Improving Network Monitoring Through Aggregation of HTTP/1.1 Dialogs in IPFIX

Felix Erlacher*, Wolfgang Estgfaeller*, and Falko Dressler†
*Department of Computer Science, University of Innsbruck, Austria
†Department of Computer Science, Paderborn University, Germany
{erlacher,wolfgang.estgfaeller,dressler}@ccs-labs.org

***Abstract*—Network monitoring is the basis to analyze communications for preventing network attacks and misuse. Common practice for today's high throughput networks are flow-based network monitoring techniques like IPFIX. However, these solutions do not support the observation of HTTP/1.1 request/response dialogs. At the same time, we observe an increasing trend to use HTTP pipelining in the Internet as well as the use of HTTP as a generic transport protocol. IPFIX-based network monitoring and attack detection is no longer able to identify misbehavior in the complex and often interlaced HTTP dialogs. This work presents a new monitoring concept, which is able to aggregate HTTP/1.1 dialog information into bidirectional IPFIX flows. It is thereby laying the foundation for network security tools like IDS to process the resulting flows faster and more efficiently. The evaluation shows that the implemented parsing mechanism can even deal with complex HTTP traffic and clearly outperforms existing solutions.**

I. Introduction

The observation and analysis of network communication is indispensable for every operator. It provides valuable information about the traffic and the overall state of the network and helps to detect and prevent network attacks and misuse [1]. To serve this purpose, most network monitoring tools analyze traffic up to the transport layer and possibly apply computational expensive Deep Packet Inspection (DPI) [2] operations to the payload. Recent studies show, that the amount of Hypertext Transfer Protocol (HTTP) in Internet traffic has increased significantly in the last years, leading to HTTP being the most dominant protocol, accounting for more than $50\,\%$ of the overall traffic volume [3], [4]. There are mainly two reasons for this: first, HTTP was adopted by a variety of applications to serve as a 'transport protocol' [5], and second HTTP is used by these and other applications to transport high-volume content.

As a consequence of this forced 'push' of HTTP from the application layer down to the transport layer, it is required to treat it similarly as other transport layer protocols (e.g., Transmission Control Protocol (TCP)) to further analyze the carried HTTP payload. If for example, a firewall wants to categorize web application traffic, it applies DPI methods to search through the HTTP header and payload and possibly find matching patterns. This is a resource intensive task. This work simplifies this by providing HTTP elements in the form of IPFIX flow Information Elements (IEs) which allow for quick and easy analysis. For the case of encrypted HTTP (https) we propose to use TLS interception proxies like most big firewall manufactures do (e.g. Genuas GenuGate[1]).

Flow based monitoring strategies are the state of the art solution to approach today's high transmission speeds. The Internet Protocol Flow Information Export (IPFIX) protocol [6] is a widespread standard, that is used to aggregate packet information of a flow (a sequence of packets sharing the same properties) into a single structure (flow record) that can be exported. But the IPFIX protocol does not foresee the aggregation and export of HTTP information, and thus is extended in this work to include it. Existing solutions, which are able to add some HTTP information to IPFIX, show various problems: performance issues (slow DPI solutions), inability of aggregating payload, HTTP parsing problems and unclear separation of the HTTP information in the exported data. Another factor to consider is the dialog-based nature of HTTP, which is particularly relevant for HTTP/1.1 (and later protocol versions).

Within this work, a new approach of monitoring HTTP is designed and implemented that provides a solution to the above issues. A mechanism is introduced that enables the aggregation of HTTP dialogs (request and response messages, which belong to each other) into bidirectional flow (biflow) records using the IPFIX protocol. Our approach has been implemented in the network monitoring toolkit Vermont [7]. Throughout this paper, the term $Vermont_{HTTP}$ is used to denote the enhanced implementation of Vermont.

We believe that our work provides a foundation for other network monitoring tools, allowing them to efficiently process the resulting IPFIX flows containing HTTP information.

The contributions of this work are the following:

- We present a methodology for aggregating HTTP/1.1 dialogs into IPFIX flows and reducing the exported HTTP dialogs to the first N bytes per direction;
- we performed an in-depth evaluation and functional comparison to related tools;
- we made our implementation freely available as Open Source[2] running on standard Linux systems.

II. Related Work

In the scope of this work, we are mainly interested in the analysis of HTTP traffic. A straightforward approach is to use

[1]https://www.genua.de/loesungen/high-resistance-firewall-genugate.html
[2]https://github.com/felixe/ccsVermont, branch: http-aggregation

DOI 10.1109/LCN.2016.88

Regular Expressions (RegExes) to identify application layer protocols out of the payload (e.g. L7-filter [8]). It is clear that this approach is computational expensive [9] and, thus, not suited for continuous monitoring in high speed networks.

There are several tools available that make it possible to investigate HTTP. The packet analyzer *Wireshark* [10] for example, includes an HTTP dissector. While it is perfectly suited to analyze a particular network trace, it is not intended for continuously monitoring and exporting network traffic at all. The same is valid for the well known “network security monitor” and highly adaptable tool *Bro* [11].

Usually, there is a semantic correlation between subsequent traffic flows in bidirectional communication (in HTTP a request is followed by the corresponding response). This has been exploited in the Dialog-based Payload Aggregation (DPA) approach [12]: by collecting the first N (in this case $N = 2\,\mathrm{kB}$) bytes after every direction change the authors were able to reduce the exported IPFIX traffic to $3.7\,\%$ while retaining $89\,\%$ of all reported events by the Intrusion Detection System (IDS) Snort.

To the best of our knowledge the only network monitoring tools that have a similar scope and functionality than ours are nProbe [13] and YAF [14]. Both support live capturing or input from `libpcap` files, and offer the possibility to export HTTP related attributes using IPFIX. For nProbe, the exported IPFIX templates can be configured dynamically and support a few HTTP IEs if the HTTP plug-in is used.

YAF has been build as a reference implementation for IPFIX. The DPI plugin checks the payload against a list of RegExes. Thus it is not aware of the structure of HTTP messages and applies the search also on the HTTP body, which is very expensive from a performance point of view and might lead to false positives if the body carries a matching string. While nProbe is not able to export biflows, YAF exports them aggregating all messages of a TCP flow using one IPFIX flow, thus, failing for advanced HTTP features like pipelining.

III. Architecture

For the implementation of this work the monitoring toolkit Vermont [7] has been extended. As input Vermont accepts raw packets as well as IPFIX flows. Export of data is supported through IPFIX, Packet Sampling (PSAMP), or Intrusion Detection Message Exchange Format (IDMEF). The functionality of Vermont is divided among many different modules that can run in parallel, each having its own functionality. One of these modules is the *PacketAggregator* module which aggregates packets to IPFIX flows. For this work we extended this module by adding TCP reassembly functionalities and HTTP protocol dissection capabilities. Figure 1 outlines the work flow: Packets entering the *PacketAggregator* module are passed to the TCP reassembly engine. Than the TCP payload of the packets gets analyzed by the HTTP parser, which parses HTTP dialogs in the passed sequence of bytes. Aggregation of flow information happens in all modules by annotating the flow accordingly: First, information gathered during TCP reassembly is aggregated in the TCP reassembly engine.

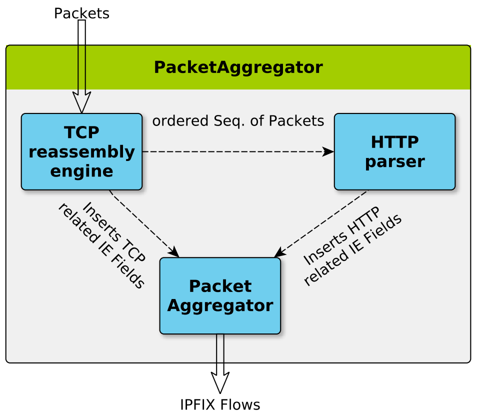

Fig. 1. Basic workflow in the novel PacketAggregator module in Vermont$_{\text{HTTP}}$

Second, HTTP information and payload gets aggregated by the HTTP parser and lastly, other standard flow information fields are aggregated in the *PacketAggregator*.

For this work, we developed what we call a stateful on-the-fly HTTP parser. Because of the unknown length of HTTP messages, we are not buffering the entire HTTP message until completion but work on the payload of single TCP segments. The parser processes as much of the segment’s payload as possible and, if necessary, buffers the rest combining it with the next segment’s payload as soon as possible. The first parsing step is to check the message type (request/response). The second step is to extract the most important header field information, and aggregate them into the respective IPFIX flow using so called enterprise specific IEs. Depending on the configuration, the parsing of the HTTP header can be skipped and the parser will proceed with the next step. Now, a configurable amount of the message body data is exported into the respective IPFIX IE. When all the message body has been processed, the HTTP parser can continue with the next segment. If the message is finished and the type is ‘response’, it will be combined with the former request to an HTTP dialog and exported into an IPFIX flow. Similar to the aforementioned Dialog-based Payload Aggregation (DPA) approach, Vermont$_{\text{HTTP}}$ allows to export the first N bytes of both directions of an HTTP dialog. We implemented also some IEs to annotate issues encountered during parsing.

IV. Evaluation

The evaluation of Vermont$_{\text{HTTP}}$ concentrates mainly on functionality and performance. All the traffic traces for the tests where either previously captured and are published at http://www.ccs-labs.org/~erlacher/resources/, or obtained from public sources to make them reproducible. We focused on having a complete collection of diverse traces striving to have a representative set of realistic traffic scenarios but also including possible rare cases.

1) Functional Validation: To test Vermont$_{\text{HTTP}}$ for correctness, we first manually assessed all traces and then compared

TABLE I
NUMBER OF HTTP MESSAGES (REQUESTS AND RESPONSES) DETECTED BY THE DIFFERENT TOOLS, WRONG NUMBERS ARE INDICATED IN BOLD, MD = MATCHED DIALOGS

Tool \ Trace		riverbet-two	cnn2012	pipelining	anomalous
Correct	Req.	94	146	81	3966
Value	Res.	94	146	81	3966
Bro	Req.	**54**	146	**8**	3966
	Res.	**54**	146	**8**	**3941**
Wireshark	Req.	94	146	**85**	**3816**
	Res.	**96**	**149**	81	**2907**
nProbe	Req.	94	**145**	**20**	**3965**
	Res.	94	**145**	**19**	**3875**
YAF	Req.	**14**	**131**	**4**	3966
	Res.	**9**	**145**	**0**	**4157**
Vermont$_{\text{HTTP}}$	Req.	94	146	81	3966
	Res.	94	146	81	**3903**
	MD	94	146	**79**	**3102**

the flow results to the state of the art tools Wireshark (version 1.12), Bro (2.2), nProbe (7.1), and YAF (2.7.1).

For Wireshark, we used the *Conversations* and the *HTTP-Load Distribution* function. A small script in Bro's own Domain Specific Language (DSL) counts events generated during traffic processing to compute the number of HTTP messages. To ensure a better comparability we increased all TCP timeout values of Bro from 5 s to 20 s (the value we also used for Vermont$_{\text{HTTP}}$). Since nProbe creates two separate flows when bi-directional export is turned on, the flows were counted using the unique flow identifiers contained in the output. To make YAF export HTTP information, *application label* support and the enclosed HTTP plugin were turned on.

For the sake of brevity only an exemplary fraction of the tested traffic traces is shown. But all other tests showed a similar behavior.

Table I shows the detected number of HTTP messages. The first two traces are freely available and taken from [10]. *riverbed-two* is a repeated visit to the www.riverbed.com site, lacking typical DNS queries and making heavy use of the browser's cache. *cnn2012* visits www.cnn.com and contains TCP keepalive packets and retransmissions. The third trace contains traffic where both server and client support and use HTTP pipelining. The last trace contains anomalous requests to test the parser's robustness. One peculiarity is that all dialogs have a valid structure but some headers contain odd and unusual values. Some length related header fields contain wrong values meaning that some messages can be interpreted as not ending regularly.

The reasons for the different outcome are the following. Only Vermont$_{\text{HTTP}}$ and Wireshark wait for HTTP messages to end before counting them. The other tools already increase the HTTP message counter as soon as the start of a message is detected. Bro's HTTP analyzer only seems to work correctly if the complete TCP handshake has been observed, this is the main reason for the missed HTTP messages. On the other hand, Bro has the least problems with the anomalous trace.

In Wireshark if HTTP messages can not be parsed completely they are not counted, but the description indicates "Continuation or non-HTTP traffic". If a trace contains retransmitted TCP segments, Wireshark counts the corresponding HTTP messages twice (or more). These is more a design decisions then an errors. As the numbers reveal, Wireshark seems to have quite some problems with the anomalous trace.

nProbe does not recognize HTTP requests if the URI is very long. Also, two POST requests that were split over two messages are not detected, the same applies to some HTTP requests that were split over multiple TCP segments. Again, nProbe too counts unfinished HTTP responses as full messages. On traces with pipelined requests nProbe did not export the right values, this suggest the lack of pipelining support.

YAF's wrong numbers are the consequence of its DPI/RegEx approach: E.g., YAF uses a vague RegEx to detect HTTP responses, because it only matches against the three digit response code followed by text. This, for example, matches also against the common message body "404 Not Found". YAF lists a value of 4157 messages (instead of 3966) on the anomalous trace, meaning a lot of false positives were exported. Here again, the suspicion is that, as above, the RegEx approach is to blame.

Vermont$_{\text{HTTP}}$ detects almost all messages correctly, and is the only tool able to also annotate partial HTTP messages. The only case when the message count differs (not shown) is when an IPFIX flow timeout occurred, and thus one HTTP message is counted twice. This only suggests to configure the IPFIX flow timeout values according to the expected traffic. When inspecting the exported IPFIX flows, the request and response pairs where matched correctly for 99.7 % and thus exported as a dialog in one IPFIX flow. It only failed in the rare cases where the parsing resulted in partial messages or a timeout occurred. This underlines the necessity for a robust and highly operational parsing engine. On the anomalous trace Vermont$_{\text{HTTP}}$ does very well. There are only minor issues with the wrong message length information: In case of a greater number than the actual size it counts these messages as partial requests, expecting more data, which can be assumed as correct behavior. As all the other tools do not seem to maintain any HTTP message state over multiple segments, Vermont$_{\text{HTTP}}$ is also the only tool in this test handling HTTP messages correctly if the header is bigger than the Maximum Transmission Unit (MTU).

2) Performance Tests: We also carried out a number of performance tests. The used test environment consists of two PCs (Linux 3.13, i7-3920k CPU with 6 cores and 32 GB RAM), directly connected with a 10 Gbit/s link (Intel 82599ES Network Interface Controller (NIC)). All the tests are performed by replaying a network trace at different speeds at host A and using Vermont$_{\text{HTTP}}$ at host B to capture the network traffic. To be able to capture packets at high line rates, we used the `libzero PF_RING` library in combination with Vermont$_{\text{HTTP}}$. We used the same configuration of Vermont$_{\text{HTTP}}$ as before, and exported 2 kB of payload in each IPFIX flow. It should be noted that more payload has only a marginal impact on performance.

TABLE II
GENERAL PERFORMANCE STATISTICS OF $\text{Vermont}_{\text{HTTP}}$

Trace ID	Pkt. Rate [k/s]	Through-put [Gbit/s]	Avg. Pkt. Size [B]	Avg. HTTP Header [B]	TCP Conn. [#]	Avg. HTTP Req. [$\times 10^3$/s]	Avg. HTTP Buffer [kB/s]
1	810	4,13	675	475	17762	**29.78**	1540
2	790	2,50	375	448	1167631	**42.16**	481
3	730	4,41	732	488	8501	**32.03**	1
4	700	6,65	1161	**1880**	63	14.89	**27310**

The traces used are all considerably bigger than the traces used in the functionality tests (avg. number of packets here is 12 Mio). We tried to create different traces covering all possible traffic characteristics. To put the system under maximum load all traces consisted solely of HTTP traffic, all other traffic was filtered out. Again, only an exemplary fraction of the tested traces is discussed here.

To cover typical Internet traffic, we created a trace by capturing the traffic going through an HTTP proxy used by a scientific work group for one week. We than removed messages with a size bigger than $5\,\mathrm{MB}$ resulting in trace 1. To create traces with more or less equal packet sizes we used a crawler and limited the MTU. In trace 2 and 3 the MTU was limited to $1000\mathrm{B}$. The difference is the number of TCP connections. For trace 4 the MTU was set to $1500\mathrm{B}$, the peculiarity of this trace is that the crawler requested Universal Resource Identifiers (URIs) for possibly indefinitely deep directory structures ending up with extremely long message headers. Apart from that only very few different TCP connections occur in this trace.

Table II shows the experiment results. It is sorted by the packet rate and depicts the performance data of $\text{Vermont}_{\text{HTTP}}$ at the highest rate possible without packet drops (or drop rates in the per mill range). The CPU usage is with all traces between 90% and 94%. The first observation is that the overall high packet rates and throughput are only affected negatively if the trace contained a high number of HTTP requests per second (trace 2, 3), or if the average HTTP header size is quite large (trace 4). A high frequency of HTTP requests requires more work by the HTTP parser, but also to open more buckets for aggregation, which is a resource intensive task. A long header implies more effort by the HTTP parser, as more bytes have to be processed (and buffered) before aggregation. Most critical is that headers are split over multiple TCP segments, which is the case for trace 4. The impact on memory consumption is negligible as it correlates directly with the number of packets currently processed. The CPU performance is the bottleneck and packets are dropped long before memory fills up.

V. Conclusion

This paper introduces a new mechanism of monitoring HTTP/1.1 traffic. Our monitoring concept treats HTTP as a transport protocol and provides the aggregation of HTTP dialogs (request and response messages which belong to each other) into bidirectional flow records, and makes export to other network monitoring entities possible using the IPFIX protocol. The implementation allows the aggregation and export of HTTP header fields and the first *N* bytes of HTTP (or TCP) payload of both messages of a dialog, into a set of custom, so called enterprise specific IPFIX IEs. Network analysis tools like IDSs can than process the resulting flows faster and more efficiently and thus prevent network attacks and misuse with a new quality.

The evaluation has shown that the detection rate of HTTP messages of the introduced HTTP parser outperforms all other network monitoring tools in the test. We showed that $\text{Vermont}_{\text{HTTP}}$ at its limits can deal with packet rates of $800\,\mathrm{kpps}$ and with line rates of $6\,\mathrm{Gbit/s}$ of pure HTTP traffic.

The bottleneck of the implementation is the high CPU load of the new PacketAggregator module induced by the HTTP parser/aggregator and TCP reassembly engine, which were incorporated into this module. A possible remedy to improve the performance of $\text{Vermont}_{\text{HTTP}}$ could be to implement the TCP reassembly engine and HTTP parser in two separate modules, allowing them to run as two single threads.

References

[1] S. Lee, K. Levanti, and H. S. Kim, "Network monitoring: Present and future," *Elsevier Computer Networks*, vol. 65, pp. 84–98, 2014.

[2] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. Gó-Dor, G. Szabó, and T. Westholm, "Deep Packet Inspection Tools and Techniques in Commodity Platforms: Challenges and Trends," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1863–1878, 2012.

[3] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet Inter-Domain Traffic," in *ACM SIGCOMM 2010*. New Delhi, India: ACM, Aug. 2010, pp. 75–86.

[4] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger, "Anatomy of a Large European IXP," in *ACM SIGCOMM 2012*. Helsinki, Finland: ACM, Aug. 2012, pp. 163–174.

[5] W. Li, A. Moore, and M. Canini, "Classifying HTTP Traffic in the New Age," in *ACM SIGCOMM 2008*, Seattle, WA, Aug. 2008.

[6] B. Trammell and E. Boschi, "An Introduction to IP Flow Information Export (IPFIX)," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 89–95, Apr. 2011.

[7] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont - A Versatile Monitoring Toolkit for IPFIX and PSAMP," in *IEEE/IST MonAM 2006*. Tübingen, Germany: IEEE, Sep. 2006, pp. 62–65.

[8] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding, "A Scalable Multithreaded L7-filter Design for Multi-core Servers," in *ACM/IEEE ANCS 2008*. San Jose, CA: ACM, Nov. 2008, pp. 60–68.

[9] K. Namjoshi and G. Narlikar, "Robust and Fast Pattern Matching for Intrusion Detection," in *IEEE INFOCOM 2010*. San Diego, CA: IEEE, Mar. 2010.

[10] L. Chappell, *Wireshark Network Analysis The Official Wireshark Network Analyst Study Guide*. Laura Chappell University, Mar. 2010.

[11] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Elsevier Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999.

[12] T. Limmer and F. Dressler, "Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation," in *IEEE INFOCOM 2011, GI Workshop*. Shanghai, China: IEEE, Apr. 2011, pp. 833–838.

[13] L. Deri, "nProbe: an Open Source NetFlow Probe for Gigabit Networks," in *TNC 2003*, Zagreb, Croatia, May 2003.

[14] C. Inacio and B. Trammell, "YAF: Yet Another Flowmeter," in *USENIX LISA 2010*, San Jose, CA, Nov. 2010.