INTERDISCIPLINARY CENTER
HERZLIA

ABSTRACT

DISTRIBUTED DENIAL OF SERVICE ANALYSIS

By Henrique Wajcberg & Iftach Amit

This paper will present the work done to analyze and dissect a common Denial-of -Service attack, in order to broaden the understanding of the attack and to help find a defense mechanism for that attack.

Most work has focused on the "SYN flood" attack in distributed configuration, and some other attacks have been analyzed for the sake of comparison.

July 2001

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

SCOPE, GOALS AND METHODOLOGY

Denial of service attacks have been targeting computer systems for a long time now. Dealing with an individual flooder was relatively easy, but then the attackers have evolved the attack and the new "Distributed" denial of service attack (aka DDoS) has been devised.

Goals and Scope

Nowadays DDoS poses one of the main unresolved security threats in computer security.

This work performs a thorough analysis of the DDoS mechanism, focusing on one of the more common attacks called the "SYN flood". The main goal of this work is to establish enough information and know-how about these attacks in order to provide a foundation for later work which will try to resolve a "cure" for these kinds of attacks.

Methodology

The methodology consisted of several phases for understanding the phenomena of DDoS:

1. Analyzing sniffer data of attacks on a host.

2. Examining the common tools that are used to generate such attacks.

3. Examine the Operating system source code that deals with common DDoS attacks.

4. Perform performance analysis of an attacked host under several situations and load subjects.

The output of these steps was then used to generate a profile of the attack and perform quantitative analysis on the raw data. The results of these are presented in this work.

Tools of the trade

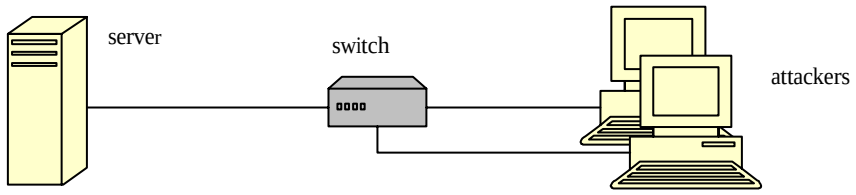
The tools used for simulating the attacks were the common denial of service tools used by crackers to attack websites on the Internet.

1. “Tribal Flood Network 2000” (or simply TFN2K)
2. ”Trinoo”
3. “Stacheldraht V4”
4. “synk4”

In order to examine the network layer, we have used simple network packet sniffer utilities such as the native *tcpdump* in UNIX, as well as “Ethereal” for Linux in order to parse the sniff log.

The web traffic used to test the web server capabilities, and simulate regular user traffic, was “Apache Benchmark” (or *ab*) which is a part of the Apache web server distribution. This tool enabled us to generate any number of concurrent requests over a specified amount of time.

Our network configuration was very simple – single target hosts, connected to a network switch, and with other one or two attacking hosts connected to the same switch. Sniffers were installed on all hosts.



Chapter 2

ANALYSIS OF THE SYN FLOOD ATTACK

The SYN flood attack consists of sending SYN requests (i.e. TCP packets with the SYN flag up) that form the first part of the "Three way handshake" in TCP connection establishment. These packets are sent with the source IP spoofed (i.e. randomly generated - see fig. 1) so when the attacked host tries to continue the three way handshake, the destination is either unreachable or does not expect the "SYN-ACK" packet that is sent in response to the "SYN" request. Either way, the connection establishment will not succeed and the resource that the attacked host allocated for that connection will stay "busy". This causes the situation of resource shortage on the attacked host and no clients can connect to the host.

This kind of attack is hard to trace because of the spoofed source IP's looks just like a regular client access, and trying to generalize the attack and identify it in the overall communications is impossible due to the fact that the source IP distribute equally over the whole IP address range.

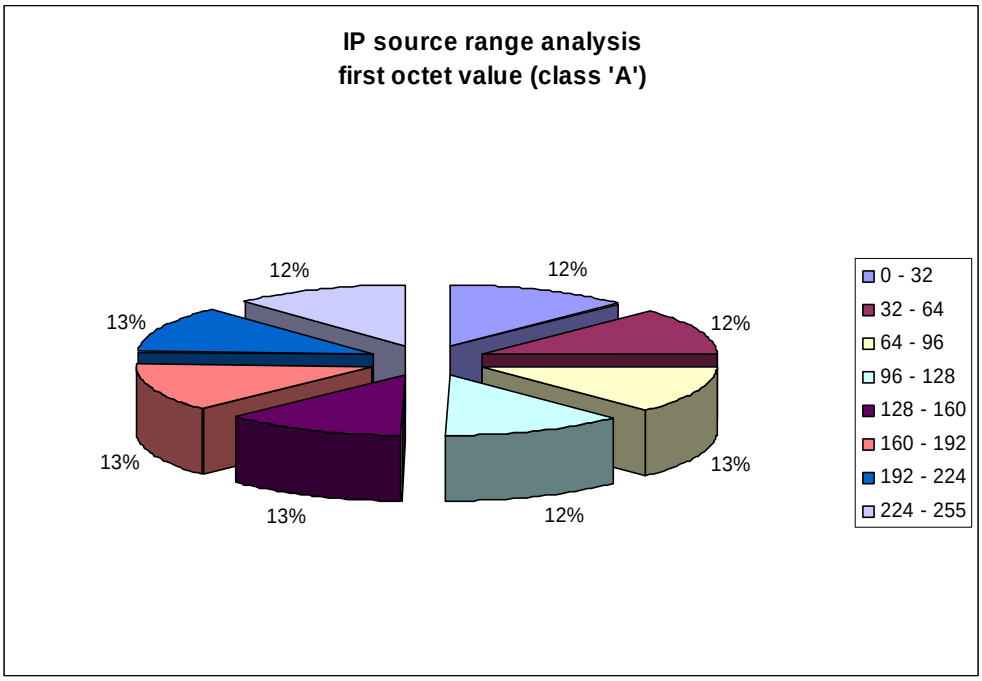


Figure 1 - IP source range distribution

This figure shows the distribution of the source IP addresses by analyzing the first octet of address (the class 'A' identifier in an IP address). It clearly shows that the distribution was indeed random for the address ranges are evenly distributed over all the octet range (255 values).

Server side

The server under attack allocates resources to every SYN request in it TCP stack implementation. These resources are limited and bounded by a finite number. This number is OS dependent (and usually also depends on hardware settings) and in our case was 1024 places in every socket's backlog. Therefore, altering the stack implementation by changing this upper bound is not efficient, for the amount of traffic that forms the attack will flood any amount of bounded resource in a matter of seconds (see fig. 2).

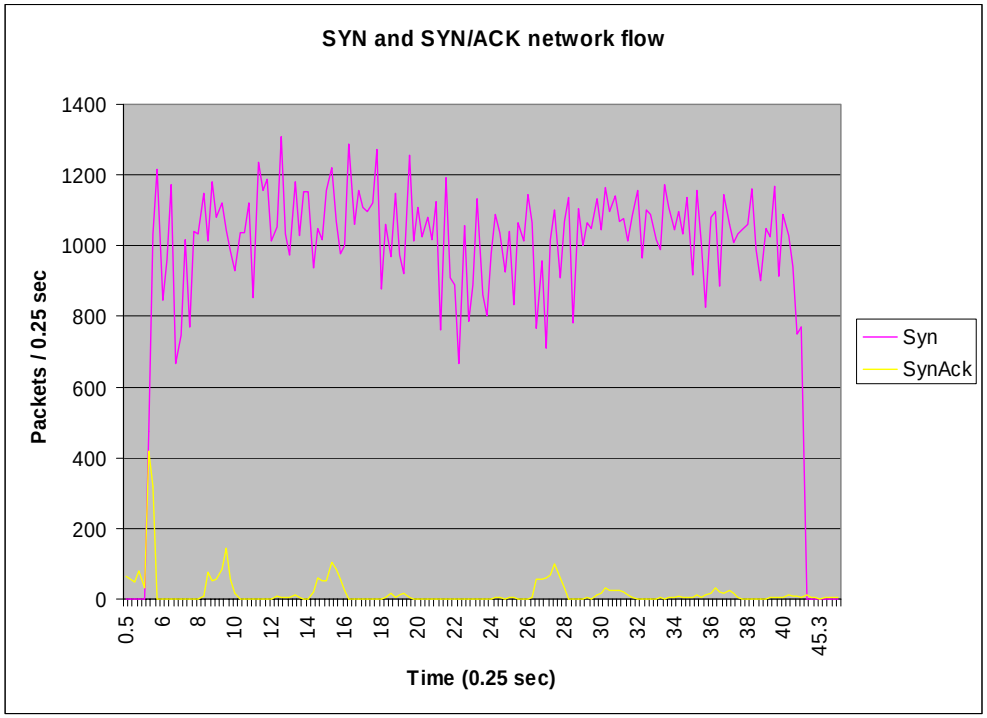


Figure 2 - SYN and SYN/ACK analysis under a SYN flood.

This diagram shows the behavior of the attacked host, and the timeouts for retransmission and socket destruction. The server cannot handle the incoming flood of SYN requests, and as soon as some sockets are freed, a new SYN request captures it.

Regarding the SYN/ACK behavior observed in fig. 2, we have superimposed three sessions of similar attacks in order to verify the observed behavior (fig. 3). We can clearly see the first burst of packets which sums up to 1016 which is close enough to the 1024 limit mentioned before (some addresses were probably unreachable), then after 4 seconds of timeout (3 seconds in kernel), another burst of retransmits; 6 seconds later another burst which is the last retransmission (allowed after maximum of 5 seconds for last retransmit in kernel). Following that is a pause of 12 seconds after which a new burst of SYN/ACK arrives. The last delay is the time that the socket is destroyed after, if no ACK has been received (configured for 10 seconds in kernel).

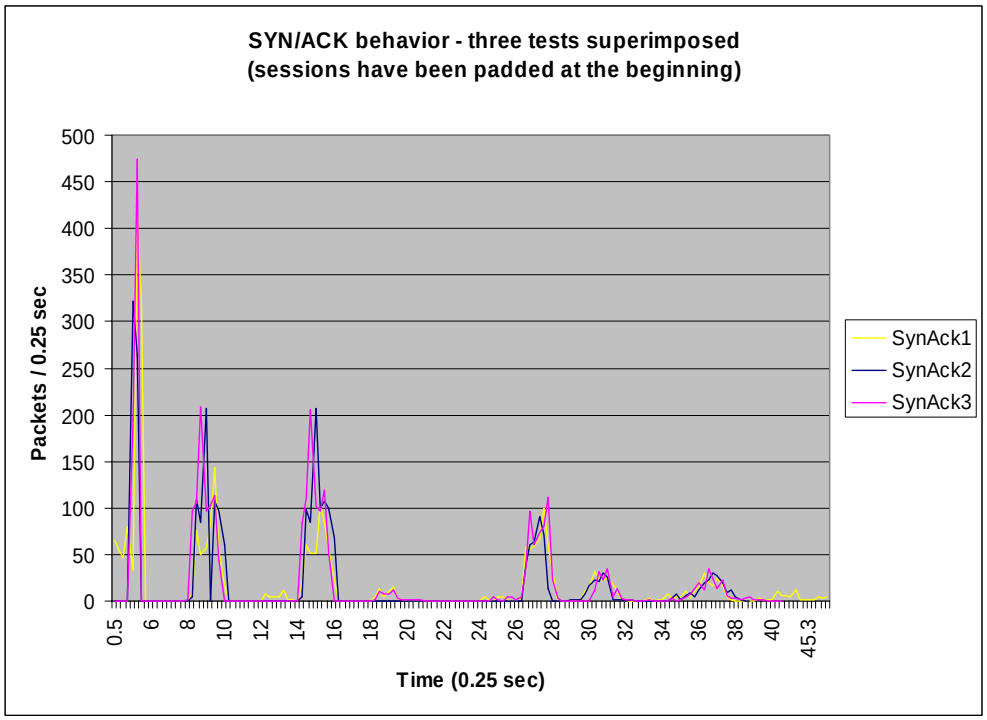


Figure 3 - Three SYN/ACK session recordings superimposed

We have superimposed the SYN/ACK behavior of three sessions by using the first peak as a correlation point, and padding the beginnings with 0's (zeroes). This clearly shows that the behavior observed is consistent and is explained in the paragraph above.

Chapter 3

HANDLING SYN FLOOD USING LINUX "SYNCOOKIES"

Trying to develop a solution that will handle flood situations as described before, resulted in a mechanism called "syncookies". This mechanism implements the following procedure instead of the regular behavior of a TCP three way handshake:

- When a SYN request is received, a hash is calculated, using the main packet identifiers: source address and port, destination address and port, sequence number, the MSS, and a couple of secret numbers that the server knows about. Then, the TCP stack resources for that packet are immediately released.

- After the packet information has been recorder (i.e. A "cookie" was generated), a SYN-ACK response is initiated using a new socket structure that does not conflict with the incoming sockets resource. This connection is initiated using the "cookie" previously recorded for the packet. This means that a brand new connection is established from the server point of view, and the connection information is regenerated from the cookie.

- When the server receives an ACK, it checks that the secret function works for a recent value of t (where t is a 32-bit time counter that increases every 64 seconds), and then rebuilds the SYN queue entry from the encoded MSS.

As the SYN flood was described, it is simply a series of SYN packets from forged IP addresses. The IP addresses are chosen randomly and don't provide any hint of where the attacker is. The SYN flood keeps the server's SYN queue full. Normally this would force the server to drop connections. A server that uses SYN cookies, however, will continue operating normally. The biggest effect of the SYN flood is to disable large windows.

As part of this research several simulations were done in effort to understand the attacks implication, such as the resistance of the server to the SYN attack scenario and the network response to different load. Some of the test included concurrent naïve users hoping that the server would allocate them some minimum essential connections and resources. Analyzing the performance degradation that this extra handling adds to the server leads to the following conclusions:

1. The server is working a bit harder. Analyzing the graph (fig. 4 - System CPU time) the first five seconds compare to server not running the syncookie mechanism, it seems the server reach a peek of 50 percent of usage and lasting till the end of the load with an average of 40 to 50 percent.

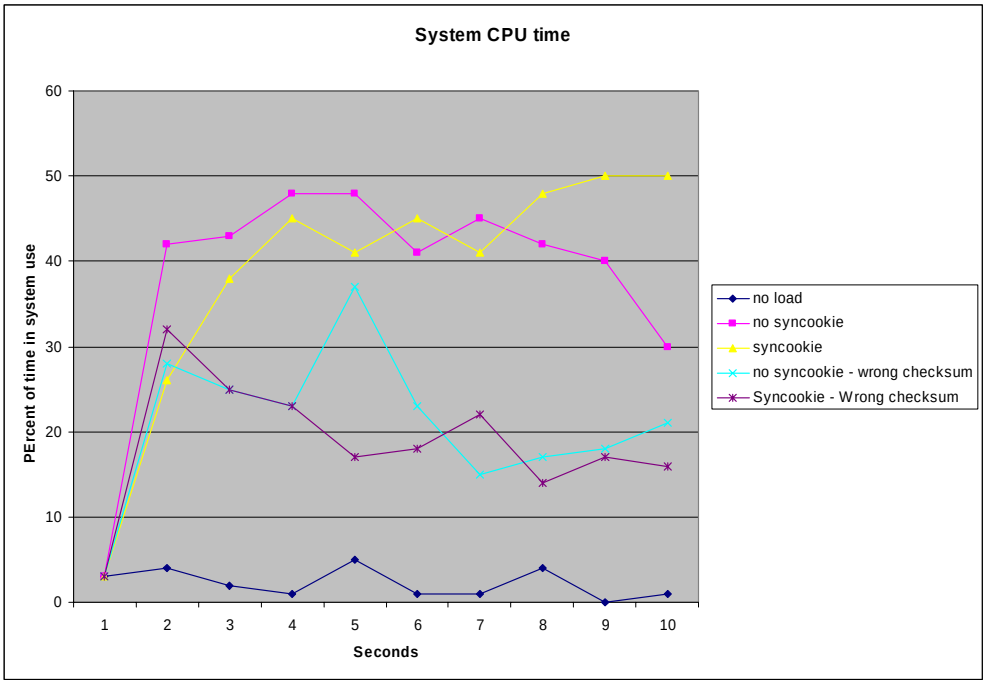


Figure 4 - System space percent of CPU time

This chart shows the use of CPU time in percent, by the system space process in our test host. The host is completely idle and the measurements show the computational load imposed by the different attack types.

2. The server networking behavior shows that the mechanism is extremely useful in eliminating the DoS situation (fig. 5). We can clearly see the impact of the syncookie mechanism as the network flow includes almost the same amount of SYN and SYN/ACK packets as opposed to the server that did not have syncookies, where the network is full of SYN packets, and once in a while (timeouts) a small bulk of SYN/ACKs are sent from the server.

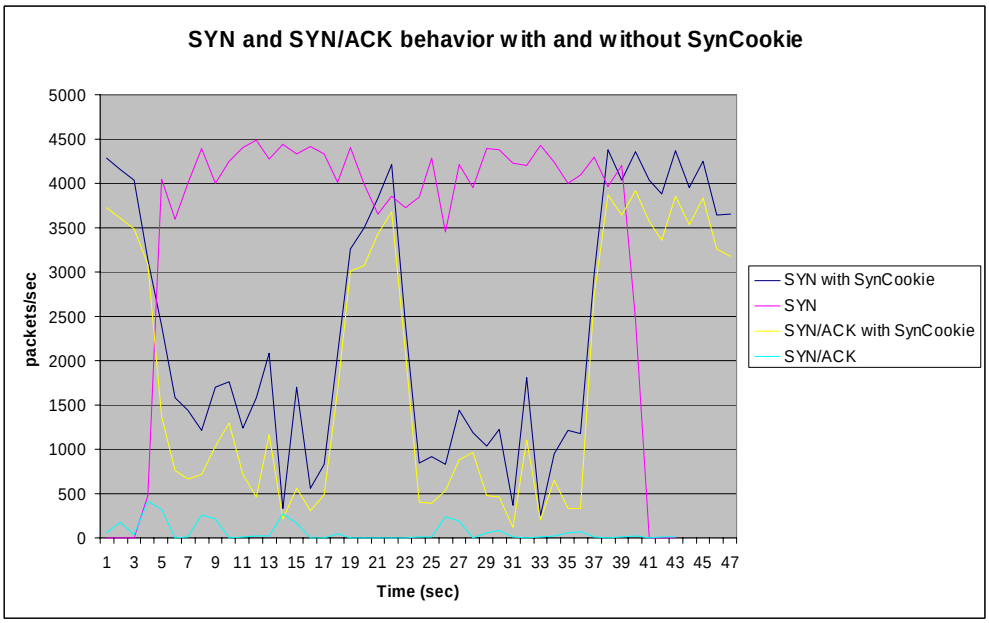


Figure 5 - SYN and SYN/ACK behavior with/out syncookies

This chart visualizes the network traffic under different host configurations – with normal operation, and with the syncookie mechanism enabled.

A question arises regarding the throughput of the syncookie-enabled scenario, for in regular traffic, the medium seems to be able to hold about 4k-5k packets/second, where some situations in the syncookie scenario show very poor performance (about 2k packets/seconds), and other peaks show almost 8k packets/second. This behavior is clearly the result of collisions over the Ethernet medium. When the throughput is extremely high, many collisions occur, and both sides slow down abruptly, then there is a long phase of low throughput where the collision rate is high, which makes both sides try to send more packets per second, which causes another high throughput peak and collisions, and so on…

3. Analyzing the time-sharing between the user space and system space processes (fig. 6), we can clearly see that a loaded web server, requires more system space processing (i.e. handling the network layer) than user space processing (i.e. serving web pages). This information, along with the analysis of the SYN attack implications on the attacked server (fig. 4), helps us understand why the attack is so effective. The SYN attack results in high system space usage, and as the number of current user is higher, the system load is higher.

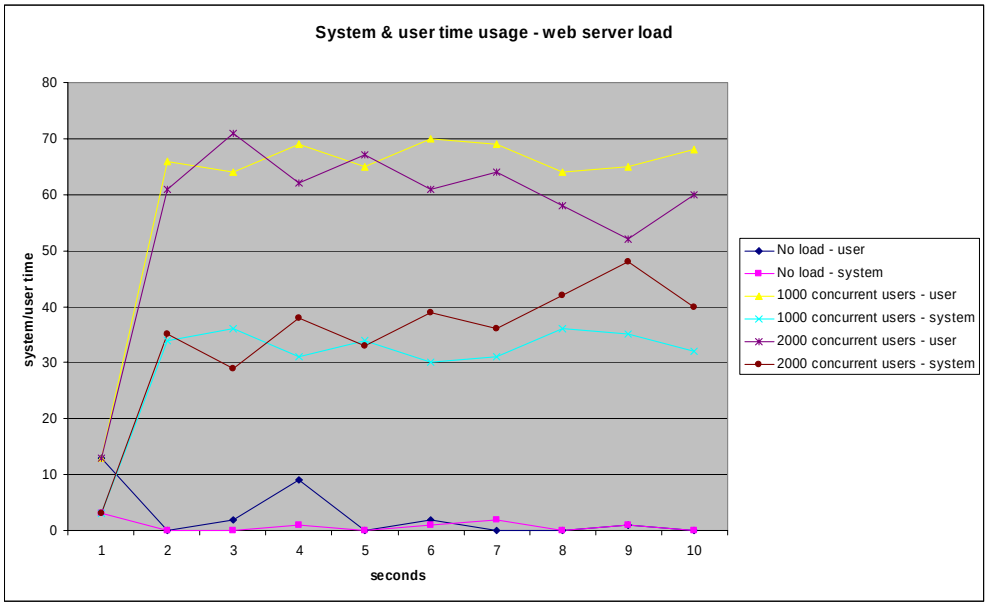


Figure 6 – System space and user space CPU time sharing under web load

Here, the web server performance was tested in order to get an idea of the impact that a DoS attack will have on legitimate users. As the load rises, more system space CPU time is required, because it seems that handling the low level networking load requires more attention than handling the web serving itself.
The maximum number of concurrent users achieved on the test server was about 1800.

4. Trying to use invalid TCP/IP packets (with wrong checksums) does not impose the network unavailability, but still puts the server under some pressure, as can be seen from the behavior under an attack using 0 (zero) checksum packets (fig. 4).

Chapter 4

COMPARISON TO OTHER ATTACKS - PERFORMANCE-WISE

We have chosen to focus on the SYN flood attack, for some reasons. One of them (amongst with the rest mentioned before) is the fact that the SYN attack is considered to be the most effective in spending system resources. We have performed some measurements to quantify these assumptions and reached the following conclusions (see fig. 7):

1. The SYN attack has the fastest impact on the attacked system.

2. Only ICMP attacks can begin to measure up to the impact that SYN floods have on the system.

3. UDP has some impact, although it was expected to be much more effective.

4. Most modern operating system, are not affected anymore by mangling the IP packets themselves in order to invalidate the IP protocol or cause the system to work harder on strange packets.

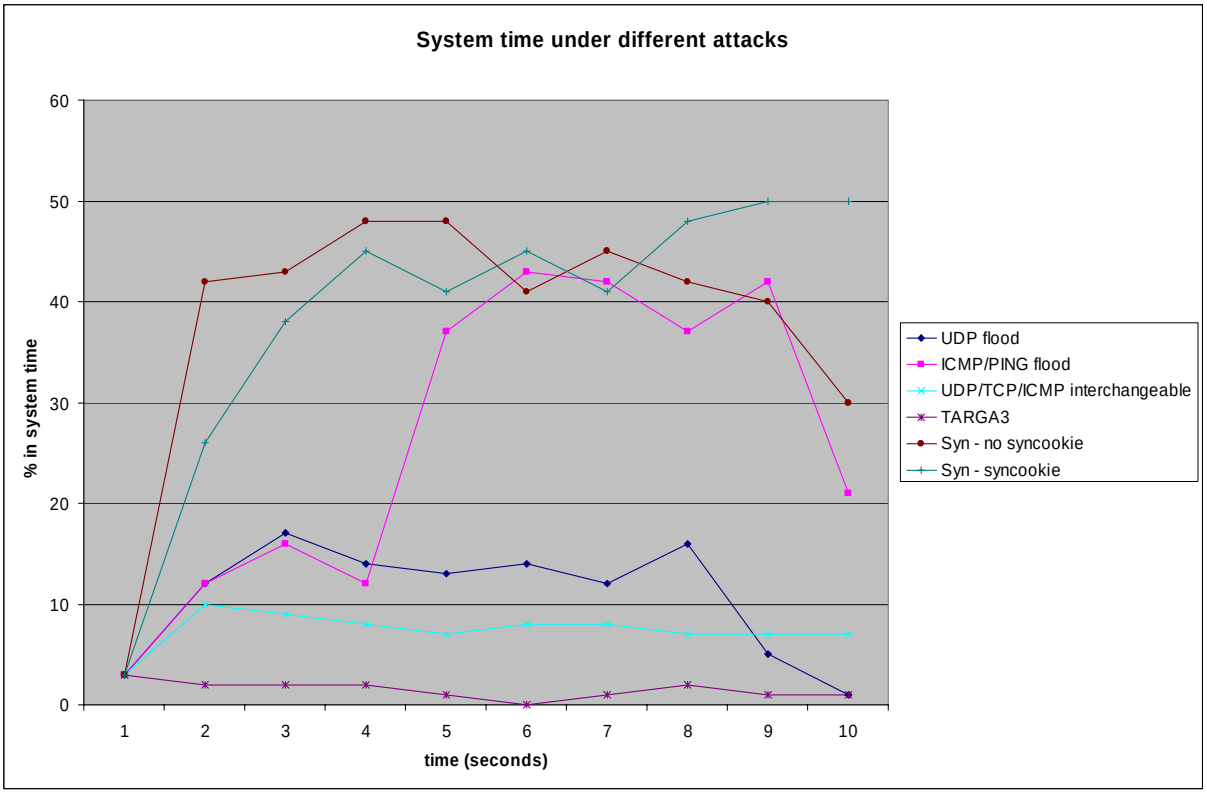


Figure 7 - System space percent of CPU time under different attacks

Besides the SYN attack which has been explained before, hereby is an explanation of the other attack types used in our research:

- UDP flood – sends spoofed UDP packets to the host, similar to the SYN flood, but the emphasis is on network flooding rather than resource exhaustion.

- ICMP/PING flood – send spoofed ICMP-ECHO-REQUEST packets, which will cause the attacked server to try and respond to them.

- TARGA3 attack (IP stack penetration) – sends mangled packets with: invalid fragmentations, protocol, packet size, header value, offset, TCP options, TCP segments and routing flags. These are picked randomly. The target is to "penetrate" the IP stack – i.e. cause the attacked host to halt because of improper stack implementation.

To be able to understand the real implications of the network flow, we must bear in mind that each attack has a different packet size, thus the load (packets/second) is different. The packet sizes are: 60 bytes for the UDP flood, 60 bytes for the SYN flood, and 106 bytes for the ICMP/PING flood. TARGA3 has a varying packet size because of its nature.

Chapter 5

SUMMARY

This report focused on the "SYN flood" attack; although the actual work has been conducted using other attacks as well (see fig. 7). We chose to focus on the SYN attack, because it is one of the simpler, yet still irresolvable DoS attacks that can be found on the Internet. Other attacks usually use some vulnerability in the TCP stack implementation of the attacked host (like SMURF or TARGA3 attacks), and most modern stack implementations do not have these vulnerabilities that cause the race conditions that evolved during these attacks.

We also found that the SYN attack targets the main asset of the attacked host – the number of available receiving sockets, which when in starvation, the host is unreachable for regular clients (the exact meaning of denial of service…).

Conclusion

We have analyzed the main impact of the DDoS attack on the attacked server, and clearly the first damage is the actual denial of network services (fig. 5), while in the background the system itself is overloaded with work while trying to handle the incoming flow of SYN requests (fig. 4). While measures like syncookie propose a solution to the first problem, the second one is still unresolved and even gets a bit worse (fig. 4).

Possible solutions

There exists no TCP implementation nowadays that can handle the SYN attack in a resource friendly way, except for dedicated hardware solutions such as routers. We feel that the real solution will be in such devices along with some use of an altered routing algorithm so that these attacks will be rendered ineffective and will be dealt with not at the private network's border, but along the way at least 2-3 hops away using distributed algorithms and data mangling. In that way, the attack will be identified along its way towards the target, and the remedy will involve throttling the traffic in strategic locations before the target so that the target will not have to deal with the full flood.

BIBLIOGRAPHY

The Linux kernel sources (/usr/src/linux/net/ipv4) of a 2.4.6 kernel.

Linux kernel mailing list archive (http://www.tux.org).

Appendix A

SYNCOOKIES IMPLEMENTATION FOR THE LINUX KERNEL

SYN cookies provide protection against SYN flood attacks. With this option turned on the TCP/IP stack will use a cryptographic challenge protocol known as SYN cookies to enable legitimate users to continue to connect, even when your machine is under attack. The following code snippets are part of the syncookie implementation written by Andi Kleen (syncookie.c v 1.13). For a better understanding of the results described in the body of this paper, we found it necessary to clarify the basic functionality of this unique feature, which turn to be a vital mechanism against DDoS.

We start with the `cookie_v4_init_sequence` method:

```
/ * Generate a syncookie.  mssp points to the mss, which is returned
  * rounded down to the value encoded in the cookie.
  */
__u32 cookie_v4_init_sequence(struct sock *sk, struct sk_buff *skb,
                              __u16 *mssp)
{
    int mssind;
    const __u16 mss = *mssp;

    tcp_lastsynq_overflow = jiffies;
    /* XXX sort msstab[] by probability?  Binary search? */
    for (mssind = 0; mss > msstab[mssind+1]; mssind++)
    ;
    *mssp = msstab[mssind]+1;

    NET_INC_STATS_BH(SyncookiesSent);

    return secure_tcp_syn_cookie(skb->nh.iph->saddr,
                                 skb->nh.iph->daddr,
                                 skb->h.th->source, skb->h.th->dest,
                                 ntohl(skb->h.th->seq),
                                 jiffies / (HZ*60), mssind);
}
```

This method calculates a secure_tcp_syn_cookie (syncookie) at the very beginning stage of the connection. It Computes the secure sequence number where the output is: *(from random.c)*

HASH (sec1, saddr, sport, daddr, dport, sec1) + sseq + (count * 224) + (HASH (sec2, saddr, sport, daddr, dport, count, sec2) % 224)

Where sseq is their sequence number and count increases every minute by 1.

As an extra hack, a small "data" value is added that encodes the MSS into the second hash value. (saddr – source address, sport – source port, daddr – destination address, dport – destination port, sec1/2 – secret values generated randomly)

The resulting “cookie” is an unsigned 32bit numeric value, which is used as the sequence number in the SYN/ACK packet. This is how the server differentiates between “real” and flood SYN requests (flood will not respond to the SYN/ACK, while real requests will). This also prevents an attacker from sending spoofed ACK packets, because he cannot know what is the sequence number the server used (the secure cookie).

So far we have discussed the question of how to construct the packet that elicits response from the remote host (by sending a SYNACK packet that has a sequence number).

The remote TCP (the client) must respond with a packet containing a valid sequence number, which is equal to hash number stored at the server side (the cookie).

The first lines of the `cookie_v4_check` checks basic information:

Retrieving the cookie value:

```
__u32 cookie = ntohl(skb->h.th->ack_seq)-1;
```

And mss size:

```
mss = cookie_check(skb, cookie);
if (mss == 0) {
    NET_INC_STATS_BH(SyncookiesFailed);
    return sk;
}
```

The `cookie_check` actually perform the call to the cryptographic validation function (`check_tcp_syn_cookie` in random.c) as in: *(from syncookie.c)*

```
mssind = check_tcp_syn_cookie(cookie, skb->nh.iph->saddr,
                              skb->nh.iph->daddr, skb->h.th->source,
                              skb->h.th->dest, seq, jiffies/(HZ*60),
                              COUNTER_TRIES);
```

The return value, along with the verification of the correct MSS validates the cookie, and the creation of a new socket for the finalization of the three way handshake is taking place at the end of the validation method (`cookie_v4_check`).