No More DoS? An Empirical Study on Defense Techniques for Web Server Denial of Service Mitigation

Marta Catillo[a,*], Antonio Pecchia[a], Umberto Villano[a]

[a] *Università degli Studi del Sannio, Benevento, Italy*

Abstract

Denial-of-Service (DoS) attacks are becoming increasingly common and undermine the availability of commonly used web servers. Even if DoS attacks cannot be rendered completely harmless, ready-to-use defense modules and solutions to mitigate their effect are highly beneficial for site administrators. Unfortunately, there is a lack of measurement studies that explore the pros and cons of common DoS web server defense modules in order to understand their limitations and to drive practitioners' choices.

This paper presents an empirical study of the ubiquitous Apache web server, with an assessment of two well-known pluggable defense modules and an enlargement technique that provides the server with additional resources. Measurements are based on a mixture of flooding and slow DoS attacks. The experimentation shows that, in spite of the large availability of pluggable security modules that can be usefully deployed in practice, there is not a bulletproof defense solution to mitigate the DoS attacks in hand. The findings of our analysis can be useful to support the deployment of proper defense mechanisms, as well as the development of robust and effective solutions for DoS protection.

Keywords: Denial of Service, web server, defense, enlargement, availability.

---

[*] Corresponding author (phone: +39-0824-305805)

Email addresses: `marta.catillo@unisannio.it` (Marta Catillo), `antonio.pecchia@unisannio.it` (Antonio Pecchia), `villano@unisannio.it` (Umberto Villano)

Preprint submitted to Journal of Network and Computer Applications February 15, 2023

1. Introduction

Denial-of-Service (DoS) attacks pose a relevant threat to commonly used web servers. Although DoS attacks have been well-known for years –as early as 1993 Needham pointed out that DoS are "incontrovertible" main threats (Needham, 1993)– variations and evolutions of DoS have spread over time, making them dangerous, if not disruptive, even for modern networked environments. Indeed, according to a recent report, attackers intensified their DoS activity in the second half of 2020[1]. The nature of a DoS attack retains its initial definition: *denying a service*. In this respect, attackers aim to hit the victims by probing for weaknesses and trying out different attack vector combinations. DoS attacks can be potentially harmful and unexpected for any network infrastructure. This has led to the rapid spread of a variety of **attack patterns** that implement different malicious behaviors. In its most classic form, during a DoS attack an attacker intentionally **floods** the victim server by means of many service requests with the aim of slowing it down, or even interrupting, its normal activity. In this context, the server is forced to allocate resources to process a multitude of requests, so that it fails to provide service to legitimate users. In the last few years, DoS attacks evolved into a "second" generation, called **slow** DoS attacks (Sikora et al., 2019). These types of attack use low-bandwidth approaches that exploit application-layer vulnerabilities. Given the plethora of versatile and efficient attack tools available on the Internet, it is extremely simple to perform both flooding and slow DoS attacks, and their setup takes place in a short time. Although the perspective of the attacker community has broadened considerably, also that of the defender is potentially based on solutions that, at least in theory, aim at providing a "bulletproof" environment.

Nowadays **web servers** are massively used by both organizations and customers (Ferris and Farrell, 2003). This dependency has increased the load of web servers: accessing servers *quickly*, *continuously*, and *accurately* is the main concern for developers and users. DoS attacks undermine this balance by attempting to make web servers unavailable to clients. Notwithstanding the substantial and valuable body of research on DoS defense, **state-of-the-practice** artifacts and techniques available to practitioners for hardening a given web server are much more simplistic. In fact, well-known web servers,

---

[1] `https://www.helpnetsecurity.com/2021/03/18/ddos-attacks-pandemic/`

such as the Apache[2] web server, provide **modules** that extend the core server functionality for special purposes. Many modules are recommended for hardening web server installations: they can be "plugged" into the web server in order to achieve a *layered defense* strategy. We observe that "ready-to-use" defense modules and solutions that can be used in practice by system administrators do not seem particularly effective against DoS attacks. While there are plenty of tech blogs and references that provide practical guidance for installation and functional testing of defense modules, to the best of our knowledge there is a lack of direct measurements to explore *pros* and *cons* of common DoS defense modules.

In this paper we propose an **empirical study** of well-established DoS defense techniques in the context of the Apache web server. Our study is based on direct measurements during a variety of DoS attacks against a victim server in a controlled testbed. Experiments capitalize on a balanced mixture of **DoS attacks** –emulated through well-established public tools– that leverage both (i) flooding activities and (ii) slow attacks, which capitalize on the intrinsic design of the HyperText Transfer Protocol (HTTP). Overall, the attacks elicit different outcomes by the web server depending on the specific defense in place. As for the **defense**, we consider `evasive` and `reqtimeout`, i.e., two well-known Apache modules intended for DoS, DDoS (distributed DoS) and brute-force attacks mitigation, and a resource *enlargement* approach implemented by adjusting the configuration of the web server. It is worth noting that all the defenses have been tested against all the attacks in hand in order to infer a coverage matrix aiming to provide a clear picture on the limitations of the techniques assessed and in order to drive practitioners' choices.

Our study is based on a **holistic approach** to measurement, where we collect and look at metrics and data at different layers including application, operating system and network. It is worth noting that assessing a given defense technique is a complex matter and an open problem in the literature. As for any well-hardened system, there might be a strong-enough attack – depending on the specific magnitude and duration– capable of subverting a previously-proven valid defense. In consequence, finding a "meaningful" metric that unifies alternative methods for defense evaluation is challenging. In this respect, we supplement traditional, client-side, service metrics with

---

[2] `https://httpd.apache.org`

quantitative insights on the user-perceived availability and the **effectiveness** of the defense. More importantly, we dig into server-side access and error logs, and CPU usage; where needed, additional insight is gained by analyzing the packets sent from the attacker to the victim server. Overall, the use of different data sources beside the traditional metrics allowed us to gain a deeper understanding on the specific defense technique. As a byproduct of this work, we released a public dataset (Catillo et al., 2021a) encompassing network flow records collected in our controlled testbed; the dataset can be accessed through our institutional webpage[3].

The key outcomes and findings of our study, with respect to the attacks and defenses in hand, are:

- none of the defenses was truly effective to provide "full" protection from any of the attacks assessed in this study; rather, defenses either (i) mitigated the attacks for a short timeframe (i.e., in the range of 1 to 2 *minutes* depending on the loss of service an administrator is willing to tolerate) or (ii) were able to recover the server only after a given period of complete unavailability of the server;

- each defense technique mitigated only one out of the entire set of attacks assessed in this study; for example, according to our findings, `reqtimeout` –partially effective against a specific instance of slow DoS attack– could be successfully subverted by relying on a different implementation of the same type of attack, while resource *enlargement* was somewhat effective only in the case of flooding attacks;

- some of the attacks were mitigated by none of the defenses assessed in this study, which means that in spite of the large availability of security modules that can be usefully deployed in practice, there is not a "bulletproof" defense solution against DoS attacks; more importantly, tech blogs lack a clear picture on the coverage practitioners can expect from a given defense module once deployed in production.

The findings of this paper should be contextualized with respect to the attacks and server in hand. Magnitude and duration of the DoS attacks is tuned in order to surely impact a *baseline* web server installation (i.e., default

---

[3] `http://idsdata.ding.unisannio.it/`

configuration and no defense in place); then, attacks are executed –at the *same* magnitude and duration– against the web server hardened through a given defense technique. This is done to ensure the same attack conditions before and after defense. Different tuning of the attacks, i.e., weaker or stronger, may reflect into different values of the evaluation metrics. The findings of our study provide a better understanding of the capability of the modules and their potential limitations in a production environment.

The rest of this paper is organized as follows. Section 1 presents related work on DoS attacks measurement, detection and defense. Section 3 describes the controlled testbed, defense modules and evaluation metrics adopted in this study. Section 4 describes the attacks, the experimental procedure and characterizes the attacks against the baseline server installation. Section 5 and 6 discuss the findings on the mitigation of flooding and slow attacks, respectively. Section 7 summarizes lessons learnt, limitations and threats to validity of our work, while Section 8 concludes the paper and provides future research directions.

2. Related Work

2.1. Denial of Services: measurements and detection

The escalation of DoS attacks has pushed security experts to work in order to face their effects. Over the years DoS attacks have changed significantly in both strength and intelligence. Therefore, a number of approaches have been proposed to measure and detect DoS efficiently.

In order to assess the impact of a DoS attack, its severity and the effectiveness of a potential defense, precise, quantitative and comprehensive metrics are required. In the literature, many solutions have been proposed to evaluate the impact of DoS attacks. They aim to compare goodput (the throughput of "useful" data) without attack, under attack, and with defense. Mostly often, percentage of failed transactions, Transmission Control Protocol (TCP) retransmission time out, goodput, mean time between failures and average response time have been used as the key parameters for analyzing attack symptoms; however, there are no benchmarks (Mirkovic et al., 2006) that allow to evaluate effective metrics. The authors of (Catillo et al., 2021b) analyze DoS traffic from different public intrusion detection datasets and measure the impact under different configurations of a victim server. Results indicate that a tuned-up configuration of the server can mitigate the impact of a DoS attack; the authors also show the partial ineffectiveness of the attacks in the presence of defense mechanisms and suitable server configurations. The work (Mirkovic et al., 2007) proposes the use of the percentage of failed transactions (PFT) as a metric to measure DDoS impact. They define a threshold-based model, which is application specific. When a measured value exceeds the threshold, it indicates poor quality of service. Furthermore, since the transaction duration depends on the volume of data being transferred and on network load, the absolute duration threshold cannot be set. *Server timeout* has been used as a metric in (Ko et al., 2006); however, collateral damage in terms of legitimate traffic drop is not disclosed.

DoS vulnerabilities are described in (Deng et al., 2019). In (Giralte et al., 2013) the authors use three types of analyzers to detect DDoS attacks. They describe a normal user behavior in a statistical way and aggregate packets with the same source address and protocol type. Flow count, flow size and flow rate of the HTTP protocol are used to compute statistics for each user, who is mapped to the source IP address. Any user with a statistical behavior different from the standard one is identified as suspicious. However, the proposed technique is ineffective against distributed slow HTTP DoS attacks, due to the lower amount of attack traffic that allows it to evade detection. (Aiello et al., 2014) describe a method that monitors the number of packets received by a web server in different time horizons for anomaly detection. If the number of packets received in the interval exceeds a predefined threshold, the interval is considered as containing attack traffic. As for the detection of slow DoS attacks, in (Sikora et al., 2019) is described a solution that analyzes, processes, and aggregates the data packets in order to dynamically detect the anomalies. It is worth pointing out that although these techniques are often effective, monitoring the number of packets can cause high false positive rates, because bursts of traffic can be produced even in legitimate scenarios.

In recent years, a substantial body of research deals with the detection of DoS attacks using **machine** and **deep learning techniques**. Several state-of-the-art anomaly detectors have spread in the literature, along with more classic detection techniques. For example, in (Adi et al., 2017) is described a machine learning approach that aims to detect DoS attacks. A machine-learning-based DoS detection system is presented in (de Lima Filho et al., 2019), where the authors use an inference-based method, obtaining a 96% detection rate. The work (Qu et al., 2019) proposes the statistic-enhanced directed batch growth self-organizing mapping (SE-DBGSOM), a recent model based on self-organizing maps (SOM), for DoS attack detection. The proposal is evaluated on the CICIDS2017 dataset. In order to solve the challenges in DoS detection, Nguyen et al. (Nguyen et al., 2018) propose an intrusion detection system that leverages a convolutional neural network model; the authors evaluate the performance of the proposed method using the UNSW-NB15 and NSL-KDD datasets. The results are valuable as compared to the state-of-the-art DoS detection methods. Deep learning models are playing an increasingly important role and have become a promising direction (Liu and Lang, 2019). For example, authors of (Catillo et al., 2022) propose a method to detect different classes of anomalies including DoS attacks. Detection is addressed by means of system log analysis and a semi-supervised deep autoencoder: the proposed approach, called AutoLog, achieves up to 99% recall and 98% precision across different system logs and types of attacks.

2.2. Denial of Service: defenses and countermeasures

Available defense mechanisms for DoS are surveyed in (Zargar et al., 2013). They include defense mechanisms against network/transport-level DoS flooding attacks or defense mechanisms against application-level DoS flooding attacks, to mention some examples. Conventional defense approaches analyze the connection request rate for a particular client (Kang et al., 2015); if this is determined to be above a pre-established threshold, the client is marked as an attacker. However, the technique is ineffective for slow DoS, as shown in (Aiello et al., 2014). As a matter of fact, whereas attacks at the communication layer typically require flooding the victim with a continuous stream of packets, attacks at the application layer send relatively few packets with suitable timing. Furthermore, in some cases, even a legitimate user could generate multiple requests that are processing-intensive without leading to an attack (Nagaratna et al., 2009). The authors of (Beitollahi and Deconinck, 2012) present a taxonomy of DoS/DDoS attacks and defense mechanisms. In particular, the countermeasures against attacks are broadly classified into *proactive*, *reactive* and *survival* mechanisms. Proactive mechanisms aim to detect an attack before it can hit the victim. Reactive mechanisms detect and mitigate the attack after that the victim actually encounters a DoS/DDoS attack. Survival mechanisms, instead, equipe the possible victim system with resources that may be sufficient to serve legitimate users in case of attack. In (Aamir and Zaidi, 2013) an in-depth analysis of DDoS countermeasures is provided. It focuses on strengths of each defense technique and also considers the countermeasures that can be taken against each defense mechanism from the attacker's point of view. In particular, the DDoS defense mechanisms are classified on the basis of the position of defense (*source-end*, *victim-end*, *distributed* and *core-end* defense techniques), and also on the basis of the reaction time (*proactive*, *reactive* and *survival* techniques). A taxonomy of DDoS defense mechanisms can be found in (Chang, 2002). The authors state that, with respect to start-end of a DDoS, there are three lines of defense against the attack: attack prevention and preemption (*before* the attack), attack detection (*during* the attack), and attack response (*during* and *after* the attack). As shown in (Gupta et al., 2012), defense at the Internet Service Provider level can be useful in case of DDoS attacks originating from specific networks. In particular, the authors provide two statistical metrics (traffic volume and flow), which are used as parameters to detect traffic anomalies.

The discussion of the work presented in Section 2.1 and 2.2 is supplemented by Table 1, which summarizes key aspects including datasets, main contribution and category, i.e., measurements (M), detection (D) and countermeasures (C). It is worth pointing out that –different from the discussion

Table 1: Overview of the work presented in Section 2.1 and 2.2 by datasets, main contribution and category, i.e., measurements (M), detection (D) and countermeasures (C).

paper	**dataset(s)**	**main contribution**	**M**	**D**	**C**
(Chang, 2002)	*N/A*	Defense techniques **(taxonomy)**			✓
(Ko et al., 2006)	lab-made attacks	Server timeout evaluation	✓		
(Mirkovic et al., 2006)	lab-made attacks	DoS benchmark suite	✓		
(Mirkovic et al., 2007)	lab-made attacks	Application QoS requirements	✓		
(Nagaratna et al., 2009)	lab-made attacks	Encryption and filtering		✓	
(Beitollahi and Deconinck, 2012)	*N/A*	Defense techniques **(review)**			✓
(Gupta et al., 2012)	lab-made attacks	Combined statistical-based approach			✓
(Aamir and Zaidi, 2013)	*N/A*	Defense techniques **(survey)**			✓
(Zargar et al., 2013)	*N/A*	Defense techniques **(survey)**			✓
(Aiello et al., 2014)	lab-made attacks	Spectral features analysis		✓	
(Kang et al., 2015)	KDD'99	Real-time connection monitoring			✓
(Adi et al., 2017)	Non-public data	Naïve Bayes, decision tree, JRip, SVM		✓	
(Nguyen et al., 2018)	UNSW-NB15, NSL-KDD	Convolutional neural network		✓	
(de Lima Filho et al., 2019)	CIC-DoS, CICIDS2017 CSE-CIC-IDS2018, Non-public data	Random forest		✓	
(Deng et al., 2019)	lab-made attacks	DoSDefender architecture		✓	
(Liu and Lang, 2019)	*N/A*	Machine learning **(survey)**		✓	
(Qu et al., 2019)	CICIDS2017	Artificial neural network		✓	
(Sikora et al., 2019)	lab-made attacks	Packet monitoring		✓	
(Catillo et al., 2021b)	CICIDS2017, ISCXIDS2012, NDSec-1, MILCOM2016, SUEE2017	DoS traffic replay	✓		
(Catillo et al., 2022)	Non-public data, HADOOP, BG/L	Deep autoencoder		✓	

above– papers in Table 1 are arranged by year of publication and not by category; the order of appearance of each paper in the table is different from that observed in the text.

2.3. Our contribution

Web servers are one of the most vulnerable services to DoS attacks in a production environment. Moreover, the use of a default web server configuration can open the way for attackers, and so it needs "hardening" techniques, which should make it more complex to accomplish a successful attack. Beside regular updating and patching a given web server, it is essential to configure it for better security performance. One of the key strengths of modern web servers is their modular structure. This enables introducing additional modules that make it possible to build extensions to mitigate a number of security threats. In (Moustis and Kotzanikolaou, 2013) the authors implement custom web server modules at different layers of the communication protocols. They limit the number of connections and monitor the connections per IP address. A web server with such configuration can really mitigate an attack. However, if the attack is launched by an increasing number of bots, there is a noticeable delay in the server response; as such, a massive attack scenario can affect the performance of the server. In (Hirakawa et al., 2016) the authors propose and evaluate a defense method against distributed slow HTTP DoS attack by disconnecting the attack connections selectively and focusing on the number of connections for each IP address and the duration time. The defense solution is effective against distributed slow HTTP DoS attacks.

Differently from these papers, our **empirical study** aims to analyze "ready-to-use" solutions and modules that can be used in practice by system administrators. Although the use of these modules is highly recommended, studies on "real-life" defense modules are lacking. While there is a substantial and valuable body of research on DoS defense –as for many of the papers referenced in Section 2.2– we take a different perspective by addressing the gap between the sophistication of research proposals for DoS defense and the oversimplification of the techniques that are concretely available to practitioners. Through the analysis of two defense modules and a resource enlargement technique, we provide a methodology that aims to *holistically* measure the impact of different DoS attacks and the effectiveness of the defense. The result is a comprehensive analysis and a set of measurements, which outline security facets and may drive the development of more resilient solutions for

DoS protection. To the best of our knowledge, there are no similar studies in the literature.

3. Testbed, Defenses and Evaluation Metrics

Our analysis is based on direct performance measurements of a victim server during DoS attacks performed in a controlled testbed. We collect a variety of service metrics to gain insight into the impact of the attacks in case of *no* and *with defense*. In the following we describe the experimental testbed, defenses assessed in this study and the service metrics adopted.

3.1. Experimental testbed

Experiments are conducted with a private infrastructure hosted by a datacenter at the University of Sannio. The experimental testbed capitalizes on our previous work (Catillo et al., 2020) and consists of three Ubuntu 18.04 LTS nodes, equipped with Intel Xeon E5-2650V2 8 cores (with multithreading) 2.60 GHz CPU and 64 GB RAM within a local area network (LAN), described in the following and using the naming shown in Figure 1.

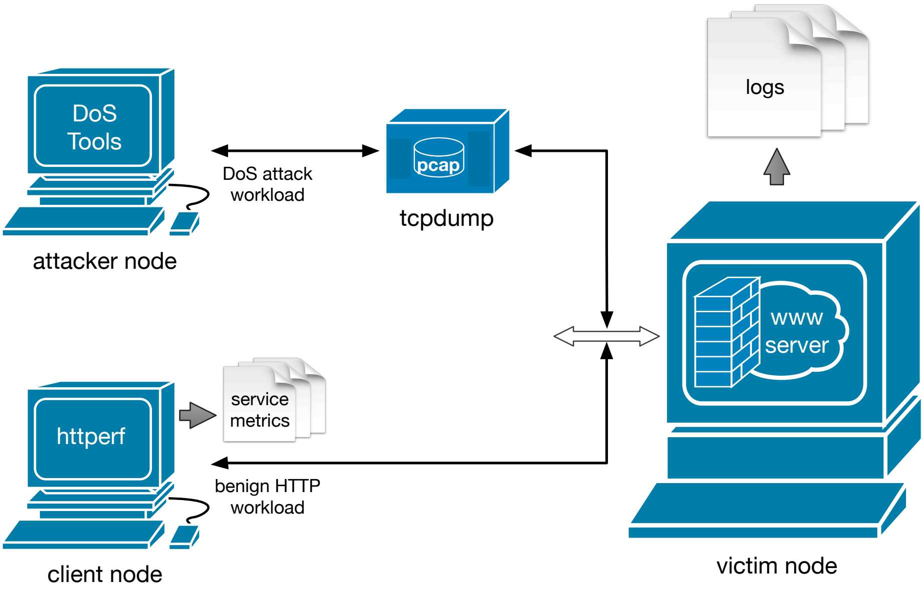


Figure 1: Experimental testbed.

The **"victim" node** hosts an installation of Apache 2.4. This web server is a significant case study, given its widespread use. It can fit a wide-range of websites, ranging from personal blogs to websites that serve millions of users; moreover, it is open source and cross-platform. As discussed later, the Apache web server supports a variety of pluggable **modules** –including security-related capabilities– that can be enabled by adjusting the configuration of the *baseline* server installation. In our study we address the adoption of the `evasive` and `reqtimeout` modules and a resource *enlargement* technique for defense purposes, as discussed in Section 3.2.

As for the remaining components in Figure 1, the **"attacker" node** generates the DoS traffic intended to disrupt server operations. To this aim, we use several state-of-the-art attack tools, which are described in Section 4. Finally, the **"client" node** hosts `httperf`[4], which is a widely-used workload generator. This tool makes it possible to set a desired level of workload by regulating different parameters, such as *total connections*, *connections per second* and *requests per connection* in order to trigger the normative HTTP requests, which aim to emulate the *benign* workload by a legitimate client. During the experiments, the web server is exercised with both DoS traffic and benign workload. We use native metrics produced by `httperf` and a set of derived metrics –detailed in Section 3.3 – to monitor the web server and to measure the impact of an attack.

3.2. Defense modules and techniques

The Apache web server is modular in design and many extensions may be added to the baseline server to provide additional capabilities, in the form of *modules*. Among the wide range of modules available through online repositories[5], it is possible to find a number of **security modules**. In particular, in this work we deal with `reqtimeout` and `evasive` module. Nevertheless, many additional modules are available to extend the core functionality of the web server for special purposes.

For example, the `modsecurity2` module acts as a sort of intrusion detection and prevention system (IDPS). Just like a regular signature-based IDPS, it relies on a set of rules related to known attack patterns available from free or pay-per-use repositories. These patterns may be used to check

---

[4] `https://github.com/httperf/httperf`

[5] `http://httpd.apache.org/docs-2.0/mod/`

different sections of an incoming request, according to the type of attack and to the underlying protocol vulnerabilities they refer to. Similarly, the `fail2ban` module also acts as an intrusion prevention tool. It detects various attacks based on system logs and automatically initiates prevention actions. In particular, it performs periodic checks in the log files in order to search for specific messages, e.g. repeated ssh access errors, and blocks the source IP by creating a rule in iptables. The `modqos`, instead, is a quality of service (QoS) module, which implements control mechanisms that can provide different priority to different requests. It determines which requests should be served primarily in order to avoid resource oversubscription. Specifically for slow DoS attacks, on the other hand, are the `modantiloris` and `modnoloris` modules. The former prevents new connections from the same IP address after the connection count of the IP exceeds a configurable limit; the latter is also based on `modantiloris`, but it runs an IP ban check on a (default) 10-second timer instead of on every request.

The use of security modules is not the unique way to set up a more "robust" Apache web server than the basic one. There are also *concrete* defense techniques which simply consist in increasing the service resources (e.g., memory and sockets) that might be depleted under attack (Beitollahi and Deconinck, 2012). This strategy –typically known as resource *enlargement*– may help the mitigation process, possibly allowing it to gain additional time to face the attack. In the following we describe the defenses assessed in this study.

3.2.1. `reqtimeout` module

The module can protect from DoS attacks, such as slow attacks, and is typically enabled by default in the baseline server after installation from the standard Ubuntu repository. This means that its disablement requires explicit changes of the configuration by the user. The `reqtimeout` module is used to set –according to the environment and domain where the web server is deployed– a time-out for client HTTP requests received by the Apache server. In particular, since HTTP requests consist of header and body, `reqtimeout` makes it possible to set different time limits for the two parts. Furthermore, `reqtimeout` can be used to set the minimum allowed transfer rate for data received from the client-side. If the client fails to meet the time frame limit and the minimum transfer rate for sending the data, the connection is dropped and the server responds with a `408 REQUEST TIMEOUT` error.

For our experiments we configured `reqtimeout` according to the instruc-

```
<IfModule reqtimeout_module>
    RequestReadTimeout header=20-40, minrate=500
    RequestReadTimeout body=10, minrate=500
</IfModule>
```

Figure 2: `reqtimeout` configuration.

tions in the Apache docs[6]. Our configuration is shown in Figure 2. In particular, the first `RequestReadTimeout` directive gives the client a 20-second maximum time frame for sending the first bytes of the HTTP request header. If the client has sent the first segment of the request, then the directive checks the transfer rate to be at least 500 bytes/s. If the client does not send the complete request in 20 seconds, the time-out is incremented by one second for each received 500 bytes up to a maximum of 40 seconds. A similar rule is applied for the body portion of the HTTP request. In this case we omit the upper time limit, thus setting a strict time limit of 10 seconds.

3.2.2. `evasive` *module*

This is a consolidated defense module intended to protect a server from DoS, DDoS and brute-force attacks. It is mainly conceived to mitigate those attacks, such as `hulk`, that try to make a server unavailable by consuming its resources through a huge amount of requests. The module stores all incoming and previous IP addresses and Universal Resource Identifiers (URIs) in a table, which is used to lookup if a specific request should be allowed or not. In particular, the module creates an internal and dynamic hash table of IP addresses and URIs, and denies any single IP address that: (i) requests the same page more than a few times within the past 1 second, (ii) makes more than 50 concurrent requests on the same Apache child process per second, and (iii) makes any request while temporarily blacklisted. If any of the above conditions is true, a `403` response is sent and the IP is blacklisted for a configurable amount of time (10 seconds is the default value). The configuration of `evasive` relies on the following directives:

- `DOSHashTableSize`: specifies the size of the hash table. Increasing the size will provide faster performance by decreasing the number of

---

[6] `https://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html`

iterations required to get the record, but at the expense of consuming more memory;

- `DOSPageCount`: indicates the number of identical requests to a specific page (or URI) a visitor can make over the `DOSPageInterval` (typically one second). Once the threshold for that interval has been exceeded, the IP address of the client will be added to the blocking list.
- `DOSSiteCount`: specifies the total number of requests for any object that is allowed to be made by the same client per `DOSSiteInterval` (typically one second). Once the threshold has been exceeded, the IP address of the client will be added to the blocking list.
- `DOSPageInterval`: the interval during which the `DOSPageCount` threshold has not been exceeded. The default value is one second.
- `DOSSiteInterval`: the interval during which the `DOSSiteCount` threshold has not been exceeded. The default value is 1 second.
- `DOSBlockingPeriod`: is the amount of time (in seconds) that a client will be blocked if it is added to the blocking list. During this time interval all requests from the blocked client will result in a 403 response from the web server. The timer is set to 10 seconds by default and it is reset for every subsequent request.

The `evasive` configuration considered for our experiments is shown in Figure 3. At any time, the module can be seamlessly enabled or disabled by acting on the configuration and re-starting the web server.

```
<IfModule mod_evasive>
    DOSHashTableSize 3097
    DOSPageCount 2
    DOSSiteCount 50
    DOSPageInterval 1
    DOSSiteInterval 1
    DOSBlockingPeriod 10
</IfModule>
```

Figure 3: `evasive` configuration.

3.2.3. Resource enlargement

As mentioned above, **resource enlargement** aims to increase the capacity of the victim to serve requests. To this aim, we edited the default configuration of the web server in order to boost its capacity and multi-threading capability. Apache extends his modular design to the most basic functions of a web server, such as multi-processing. In particular, the server is provided with a selection of Multi-Processing Modules (MPMs) which are responsible for binding to network ports on the machine, accepting requests, and dispatching children to handle the requests. Currently, the Apache server has three stable MPM modes: *prefork*, *worker* and *event*. They represent the evolution and the development of Apache. Prefork MPM implements a non-threaded web server. It launches multiple Apache child processes. Each Apache child process handles one connection at a time. The worker mode, compared with the prefork one, uses a hybrid mode of multi-process and multi-threading. It generates multiple Apache child processes similar to prefork. Each Apache child process runs many threads, and each thread handles one connection at a time. Finally, the event MPM mode, introduced in Apache 2.4, is pretty similar to worker MPM but it is designed for managing high loads. It allows more requests to be served simultaneously by passing off some processing work to supporting threads. The server can be conveniently customized for the needs of the particular site. For example, sites that need a great deal of scalability may prefer a threaded MPM like worker or event, while sites requiring stability or compatibility with older software may adopt a prefork mode. In order to make the enlargement operation in our testbed –in *event* mode– we considered the following Apache MPM common directives:

- `StartServers`: number of child server processes created on startup;
- `MinSpareThreads`: minimum number of idle threads to handle request spikes;
- `MaxSpareThreads`: maximum number of idle threads to handle request spikes;
- `ThreadLimit`: sets the maximum configured value for `ThreadsPerChild` for the lifetime of the Apache httpd process;
- `ThreadsPerChild`: number of threads created by each child process. The child creates these threads at startup and never creates more;

Table 2: Configuration parameters of the web server.

directive	**default configuration**	**enlarged configuration**
`StartServers`	2	4
`MinSpareThreads`	25	25
`MaxSpareThreads`	75	75
`ThreadLimit`	64	128
`ThreadsPerChild`	25	50
`MaxRequestWorkers`	150	200

- `MaxRequestWorkers`: sets the limit to the number of simultaneous requests that will be served.

For our experiments we edited the default configuration of the Apache web server in order to improve its capacity and multithreading capability. The result is an **enlarged configuration**, whose settings are shown in the rightmost column of Table 2.

3.3. Evaluation metrics

DoS attacks and defense techniques are usually investigated by researchers and practitioners via live experiments in controlled environments. However, even if the deployment of attacks and defenses is pretty straightforward, the evaluation of a defensive solution is not a trivial task. In fact, finding a uniform method for defense evaluation and a valuable mechanism to compare different defense techniques is challenging. In this context, it would be beneficial to adopt a comprehensive and quantitative *metric of distinction* of the defense. Even if the available literature on DoS presents several different methods being used to assess defenses (Mirkovic et al., 2009), these offer little possibility of an objective and commensurable comparison. Many current approaches to evaluate the quality of a defense technique involve the collection of well-known "legacy" metrics as throughput, request-response delay or allocation of resources. Sometimes the metrics are based on a combination of these legacy indexes. We adopt a holistic approach to measurement, where we collect and look at metrics and data at different layers including application, operating system and network.

Legacy (traditional) metrics. These metrics are collected by running `httperf` at the "client" node, which is used to continuously probe the operational status of the server. While collecting the metrics, we set a request timeout of 10 s to avoid that `httperf` could hang waiting for responses to requests that might never be received in the case of attack. We focus on the following metrics obtained from `httperf`:

- **reply rate** or **throughput (T)**: HTTP requests accomplished by the server within the time unit, measured in *reqs/s*;
- **mean response time (MRT)**: mean time taken to serve an HTTP request measured in milliseconds (*ms*);
- **successful** and **failed requests**: number of HTTP requests successfully handled by the server (`2xx` response) and failed requests.
- **connection errors**: number of connection errors experienced by the client.

User-perceived availability and effectiveness. It is worth pointing out that DoS attacks may impact all network services, not only web servers. The amount of degradation of quality of service (QoS) perceived by the user is heavily dependent on the application at hand. In the literature there exist several attempts to define thresholds for common application QoS requirements (see for example (Mirkovic et al., 2007)). However, to consider the effect of DoS attacks at application level is out of the scope of this paper. In the following, we will limit ourselves to measure the impact of the attack on web user experience by a lower-level index, the **User-Perceived Service Availability** (UPA). The UPA is a low-level specialization to web client-server interactions of the *availability*, known by various names in the literature (Mikic-Rakic et al., 2005), (Shao et al., 2009), and computed as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions over a period of time. In our context, the *UPA is defined as the number of HTTP requests that receive a 2xx response to the total number of HTTP requests issued to the server*. Though not linked to any particular user network application, the UPA is a synthetic indicator of the decay of server performance due to an attack. Furthermore, it paves the way to make considerations on the effectiveness of a defense by evaluating its ability to maintain an acceptable level of availability under attack. In particular, we perform two tests with the same timing and the same attack conditions, with and without defense. We define **effectiveness** *the difference between the times the availability falls for the first time under a given threshold level with and without defense, respectively.* In particular, in the experimentation that will be described next, we will consider the following fractions of full UPA as thresholds: 0.9, 0.95, 0.99, which will be denoted for the sake of brevity 1 *nine*, 1.5 *nines* and 2 *nines*, respectively.

Network-level and server-side data. In addition to the above mentioned indicators, which are measured at the client-side, in our experiments we also collect information at network level and at the server-side. At network level, all the **data packets** transmitted on the testbed network are stored and made available for later examination. At the server-side, in order to highlight possible CPU resource depletion due to attacks, we measure **CPU usage** at the server node by means of `atop`[7], a well-known Linux performance monitor, which can log and report the activity of all server processes. Furthermore, the logs of the web server (i.e., `access.log` and `error.log`) produced during the experiments are stored in order to analyze how the defense modules handle malicious requests.

4. Attacks and Baseline Experiments

This section presents DoS attacks and related tools we used to conduct the experiments; more importantly, we demonstrate the effectiveness of each attack against the *baseline* server installation, i.e., default configuration and "no defense" module in place, by measuring the UPA and additional user-perceived metrics at the client node, such as the number of successful/failed HTTP requests, reply time of the server and connection errors under attack. The measurements presented in this section form the basis for assessing the effectiveness of the defense techniques.

4.1. Attacks and tools

The experimental campaign is based on a mixture of **DoS attacks**, i.e., *flooding* and *slow* attacks. Each of the tools used for the experiments – described in the following– can potentially circumvent existing defense techniques:

---

[7] `https://linux.die.net/man/1/atop`

- `hulk`: it is conceived as a flooding attack aiming to overwhelm the victim server by a massive amount of HTTP requests. Its strength is the ability to generate a *unique* HTTP GET request with randomly generated headers and URL parameters. Thereby attack patterns cannot be easily detected. The attack leverages different strategies, such as the *obfuscation of the source client*. In this context the attack is accomplished by sending different patterns of attack requests that can obfuscate the source client for each request. For our experiments we use the *grafov* `hulk` Python script[8]. It is the most popular and well-consolidated `hulk` implementation.

- `TCP flood`: it is a popular DoS attack tool, which allows to conduct a further form of flooding attack. The attacker sends TCP connection requests in order to lock the ports available at the server and to cause incapability to accept legitimate connections from benign clients. For our experiments we use a GitHub `TCP flood` script[9]. It is a Python script that allows launching a `TCP flood` attack against the victim host.

- `slowhttptest`: it allows implementing a *slow* rate and *low* volume of traffic, which is difficult to detect by standard DoS detection systems. In particular, this kind of DoS attack uses *low-bandwidth* approaches, which leverage a weakness in the management of TCP fragmentation of the HTTP protocol: it requires HTTP messages to be completely received before they are processed. The `slowhttptest` tool[10] allows to generate *slow* DoS attacks. For our experiments we use it in the "slowloris" mode, which allows sending incomplete HTTP requests to the target server.

- `slowloris`: it is a well-known Python attack script[11] that allows implementing *low* and *slow* DoS attacks. It implements a *slow header* attack by sending incomplete HTTP requests (i.e., without ever ending the header) and by establishing a number of connections to the target server. Connections are kept "alive" as long as possible by means of

---

[8] `https://github.com/grafov/hulk`

[9] `https://github.com/Leeon123/TCP-UDP-Flood`

[10] `https://tools.kali.org/stress-testing/slowhttptest`

[11] `https://github.com/gkbrk/slowloris`

keep-alive headers on all the connections at 15 *s* intervals; if the server closes a connection, this is restored by the script.

Selection of the attacks. Attacks are purposely chosen in order to cover in a comprehensive way existing DoS strategies aiming to consume all the resources of a victim server (e.g., sockets and CPU time) by capitalizing on network, transport and application layer protocol vulnerabilities. The most common DoS family encompasses the attacks that try to spawn a large number of requests (the so-called *flooding*) so as to exhaust the server resources, making it unable to serve legitimate requests: `hulk` and `TCP flood` belong to this family, and are widely used as reference attacks in the literature. On the other hand, the so-called *slow* attacks leverage potential HTTP weaknesses by means of purposely-formatted messages, without generating a large number of messages and consuming excessive bandwidth. We chose the widely used `slowhttptest` and `slowloris` as representative of this second category. It is worth noting that both `slowhttptest` and `slowloris` implement slow attacks. We use both, because the results produced by the two tools in case of defense –shown in the following– are different, thus allowing more general claims. Another typical DoS attack is `SYN flood`, which capitalizes on a weakness of the TCP handshake. `SYN flood` is not considered in this study because modern operating systems, in particular Linux, use the *syn-cookie* technique (Fontes et al., 2006) as a countermeasure: this is typically applied as a default by the kernel, which makes the most part of real-world servers protected by the attack with no need for any additional defense.

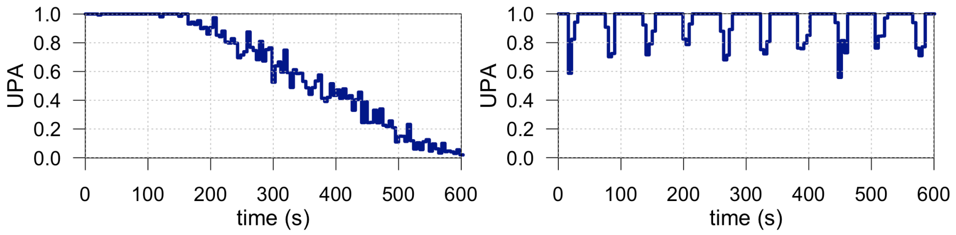
4.2. Execution of the attacks

Each attack tool presented above is launched against the baseline web server while capturing metrics and data at different levels; it is worth noting that one attack per time is performed in the context of a single experiment.

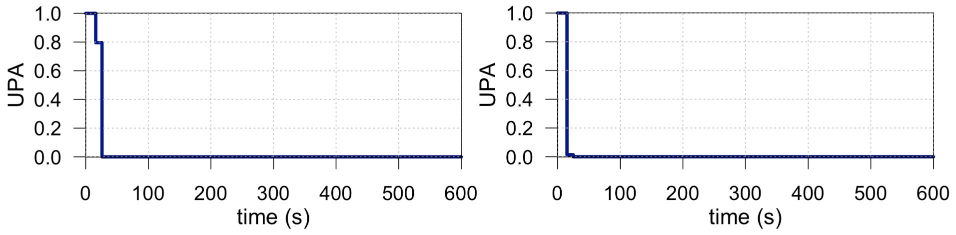
Experimental procedure. The duration of an experiment is set to 600 *s*, a time interval which is long enough to collect a large sample of service metrics generated by `httperf`. The attack starts at t=15 *s* since the beginning of the experiment and the web server is exercised with a client load of L=1,000 *req/s* by `httperf`, a rate that can be safely handled at UPA=1.0 (i.e., no loss of legitimate requests). In consequence, any point where UPA<1.0 points to the presence of a DoS attack, because in our controlled testbed the only source of legitimate activity is the “client” node. At the end of each experiment we (i) store measurement data and logs for subsequent analysis, (ii) clear the logs of the web server, such as `access.log` and `error.log` (iii) stop the workload generator, attack scripts and the web server. We reboot the nodes of the testbed to ensure independent experimental conditions prior to the next experiment.

Figure 4 shows the UPA measured at the client node during the execution of the attacks in hand. Attacks are run in case of “no defense” at the server-side; we will discuss the mitigation offered by different defense techniques for flooding and slow attacks in Section 5 and 6, respectively. Interestingly, the attacks cause a variety of outcomes by the victim web server. For example, we note either a progressive UPA degradation for `hulk` (Figure 4a) or periodic drops caused by `TCP flood` (Figure 4b). On the other hand, slow attacks are characterized by the typical “on-off” behavior, as shown in Figure 4c and 4d, which means that UPA drops sharply from 1.0 to 0.0 in a few seconds.

Table 3 shows additional evaluation metrics collected at the client node; as the UPA above, metrics are collected with the default configuration and “no defense” at the server-side. The total number of HTTP requests at-



(a) `hulk` (b) `TCP flood`



(c) `slowhttptest` (d) `slowloris`

Figure 4: Impact of each attack on the UPA during the “no defense” experiments.

Table 3: Number of HTTP requests, MRT and connection (conn.) errors measured at the client node while the server is under attack.

	HTTP requests				**max**	**conn.**
	succeeded (by MRT)			**failed**	**MRT**	**errors**
	[0, 1] *ms*	(1, 10] *ms*	(10, +∞] *ms*		(*ms*)	
`hulk`	353,331	5,596	196	240,877	14.2	2,877
	58.89%	*0.93%*	*0.03%*	*40.15%*		
`TCP flood`	558,962	0	0	36,038	0.4	456
	93.94%	*0%*	*0%*	*6.06%*		
`slowhttptest`	3,979	0	0	296,021	0.8	2,962
	1.33%	*0%*	*0%*	*98.67%*		
`slowloris`	0	0	69	299,931	54.9	3,000
	0%	*0%*	*0.03%*	*99.97%*		

tempted by the client –while the server is under attack– is broken down by *succeeded* and *failed*; moreover, successful requests are further divided into three ranges based on the MRT taken to complete the requests. For example, `hulk` causes the failure of 240,877 HTTP requests attempted by the client, i.e., 40.15% of the total requests; on the other hand, 353,331 HTTP requests, i.e., 58.89% of the total, succeed with MRT within [0, 1] *ms*. A similar finding is noted for `TCP flood`, where the requests that succeed, i.e., 93.94% of the total requests, are accomplished within [0, 1] *ms*. With respect to our experimental setting and duration of flooding attacks, HTTP requests either succeed within a "reasonable" time or fail. As for slow attacks, almost all the HTTP requests attempted by the client fail, i.e., 98.67% and 99.97% for `slowhttptest` and `slowloris`, respectively. Another interesting outcome is noted for the number of connection (conn.) errors shown by the rightmost column of Table 3: `hulk` causes almost the same number of connection errors of slow attacks, i.e., 2,877, but it is less effective to make HTTP requests fail. This finding is consistent with Figure 4, where it can be noted that `hulk` takes around 600 *s* to make the server unavailable in our setting.

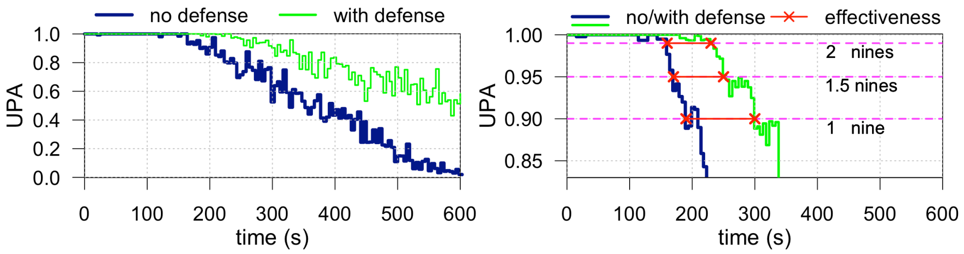
5. Mitigation of Flooding Attacks

We start the analysis of **flooding attacks** by discussing the results obtained by launching `hulk` and `TCP flood` after having hardened the web server either by means of `evasive` or by resource *enlargement*. Attacks are run according to the experimental procedure presented in Section 4.2, with the addition that the server is hardened with a given defense technique before the beginning of the experiment. It is worth noting that `reqtimeout` had no mitigation effect against flooding attacks in the testbed in hand; in consequence, `reqtimeout` will be addressed in Section 6, in the context of slow attacks.

5.1. Effectiveness of the `evasive` *module*

Figure 5a shows the **UPA** –"with defense" series– measured at the client node during the `hulk` attack when the web server is defended by means of the `evasive` module. In order to appreciate the effect of the mitigation, the data series from Figure 4a is plotted again in Figure 5a, and indicated as the "no defense" series. In both cases the attack starts at $t=15$ s after the beginning of the experiment. Figure 5a shows that the "with defense" UPA is higher than "no defense", which means the `evasive` module can mitigate the `hulk` attack to some extent.

The detail is shown in Figure 5b, which presents the **effectiveness** of the `evasive` module at various UPA thresholds as ×-*edged* horizontal segments. Different from Figure 5a, UPA is "smoothed" by replacing each original UPA value at time i since the beginning of the experiment (u_i) with $\frac{u_{(i-1)}+u_i+u_{(i+1)}}{3}$, i.e., the average of u_i and its preceding/subsequent values in the series. This is done to mitigate sporadic UPA fluctuations in order to obtain a better evaluation of the effectiveness. Figures corresponding to each segment in Figure 5b, i.e., length, start (t_S) and finish (t_F) data points, are reported in Table 4, where each row relates to a given segment.



(a) availability

(b) effectiveness

Figure 5: Impact of `hulk` on the UPA for *no defense* and with the `evasive` module.

Table 4: Effectiveness of `evasive` against `hulk`.

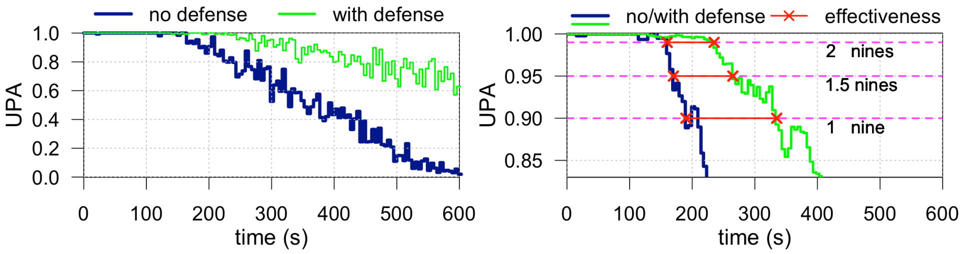
UPA	*nines*	**effectiveness**	
		length	(start; finish)
0.99	*2*	**70 *s***	(t_S=160 *s*; t_F=230 *s*)
0.95	*1.5*	**80 *s***	(t_S=170 *s*; t_F=250 *s*)
0.90	*1*	**110 *s***	(t_S=190 *s*; t_F=300 *s*)

As shown in Figure 5b, `hulk` takes around 160 *s* (t_S) to make UPA less than 0.99 in case of "no defense"; on the other hand, `hulk` takes 230 *s* (t_F) to make UPA=0.99 after the enablement of `evasive`. *With respect to the magnitude and duration of the attack, and server in hand*, the `evasive` module assures additional 70 *s* (i.e., t_F-t_S) UPA=0.99 (2 *nines* availability) when compared to its corresponding "no defense" experiment. Similarly, as reported in the bottom row of Table 4, `hulk` takes around 190 and 300 *s*, i.e., t_S and t_F, respectively, to make UPA less than 0.9 (1 *nine* availability) in case of "no defense" and "with defense": in consequence, effectiveness is 110 *s*.

The most striking outcome is that `hulk` takes much longer for impacting the UPA when the defense is enabled; however, in spite of the user-perceived mitigation effect, the UPA remains strongly affected anyway. Whilst the `evasive` module does not guarantee long-term protection from `hulk`, it can contribute to saving a desired UPA level for an additional time that depends on the magnitude of the attack and the network/server configuration.

5.2. Effectiveness of resource enlargement

Resource *enlargement* is considered a viable means to mitigate DoS attacks. Firstly, we assess this practice in the context of `hulk`. Figure 6 –"with defense" series– shows the UPA measured at the client node during the `hulk` attack when the web server is *enlarged* with respect to the default configuration; again, the data series from Figure 4a is reproduced in Figure 6a as "no defense". Similar to `evasive`, resource *enlargement* is able to mitigate `hulk` to some extent: in fact, the UPA "with defense" is higher than the corresponding "no defense" experiment. Regarding the effectiveness, we observe that resource *enlargement* can slightly improve the metrics obtained with the `evasive` module, as it can be noted from the length of the ×-*edged*



(a) availability

(b) effectiveness

Figure 6: Impact of `hulk` on the UPA for *no defense* and *enlargement*.

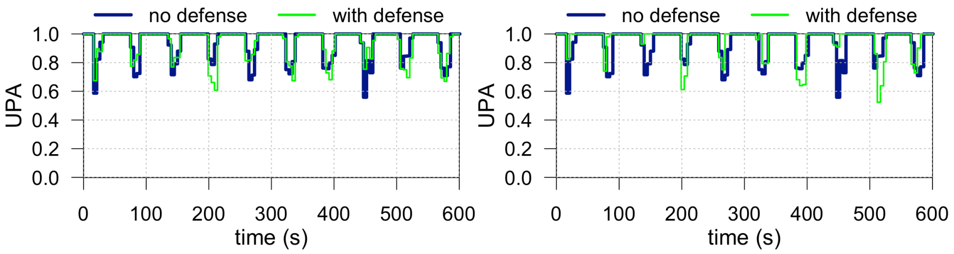
Table 5: Effectiveness of resource *enlargement* against `hulk`.

UPA	*nines*	**effectiveness**	
		length	(start; finish)
0.99	*2*	**75** $\boldsymbol{s}$	(t_S=160 s; t_F=235 s)
0.95	*1.5*	**95** $\boldsymbol{s}$	(t_S=170 s; t_F=265 s)
0.90	*1*	**145** $\boldsymbol{s}$	(t_S=190 s; t_F=335 s)

segments in Figure 6b and the corresponding numbers in Table 5, where each row corresponds to one segment. For example, the effectiveness of resource *enlargement* at UPA=0.99 (2 *nines* availability) is 75 s, which is 5 s more than `evasive`; improvement with respect to `evasive` goes up to 35 s at UPA=0.9 (1 *nine* availability). As noted for `evasive`, resource *enlargement* does provide a mitigation effect; however, it is not a long-term defense from DoS attacks.

5.3. Analysis of `TCP flood`

As for `TCP flood`, neither `evasive` nor *enlargement* were able to assure any form of mitigation. Figure 7a and 7b –"with defense" series– show the UPA measured at the client node after `evasive` and *enlargement*, respectively; in both the cases we reproduce Figure 4b as "no defense" for the sake of better visual comparison. `TCP flood` is able to bring the UPA below the 0.9 threshold regardless of the defense in place. This finding indicates that a flooding activity, which evades the simplistic threshold-based detection scheme of `evasive`, can easily affect the victim server.



(a) *no defense* and `evasive` (b) *no defense* and `enlargement`

Figure 7: Impact of `TCP flood` on the UPA.

```
—– access.log —–
[01/Aug/2020:22:44:31 +0200] ”GET /index.html?ZBWRUMRX=ANINEAZY HTTP/1.1”
200 11192 ”http://engadget.search.aol.com/search?q=PPQOBOV” ”Mozilla/4.0
(compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2;.NET CLR
2.0.50727; InfoPath.2)”
```

Figure 8: Response to a Hulk DoS HTTP request (*no defense*).

5.4. Server-side insights

Analysis is supplemented by a closer look at data and metrics collected at the server. We use the logs of the web server, i.e., `access.log` and `error.log`, to gain a better insight into the UPA caused by `hulk` in “no defense” and “with defense” conditions. As mentioned above, `hulk` *floods* the server through malicious HTTP requests that consist of random URL query string, user agent and referee. Figure 8 shows an instance of `hulk` request extracted from the `access.log` in a “no defense” experiment. Surprisingly, with no specific defense in place, the server fulfills the malicious request by returning the `200` (OK) HTTP status code (enclosed in a box in Figure 8), as for any legitimate request: this behavior causes a tremendous waste of CPU at the server node.

Figure 9 shows the CPU usage at the server node during different experiments. On average, the CPU usage measured when the server is exercised solely by the legitimate client workload, i.e., •-*marked* (no attack) series in Figure 9, is around 11%; on the other hand, the average CPU usage is around 37% until 400 *s* –most of the duration of the attack– in case “no defense”, i.e., △-*marked* series.

The effect of hardening the server by means of the `evasive` module is

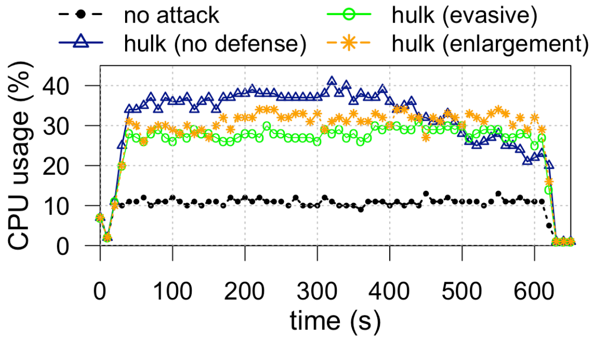


Figure 9: CPU usage measured at the server node during a legitimate experiment (*no attack*) and different instances of `hulk`.

```
-- access.log --
[01/Aug/2020:22:27:50 +0200] "GET /index.html?TOIHJNNI=ZXGMI HTTP/1.1"
403 459 "http://www.google.com/?q=QILIYIIDL" "Mozilla/5.0 (X11; U;Linux
x86_64; en-US; rv:1.9.1.3) Gecko/20090913 Firefox/3.5.3"
-- error.log --
[Sat Aug 01 22:27:50.377578 2020] [evasive20:error] [pid /*omitted*/]
[client 192.168.111.65:54508] client denied by server configuration:
/var/www/html/index.html, referer: http://www.google.com/?q=QILIYIIDL
```

Figure 10: Response to a Hulk DoS HTTP request (*with defense*).

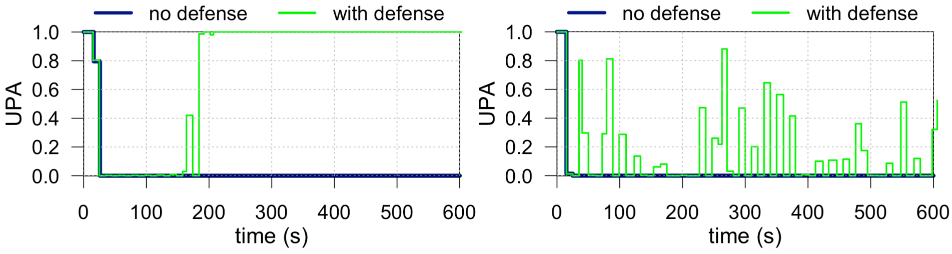
twofold. First, the way malicious requests are handled by the server. Figure 10 shows how a malicious HTTP request by `hulk` is tracked by the logs of the server after enabling `evasive`. It can be noted that the malicious request is now **forbidden** access to the resource with the `403` HTTP status code (enclosed in a box in Figure 10); moreover, it raises a corresponding "client denied" notification in the `error.log`. Differently from the absence of defense, a malicious request is thus aborted, instead of being handled successfully; however, it still requires busy cycles from the server in order to log the request and to "flag" it as forbidden. This can be noted in Figure 9, ○-*marked* series, which provides the server-side CPU usage caused by `hulk` with the `evasive` module: on average, it is around 28%. Although lower than the "no defense" case mentioned above, i.e., 37%, the CPU usage is significantly higher than the no attack series, which means that the overall contribution of a massive flooding activity will eventually affect the server also in case of defense.

Another interesting outcome is noted for resource *enlargement.* It is clear here that the "apparent" improvement of UPA and effectiveness of resource *enlargement* over `evasive` –documented in Section 5– is obtained at the cost of a higher CPU usage. Figure 9, $*$-*marked* series, provides the server-side CPU usage caused by `hulk` under resource *enlargement*: on average, it is 31%, in comparison to the above mentioned 28% of `evasive`. As a further remark, in case of resource *enlargement* the server accomplishes all the HTTP requests by `hulk` with the `200` (OK) status code, as in absence of defense, which indicates a bad handling of malicious requests.

6. Mitigation of Slow Attacks

The `reqtimeout` module is a default solution in the Apache web server to face slow attacks. Figure 11a and 11b show the UPA at the client node under `slowhttptest` and `slowloris`, respectively. For each figure, the "no defense" UPA series –originally presented in Section 4– is superimposed here to that obtained after enabling `reqtimeout`, i.e., "with defense" series. It can be noted that in case of "no defense" both the attacks have the same impact on the server. In fact, they cause the UPA to drop sharply from 1.0 to 0.0 (i.e., total unavailability) in just a few seconds. More importantly, UPA remains stuck at 0.0 through all the remainder of the experiment, which means the web server is clearly denied to legitimate clients.

Differently from the "no defense" case, the enablement of `reqtimeout` leads to different outcomes depending on the specific attack, i.e., "with defense" series in Figure 11. As for `slowhttptest` in Figure 11a, the UPA is



(a) `slowhttptest`

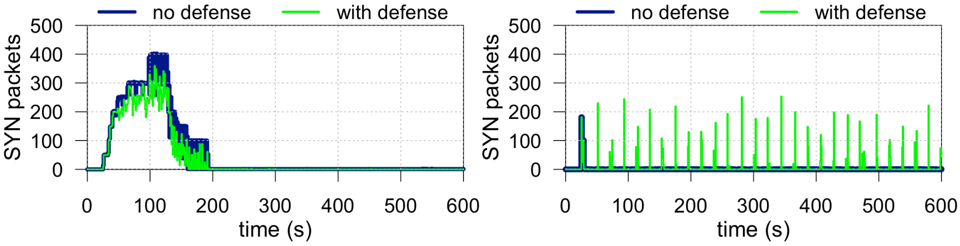
(b) `slowloris`

Figure 11: Impact of slow attacks on the UPA for *no defense* and after defense by means of the `reqtimeout` module.

restored to 1.0 at around t = 180 *s*: the server becomes available again to the client node. Overall, it seems that `reqtimeout` allows to successfully recover from a slow attack if not for a transitory period of unavailability; however, this is not a general finding for slow attacks. In fact, Figure 11b –obtained for `slowloris`– provides a different picture: in spite of `reqtimeout`, the UPA is very unstable and it switches between sporadic spikes and very low values, which means the server is mostly denied to legitimate clients.

In order to explore the limitations of `reqtimeout`, we investigate the **low-level behavior** –in terms of transmitted packets– of the attacks in hand. To this aim, Figure 12 shows the number of `SYN` *packets per second* sent from the attacker node to the server in "no defense" and "with defense" experiments. It is worth noting that `SYN` packets are generated by a node attempting to start a TCP connection. As for `slowhttptest`, the behavior of the attack is almost the same regardless of the defense, as shown by the data series in Figure 12a. The attack generates up to 402 `SYN` *packets per second* in "no defense" (359 packets per second in "with defense") and keeps going until t = 200 *s*. As shown above in Figure 11a, this is enough to bring down the server in "no defense"; however, the attack has a transitory impact when `reqtimeout` is enabled. On the other hand, `slowloris` is able to *adjust* its behavior depending on the absence/presence of the defense. Figure 12b indicates that `slowloris` in "no defense" consists of a single burst of `SYN`s at t = 25 *s*; different from this behavior, the same attack –once enabled `reqtimeout`– generates periodic bursts of `SYN` packets in order to keep subverting the defense successfully.

With respect to the magnitude and duration of the attacks in hand, it can



(a) `slowhttptest`

(b) `slowloris`

Figure 12: Number of `SYN` *packets per second* generated by each slow attack.

be reasonably claimed that `reqtimeout` can defend from *short* slow attacks, whose scope is quite limited in time; however, it is not effective against *persistent* slow attacks. In this respect, the usefulness of `reqtimeout` is opaque, because it provides scarce, if not none, mitigation: even with the defense module on, a well-crafted slow attack is successful.

7. Lesson Learnt, Limitations and Threats to Validity

Table 6 summarizes the effectiveness of each defense technique for each attack assessed in this study. We mark by ✓ all combinations where the attack is successful; the attack is deemed to be *mitigated* if the defense module was beneficial, in that it led to reasonably higher UPA than the corresponding "no defense" experiment. Table 6 indicates that two attacks –namely `TCP flood` and `slowloris`– are mitigated by none of the defenses assessed in this study; more importantly, each defense technique is able to mitigate just one of the attacks. Notwithstanding the substantial and valuable body of research on DoS defense, "ready-to-use" solutions and modules that can be used in practice by system administrators for hardening a given web server do not seem particularly effective. For example, according to our findings, `reqtimeout` can be successfully subverted depending on the specific implementation of the slow attack. The scientific literature has proposed and proven a variety of sophisticated solutions; however, they do not seem to have yet converged into pluggable or "ready-to-use" artifacts. In fact, there is a gap between the sophistication of research proposals for DoS defense and the oversimplification of artifacts and techniques concretely available to practitioners.

Table 6: Effectiveness of each defense technique by attack (✓ indicates the attack is successful; the attack is *mitigated*, otherwise).

	defense technique			
attack	"no defense"	`evasive`	enlargement	`reqtimeout`
`hulk`	✓	*mitigated*	*mitigated*	✓
`TCP flood`	✓	✓	✓	✓
`slowhttptest`	✓	✓	✓	*mitigated*
`slowloris`	✓	✓	✓	✓

Evaluating DoS defenses is a complex matter, and it depends on various factors, such as the underlying network capacity and topology, the nature of the attacks and the legitimate activity handled by the server under assessment. The findings of this paper must be contextualized with respect to the attacks and server in hand. Magnitude and duration of the attacks is tuned in order to surely affect the availability of the server in case of "no defense"; then, attacks are executed –at the *same* magnitude and duration– against the web server hardened through a given defense technique. This is done to ensure the same attack conditions before and after defense; in fact, a different tuning of the attacks, i.e., weaker or stronger, may reflect into different values of the evaluation metrics. We are aware that we made some simplifications in our study; however, the findings will reasonably hold in a more complex production environment of "real-life" servers, which reflects the ever-evolving sophistication of the attacks, heterogeneous and non-stationary workloads.

As for any measurement study, there may be concerns regarding the validity and generalizability of the results. We discuss them based on the four aspects of validity listed in (Wohlin et al., 2000).

Construct validity. The study builds around the intuition that "ready-to-use" modules and solutions for hardening a given web server provide limited defense from DoS attacks. This *construct* has been investigated in the context of a widely-used web server, three defense techniques and four attacks implementing a mixture of flooding and slow activity. The study is supported by extensive experimentation based on the analysis of consolidated metrics and data collection at different levels, i.e., application, operating system and network level.

Internal validity. The results and key findings of this paper are based on direct measurement experiments, where we analyze UPA, server-side logs and CPU usage and network packets. Attacks have been simulated by means of widely-accepted tools in cyber security experimentation. For example, `hulk`, `slowhttptest` and `slowloris` are used in many network-based public intrusion datasets, such as the CIC collection. The use of such diverse data and attack tools aims to mitigate internal validity threats.

Conclusion validity. Conclusions have been inferred by assessing three independent defense techniques and the sensitivity of a key metric, such as the UPA, with respect to each attack. More importantly, we made sure that each attack was successful in the baseline "no defense" experiment. We present an extensive discussion of the results. The key findings of the study are consistent across the attacks, and this provides a reasonable level of confidence on the analysis.

External validity. The steps of our analysis can be applied to other web servers, DoS tools and defenses. Nowadays, there exist many software repositories and plenty of attack tools, which make our approach definitively feasible in practice. In fact, in this paper we successfully ported the experiments across three defenses and four attacks to mitigate external validity threats. We are confident that the experimental details provided in the paper –also pertaining to the configurations and defenses of the web server– would support the replication of our study by future researchers and practitioners.

8. Conclusion

DoS attacks against modern web servers are becoming increasingly common. Even if DoS attacks cannot be rendered completely harmless, simple and "ready-to-use" solutions to mitigate their effect would be highly beneficial for site administrators. Unfortunately, our initial assessment indicates that "ready-to-use" web server defense modules are partially un-effective as confirmed by our experiments and measurements.

In this paper, we presented our experimentation on the ubiquitous Apache web server and tested two well-known pluggable defense modules along with an enlargement technique that tries to provide the server with additional resources. Our results show that none of the modules is capable of reasonably mitigating the effect of all the DoS attack tools used in our tests, tools that can be easily found on the Inter and require no particular skill to be used; moreover, some attacks cannot be mitigated at all.

Though limited to a single –albeit widely used– web server, our study and the holistic measurement methodology adopted here pave the way to further investigation on the topic and, hopefully, to the development of more robust and effective tools for DoS protection that could be readily used by system administrators. Besides the extension of our work to other web servers (notably, `ngix`[12]), defense modules and attack tools, a further point to be explored in our future research is the quantification of the effect of DoS attacks on other network services than web servers. This will allow us to evaluate the effect of the on-going attacks perceived by network users in a more complete way than as described in this paper.

---

[12] `https://www.nginx.com`

Acknowledgment

This work has been partially supported by CINI within the "BOTITS - Boosting OT and IT Security" project funded by Silicon Valley Community Foundation.

References

References

Aamir, M. and Zaidi, M. A. (2013). A survey on DDoS attack and defense strategies: from traditional schemes to current techniques. *Interdisciplinary Information Sciences*, 19(2):173–200.

Adi, E., Baig, Z., and Hingston, P. (2017). Stealthy denial of service (DoS) attack modelling and detection for HTTP/2 services. *Journal of Network and Computuer Applications*, 91:1–13.

Aiello, M., Cambiaso, E., Mongelli, M., and Papaleo, G. (2014). An on-line intrusion detection approach to identify low-rate DoS attacks. In *Proc. International Carnahan Conference on Security Technology*, pages 1–6. IEEE.

Beitollahi, H. and Deconinck, G. (2012). Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Commun.*, 35(11):1312–1332.

Catillo, M., Del Vecchio, A., Ocone, L., Pecchia, A., and Villano, U. (2021a). USB-IDS-1: A public multilayer dataset of labeled network flows for IDS evaluation. In *Proc. International Conference on Dependable Systems and Networks Workshops*, pages 1–6. IEEE.

Catillo, M., Pecchia, A., Rak, M., and Villano, U. (2021b). Demystifying the role of public intrusion datasets: a replication study of DoS network traffic data. *Computers & Security*, 108:102341.

Catillo, M., Pecchia, A., and Villano, U. (2020). Measurement-based analysis of a DoS defense module for an open source web server. In Casola, V., De Benedictis, A., and Rak, M., editors, *Testing Software and Systems*, pages 121–134. Springer.

Catillo, M., Pecchia, A., and Villano, U. (2022). AutoLog: Anomaly detection by deep autoencoding of system logs. *Expert Systems with Applications*, 191:116263.

Chang, R. (2002). Defending against flooding-based distributed denial-of-service attacks: a tutorial. *IEEE Communications Magazine*, 40(10):42–51.

de Lima Filho, F. S., Silveira, F. A. F., de Medeiros Brito Júnior, A., Vargas-Solar, G., and Silveira, L. F. (2019). Smart detection: An online approach for DoS/DDoS attack detection using machine learning. *Security and Communication Networks*, 2019:1574749.

Deng, S., Gao, X., Lu, Z., Li, Z., and Gao, X. (2019). DoS vulnerabilities and mitigation strategies in software-defined networks. *Journal of Network and Computer Applications*, 125:209–219.

Ferris, C. and Farrell, J. (2003). What are web services? *Commun. ACM*, 46(6):31.

Fontes, S. M., Hind, J. R., Narten, T., and Stockton, M. L. (2006). Blended SYN cookies. US Patent 7,058,718.

Giralte, L. C., Conde, C., de Diego, I. M., and Cabello, E. (2013). Detecting denial of service by modelling web-server behaviour. *Computers & Electrical Engineering*, 39(7):2252–2262.

Gupta, B. B., Misra, M., and Joshi, R. (2012). An ISP level solution to combat DDoS attacks using combined statistical based approach. *International Journal of Information Assurance and Security*, 3:102–110.

Hirakawa, T., Ogura, K., Bista, B. B., and Takata, T. (2016). A defense method against distributed slow HTTP DoS attack. In *Proc. International Conference on Network-Based Information Systems*, pages 152–158. IEEE.

Kang, B., Kim, D., and Kim, M. (2015). Real-time connection monitoring of ubiquitous networks for intrusion prediction: a sequential KNN voting approach. *International Journal of Distributed Sensor Networks*, 11(10):1–10.

Ko, C., Hussain, A., Schwab, S., Thomas, R., and Wilson, B. (2006). Towards systematic IDS evaluation. In *Proc. DETER Community Workshop*, pages 20–23.

Liu, H. and Lang, B. (2019). Machine learning and deep learning methods for intrusion detection systems: a survey. *Applied Sciences*, 9(20):4396.

Mikic-Rakic, M., Malek, S., and Medvidovic, N. (2005). Improving availability in large, distributed component-based systems via redeployment. In *Component Deployment*, pages 83–98. Springer.

Mirkovic, J., Arikan, E., Wei, S., Thomas, R., Fahmy, S., and Reiher, P. (2006). Benchmarks for DDoS defense evaluation. In *Proc. Military Communications Conference*, pages 1–10. IEEE.

Mirkovic, J., Fahmy, S., Reiher, P., and Thomas, R. K. (2009). How to test DoS defenses. In *Proc. Cybersecurity Applications Technology Conference for Homeland Security*, pages 103–117. IEEE.

Mirkovic, J., Hussain, A., Wilson, B., Fahmy, S., Reiher, P., Thomas, R., Yao, W., and Schwab, S. (2007). Towards user-centric metrics for denial-of-service measurement. In *Proc. Workshop on Experimental Computer Science*, pages 1–14. ACM.

Moustis, D. and Kotzanikolaou, P. (2013). Evaluating security controls against HTTP-based DDoS attacks. In *Proc. International Conference of Information Intelligence Systems and Applications*, pages 1–6. IEEE.

Nagaratna, M., Prasad, V. K., and Kumar, S. T. (2009). Detecting and preventing IP-spoofed DDoS attacks by encrypted marking based detection and filtering (EMDAF). In *Proc. International Conference on Advances in Recent Technologies in Communication and Computing*, pages 753–755. IEEE.

Needham, R. M. (1993). Denial of service. In *Proc. ACM Conference on Computer and Communications Security*, page 151–153. ACM.

Nguyen, S., Nguyen, V., Choi, J., and Kim, K. (2018). Design and implementation of intrusion detection system using convolutional neural network for DoS detection. In *Proc. International Conference on Machine Learning and Soft Computing*, pages 34–38. ACM.

Qu, X., Yang, L., Guo, K., Ma, L., Feng, T., Ren, S., and Sun, M. (2019). Statistics-enhanced direct batch growth self-organizing mapping for efficient DoS attack detection. *IEEE Access*, 7:78434–78441.

Shao, L., Zhao, J., Xie, T., Zhang, L., Xie, B., and Mei, H. (2009). User-perceived service availability: a metric and an estimation approach. In *Proc. International Conference on Web Services*, pages 647–654. IEEE.

Sikora, M., Gerlich, T., and Malina, L. (2019). On detection and mitigation of slow rate denial of service attacks. In *Proc. International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*, pages 1–5. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction.* Kluwer Academic.

Zargar, S. T., Joshi, J. B. D., and Tipper, D. (2013). A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys & Tutorials*, 15:2046–2069.