Threshold Based Kernel Level HTTP Filter (TBHF) for DDoS Mitigation

Mohamed Ibrahim AK and Lijo George
Trichy Engineering College, Tiruchirappalli, Tamil Nadu, India
{ibrahim.awn, lijogeorgeemail}@gmail.com

Kritika Govind and S. Selvakumar
National Institute of Technology, Tiruchirappalli, Tamil Nadu, India
{kritika, ssk}@nitt.edu

Abstract — HTTP flooding attack has a unique feature of interrupting application level services rather than depleting the network resources as in any other flooding attacks. Bombarding of HTTP GET requests to a target results in Denial of Service (DoS) of the web server. Usage of shortened Uniform Resource Locator (URL) is one of the best ways to unknowingly trap users for their participation in HTTP GET flooding attack. The existing solutions for HTTP attacks are based on browser level cache maintenance, Captcha technique, and usage of Access Control Lists (ACL). Such techniques fail to prevent dynamic URL based HTTP attacks. To come up with a solution for the prevention of such kind of HTTP flooding attack, a real-time HTTP GET flooding attack was generated using d0z-me, a malicious URL shortener tool. When user clicked the shortened URL, it was found that the user intended web page was displayed in the web browser. But simultaneously, an avalanche of HTTP GET requests were generated at the backdrop to the web server based on the scripts downloaded from the attacker. Since HTTP GET request traffic are part of any genuine internet traffic, it becomes difficult for the firewall to detect such kind of attacks. This motivated us to propose a Threshold Based Kernel Level HTTP Filter (TBHF), which would prevent internet users from taking part in such kind of Distributed Denial of Service (DDoS) attacks unknowingly. Windows Filtering Platform (WFP), which is an Application Programming Interface (API), was used to develop TBHF. The proposed solution was tested by installing TBHF on a victim machine and generating the DDoS attack. It was observed that the TBHF completely prevented the user from participating in DDoS attack by filtering out the malicious HTTP GET requests while allowing other genuine HTTP GET requests generated from that system.

Index Terms — HTTP GET, Shortened URL, Kernel, Windows Filtering Platform

I. INTRODUCTION

Distributed Denial of Service (DDoS) attack uses a large number of computers to cause a coordinated DoS attack against a targeted web server. Flooding attack is usually caused by continuously sending multiple requests to bring down the targeted server. This eventually fills the server’s buffer space. Once the buffer memory is full no further connections can be made to the server, thus making the service unavailable. This scenario is attained by misusing the URL shortening service which eventually leads to the formation of HTTP botnet. URL shortening service is a technique, used in the World Wide Web (WWW) to substantially shorten a lengthy URL into a shorter one while redirecting the user to the requested webpage. This is achieved by referring HTTP redirect information from the available database of the URL shortener tool. This redirect information is required for redirecting the shortened URL to the requested original URL. Thus attacks could be launched using shortened URL. On making use of shortened URL service the user requested web page gets loaded in the web browser along with malicious scripts downloaded from the attacker. This is done by misusing the html iframe tag. Thus HTTP GET flooding attack is initiated in the background by using the client side scripts downloaded from the attacker without interrupting the user’s browser activity. The attack will continue as long as the web page is active and it will not leave any trace of code unless the cache of the browser is not cleared on the client side.

The rest of the paper is organized as follows: Related work is discussed in Section II. Section III discusses the Threshold Based Kernel Level HTTP Filter (TBHF) algorithm in detail. HTTP GET attack generation and analysis are explained in Section IV. Performance analysis of TBHF is discussed in Section V and finally, Section VI concludes the paper.

II. RELATED WORK

Client side prevention of browser initiated flooding attacks is mainly done by Client side Caching optimization and implementation of the Human Interaction Proof (HIP) protocol. In client side, a cache is maintained at browser level. All HTTP GET requests which are sent for the first time are cached in the local persistent storage. Subsequent requests to the same
URL are served from the local cache. However, this method fails when the attacker requests different URLs, thereby making the technique ineffective.

Another method is Completely Automatic Public Turing tests to tell Computers and Humans Apart (CAPTCHA) where the user is puzzled with an image, which contains randomly skewed alphabetic characters. This is used to distinguish between human and bot behavior. In the normal scenario when a host is making a large number of requests to a specific URL, the system detects an anomaly and redirects the user to a CAPTCHA page. If the user is able to complete the challenge, then the actual URL is served again. In the case of a bot requesting the URL, the CAPTCHA cannot be cracked, leading to inaccessibility of the URL. But in the current HTTP-GET attack scenario, the host requests different pages under the same server. Hence, no anomaly is raised and the CAPTCHA page is not served thereby surpassing the technique which leads to the flooding attack. Techniques such as blackholing are used by ISPs to mitigate DDoS attacks, where in traffic to a specific IP is blocked. The blocked traffic may also contain genuine packets thus making the service unavailable to the legitimate user. Access Control Lists (ACLs) are altered in order to stop users from participating in DDoS attack. But this technique becomes inefficient when attackers spoof genuine IP address.

III. PROPOSED WORK

A. Threshold Based Kernel Level HTTP Filter (TBHF)

The block diagram in Fig. 1, shows the different modules of the proposed TBHF, viz., Traffic capture, Parameter extractor, and the Analyzer module. First the packets are captured at kernel level by the traffic capture module. The output of the traffic capture module, the outbound TCP packets alone, are filtered and sent to the parameter extractor module which extracts the features such as remote IP address and the arrival time of packets. Then the packets are subjected to the Analyzer which decides whether to drop or allow the packets into the network based on the threshold set.

The analyzer module contains an IP frequency list to store the number of occurrences of individual IP address over a period of time. It checks the frequency of each IP address in IP frequency list and further decides whether to block or allow the packets.

The flow chart for the basic functioning of TBHF is given in Fig. 2. Threshold value as mentioned in section IV is set to restrict the number of HTTP requests to a particular IP address for a given period of time. Based on the parameters extracted from the packets ΔT values are calculated, which gives the time interval between current and previous instance of a packet for a particular IP.
Threshold Based Kernel Level HTTP Filter (TBHF) for DDoS Mitigation

Step 1:

\[ \text{Input: Network Traffic} \]

\[ \text{IF (Outbound packets)} \]
\[ \text{THEN} \]
\[ \text{IF (Packet == HTTP GET)} \]
\[ \text{THEN} \]

Step 2:

\[ \text{// Extract Parameters} \]
\[ \text{// IP1, IP2, ..., IPi - remote IP address} \]
\[ \text{// t1, t2, ..., ti - Arrival time of packets} \]
\[ \text{// IPAddrList - IP Address List} \]
\[ \text{IPAddrList[IPi][0]} = \text{IPi}; \]
\[ \text{IPAddrList[IPi][1]} = \text{ti}; \]

Step 3:

\[ \text{// \Delta T - Difference in time between two instance of same IP address} \]
\[ \text{// N - Threshold value} \]
\[ \text{// IPFreqList - IP Frequency List} \]
\[ \Delta T = (t2(\text{IPAddrList[IPi][1]})) - t1 (\text{IPAddrList[IPi][1]}); \]
\[ \text{IF (} \Delta T < 1 \text{ second)} \]
\[ \text{THEN} \]
\[ \text{IPFreqList[i]}++; \]
\[ \text{IF (IPFreqList [i] < N)} \]
\[ \text{THEN} \]
\[ \text{Allow packet to Network;} \]
\[ \text{ELSE} \]
\[ \text{Drop packet;} \]
\[ \text{END IF} \]
\[ \text{END IF} \]
\[ \text{ELSE} \]
\[ \text{Allow packet to Network;} \]
\[ \text{END IF} \]

END IF
IV. WORK DONE

A. HTTP GET attack generation

The Fig. 4 shows the log from Web Console tool in Mozilla Firefox on the Victim machine. Mark A shows the Link which is shortened using genuine URL shortener service provided by Google namely http://goo.gl. Mark B shows the Genuine URL, to which the user is redirected, to get the user requested web page.

Fig. 5 shows the log from Web Console using Mozilla Firefox during attack scenario. The Mark A in Fig. 5 shows the shortened URL that was provided by the attacker URL Shortening Service. In a genuine case when user clicks the link provided by shortened URL service, the user is redirected to the user requested web page but in case of an attack scenario, when a user clicks a malicious link provided by the attacker the user requested web page is displayed by exploiting the HTML iframe tag. Simultaneously GET requests are generated to victim server [9]. The Mark B shows the victim server IP address to which the HTTP GET Flooding attack has been initiated. The Mark C shows the dynamic resource name which is intended to overcome the browser level caching mechanism. Dynamic resource names are randomly generated unique IDs to access the resources from the web server.

WAMP server was set up in attacker side to host the malicious web page. On clicking the malicious link, the script to generate HTTP GET attack was downloaded from attacker web server to the victim browser. The script did not install any malicious software in the victim machine rather used the browser script engine to generate the malicious requests.

<table>
<thead>
<tr>
<th>Web browsers used</th>
<th>Total No. of Packets (for 30 second)</th>
<th>HTTP GET requests (for 30 second)</th>
<th>Average HTTP GET request (per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>32184</td>
<td>15891</td>
<td>529</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>59507</td>
<td>29708</td>
<td>990</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>87442</td>
<td>43654</td>
<td>1455</td>
</tr>
</tbody>
</table>

* Results shown for 30 second time interval during attack period
A sliding window protocol is a feature of packet-based data transmission protocols. Sliding window protocols are used where reliable in-order delivery of packets is required, such as in the Data Link Layer (OSI model) as well as in the Transmission Control Protocol (TCP).

Conceptually, each portion of the transmission (packets in most data link layers, but bytes in TCP) is assigned a unique consecutive sequence number, and the receiver uses the numbers to place received packets in the correct order, discarding duplicate packets and identifying missing ones. The problem with this is that there is no limit of the size of the sequence numbers that can be required.

By placing limits on the number of packets that can be transmitted or received at any given time, a sliding window protocol allows an unlimited number of packets to be communicated using fixed-size sequence numbers. The term "window" on transmitter side represents the logical boundary of the total number of packets yet to be
<table>
<thead>
<tr>
<th>Time</th>
<th>IP Address</th>
<th>Request URL</th>
<th>Status Code</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16:14:12.455</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>18ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.475</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>16ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.485</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>16ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.505</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>93ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.535</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>44ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.555</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>94ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.605</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.625</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.645</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.665</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.685</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.705</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.725</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
<tr>
<td>16:14:12.745</td>
<td>10.1.50.42/abcd</td>
<td>HTTP/1.1 200 OK</td>
<td>63ms</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5** Web Console log using Mozilla Firefox during attack scenario.
Threshold Based Kernel Level HTTP Filter (TBHF) for DDoS Mitigation

Figure 6: Statistics of HTTP GET attack before installing TBHF

Figure 7: Statistics of HTTP GET attack after installing TBHF

Figure 8: Suppression of HTTP GET attack traffic after TBHF installation

TABLE 2 – PERFORMANCE ANALYSIS

<table>
<thead>
<tr>
<th></th>
<th>OUTBOUND TRAFFIC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total No.</td>
<td>To target server</td>
</tr>
<tr>
<td></td>
<td>Total No.</td>
<td>HTTP GET PACKETS</td>
</tr>
<tr>
<td></td>
<td>HTTP GET PACKETS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Before</td>
<td>87442</td>
<td>43692</td>
</tr>
<tr>
<td>After</td>
<td>371</td>
<td>132</td>
</tr>
</tbody>
</table>

**Results shown for 30 second time interval before and after loading TBHF filter**
When observed using Wireshark, an enormous amount of HTTP GET requests were generated to the target IP address in a very short span of time which usually does not happen in a genuine case. Based on this observation, a threshold was set to ascertain the number of HTTP GET requests generated to a particular IP address in a certain time period.

V. PERFORMANCE ANALYSIS

A. Experimental setup

The effective working of the proposed system was tested by generating HTTP GET attack in Local Area Network (LAN) and using the TBHF driver for its mitigation. HTTP GET attack was generated using client side scripting in victim web browser. Traffic analysis was done with respect to HTTP packets using Wireshark. Based on the obtained results, the parameters to be set for filtering out malicious HTTP GET packets using TBHF were decided. The same was implemented and the performance of the TBHF was studied.

B. Detection module at victim machine for kernel level packet filtering

Sun virtual box with Windows 7 installed in it was made the victim. The developed TBHF driver was installed on the victim machine. A GUI called Open System Resources (OSR), was used to load or unload the packet filter in the victim machine. The loading or unloading of the filter can also be done manually which involves making registry changes and then starting the filter through command prompt.

C. Results

A real time HTTP GET attack was generated in the victim machine. Table 1 shows the web traffic that was generated using three different browsers, viz., Safari, Google Chrome, and Mozilla Firefox.

When observed using Wireshark, an enormous amount of HTTP GET requests were generated to the target IP address in a very short span of time which usually does not happen in a genuine case. Based on this observation, a threshold was set to ascertain the number of HTTP GET requests generated to a particular IP address in a certain time period.

Fig. 7, shows drastic drop in the http packet count between the victim and target server over the same period of time after installing the TBHF filter driver. Thus from Fig. 7, it is evident that the developed filter was able to mitigate the DDoS attack.

The performance analysis of the proposed TBHF was done by measuring the number of outbound packets in the victim machine before and after loading the filter. Table 2 shows the number of HTTP GET requests sent to the target server before and after the filter installation. It is evident from the 2nd row that the flooding packets generated to target server were dropped by TBHF. The accuracy in terms of mitigating the DDoS attacks by TBHF works out to be 99.82% which is calculated as the ratio of the number of HTTP GET attack packets blocked to the total number of HTTP GET attack packets generated (43575 / 43654 * 100).

Fig. 8, shows the decline in the attack traffic after driver installation. X-axis shows the time in seconds and Y-axis shows the number of HTTP GET packets. It is observed that after the driver installation there is a drastic drop in flow of attack traffic which signifies that the HTTP GET attack is mitigated.

VI. CONCLUSION

Application layer attack has become a major threat to the internet in today’s world. The focus of this paper was to come out with an effective solution for the detection and prevention of clients from inadvertently taking part in such attacks. Accordingly, a Threshold Based Kernel Level HTTP Filter (TBHF) was proposed and implemented in Windows 7 OS. Experiments were conducted by generating HTTP GET attacks and using TBHF for its mitigation. It was evident that the TBHF suppressed the flooding packets and thus prevented the client system from taking part in such an attack. The ongoing work is to implement TBHF in other OS and mobile platforms.

ACKNOWLEDGEMENT

The authors thank the National Technical Research Organization (NTRO), New Delhi, Government of India for sponsoring this research work under the Collaborative Directed Basic Research in Smart and Secure Environment project.

REFERENCES


**Mr Mohamed Ibrahim AK** has completed his B.E. (Computer Sci. and Eng.) at Trichy Engineering College, Anna University, Chennai, Tamil Nadu. He has a keen interest towards coding and his field of interest includes Network Security.

**Mr Lijo George** has completed his B.E. (Computer Sci. and Eng.) at Trichy Engineering College, Anna University, Chennai, Tamil Nadu. His interest includes coding for cyber security.

**Ms Kritika Govind** received her B.E. (Computer Sci. and Eng.) from Sakthi Mariamman Engineering College, Anna University, Chennai, Tamil Nadu, in 2009. She is working as a Research Assistant at Department of Computer Science and Engineering, National Institute Technology (NIT), Tiruchirappalli, Tamil Nadu. She is also pursuing her Master of Science (MS by Research) in Computer Science and Engineering at NIT, Tiruchirappalli. Her interests include: Cyber security and network security.

**S. Selvakumar** is a Professor in the Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli, Tamil Nadu, India. He received his Ph.D. from the Indian Institute of Technology Madras (IITM), Chennai in 1999. His research interests include group communication in high-speed networks, routing, multimedia communication, scheduling for QoS guarantee, mobile networks, network security, wireless sensor networks, and network computing. He has to his credit of publishing 54 research papers. He is currently the Investigator of the Collaborative Directed Basic Research–Smart and Secure Environment (CDBR-SSE) Project sponsored by NTRO, Government of India, New Delhi. He is presently the member of All India Board of IT Education, AICTE, New Delhi.