SlowComm: Design, Development and Performance Evaluation of a new Slow DoS Attack

Maurizio Aiello[a], Enrico Cambiaso[a,*], Gianluca Papaleo[a]

[a]*IEIIT-CNR, National Research Council - via De Marini, 6 - 16149 - Genoa, Italy*

Abstract

Internet transformed itself from a useful tool for communicating information to a pervasive and necessary infrastructure for modern economy. Accordingly, due to economic motivations, it became an arena for cyberwarfare and cybercrime. In this paper, we present the novel threat called SlowComm and we analyze that it can successfully lead a DoS on a targeted system using a low amount of attack bandwidth. We also analyze that the proposed attack is not bounded to a specific protocol and should be considered a *protocol independent* attack, proving the ability it has to affect different Internet services.

Keywords: denial of service; lbr dos attack; cyberwarfare; slow dos attack; protocol independence

1. Introduction

In the last years, the advent of the Internet has made it the most important communication medium of the world. Because of this, Internet has to be kept a safe place, in order to provide a secure communication environment to its users.

In the arena of Internet attacks, *cyberwarfare* operations are executed in order to create an internal or external damage to the network, indirectly targeting the real/physical world too. In particular, cyberwarfare term refers to politically

*Corresponding author

Email addresses: `maurizio.aiello@ieiit.cnr.it` (Maurizio Aiello), `enrico.cambiaso@ieiit.cnr.it` (Enrico Cambiaso), `gianluca.papaleo@ieiit.cnr.it` (Gianluca Papaleo)

motivated hacking operations aimed to sabotage or espionage of an adversary information system. Currently, cyberwarfare operations equip governments with an effective and powerful opportunity to counter their adversaries, even without being a superpower. Indeed, cyberwarfare attacks only require hardware and software weapons.

Among all the methodologies used to successfully execute a cyberwarfare operation, denial of service attacks (DoS in the following) are executed to exhaust victim's resources, compromising the targeted systems' availability, thus affecting reliability for legitimate users. In case more than one attacking machine are (willing or not) involved in the same attack, a Distributed DoS (DDoS in the following) attack is executed (thus forming a *botnet*). Such approach is often adopted since it provides additional resources to the attacker, thus causing a more relevant damage on the victim system. Historically, we have assisted to different well known DDoS attacks. In 2000, several companies including some giants of the Internet such as eBay, Amazon or Yahoo have been targeted by DDoS attacks [1]. Instead, in 2009, a Slowloris based attack has been executed against the Iranian presidential elections [2]. More recently, in March 2013, the most imponent DDoS attack has been launched against an anti-spam company known as Spamhaus [3].

There are several approaches which can lead to a denial of service (e.g. physical attack, exploit, flooding, etc. . .). Over all the approaches, we are interested in analyzing Slow DoS Attacks (SDAs in the following), also known as Low-Bandwidth Rate DoS (LBR DoS in the following) attacks, which represent the second generation of network based DoS attacks. Particularly, the first generation of such threats was based on flooding the victim with a large amount of data, until its resources are overwhelmed and a DoS is reached. Differently, the characteristic of a LBR DoS attack is to be able to cause a DoS using a low amount of attack bandwidth [4]. Because of this, the resources needed to successfully execute such attacks are reduced, thus making it a more dangerous threat, as it could also be executed even from non performant devices.

In this paper, we present a Slow DoS Attack able to lead a DoS requiring a tiny amount of attacking bandwidth. In Section 2 we report the related work on the topic. Section 3 describes instead how the attack works, while in Section 4 we report implementation issues for the proposed attack. In Section 5 we report an experimental test bed analyzing the ability the attack has to affect different protocols. We then introduce important metrics for representing the success of an attack in Section 6. In Section 7 we report the obtained results accordingly to different implementation techniques. Finally, in Section 8 we report the conclusions of the paper and possible extension to the work.

2. Related Work

Although several scientific studies are focused on network threats, a minority is related to the development of novel threats. Indeed, most of the works focus on providing a detection system: in [5] a defense mechanism against DDoS attacks has been provided, based on a targeted filtering of network packets. Instead, [6] introduces a novel solution for limiting the amount of distributed attacks traffic, while [7] provides a defence mechanisms against the SYN flood attack. Relatively to slow DoS menaces, in [8] a protective solution for cloud systems has been introduced, relatively to HTTP-DoS and XML-DoS attacks.

Instead, if we analyze the evolution of Denial of Service attacks, the first generation of menaces was based on flooding the victim with a high amount of packets: a prominent flooding-based threat is the already cited *SYN flood* attack [9], exploiting the TCP three-way handshake mechanism to maintain half-open connections with the victim. *UDP flooding* and *ICMP flooding* attacks [10] send a large amount of UDP/ICMP packets to generate a network link congestion or system overload. A similar attack is *Smurf* [11], which sends spoofed echo requests to a broadcast address, forcing a large amount of hosts to send echo reply packets to the spoofed victim address.

Botnet tools such as LOIC, BlackEnergy, or Cythosia are used to execute different (flooding based or not) distributed attacks, exploiting the resources of multiple attacking hosts instead than executing the attack from a single machine.

Considering the second generation of DoS attacks, in [12] a study on the slow DoS field is provided, analyzing the current menaces and providing a taxonomy of them. A first categorization is based on the attack feasibility, distinguishing *practical attacks* from *meta attacks*: in the first case a definition of a specific and directly implementable set of executable actions is provided; instead, a meta attack abstracts the attack without bounding it to a specific implementation, defining a set of guidelines aimed at leading a DoS on the victim. A different categorization is instead based on the exploited resource, providing an accurate taxonomy of the current menaces.

The first SDA threat is the *Shrew* attack [13], designed to send an attack burst to the victim, deluding it (at the transport layer) that a high congestion is affecting the network link. Mácia-Fernández et al. [14] propose instead the *LoRDAS* attack, which is focused on concentrating the attack burst to specific instants, thus reducing the bandwidth needed for the attack.

The most popular SDA could be considered the already mentioned *Slowloris* attack [15], which exploits the HTTP protocol establishing a large amount of pending requests with the targeted web server. An attack similar to Slowloris is *Slow HTTP POST* [10], essentially varying the requests payload.

The *Apache Range Header* attack exploits the byte-range parameter of HTTP, forcing the web server to replicate in memory the requested resource, thus potentially delaying the response arrival time. Although it is currently mitigated, this attack represented a serious threat to Apache running servers.

Unlike previous threats, the *Slow Read* SDA slows down the responses of a web server, simulating a really small client side reception buffer and slowly receiving the packets coming from the server.

Another interesting attack is the *XerXes* menace for executing an application layer DDoS attack distributing it through the Tor network. A first version of the attack, implemented by a user known as The Jester, has been publicly released as a source code. Nevertheless, its behavior is similar to an HTTP flooding attack, since attack resources are not optimized. A second version of the attack has been announced (and never released) through a viral video on the web [16].

In a previous work [17] we have presented the SlowReq Slow DoS Attack. The SlowComm attack proposed in this paper has to be considered as a relevant extension to that work. Indeed, this paper extends the previous work introducing a novel and optimized menace. In order to optimize it, we also introduce proper metrics and a network traffic modelization. Moreover, we provide an accurate study relatively to the protocol independence feature of SlowComm.

3. Attack Description

The attack proposed in this paper, called SlowComm, is a *Long Request DoS* attack [12]. It exploits a vulnerability on most server applications implementations, which limit the number of simultaneous threads on the machine. Unlike flooding DoS attacks, which aim to overwhelm some network parameters of the victim host, SDAs adopt a smarter approach, seizing all the available connections with the application listening daemon. This approach requires a very limited amount of attack bandwidth.

Particularly, SlowComm sends a large amount of slow (and endless) requests to the server, saturating the available connections at the application layer on the server while it is waiting for the completion of the requests. As an example we refer to the HTTP protocol, where the characters sequence `\r\n\r\n` represent the end of the request: SlowComm never sends these characters, forcing the server to an endless wait. Additionally, the request is sent abnormally slowly. Similar behavior could be adopted for other protocols as well (SMTP, FTP, etc.). As a consequence, applying this behavior to a large amount of connections with the victim, a DoS may be reached.

The requests sent to the server must be considered slow in terms of “little amount of bytes sent per second”. Indeed, for the proposed attack this value approaches an extremely low value: the attack bandwidth requirements for the executed tests are less than 1 KB/s.

Analyzing how the attack works, since the attacker's purpose is to seize all the available connections on the targeted host, we could assume that a DoS is reached some instants after the attack is launched. Nevertheless, the server may have already established active and legitimate connections with other clients. Those connections are working until they are closed. As soon as a connection closure happens, its relative resources on the server are released, thus allowing clients to establish new incoming connections. Because of this, the purpose of the attacker is to replace all the freed connections with malicious ones. While doing that, there could be a race condition between the attacker and some other legitimate clients. Nevertheless, we could assume that sooner or later the attacker would obtain the connections, since it would repeatedly try to connect to the victim with an intelligent algorithm, turning aggressiveness and stealthiness of the attack.

From the stealth perspective, the proposed attack is particularly difficult to detect while it is active, since log files on the server are often updated only when a complete request is received or a connection is closed: being our requests typically endless, during the attack log files don't contain any trace of attack. Therefore, a log analysis can't produce an appropriate warning in reasonable times.

4. Attack Implementation

The SlowComm attack is composed by three components:

- *connect component*: establishes the connections with the server, thus opening a large amount of connections, without sending any data to the server
- *maintain component*: maintains the connections with the server alive by slowly sending urgent data to the victim through the established connections, preventing a server connection close. The exploited resource, according to taxonomy reported in [12] is the request timeout
- *check component*: detects the connections which have been closed by the server, in order to make the first component seize those connections again

The maintain component makes use of the *Wait Timeout* parameter introduced in [12] to manage the slowness of sending. In particular, for each connection the attacker sends a single character (default one is a single space; in general, each character is good) as the Wait Timeout expires. Then, the Wait Timeout is restarted. Moreover, since relatively to most Linux hosts `ACK` packets not followed by a packet including a payload are ignored by TCP [18], the maintaning phase is instantly executed after a connection has been established, thus immediately sending to the server a single character.

Instead, the check component repeatedly checks the status of the established connections, in order to notify the connect component in case a connection close has happened. Thanks to this component, the attack is able to autonomously re-establish and maintain during the attack a fixed amount m of connections with the server.

Assuming the m value is equal to the maximum number of connections accepted by the server and assuming that the server is vulnerable to a Long Request DoS attack, we will analyze in the next section how the server may react to the attack.

5. Experimental Test Bed

We have executed two different sets of experimental tests. Once per second, the number of connections established with the application server has been monitored. Such approach provides us an accurate way to report the attack status/success during the time.

5.1. HTTP Tests

We have attacked an Apache 2 web server running on a Linux based host. The choice of Apache is driven by the fact it is one of the most common web server daemons [19].

Trials have been executed comparing different attacks by changing the configuration on targeted system, enabling one of the following modules:

- `mod-security`, a module for managing the security of a web server; in particular, from version 2.5.13, a new feature has been implemented, focusing on pending requests based Slow DoS Attacks prevention: indeed, the new directive `SecReadStateLimit` can be used to reduce the number of threads in the state `SERVER_BUSY_STATE` for each IP address.

- `reqtimeout`, a module which provides the opportunity to set temporal and bandwidth limits for the received requests.

Even if other modules may be used to reach similar results (such as the `limitipconn` module, which behaves in our case similarly to `mod-security`), we have selected these two modules because they are widely used and tested and probabily the most effective and immediately available on Linux platforms.

The Apache server has been configured to serve at most $r_{max} = 150$ simultaneous connections (through a directive of Apache). This value often represents the default configuration value. Having a known r_{max} value, by checking the connections status on the targeted server, we are able to analyze the attack success during the time in a detailed way, thus being able to identify a *full DoS* on the attacked server.

During our tests the attack has been configured to create at most $n = r_{max}$ connections with the victim. By chosing this value, we are able to analyze if a DoS is reached on the victim, requiring to the attacker the minimum amount of resources potentally able to lead a DoS on the targeted server.

We have executed the attack against a server equipped with the modules described above, enabling them one at time on the victim host. We have also analyzed how the proposed tool behaves against a "pure" web server, without any protection module.

In particular, for `mod-security` we have configured it to serve at most $r_{sim} = \frac{r_{max}}{10} = 15$ simultaneous connections for each IP address. In general, as long as $r_{sim} < r_{max}$, we don't expect a DoS on the victim. In any way, our desire is to analyze the behavior of the attack.

The `reqtimeout` module has been instead configured to wait at most $T_i = 20$

seconds for the first byte of the request header. From then, a minimum bandwidth rate in reception of $B_{min} = 500$ B/s is required. Moreover, connections are closed after a wait of $T_f = 40$ seconds in total.

We choose a $T = 600$ seconds long attack and a Wait Timeout $T_{WT} = 60$ seconds, lower than the request timeout on the server, configured at $T_S = 300$ seconds, which represents the default value on Apache web servers. Therefore, we expect that connections are not closed by the server through its request timeout.

Figure 1 reports the results of the experiments we have executed on the Apache protection modules described above.

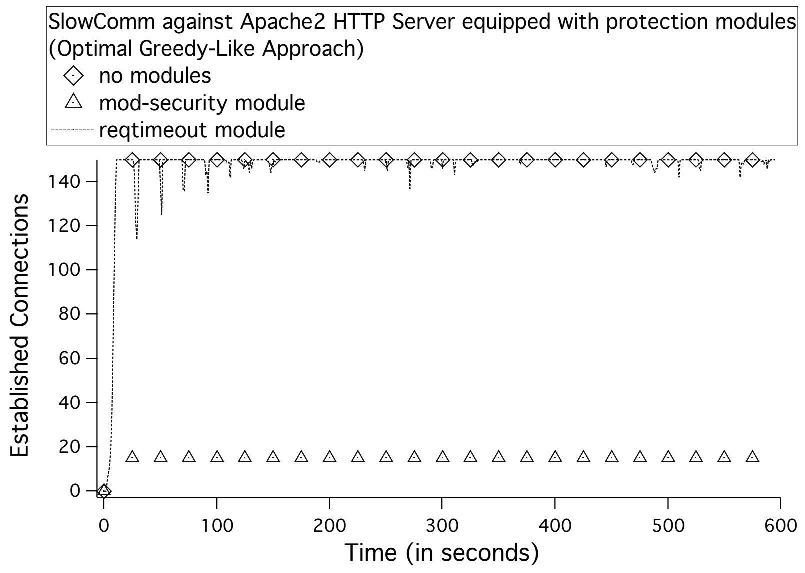

Figure 1: SlowComm Experiments Against an Apache2 Web Server

As shown in figure, the attack is completely successful if no protection modules are running. Indeed, in this case all the available connections on the victim are established by the attacker after a few seconds, and they are maintained active for the whole attack duration. Conversely, `mod-security` can successfully reduce the attack impact on the server. Nevertheless, in this case a distributed attack surely leads to a DoS, since the behavior of this module can't mitigate a running DDoS executed by m agents, when $r_{max} \leq m \cdot r_{sim}$. Finally, regarding `reqtimeout`, each connection is closed by the server after T_i seconds of inactivity: however, the attack is able to detect a server-side connection close, thus re-establishing the connections, leading to another DoS. Therefore, we can state that this module can't effectively mitigate a SlowComm attack: as can be seen in Fig. 1, connection closures are detected and re-established by the attacker, thus reaching the DoS state again.

5.2. Non-HTTP Protocols Tests

The second test set has been focused on analyzing how the attack behaves targeting protocols different from HTTP. Indeed, conversely from other slow DoS threats, SlowComm sent messages are not bounded to a specific protocol, since they cointain a single repeated character. Many server implementations are in fact designed to parse messages only after an entire line or a full request are received.

In order to prove the ability to affect different protocols without requiring amendments to the SlowComm implementation, we have targeted both FTP and MySQL services, attacking in particular *proftpd* and *postfix* daemons running on a Linux based host.

Analougously to previous experiments, we have limited the number of the maximum connections accepted by both FTP and SMTP servers to $r_{max} = 100$, which has to be considered a high value: in case of FTP, the default proftp value is 30, while in case of SMTP, the default value is 50. Then, we have executed a SlowComm attack against them, for $T = 600$ seconds, using the same Wait Timeout of $T_{WT} = 60$ seconds adopted in previous tests.

As shown in Figure 2, the attacks are both successful: in both the cases the DoS state is reached after a few seconds and maintained during the time, although a server side timeout of proftpd automatically closes the connections after almost 300 seconds, returning a `421 Login Timeout` error.

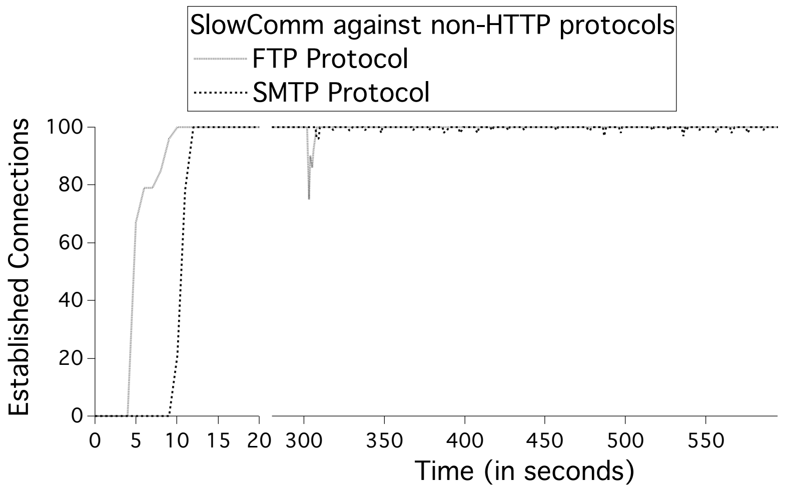

Figure 2: SlowComm Experiments Against FTP and SMTP Services

6. Metrics Definition

Analyzing Figure 1 relative to the experimental tests on HTTP protocol, it's possible to distriminate between three different cases:

(i) in case no modules are installed on the server, once the DoS is reached, it is continuously maintained,

(ii) if the `mod-security` module is installed on the server, the DoS is never reached, until $r_{sim} < r_{max}$, finally

(iii) if the `reqtimeout` module is installed on the server, an alternation of DoS and non-DoS state occurs.

Our intent here is to define various metrics which define if an attack is better than another one, under certain conditions.

6.1. Network Traffic Representation Model

Before defining metrics characterizing the success of an attack, we have first to analyze the communication between attacker and victim, by representing the connection establishment process through a sequence-like diagram. In particular, accordingly to Figure 3, we define the following time variables:

- δ the time needed to the attacker to send a packet to the victim, and vice versa,
- ε the time passed between the establishment of two sequential connections,
- n the number of simultaneous connections established with the victim by the attacker (in order to reach a DoS, $n \geq r_{max}$ is usually needed),
- ϕ the time needed to the attacker to detect a connection closure, once the connection-close packet is arrived.

Relatively to the δ, ε, and ϕ parameters, it's clear that their values are variable. Nevertheless, for the sake of simplicity, we can represent them as fixed (average) values, without losing in generality on the model.

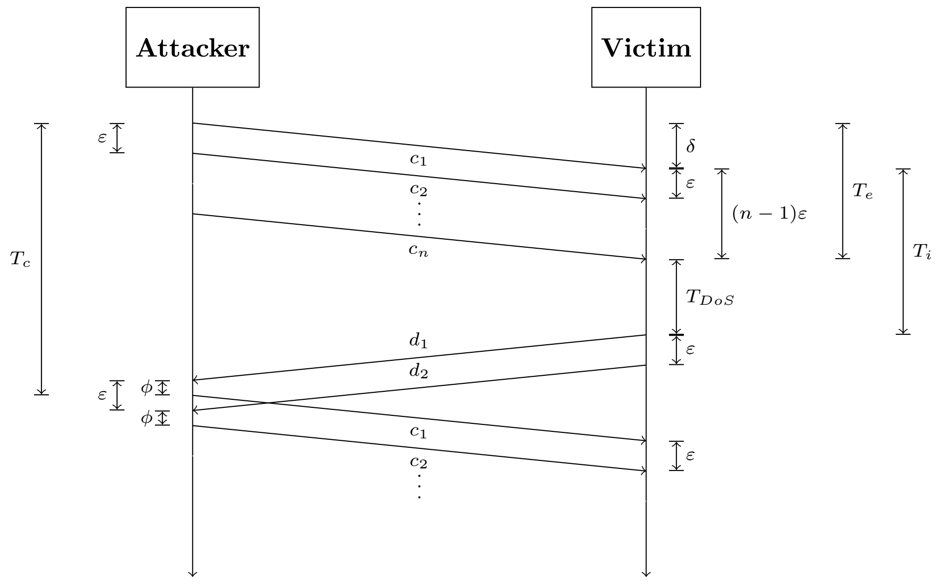

Figure 3: Connections Establishment During a SlowComm Attack

As reported in Figure 1, in case connections are closed indiscriminately (such as under `reqtimeout` conditions), behavioral *cycles* are experienced. In particular, let's define T_e the time needed to reach a DoS from the beginning of the cycle as reported in Equation 1.

$$T_e = \delta + (n-1) \cdot \varepsilon \tag{1}$$

Moreover, let's define T_i as the average connection duration. In case connections are never closed by the victim, we will potentially have $T_i \to \infty$, since after a connection has been established, it is maintained alive for the whole attack duration. On the other side, in case connections are closed after some sort of timeout expiration, as happens in case of a server equipped with `reqtimeout`, the T_i parameter will typically assume a fixed value.

During the attack execution, in case of a finite T_i , we can analyze that a series of cycles are experienced, where each cycle lasts T_c seconds, defined as reported in Equation 2.

$$T_c = 2 \cdot \delta + T_i + \phi \tag{2}$$

On the other side, in case of an infinite T_i , due to Equation 2, we also have an infinite T_c.

At this point, we can define T_{DoS} as the DoS duration for a single cycle, as reported in Equation 4.

$$T_{DoS} = T_i - (n-1) \cdot \varepsilon \tag{3}$$

Since from 4 T_i can be defined in function of T_{DoS}:

$$T_i = T_{DoS} + (n-1) \cdot \varepsilon \tag{4}$$

We can finally redefine T_c as reported in Equation 5.

$$T_c = T_e + T_{DoS} + \delta + \phi \tag{5}$$

The introduced parameters will be now used to define metrics characterizing the success of an attack.

6.1.1. Maximum Attack Peak

Let's define c_t the number of connections established with the attacker at time t. Let's define $f(t) = c_t$ the function representing the connections established by the attacker during the attack execution time.

We can define the maximum attack peak M_{peak}, defined as follows:

$$M_{peak} = max(f) \tag{6}$$

This value provides a numerical value to the maximum damage created by the attack against the targeted server. Moreover, it gives the attacker a quantitative measure for identify how many attacking machines at minimum have to simultaneously execute the attack in order to lead a DoS on the victim, without considering the DoS duration.

6.1.2. Attack Influence

The success of an attack could be defined in function of the integral of the connections established with the attacker during the attack execution time. In particular, assuming n the number of creations the attackers needs to establish, and being T the attack duration, we define η the attack influence on the targeted server, as reported in Equation 7.

$$\eta = \frac{1}{n\ T} \int_0^T f(t)\, dt \tag{7}$$

This value provides the overall damage created by the attack on the targeted server. In particular, since $0 \leq \eta \leq 1$, this value effectively represents the ability the attack has to reach the goal of the attacker. Indeed, since the attacker's purpose is to constantly seize a certain amount n of connections with the victim's server, for $\eta = 1$ the goal is reached from the beginning of the attack to the end. Conversely, for $\eta = 0$ the goal is never reached. For other η values, the goal is partially reached: in this case, the average number of connections established during the attack is $\bar{n}$, defined as reported in Equation 8.

$$\bar{n} = \eta \cdot n = \frac{1}{T} \int_0^T f(t)\, dt \tag{8}$$

In reality, it's important to notice that, due to physical limits bounded to packets transmission, we always have a first attack period where the server does not experiences a DoS. In particular, in real conditions we always have:

$$T_e > 0 \tag{9}$$

Consequently, in the initial period there surely exists a t such as $f(t) < n$. Therefore, in real conditions, analyzing the whole attack duration, we always have $\eta < 1$.

Instead, if we consider the attack influence after connections have been established by the attacker, we have $\eta \leq 1$. Therefore, let's define η_τ the attack influence limitedly to the time starting from τ and ending with T. In particular, we have $\tau \leq T$. Moreover, we can now generalize definition of η defining it in function of the starting period to consider, as reported in Equation 10.

$$\eta(\tau) = \frac{1}{n\ (T - \tau)} \int_{\tau}^{T} f(t)\, dt \tag{10}$$

At this point, we can combine Equation 7 with Equation 10 defining $\eta = \eta(0)$. Moreover, while we have $\eta < 1$, we have $\eta(\tau') \leq 1$ for each $\tau' \geq T_e$.

These definitions may be used to compare different attacks, in order to identify the influence of the menaces during the whole attack duration or during specific temporal limits. Indeed, it may be interesting compare two different threats by analyzing the effects on the targeted server, not only relatively to the whole attacks duration, but starting from the time a DoS is reached, until the end of the attack. In this case, even if one attack may reach the DoS slowly, the DoS may be maintained for a longer time.

6.1.3. DoS Percentage

In case connections are closed (such as when `reqtimeout` is installed on the victim), we want to properly define a percentage of DoS experienced on the server during the attack execution time.

Therefore, relatively to a single cycle, we can finally define P_{DoS} the percentage of DoS, as reported in Equation 11.

$$P_{DoS} = \frac{T_{DoS}}{T_c} \tag{11}$$

7. Obtained Results

We have executed tests by varying the attack implementation, adopting different approaches. We will now describe in detail the implementative techniques used.

7.1. Implementation Techniques

Analyzing the attacking results obtained after the execution of a first version of the SlowComm attack, we have noticed an interesting behavior in case `reqtimeout` module is enabled on the Apache server. As a consequence, we have decided to adopt different implementative approaches during the attack execution. Relatively to a `reqtimeout` condition, we will now describe in detail the implementation techniques we have integrated into the tests.

7.1.1. Non Optimal Greedy-Like Implementation

Our first implementation adopted a "non optimal greedy-like" approach, designed to re-establish connections as soon as a closure is identified by the attacker, maintaining the three-flows based implementation described in Section 4. Moreover, we have introduced short delays to reduce the computational resources needed to execute the attack. Being this balancing choice driven by the fact that we believe that such menaces may also be executed from non-performant hosts, we have preferred to maintain the execution as light as possible. Nevertheless, we have realized that such approach induces inefficiency. Indeed, although for short time analysis the DoS state is reached at any execution cycle, this is not true for long execution times. In particular, T_{DoS} values decrease almost at any execution cycle and after some time a full DoS is not reached anymore (note that during our tests $n = r_{max}$). This fact is highlighted in Figure 4, where it's possible to analyze that the DoS duration between first execution cycles is longer than the DoS duration related to final cycles of our tests. Indeed, in figure it's possible to notice that delays introduced in a cycle are replicated to the next cycles, thus maintaining and replicating them for the whole attack duration.

In order to solve this problem, we have decided to introduce two additional implementations, with the purpose of comparing different approaches applicable to the same attack.

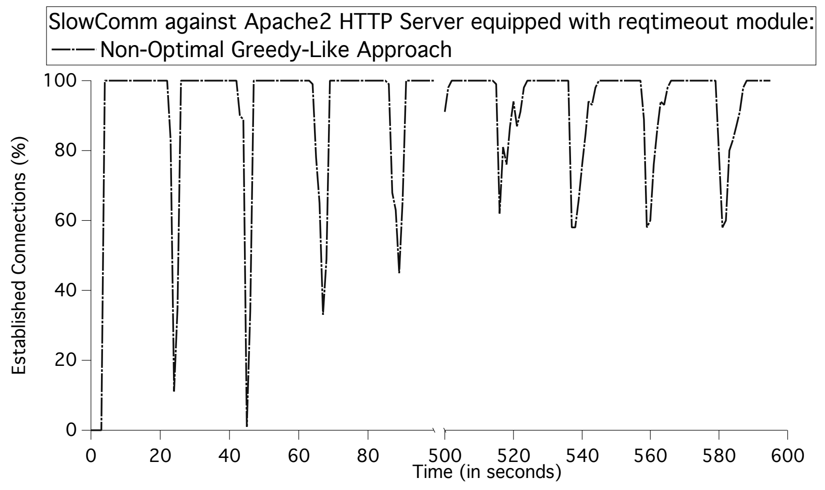

Figure 4: SlowComm Non-Optimal Greedy-Like Approach Execution Against an Apache2 Web Server equipped with `reqtimeout` module

7.1.2. Optimal Greedy-Like Implementation

Therefore, we have decided to implement an "optimal greedy-like" approach. Unfortunately, although it's possible to define a pure greedy algorihm, hardware and software limits prevent a pure greedy execution. Even if the optimal greedy-like approach represents in our case an ideal solution, due to the low requirements of the proposed attack, it's often a good choice to balance attack efficiency with resource consumption. In particular, the optimal greedy-like algorithm we have implemented treats each connection separately on a specific program execution flow, while an additional maintaining flow is used. Moreover, unlike in our non optimal greedy-like implementation, in this case we don't use delays. Therefore, accordingly to Figure 3, we typically have in this case reduced ϕ values.

7.1.3. Lazy-Like Implementation

We have also chosen to adopt a different approach. Particularly, we implemented a "lazy-like" algorithm, with the purpose of breaking the relationship between DoS durations during execution cycles. Indeed, adopting the lazy-like approach, two T_{DoS} values related to two different execution cycles are uncorrelated. Moreover, we have maintained the implementation based on the three components described above. In particular, after detecting a connection closure, the lazy-like algorithm waits for all the connections to be closed by the server, thus re-establishing them. Therefore, in case a delay is experienced, such as reported in Figure 4, it will only affect the current execution cycle and it will not be propagated during the whole attack duration.

Indeed, since connection closures delays are not related to connection establishments accomplished at the next execution cycle, the delay is not maintained during the attack execution. As a consequece, the DoS leaded on the victim potentially maintains the same characteristics of the initial DoS.

Figure 5 reports the traffic representation model relatively to a lazy-like algorithm.

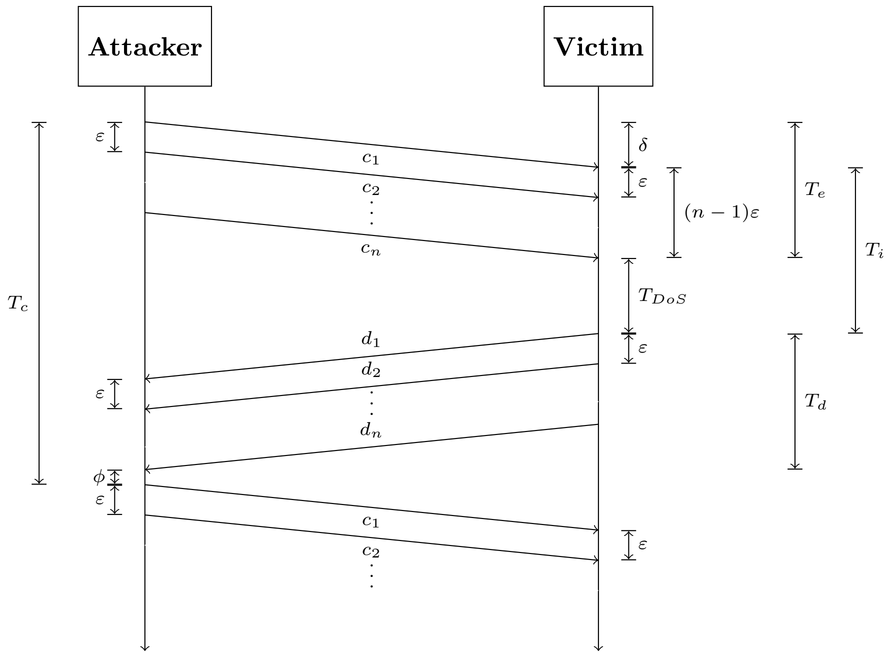

Figure 5: Connections Establishment During a SlowComm Attack Adopting a Lazy-Like Algorithm

In this case, it may be relevant to consider T_d , defined as the time needed to the victim to destroy the connections established by the attacker, as reported in Equation 12.

$$T_d = T_e \tag{12}$$

Moreover, cycle time T_c value is defined as reported in Equation 13.

$$T_c = T_e + T_{DoS} + T_d + \phi \tag{13}$$

Hence, due to Equation 12:

$$T_c = 2 \cdot T_e + T_{DoS} + \phi \tag{14}$$

Comparing Equation 14 with Equation 5, since, due to Equation 1, $T_e > \delta$, in this case the cycle times are longer than in case a greedy-like approach is adopted.

7.2. Experimental Results

We have executed trials for the implementation techniques described above. We have analyzed the results of the three implementation techniques by using different r_{sim} values (and defining $n = r_{sim}$): 10, 100, and 1000. In each case, we have analyzed the following values: M_{peak}, P_{DoS}, and η. Moreover, we have also analyzed how the Slowloris attack [15] behaves under these conditions, to highlight the enhancements carried out by SlowComm.

Results are reported on Table 1.

Table 1: Summary of the Obtained Results for Different Implementations

n	**Parameters**	**SlowComm**			**Slowloris**
		Opt. Greedy	**Non Opt. Greedy**	**Lazy**	
10	M_{peak}	10	10	10	10
	P_{DoS}	0.99329	0.9396	0.98993	0.13591
	η	0.99446	0.9849	0.99161	0.13591
100	M_{peak}	100	100	100	100
	P_{DoS}	0.92114	0.68289	0.87584	0.092282
	η	0.98399	0.92032	0.93148	0.13742
1000	M_{peak}	1000	1000	1000	252
	P_{DoS}	0.25168	0.78356	0.82886	0
	η	0.95418	0.9108	0.90732	0.034277

It's possible to notice that for all the tests we have executed, SlowComm results are better than Slowloris ones. In particular, for $n = 10$ and $n = 100$

the optimal greedy-like implementation is preferrable. Nevertheless, for high n values, the algorithm become computationally heavy and it is no longer working better than other SlowComm approaches. This is given by the fact that in this case more program execution flows are executed in a “non-stop” way. Indeed, in case of a distribute attack, if only M_{peak} and η values are considered, the optimal-greedy approach may still result the better choice. Nevertheless, since such threats may also be executed from non-performant hosts with limited processing power, attack resources consumption should also be evaluated. Therefore, in case of high n values, the optimal greedy-like approach should be discarded, if the attack is not distributed.

8. Conclusions

In this paper we have presented a novel Slow DoS Attack which may be potentially used for cyberwarfare operations. We have described in detail how the attack works, testing its ability to lead a DoS on a server, analyzing different servers configurations and different service types. We have also considered different implementation approaches, evaluating how the attack success varies for different algorithms, deeply analyzing server configurations able to mitigate the attack or trying to counter it. We have introduced three different implementative approaches in order to test the attack success by varying the adopted approach, through parameters we have introduced with the aim of defining the success of an attack. We have also compared our approaches to a current menace known as Slowloris, demonstrating that the proposed attack is more successful from any point of view.

During the execution of the tests, we realized that under some circumstances, the proposed attack is not able to maintain a full and continuous DoS on the victim, a possible extension to the work may be focused on making SlowComm a mixed attack, altering its behavior and making it also behave like a LoRDAS attack [14]. In this case, once a connection closure pattern is detected by the attacker, a small attack burst would be executed just before the closure happen, thus maintaining a full and continuous DoS on the victim.

In this paper we have also analyzed how the attack behaves with different protocols, showing that some sort of *protocol independence* feature occurs. Even if this feature is common on flooding DoS attacks, due to the fact they work at IP level, it's a novel concept relatively to SDAs, usually designed to affect a specific application protocol. Therefore, further work will be focused on an accurate analysis of the protocol independence feature for Slow DoS Attacks.

References

[1] e. a. F. Lau, "Distributed denial of service attacks," *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 3, pp. 2275–2280, 2000.

[2] e. a. L. C. Giralte, "Detecting denial of service by modelling web-server behaviour," *Computers & Electrical Engineering*, 2012.

[3] B. Genge and C. Siaterlis, "Analysis of the effects of distributed denial-of-service attacks on mpls networks," *International Journal of Critical Infrastructure Protection*, 2013.

[4] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks and counter strategies," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 4, pp. 683–696, 2006.

[5] e. a. S. Chen, "Stateful ddos attacks and targeted filtering," *Journal of network and computer applications*, vol. 30, no. 3, pp. 823–840, 2007.

[6] V. A. Siris and I. Stavrakis, "Provider-based deterministic packet marking against distributed dos attacks," *Journal of Network and Computer Applications*, vol. 30, no. 3, pp. 858–876, 2007.

[7] e. a. H. Safa, "A collaborative defense mechanism against syn flooding attacks in ip networks," *Journal of Network and Computer Applications*, vol. 31, no. 4, pp. 509–534, 2008.

[8] e. a. A. Chonka, “Cloud security defence to protect cloud computing against http-dos and xml-dos attacks,” *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1097–1107, 2011.

[9] e. a. H. Wang, “Detecting syn flooding attacks,” *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1530–1539, 2002.

[10] e. a. S.-j. Kim, “Ddos analysis using correlation coefficient based on kolmogorov complexity,” in *Grid and Pervasive Computing*, pp. 443–452, Springer, 2013.

[11] e. a. A. Kotkar, “Network attacks and their countermeasures,” *Network*, vol. 1, no. 1, pp. 85–89, 2013.

[12] e. a. E. Cambiaso, “Slow dos attacks: definition and categorisation,” *International Journal of Trust Management in Computing and Communications - In press article*, 2013.

[13] A. Kuzmanovic and E. W. Knightly, “Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants,” *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 75–86, 2003.

[14] e. a. G. Macia-Fernandez, “Evaluation of a low-rate dos attack against iterative servers,” *Computer networks*, vol. 51, no. 4, pp. 1013–1030, 2007.

[15] K. Sourav and D. P. Mishra, “Ddos detection and defense: client termination approach,” *Proceedings of the CUBE International Information Technology Conference*, pp. 749–752, 2012.

[16] A. M. Freed, “Exclusive Video of XerXeS DoS Attack - available at `http://www.infosecisland.com/blogview/2990-Exclusive-Video-of-XerXeS-DoS-Attack.html`,” Date Accessed in December 13, 2013.

[17] e. a. M. Aiello, "Slowreq: A weapon for cyberwarfare operations. characteristics, limits, performance, remediations," *International Joint Conference SOCO13-CISIS13-ICEUTE13*, pp. 537–546, 2014.

[18] G. Huston and E. Aben, "The Curious Case of the Crooked TCP Handshake - available at `https://labs.ripe.net/Members/gih/the-curious-case-of-the-crooked-tcp-handshake`," Date Accessed in December 2, 2013.

[19] e. a. Y. Liang, "The effect of real-valued negative selection algorithm on web server aging detection," *Journal of Software*, vol. 7, no. 4, pp. 849–855, 2012.