LoRDAS: A Low-Rate DoS Attack against Application Servers

Gabriel Maciá-Fernández, Jesús E. Díaz-Verdejo, Pedro García-Teodoro, and Francisco de Toro-Negro

Dpt. of Signal Theory, Telematics and Communications - University of Granada
c/ Daniel Saucedo Aranda, s/n - 18071 - Granada, Spain
gmacia@ugr.es, jedv@ugr.es, pgteodor@ugr.es, ftoro@ugr.es

Abstract. In a communication network, there always exist some specific servers that should be considered a critical infrastructure to be protected, specially due to the nature of the services that they provide. In this paper, a low-rate denial of service attack against application servers is presented. The attack gets advantage of known timing mechanisms in the server behaviour to wisely strike ON/OFF attack waveforms that cause denial of service, while the traffic rate sent to the server is controlled, thus allowing to bypass defense mechanisms that rely on the detection of high rate traffics. First, we determine the conditions that a server should present to be considered a potential victim of this attack. As an example, the persistent HTTP server case is presented, being the procedure for striking the attack against it described. Moreover, the efficiency achieved by the attack is evaluated in both simulated and real environments, and its behaviour studied according to the variations on the configuration parameters. The aim of this work[1] is to denounce the feasibility of such attacks in order to motivate the development of defense mechanisms.

1 Introduction

The problem of the denial of service attacks [1] still remains unsolved. Although multiple solutions for the defense and response against these attacks have been proposed [2], the attackers have also evolved their methods, which have become really sophisticated [3].

For carrying out a DoS attack, two main strategies are used: either using a specially crafted message that exploits a vulnerability in the victim, or sending it a flooding of messages that somehow exhaust its resources. These last attacks are called DoS flooding attacks.

Although a flooding attack implicitly implies the sending of a high rate of traffic to the victim, in the last years two special DoS flooding attacks, characterized by the sending of low-rate traffic, have been reported: the Shrew attack [4] and the low-rate DoS attack against iterative servers [5]. Although these two

[1] This work has been partially supported by the Spanish Government through MYCT (Project TSI2005-08145-C02-02, FEDER funds 70%).

attacks are different, both rely on the awareness of a specific timing mechanism involved in the communication procedure of the victims that allows to reduce the traffic rate during the attack process. The reduction of the traffic rate in a DoS attack has essential implications, mainly because it allows the attacker to bypass those defense mechanisms that rely on the detection of considerable variations in the traffic rate [6][7][8].

This paper presents the low-rate DoS attack against concurrent servers (henceforth the LoRDAS attack). It is an evolution of the low-rate DoS attack against iterative servers, adapted for damaging more complex systems like concurrent servers. A concurrent server is characterized by allowing the processing of the received requests in a parallel way, not as in the iterative servers, where these are sequentially processed. Given that the bulk of the servers in Internet are implemented as concurrent servers and, in many cases, these are a critical infrastructure, the existence of a DoS attack against them supposes a high risk and, therefore, its execution could have a wide impact.

The paper is structured as follows. In Section 2, a model for concurrent servers is contributed. An analysis of the existent vulnerabilities in the concurrent servers and the mechanisms used for carrying out the attack are discussed in Section 3. Section 4 presents the results for the evaluation of the performance of the attack in both simulated and real environments. Finally, some conclusions and future work are given in Section 5.

2 Server Model

The scenario where the LoRDAS attack takes place is composed of a concurrent server connected to a network, some legitimate users accessing to it, and one or more machines that host the attack software (the fact that the attack is distributed or not will not affect this work). From these machines, the attacker launches the attack to the server. The traffic pattern coming from legitimate users will be unpredictable, due to the fact that it is affected by the perception of denial of service. Thus, it will be modelled as a poisson distribution with a generic inter-arrival time T_a.

The server has the ability of serving several requests at a time, in a parallel-like way (real or virtual concurrency). It could be designed as a single machine, or as a load balancer [9] bound to several machines. This last architecture is usually known as a farm of servers.

The proposed model for the server is depicted in Fig. 1. It represents a farm of M machines with a common load balancer that redirects each arriving request to anyone of them. Once that a machine has been chosen, the incoming request is queued up in a finite length queue within that machine, called *service queue*. In case that no free positions are found in this queue, the request is discarded (MO event). Obviously, whenever a machine has its service queue completely full, it will not be chosen by the load balancer, so the MO events will be raised only when no free positions at all are found in the whole server.

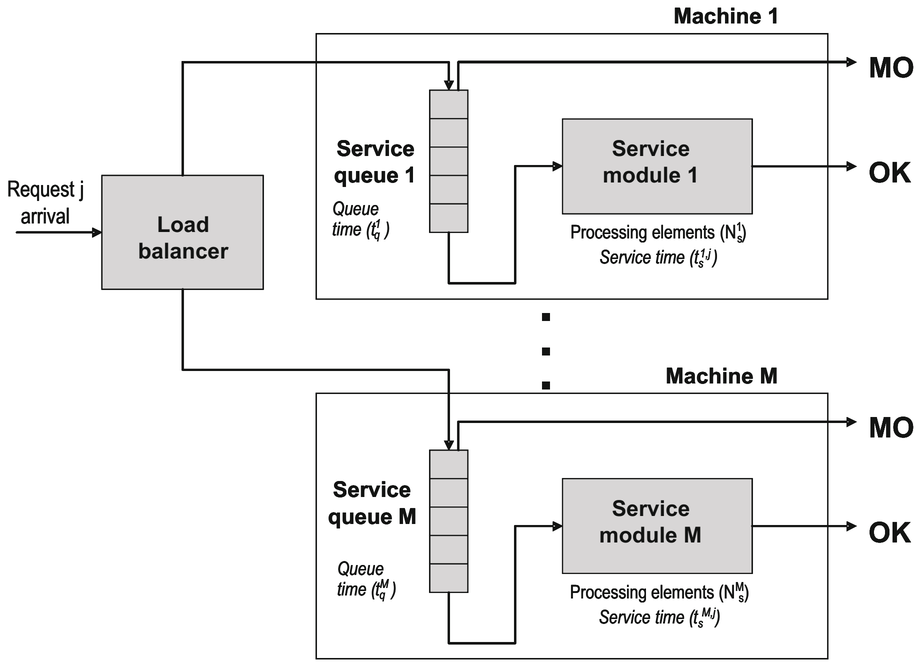

Fig. 1. Model for the server

The requests remain in the service queue during a *queue time*, t_q^i (being i the number of the considered service queue, $1 \leq i \leq M$), before passing to the module in charge of processing the petitions, that is, the *service module*. Inside this module, a number of N_s^i processing elements can exist. These elements play the role of either threads or children processes of the parent process implementing the server functionality on every machine. Each processing element is able to serve only one request at a time. Moreover, they could be running either in only one or in several processors within the machine. The total number of processing elements in the complete server is $N_s = \sum_{i=1}^{M} N_s^i$.

Each processing element in the service module i spends a time called *service time*, $t_s^{i,j}$, in processing a request j. After the processing is finished in the service module, an answer is sent to the corresponding client. We will refer to these answers from the server as *outputs*.

Even considering that all the machines in the server are identical, note that the service time $t_s^{i,j}$ is expected to be different for each request j. This is mainly due to the different nature of the requests. Namely, in a typical concurrent server like a web server, the service time has been typically modelled as a heavy-tailed distribution, as it is dependent on some different parameters, as the size of the requested resource [10].

We are interested in such a situation that identical requests are queued up in the server. In this case, considering also that all the machines have the same characteristics, it is expectable to get identical values for the service times of all the requests. However, in a real situation, the requests could be served by different machines, which normally have different features between them, and even if they are served by the same machine, the service time will also depend on the local conditions, namely the CPU load, memory utilization, disk usage, number of interruptions, and multiple other factors will vary the final value of the service time. The authors in [5] proposed, using the central limit theorem [11], that these variations could be modelled by a normal distribution. Hence, the service time, when identical requests are considered, becomes a random variable, T_s, that will follow a normal probability density function:

$$f(T_s) = \mathcal{N}(\overline{T_s}, var[T_s]) \tag{1}$$

3 Fundamentals of the LoRDAS Attack

The LoRDAS attack is a DoS attack that tries to exhaust the resources of the target server. Its particularity consists in carrying this out by means of a low-rate traffic in order to bypass some possible security mechanisms disposed to protect the server. For that, the attack exploits a vulnerability in the server that allows to intelligently reduce the traffic rate.

Next, the basic strategy followed for carrying out the attack and its implementation design will be presented.

3.1 Basic Strategy for Carrying Out the Attack

We consider a situation in which all the service queues of the server modelled in Fig. 1 are full of requests. Under this circumstance, every new arriving petition could not be queued up, and will be consequently discarded. Obviously, if an attacker manages to occupy all the positions in the queue, a denial of service will be experienced by the legitimate users, as their requests are going to be rejected. Normally, DoS flooding attack techniques try to achieve and maintain this situation by sending a high rate of requests to the victim.

In the LoRDAS attack, the strategy for reducing the traffic rate to be sent to the victim server is to concentrate the attack traffic only around specific instants wisely chosen. Therefore, it is not neccessary to send attack packets when all the service queues are full, as these will be rejected, but only when a new queue position is freed. Hence, the key aspects in this strategy are: a) to forecast the instants at which the outputs are generated in the server, and b) to manage the sending of attack requests in such a way that they arrive to the server at these predicted instants. Remark that the attack requests have the same form as any legitimate request, as the objective is only to occupy a position in a service queue.

For predicting the instants at which the outputs are going to be raised at the server, the attacker has to exploit a certain vulnerability. Although we can not say that there is a general and common vulnerability in all the concurrent servers that allows this prediction, we are afraid that, whenever the server exhibit a guessable fixed temporal pattern or deterministic behaviour, it is likely that the instants of the outputs could be forecasted. This fact becomes a vulnerability that allows an attacker to strike a LoRDAS attack against the server.

At first sight, it could seem that it is difficult to find such vulnerabilities, but we have found that it is not certain. For example, consider a media server that plays a publicity video on demand as an user ask for it. The number of licenses for simultaneously reproducing the video is limited. The vulnerability consists in the fact that the video always lasts the same time. Thus, if anyone ask for its playback, to estimate the instant at which it finishes is similar to consider its duration time. Therefore, an attacker could permanently seize a license by repeating the requests just when the license is released. If he manages to take all the licenses with this strategy, the DoS is achieved.

These vulnerabilities can also be found in more widespread servers in Internet. A very important example is the persistent HTTP server. As discussed in the following, it is possible to forecast the instants at which these servers rise the outputs and, therefore, they will be vulnerable to the LoRDAS attack.

Case Study: The Persistent HTTP Server

The persistent connection feature, that appears in the HTTP 1.1 specification [13], allows a web server to maintain a connection alive during a specified time interval after that an HTTP request has been served. This feature is used for reducing the traffic load in the case that several requests are going to be sent to the server on the same connection and in a reduced interval of time. Thus, before the sending of the first request, a connection is established with the server; then the request is sent and, after that, the server waits for a fixed amount of time before closing the connection[2]. If a new request arrives on this connection before the expiration of the mentioned timer, the timer is reset again. This mechanism is repeated a fixed number of times[3], after which the connection is closed. In this scenario, the attacker could follow the next strategy in order to predict the instant at which the output corresponding to a given request is going to be raised:

1. The attacker establishes a connection with the server. Making an analogy to the server model, this connection will occupy a position in the service queue and, thus, will play the role of a request.
2. The attacker sends an HTTP request to the server on the established connection, which will be redirected to the machine i $(1 \leq i \leq M)$.
3. The connection will be awaiting in the service queue i during a queue time for its turn to enter the service module i.
4. After t_q^i, a processing element extracts the connection from the service queue i, processes the request and answers (HTTP response) to the attacker. This response will reach the attacker at the instant t_{resp}.
5. A *timeout* with a fixed value t_{out} is scheduled in the processing element before closing the connection. t_{out} will play the role of T_s –Eq. (1)– in our model of the server, because all the requests will consume this time.

[2] In an Apache 2.0 server, the directive *KeepAliveTimeout* controls this timeout.

[3] In an Apache 2.0 server, the directive *MaxKeepAliveRequests* controls this number.

6. When t_{out} expires, the connection is closed and, consequently, a new connection is extracted from the service queue, generating a free position in the queue. Therefore, the action of closing a connection is what, in our model, is called an output.

In this particular case, the instant around which the attack packets should be sent from the attacker to the victim server, t_{attack}, could be calculated as:

$$t_{attack} = t_{resp} - \overline{RTT} + t_{out} \tag{2}$$

where we have considered $\overline{RTT} \simeq 2 \cdot \overline{T_p}$ as the mean value of the round trip time and $\overline{T_p}$ the mean propagation time between the server and the attacker. The above expression is obtained considering that the output happens t_{out} seconds after the HTTP response is sent to the attacker, which occurs at $t_{resp} - \overline{T_p}$. Consequently, the attacker should schedule the sending considering that it has to travel through the network and will experience a delay of $\overline{T_p}$.

As can be guessed, the persistent HTTP server example could be extended to any concurrent server that exhibits a behavior in which a timing scheme could be known by a potential attacker. Of course, the strategy for predicting the instants of the outputs should be adapted for each particular case.

3.2 Design of the Attack

Due to the fact that the instant at which an output is going to be raised depends on the service time T_s, and this is a random variable, it is expected that the forecasted instant varies with respect to the real instant of occurrence of the output. Moreover, the attacker should manage to synchronize the arrival of attack packets with the occurrence of the output, in order to seize the freed position. In this task, the variance of the *RTT* between the attacker and the server will also affect and therefore contribute to the mentioned variations.

In order to consider these variations, the attacker will send more than one attack packet, and will try to synchronize their arrival to the server around the predicted instant for the output. In other words, the attacker uses an ON/OFF attack waveform, called *attack period*. A different attack period should be scheduled for every predicted output k. The following features characterize the attack waveform:

– *Ontime interval* for the output k, $t_{ontime}(k)$: the interval during which an attempt to seize a freed position in the service queue due to the output k is made by emitting attack packets. These packets should be sent around t_{attack} –Eq. (2)–.
– *Offtime interval* for the output k, $t_{offtime}(k)$: the interval before *ontime* in the period of attack corresponding to the output k during which there is no transmission of attack packets.
– *Interval* for the output k, $\Delta(k)$: the period of time comprised between the sending of two consecutive packets during the ontime interval.

In the case of the persistent HTTP server, an attack period starts whenever the attacker receives an HTTP response, t^k_{resp}, and the value of $t_{offtime}(k)$ is obtained, by using the Expression (2), as:

$$t_{offtime}(k) = t_{out} - \overline{RTT} - \frac{t_{ontime}(k)}{2} \tag{3}$$

For simplicity, the parameters for the design of the attack period, $t_{ontime}(k)$, $\Delta(k)$, and consequently $t_{offtime}(k)$ are made equal for all the attack periods, independently of k, becoming t_{ontime}, Δ, and $t_{offtime}$. This way, there is no need to recalculate these parameters for every attack period, therefore being the computational burden for the attacker reduced.

The different attack periods are scheduled as the outputs are being predicted. One possible strategy to be followed in the implementation of the attack software (malware) is a sequential scheduling of the different attack periods, as the outputs are predicted. The main problem of this design is that, when two or more outputs are raised very close in time, the corresponding attack periods overlap, making the control of the attack software more complicated and not scalable. For this reason, the attacker could design the malware as a multithreaded process, in which every thread is in charge of seizing only one queue position and maintaining it as long as possible.

In this multithreaded malware, a new design parameter appears for the attack: the *number of attack threads*, N_a. It should be adjusted depending on the required level of DoS and the maximum traffic level allowed to be sent against the server.

On the other hand, if an attack period fails in seizing the wanted position in the server, possibly because another user has seized it before, the corresponding attack thread should try to obtain a new position. Note that, in the persistent HTTP server example, an attack thread can only forecast the instant of an output whenever it owns a position in the server. If the attack period fails, the only way of gaining again a position is by "blindly" sending attack packets. That's why the attack is designed in such a way that when an attack thread does not have any position seized, an attack packet is sent every interval that is setup as a new attack parameter and that will be called *trial interval*, Δ_t. When the attack thread gets again a position, it continues its normal operation with the attack period.

4 Evaluation of the Attack

In the following, the results obtained from the evaluation of the performance of the introduced attack, in terms of both its efficiency and the rate of the traffic involved, are presented. For this task, three indicators are used:

– *Effort* (E): it is the ratio, in percentage, between the traffic rate generated by the intruder and the maximum traffic rate accepted by the server (server capacity).

- *User perceived performance* (*UPP*): it is the percentage given by the ratio between the number of legitimate users requests processed by the server, and the total number of requests sent by them.
- *Mean occupation time* (*MOT*): this indicator is defined for a scenario where legitimate users do not send traffic. In this environment, *MOT* is the mean percentage of time during which the server has no free positions at all in any of its service queues, related to the total duration of the attack.

UPP is a measure for the efficiency of the attack, that is, it signals the DoS degree experienced by the legitimate users. *MOT* gives also a measure of the efficiency but the difference with *UPP* is that it considers an environment free of user traffic, thus allowing to evaluate the efficiency without any dependency on the user behaviour. The relationship between *UPP* and *MOT* is proportionally inverse, due to the fact that the user will succeed in seizing a position of the service queue with a higher probability as *MOT* gets lower.

E represents the traffic rate needed to carry out the attack. As the efficiency constraints of the attack grow, it is expected to need higher *effort*. Thus, the aim of the attack is to minimize *UPP* (similar to maximize *MOT*), trying not to reach a threshold in the *effort* that would make the attack detectable.

4.1 Simulation Results

The performance of the attack has been evaluated in a simulated environment where the LoRDAS attack as well as the legitimate users traffic and the concurrent server have been implemented using the Network Simulator 2 [12].

Regarding the efficiency achieved by the attack, in terms of *UPP*, we have tested it against 20 different server configurations, with a number of processing elements in the range $4 \leq N_s \leq 50$, and for each one of these configurations, several settings of the parameters of the attack have been selected: $t_{ontime} \in (0.1\,s, 0.6\,s)$, $\Delta \in (0.1\,s, 0.4\,s)$, $\Delta_t \in (1\,s, 5\,s)$ and $N_a = N_s$. The user traffic has

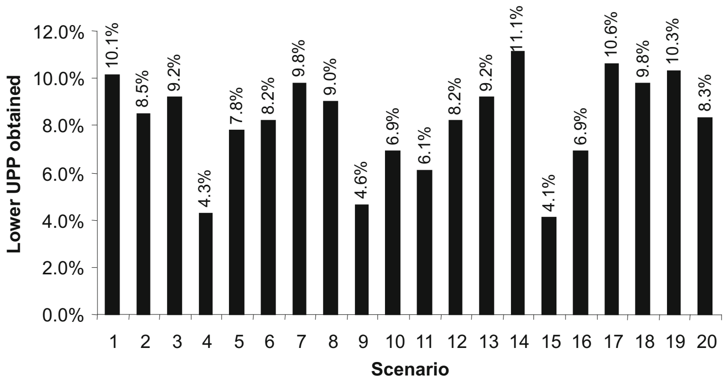

Fig. 2. Best efficiency of the attack (lower *UPP*) obtained for 20 different scenarios

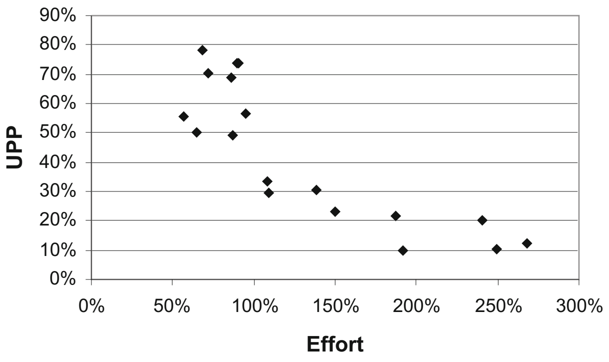

Fig. 3. Possible operation points for the attack in an scenario with $N_s = 4$, $\overline{RTT} = 0.1\,s$, and $f(T_s) = \mathcal{N}(12\,s, 0.1\,s)$: *UPP* vs *E*

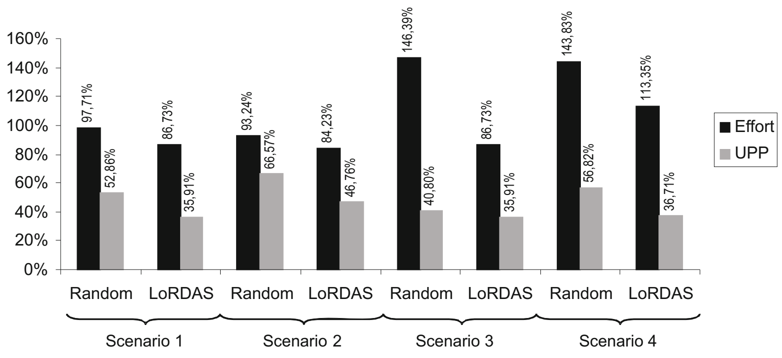

Fig. 4. *UPP* and *E* obtained by the LoRDAS attack compared with a random flood of packets, for 4 different scenarios

been tuned both with a rate nearly equal to the processing rate of the server, and also with a very low rate. The best efficiency results obtained for each server are represented in Fig. 2. The maximum attack traffic rate involved in all of these cases is $E = 315\%$. Note that the worst value obtained, 11.1%, (that is, for ten requests sent by the users, only one is served) represents a very high efficiency.

Moreover, from the previous experiments, we have inferred another conclusion: the attacker has a lot of operation points for adjusting the attack parameters to obtain a lot of possible combinations of efficiency and effort. As an example, the values for the *E* and *UPP* indicators obtained for 18 possible configurations of the attack to a server where $N_s = 4$, $\overline{RTT} = 0.1\,s$, and $f(T_s) = \mathcal{N}(12\,s, 0.1\,s)$ are shown in Fig. 3. Note that, for the attacker, a lot of parameters settings could be eligible. Regretfully, this means that it is possible to tune the attack parameters in order to bypass possible security mechanisms while a DoS is being made.

Additionally, other set of experiments have been made to check if the LoRDAS strategy implies an improvement for the DoS attack when compared with a

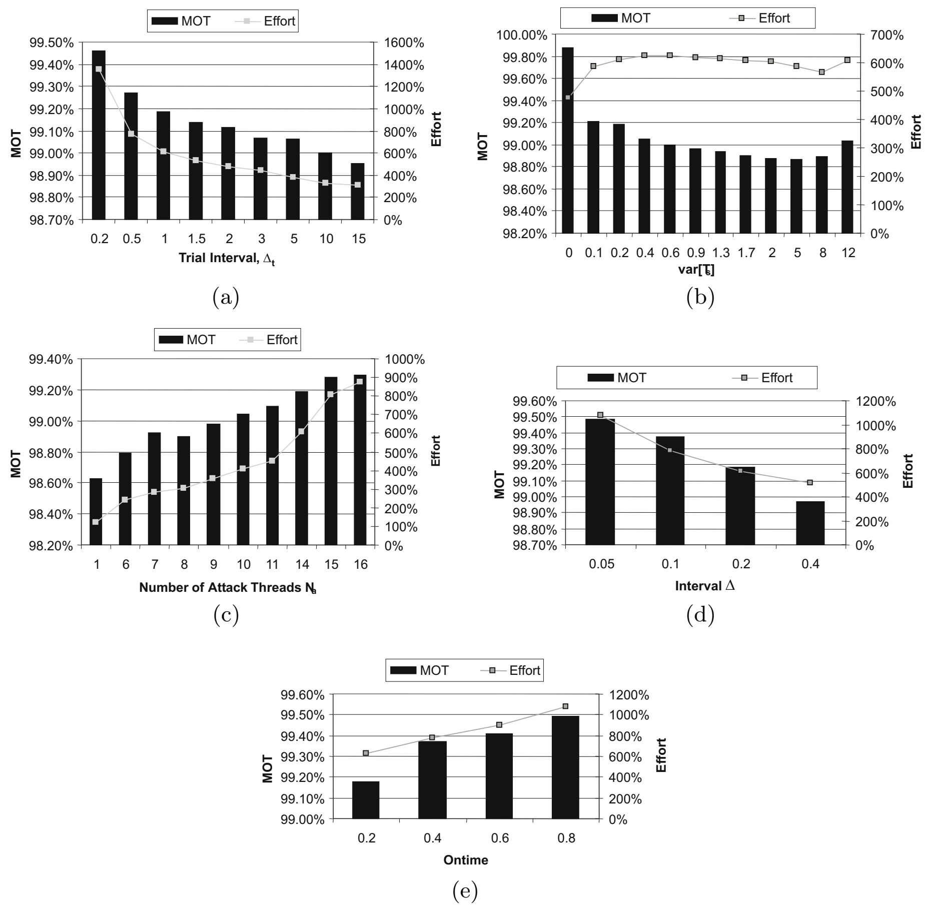

Fig. 5. Evolution of the indicators *MOT* and *effort* when some variations in the parameters of the attack are made: (a) trial interval Δ_t, (b) $var[T_s]$, (c) number of attack threads N_a, (d) interval Δ, and (e) t_{ontime}

similar rate of packets randomly sent (with no intelligence) to the server. Fig. 4 shows the comparison for four different scenarios. For each one, both the values of *UPP* and *E* are represented for the LoRDAS attack as well as for a random flood of packets. As it is difficult to adjust both strategies to obtain an equal value for *E*, we have taken always an attack configuration that generates a lower *E* value than in the random strategy (worst case for our attack). Note that in all the scenarios, as expected, the efficiency obtained by the LoRDAS attack is significatively higher, even when the effort involved is lower.

Finally, the evolution of *MOT* and *E* has been obtained by varying different parameters of the model. This study gives a better understanding of the behaviour of the attack, what could be applied later for the development of potential defense techniques. For this, fixed values have been set for all the parameters, only the value of one of them have been varied, independently of the others. Fig. 5 shows the results obtained for the most representative parameters: Δ_t, $var[T_s]$, N_a, t_{ontime} and Δ. The results show that, as expected, a higher efficiency is obtained when a higher effort is also employed. Besides, we confirm that, by configuring the attack parameters t_{ontime}, Δ_t, Δ and the number of attack threads N_a, the attacker could tune the attack process to get different values *UPP-E*, as deducted in the previous experiments. Finally, note that a variation in $var[T_s]$ only slightly affects the efficiency. When the variance is higher, the forecasted instants are usually wrong and, thus, the ontime period of the attack will be wrongly situated. Although this makes the efficiency to decrease, a limit is reached because the ontime intervals corresponding to the different attack threads help others to seize positions in the queue. This is why we hypothesize that a randomization in the service time will not constitute a good technique of defense against the LoRDAS attack.

4.2 Real Environment Results

A prototype of the LoRDAS attack has been also implemented in a Win32 environment in order to check its feasibility. The attack is carried out against an Apache 2.0.52 web server hosted in a machine with the Windows XP operating system. The server has been configured with the directive $KeepAliveTimeout =$ 10 seconds, which corresponds to the parameter t_{out}, which plays the role of $\overline{T_s}$ in our model. Besides, the directive $ThreadsPerChild$, which represents the number of threads for the processing of requests in the server, N_s, has been set in a range from 12 to 50. The scenarios chosen for the different considered experiments are analogous to that one presented in Section 2. The traffic from the legitimate users has been synthetically generated following a Poisson distribution with a mean traffic rate equal to that for the outputs generated by the server.

Table 1. Comparison between real and simulated environment results for 8 experiments

	E	UPP
simulated	86.73%	35.91%
real	81.74%	36.14%
simulated	84.23%	46.76%
real	74.57%	48.22%
simulated	113.35%	36.71%
real	109.00%	38.01%
simulated	268.47%	12.31%
real	269.82%	12.40%
simulated	68.04%	78.25%
real	76.00%	72.60%
simulated	249.49%	10.22%
real	256.41%	10.01%
simulated	89.88%	73.79%
real	92.41%	74.00%
simulated	191.56%	10.08%
real	183.80%	12.34%

Traces for the legitimate users as well as for the intruder have been issued for collecting the necessary data to calculate the attack indicators.

Table 1 shows the results obtained from eight different experiments taken from the set of trials. For each different attack configuration, both simulation and real environment values for UPP and E have been obtained. Note that in the results there is a slight variation between the simulation and the real values, with even better results in some cases in the real environment than in the simulated one. These variations are mainly due to deviations in the estimation of $\overline{RTT}$ and the distribution of the service time. Nevertheless, the results obtained in the real environment confirm the worrying conclusions extracted from simulation, that is, the LoRDAS attack can achieve very high efficiency levels and its implementation is perfectly feasible.

5 Conclusions and Future Work

The LoRDAS attack appears as a new kind of low-rate DoS attack that relies on the presence of known timing mechanisms in the victim server. We have shown that this attack is feasible, and an example for a persistent HTTP 1.1 web server has been contributed for that. The attack can be carried out in such a way that both the efficiency level and the traffic directed against the server are adjustable, which allows the attacker to tune the attack parameters in order to bypass possible detection mechanisms.

The effectiveness of the attack, in terms of the denial of service level and the amount of traffic directed to the server, has been evaluated in simulated and real scenarios, obtaining worrying results from both of them. Finally, a review of the behaviour of the attack when the different parameters of the attack and the server are changed is given.

Some further work is currently being made in this field, mainly focused on the development of detection and defense techniques for these attacks to mitigate their effects. The contributions of this study should be the starting point for this work.

References

1. CERT coordination Center. Denial of Service Attacks, `http://www.cert.org/tech_tips/denial_of_service.html`
2. Mirkovic, J., Reiher, P.: A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Comput. Commun. Rev. 34(2), 39–53 (2004)
3. Mirkovic, J., Dietrich, S., Dittrich, D., Reiher, P.: Internet Denial of Service. Attack and Defense Mechanisms. Prentice-Hall, Englewood Cliffs (2004)
4. Kuzmanovic, A., Knightly, E.: Low Rate TCP-targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants). In: Proc. ACM SIGCOMM 2003, August 2003, pp. 75–86 (2003)
5. Maciá-Fernández, G., Díaz-Verdejo, J.E., García-Teodoro, P.: Assessment of a Vulnerability in Iterative Servers Enabling Low-Rate DoS Attacks. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 512–526. Springer, Heidelberg (2006)

6. Siris, V.A., Papagalou, F.: Application of anomaly detection algorithms for detecting SYN flooding attacks. Computer Communications 29(9), 1433–1442 (2006)
7. Huang, Y., Pullen, J.: Countering denial of service attacks using congestion triggered packet sampling and filtering. In: Proceedings of the 10th International Conference on Computer Communications and Networks (2001)
8. Gil, T.M., Poleto, M.: MULTOPS: a data-structure for bandwidth attack detection. In: Proceedings of 10th USENIX Security Symposium (2001)
9. Zaki, M.J., Li, W., Parthasarathy, S.: Customized dynamic load balancing for a network of workstations. In: Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC-5 1996), pp. 282–291 (1996)
10. Liu, Z., Niclausse, N., Jalpa-Villanueva, C.: Traffic model and performance evaluation of Web servers. Performance Evaluation 46(2-3), 77–100 (2001)
11. Song, T.T.: Fundamentals of Probability and Statistics for Engineers. John Wiley, Chichester (2004)
12. Network Simulator 2, `http://www.isi.edu/nsnam/ns/`
13. Fielding, R., Irvine, U.C., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: RFC2068, Hypertext Transfer Protocol - HTTP/1.1, Network Working Group (January 1997)