SeVen: A Robust Defense for Mitigating Low-Rate Application Layer DDoS Attacks

Omitted for Blind Review

***Abstract*—A new generation of sophisticated DDoS attacks are being carried out on the application layer as opposed to traditional Network/Transport Layer DDoS attacks, *e.g.*, TCP SYN-ACK attack. This brings new challenges to system administrators. Application Layer DDoS attacks (ADDoS) exploit protocols by simulating legitimate client traffic and can deny service without generating a great number of packets. This renders most existing defenses based on the volume of traffic ineffective in mitigating these attacks. Web-servers, such as Apache, are able to mitigate existing implementations of these attacks by using traffic analysis defenses, such as timeout-based defenses which drop requests that are not completed within some time frame. However, these defenses are not robust as simple modifications of attack profiles can render these defenses ineffective. We demonstrate this by proposing a new attack, called Resurrected Slowloris, and showing that it can bypass state-of-the-art time-out based defenses implemented in existing web-servers (Apache and nginx). Our second and main contribution is SeVen a selective defense for mitigating Low-Rate ADDoS attacks. SeVen works as a proxy and whenever the web-server is overloaded it selects randomly which requests will be processed. As it treats all requests equally, it is not tailored to mitigate a specific attack profile, but rather the whole class of Low-Rate Application Layer DDoS attacks. We demonstrate with a large number of experiments that SeVen is suitable for mitigating Slowloris, HTTP POST and Slowread attacks as well as the Resurrected Slowloris attack. Whenever under attack, SeVen guarantees service to most of legitimate client requests with little impact to the Time-to-Service, memory and CPU consumption.**

I. Introduction

Denial of Service Attacks (DDoS) have always been a major concern to network administrators. Traditionally, DDoS attacks are carried out on the transport layer by sending a number of packets to a server much greater than the server's processing capabilities making it unavailable to legitimate users. Although still dangerous, such attacks can be identified by using traffic analysis tools and mitigated by existing defenses [ZJT13], such as the Adaptive Selective Defense [KVF+08], [KVF+12].

In the recent years, a new generation of sophisticated attacks has emerged that exploit application layer protocols, such as HTTP and SIP protocols. Application Layer DDoS attacks (ADDoS) can target a single application in the server, *e.g.*, a web-server, instead of the whole server machine. The attack does not need to generate a huge amount of traffic, thus rendering existing defenses based on traffic volume ineffective [ZJT13]. Moreover, since the traffic generated by the attack follows the established protocols, it is hard to distinguish when a request is part of an attack or it is a legitimate request. These attacks are becoming increasingly preferred by attackers. In 2015, DDoS exploiting the HTTP protocol represented more than 30% of the total DDoS attacks just behind the traditional SYN-ACK DDoS with above 50% of all DDoS attacks[1].

We investigate the class of Low-Rate ADDoS attacks.[2] Low-Rate ADDoS attacks generate traffic in the same order as legitimate traffic and deny service by exploiting in clever ways the protocols used by the target application. The group "Anonymous" popularized these attacks by developing tools for carrying them out and by launching successful attacks. For example, in 2010, they carried out a number of attacks on organizations such as Mastercard.com, PayPal, and Visa.com [mas10].

Slowloris [slo13a], HTTP POST [rudy13] and Slowread [Slo13b] are examples of Low-Rate ADDoS attacks that target web-servers by exploiting the HTTP protocol. The main idea is not to overload the web-server with a large number of requests, but rather exhaust the web-server's resources by intentionally delaying the completion of a request. For example, Slowloris exploits the HTTP GET method by sending pieces of a GET request header, but never completing it. When such an incomplete request is received by the web-server, it allocates one of its worker and waits, as specified in the HTTP standard, for some time until the request header is completed. Service is denied when Slowloris sends enough such requests and leaving no available worker in the targeted web-server to serve legitimate clients.[3] To illustrate their power, a single computer is able to render in this way a medium size web-server unavailable. We explain these attacks in detail in Section II.

Our contribution is many-fold:

1) **Bypassing Existing Defenses (Section III):** Existing solutions for mitigating such attacks try to infer the profile of the attack by studying their behavior, *e.g.*, which messages are transmitted and in which intervals. *While they may be effective in mitigating particular instances of an attack, they*

[1] https://securelist.com/blog/research/70071/statistics-on-botnet-assisted-ddos-attacks-in-q1-2015/

[2] High-Rate ADDoS attacks, such as GET-Flooding attack, is also a major challenge to network security. These attacks send a large number of requests to a web-server, consuming the applications resources, but the traffic generated is still small when compared to Transport-Layer DDoS attacks.

[3] A medium size web-server has around 350 workers.

are not robust as simple attack modifications render these defenses ineffective.

We illustrate this with the current state-of-the-art which is implemented in web-servers, such as Apache and nginx, and are based on timeouts. Whenever a client is too slow to complete a request, that is, some timeout is elapsed, the request is dropped. Such timeout-based defenses can be used to mitigate existing implementations of the Low-Rate ADDoS mentioned above (Slowloris, HTTP POST and Slowread). However, as we show here, it is easy to bypass such timeout-based defenses. The attacker can discover such timeouts and carry out a novel attack which we call the *Resurrected Slowloris attack*. We implemented this attack and showed that both Apache and nginx are vulnerable to this attack when using its timeout-based defenses: under attack only less than 8% of clients are served by these web-servers. As Apache and nginx correspond to around 55% of all active web-servers, this means that, in principle, more than half of the active web-servers could be successfully attacked;

2) **SeVen:**[4] **A Selective Strategy Defense (Section IV):** We propose a novel defense called SeVen which works as a proxy and implements a selective strategy for mitigating Low-Rate ADDoS attacks including the Resurrected Slowloris Attack.

A selective strategy uses probability distributions to decide how likely a request is going to be served by the web-server. If the web-server's resources are not exhausted, *i.e.*, there are free workers, all requests are served. Otherwise, requests are randomly selected to be served or to be dropped by the web-server.

Here, we show that using a simple uniform distribution, *i.e.*, *all requests have the same probability of being selected and of being dropped*, can mitigate Low-Rate ADDoS attacks. By using an uniform distribution, it means that SeVen is not tailored to mitigate a particular attack making it much more robust to modifications of existing attacks.

SeVen is effective in mitigating a wide range of different attacks, such as Slowloris, HTTP POST and Slowread mentioned above. It works because it mitigates attacks which attempt to deny service by occupying the target web-server's resources by making them wait as it is the case of Low-Rate ADDoS attacks;

3) **Load Test of SeVen (Section V-A):** We evaluated SeVen with a large number of experiments. We tested SeVen using the benchmark/load test tool Siege and the against four different attacks Slowloris, HTTP POST, Slowread and our Resurrected Slowloris attacks. These tests provide us with a *lower-bound on the effectiveness* of SeVen. In most of the experiments, SeVen maintained high levels of availability over 95% and low TTS (less than 0.5s). These results are comparible with defenses tailored to mitigate specific attack instances [dA13] (described above). Our experiments also show that SeVen is lightweight as it has requires low CPU processing (less than 3%) and memory (less than 8 MB of memory).

4) **Quality of Service Test of SeVen (Section V-B):** We also implemented a more realistic scenario with robots simulating human interaction on a site with multiple linked pages with forms. We defined some quality measures, such as total number of times a robot presses the reload button or the time it waits for a web-server response, which reflect human experience. Whenever some quality measure fails, we considered that the robot failed to be served and successful otherwise. We constructed a more complex web-site with a sequence of forms to be filled, as in typical registration sites. We carried out a number of experiments using the Apache based web-server Tomcat. Our experiments demonstrated that when the web-server is running SeVen, almost all robots are successful to fill out all forms and register despite the web-sever being under attack.

[4]This name is a reference to the seventh layer of the ISO/OSI network model.

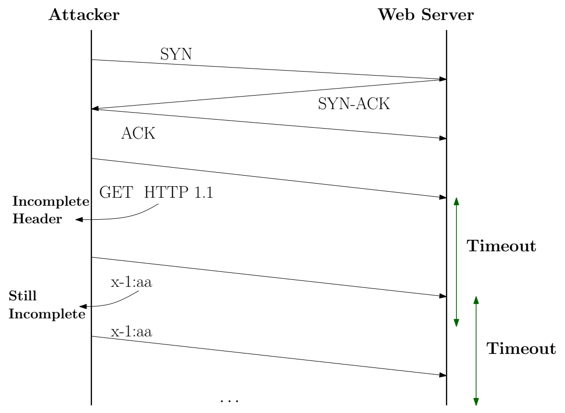


Fig. 1. Sequence of messages in the Slowloris Attack.

Finally, we conclude in Sections VI and VII by discussing related work and future work.

II. Attack Descriptions

We now briefly describe the main Low-Rate ADDoS attacks targeting web-servers, namely, the Slowloris, HTTP POST and Slowread attacks. All these attacks exploit the HTTP protocol.

A. Slowloris Attack

The Slowloris attack [slo13a] was developed by the group Anonymous in 2009. It exploits the HTTP GET method, used to retrieve web content. The Anonymous group implemented a tool available on the Internet for anyone to download. One does not require any network expertise to carry out the attack, but simply run the tool informing the duration, the target and the intensity of the attack. Thus, any amateur can target easily target any web-server.

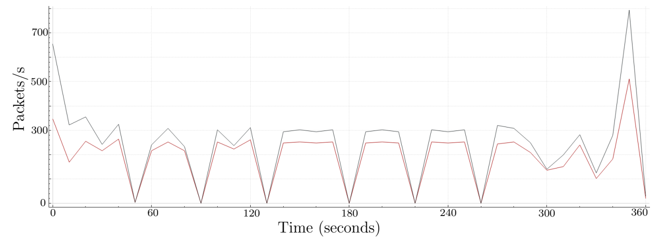


Fig. 2. Attacker traffic requests sent (in black) and responded (in red) by the web-server.

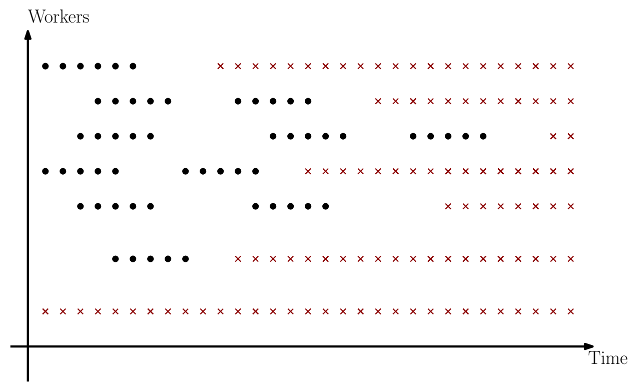


Fig. 3. Illustration of the effectiveness Low-Rate ADDoS attacks. Circles (in black) denote legitimate requests and crosses (in red) denote attacker requests.

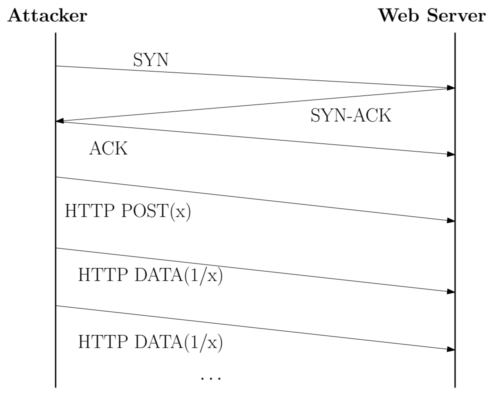


Fig. 4. Sequence of messages in a HTTP POST Attack.

The attack follows by using the sequence of messages shown in Figure 1. The attacker completes the SYN-ACK handshake with the target web-server thus establishing a connection with the web-server machine exactly as a legitimate client would do. Then the attacker sends an incomplete header with a GET request. At this moment the web-server allocates one of its workers to handle this incoming request. However, since the request header is not yet complete, the web-server waits, as specified by the HTTP protocol version 1.1 [RFC99], until a new piece of the request header is received. If such a piece is not received until a given timeout elapses, the web-server rejects this request.[5] The attacker can easily infer, beforehand, this timeout by measuring the time until a request with an incomplete header is rejected. Thus, once the timeout is almost reached, the attacker sends another piece of the header, which can be anything actually as long as it does not complete the header of the request. The tool implemented by Anonymous sends “x-1:aa”, but this could be a random string. This causes the web-server to reset the timeout count and wait once again for another piece of the header.

By sending a number of such incomplete requests greater than the number of workers available in a web-server, the attacker can deny service to legitimate clients. A small to medium web-servers have around 250 workers which means that the application can handle up to 250 requests simultaneously. Thus, the attacker can deny service sending for instance 300 requests. The Slowloris attack does not generate a large amount of traffic as the number of requests sent is low and they are renewed after long periods (the timeout set in the web-server, typically 40 s). Moreover, the web-server does not need to perform any CPU/Memory-intensive operations. Thus for the network administrators (or automated tool) monitoring traffic intensity, CPU and memory consumption, the web-server does not seem to be under an attack. Indeed in our experiments, the Apache web-server’s CPU usage remained close to 1% and its memory usage corresponded to only 18% of the total memory.

Figure 2 illustrates the traffic generated by the implementation of Slowloris easily available online. It sends bursts of incomplete headers periodically, thus generating low traffic. Moreover, we can also observe that most of them are responded by the web-server indicating a successful attack as the web-server’s workers are maintained busy by the attacker.

However, legitimate users cannot be served because there are no free workers available to serve them. The intuition for why the attack works is illustrated by Figure 3. While legitimate traffic normally sends a complete request thus being served promptly by the web-server, the attacker is able to occupy a worker leaving it waiting for the attacker request to be completed for long periods of time. Once all the workers are left in this state, there are no more workers available to serve legitimate client requests and therefore the service is denied.

B. HTTP POST Attack

The HTTP POST attack [rudy13] exploits the POST method from the HTTP protocol. It works on web-pages that have some type of form to be filled. Figure 4 depicts the sequence of messages used in the HTTP POST attack.

The attacker establishes a connection by completing the SYN-ACK handshake with the web-server’s machine. At this

[5]By default Apache web-servers set this timeout to 300 s, but some recommend to set it to 40 s to improve the quality of service. In our experiments, we set it to 40 s.

point, the attacker sends a (complete) HTTP POST request informing that it will send a large number, x, of bytes to the web-server corresponding to the contents of some form. The web-server, then, allocates one of its workers to serve this request. However, instead of sending big chunks of this data to the web-server as a legitimate client would do, the attacker sends very small pieces, typically a single byte per subsequent message. Moreover, it sends each piece in large intervals (typically 10s per byte). In the meantime the allocated worker waits until it receives all x bytes and cannot serve any other (legitimate) requests. By sending a number of POST requests greater than the number of workers in the web-server, all workers are busy waiting for attacker requests to be completed and therefore the web-server is not able to serve legitimate clients.

HTTP POST attacks are effective against connection-based web-servers, such as Apache, where requests are responded synchronously, while this attack is not effective against event-based web-servers, such as nginx, where requests are responded asynchronously.

As with the Slowloris attack, HTTP POST attack does not generate high traffic load, nor high CPU and memory consumption. Therefore, it is easy for network administrators (or automated tool) monitoring these parameters to believe that their application is not under attack.

C. Slowread Attack

Slowread [Slo13b] is an ingenious attack that combines elements from both the application and the transport layers. The attacker exploits TCP flow control by reducing the rate in which it can read web-server responses and thus occupying the web-server's resources for a long time. Figure 5 depicts the sequence of messages used to carry out a Slowread attack.

After completing the SYN-ACK handshake with the web-server's machine, the attacker sends a GET request setting an initial TCP window (in Figure 5, the window 1472 bytes). At this point, the web-server allocates one of its workers to serve this request and starts to send the requested content. The machine sends this data (in the transport layer) according to the established window. For instance, in the first ACK packet in Figure 5, the machine sends 1410 bytes. However, in the attackers ACK response, he sets an even lower TCP window of 778 bytes. The web-server's machine is now limited to send this number of bytes. By proceeding this way, the attacker is able to reduce the TCP window to 0, causing a halt on the transmission of the requested data. The connection remains open until the Keep-Alive timeout elapses.

As with the two previous attacks, there is no burden in traffic, memory nor CPU processing power. Thus, for a network administrator (or automated tool) monitoring these parameters, the web-server is not under attack as it seems to be receiving legitimate requests.

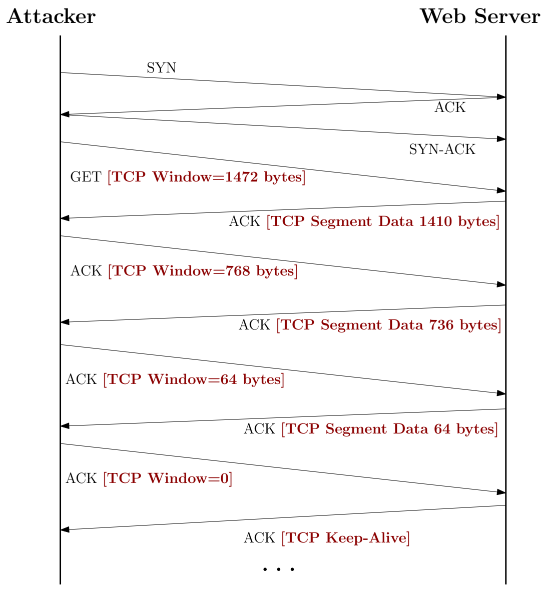

Attacker
Web Server
SYN
ACK
SYN-ACK
GET [TCP Window=1472 bytes]
ACK [TCP Segment Data 1410 bytes]
ACK [TCP Window=768 bytes]
ACK [TCP Segment Data 736 bytes]
ACK [TCP Window=64 bytes]
ACK [TCP Segment Data 64 bytes]
ACK [TCP Window=0]
ACK [TCP Keep-Alive]
. . .

Fig. 5. Sequence of messages in a Slowread Attack.

III. Bypassing Timeout-Based Defenses: Resurrected Attacks

A. Possible Countermeasures

Unfortunately, due to the low-rate of the attacks described above, simply monitoring traffic volume cannot be used to identify an attack and even less to prevent it from happening.

How could one mitigate such attacks? We discuss some options, which have been proposed in blogs and literature, but that are unfortunately *too brittle* as simple modifications of attack instances render these solutions ineffective. Therefore, these are not suitable solutions for network-security.

- **Inspecting Packet Contents:** For the Slowloris attack, one could monitor the packets that are received. Whenever a contents of a message is repeated twice with the string "x-1:aa", for instance, the defense would block the corresponding IP [dA13]. While this defense is indeed effective for mitigating some existing implementations of the Slowloris attack, it can be easily bypassed by instead of using the same string "x-1:aa", using a random string;
- **Monitoring the Intervals Between Packets:** Alternatively, one could monitor the intervals between packets and if the interval is regular enough, block the corresponding IP. Once again, while this may work with current implementations of Slowloris, which sends periodically bursts of requests, this defense can be easily bypassed, by instead of sending busts of request periodically, randomizing the intervals;
- **Length of HTTP Body:** One could also monitor the length HTTP Body in order to detect whether it is an attempt to carry out a HTTP POST attack. If the length is too high, *i.e.*, is greater than a threshold, then block the corresponding IP. Again, this defense is not robust, as the attacker can by try and error estimate this threshold and

use the upper limit during the attack. None of his requests will be blocked. Moreover, this defense may also result in many false positives, as legitimate users might attempt to send large amounts of data passing the threshold;

- **Monitor the TCP Window Length:** One could also monitor the TCP Window Length and block requests/IP that use very low window lengths, *i.e.*, less than some threshold. This solution suffers the same problems as the previous point of monitoring the length of HTTP Body, as the attacker can by try and error estimate this threshold and can bypass this defense. Moreover, this defense may block users that have slow connections.

The main question is, therefore, *how can we build a defense that is robust to many/all variations of such low-rate attacks?*

The current most successful solution which is implemented in Apache and nginx are time-based solutions which we will discuss next. However, as we demonstrate here, while these solutions are robust to many variations of current implementations of low-rate attacks, they can be bypassed by using our novel Ressurected Slowloris attack.

B. Time-Based Defenses

The main solution adopted to mitigate the Low-Rate DDoS attacks described in Section II is to monitor each request and set different timeouts for completing requests. Such timeout-based defenses have been implemented as an Apache module, called ReqTimeOut [ReQ14] which is the official solution for mitigating the Slowloris attack. A similar solution is default in the nginx web-server. They are effective in defending against Low-Rate DDoS attack, in particular, in defending attacks generated by the tools available on the Internet although our experiments demonstrate that nginx is much less efficient than the module ReqTimeOut. They implement different timeouts. The following are the most relevant for the types of attacks we are considering:

- **Header Completion Timeout (t_H):** If a request header is not completed within t_H seconds since the first received header information, then drop this request;
- **Body Completion Timeout (t_B):** If the body of a request is not completed within t_B seconds since the first piece of the body information, then drop this request;
- **Read Timeout (t_R):** If a response is not read by the client within t_R seconds since the response was issued by the web-server, then drop this request.

The Header, Body and Read timeouts are used to mitigate Slowloris, HTTP POST, and Slowread attacks, respectively. For example, due to the Header Completion Timeout, the strategy used by Slowloris to consume the workers of a web-server does not work. After t_H seconds all the subsequent messages sent by the attacker will be ignored and therefore the worker will be free to serve other requests. The same is true when using the Body Completion Timeout and Read Timeout for mitigating the HTTP POST and Slowread attacks, respectively. Indeed our experiments demonstrated that when under the Slowloris attack, an Apache server with the module ReqTimeOut is able to serve 93.8% of all client requests.

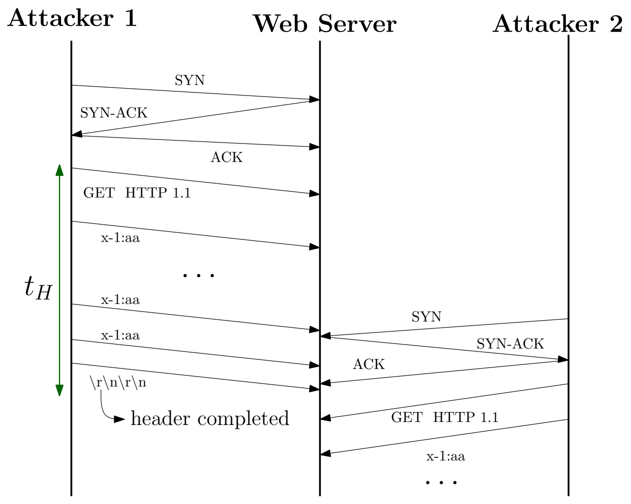


Fig. 6. Resurrected Slowloris Attack Illustration. The Header Completion Timeout is denoted by t_H.

However, although these timeout-based defenses mitigate these attacks, it is easy to come up with other attacks that can bypass such defenses. This is because the attacker can easily infer beforehand each one of these timeouts by measuring the time until his requests are rejected. For example, to infer the Header Completion Timeout, the attacker can simply send an incomplete header and then pieces of this header with completing the header, just as done in the Slowloris attack. Then, he measures the duration between the first message and when he is informed that his request is dropped. This duration will correspond roughly to the timeout t_H used.

By knowing this timeout t_H, the attacker can construct a new attack based on Slowloris called Resurrected Slowloris, which is able to bypass timeout-based defenses by appropriately creating new instances of the corresponding attack. Similarly, the attacker can also construct Resurrected versions of the HTTP POST and Slowread attacks. We explain in detail the Resurrected Slowloris Attack.

C. Resurrected Slowloris

Assume the attacker knows the Header Completion Timeout t_H used by a web-server. The attack is illustrated by Figure 6. It follows by coordinating the launch of a new Slowloris attack when the time t_H elapses. We assume that there are two colluding attackers, Attacker 1 and Attacker 2, but it can also be carried out in the same machine using multiple threads. Attacker 1 starts an instance of the Slowloris attack, *i.e.*, completing the TCP handshake, sending a GET request with incomplete header, and then keeps sending pieces of the header. However, before the Header Completion Timeout t_H elapses, Attacker 2 starts a new instance of the Slowloris attack. This new instance of the Slowloris attack should have enough time to perform the TCP handshake with the web-server machine. In the meantime, close to the time when t_H elapses, Attacker 1 finishes the GET request header completing its interaction with the web-server thus not causing much suspicion (the message "\r\n\r\n" in Figure 6). The attack now follows by alternating the roles of the attackers, *i.e.*, Attacker 1 waiting for a bit less than t_H seconds to elapse while Attacker 2 carries out the Slowloris attack.[6]

The Resurrected Attack can bypass the timeout-based defenses currently used in most web-servers. We implemented this attack and carried out a number of experiments with both Apache with the timeout-based defense ReqTimeOut and the nginx web-server which already contains a timeout-based defense.

- **Header Completion Timeout:** We used as Header Completion Timeout the value specified as default by nginx, which is of $t_H = 60$ s. The module ReqTimeOut documentation does not suggest a value, but points out that the value for t_H should not be too low as it may affect legitimate clients which have slow connections leading to many false positives, *i.e.*, block legitimate requests of slow clients;
- **Web-Servers:** We used the web-servers Apache version 2.4.10[7] and nginx version 1.6.2. These are the two most used web-servers, Apache hosting more than 40% of all active sites and nginx hosting more than 15% of all active sites[8]. Another reason for choosing these web-servers is because they have different ways of treating incoming requests. Apache is a *connection-based web-server*, where requests are served synchronously, while nginx is an *event-based web-server*, where requests are served asynchronously. The treatment of request responses has an impact on the effectiveness of the attacks as our experiments demonstrate. In our experiments, these web-servers had 200 workers both.
- **Modified Slowloris:** We modified the Slowloris Tool so that it carries out the Resurrected Slowloris attack given the Header Completion Timeout t_H. The attacker traffic is generated by this tool. Our tool restarts Slowloris every 50 seconds and each instance of Slowloris constructs 250 simultaneous connections, *i.e.*, a bit greater than the number of workers (200) in the tested web-servers.
- **Client Traffic:** We used the tool Siege [Sie14] to simulate legitimate client traffic. Siege is a tool used to carry out load and benchmark tests which estimate a lower-bound on the response of systems. It simply attempts to get a random object in the web-page being tested. We used Siege to generate client traffic at a rate of 10 requests per second which is a rate for which both Apache and nginx are able to serve when not under attack. Siege keeps track of which requests have been successfully responded and the time to respond. All the numeric results for success rate and TTS in the this paper involving Siege (Tables I and II) were taken from Siege's output.
- **Experiment Duration:** Each one of the experiments took 30 minutes.

[6]Recall that the current implementations of Slowloris sends bursts of requests. The two attackers are therefore needed to coordinate when these bursts are send. We believe that a similar attack could be carried out if implementations send request in a constant rate instead of bursts of requests. This study is left to future work.

[7]Using the prefork module.

[8]http://news.netcraft.com/archives/2015/01/15/january-2015-web-server-survey.html

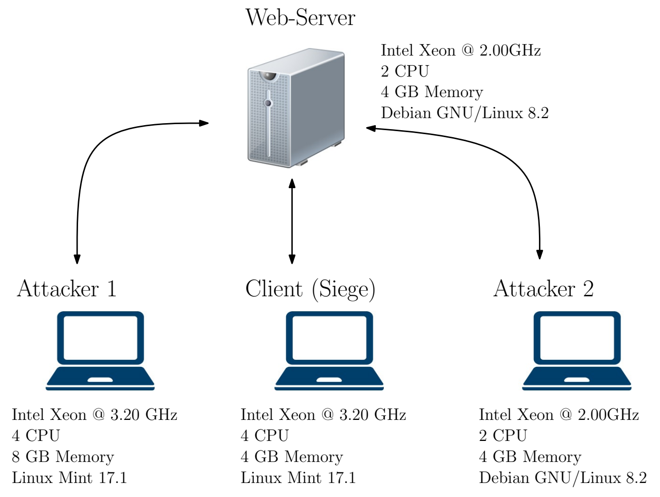


Fig. 7. Experimental Set-Up without SeVen.

The experimental setting is illustrated by Figure 7. We used four different machines, one hosting the web-server, one generating client traffic, *i.e.*, running Siege, and two computers generating attacker traffic. The configurations of the machines are also depicted in Figure 7.

We measured *Success Rate*, that is, the percentage of requests sent by Siege that were successfully served, and the average *Time-to-Service* (TTS), which is the duration needed for a client request to obtain a server response.

Table I summarizes the results we obtained from our experiments. Indeed, the timeout-based defense ReqTimeOut is able to mitigate the Slowloris attack and Apache is able to serve most of client requests despite a high average time-to-service of 1.13 s, while without the defense, Apache's service is denied. However, when faced with the Resurrected Slowloris attack, ReqTimeOut does not perform that well. Apache is only able to serve 7.6% with an average TTS of 1.60 s. As the Resurrected Slowloris attack was tailored to bypass the ReqTimeOut defense, it is not so surprising that Apache performed a bit better, although still poorly, when under the Resurrected Slowloris attack and not using ReqTimeOut, being able to serve only 31.9% of clients with an average TTS of 1.28 s. In practice, the attacker can choose whether he wants to carry out a Slowloris or a Resurrected Slowloris attack depending on whether Apache is using or not a timeout based defense.

Attack	Apache		Apache with ReqTimeOut		nginx	
	Success Rate	TTS	Success Rate	TTS	Success Rate	TTS
Slowloris	0.0%	∞	93.8%	1.13s	15.3%	0.00s
Resurrected Slowloris	31.9%	1.28s	7.6%	1.60s	4.3%	0.00s

TABLE I
EXPERIMENTAL RESULTS WITH THE RESURRECTED SLOWLORIS AND TIMEOUT-BASED DEFENSE REQTIMEOUT AND NGINX WHICH ALREADY INCORPORATES SUCH DEFENSE BY DEFAULT. THE DURATION OF ALL EXPERIMENTS WAS OF 30 MINS. WE MEASURED THE SUCCESS RATE AND THE TIME-TO-SERVICE (TTS).

On the other hand, nginx was not so successful in mitigating Slowloris, being able to serve only 15.3% of the legitimate client requests. However, it was able to respond to legitimate requests almost immediately. Moreover, when under the Resurrected Slowloris attack, nginx also performed poorly being able to serve only 4.3% of legitimate client requests responding these requests immediately. The low TTS for nginx, as compared to Apache, is due to the fact that it is event based web-server while Apache is connection based. We were expecting that nginx would perform as well as Apache with ReqTimeOut against the Slowloris attack, but taking a closer look, as nginx is able to respond requests more quickly, it allowed the Slowloris tool to hit nginx more often than with Apache leading to lower success rates.

These attacks had a low impact on memory and CPU processing was low. Under all attack scenarios, the web-servers used at most 1% of CPU processing and 18% of the total memory. Thus, tools that monitor these parameters would not be able to detect that the web-server is under attack.

Finally, it is possible to implement in a similar fashion Resurrected versions of the HTTP POST and Slowread attacks follow the same idea, but by using as underlying attack the HTTP POST and Slowread attacks, respectively. Experimenting with these attacks is left to future work.

IV. SeVen

Selective strategies have been used for mitigating (high-volume) Transport-Layer DDoS attacks [KVF+08], [KVF+12] which involve stateless connections only. The selective strategy used by SeVen, on the other hand, deals with Low-Rate ADDoS which exploit state dependent connections as described in Section II.

This paper introduces SeVen a lightweight and robust defense implementing this selective defense for mitigating Low-Rate ADDoS attacks. We first explain in detail the selective strategy that we used and then we provide details of how SeVen is implemented and deployed.

The selective strategy used by SeVen has been formalized in [DNF14] using the rewrite tool Maude [CDE+07]. Dantas *et al.* carried out a number of simulations and formal analysis using Symbolic Statistical Model-Checking. The results obtained by these analysis showed that it is a promising defense against Low-Rate ADDoS. This paper validates these results by means of a large number and variety of experiments.

A. *Selective Strategy for Low-Rate ADDoS Attacks*

A selective strategy is composed of two functions:

$$\begin{array}{lcl}\mathsf{Select} & : & \mathsf{Request} \times \mathsf{State} \to \mathsf{bool} \\ \mathsf{Drop} & : & \mathsf{State} \to \mathsf{Request}\end{array}$$

where State is the set of states of the system being defended, which normally includes the requests that are currently being served and their meta-data, such as their internal states, number of requests being received, etc and Request is the set of requests, which among other data contain a unique a identifier id.

The function Select specifies when an incoming request should be selected to be served, and Drop specifies which request being processed should be dropped. What makes selective strategies powerful is the fact that the functions Select and Drop are *governed by some given probability functions* which may depend on the state of system. So it is not predictable whether a request is going to be selected by Select and which request is going to be dropped by Drop. Therefore, different from the timeout-based defenses explained above, it is much difficult for an attacker to orchestrate an attack even if he knows the probability distributions governing Select and Drop.

Its general algorithm works as follows. Assume that N_T is the maximum number of requests that the application can handle simultaneously. For web-servers N_T is the number of workers it has. Assume that an incoming request req_ν arrives to the application which has currently state $\mathcal{S}$. Let $N_\mathcal{S}$ be the number of requests being processed in the state $\mathcal{S}$.

1) If $N_\mathcal{S} < N_T$ or req_ν is a continuation of a request already being processed in $\mathcal{S}$, accept the request req_ν;
2) Otherwise, the application's maximum capacity is reached and a new incoming request req_ν wants to be served. Let $b = \mathsf{Select}(\mathcal{S}, \mathsf{req}_\nu)$.
   a) If b is false, then the selective strategy did not select the request req_ν and therefore this request is dropped and a message is sent to the corresponding user informing that the service is unavailable;
   b) Otherwise if b is true, then the defense should decide which request currently being processed should be dropped and

be replaced by req_ν. Let $\mathsf{req}_i = \mathsf{Drop}(\mathcal{S})$. The request req_i is dropped from $\mathcal{S}$ and the req_ν takes its place.

Notice that requests (from both clients and attackers) are dropped *only when the maximum capacity of the web-server is reached.* This means that in normal conditions, *i.e.*, not under attack, SeVen does not generate false positive.

Notice as well that the Step 2(b) may drop a request that is currently being processed similarly to timeout-based defenses described in Section III that also drop requests. This is a necessary evil in attacks such as Low-Rate ADDoS attacks where it is hard to distinguish legitimate client traffic from attacker traffic. However, while timeout-based defenses penalize slow requests and therefore are vulnerable to the Resurrected Slowloris attack, selective strategies penalize requests randomly. Therefore, it is much harder for an attacker to orchestrate an attack against such a defense. Moreover, as our experiments demonstrate in Section V, legitimate clients are still served using selective strategies.

Finally, by suitably configuring the functions Select and Drop according to the domain application, it is possible to mitigate other types of attacks, for example, Telephony Denial of Service attacks on VoIP services [LDNS16], [DLFN16]. We detail next these functions for mitigating Low-Rate ADDoS Attacks.

a) Application Defense State: A state of the defense system has the following form:

$$\mathcal{S} = \langle \mathcal{R}, \mathsf{Factor} \rangle$$

where $\mathcal{R} = \{\mathsf{req}_1, \ldots, \mathsf{req}_n\}$ is the set of requests being processed by the web-server and Factor is a natural number indicating the rate of incoming requests once the application has reached its maximum capacity. In practice, Factor is a counter that is incremented whenever all web-server's workers are busy and an incoming request arrives and it is reset every 100 ms.

We assume requests are of the form $\langle id, par_1, \ldots, par_n \rangle$ where id uniquely identifies the request, containing the origin IP and socket, and $par_1, \ldots, par_n$ are parameters stored by the web-server which may include the header information, the time since that request has been received, etc. These parameters may be used in the definition of the functions Select and Drop.

b) Selective Strategy Functions: The functions Select and Drop are probabilistic functions. The function Select is used to determine when a new incoming request should be selected to be served, while Drop is used to determine which request currently being processed should be dropped.

The function Select for a new incoming request req_ν and state $\langle \mathcal{R}, \mathsf{Factor} \rangle$ is defined as follows:

1) Generate a random number $r \in [0, 1]$ using a uniform distribution;
2) Return true if
$$r \leq \frac{N_T}{N_T + \mathsf{Factor}}; \qquad (1)$$
3) Return false otherwise.

Intuitively, Equation 1 specified to penalize incoming traffic according to the incoming traffic rate, so that requests being processed are not constantly being replaced. Thus the higher the rate of incoming request stored in Factor the lower is the chance of selecting a new incoming request, that is, the probability of accepting new incoming requests reduces progressively once the buffer contains N_T requests. This choice was motivated by the work on mitigating Transport-Layer DDoS attacks [KVF+08], [KVF+12] and also validated by using formal methods [DNF14].

The function Drop simply selects using a uniform probability distribution which of one of the request being currently being processed should be dropped. That is, all requests currently being processed have equal chance of being dropped.

B. Example

Assume that $N_T = 4$ and that initially there are three requests being processed, where for simplicity we use natural numbers for data parameter of requests:

$$\mathcal{S}_0 = \langle \{\langle \mathtt{id}_1, 5\rangle, \langle \mathtt{id}_2, 5\rangle, \langle \mathtt{id}_3, 2\rangle\}, 0\rangle$$

Moreover, assume that $\mathsf{Factor} = 0$. The application has received partial data from three requests, $\mathtt{id}_1, \mathtt{id}_2, \mathtt{id}_3$. Say that it receives the request $\langle \mathtt{id}_1, 5\rangle$ with 5 pieces of data. The state of the system is updated to:

$$\mathcal{S}_1 = \langle \{\langle \mathtt{id}_1, 10\rangle, \langle \mathtt{id}_2, 5\rangle, \langle \mathtt{id}_3, 2\rangle\}, 0\rangle$$

with Factor still 0, as the application still has available workers.

Assume now that a new request arrives: $\langle \mathtt{id}_4, 7\rangle$ with a fresh identifier $\mathtt{id}_4$. Since the number of requests being processed in $\mathcal{S}_1$ is less than $N_T = 4$, the application can safely process this request leading to the state:

$$\mathcal{S}_2 = \langle \{\langle \mathtt{id}_1, 10\rangle, \langle \mathtt{id}_2, 5\rangle, \langle \mathtt{id}_3, 2\rangle, \langle \mathtt{id}_4, 7\rangle\}, 0\rangle$$

Assume now that another request $req_\nu = \langle \mathtt{id}_5, 1\rangle$ has arrived. Since the number of requests in $\mathcal{S}_2$ is equal to $N_T = 4$, the application must decide whether to serve req_ν or not. It uses the function Select, which generates a random number in the interval $[0, 1]$ and checks whether it is less or equal to:

$$\frac{N_T}{N_T + \mathsf{Factor}} = \frac{4}{4+1} = \frac{4}{5} \geq 0.80$$

Thus, this request has 80% of chance of being selected to be served.

Let req_ν be selected. The defense needs, now, to decide which one of the requests currently being processed should be dropped. It runs Drop, which selects at random one such requests. Say that it selected the request $\langle \mathtt{id}_2, 5\rangle$. The new state is then:

$$\mathcal{S}_3 = \langle \{\langle \mathtt{id}_1, 10\rangle, \langle \mathtt{id}_3, 2\rangle, \langle \mathtt{id}_4, 7\rangle, \langle \mathtt{id}_5, 1\rangle\}, 1\rangle$$

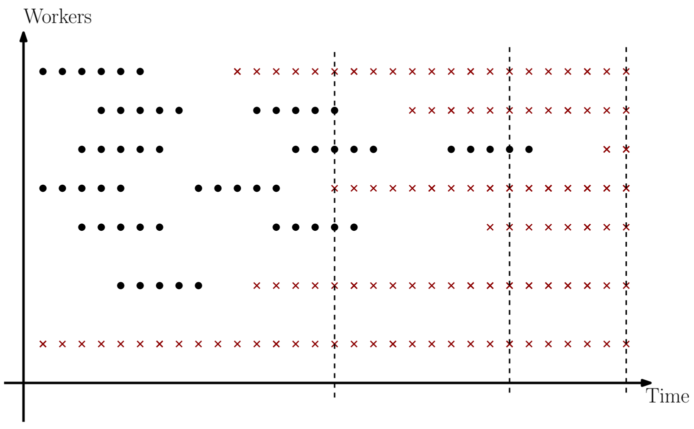


Fig. 8. Illustration of the effectiveness of selective strategies against Low-Rate ADDoS attacks. Circles (in black) denote legitimate requests and crosses (in red) denote attacker requests.

Consider now that another request, $\mathsf{req}'_\nu = \langle \mathtt{id}_6, 3\rangle$ arrives. Since the application is still at its maximum capacity, the defense runs Select and req'_ν has the following chance of being selected:

$$\frac{N_T}{N_T + \mathsf{Factor}} = \frac{4}{4+2} = \frac{2}{3} \leq 0.66$$

Let req'_ν not be selected. The set of requests being processed remains the same, but $\mathsf{Factor} = 2$.

Finally, whenever a request is finished to be served, it is removed from the state. For example, if the request identified with $\mathtt{id}_1$ is finished, the new state of the application is:

$$\mathcal{S}_4 = \langle \{\langle \mathtt{id}_3, 2\rangle, \langle \mathtt{id}_4, 7\rangle, \langle \mathtt{id}_5, 1\rangle\}, 0\rangle$$

where the application is no longer in its maximum capacity and therefore, Factor is also reset to 0.

C. Intuition, Implementation and Extensions

c) Intuition for the Effectiveness of Selective Strategies: The strategy that we just described above is not tailored to mitigate any particular attack. We are simply selecting requests by using traffic rate which incoming requests to select and by random with uniform probability which requests to drop. It may seem puzzling that such a simple strategy works.

To comprehend the intuition of why this is true (as observed by our experimental results), consider Figure 8 in Section II. Recall that the the goal of a Low-Rate ADDoS attack is to occupy for long periods of time the workers of the web-server. This is represented by the red crosses in Figure 8. Once the application is at its maximum capacity, it is likely that it is under attack[9] and it is likely that there is larger number of attackers consuming the web-server's resource than legitimate clients.

[9]The application may also be in its maximum capacity if there is a great influx of legitimate clients, such as an Internet hit or it is some period that has much demand. In both cases, the application should be designed/increased to handle such a traffic.

Therefore, although the requests to be dropped are selected at random, there is a greater chance of selecting a request from an attacker. This is illustrated in Figure 8 by the vertical dashed lines representing moments when SeVen needs to drop some request that is being processed by a worker. During an attack, the number of red crosses representing attacker requests being processed is much greater than the number of black dots representing legitimate client requests. Thus the chance of removing an attacker request is higher.

This behavior has been observed not only experimentally as described here (Section V, but also by the use of symbolic simulations and statistical model checking reported in [DNF14].

d) Alternative Selective Strategies: While it is not in the scope of this paper, we briefly described other possible selective strategies which can be used provided we have some more knowledge about legitimate and attacker traffic. We have implemented some of these alternative strategies, but a more in-depth analysis of such strategies is left for future work.

- **Black and White Lists:** It is possible to incorporate in a selective strategy black lists, specifying which requests for which service should be denied, and white lists, specifying which requests for which service should be provided. It is easy to adapt our selective strategy to incorporate black and white lists. We already implemented this feature in our tool SeVen.
  The selective defense would set probability of being selected to 0 to the requests that are in the black list and 1 to the requests that are in the white list, that is, never select requests that are in black lists (even if the application is not in its maximum capacity) and always select requests that are in the white list. Moreover, the function Drop would no longer consider all requests equally using a uniform distribution. The application can maintain two sets of requests, S_W and S_N, containing, respectively, the requests being served that are in the white list and that are not in the white list. Then Drop would choose requests to drop with uniform distributions from the set S_N. If $S_N = \emptyset$, then it would select also with uniform distribution requests from the set S_W. In this way, the requests in the white list would have a higher chance of being served.
- **Time-Based:** In many cases it is possible to predict from, for example, the behavior of usual legitimate clients, web-page content, *e.g.*, long video streaming, and information from logs, the average time that a typical legitimate client takes to be served. This information can be incorporated in the probability distributions used to drop requests, that is, it can be time-dependent. For example, if the average time a legitimate client takes is t_A, the probability of dropping requests can increase for requests that are taking longer than t_A, *e.g.*, increase in a exponential rate. Thus, slow users would be penalized. The extreme would be to use the time-out based defenses described in Section III with thresholds.

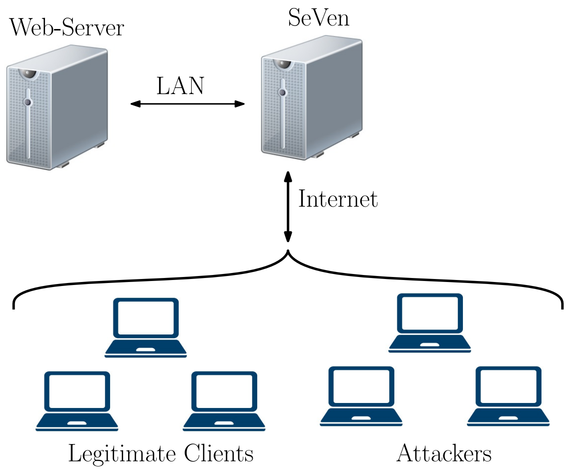


Fig. 9. SeVen deployment as a proxy.

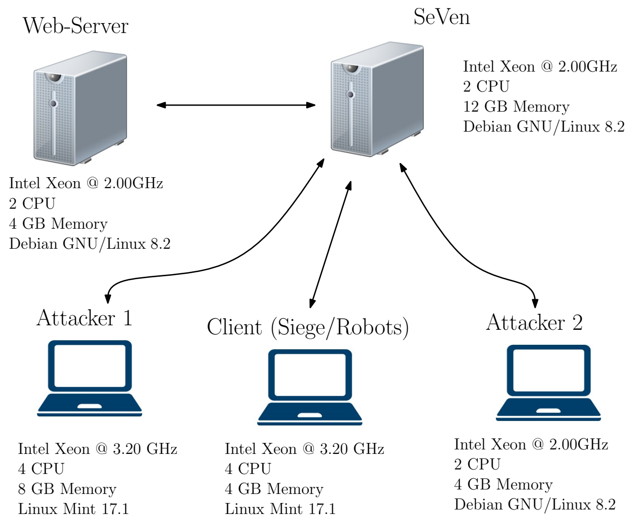


Fig. 10. Experimental Set-Up with SeVen

A time-based selective strategy has been used for mitigating Telephony Denial of Service Attacks [LDNS16]. Calls that are calling longer than an average t_M become more likely to be dropped by Drop increasing its chances with an exponential rate once the duration of the call reaches t_M.

- **Authentication/Credentials Based:** If an application has some authentication mechanism or some credential system (*e.g.*, classes of users), the selective strategy can incorporate such systems to give priority to serve users that have higher credentials. This is done in a way similar as described for incorporating Black and White Lists. The probability to select/keep requests of users with higher credentials can be set to be greater than the probability of selecting users with lower credentials.
- **History Based:** Applications may keep a log of interaction with users and this information can be used to configure a selective strategy in different ways. For example, it might be desirable to prioritize users that have used less the service than users that are over-using the service. This can also be incorporated in the selective strategy by setting a higher probability of selecting/keeping the former request than the latter requests.

e) Implementation and Deployment: SeVen is a C++ implementation of the selective strategy described above. It currently works on Linux distributions. SeVen is a proxy as illustrated by Figure 9. All traffic directed to and from the web-server passes through SeVen which keeps track of all the requests that are being processed by the web-server. SeVen uses a thread pool to manage efficiently connections between the web-server and between clients. As our experiments demonstrate in Section V, SeVen is lightweight not needing much memory nor CPU processing power to carry out the defense. Thus SeVen can also be deployed in the same machine as the web-server.

It is possible to implement SeVen as an Apache module [PCNF16], instead of a proxy. While there are many advantages in doing so, we propose using it as proxy as it allows us to defend different web-servers. In our experiments, we used Apache, nginx and Tomcat web-servers.

Finally, we point out that we are currently implementing a version of SeVen that supports HTTPS connections and that keeps track remotely of the web-server's state, *i.e.*, workers available, used, etc, using *e.g.*, Apache's Mode_Status Module.

V. Experimental Analysis

We carried out a large number of experiments comprising more than 12 hours of different type of experiments. They are classified into Benchmark/Load test, in Section V-A, and Quality of Service tests, in Section V-B.

In the Benchmark/Load test, we used the tool Siege [Sie14] to check how SeVen behaves when it receives a constant rate of GET request, generated by Siege, and when under attack. Benchmark/Load Tests provide lower-bounds on the effectiveness of SeVen.

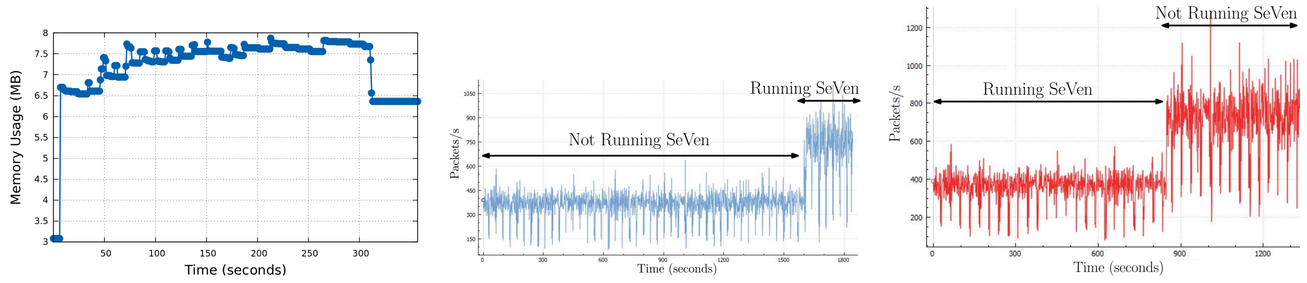
In the Quality of Service Test, we use robots to simulate the human interaction with a web-site. These experiments provide a measurement of the actual experience of users when, for instance, using a web-browser.

For the experiments without SeVen, we used the setting shown in Figure 7, the same as used in our experiments in Section III. For the experiments involving SeVen, we used the setting depicted in Figure 10 with an extra machine where SeVen is executed as a proxy for the target web-server. In the client machine we executed either Siege, in our Load Tests, or our robots, in our Quality of Service Tests.

We also made sure to create attacker traffic large enough so to reduce considerably the service available when not running SeVen. Typically, the attacker generates a bit more than the number of connections as the workers available in the web-server, *e.g.*, if the web-server has 200 workers, the attack would create 250 simultaneous connections. Notice, however,

Attack	Apache		Apache with SeVen		nginx		nginx with SeVen	
	Success Rate	TTS	Success Rate	TTS	Success Rate	TTS	Success Rate	TTS
Slowloris	0.0%	∞	98.7%	0.15s	15.3%	0.00s	81.4%	0.01s
HTTP POST	0.0%	∞	97.3%	0.14s	–	–	–	–
Slowread	13.8%	1.99s	94.4%	4.68s	5.2%	1.29s	97.9%	3.33s
Resurrected Slowloris	31.9%	1.28s	95.6%	0.58s	4.3%	0.00s	81.7%	0.01s
Slowloris + HTTP POST	0.0%	∞	95.2%	0.20s	–	–	–	–

TABLE II
EXPERIMENTAL RESULTS WITH SEVEN. THE DURATION OF ALL EXPERIMENTS WAS OF AT LEAST 30 MINS. WE MEASURED THE SUCCESS RATE AND THE TIME-TO-SERVICE (TTS).



(a) SeVen Memory Usage.   (b) Responses to Clients.   (c) Responses to Attackers.

Fig. 11. Illustration over time of SeVen memory usage when mitigating an attack and of client and attacker responses when using and not using SeVen.

that despite the great number of connections, since these are low-rate attacks, each connection sends requests with a slow rate (as described in Section II), and therefore, the attacker traffic hitting the server is low.

We generated the Slowloris, HTTP POST, Slowread and Resurrected Slowloris attacks using standard tools available on the Internet: The Slowloris tool [slo13a], the test tool called Slow HTTP Test [Tes15] for generating the HTTP POST attack and the Slowread attacks. Moreover, we used our adaptation of the Slowloris tool to generated the Resurrected Slowloris as described in Section III.

A. Benchmark/Load Test

Benchmark/Load Tests provide lower-bounds on the effectiveness of SeVen. We used the same machinery as used in Section III, but now using SeVen. We used Siege to generate legitimate client traffic, Apache and nginx web-servers with the same configurations for the as described in Section III. Moreover, each one of the experiments had a duration of at least 30 minutes.

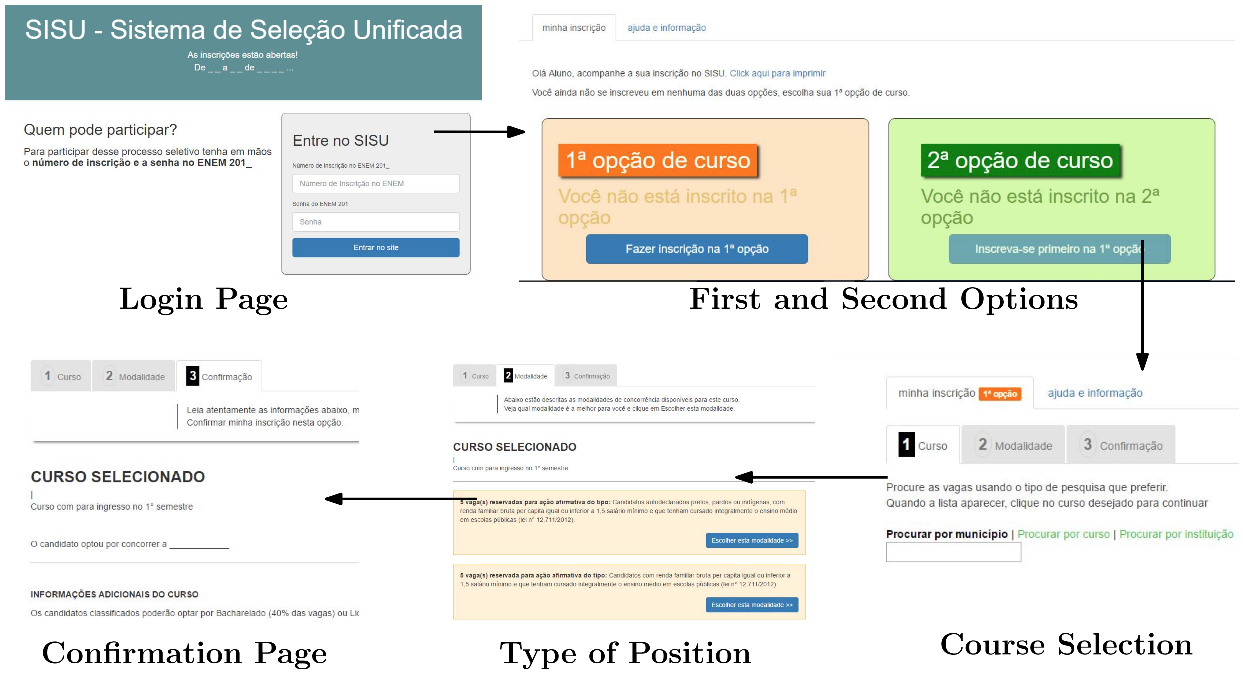
Table II summarizes the results obtained from our main experiments. In general, SeVen performed well in mitigating all Low-Rate ADDoS attacks considered including the new Resurrected Slowloris attack. We also carried out some experiments with the combination of Slowloris and HTTP POST attacks.

For the connection-based Apache web-server, the attacks were able to greatly reduce its service level yielding low success rates when not using SeVen. However, when using SeVen, most of the legitimate client requests are served. Moreover, the average TTS is also greatly improved when defending against most of the attacks being less than 0.6s. The only exception is for the Slowread attack where the average TTS was of 4.68s. Taking a closer look, however, we noticed that a small number requests had a huge TTS being responded only 70s after the request was issued affecting thus the TTS average.

When using the event-based web-server nginx, the attacks were also able to reduce the nginx's performance greatly affecting its success rates. We did not experiment with HTTP POST attack as it is known to not affect nginx[10] because nginx is an event-based sending asynchronously responses. Moreover, the TTS was still low with the exception of the Slowread attack where nginx served request in an average time of 1.29s. When using SeVen, however, the service improved considerably, although less than for Apache, with success rates of over 80% for the Slowloris, and Resurrected Slowloris, and of 97.9% for Slowread. The average TTS was not affected being low for most cases. However, as with Apache, the average TTS increased to 3.33s when using SeVen and facing a Slowread attack for the same reasons as described above for when using Apache.

The main reason why Slowloris and Resurrected Slowloris

[10]See https://www.nginx.com/blog/nginx-vs-apache-our-view/ and also https://www.youtube.com/watch?v=AfzhRxtNapg.


SISU - Sistema de Seleção Unificada

As inscrições estão abertas!
De _ _ a _ _ de _ _ _ _ ...

Quem pode participar?

Para participar desse processo seletivo tenha em mãos o **número de inscrição e a senha no ENEM 201_**

Entre no SISU

Número de inscrição no ENEM 201_

Senha do ENEM 201_

Olá Aluno, acompanhe a sua inscrição no SISU. Click aqui para imprimir

Você ainda não se inscreveu em nenhuma das duas opções, escolha sua 1ª opção de curso.

Você não está inscrito na 1ª opção

Você não está inscrito na 2ª opção

Procure as vagas usando o tipo de pesquisa que preferir.
Quando a lista aparecer, clique no curso desejado para continuar

Abaixo estão descritas as modalidades de concorrência disponíveis para este curso.
Veja qual modalidade é a melhor para você e clique em Escolher esta modalidade.

CURSO SELECIONADO

Curso com para ingresso no 1° semestre

5 vaga(s) reservadas para ação afirmativa do tipo: Candidatos autodeclarados pretos, pardos ou indígenas, com renda familiar bruta per capita igual ou inferior a 1,5 salário mínimo e que tenham cursado integralmente o ensino médio em escolas públicas (lei n° 12.711/2012).

5 vaga(s) reservada para ação afirmativa do tipo: Candidatos com renda familiar bruta per capita igual ou inferior a 1,5 salário mínimo e que tenham cursado integralmente o ensino médio em escolas públicas (lei n° 12.711/2012).

Leia atentamente as informações abaixo, m
Confirmar minha inscrição nesta opção.

CURSO SELECIONADO

Curso com para ingresso no 1° semestre

O candidato optou por concorrer a ___________

INFORMAÇÕES ADICIONAIS DO CURSO

Os candidatos classificados poderão optar por Bacharelado (40% das vagas) ou Lic

Fig. 12. Web-pages emulating the Brazilian University course registration SISU. A student first logs to the system in order to select his first and second course options. When selecting any of the options, the student is forward to the third page where he searches for available courses which are fetched from a local database. Once a course is selected, the student may opt for a standard position or a position (fourth page) available from some quota scheme, *e.g.*, vacancies for poor families. Finally, a confirmation page is returned and the student may repeat this process to select his second option (second page).

cause more damage for nginx than for Apache is because the attacker can send many more requests to nginx than to Apache, due to the event-based nature of nginx where requests are served asynchronously. This increases the effect of the attack leading to more requests hitting the nginx than Apache.

f) CPU and Memory Consumption: The overall memory and CPU usage of SeVen was surprisingly low. SeVen consumed *less than 3% of CPU processing and less than 8 MB of memory* in all our experiments. Thus, SeVen is a lightweight application which could also be used in the same machine as the web-server. Indeed, we repeated some of the experiments by installing SeVen in the same machine as the web-server obtaining similar results as when SeVen is running in a separate machine.

The left-most graphic in Figure 11 depicts SeVen memory usage during an attack. The other two graphics illustrate how clients are able to be served when SeVen is activated and how attackers are able to deny service by monopolizing the web-server's resources when SeVen is turned off.

g) SeVen against a Pseudo-High-Rate Attack: We also tested SeVen against a large Slowloris attack. We attacked a web-server (Apache) with only 200 workers, but using SeVen and against a Slowloris attack with 2000 simultaneous connections against. That is, the attack created 10 times connections as the web-server can handle. This is not a usual Low-Rate ADDoS attack as a typical attack would use only 250 simultaneous connections. However, it is also not a high-rate ADDoS attack which floods a web-server with a much larger number of requests. We call it a Pseudo-High-Rate Attack.

Despite the great number of connections created by the attacker, SeVen was able to mitigate this pseudo-high-rate ADDoS attack obtaining a success rate of 96.3% of success rate with a TTS of 0.37 seconds. Moreover, SeVen did not consume much CPU, only 4.7%, nor memory, using less than 14 MB. Therefore, it seems possible to use SeVen to defend a web-server against larger coordinated attacks.

B. Quality of Service Test

In order to understand better the impact of SeVen in actual web experience, we implemented a web-site simulating the Brazilian University Registration web-site SISU[11] where high-school students select which University and which course they would like to register to. It consists of the five pages depicted in Figure 12. Each page contains forms with a 'next' button that forwards the user to the next page. The user is supposed to select his first and second options. For some forms, such as selecting which university, some queries are sent to a central database in order to display available options to users. In the first page, users log-in and their progression in filling out forms is saved whenever a user clicks the next button.

We simulated actual users by using robots. They simulate human interaction, navigating our five-page web-site filling out

[11]http://www.sisu2015.com/

Rate of 10 Robots per Minute

Attack	Tomcat		Tomcat with SeVen	
	Successful Robots	Failed Robots	Successful Robots	Failed Robots
Slowloris	0	1000 (2/998)	1000	0
HTTP POST	204	796 (21/775)	1000	0
Slowread	3	997 (18/979)	1000	0
Resurrected Slowloris	20	980 (12/968)	1000	0

Rate of 25 Robots per Minute

Attack	Tomcat		Tomcat with SeVen	
	Successful Robots	Failed Robots	Successful Robots	Failed Robots
Slowloris	0	1000 (4/996)	1000	0
HTTP POST	174	826 (27/799)	999	1 (1/0)
Slowread	121	879 (9/870)	1000	0
Resurrected Slowloris	4	996 (8/988)	1000	0

Rate of 50 Robots per Minute

Attack	Tomcat		Tomcat with SeVen	
	Successful Robots	Failed Robots	Successful Robots	Failed Robots
Slowloris	0	1000 (1/999)	1000	0
HTTP POST	104	896 (21/873)	997	3 (3/0)
Slowread	148	852 (6/846)	1000	0
Resurrected Slowloris	10	990 (3/987)	921	79 (0/79)

Fig. 13. Experimental results with SeVen using robots. In all experiments, we generated 1000 robots. For failed robots, we indicate which quality measure it failed indicated by the values N_R/N_{TG}, where N_R and N_{TG} specifies how many robots failed for, respectively, the criteria Max Number of Reloads and the Max Time to Receive a Server Response. For example, 20/100 specifies that 20 robots failed by needing to reload pages too many times and 100 for waiting too long for getting a page from the web-server.

the existing forms in the order of speed as usual users. Some robots may be faster than other robots, as their speed has a random factor. Moreover, robots may reload pages whenever it gets a response from the server stating that the service is unavailable. In this case, the robot will need to log-in again and proceed from the last saved state.

To evaluate how well our defense is, we monitored the following quality of service parameters reflecting realistic factors for a user to stop using a service:

1) **Max Number of Reloads (R):** Each robot keeps track of the total number of reloads per page. If the robot needs to reload a page many times, we consider that this robot failed to be served. We set $R = 3$;
2) **Max Time to Receive a Server Response (TG):** Each robot keeps track of the time that the web-server needs to respond a robot request, such as when the robot clicks the 'next' button. If the web-server is not able to respond after some time, then we consider that this robot failed to be served. We set $TG = 20$ seconds.

If a robot is able to fill out all the forms without violating any one of the quality measures above, we consider that this robot has been successfully served by the web-server.

We carried out a number of experiments using the web-server Apache Tomcat, version 8.0.14, which is a web-server based on Apache. We used Tomcat because it had all the libraries we needed for hosting our web-site and it was yet another web-server we could test SeVen on. Tomcat was configured with 400 workers, *i.e.*, the configuration of a medium size web-server. We considered three main scenarios depending on the client demand: 1) Low demand of legitimate clients accessing our web-page where we create only 10 robots per minute, 2) a medium rate, where we create 25 robots per minute and a 3) high rate where we create 50 robots per minute. In order to be able to perform these tests, we had to use 400 workers as opposed to 200 workers used in our Load Tests, as with 200 workers, Tomcat (without SeVen) is not even able to handle an incoming rate of 20 robots per minute.

Finally, each attack, including each colluding attacker in the Resurrected Slowloris attack, constructed 450 simultaneous connections, *i.e.*, a bit greater than the number of available workers (400) as expected.

Figure 13 summarizes our main experimental results using robots. As our experimental results demonstrate, when Tomcat was not used with SeVen, the great majority of robots failed to fill out the forms in our web-page. Most of them stayed waiting to get a response from the server. Some also received a response stating that web-server is unavailable and causing the robots to reload many times a web-page.

When Tomcat was used in conjunction with SeVen, on the other hand, in most of the cases, the robots were able to fill out all the forms. Even for a high rate of clients (50 robots per minute), more than 92% of all robots were successful in filling out all forms. We did not observe any problems for the robots to fill out all the form even when under a Slowread attack, which in load tests resulted in higher TTS.

VI. Related Work

Although there have been many studies on mitigating Transport Layer DDoS attacks, there are not many general solutions for Application Layer DDoS attacks. Most of the proposed solutions use the traffic pattern of the attack into consideration, such as the round trip of messages, location, and IP, or rely on some trust mechanisms [YFLL09] to infer when to filter packets or how suspicious a request is [RSU$^+$09]. There are models based on machine learning techniques, such as Neuro-Fuzzy [KS13], or models using Markov-Chains [XY09], or hard-wired into the defense [dA13]. Some of these defenses have been validated using real experiments on the network using a small number of attackers. However, none of them have been validated using such a wide range of attacks. Moreover, they assume that it is possible to distinguish client from attacker traffic, whereas we do not make this assumption here (although we discuss how this can be incorporated in our solution in Section IV). While in many situations it is possible to make such assumption, in general, the traffic of the attacker is indistinguishable from legitimate client traffic making Low-Rate ADDoS attacks hard to mitigate.

Park *et al.* [PITK15] have analyzed the effectiveness of Slowread Attack against off-the-shelf defenses. They showed that an ordinary Slowread DoS attack, *i.e.*, carried out by a single machine, can be prevented by using the the ModSecurity module for Apache which limits the number of connections from the same source IP Address. Then, they investigated a DDoS version of such attack, which two or more attackers collude and can maintain the attack success status by repeating it. They showed that this attack with colluding attackers is effective even if the target application limits the number of connections from the same source IP Address. The Resurrected Slowloris when carried out with a small botnet would also be vulnerable to defenses which limit the number of IP connections. (Notice that IP spoofing is not possible as the attacker needs to complete the SYN-ACK handshake.) However, as with Slowread, such a defense can be bypassed using a larger botnet.

There are also some more ad-hoc solutions which use more complicated systems such as Hardware Load Balancers [Fun12] and solutions that constraint the number of connections of clients [SIYL08], and Apache modules customized for mitigating Slowloris attacks [Mon13] . It seems possible to customize an attack, as we did with ReqTimeOut, to mitigate these defenses. Moreover, these solutions do not address other type of attacks besides Slowloris. Furthermore, we believe that these solutions can be used together with SeVen to strengthen even further the effectiveness of the defense.

There are also many companies that claim to have defenses for ADDoS attacks, but the mechanisms used to do so are not revealed, and therefore a detailed comparison with their mechanisms is not possible.

Finally, there is a number of blogs with guidelines on how to configure the parameters of web-servers for mitigating low-rate ADDoS attacks. Most of the guidelines suggest limiting the total number of connections per IP, as the ModSecurity module, or establishing timeouts, as the ReqTimeOut modules, or blacklisting/whitelisting IPs. There are some disadvantages with these solutions. Firstly, any way one sets these parameters would make the web-server resistant to some particular attack, but probably still vulnerable to adaptations of existing attacks as the attacker can easily determine the parameters used and orchestrate a different attack as we did with the Resurrected Slowloris attack. Secondly, it is difficult to set these parameters as they would depend on the hosted web-sites, user profile, etc. SeVen, on the other hand, is not constructed to mitigate a particular low-rate ADDoS attack and therefore, there are not many parameters to configure and still is able to mitigate a wide-range of different low-rate ADDoS attacks. Moreover, it seems possible to incorporate many of these guidelines, such as Black and White lists and time analysis, in the selective strategy used as described in Section IV-C.

VII. Conclusions and Future Work

We first demonstrated that the defenses deployed in existing web-servers, which are timeout-based, can be bypassed by a simple modification of the Slowloris attack where the Slowloris attack is reinitialized. We call this attack the Resurrected Slowloris attack. Similar modifications are also possible for the HTTP POST and Slowread attacks. Then we introduced SeVen, a new lightweight defense using a selective strategy for mitigating Low-Rate ADDoS attacks. We demonstrated by a great number of experiments that SeVen can successfully mitigate a wide-range of Low-Rate ADDoS attacks, namely, Slowloris, HTTP POST, Slowread and the Resurrected Slowloris attacks. The main reason why SeVen is effective is because SeVen is not tailored to mitigate a particular attack, but by randomly selecting and dropping requests, it increases the chance of mitigating attacks which attempt to keep a web-server's worker busy. We tested the efficiency of SeVen in mitigating the attack by load test and by using a more qualitative test with automated robots. In both cases, SeVen performed well.

As future work, we are further improving our implementation and testing it with other web-servers. We are implementing the alternative selective strategies discussed in Section IV-C. We have already implemented the Black and White list feature in SeVen. In order to protect multiple servers, we are incorporating load balancers strategies into SeVen. We have been successful to carry out tests when the web-servers being protected are equally configured, *i.e.*, same server type and configuration, using a simple weighted Round Robin strategy. However, we are understanding better how a selective strategy relates with different load balancer strategies. We are constructing the mathematical machinery to have a more foundational understanding of why SeVen is so effective. We expect that this mathematical model will help engineers in designing defenses with selective strategies such as SeVen. We are also investigating whether SeVen can be used to mitigate second order DoS attacks [ODL15].

Finally, we are collaborating with our industry partner RNP in the development of SeVen and using it to protect web-sites in Brazil. We are currently deploying SeVen to protect the web-services of different educational institutions.

References

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[dA13] Leandro C. de Almeida. Ferramenta computacional para identificação e bloqueio de ataques de negação de serviço em aplicações web. Master Thesis in Portuguese, 2013.

[DLFN16] Yuri Gil Dantas, Marcilio O. O. Lemos, Iguatemi Fonseca, and Vivek Nigam. Formal specification and verification of a selective defense for tdos attacks. In *11th International Workshop on Rewriting Logic and its Applications (WRLA)*, 2016.

[DNF14] Yuri Gil Dantas, Vivek Nigam, and Iguatemi E. Fonseca. A selective defense for application layer DDoS attacks. In *IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, pages 75–82. IEEE, 2014.

[Fun12] Funtoo. Slowloris dos mitigation guide – http://www.funtoo.org/Slowloris_DOS_Mitigation_Guide. 2012.

[KS13] P. Arun Raj Kumar and S. Selvakumar. Detection of distributed denial of service attacks using an ensemble of adaptive and hybrid neuro-fuzzy systems. *Computer Communications*, 36(3):303 – 319, 2013.

[KVF+08] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selectiveverification. In *INFOCOM*, pages 529–537, 2008.

[KVF+12] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selective verification: An efficient adaptive countermeasure to thwart dos attacks. *IEEE/ACM Trans. Netw.*, 20(3):715–728, 2012.

[LDNS16] Marcilio O. O. Lemos, Yuri Gil Dantas, Vivek Nigam, and Gustavo Sampaio. A selective defense for mitigating coordinated call attacks. In *34th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, 2016.

[mas10] Operation payback cripples mastercard site in revenge for wikileaks ban http://www.theguardian.com/media/2010/dec/08/operation-payback-mastercard-website-wikileaks. 2010.

[Mon13] Kees Monshouwer. mod_antiloris – http://sourceforge.net/projects/mod-antiloris/. 2013.

[ODL15] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 616–628, 2015.

[PCNF16] Tulio Pascoal, Joo Corrła, Vivek Nigam, and Iguatemi E. Fonseca. Um mdulo de defesa para ataques ddos na camada de aplicao usando estratgias seletivas. In *SBRT*, 2016. In Portuguese.

[PITK15] Junhan Park, Keisuke Iwai, Hidema Tanaka, and Takakazu Kurokawa. Analysis of slow read dos attack and countermeasures on web servers. In *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2015.

[ReQ14] ReQTimeOut. https://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html. 2014.

[RFC99] HTTP RFC. https://tools.ietf.org/html/rfc2616. 1999.

[RSU+09] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward W. Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, 2009.

[rudy13] r-u-dead yet. https://code.google.com/p/r-u-dead-yet/. 2013.

[Sie14] Siege. https://www.joedog.org/siege-home/. 2014.

[SIYL08] Mudhakar Srivatsa, Arun Iyengar, Jian Yin, and Ling Liu. Mitigating application-level denial of service attacks on web servers: A client-transparent approach. *ACM Trans. Web*, 2(3):15:1–15:49, July 2008.

[slo13a] slowloris. http://ha.ckers.org/slowloris/. 2013.

[Slo13b] Slowread. https://blog.qualys.com/securitylabs/2012/01/05/slow-read. 2013.

[Tes15] Slow HTTP Test. https://github.com/shekyan/slowhttptest. 2015.

[XY09] Yi Xie and Shun-Zheng Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw.*, 17(1):15–25, 2009.

[YFLL09] Jie Yu, Chengfang Fang, Liming Lu, and Zhoujun Li. A lightweight mechanism to mitigate application layer ddos attacks. In *ICST*, 2009.

[ZJT13] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys and Tutorials*, 15(4):2046–2069, 2013.