2005

Denial of Service Attack Techniques: Analysis, Implementation and Comparison

Khaled M. Elleithy
University of Bridgeport

Drazen Blagovic
Sacred Heart University

Wang K. Cheng
Sacred Heart University

Paul Sideleau
Sacred Heart University

Denial of Service Attack Techniques: Analysis, Implementation and Comparison

Khaled M. Elleithy
Computer Science Department, University of Bridgeport
Bridgeport, CT 06604, USA

Drazen Blagovic, Wang Cheng, and Paul Sideleau
Computer Science Department, Sacred Heart University
Fairfield, CT 06825, USA

ABSTARCT

A denial of service attack (DOS) is any type of attack on a networking structure to disable a server from servicing its clients. Attacks range from sending millions of requests to a server in an attempt to slow it down, flooding a server with large packets of invalid data, to sending requests with an invalid or spoofed IP address. In this paper we show the implementation and analysis of three main types of attack: Ping of Death, TCP SYN Flood, and Distributed DOS. The Ping of Death attack will be simulated against a Microsoft Windows 95 computer. The TCP SYN Flood attack will be simulated against a Microsoft Windows 2000 IIS FTP Server. Distributed DOS will be demonstrated by simulating a distribution zombie program that will carry the Ping of Death attack. This paper will demonstrate the potential damage from DOS attacks and analyze the ramifications of the damage.

Keywords: Communications systems security, Denial of Service Attack (DOS), TCP SYN Flood, Ping of Death.

I. Introduction

Denial of services attacks (DOS) is a constant danger to web sites. DOS has received increased attention as it can lead to a severe lost of revenue if a site is taken offline for a substantial amount of time; see [1-4]. There are many types of denial of service attacks but two of the most common are Ping of Death and TCP SYN Flood. We have chosen to implement these two techniques and add Distributed DOS (DDOS) as well.

In a Ping of Death attack, a host sends hundreds of ping requests (ICMP Echo Requests) with a large or illegal packet size to another host in attempt to knock it offline or to keep it so busy responding with ICMP Echo replies that it cannot service its clients.

A TCP SYN Flood attack takes advantage of the standard TCP three-way handshake by sending a request for connection with an invalid return address.

In this paper we demonstrate DDOS by creating a worm like program that installs programs on remote machines to attack a particular server. These attackers listen in the background for a message from a master program that will tell these attackers to launch a DOS attack against a machine.
DDOS attacks are difficult to stop because they can be coming from anywhere in the world. We will implement a DDOS attack by launching the Ping of Death implementation against a victim computer from several other workstations.

II. Distributed Denial of Service with Ping of Death Payload Implementation

To implement DDOS, a worm like program is created to simulate self-propagation onto many hosts on a network. However, creating an actual worm is beyond the scope of this paper, therefore, we used a small Java program to simulate such a worm. Though it carries the payload and waits to receive orders from a master program, the worm does not self propagate. We simply placed the application on each host machine manually for simulation purposes.

The worm-like zombie program will launch a Ping of Death attack from multiple hosts coordinated by a master program. The applications handle all communication between each other. When the master program orders the attack, a message is sent to all the zombies that makes them release their Ping of Death payload against a victim host that is specified by the master program.

The Java implementation has been built using TCP sockets and serializable Java objects. Serializable Java objects can be transferred to remote servers and then executed with all of its information intact. The serializable Java objects have all the instructions needed to launch a particular type of attack. When a user wishes to initiate an attack, he or she starts up the master program and specifies which server to attack. The master program then looks up the IP addresses of all known zombie programs and what ports they are listening on by accessing a configuration file. It then constructs a Java serializable object based on the DOS attack type specified, and sends it to every zombie listed in its configuration file over a TCP socket. The zombie program recognizes that it has received a message and reads from a TCP socket the serializable java object. It then deserializes it and executes it, which in turn will launch the DOS attack. Figures 1,2 show the scenario of this implementation.

In figure 3, the DOSZombie class acts as a server that services DOS attacks. It is created by specifying what port it should listen on. It then creates a TCP socket and waits in the background until a communication message is received. It is multithreaded; therefore, it can receive multiple connections at the same time.

The DOSAttackLauncher class acts as the client and can communicate with a DOSZombie. It is created by specifying the zombie's IP address and the port that it is listening on. The DOSAttackLauncher then creates a TCP connection to the

zombie specified. Its method, "launchAttack", takes a DOSAttack object. It will then send this object to the DOSZombie and the DOSZobmie will then invoke the DOSAttack's attack method. The DosZombie also sends a return message to the DOSAttackLauncher notifying if it successfully began the attack or if an error occurred.

The DOSAttackManager is used by the master program that launches the attack. It finds all the zombies that are running, constructs a DOSAttackLauncher for each zombie, and then instructs all DOSAttackLaunchers to send the type of DOSAttack to the zombie.

The DOSAttacker class is an abstract class to support polymorphic behavior and implements the Java Serializable interface so that it can be sent through a TCP socket and executed on a remote server.

The PingDOSAttacker and SYNFloodAttacker have the implementations for the "Ping of Death" attacker and the TCP SYN flood attack, respectively. The PingDOSAttacker makes an external call to the C# Ping of Death implementation program when their attack methods are called.

There are two Ping of Death implementations. The first and most simple implementation is simply by calling a ping application that comes with any modern network operating system. The packet sizes are modified to be larger then the default 32 bytes. The strength of this attack is not in the ping application itself by rather in the fact that when used in a DDOS scenario, the victim computer is simply overwhelmed by the large quantity of ICMP Echo Request packets.

The second implementation uses a C# program and RAW Sockets to increase the amount and speed of the ECHO Request packets in addition to the size. RAW Sockets are a form of TCP sockets that allow the programmer to build each packet from scratch. The application therefore, must define all the parameters in the header as well as allocate all necessary buffers to contain the packet. All checksums and validations have to be done by the application rather then handled by the operating system. The advantage is speed: by removing checks and safeguards that the standard TCP socket has in place and allocating exact buffers, resource use is significantly reduced as a whole when considered in volume. This also allows the application to do things like ignore ECHO Reply packets and concentrate solely on sending ECHO Request packets. The implementation is extremely straightforward being small and concise, perfect for use in a covert DOS attack.

The second implementation uses a C# program and RAW Sockets to increase the amount and speed of the ECHO Request packets in addition to the size. RAW Sockets are a form of TCP sockets that allow the programmer to build each packet from scratch. The application therefore, must define all the parameters in the header as well as allocate all necessary buffers to contain the packet. All checksums and validations have to be done by the application rather then handled by the operating system. The advantage is speed: by removing checks and safeguards that the standard TCP socket has in place and allocating exact buffers, resource use is significantly reduced as a whole when considered in volume. This also allows the application to do things like ignore ECHO Reply packets and concentrate solely on sending ECHO Request packets. The implementation is extremely straightforward being small and concise, perfect for use in a covert DOS attack.

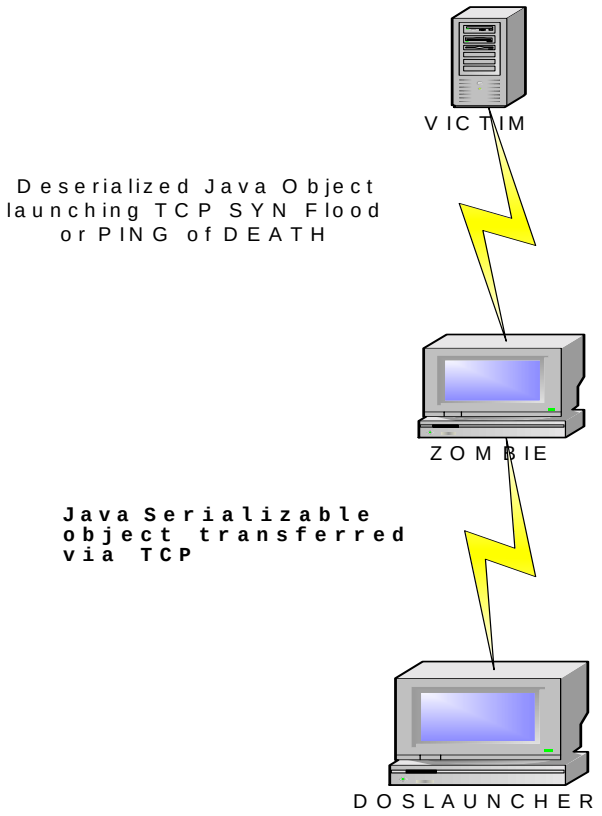

Fig. 1. Communication Framework

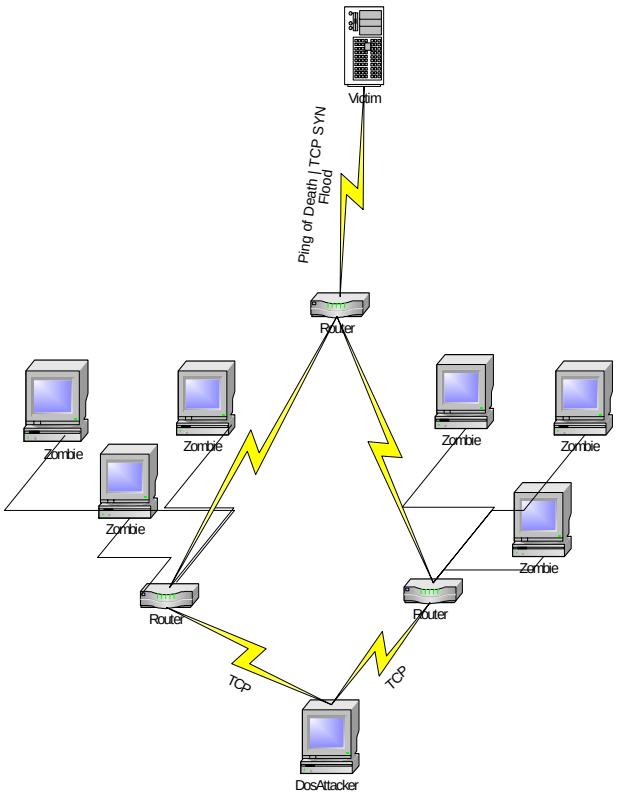

Fig. 2. Overview of communication process.

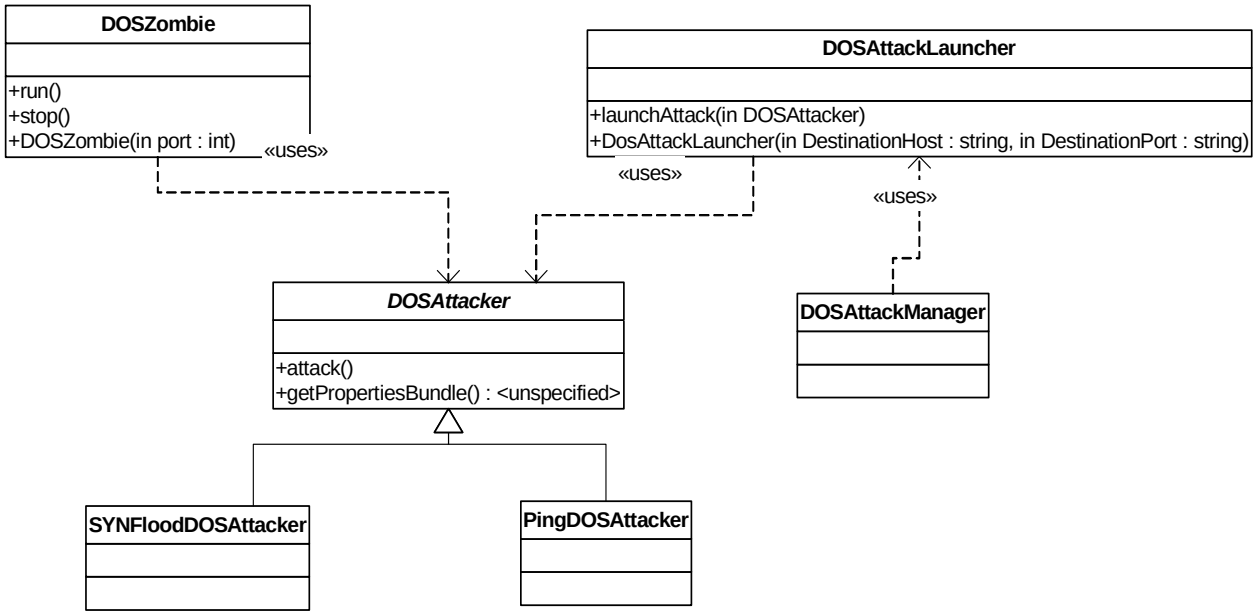

Figure 3. Class Diagram of Java DDOS

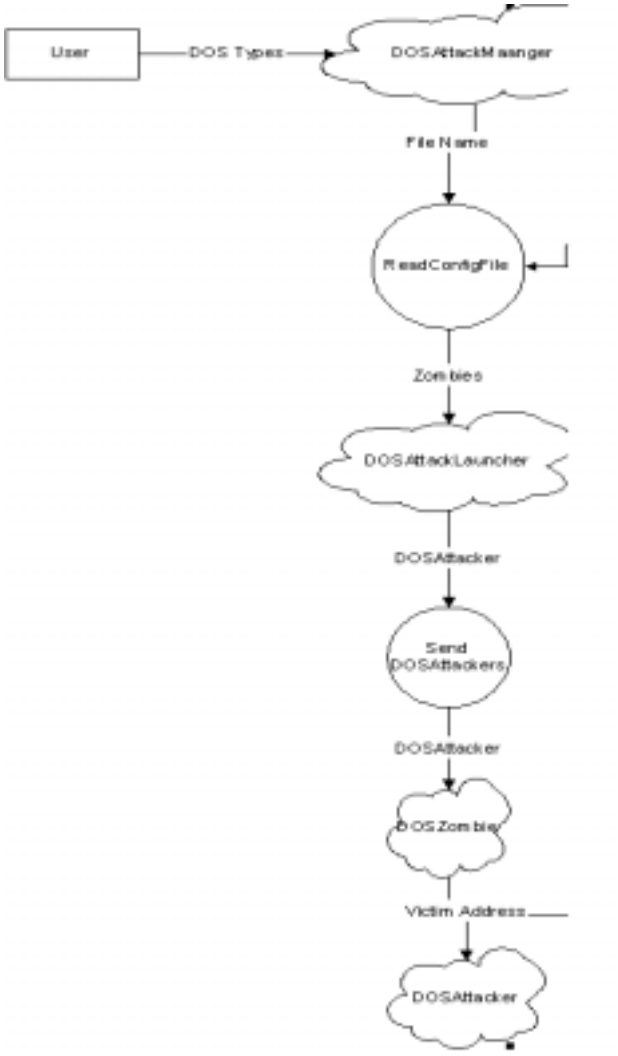

Figure 4. Data Flow Diagram

III. TCP SYN Flood Implementation

When hosts need to establish communications via the TCP transport protocol, they must do a session initiation, which consists of a three-way handshake:

1. The source host initiated the communication by sending a TCP packet to the destination host the SYN flag (SYNchronize sequence numbers) set to 1. In this packet reside the source IP address and port number as well as the destination IP address and port numbers (in addition to several other fields which are inconsequential for this discussion).

2. The destination host responds by sending a TCP packet to the source host with the flags SYN and ACK (ACKnowledge) set to 1. The response is sent to the source IP address and port of the initial packet in step 1.

3. The source host sends the destination host another TCP packet with the ACK flag set to 1. This completes the 3-way handshake and normal data communication can start.

In a TCP SYN Flood attack, the source (attacker) host simply fails to complete step 3 leaving the destination (victim) host with an unfinished communication session. When the victim's TCP socket receives the message in step one, it allocates buffers, increments counters, initiates timers, and increases communication stacks in preparation for the communication that is to follow. In addition, processor time is spent building the reply packet (step 2) and sending it back. The attacker can overwhelm the victim's computer resources by sending a "flood" of packets with the SYN flag set to 1 (step 1) and never bothering returning any response (step 3).

The TCP SYN Flood attack implemented is the Neptune algorithm and implementation. In this algorithm, not only is step 3 of the TCP handshake ignored, the source address in the SYN packet of step 1 is set to an unreachable destination (for

example a non-routable IP address). IP spoofing is used in this implementation therefore; it is virtually impossible to track the origin of the packet since the return address is fake. The victim's computer now expends time to try to deliver a packet to an inexistent destination.

The Neptune implementation also allows the attacker to specify a specific service to deny. In a classical TCP SYN Flood attack, the attacker generally tries to prevent the victim's computer from servicing any legitimate requests. The Neptune implementation however, allows the attacker to choose a specific TCP service port to overwhelm. In other words, the attacker can choose to bring down only a web server for example (port 80).

A simulation for an attack on a FTP server running Windows 2000 IIS FTP has been tested. Figure 5 shows a small TCP SYN Flood attack against an FTP server (IP address 148.166.161.115). Notice that the source IP address is spoofed (part of the non-routable 10.x.x.x class B range). In this particular attack, only three SYN flood packets were sent (Nos. 1, 3, 5) against an FTP server (port 21 destination). For each of the packets, the server replies with and ACK-SYN packet which in turn ends up nowhere (Nos. 2, 4, 6). The server then retries to send replies a further two times before giving up (Nos. 7-12).

When looking closer at the actual packets we can see the spoofed packet clearly with the SYN flag set to 1 (Figure 6) and the spoofed source IP address of 10.10.1.1. Similarly, the return packet (Figure 7) is destined for nowhere and has the ACK and SYN flags set to 1. The application also takes care of using different source port numbers and sequence numbers. This prevents the victim's computer from assuming that packets all come from the same client in the same host. By changing the return port and sequence numbers, a single computer can force another host to allocate several connection resources.

IV. Analysis

Figure 8 shows an excerpt from the source code. At this point, the TCP header is being built. Notice the random source port, sequence number, and SYN flag set to one in the bolded sections. In this particular example three packets where sent, however, when a true flood of invalid packets are sent, it will overwhelm the computer and the more specifically the targeted service. The packets are very small (since they contain no data) so that even a slow computer with a slow dial-up connection can overwhelm a server in a matter of seconds. As the server spends time and resources trying to handle these fake connections, it starts to drop packets as it becomes overwhelmed, in doing so, it starts to also drop legitimate packets from legitimate users. If such an attack were to be delivered via DDOS, the results could be devastating for a victim's computer. One simple computer can already overwhelm most TCP stacks; a distributed attack would most likely crash those stacks and the operating systems with them.

The DDOS program is implemented in Java and can be used in virtually any operating environment that supports a Java JIT compiler making it cross-platform. Since every operating system with a TCP socket support has a ping application, it would be easy enough to launch a Ping of Death attack from a wide range of hosts. The only part missing is the self-propagation piece of the zomebie worm. The architecture is also open to allow it to deploy almost any attack via serializable objects. This is to say, it could deliver any attack as its payload, even our TCP SYN Flood application that is written to take command line parameters as well. The TCP SYN Flood application however, has the draw back of only working in a Linux environment. It can be ported to a UNIX environment and with the advent of RAW sockets in Microsoft Windows 2000/XP, it could also be ported over to the Windows world.

V. Conclusion

All the implementations done in these simulations consist of very simple and light loaded attacks, which can cause severe amounts of damage. DOS attacks can be stealthy covert and easily delivered. The Neptune implementation for example, is only 10Kbytes in size and can cause devastation to a service. When combined with the power of a DDOS attack, Denial of Service is a truly powerful attack. Although our implementations are not sophisticated, they serve as examples of what such programs can do and the damage they can cause.

VI. References

[1] Christoph L. Schuba, et. al., "Analysis of a Denial of Service Attack on TCP," *1997 IEEE Symposium on Security and Privacy* , May 1997, pp. 208.

[2] Frank Kargl, Joern Maier, Michael Weber, "Protecting web servers from distributed denial of service attacks," *Proceedings of the tenth international conference on World Wide Web*, April 2001, pp. 514.

[3] Errin Fulp et. al. "Preventing Denial of Service Attacks on Quality of Service," *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*Volume II-Volume 2 , June 2001, pp. 1155.

[4] Gresty, Q. Shi, M. Merabti, "Requirements for a General Framework for Response to Distributed Denial-of-Service," *17th Annual Computer Security Applications Conference (ACSAC'01)*, December 2001, pp. 422.

VII. Biographies

Khaled M. Elleithy (M'1988) received the B.Sc. degree in computer science and automatic control from Alexandria University in 1983, the M.Sc. Degree in computer networks from the same university in 1986, and the M.Sc. and Ph.D. degrees in computer science from The Center for Advanced Computer Studies at the University of Southwestern Louisiana in 1988 and 1990, respectively.

From 1983 to 1986 he was with the Computer Science Department, Alexandria University, Egypt, as a lecturer. From September 1990 to May 1995 he worked as an assistant professor at the Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia. From May 1995 to December 2000 he has worked as an Associate Professor in the same department. In January 2000 Professor Elleithy joined the Department of Computer Science and Engineering in University of Bridgeport as an associate professor. In May 2003 he was promoted to full professor.

Professor Elleithy published more than sixty research papers in international journals and conferences. He has research interests in the areas of network security, mobile / wireless communications, computer arithmetic and formal approaches for design and verification.

No.	Time (h:m:s:ms)	Frame	Protocol	Addr. IP src	Addr. IP dest	Port src	Port dest	SEQ	ACK	Size
1	16:25:44:362	IP	TCP-> FTP	10.10.1.1	148.166.161.115	12552	21	2704142594	0	60
2	16:25:44:362	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12552	4174686522	2704142595	58
3	16:25:44:382	IP	TCP-> FTP	10.10.1.1	148.166.161.115	12808	21	2720919810	0	60
4	16:25:44:382	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12808	4174729677	2720919811	58
5	16:25:44:422	IP	TCP-> FTP	10.10.1.1	148.166.161.115	13064	21	2737697026	0	60
6	16:25:44:422	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	13064	4174805070	2737697027	58
7	16:25:47:296	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12552	4174686522	2704142595	58
8	16:25:47:296	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12808	4174729677	2720919811	58
9	16:25:47:396	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	13064	4174805070	2737697027	58
10	16:25:53:315	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12808	4174729677	2720919811	58
11	16:25:53:315	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	12552	4174686522	2704142595	58
12	16:25:53:415	IP	TCP-> FTP	148.166.161.115	10.10.1.1	21	13064	4174805070	2737697027	58

Figure 5. TCP SYN Flood run against an FTP server

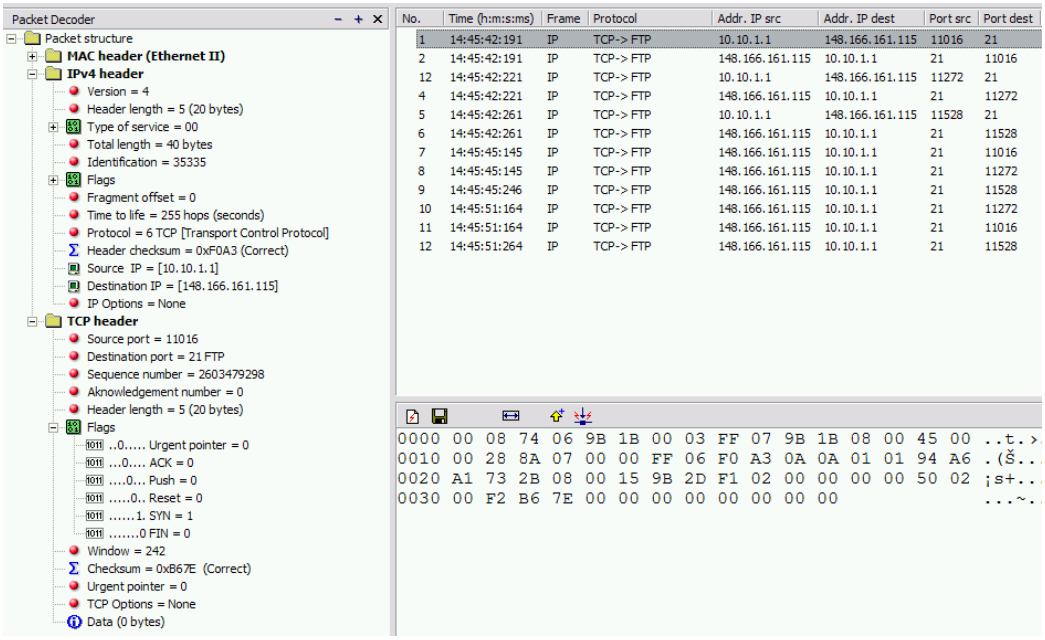

Figure 6. SYN Flood packet with SYN flag set to 1

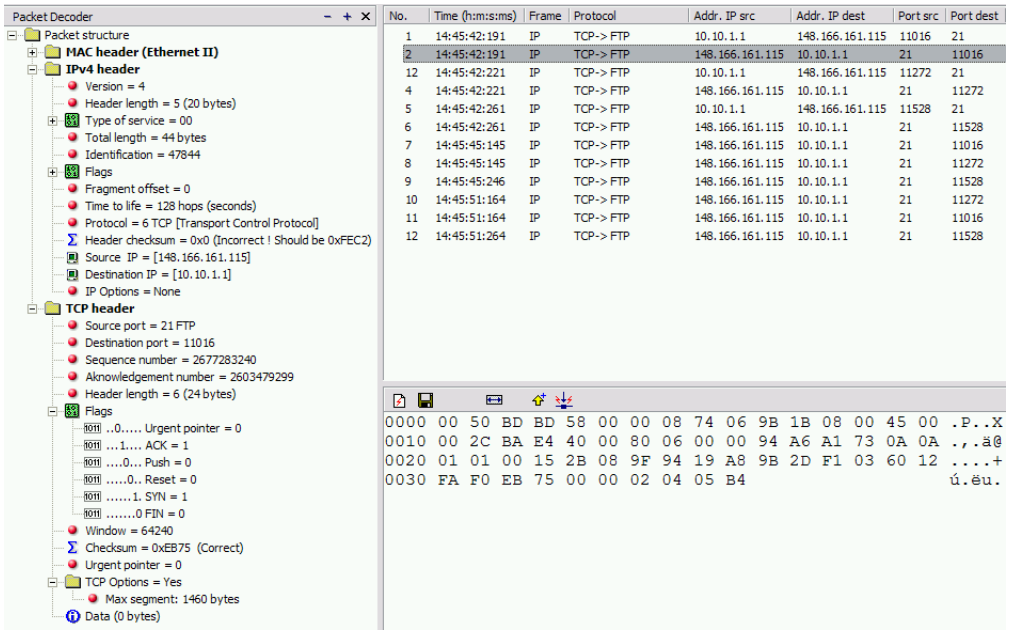

Figure 7. Response packet with ACK-SYN flags set to 1

```
/* IP address information */
struct sockaddr_in sin;
register int i=0,j=0;
int floodcontrol=0;
unsigned short sport=161+getpid();
        . . .
   /* Build TCP header */
packet.tcp.source=sport;          /* 16-bit Source port number */
packet.tcp.dest=htons(dport);          /* 16-bit Destination port */
packet.tcp.seq=49358353+getpid();       /* 32-bit Sequence Number */
packet.tcp.ack_seq=0;         /* 32-bit Acknowledgement Number */
packet.tcp.doff=5;            /* Data offset */
packet.tcp.res1=0;            /* reserved */
packet.tcp.urg=0;             /* Urgent offset valid flag */
packet.tcp.ack=0;             /* ACK flag */
packet.tcp.psh=0;             /* Push flag */
packet.tcp.rst=0;           /* Reset flag */
packet.tcp.syn=1;             /* SYN flag */
packet.tcp.fin=0;             /* Finish sending flag */
packet.tcp.window=htons(242);       /* 16-bit Window size */
packet.tcp.check=0;           /* 16-bit checksum (to be filled in below) */
packet.tcp.urg_ptr=0;         /* 16-bit urgent offset */
   /* Build IP header */
packet.ip.version=4;          /* 4-bit Version */
packet.ip.ihl=5;            /* 4-bit Header Length */
packet.ip.tos=0;            /* 8-bit Type of service */
packet.ip.tot_len=htons(40);          /* 16-bit Total length */
packet.ip.id=getpid();        /* 16-bit ID field */
packet.ip.frag_off=0;         /* 13-bit Fragment offset */
packet.ip.ttl=255;            /* 8-bit Time To Live */
packet.ip.protocol=IPPROTO_TCP;       /* 8-bit Protocol */
packet.ip.check=0;            /* 16-bit Header checksum (filled in below) */
packet.ip.saddr=sadd;         /* 32-bit Source Address */
packet.ip.daddr=dadd;         /* 32-bit Destination Address */
```

Figure 8. Excerpt from the source code