDS/IPS point of view without high chance of FP or potential performance impact, if it wasn't for the unintended mistake committed by the malware author when setting the value of the Accept header. This is the case of Vietmefond malware, that sends a POST request similar to the following:

```
POST /login.php HTTP/1.1
Accept: text/html,application/xhtml+xm.plication/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:60.0) Gecko/20100101 Firefox/60.0
Host: 10.10.10.1:8080
Content-Length: 32
Cache-Control: no-cache

344431B6136D10C2BEC65491025A1EB7
```

The mistake is highlighted in dark orange, and the correct value should have been “xml,application”. The 32-byte string in the body of the POST request represents a UUID generated by the call to the API *CoCreateGuid*; thus, it is dynamic and changes per request. The rest of the request headers and URL are benign, and are not unique for detection.

There is also the case of HTTP request smuggling attacks that date back to 2005 as documented by Watchfire, and resurfaced again in 2019 with new attack vectors and techniques by James Kettle from PortSwigger. The idea is the same in which the RFC specification is not fully honored by every implementation.

Checking for data type violation that a given header-field value accepts according to RFC specification could also unravel, in a non-specific way, suspicious traffic. For example, in the case of the FakeUpdates malware, it sends a POST request with the Age header that should take an integral type of type unsigned integer, but actually takes a dynamically generated hexadecimal value that doesn’t represent the intended meaning; For example, it sends “Age: 5496a0f8927fbe57”. Other than the fact that it’s a violation to use this header in the HTTP request, the value it takes in this context is meant to function as an authentication mechanism with the C&C server, and not as a representation of the time (in seconds) requested object has been in the cache of a proxy.

CONCLUSION

The HTTP protocol is a text-based protocol, with a clearly defined syntax and semantics, extensible and generic. It is the fundamental protocol of the web. For a client (user agent) to blend with the traffic of the web, it needs to use the HTTP protocol, even if its not actually using the protocol. And, this is what malware do, they use the HTTP protocol for their C&C communications in an attempt to merge with benign network traffic.

As demonstrated in this paper, we have uncovered different malware families that have committed multiple errors in the construction of the HTTP request and response headers, be it intended or inadvertently. Those types of mistakes help in identifying and blocking malware C&C communications via a heuristic or specific IDS/IPS signatures.

While not all of the documented errors are deviation from the HTTP protocol specification, they nonetheless stand out from typical network traffic. Threat hunters looking for anomalous or suspicious traffic on the network, should look into these cases with the suggested detection guideline, and adapt them to their network, post-baselining, so that false-positive hits are either reduced or fully accounted for.

APPENDIX A: LIST OF MALWARE HASHES

Malware Family	Hash (MD5)
PCRat	4D05BB6E2628E1DA41A48B1715D89F4C
PKEY/Adelinoq	03398D02094EE36859C5E3880114BB77
Reverse Gear	580DEDCF4153C45BA351407D4FD015DD
Volgmer	53098C29B748E881E4D62720D7190AC5
ProtonBot	1AF50F81E46C8E8D49C44CB2765DD71A
Crosswalk	9713333328D201A65628B87B5839EAA9
Socnet	48B9C63D4F54E7EF4136AB9C8C1735CE
Cobra	C94F9A225392619EEC6AE6F0B31CF572

Aytoke	EDEB022852F00F652627AC04E24B0B50
Sefnit	2CC72E53AADDFF52DF5F267C255932DA
NooBot	4943A255952E107FEC41E9C29A5B2724
DanBot	9DF776B9933FBF95E3D462E04729D074
Nalodew	ED4BA7357816C044F6E5149419947AC3
Darkcpn	679C4AD55EF2A44EFB0DFDD90B35F0B1
Bluesummer	0286E6CA55801F197DD6C2B156DA06A5
Ultralocker	B333265CFFC0BC7BD502C5966D40D7E4
Silence/TrueBot	1B17531E00CFC7851D9D1400B9DB7323 B43F65492F2F374C86998BD8ED39BFDD CFFFC5A0E5BDC87AB11B75EC8A6715A4
FakeUpdates	BDFDB72D257966E7A954FB25E2070B51 1653A0463B843CABD2DCBFC758A238A4
Klokla	CB96BC9B199C35A593577AABCEA1CADF
Beerish	6F324F8D3D8B9C7A80B40944FB82167A
Vietmefond	A9B5C9FF8B13115A3CCBD8AF5655D882