Volume 27, issue 7–8, December 2008 ISSN 0167-4048

Computers & Security

The International Source of Innovation for the Information Security and IT Audit Professional

Vol. 27 No. 7–8

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Evaluation of a low-rate DoS attack against application servers

Gabriel Maciá-Fernández[*], *Jesús E. Díaz-Verdejo*, *Pedro García-Teodoro*

Department of Signal Theory, Telematics and Communications, E.T.S. Computer and Telecommunications Engineering, University of Granada, c/ Daniel Saucedo Aranda, s/n 18071 Granada, Spain

ARTICLE INFO

Article history:
Received 9 April 2008
Accepted 9 July 2008

Keywords:
Denial of service
Low-rate attack
Network security
Application servers
Intrusion event

ABSTRACT

In the network security field there is a need to identify new movements and trends that attackers might adopt, in order to anticipate their attempts with defense and mitigation techniques. The present study explores new approaches that attackers could use in order to make denial of service attacks against application servers. We show that it is possible to launch such attacks by using low-rate traffic directed against servers, and apply the proposed techniques to defeat a persistent HTTP server. The low-rate feature is highly beneficial to the attacker for two main reasons: firstly, because the resources needed to carry out the attack are considerably reduced, easing its execution. Secondly, the attack is more easily hidden to security mechanisms that rely on the detection of high-rate traffic. In this paper, a mechanism that allows the attacker to control the attack load in order to bypass an IDS is contributed. We present the fundamentals of the attack, describing its strategy and design issues. The performance is also evaluated in both simulated and real environments. Finally, a study of possible improvement techniques to be used by the attackers is contributed.

1. Introduction

Nowadays, the use of application servers connected through telecommunication networks is commonplace. In a typically distributed environment, like Internet, many of the services provided, especially those destined to final users, are implemented upon application servers. Examples of this include HTTP, mail, and DNS servers (Liu and Albitz, 1993).

A notable feature in the security field, nowadays, is the importance and seriousness of denial of service attacks (DoS) (Mirkovic et al., 2004; Jung et al., 2002). These have constituted an important focus of interest for much recent research (Mirkovic and Reiher, 2004), because of their harmful effects and frequency. DoS attacks seek to degrade the availability of a service by making users' access difficult or impossible, or by degrading the quality of service provided. Traditionally, this attack has been performed by one of two different strategies: on the one hand, many denial of service attacks exploit a specific vulnerability, and so are termed DoS vulnerability attacks. Others try to exhaust a resource in the individual target or its local network by flooding it with messages in such a way that the destination cannot handle the burden involved in their processing. These are called DoS flooding attacks. These two strategies could be used either separately or combined. An example of the latter is the well-known TCP SYN flooding attack (CERT Coordination Center, 1996), one flooding attack that takes advantage of a vulnerability caused by the asymmetry of the TCP protocol.

Flooding attacks are usually carried out using high-rate traffic against the victim. However, the technique used by DoS

[*] *Corresponding author*. Tel.: +34 958 24 23 05; fax: +34 958 24 08 31.
E-mail addresses: gmacia@ugr.es (G. Maciá-Fernández), jedv@ugr.es (J.E. Díaz-Verdejo), pgteodor@ugr.es (P. García-Teodoro).

0167-4048/$ – see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2008.07.004

attacks has recently evolved in such a way that some current approaches are able to attack using low-rate traffic (see Section 2).

In this paper, we analyze a new kind of low-rate DoS attack. We show that it is possible to use low-rate DoS attack against generic application servers. In fact, the procedure to be followed in order to attack a persistent HTTP server is detailed. The basic strategy is to overload the service queues of the server, where the requests are temporarily stored, in such a way that the resulting traffic directed to the server is low-rate. For this, the attacker aims to predict the instants at which every new position is freed in the queues and manages to insert a new request inside it.

Due to the extended use of application servers in Internet, these attacks represent a great threat. In addition, the use of low-rate traffic has two main benefits for the attacker. Firstly, the attack could be launched with fewer resources, simplifying the zombie recruitment process in a distributed DoS attack. Secondly, it facilitates concealment of the attack, as many security systems (normally IDS (Axelsson, 2000)) rely on the detection of traffic rates that exceed a given threshold, normally configured for a high-rate traffic in order to reduce false positives (Siris and Papagalou, 2006; Huang and Pullen, 2001; Gil and Poleto, 2001; Feinstein et al., 2003; Wang et al., 2002; Li, 2006, 2004; Scherrer et al., 2007; Kuzmanovic and Knightly, 2006). In this paper, we show how the load generated by the attack could be dynamically adjusted in order to bypass a load detection threshold for DoS in an IDS.

In summary, the aim of this paper is to illustrate the fundamentals of this low-rate attack against application servers. Our objective is to explore its viability and to describe its behaviour in order to motivate and promote the development of defense techniques against it. Worrying results are obtained regarding the simplicity of this type of attack and its effectiveness.

The structure of the paper is as follows: first, a review on related work is done. Second, a model of the scenario in which the attack takes place is presented in Section 3. Section 4 discusses the vulnerabilities in the server that could be exploited, fundamentals of the attack and design issues. Then, Section 5 contributes an analysis of the performance of the attack in both simulated and real scenarios. Section 6 evaluates some techniques that the attacker might use in order to improve the attack. Finally, some conclusions are drawn and suggestions are made for further study.

2. Related work

The first proposed work in the field of low-rate DoS attacks was the attack denounced by Kuzmanovic et al. (Kuzmanovic and Knightly, 2006) against TCP flows. This attack sends a burst of well-timed packets, creating packet loss and incrementing the retransmission timeout for certain TCP flows.

The use of low-rate traffic to carry out DoS attacks against applications has also been studied by several research groups. First, Guirguis et al. described the reduction of quality (RoQ) attacks, which try to degrade the performance by disrupting the feedback mechanism of a control system, with a small amount of attack traffic. The authors have studied these attacks in several scenarios, like bottleneck queues with Active Queue Management (AQM) employing Random Early Detection (RED) (Guirguis et al., 2004), internet end systems (Guirguis et al., 2005), dynamic load balancers (Guirguis et al., 2007a) and content adaptation controllers (Guirguis et al., 2007b). The main difference between these mechanisms and the proposal of this paper is that they get advantage of the transient mechanisms of the systems, whilst the attack described in this paper uses an estimation of the service time for the requests employed by an application server. Then, we claim that the deployment of mechanisms for avoiding transient behaviour exploitations (RoQ attacks), as those proposed in Shevtekar and Ansari (2008) and Li et al. (2006) is not enough for protecting end systems against low-rate DoS attacks.

Moreover, Chan et al. proposed an scheme for striking low-rate DoS attacks against applications in Internet that utilize periodic updates (Chan et al., 2006). This mechanism depends on the use of periodic updates in a relative manner by the server. The attack proposed in this paper targets generic application servers, and is not restricted to the periodic updates scenario only.

Finally, in a previous work (Maciá-Fernández et al., 2007), we presented an approach that aims at denying the service of an iterative server with low-rate. This approach uses the estimation of the inter-output time in the server to strike the attack. The work presented here is an extension of this attack to concurrent servers. A typical design for an application server in Internet follows a concurrent architecture. It enables to process requests in such a way that the service seems to be provided to all the clients at the same time (really or apparently), i.e., the different requests are served in a parallel way. This feature is usually implemented either by using several CPUs or machines, or by spawning threads or processes in one or more machines. In the latter case, each one of these threads or processes executes the corresponding code for providing the service. The complexity of this architecture makes it necessary, for the attacker, to refine the mechanisms that allow to strike a low-rate attack. We show here how this is still possible.

3. Scenario modelling and server basics

As a previous step in developing the idea behind the attack, it is necessary to model the scenario in which it is to take place, as shown in Fig. 1. In a normal situation, legitimate users try to access a specific service located in a server. For this purpose, they send one or several requests that traverse a network to reach the server. For the sake of simplicity, we consider that there is neither congestion nor losses of messages within the network.

Normally, it is difficult to determine the statistical pattern of users' incoming traffic. Some studies have experimentally deduced specific statistical distributions for this traffic, depending on its nature. In the case of HTTP requests sent to a web server, heavy tailed distributions (Liu et al., 2001) seem accurate. However, these distributions model the behaviour of user traffic for normal operation by the server. As we are interested in a scenario in which the server is victim of a DoS attack, users' behaviour cannot be represented through these

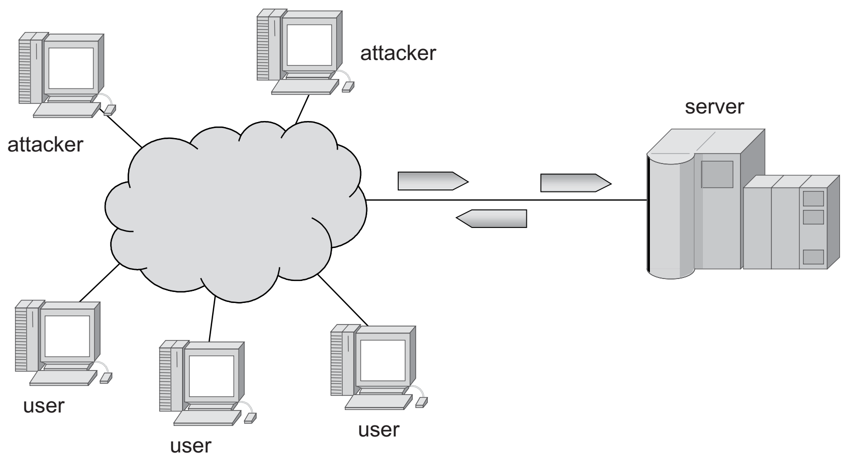


Fig. 1 – Study scenario.

models. For this reason, we make the traditional assumption that the traffic generated by the users follows a Poisson distribution. This means that the inter-arrival time T_a for the requests coming from all legitimate users can be modelled by the exponential distribution probability function (Song, 2004).

$$P(T_a = t) = \lambda \cdot e^{-\lambda t} \tag{1}$$

where λ is the mean inter-arrival rate of the user requests.

In our study scenario, an attacker tries to afflict a low-rate DoS attack on the server, which is located somewhere in the network. The attack could be launched in a distributed way (DDoS) by means of a typical multilayer structure of indirection (Mirkovic et al., 2004), or simply from a single machine (DoS). For the purposes of our study, the strategy chosen by the attacker is not a relevant factor.

The time involved for a request to reach the server from the attacker, and for its corresponding answer to arrive back is the round trip time, RTT. Due to the usual variable conditions in the network, RTT will be considered a random variable. In Elteto and Molnar (1999), a truncated normal distribution is proposed for this time, and so for the sake of simplicity, we will consider a normal variable distribution.

$$P(RTT = t) = N(\overline{RTT}, \mathrm{var}[RTT]) = \frac{1}{\sqrt{2\pi \cdot \mathrm{var}[RTT]}} \cdot e^{-\frac{(t-\overline{RTT})^2}{2\mathrm{var}[RTT]}} \tag{2}$$

The server is a principal element in the scenario. Its design could make use of either an iterative or a concurrent approach, the iterative being a special case of the concurrent. As previously mentioned, the distinctive characteristic of a concurrent server is its capability of serving different requests in parallel. This could be done by using either several machines or CPUs (real concurrency), or by just a single CPU (virtual concurrency). In the latter case, two main approaches may be adopted for its implementation: first, the parent process of the server spawns several child processes, and each one takes charge of a request. The second possibility is to have only one process with several running threads. Here, each thread is in charge of processing a request. Obviously, although at any specific instant t the processor is engaged in processing only one of the requests, the scheduler assigns a quantum of processing time to every process or thread, which makes it seem as if all the requests are being processed in parallel (virtual concurrency). Henceforth, we shall term the different processes or threads that execute the processing of requests in the concurrent server *processing elements*. As remarked previously, an iterative server could be considered as a special case of a concurrent architecture in which only one processing element exists and, therefore, the processing of the requests is done sequentially rather than in parallel.

The model proposed for the server is depicted in Fig. 2. The server could be composed of either a single machine or implemented as M replicas in different machines (a farm of servers). In the latter case, a load balancer is typically in charge of redirecting the incoming requests to the appropriate machine according to a predefined policy (Zaki et al., 1996; Hofmann and Beaumont, 2005).

As can be seen, the requests arriving at the server and redirected by the load balancer are queued up within a machine in a finite queue called *service queue*, whenever at least one free position is available. If there is no space left in the queue, the requests are discarded. Then, either an overflow message (MO) or a specific event could be raised (a log, an alarm, etc.), or even nothing could happen. Obviously, if the load balancer takes into account the occupation of the service queues of the different machines, this event will only occur when the positions in all the service queues are occupied. We then say that the server is in the *saturation state*. Moreover, we use the term *seizure* to indicate the capture of a position in the service queue.

At the same time as incoming requests are being queued up, in each machine $i \in [1, M]$, N_s^i different processing elements are ready to extract the requests from the queue and process them. The total number of processing elements present in a server is denoted by $N_s = \sum_i^M N_s^i$.

Whenever a processing element becomes free, a request in the service queue is extracted and passed to a module in charge of its processing, called *service module*. The request selected depends on the chosen queue discipline. For our purposes, it is irrelevant which queue discipline is applied, and so an FIFO discipline is assumed.

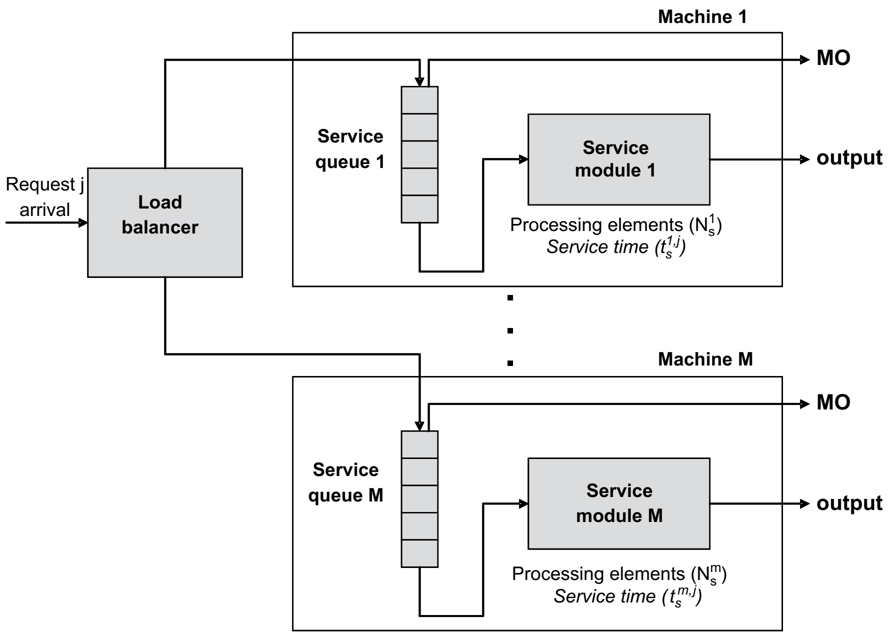


Fig. 2 – Model for the server.

A request j is served during a *service time*, $t_s^{i,j}$, that depends on the service module i, after which an answer is raised and sent to the corresponding user. Henceforth we will refer to this answer as an *output*.

The service time can be considered a random variable, T_s, whose values depend on several parameters. One of the most important is the amount of time that the request itself needs to be processed. Several studies have shown that the service time, when considering variations in the size of HTTP requests, can be modelled by heavy tailed distributions (Liu et al., 2001). Thus, if we consider a special case, in which all the incoming requests in the server are identical, a deterministic and fixed service time is to be expected. However, even in this case the service time is also affected by other parameters, like CPU load, the amount of available memory, the number of processes running in the same machine, the amount of interruptions in the host, etc. This means that, even when all the requests are identical, we should expect slight variations in service time values. As many variables influence this variation, by using the central limit theorem (Song, 2004), the distribution of the service time, $f(T_s)$, when the requests are equal, could be approximated by a normal variable.

$$f(T_s) = N(\overline{T_s}, \mathrm{var}[T_s]) \quad (3)$$

The case in which all the requests are identical is of importance, as this characterization of the service time is used in the attack design. Note that this statement does not mean that the attacker is restricted to use always identical requests, as we will discuss later.

4. Low-rate DoS attack specification

Having described a typical service scenario, the fundamentals for carrying out a low-rate DoS attack against application servers are presented in the following. First, the basic strategy to be followed is outlined. Next, the possible vulnerabilities to be exploited in the server are analyzed. Finally, the design of the proposed attack is discussed.

4.1. *Basic strategy for carrying out the attack*

The strategy to be followed by a low-rate DoS attack to cause unavailability in a target server consists in moving it to the saturation state. Thus, all the service queues will have no free positions and any new arriving request will be discarded, therefore causing a DoS.

The attacker will try to make the server process only his own requests by filling the service queues only with attack messages. Let us suppose that the attacker somehow manages to make the server reach the saturation state. In this situation, whenever a position is freed in any of the service queues, the attacker will try to seize it before any other user does. Thus, the aim of the attacker will be to achieve the maximum number of seizures.

Observation of the server model shows that a new position in a service queue is issued whenever a processing element produces the answer to a request, sends it to its remitter and extracts a new request from the corresponding service queue. There is a clear relation between the instant at which an output is raised in the server and that when a free position appears.

The attacker, in order to seize as many positions as possible, could follow the traditional strategy of flooding the server with requests at a high-rate, thus maximizing the probability of seizing the positions. However, it is desirable for the attacker to use low-rate traffic instead, mainly because this allows the attack to be carried out with many fewer resources, and also because it could thus bypass security mechanisms that rely on the statistical detection of high-rate traffic (Siris and Papagalou, 2006; Huang and Pullen, 2001; Gil and Poleto, 2001).

In order to reduce the rate of traffic sent to the server, the attacker could try to send attack requests in such a way that they arrive at the server only around the forecast instants at which the outputs take place, and not continually; if this is achieved, then the traffic is low-rate. Of course, the term low-rate is relative, because it depends on the rate used for comparison. For the purposes of our study, a rate is considered low if it is able to bypass an IDS system based on the detection of abnormal traffic rates. The main issue here is that, even though an IDS monitoring incoming traffic does not consider a given increase in some flows to constitute an attack, a suitable low-rate could achieve a DoS at the application level.

Two important aspects should be indicated in this attack strategy. First, even when a server is under a DoS attack, it is not really unavailable. The problem is that, while it is constantly processing attack requests, the user has the impression that the server is down. It could be said that the server is suffering a ‘‘partial DoS’’. Second, there is no need for the attack requests to be specially crafted. This means that the server will not appreciate the difference between the attack requests and others that are legitimate and will therefore process them normally.

4.2. *Analysis of server vulnerabilities*

In order to follow the attack strategy presented above, it is necessary to find a vulnerability in the server that makes it possible to forecast the instants at which the outputs are generated.

First, consider a server in a saturation state with identical requests placed in its service queues by an attacker. Let us also suppose that this attacker discovers the time invested in serving any of these identical requests (service time). In this situation, whenever the attacker observes the generation of an output (by receiving the response after it travels across the network), he could conclude that, after having waited the known service time, another output is going to be generated. Obviously, this is true because the next request processed by the processing element involves the same known service time.

An important issue in this process is the fact that the attacker must guess the service time for a specific request. In some cases, this could be done by simply sending these requests to the server and observing the elapsed time for receiving the response, δ. In this delay, three contributions appear: first, the round trip time RTT between the attacker and the server, which could be estimated and approximated by its mean value, $\overline{\mathrm{RTT}}$. Second, the time that the request spends in the service queue, that is, the queue time, t_q. This time depends on the number of requests located in the queue at the time the request arrives, their associated service times, and the number of processing elements in the server. Finally, also the service time, t_s, which depends on the own request and the server load and dynamics, contributes to δ. That is,

$$\delta = \overline{\mathrm{RTT}} + t_q + t_s \tag{4}$$

The parameter t_s is generically unknown in our scenario. To obtain it, as δ and RTT are observable variables, the value of t_q should be determined. At first sight, it is not possible for the attacker to predict the value of t_q so, to solve the problem, the only way is to eliminate this contribution. This could be done by sending probe requests when the server is relatively idle. Obviously, it is a difficult task for the attacker to determine the instant at which the occupation is low in the server. Only by intelligently choosing the instants of the scanning could this task succeed, and in many cases it will not be possible. Assuming that the attacker does succeed, in this situation, we would expect not to find requests in the selected service queue for the attack request and, thus, the value of t_q will be zero. Obviously, this probing activity has to be done carefully by the attacker in order to obtain an accurate value for the service time. The samples of t_s will conform the distribution of T_s – Expression (3).

Once T_s is estimated by following the above procedure, the attacker could use the knowledge of the service time to predict the instants at which the outputs are generated in the server. Note that, using this technique, another drawback arises. In effect, if we consider a situation in which not all the requests belong to the attacker, the prediction of some outputs will sometimes fail due to the fact that not all the requests involve the same service time. Indeed, as the requests that belong to legitimate users do not generate a response that is sent to the attacker, the number of correctly forecast outputs will be reduced to the number of attack requests placed in the service queues. As we will show later, the attack design will have to consider this aspect.

On the other hand, we have explored other types of vulnerabilities that allow to apply other strategies for the prediction of the instants of occurrence of the outputs, obtaining the worrying conclusion that the service time could simply be inferred, in some cases, from the behaviour of the server itself. An example can illustrate this.

Consider a media server that plays a publicity video on demand when a user asks for it. The number of licenses for simultaneous playbacks of the video is limited (limited number of positions in the service queue). The vulnerability consists in the fact that the video always lasts the same time. Thus, if anyone asks for its playback, it is only necessary to observe the beginning of the reproduction to estimate the instant at which it will finish (output). Thereby, a license could be permanently seized by repeating the requests just when the license is released (attack strategy). If the attacker manages to get all the licenses with this strategy, the DoS is achieved. Note that the attack would be even possible in the case the server queues up license requests, whenever the size of pending license request is finite. Moreover, even if the server deploys defense techniques such as allowing only one license per IP address, the attacker needs only to recruit as many zombies as the number of possible licenses in the server.

The difference between these cases and those in which the service time has to be estimated by probing the server consists in the fact that the attacker knows the instant at which the service time begins – in the media server example, when the playback starts. This has two benefits. First, the procedure for obtaining the value of the service time is simpler, because there is no need to have non-occupied service queues during the probing. Second, the errors generated by the presence of requests in the service queue that do not belong to the attacker disappear. In effect, whenever the attacker knows the service start time, this always has the same value. If other user's present requests, the attacker will not even notice the beginning of the service time.

Finally, we have found that, for some commercial and non-commercial concurrent servers, this second type of vulnerability could be found. A meaningful example is the case of a web server running the HTTP 1.1 protocol with the persistent feature. We will present this case study in the following.

4.2.1. *Case study: the persistent HTTP server*

The persistent connection feature, which appears in the HTTP 1.1 specification (Fielding et al., 1997), allows a web server to maintain a connection alive for a specified time interval after an HTTP request has been served. This feature is used to reduce the traffic load when several requests are sent to the server on the same connection for a short period of time.

When using a persistent connection, the communication between the client and the server is as follows. Before sending 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 (the persistent connection timeout[1]). If a new request arrives on this connection before the expiry of the mentioned timer, the timer is reset again. This mechanism is repeated a fixed number of times, after which the connection will be closed.

In this scenario, it is quite easy for an attacker to guess the value of the persistent connection timeout, t_{out}, with just a few tests. Using this knowledge, the process that an attacker could follow to forecast the instant of an output is the following (see Fig. 3):

(1) The attacker establishes a connection with the server. This connection occupies a position in the service queue.
(2) The attacker sends a request (HTTP request) to the server on the connection established.
(3) The request is received at t_0 and placed in the service queue for a queue time awaiting its turn to enter the service module.
(4) After t_q, that is, at $t_0 + t_q$, a processing element extracts the request from the service queue, processes it and sends an answer (HTTP response) to the attacker.
(5) The persistent connection timeout is scheduled in the processing element. When the timeout, t_{out}, expires, the connection is closed. As we are considering in this example that a connection in the system occupies a position, the occurrence of this event is similar to raising an output in our model.

In this process, to predict the instant at which an output is raised, t_{output}, the attacker only has to record the instant of the reception of the HTTP response, t_{rec}, and to consider the known value for t_{out}. Moreover, as discussed in Section 3, the service time – t_{out} in this case – should be considered a random

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

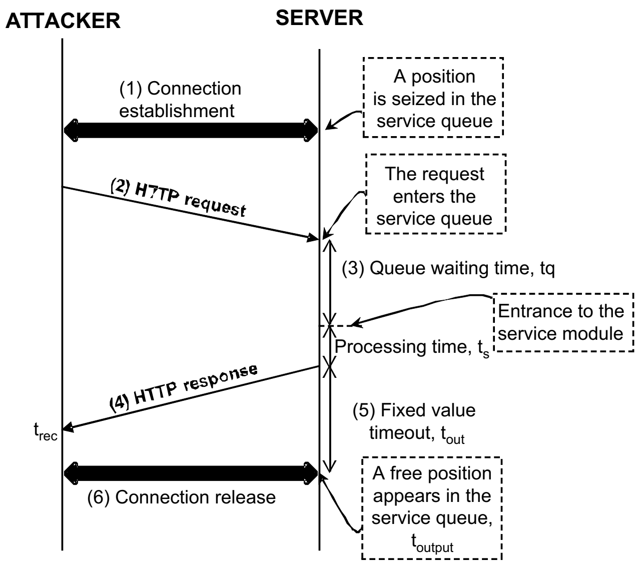

ATTACKER
SERVER
(1) Connection establishment
A position is seized in the service queue
(2) HTTP request
The request enters the service queue
(3) Queue waiting time, tq
Entrance to the service module
Processing time, t_s
(4) HTTP response
t_{rec}
(5) Fixed value timeout, t_{out}
A free position appears in the service queue, t_{output}
(6) Connection release

Fig. 3 – Process to forecast the instant at which an output is raised in the server.

variable, T_{out}, following Expression (3). That is, t_{output} also becomes a random variable, T_{output}, which corresponds to

$$T_{output} = N\left(t_{rec} - \frac{\overline{RTT}}{2} + \overline{T}_{out}, \mathrm{var}[T_{out}] + \mathrm{var}\left[\frac{RTT}{2}\right]\right) \quad (5)$$

where, for the sake of simplicity, we consider that the time invested by the answer in travelling from the server to the attacker is $\overline{RTT}/2$. As we will see later, this assumption does not affect the attack results.

This example illustrates how, in a widely used server, the existence of a temporal deterministic behaviour constitutes a vulnerability that allows a potential attacker to forecast the instants of the outputs and, thus, to launch a low-rate DoS attack.

4.3. *Design of the attack*

At this point it is clear that the aim of the attacker is to keep seizing all the free positions in the service queues as they appears. For this task, the instants at which the positions are freed are forecast through the mechanisms introduced in Section 4.2. Once these instants are predicted, the attacker tries to seize the freed positions before any legitimate user does so.

For this reason, the attacker sends attack packets to the server so that they arrive just after the outputs occur. Considering that the value of the predicted instants for every output is a random variable – Expression (5) –, more than one request (arriving around $\overline{T}_{output}$) should be used, in order to raise the probability of seizing the position freed by an output. This set of requests constitutes a part of the 'in advance' a *basic attack period*, which will be detailed in the following.

4.3.1. *The basic attack period*

For every output that is forecast, the attacker will launch a *basic attack period*, that is an ON/OFF waveform, composed of an inactivity phase followed by another of activity (requests sending). It is defined by these parameters.

- *Interval* ($\varDelta$): the period of time between the sending of two consecutive requests during the activity interval.
- *Ontime phase* (t_{ontime}): the activity interval during which an attempt to seize a freed position in the service queues is made by emitting requests at a rate given by $1/\varDelta$. Thus, $t_{\text{ontime}} = \varDelta \cdot (n_r - 1)$, where n_r is the chosen number of attack requests to be sent to the server.
- *Offtime phase* (t_{offtime}): the inactivity interval before *ontime* in the basic attack period, and during which there is no transmission of attack requests.
- *Start Of the Attack period* (SOA): the instant at which the offtime phase starts. This parameter defines the beginning of the basic attack period.

In order to synchronize the arrival of the ontime phase of the basic attack period around the mean value of the predicted instant of the output in the server, $\overline{T}_{\text{output}}$, this condition must be met.

$$t_{\text{offtime}} = \overline{T}_{\text{output}} - \frac{\overline{\text{RTT}}}{2} - \frac{t_{\text{ontime}}}{2} \tag{6}$$

where, for simplicity, we consider that the time needed for a request to travel from the attacker to the server is $\overline{\text{RTT}}/2$.

The above expression can be particularized for any server case. Specifically, for the persistent HTTP server, the value of $\overline{T}_{\text{output}}$ can be substituted by using Expression (5), thus leading to

$$t_{\text{offtime}} = t_{\text{rec}} - \overline{\text{RTT}} + \overline{T}_{\text{out}} - \frac{t_{\text{ontime}}}{2} \tag{7}$$

As is logical, the attacker should adjust the parameters of the basic attack period, that is, t_{offtime}, t_{ontime} and $\varDelta$, to maximize the probability of seizing the position and to minimize the rate of traffic sent to the server.

In summary, during the execution of the attack, for every output whose instant of occurrence is forecast, a basic attack period is scheduled trying to seize the corresponding free position. This behaviour is complemented by an additional aspect: whenever an output arrives at the attacker, another attack request is sent to the server. The goal of this methodology is to reduce the time during which the position is available, especially when the ontime phase has not succeeded in seizing it.

In conclusion, the total number of attack requests sent to the server due to the execution of a basic attack period and also to the response mechanism triggered at the reception of an output from the server is depicted in Fig. 4. Note that the arrival of the attack requests at the server is centered around $\overline{T}_{\text{output}}$.

4.3.2. *Software architecture design for the attack*

One more aspect needed for implementing the attack is to define how the attack periods are scheduled in time in order to seize as many positions as possible. One strategy could consist in having only one process which linearly schedules the basic attack periods as the instants of the outputs are predicted. However, it must be taken into account that it could be difficult to implement this strategy using only one process on the attacker's side, mainly because the outputs may occur very close in time, which would produce an overlap of the *ontime* phases associated to these outputs. For this reason, the implementation of the only-one-process architecture for the malware would become complicated and, even worse, not scalable.

This reasoning has led us to propose a multithreaded architecture for the malware, composed of a number of attack threads specified as a design parameter, N_a. Each of them is in charge of seizing one position in the service queue. For this purpose, these attack threads have two responsibilities: (a) to forecast the instant at which an output will be generated, and (b) to execute the basic attack periods centered around the predicted instants. At first sight, it is to be expected that the attacker will have to tune the value of the parameter N_a in accordance with the total number of positions in the service queues at the server. However, it is difficult for the attacker to estimate this number of positions, and so an alternative procedure is presented below.

4.3.3. *The basic seizure following strategy*

Whenever an attack thread executes a basic attack period to seize one position in any service queue of the server, the result of this execution is either a success, that is, the position is seized, or a failure.

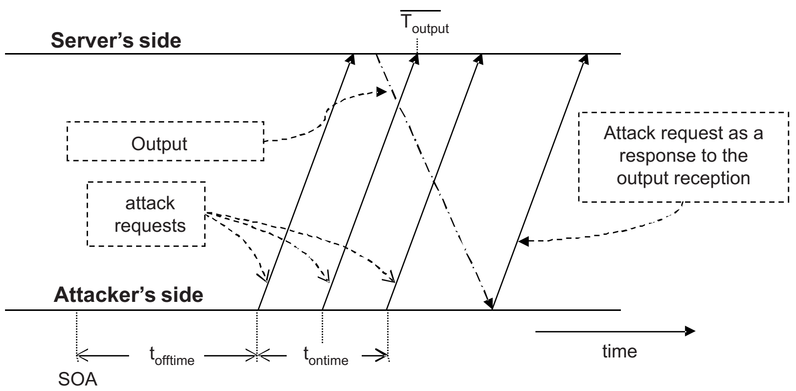


Fig. 4 – Diagram for the execution of a basic attack period and the response mechanism at the reception of an output.

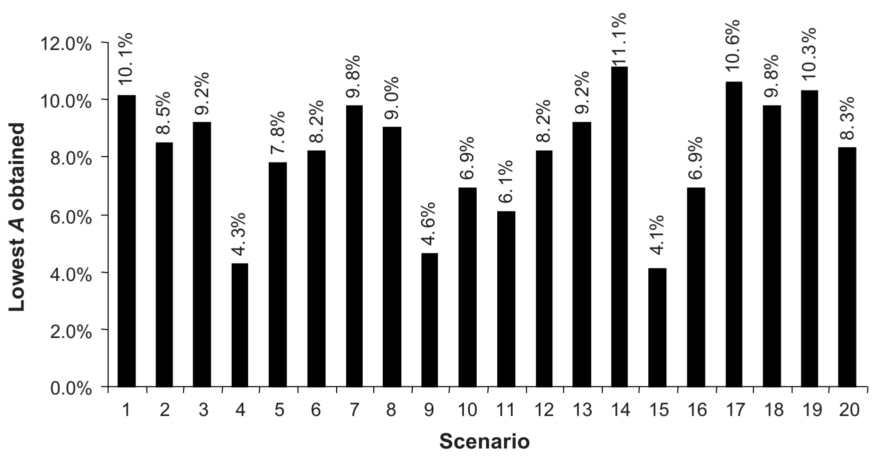


Fig. 5 – Best efficiency of the attack (lowest *A*) obtained for 20 different scenarios.

Possible causes for a failure in the seizure of a free position include:

- A legitimate user has seized the position, leading to the attacker's requests being rejected. Note that this situation is more likely to be reached when the deviations in the estimation of the output instant are high, or when a low number of attack requests is used in the *ontime* phase.
- Another attack thread has seized the position. As the *ontime* phase may involve sending several attack requests, two or more positions could be seized by a single attack thread, thus leading other attack threads to experiment a failure in their goal.

We should now define the behaviour to be followed by an attack thread in each case, success or failure, in the seizure of a position. Henceforth, we shall term this behaviour a *seizure following strategy*. First, we present the basic seizure following strategy, and then, in Section 6, some proposals will be suggested for improving this strategy.

To estimate the instant of an output, an attack thread needs first to have situated a request in a service queue, because the procedure for the estimation is triggered by the reception of a response from the server. Thus, in the case of a successful seizure of a position, the basic seizure following strategy establishes that the attack thread should behave normally, estimating a future output instant and executing a basic attack period.

On the other hand, in the case of a failure to seize a position, the attack thread will be incapable of forecasting the instant of another output, and the basic seizure following strategy establishes that the thread must ''blindly'' try to seize a position. This is done by sending attack requests at a rate $1/\Delta_r$, where Δ_r is a new design parameter of the attack, called *recovery interval*. It is expected that the lower the value chosen for Δ_r, the more likely it is that the attack thread will recover a position in a service queue. However, this will increase the traffic rate of the attack. A compromise between these two factors must be adopted. In Section 6, some mechanisms to make this adjustment are proposed.

5. Performance analysis for the attack

We are interested in testing the performance of the proposed attack in terms of two main characteristics: first, the efficiency achieved by the attack, measured as the DoS degree afflicted on the server, and second, the amount of traffic needed to achieve a specified efficiency.

In order to measure the traffic rate needed to carry out the DoS attack, we define the *overhead of the attack* (*O*), as the percentage ratio between the traffic rate generated by the intruder and the maximum traffic rate accepted by the server. It is important to note that, although *O* measures the relative traffic rate involved in the attack, this parameter does not represent the congestion level in the server, due to the fact that the latter depends not only on the traffic coming from the intruder but also on legitimate users' requests.

To measure the degree of DoS achieved by the attack, we define the *availability* (*A*) as the percentage ratio between the number of user requests served by the server, and the total number of requests sent by the legitimate users. This parameter gives an idea of the service level experienced by the legitimate users.

Although *A* is a good indicator of the DoS attack efficiency, it depends on the traffic pattern of the legitimate users. Alternatively, we could consider a measure that indicates the efficiency of an attack independently of the user traffic pattern. This hypothetical indicator would enable a comparison between different design strategies and would make it easier to take a decision about the values to adopt for the attack parameters. With this fact in mind, we define, in a scenario free of user traffic, the *percentage of available time* (T_{AV}), as the average time, in percentage terms, during which at least one free queue position in the server is available. Note that the probability of a legitimate user seizing a queue position will be directly proportional to the value of T_{AV}.

One aim of the attack, in terms of the indicators defined, should be to minimize user's perception of the availability of the service (*A*). This task is similar to that of minimizing the *percentage of available time* in the server, which reduces the probability of a legitimate user seizing a position in the queue.

Additionally, the attack should also minimize its *overhead* (O), thus making it less detectable by intrusion detection systems and requiring fewer resources in order to be executed. Obviously, it is expected that a reduction in the *percentage of available time* will necessarily increase the *overhead* of the attack, and vice versa.

5.1. *Attack capabilities*

According to the previously defined performance indicators, A, T_{AV} and O, the attack capabilities have been tested. We will describe in the following the experiments made for doing this. They have been carried out both in a simulation platform and in real environments.

5.1.1. *Simulation results*

In the first place, we evaluated the performance of the attack in a simulated environment in which a low-rate DoS attack module, as well as legitimate user traffic and a server were implemented using Network Simulator 2.

Several experiments were carried out to test the efficiency and the traffic rate needed for the attack. Regarding efficiency, in terms of A, the attack was tested against 20 different server configurations, with a number of processing elements in the range $4 \le N_s \le 50$, and for each one of these configurations, several settings of the attack parameters were selected: $t_{\text{ontime}} \in [0.1\text{ s}, 0.6\text{ s}]$, $\Delta \in [0.1\text{ s}, 0.4\text{ s}]$, $\Delta_r \in [1\text{ s}, 5\text{ s}]$ and N_a equal to the number of service queue positions in the server. The user traffic was tuned, on the one hand, with a rate nearly equal to the server's processing rate, and on the other, with a very low one (around 5% of the server capacity). The best efficiency results obtained for each scenario are shown in Fig. 5. In all the scenarios, the value obtained for the overhead was below 280%, that is, 2.8 times higher than the server capacity. Note that even the worst value obtained, 11.1% (that is, for every 10 requests sent by the users, only one is served), represents a very high level of efficiency.

We also carried out some experiments aimed at testing the flexibility of the attack. These consisted in choosing a large number of different scenarios and configurations, in order to test the attacker's possibilities of achieving a compromise between efficiency and traffic rate. The values for O and A obtained for 18 possible attack configurations in the same scenario are shown in Fig. 6. The scenario considered is

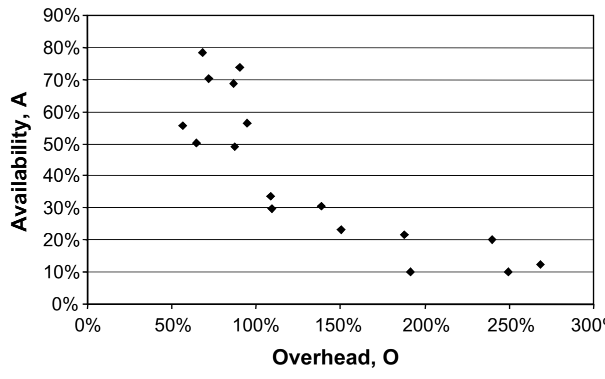


Fig. 6 – Possible attack configurations: *A vs O*.

characterized by the values $N_s = 4$, $\overline{RTT} = 0.1s$, and $T_s = N(12s, 0.1)$. As expected, a higher efficiency (lowest A) is obtained at the cost of higher traffic rates. Note that, for the attacker, many configurations are eligible and, what is more worrying, high DoS levels (around $A = 10\%$) are attainable in some cases with only $O = 200\%$.

These results lead us to conclude that the attacker has many possibilities of adjusting the attack parameters to obtain many possible combinations of efficiency and overhead. Regrettably, this makes it possible to tune the attack parameters in order to bypass possible security mechanisms while a DoS is being made, or to carry out the attack with a lower level of resources if needed.

In addition, we are interested in determining whether the proposed attack strategy produces an improvement for the DoS attack compared with a similar rate of requests randomly[2] sent (i.e., with no intelligence) to the server. For this purpose, some experiments were carried out to compare the efficiency and the traffic rate involved in the two strategies (random and intelligent). Our first strategy was to compare the efficiency values when the overhead in the two strategies is equal. However, there is a drawback to this approach, as it is not easy to control the resulting overhead once the parameters of the attack have been selected. For this reason, we always compared the two strategies by making the overhead lower in the intelligent strategy.

Fig. 7 shows the comparison for four different scenarios. For each one, both the values of A and O are represented for the ''intelligent'' attack as well as for a random flood of requests. Note that, for all the scenarios, the efficiency obtained in the intelligent attack is considerably higher – 35% in scenario 4, and 12% in the worst case (scenario 3) –, even when the overhead involved is lower. This confirms that the attack strategy presented in this paper represents a notable improvement over high-rate random flooding strategies.

5.1.2. *Real environment results*

A prototype that executes the low-rate DoS attack was implemented in a Win32 environment in order to test its feasibility. The attack was carried out against an Apache 2.0.52 web server (with the persistent connections feature enabled), hosted in a machine with the Windows XP SP2 operating system. Although the server model is general and considers a farm of servers, we assume that a testing scenario with only one server is enough to extract general conclusions about the attack behaviour. The server was configured with the directive *KeepAliveTimeout* = 10 s,[3] which corresponds to the parameter T_{out} in our model. The directive *ThreadsPerChild*, which represents the number of threads for the processing of requests in the server, N_s, was set in a range from 12 to 50.

The scenarios for the different experiments are analogous to those presented in Section 3. The traffic from legitimate users was synthetically generated following a Poisson distribution within a range of inter-arrival time values, T_a. Traces for the legitimate users as well as for the intruder sides were

[2] Following a Poisson arrivals' distribution.

[3] The value for this parameter is 15 s by default in the Apache configuration file. It was reduced by 5 s to speed up the experiments, as the expected results are the same with both values.

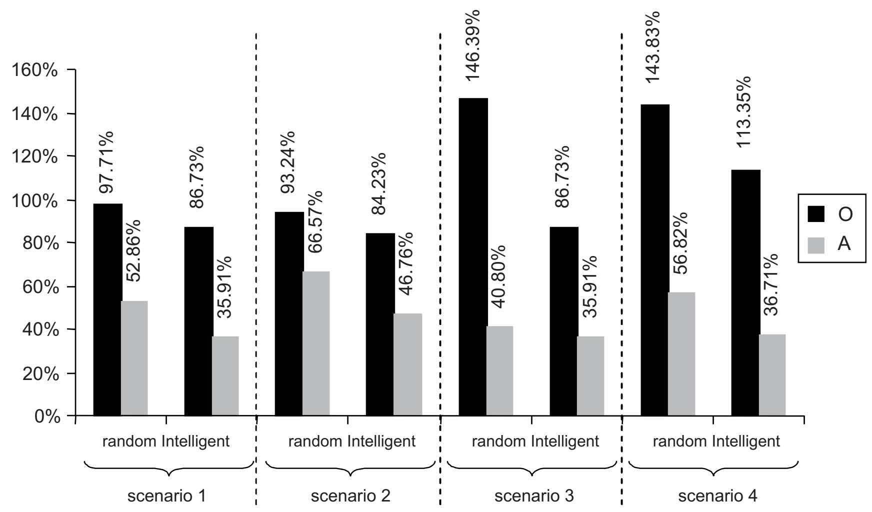


Fig. 7 – Efficiency and overhead obtained with the intelligent strategy compared with a random flood of requests, shown for four different scenarios.

issued in order to obtain the data needed to calculate the attack indicators.

Table 1 shows the results obtained from eight different scenarios taken from the set of experiments. For each different attack configuration, both simulation and real environment values for A and O were obtained. Note that in the results there is a slight variation between the simulation and the real values, with even better results being obtained in some cases for the real environment than for the simulated one. These variations are mainly due to deviations in the estimation of the parameters for the statistical distributions of RTT and the service time. Nevertheless, the results obtained in the real environment confirm the worrying conclusions derived from the simulation, that is, the proposed low-rate attack can achieve very high efficiency levels and, moreover, its implementation is totally feasible.

Table 1 – Comparison between real and simulated environment results in the attack against a concurrent server

	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

6. Improving the attack strategy

As shown above, not only is it possible to execute a low-rate DoS attack against servers but also it could achieve a very high level of efficiency. Although this is true, there are also some improvements that the attacker could adopt in order to obtain even better results. In the following, some of these improvements are described:

• *Attack distribution*:
The attacker can decide to execute the attack in a centralized manner (DoS) or by several distributed attack machines (DDoS). In the latter case, the benefits are clear (Geng and Whinston, 2000). The main disadvantages of distributing of the attack are the need to recruit a considerable number of zombies and to establish communication mechanisms between each of them and also with the attacker.

• *Use of spoofed addresses*:
The attack mechanism implies that the attacker must receive responses from the server in order to forecast the instants at which the outputs occur. This does not mean that the spoofing technique is not allowed. In fact, spoofing can be used if the range of spoofed addresses belongs to the same attacker machine network. Thus, this machine could sniff the packets sent to the spoofed address. Particularly when the local area network range of addresses is wide, the use of this technique will help the attacker to conceal his location.

• *Attack requests diversification*:
The general mechanism for exploiting the vulnerability consists in sending identical requests to it. However, we have shown that, in some cases, the vulnerability consists of a timing scheme that does not depend on the nature of the request made to the server. This is the case of the persistent HTTP server, in which the value of the persistent connection timeout does not depend on the HTTP request, but is always the same. In these cases, it is advisable for the attacker to

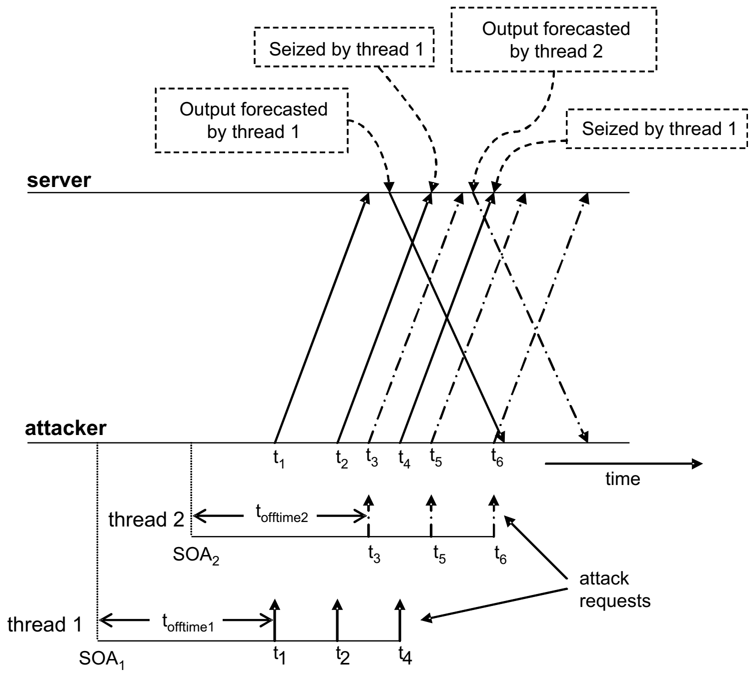


Fig. 8 – Diagram of the execution of two basic attack periods in which one attack thread seizes two positions but the other fails in the seizure.

diversify the attack requests in order to bypass possible security mechanisms that rely on the detection of identical or similar requests to the server.

It is also important to note that the attacker could improve the performance of the attack through two additional procedures:

- *Maximization of the service time of the attack requests*:
Once an attack thread has occupied a position in a service queue, it is desirable to maintain it as long as possible. Thus, the number of outputs in the server is reduced and, therefore, so is the traffic rate needed for occupying the freed positions. This tactic is not possible in some cases, but there are others in which it certainly is. Consider the persistent HTTP server case. If the attacker wanted to extend the time that a connection persists, he would only have to repeatedly send requests to the server on the same connection just before the persistent connection timeout expired. Depending on the configuration of the server, this could be done a specified number of times, or even with no limit to the number of requests.[4] This kind of strategy has previously been seen in other attacks like *Naptha* (SANS Institute, 2001).
- *Attack strategy optimization*:
The attacker can also optimize the attack strategy itself in order to gain better performance. Concerning the basic seizures following strategy, it has been stated that an attack thread should recover a position by sending requests at a rate

[4] In the Apache 2.0 server, this value is given by the directive *MaxKeepAliveRequests*. When its value is 0 there is no limit to the number of consecutive requests at which the connection is closed.

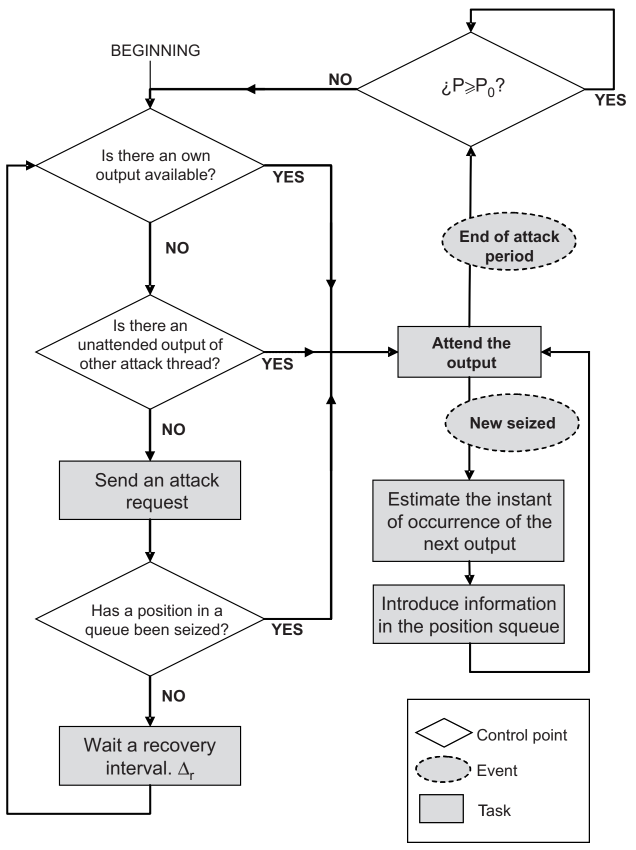

BEGINNING
NO
¿P⩾P0?
YES
Is there an own output available?
YES
NO
End of attack period
Is there an unattended output of other attack thread?
YES
Attend the output
NO
New seized
Send an attack request
Estimate the instant of occurrence of the next output
Introduce information in the position squeue
Has a position in a queue been seized?
YES
NO
Wait a recovery interval. Δr
Control point
Event
Task

Fig. 9 – Flow diagram for the behaviour of the attack threads with the ISS strategy.

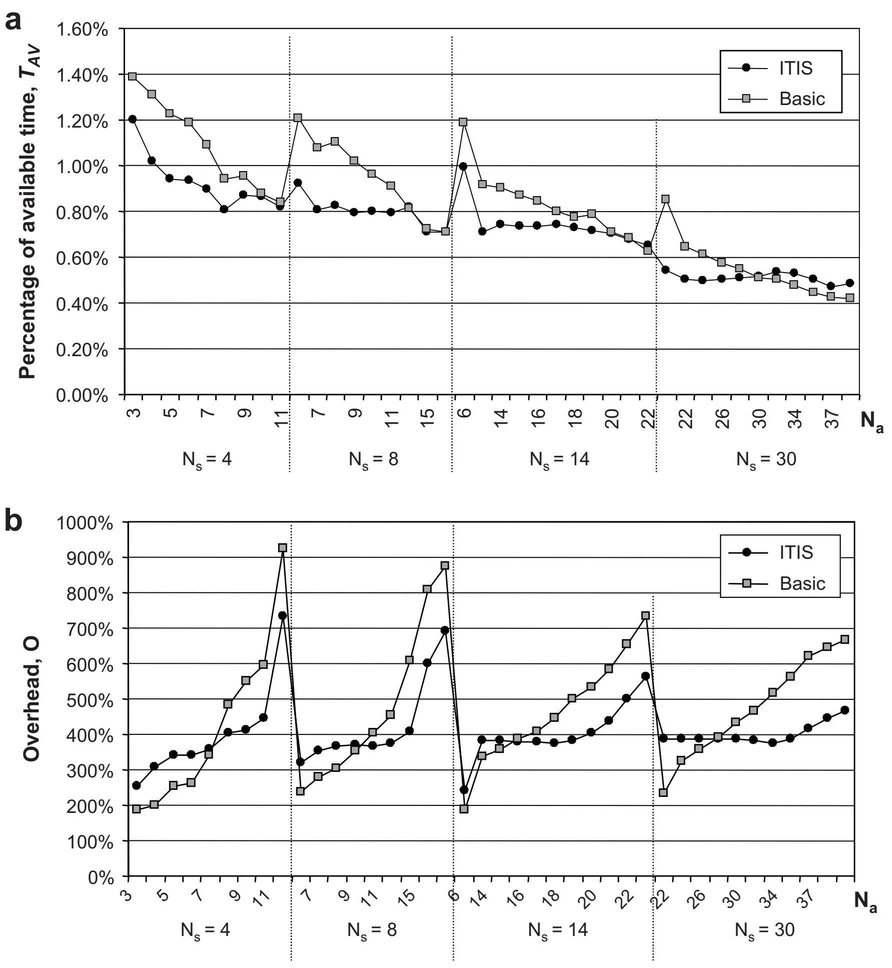


Fig. 10 – Comparison between the values T_{AV} and O obtained from executing the attack with both the basic and the ISS strategies, when the value of N_a is varied, for four values of N_s.

of $1/\Delta_r$ in the case of failure to achieve seizure. In the following, an improved strategy is described. This new technique makes the attack more efficient and enables the attacker to have more control over its execution.

6.1. *Inter-thread information sharing and seizure threshold strategy (ISS)*

If we analyze the behaviour of the attack implemented with the basic seizures following strategy, one aspect stands out, namely the fact that, although the objective of an attack thread is to seize only one position at a time in a service queue, as it may send more than one request during a basic attack period, the number of seized positions could also be more than one. This phenomenon is illustrated in Fig. 8, where the instants of occurrence for two outputs are represented. One of them has been estimated by attack thread 1, while attack thread 2 has forecast the other one. Both attack threads have programmed basic attack periods with the sending of three attack requests in this example. These sendings are represented by continuous lines for attack thread 1 and dashed lines for attack thread 2. It can be seen in the figure that, when the output forecast by thread 1 occurs, the freed position is correctly seized by a request from this thread. However, when the output whose instant of occurrence was estimated by thread 2 occurs, the next attack request also belongs to thread 1. This means that thread 1 seizes two positions in the queue, whilst thread 2 loses a position. Initially, this is not a drawback, as the aim of the attack is to maintain the positions seized. However, two problems that are described in the following then arise.

First, if the attacker runs as many attack threads as there are positions in the service queues, the fact that one attack thread seizes more than one position makes others enter the recovery phase during which a request is sent every Δ_r seconds. Obviously, if the rest of the positions are seized, the attacker is foolishly sending requests to the server, thus increasing the traffic rate with no benefit at all.

On the other hand, each attack thread has been designed to launch a basic attack period whenever the instant of an output has been forecast. This means that, even if the attack thread had more than one position seized in the queues, it would choose only one of them to launch an attack period. Henceforth, we will say that the output in a thread for which an attack period is scheduled is an *attended output*. On the

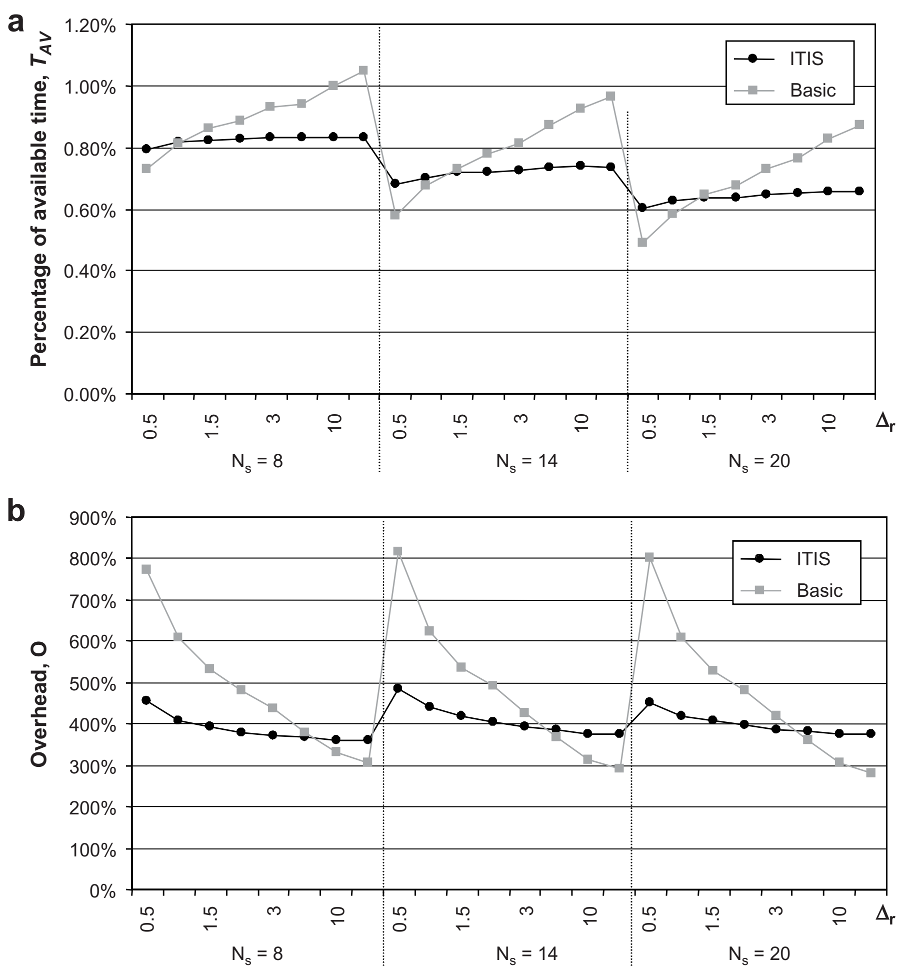


Fig. 11 – Comparison between the values T_{AV} and O obtained from executing the attack with both the basic and the ISS strategies, when the value of Δ_r is varied, for three values of N_s.

contrary, the remaining positions seized in the queue will generate outputs that we term *unattended outputs*. The second problem that appears when an attack thread seizes more than one position consists of the fact that it will only attend one output, and the rest will be unattended, thus lowering the efficiency of the attack.

To resolve these two problems, we propose a new strategy, called the inter-thread information sharing and seizures threshold (ISS), which modifies the proposed behaviour by the basic seizures following strategy in order to improve both the efficiency of the attack and the control over its execution, as described below. A flow diagram for the behaviour of the attack threads with this new strategy is shown in Fig. 9, and can be summarized by describing two main features:

• *Inter-thread information sharing (ITIS) feature*:
Whenever an attack thread, after the execution of a basic attack period, fails in the seizure of a position in a service queue, before entering the recovery phase as specified in the basic strategy, it will ask other attack threads for seizures that are unattended. If any of them exists, the attack thread will attend it by programming a basic attack period around its predicted instant of occurrence.
There are two remarkable implementation issues in this feature. First, each attack thread will have to save the information of unattended outputs in a common *positions queue*, that should be accessible to all the attack threads. In order to enable the distribution of the attack, the information inserted in the positions queue should not be host-dependent – for example, it could not be a socket descriptor. Therefore, a good candidate is the time-stamp of the predicted instant for the output. This implies that every attack thread that seizes a position must estimate the instant of the output before inserting an attack request in the positions queue. Thus, in the case of the persistent HTTP server, the attack thread should wait for the HTTP response, at t_{rec}, to calculate $\overline{T}_{output}$.
At the same time that the attack threads are inserting information about the instants of the outputs, a maintenance task should be accomplished on the positions queue, in order to eliminate the information corresponding to those outputs that have already occurred. When this happens, the eliminated positions have become unattended, which is not desirable, as it causes the efficiency to fall. In order to avoid this, the extraction policy from the positions queue should be to take those for which the expiry timer is closest to ending.

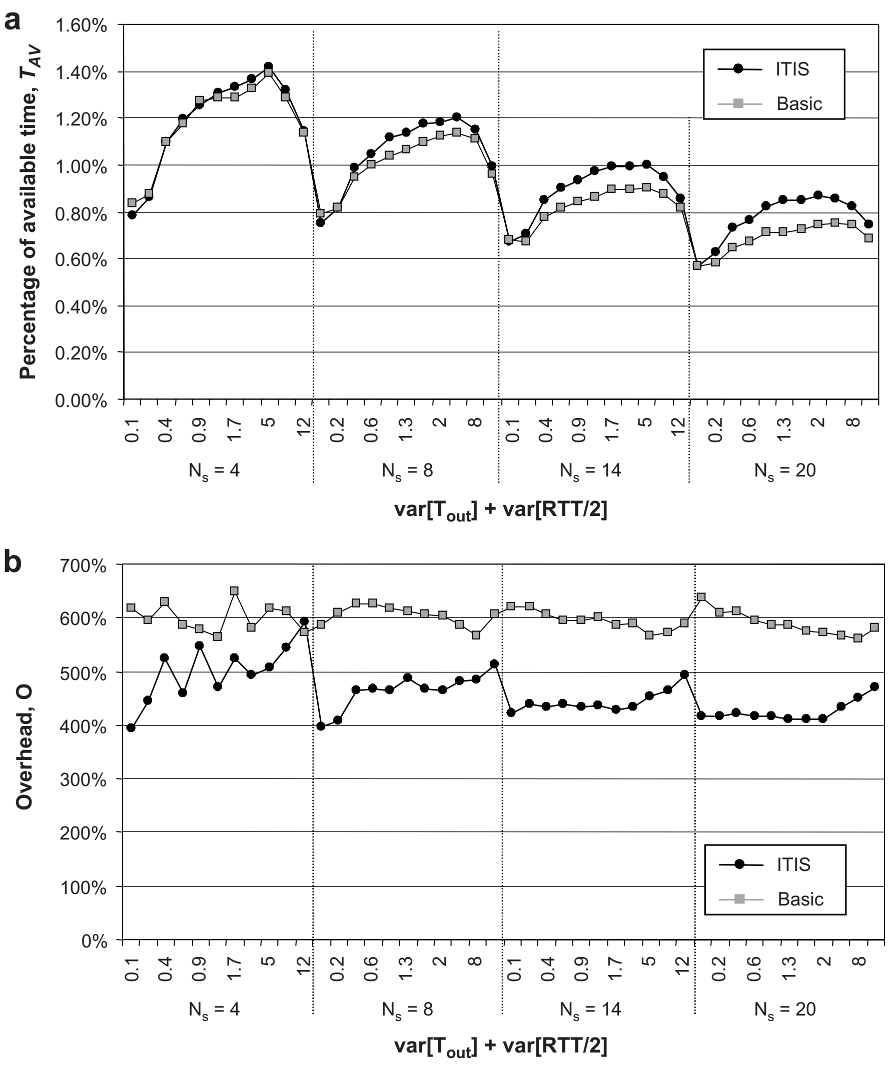


Fig. 12 – Comparison between the values T_{AV} and O obtained from executing the attack with both the basic and the ISS strategies, when the value of var[T_{out}] + var[RTT/2] is varied, for four values of N_s.

On the other hand, it is also possible that, when an attack thread fails in a seizure, it has already seized positions corresponding to unattended outputs that have not been inserted in the queue (known as *own outputs*). In this case, the thread can attend an own output instead of fetching information from the positions queue, thus simplifying and speeding up the process of information sharing.

Finally, if there are available neither own seized positions nor information on the positions queue, the attack thread will have to wait a recovery interval, Δ_r, to repeat the fetching process previously described.

• *Seizures threshold (ST) feature*:

This feature allows the attacker to establish a threshold for the number of seizures to be simultaneously possessed, P_0. This threshold could be dynamically modified during the execution of the attack.

The number of total positions seized in the server, P, is continuously monitored. As long as the condition $P < P_0$ is maintained, the behaviour of the attack threads is that specified by the ITIS feature. However, when the condition $P \geq P_0$ is reached, the behaviour of those attack threads which are not attending outputs is modified, in such a way that they remain "asleep", waiting for the condition to change.

6.1.1. *Analysis of the ITIS feature*

We are now interested in testing whether, as expected, the ITIS feature improves the efficiency of the attack. For this purpose, the new strategy was implemented in the NS-2 simulator. For the sake of simplicity, all the features of the ITIS strategy except that of the distribution of the attack among different compromised machines were implemented.

The method chosen for comparing the basic strategy and the ITIS feature is based on observation of the performance indicators that measures both the efficiency (in terms of the percentage of available time, T_{AV}), and the traffic rate (overhead, O). This observation is carried out for a large number of different simulated scenarios. For each one, the variation of several attack parameters was studied. These parameters were varied independently, assuming a default value for them when a parameter was not studied. The following parameters were used.

- Number of attack threads, N_a. The default value is equal to the number of positions in the service queue, $N_a = 14$.
- Recovery interval, Δ_r. The default value is $\Delta_r = 1$ s.
- Deviations between the occurrence of an output and the arrival of the basic attack period at the server. Taken from Expression (5), these variations result in $\text{var}[T_{out}] + \text{var}[\text{RTT}/2]$. The default value is $\text{var}[T_{out}] + \text{var}[\text{RTT}/2] = 0.2$.
- Duration of the activity phase of the basic attack period, t_{ontime}. The default value is $t_{ontime} = 0.4$ s.
- Interval, Δ. The default value is $\Delta = 0.2$ s.

Figs. 10–14 display the results of these experiments. Firstly, in Fig. 10, it can be seen that the efficiency of the ITIS feature is higher (lower T_{AV}) for most of the N_a values, even when the overhead is lower. Note that, for low N_a values, the overhead is higher with ITIS, which seems to be the opposite of the expected behaviour, that is, an overhead reduction. However, this is because, as more outputs are attended with ITIS, the overhead introduced by the associated basic attack periods is considered. This is why a higher efficiency is also obtained. On the other hand, it can be seen that, when using a high value for N_a, the use of a coordinated strategy between them (ITIS) reduces the overhead.

Fig. 11 shows the results obtained when the variations are performed on the Δ_r parameter. It can be seen that only when the value of Δ_r is low is there a better efficiency in the basic strategy, although this is, of course, at the cost of a considerably higher overhead. On the other hand, for non-low values of Δ_r, ITIS is more efficient, even when a lower overhead is considered.

Fig. 12 shows the results concerning $\text{var}[T_{out}] + \text{var}[\text{RTT}/2]$. We see that, for similar efficiency values, the overhead is much higher in the basic strategy (in the scenarios considered there is an absolute mean increase of 142.83%).

Fig. 13 shows the results obtained when varying t_{ontime}. Note that, while the overhead with ITIS is always lower, the efficiency obtained is better (lower T_{AV}). Among all the scenarios shown, only when $t_{ontime} = 0.2$ s can there be found an improvement with the basic strategy, although the cost to be paid is an overhead increase of 200%.

Finally, Fig. 14 shows the results from the variations in the Δ parameter. Again, in every case a lower overhead is needed with ITIS, which even obtain better efficiency in some cases.

In summary, the above-described experiments confirm that the ITIS feature leads to a better performance compared to the basic strategy, and that this is due to the sharing of information between the different active attack threads.

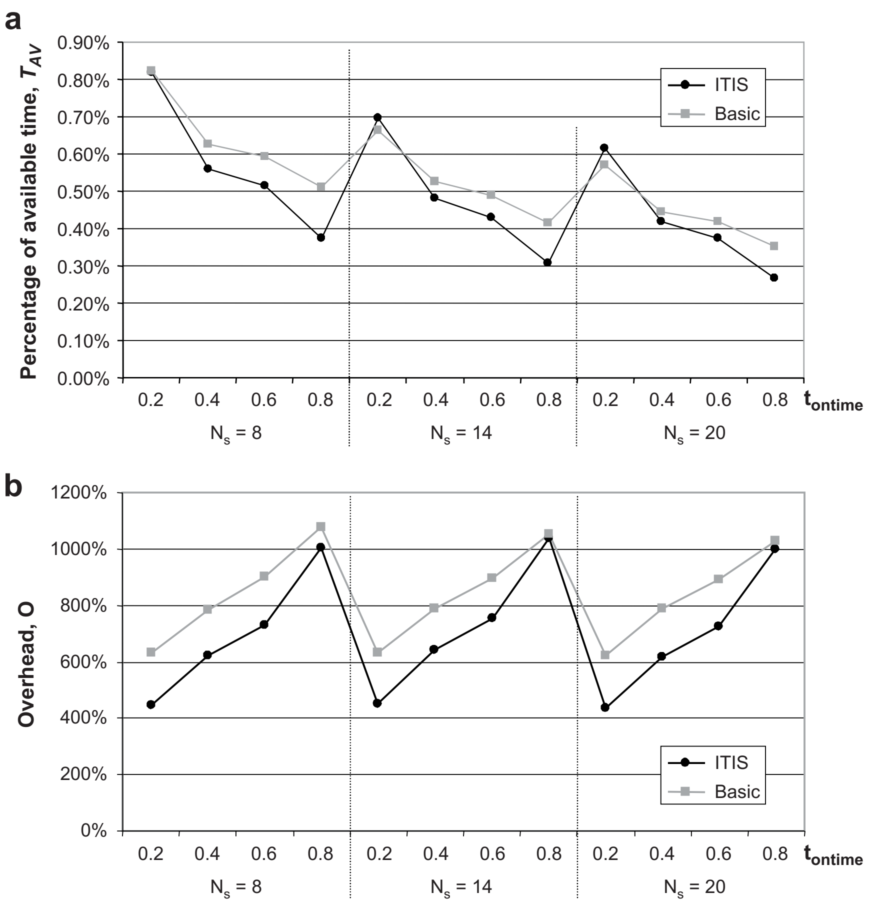


Fig. 13 – Comparison between the values T_{AV} and O obtained from executing the attack with both the basic and the ISS strategies, when the value of t_{ontime} is varied, for three values of N_s.

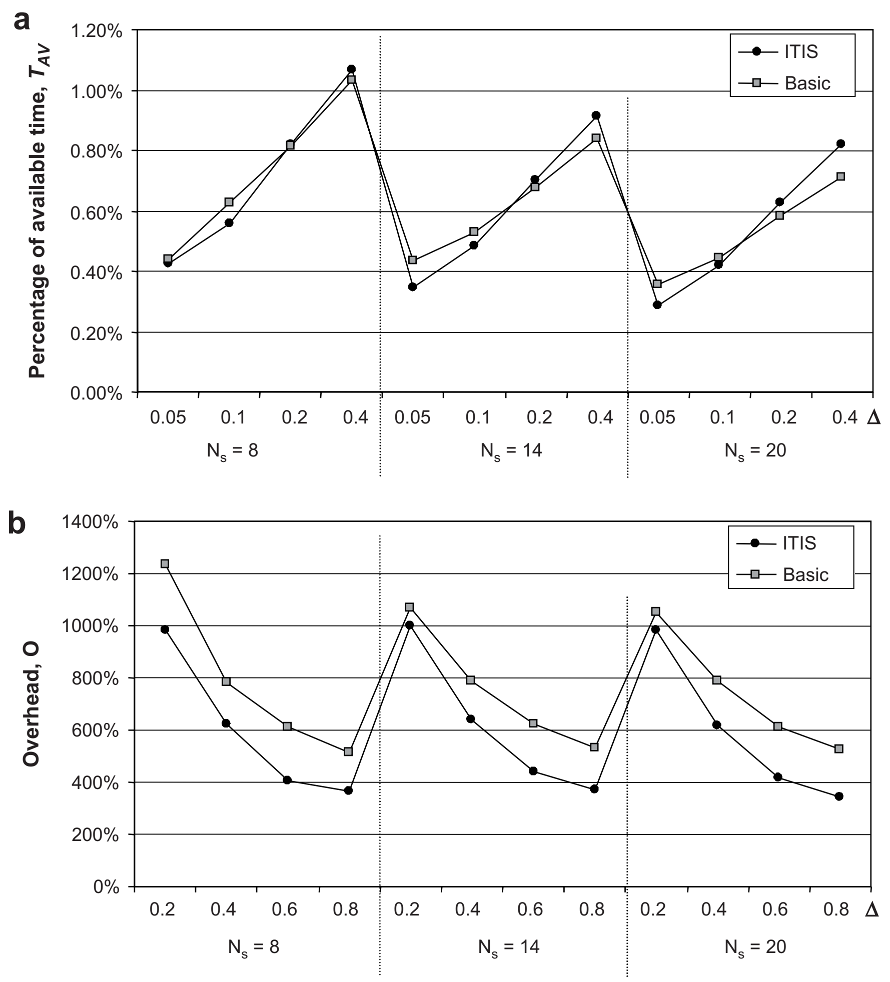


Fig. 14 – Comparison between the values T_{AV} and O obtained from executing the attack with both the basic and the ISS strategies, when the value of Δ is varied, for three values of N_s.

6.1.2. *Analysis of the ST feature*

We are now interested in the usefulness of the ST feature. In particular, it is important to determine under which conditions the attack is able to reach a configured seizures threshold and if it is able to maintain it over time. For this purpose, experiments were carried out in a worst case scenario for the attack, consisting of a server receiving high-rate traffic from legitimate users. Thus, performance of the attack is complicated, as the users will have a higher probability of seizing positions in the queue.

In the experiments carried out, some conclusions about the behaviour of the attack with the ST feature were reached, and these led to the introduction of some design rules for the attack. Let us now describe these conclusions and the experiments from which they are derived:

- *The attack is able to reach a number of positions oscillating around the threshold*:

The temporal evolution of the number of seized positions during an attack when three different thresholds are set is represented in Fig. 15. In this case, the server owns a total of 40 positions in the queues. It can be seen that, after a short transitory period, P oscillates around the threshold, P_0. The values $P > P_0$ are obtained due to the activation of all the threads after P falls below P_0, which causes an excess in the number of seizures. On the other hand, the values $P < P_0$ are due to the existence of user traffic, which is also fighting to seize positions in the queues.

- *The threshold allows the attacker to establish an operation point for the attack*:

When using the ST feature, the overhead and the efficiency of the attack are affected by the value chosen for the seizures' threshold. Thus, the choice of high values for the threshold implies high efficiency and, consequently, a high overhead. On the other hand, if a low value is selected for the threshold, it will result in a lower overhead and, of course, worse efficiency.

In Fig. 16, the performance indicators for an example attack, when launched against a server with 40 positions, are shown. The figure represents both the availability, A, and the overhead, O, for different values of the threshold, P_0. Note that, as P_0 increases, the attack becomes more efficient (lower value for A) and the overhead rises.

In conclusion, the attacker could use the threshold in order to gradually increase the DoS level. By these means, the attack is

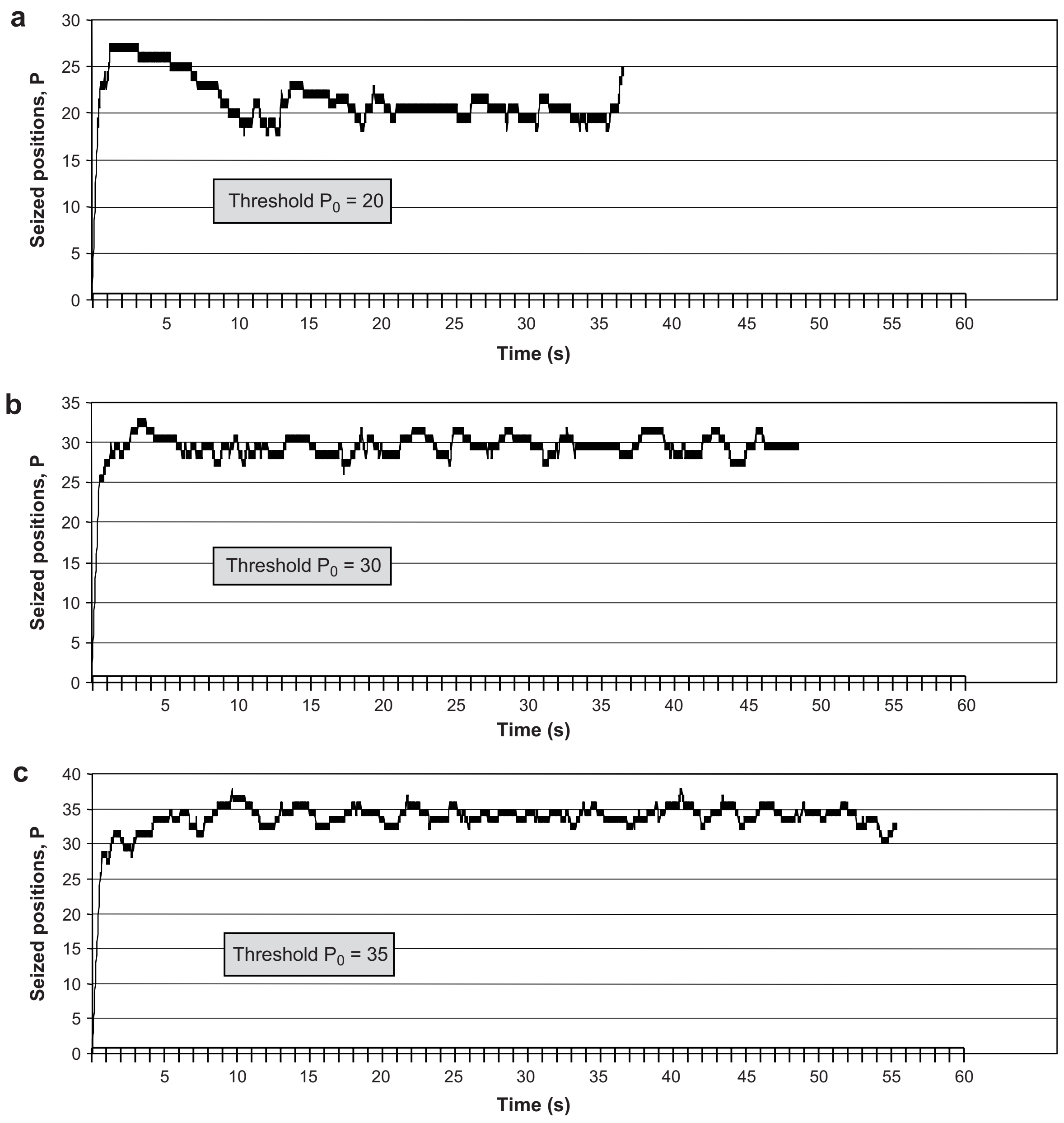


Fig. 15 – Temporal evolution of the number of seized positions, P, for three different P_0 values in an attack against a server with 40 positions: (a) $P_0 = 20$, (b) $P_0 = 30$ and (c) $P_0 = 35$.

controlled and this technique could be used for bypassing security mechanisms (e.g. mis-training an anomaly based IDS that learns normal behaviours based on recent traffic measures).

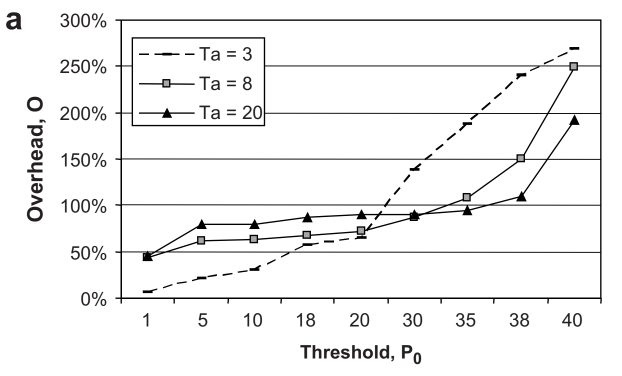
- *There is a threshold value above which the attack is not able to seize more positions*:

It can be seen that, starting from a given value for the threshold which we term *critical occupation value*, the attacker is not able to reach the number of positions specified by the threshold P_0 without considerably raising the overhead of the attack. Obviously, this is always true for threshold values greater than the number of positions in the server. But the existence of user traffic can also lead to the critical occupation value being situated below the number of positions in the server.

In the example scenario results, illustrated in Fig. 15, it can be seen that, although the threshold value $P_0 = 35$ can be easily reached, when the value is set to $P_0 = 38$ (see Fig. 17), and with 40 positions in the server, the attack is not able to reach it.

In summary, the critical occupation value is situated below the total number of positions in the server and its value depends on the legitimate users' traffic rate, decreasing as the latter increases. Note that by observing the level at which the threshold is attained, the attacker could decide whether the critical occupation value had been reached or not.

Finally, consider also that the critical occupation value might be reached (if it is below the total number of positions in the



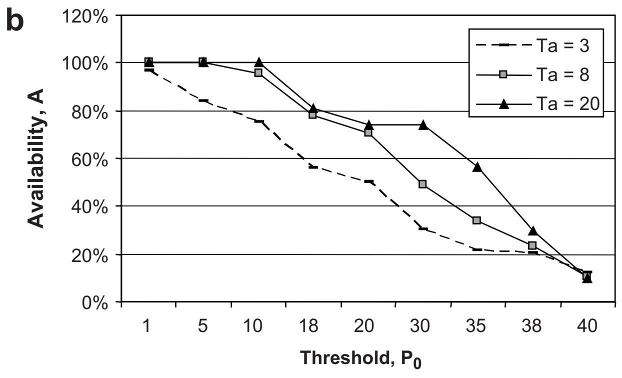


Fig. 16 – Performance of an example attack with the ISS strategy against a server with 40 positions, for different values of the seizures threshold, P_0, and the interval between arrivals of legitimate users requests, T_a: (a) overhead, O, and (b) availability, A.

server) at the cost of a considerable increase in the attack overhead. This can be seen in Fig. 16(a) where, for the same attack scenario in which the users' traffic rate has been varied, we observe that the increase in the overhead becomes higher for the values $P_0 = 20$ ($T_a = 3$), $P_0 = 35$ ($T_a = 8$), and $P_0 = 38$ ($T_a = 20$). Thus, in all cases, it is not recommendable for the attacker to surpass the critical occupation value if a high overhead is not acceptable.

6.2. *Attack procedure with the ISS strategy*

This section uses the conclusions obtained through the analysis of the ST and ITIS features to build an effective procedure for carrying out the attack.

The previous step to launching an attack is to study the server characteristics, namely to model the vulnerability to be exploited and to choose the configuration values for the attack parameters. Specifically, the basic attack period parameters, t_{ontime}, $t_{offtime}$ and Δ, should be established mainly depending on the overhead that is considered acceptable by the attacker. Additionally, the number of attack threads must be set equal to the number of positions needed, that is, equal to the value of the threshold P_0. Finally, the recovery interval, Δ_r, should be chosen in such a way that the existence of many attack threads without a position in the queue will not cause a high level of traffic. Therefore, we consider that a reasonable choice for this value is to approximate Δ_r to N_a, so that a mean value of one attack thread per second is attempting to recover a position.

Summarizing these design rules, the steps that should be followed for carrying out the attack are as follows.

Step 1:

- Initialize the attack parameters with the following values:

  $P_0 = 1$

  $N_a = 1$

  $\Delta_r = N_a$

Step 2:

- Monitor the value of the seized positions, P, during an observation time.
- If the value of P reaches P_0 and keeps oscillating around this value, wait for an *adaptation time*, T_{adapt}, and go to Step 3. This time is introduced before increasing the value of P_0, thus allowing the attacker to gradually increase the traffic rate against the server. The value of T_{adapt} will depend on the needs to launch a "slower" (in the sense of gradual) attack.

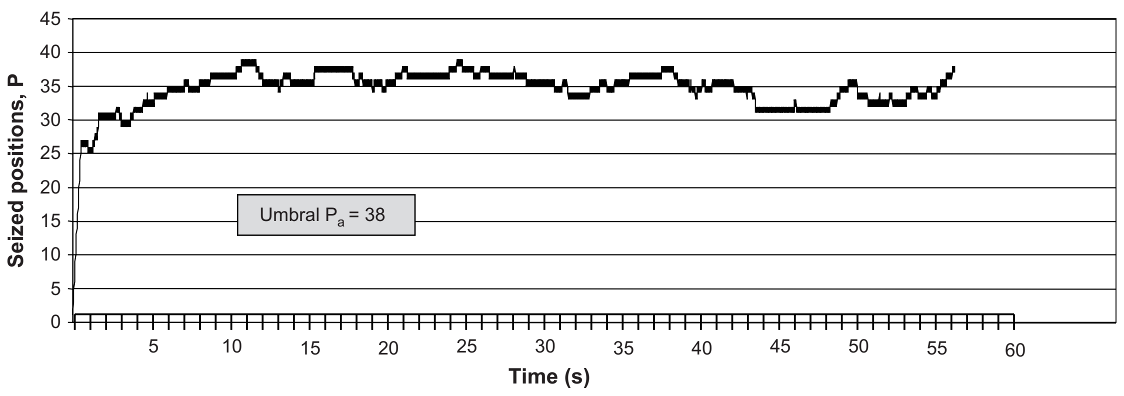


Fig. 17 – Temporal evolution of the number of seized positions, P, in an attack against a server with 40 positions with a threshold $P_0 = 38$.

- Otherwise, if the value of P does not oscillate around P_0 during an observation time, it is concluded that the critical occupation value has been reached. Go to Step 4.

Step 3:

- Update the attack parameters in this way:

$$P_0 = P_0 + 1$$

$$N_a = N_a + 1$$

$$\Delta_r = N_a$$

- Go to Step 2.

Step 4:

- Update the attack parameters in this way:

$$P_0 = P_0 - 1$$

$$N_a = N_a - 1$$

$$\Delta_r = N_a$$

- Go to Step 2.

7. Conclusions and further work

As its main contribution, this paper presents the fundamentals of a DoS attack launched against concurrent servers, which has the feature of using a low-rate of traffic. This feature could be used by an attacker for bypassing security systems that rely on the detection of traffic rate thresholds, or simply for launching the attack with fewer resources.

The existence of temporal deterministic patterns in the behaviour of a system or a protocol has been shown to represent a vulnerability upon which these kinds of attacks can be built. We provide some examples of vulnerable systems and study particularly the persistent HTTP server case.

The performance of these attacks was evaluated, yielding worrying results, due to the high efficiency they achieved. Moreover, it has been shown that the attacker could make use of techniques by which the traffic rate generated during the attack can be controlled.

A prototype of the attack has been created to show that its implementation is perfectly feasible. Moreover, the results obtained in the real scenarios show that the simulation results correctly represent the attack behaviour.

Obviously, this work needs to be continued by generating defense techniques against this kind of attacks. We are currently doing some further work on this line. First, we have checked in preliminary experiments that techniques consisting on the randomization of the service time does not work to protect servers from this attack, as it is designed in such a way that all the attack threads help each other to seize positions in the queue, and the randomization will influence more in a degradation of the service quality than in the attack effects mitigation. Moreover, we are exploring buffer management techniques in the server, which seem to work more efficiently against these attacks. In summary, the detailed description of the attack behaviour made in this paper should help in the development of defense techniques.

Acknowledgments

We would like to specially thank Aleksandar Kuzmanovic (Northwestern University of Chicago) and the anonymous reviewers for their valuable comments that have considerably increased the quality of this paper. This paper has been partially supported by the Spanish Government through MEC (Project TSI2005-08145-C02-02, FEDER funds 70%).

REFERENCES

Axelsson S. Intrusion detection systems: a survey and taxonomy. Technical report. Goteborg, Sweden: Department of Computer Engineering, Chalmers University; 2000.

Chan MC, Chang E, Lu L, Ng S. Effect of malicious synchronization, ACNS, Singapore, Jun 6–9, 2006. In: Springer Lecture Notes in Computer Science, vol. 3989; 2006. p. 114–29.

CERT Coordination Center. CERT advisory CA 1996-21, TCP SYN flooding and IP spoofing attacks. September 1996. revised November 2000, http://www.cert.org/advisories/CA-1996-21.html; 1996.

Elteto T, Molnar S. On the distribution of round-trip delays in TCP/IP networks. In: Proceedings of the 24th annual IEEE conference on local computer networks, LCN'99; 1999, p. 172–81.

Feinstein L, et al. Statistical approaches to DDoS attack detection and response. In: Proceedings of DARPA information survivability conference and exposition, vol. 1. IEEE CS Press; 2003. p. 303–14.

Fielding R, Irvine UC, Gettys J, Mogul J, Frystyk H, Berners-Lee T. RFC2068, hypertext transfer protocol – HTTP/1.1. Network Working Group January 1997.

Guirguis M, Bestavros A, Matta I, Zhang Y. Reduction of quality (RoQ) attacks on internet end-systems, INFOCOM 2005. In: 24th Annual joint conference of the IEEE computer and communications societies; 2005. p. 1362–72.

Guirguis M, Bestavros A, Matta I, Zhang Y. Reduction of quality (RoQ) attacks on dynamic load balancers: vulnerability assessment and design tradeoffs, INFOCOM 2007. In: 26th IEEE international conference on computer communications; 2007, p. 857–65.

Guirguis M, Bestavros A, Matta I, Zhang Y. Adversarial exploits of end-systems adaptation dynamics. J Parallel Distrib Comput 2007b;67(3):318–35.

Guirguis M, Bestavros A, Matta I. Explaining the transients of adaptation for RoQ attacks on internet resources. Proceedings of International Conference on Network Protocols 2004: 184–95.

Gil TM, Poleto M. MULTOPS: a data-structure for bandwidth attack detection. In: Proceedings of 10th USENIX security symposium; 2001.

Geng X, Whinston AB. Defeating distributed denial of service attacks. IEEE IT Professional 2000;2(4):36–42.

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.
Hofmann M, Beaumont LR. Content networking: architecture, protocols and practice. Elsevier, ISBN 1-55860-834-6; 2005.
Jung J, Krishnamurthy B, Rabinovich M. Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites. In: Proceedings of international world wide web conference, ACM Press; 2002, p. 252–62.
Kuzmanovic, Knightly E. Low-rate TCP-targeted denial of service attacks and counter strategies. IEEE/ACM Trans Network August 2006;14(4):683–96.
Li M. Change trend of averaged Hurst parameter of traffic under DDOS flood attacks. Computers Security 2006;25(3): 213–20.
Li M. An approach to reliably identifying signs of DDOS flood attacks based on LRD traffic pattern recognition. Computers Security 2004;23(7):549–58.
Li M, Wang S, Zhao W. A real-time and reliable approach to detecting traffic variations at abnormally high and low rates. Springer LNCS 2006;4158:541–50.
Liu C, Albitz P. DNS and BIND. O'Reilly & Associates, Inc., ISBN 1565920104; 1993.
Liu Z, Niclausse N, Jalpa-Villanueva C. Traffic model and performance evaluation of Web servers. Performance Evaluation 2001;46(2–3):77–100.
Maciá-Fernández G, Díaz-Verdejo JE, García-Teodoro P. Evaluation of a low-rate DoS attack against iterative servers. Comput Networks 2007;51(4):1013–30.
Mirkovic J, Dietrich S, Dittrich D, Reiher P. Internet denial of service. Attack and defense mechanisms. Prentice Hall, ISBN 0-13-147573-8; 2004.
Mirkovic J, Reiher P. A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Comput Commun Rev 2004; 34(2):39–53.
Network Simulator 2. Available at: http://www.isi.edu/nsnam/ns/.
Scherrer, Larrieu N, Owezarski P, Borgnat P, Abry P. Non-Gaussian and long memory statistical characterizations for Internet traffic with anomalies. IEEE Trans Depend Secure Comput 2007;4(1): 56–70.
Shevtekar A, Ansari N. A router-based technique to mitigate reduction of quality (RoQ) attacks. Comput Networks 2008; 52(5):957–70.
SANS Institute. NAPTHA: a new type of denial of service attack; 2001.
Siris VA, Papagalou F. Application of anomaly detection algorithms for detecting SYN flooding attacks. Comput Commun 2006;29(9):1433–42.
Song T. Fundamentals of probability and statistics for engineers. John Wiley & Sons, ISBN 0-470-86813-9; 2004.
Wang H, Zhang D, Shin K. Detecting SYN flooding attacks. In: Proceedings of 21st joint conference on IEEE computer and communications societies (IEEE INFOCOM), IEEE Press, 2002; p. 1530–39.
Zaki MJ, 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 '96); 1996. p. 282–91.

Gabriel Maciá-Fernández is an Assistant Professor in the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He received a MS in Telecommunications Engineering from the University of Seville, Spain, and got a Ph.D in 2007 from the University of Granada. From 1999 to 2005 he worked as a specialist consultant in 'Vodafone Spain'. His research was initially focused on multicasting technologies but he is currently working on computer and network security, especially in the field of intrusion detection and response systems, denial of service, web security and secure protocols design.

Jesús E. Díaz-Verdejo is Associate Professor in the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He received his B.Sc. in Physics (Electronics speciality) from the University of Granada in 1989 and has held a Ph.D. grant from Spanish Government. Since 1990 he is Associate Professor at this University. In 1995 he obtained a Ph.D. degree in Physics. His initial research interest was related with speech technologies, especially automatic speech recognition. He is currently working in computer networks, mainly in computer and network security, although he has developed some work in telematics applications and e-learning systems.

Pedro García-Teodoro received his B.Sc. in Physics (Electronics speciality) from the University of Granada, Spain, in 1989. This same year he was granted by "Fujitsu España", and during 1990 by "IBM España". Since 1989 he is Associate Professor in the Department of Signal Theory, Telematics and Communications of the University of Granada, and member of the "Research Group on Signal, Telematics and Communications" of this University. His initial research interest was concerned with speech technologies, especially automatic recognition, field in which he developed his Ph.D. Thesis in 1996. From then, his profile has derived to that of computer networks, and although he has done some works in telematics applications and e-learning systems, his main current research line is centred in computer and network security.