A Defense Against Address Spoofing Using Active Networks

by

Van C. Van

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author . . . .
Department of Electrical Engineering and Computer Science
May 29, 1997

Certified by . . . .
Stephen J. Garland
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A Defense Against Address Spoofing Using Active Networks

by

Van C. Van

Submitted to the Department of Electrical Engineering and Computer Science
on May 29, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis studies a prevalent denial-of-service attack known as SYN-Flooding and presents a possible defense using active network technology. This attack uses "spoofed" Internet addresses to exploit a weakness in the 3-way handshake used by the Transmission Control Protocol (TCP). It can render a server inaccessible to legitimate users or, even worse, bring a server down completely. As yet, there is no accepted solution to this problem in today's network environment.

Active networks, which provide increased computational power within the network itself, have recently been proposed as a means of deploying new network applications and of performing application-specific computation within the network. This thesis illustrates how active networks can be used as defense against this denial-of-service attack. We have built a filter that can be dynamically deployed to filter out forged packets within the network, thereby foiling SYN-Flooding attacks. The current implementation of the filter requires symmetric routing within the network. A future extension to this thesis may be a more general filter that works in less restrictive environments.

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Acknowledgments

Despite what the title page claims, a thesis is never the work of a single individual. If anyone claims otherwise, distrust that person immediately. Many people have contributed countless hours (some during sleepless nights) to make this thesis a reality. At this time, I would like to thank these specific individuals for their valuable contributions.

Stephen Garland, my thesis supervisor, for his guidance, insight, and patience throughout this endeavor without which this thesis would never come to fruition, and for his autograph on the front cover of this thesis without which I would not be able to graduate.

David Wetherall, for his implementation of ANTS, without which this thesis would have no foundation, and for his valuable input on the system design.

David Evans, for helping me get started in this project, and for being such a pleasant and supportive officemate and friend.

Mark Handley, for his valuable input in the initial design phase of the system.

John Guttag, for his faith and trust in me to finish the project after a shaky start.

The other members of the SOFTWARE DEVICES AND SYSTEMS GROUP, especially *Ulana Legedza*, *Li-wei Lehman*, and *David Murphy*, for their helpful comments, feedback, and ideas.

Siu-Li Khoe, for the last-minute midnight reading of this thesis that brought into the light one of the most laughable spelling errors.

Amy Van, my sister, for her love, support, and encouragement, and for the countless number of late night phone calls which helped me relieve a great deal of stress.

My parents, for bringing me into this world and making me the person I am now.

And most of all, **God**, for his undying love and eternal grace. He gives me strength to carry on when it seems all hope is gone.

Contents

List of Figures

Chapter 1

Introduction

The security of computer systems involves maintaining three distinct requirements: secrecy, integrity, and availability [9, 15, 17, 24]. *Secrecy* means that only authorized users are allowed access to system resources. Different types of access include viewing, copying, printing, or even just knowing the existence of the resources. *Integrity* means that only authorized users can modify system resources. Different types of modification include writing, changing, deleting, and creating system resources. *Availability* means that authorized users are not denied access to resources to which they have legitimate access privileges.

Secrecy and integrity have been studied extensively. Robust cryptographic algorithms and protocols provide a defense against attacks on secrecy and integrity. Availability, however, has been studied much less. In fact, the tendency has always been to accept the fact that availability is difficult to maintain. As the information age dawns, however, availability is becoming extremely important. In fact, it is becoming intolerable for some services to be unavailable. America-on-Line (AOL), one of the leading Internet Service Providers (ISPs), was featured in *The Los Angeles Times* when their servers collapsed under increased client load [19]. International Business Machine (IBM) was ridiculed when their web servers could not handle the large number of incoming requests during the first few days of the first tournament between International Grandmaster and World Chess Champion, Garry Kasparov, and the IBM computer, Deep Blue [12]. Availability is something that is very important to the Internet community today and will be for the foreseeable future. Because of this, prevention of denial-of-service attacks, a security threat to availability, is becoming a hot topic in computer systems research.

1.1 Background

Within a network community, denial-of-service attacks come in two different forms, network attacks and host attacks.

Network attacks are aimed at sabotaging an entire network, denying service to everyone on that particular network. They are analogous to blocking off roads so that no one in a community can get anywhere. They can be achieved in many different ways, e.g., congesting the roads, blocking off the roads, or even destroying the roads. In the world of computer networks, the same can be achieved. One could flood the entire network with useless information so that nothing gets through, attack the routers which route packets for hosts, or even physically destroy the network links or routers within a network. This type of attack is hard to mount successfully because it requires a great deal of system resources.

Alternatively, host attacks are aimed at denying users access to a particular server by attacking the specific host, making it unable to perform its services. They can be achieved in many ways. One could monopolize all the resources at a host or corrupt the services on the host by modifying or even deleting them. Usually, this requires "breaking into" the system first by exploiting a security weakness of the system or protocol it uses. Because these types of attacks are aimed at a specific host, other hosts are not affected during the attack. In addition, only access to the particular host being attacked is denied; other hosts will continue to function and perform their services as usual with minimal impact. In this thesis, we focus on a specific host attack in the Internet today.

1.1.1 TCP/IP

Data communication has always been an essential part of computer systems. Unfortunately, different systems have different requirements and specifications, which make them incompatible with each other. In order for communication to occur, a standard must be created and used by everyone who wishes to participate in the communication. This standard dictates exactly how systems communicate with each other.

The most common standard in today's networking environment is the Transmission Control Protocol and Internet Protocol suite, more readily known as TCP/IP [7, 20]. A conceptual model is shown in Figure 1-1. At the lowest level, a connectionless, unreliable delivery service called IP provides the basis on which the entire suite is built upon. This layer is responsible for delivering IP datagrams, the basic unit of information, between hosts in a TCP/IP network. It is *connectionless* because each datagram is treated independently. It is *unreliable* because delivery is not

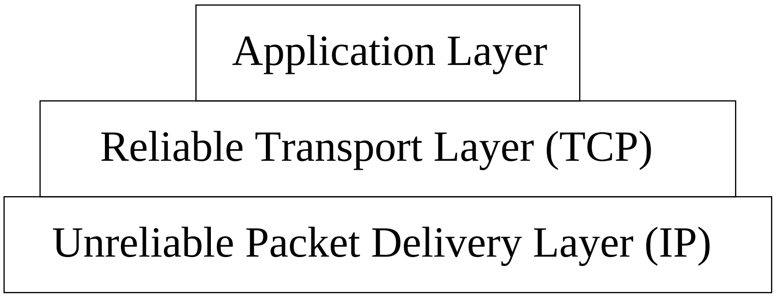


Figure 1-1: Three conceptual layers of network services [7]

guaranteed. Packets[1] may be lost, corrupted, duplicated, or delivered out of order. The next level higher is a reliable transport service layer called TCP. This service provides guaranteed delivery of data through virtual connections. Applications such as FTP, Gopher, and web browsers depend on this service to communicate with other applications on different hosts within a network.

In order to exchange data using TCP, hosts must establish a connection via a three step process called the 3-way handshake [Figure 1-2]. First, a client notifies a server that it wants a connection by sending a synchronization (SYN) packet to the server. Upon receiving this packet, the server will respond with a synchronization-acknowledgment (SYN-ACK) to the requesting client. When the SYN-ACK is received by the client, it sends an acknowledgment (ACK) to the server. When the server receives the ACK, a connection is established and data transfer may begin.

1.1.2 The SYN-Flooding Attack

The TCP SYN-Flooding attack [1, 3, 4, 6, 8, 18] exploits a weakness in the 3-way handshake protocol used by TCP to establish connections. Because the source code for this attack is highly available (it can be found in a number of sites across the Internet) and because it is easy to mount such an attack, this denial-of-service attack has been on the rise [13].

The point at which the TCP handshake can be exploited is when a server has responded with SYN-ACK and is waiting for an ACK from the client. At this point, a pending (half-open) connection exists. The server keeps track of all the information needed to complete the connection in a data structure, in this case, a stack. If a client responds with an ACK, the connection is completed, and the pending connection is removed from the pending connections stack. If not, the server waits a certain

[1] The term *packet* is used loosely to refer to any small block of data sent across a packet switching network. It is often used synonymously with the term *datagram*.

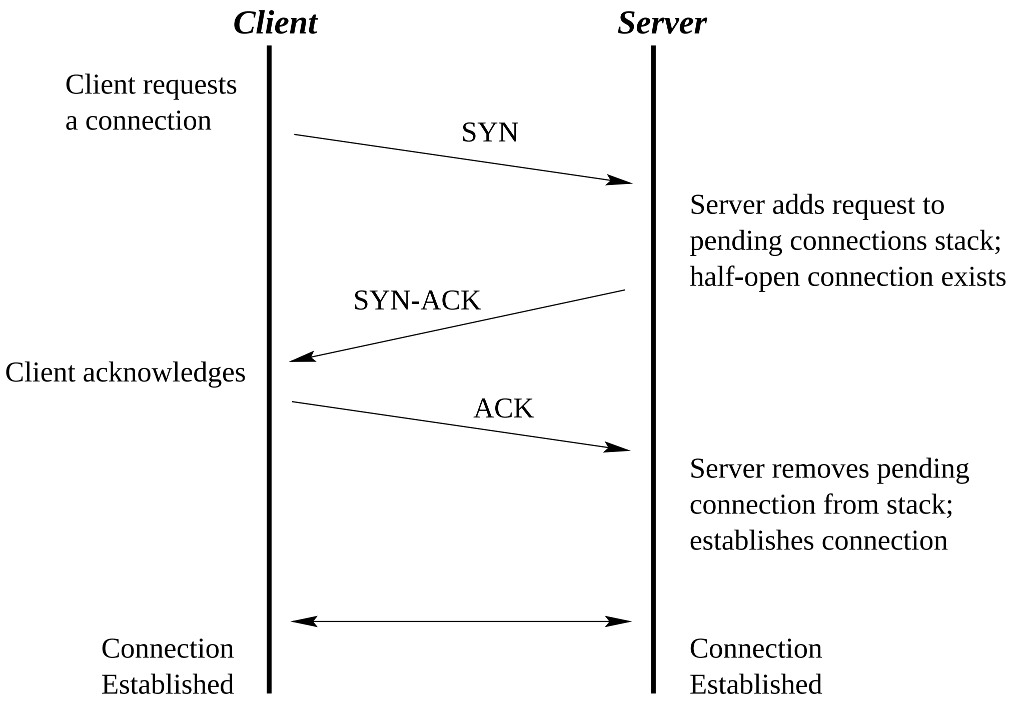


Figure 1-2: TCP 3-Way Handshake

amount of time (which varies depending on implementation, but usually around 75 seconds), after which the pending connection times out and is removed from the stack.

An attacker can mount a denial-of-service attack on a server by sending many SYN packets to the server, thereby filling up the pending connections stack and making the server unable to honor further requests for connections. Although the pending connections will eventually time out, an adversary can continue to flood the server with SYN packets faster than the time-out rate. This is very easy to achieve because stack sizes are usually very small. For example, the BSD implementation has a stack size of 5, while Linux has 6. The majority of the implementations of TCP have a stack size of no more than 10.

This attack can affect different systems in different ways. Some systems are just rendered inoperative as they sit and wait for ACKs. Others may "crash" under the overload. Those that do not crash can work with existing connections and can originate requests if memory is not exhausted, but they cannot handle new connection requests. Because this attack is only aimed at availability, data integrity and secrecy is not compromised.

The key to this and similar attacks is forged IP source addresses known as *IP Spoofing* [2]. For this attack to be effective, the adversary must forge the source IP address to reflect 1) a client other than itself (to avoid being caught) and 2) a client that does not exist or cannot respond to SYN-ACKs. The reason that (2) must be true is that if the client does exist and can respond, it will respond with a RST (reset) when it receives a SYN-ACK from the server. The RST will tell the server to destroy the half-open connection because the client did not request it, thereby foiling the adversary's plan to fill the server's data structure.

In most cases, a server cannot defend itself against a SYN-Flooding or other IP Spoofing attacks because it cannot distinguish legitimate packets from fake ones. In addition, it is extremely difficult, if not impossible, to ferret out the perpetrator of such attacks. When a server is being attacked, there is no way to determine the true source of the fake packets since network routers have no concept of going "backwards" from destination to source. Usually, a server under attack has no choice except to ride-out the attack (which may or may not be possible) or shutdown until the attack has ceased.

IP Spoofing can be used to attack a server in ways other than SYN-Flooding. For example, an adversary can use IP Spoofing to exploit certain applications that use IP addresses for authentication purposes to gain user and/or possibly root access to systems [2]. For this thesis, we are only interested in studying IP Spoofing as a mechanism to deny service to other users, such as the SYN-Flooding attack.

1.2 Goals

The TCP SYN-Flooding attack is a recent phenomenon. Although there are many "fixes" proposed by the Internet community, there is no accepted solution in today's network environment [**?**]. The goal of this research is to find an acceptable solution to this problem using active network technology [21]. An acceptable solution, we propose, is one that meets the following criteria.

- It should stop a SYN-Flooding attack.
- It should be transparent to end-users.
- It should be easy to deploy.
- It should not degrade system performance dramatically.
- It should not degrade network performance dramatically.

1.3 Thesis Overview

In the next chapter, we examine existing “solutions” to the problem and evaluate their effectiveness using our proposed criteria. In Chapter 3, we present an overview of active networks and ANTS [23], a toolkit for building active network applications. In Chapter 4, we present our solution to this problem using active networks. Chapter 5 draws conclusions on the effectiveness of our solution, discusses some issues that need further investigation, and suggests some possible future work.

Chapter 2

Existing "Solutions"

Since the onset of the SYN-Flooding attack, a great deal of effort has been spent on trying to find a solution to the problem. To date, no acceptable solution has emerged; however, a number of fixes have been proposed which have some, but not all, the properties presented in Section 1.2.

2.1 Increased Table Size

A simple, but naive, fix to this problem is to increase the stack size [1, 18]. If there are more entries in the stack, then it is that much harder to fill up. While this allows a system to handle more half-open connections, it is not an acceptable solution because it does not stop an attack and it can degrade system performance dramatically. To successfully mount an attack against a server with a larger stack size, an adversary simply needs to send more packets to flood the server. In addition, increasing the stack size may affect system performance as a whole. Each entry within the stack takes up a great deal of system resources. In one version of RedHat Linux, each entry requires 520 bytes of I/O buffer space [18]. If there are 2048 entries, it would require about a megabyte of memory *per port*[1]. Increasing the stack size will decrease available resources, thereby degrade system performance.

[1]TCP uses 16-bit port numbers to identify different applications running on a system. For example, the FTP service is on TCP port 21, while Telnet is on port 23. Ports 1 to 1023 are well-known ports and are managed by the *Internet Assigned Numbers Authority* (IANA). In addition, most TCP/IP implementations allocate ephemeral (short lived) ports between 1024 and 5000 [20].

2.2 SYN Cookies

As a complement to increased table size, Berkeley Software Design, Inc. (BSDI) has proposed "SYN Cookies" as a countermeasure [1]. When a SYN is received, only a few bits of information are stored in the data structure, e.g., the sender address. The rest of the information is sent back to the client as a "cookie" along with the SYN-ACK. In order to establish a connection, the client must reply with an ACK that contains the cookie. This significantly reduces the resources that an attacker can consume at the server and greatly increases the number of pending connections the server can handle.

This is not an acceptable solution because it is not transparent to users, it does not stop the problem, and it may degrade system/network performance. In order for this scheme to work, the TCP protocol itself must be changed to accommodate SYN cookies. Changing the protocol means that everyone using the TCP protocol must upgrade. This is a long and difficult process. In addition, this introduces problems with backwards compatibility and others associated with software upgrades.

Even after the change has taken place, the problem remains. Like the previous proposal, an adversary simply needs to send more packets to make the attack successful. In addition, this fix may adversely affect network performance. The cookies contain a great deal of information and therefore are very large, just as the stack entries are large. In other words, they take up a lot of network bandwidth. An adversary can cause a victim server to inadvertently flood the network with many cookies by sending it many SYN packets. Hence, the adversary can cripple not only a server, but the entire network as well, degrading both system and network performance.

2.3 Management of Pending Connections

Another way to cope with SYN-Flooding is to replace pending connections with new requests as they arrive [18]. If the stack is full and a new request for a connection arrives, the server evicts one of the pending connections and replaces it with the new request. Two replacement strategies have been proposed: 1) replace the oldest pending connection, and 2) replace a pending connection at random.

Both strategies are not acceptable because they do not stop an attack. With the first strategy, an adversary can send SYN packets fast enough that a "real" request will be replaced before the client can respond with an ACK. The second strategy, random replacement, works better as there is no guarantee that an adversary can successfully deny service to every user. If the adversary sends enough packets, however, it is probabilistically unlikely that a legitimate client can establish a connection. Pending connection will be replaced while the client is still trying to reply with an ACK. In this manner, an adversary may not be able to deny service completely, but can still generate a great deal of annoyance.

2.4 Firewalls

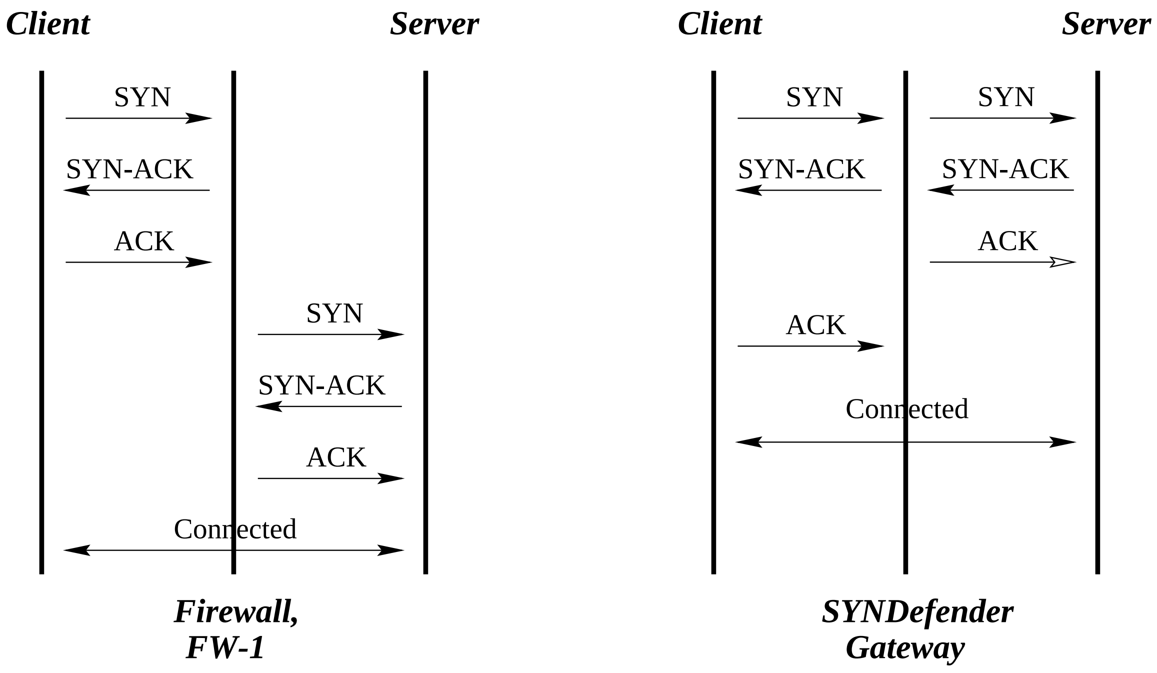


Figure 2-1: Firewall filters

Another idea is to set up a firewall that filters incoming packets. A firewall is nothing more than a computer, router, gateway, or similar device that serves as an access control for a network. Depending on the software, a firewall can be used for different security purposes. Check Point Software's firewall, FW-1 [4], prevents a connection from being requested from the server until the three-way handshake is completed with the firewall [see Figure 2-1]. At that point, the original SYN packet is sent to the server. The server replies to the firewall with a SYN-ACK, and the reply is once again forwarded. Once this is done, the server is connected to the client and further data transfer can continue without any intervention from the firewall.

Another of Check Point's products, SYNDefender Gateway, establishes the full connection with the servers, while the firewall awaits the ACK from the client [see Figure 2-1]. Basically, when a client sends a SYN to a server via the gateway, the firewall relays that SYN packet to the server. The server responds with a SYN-ACK and this is relayed onto the client. At the same time, the firewall completes the connection with the server by responding with an ACK to the server. The connection is established and the pending connection is taken off the stack. Then the firewall waits for the real ACK from the client. When the firewall receives the ACK, a connection is established between the client and the server.

Both of these firewalls, however, do not solve the problem and are not user transparent. Instead, they "push" the problem away from the server and into the firewall, introducing an even greater problem. The weakness in these schemes is, of course, the firewall. If an adversary is successful in crippling the firewall (with the same methods), then the entire subnet behind the firewall is affected. In addition, firewalls are not user transparent. Setting up a firewall requires reconfiguring an entire network to use the firewall. Since the firewall is a single point of failure, it is important to keep it up and running, no matter the cost. A great deal of resources is spent doing this because if the firewall malfunctions, due to an attack or otherwise, an entire network is affected. This alone makes it an undesirable option.

2.5 Ingress Filtering

Another proposal is to require all Internet Service Providers (ISPs) to filter packets to ensure IP header addresses are not forged, as shown in Figure 2-2. Basically, if all connecting routers (routers that connect a subnet to the Internet) filter their packets to ensure that outbound packets are really from the subnet they are responsible for, they can filter out forged packets and prevent the forgery essential to this type of attack. This type of filtering, known as Ingress Filtering [1, 2, 3, 6], is the leading proposal of many experts.

This proposal, however, has its own defects. Although it does stop an adversary from spoofing addresses outside of a subnet, it does not stop an adversary from spoofing addresses within the same subnet. For example, if the MIT domain (18.X.X.X) was Ingress Filtering, it forbids someone from within MIT to spoof an address outside of MIT, but it does not prevent an adversary from spoofing other hosts within the MIT domain. It also requires a concerted effort by *all* ISPs. In addition, it requires many changes to existing routers, switches, and gateways. Some routers in the Internet today do not have filtering capabilities. Routers that have filtering capabilities may not have the bandwidth to support additional filtering requirements. In order for this to work, many of these routers will have to be replaced. Finally, subnets may experience a significant performance hit since Ingress Filtering will have to be done constantly, checking every packet, in order to prevent attacks. All of these reasons make Ingress Filtering an undesirable solution.

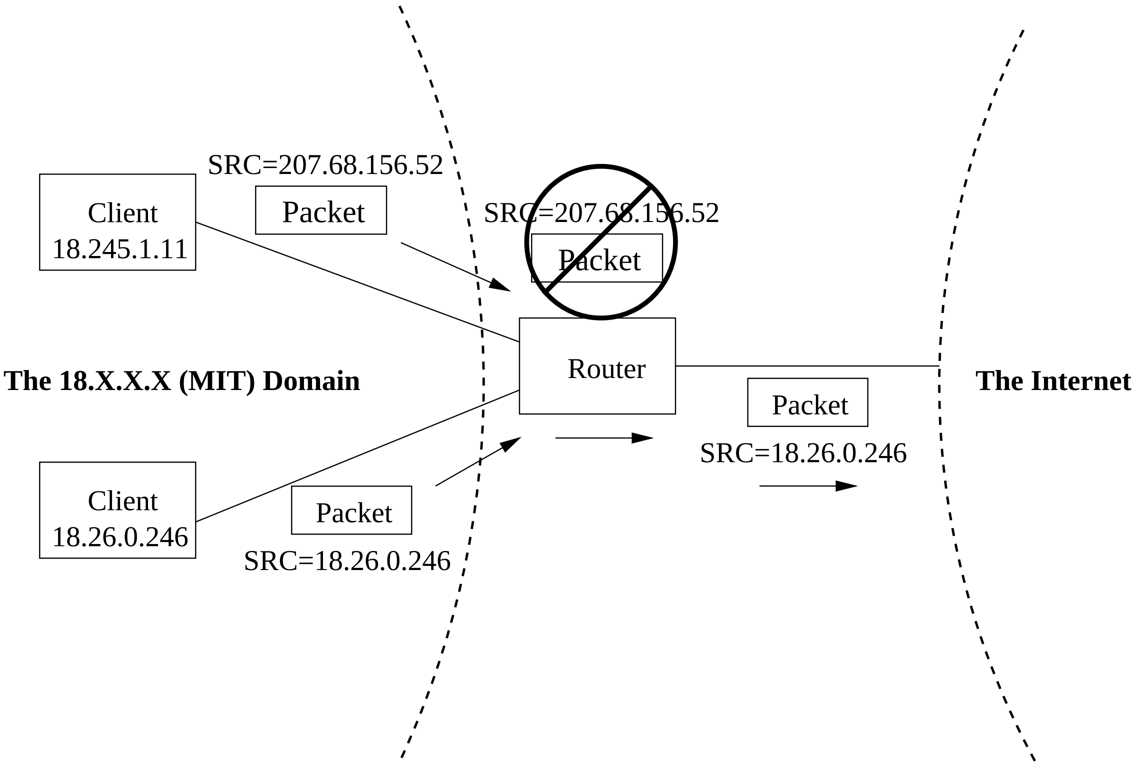


Figure 2-2: Ingress Filtering

2.6 Summary

All these proposals lack some of the fundamental qualities that we listed in Section 1.2. Most of the proposals do not stop an attack. They were designed to help systems ride out an attack while taking a serious performance hit. The leading proposal is Ingress Filtering, which partially stops the problem by restricting address spoofing to within a single subnet. What we would like to do is stop address spoofing completely. The current network technology offers little assistance in this area. Instead, we use a new technology, active networks, to help us stop address spoofing.

Chapter 3

Active Networks

Active networks [21] provide a new way of thinking about a network environment. Traditional networks are based on an "end-to-end" model, in which computation is done at the endpoints, with only routing and header processing occurring within the network. All packets are routed through a network independent of content. Active networks break from this model by allowing computation to occur within the network. In an active network, the nodes (i.e., the routers and switches) can perform computations on individual packets in addition to routing them. This extra computation can be defined by users, who can inject programs into the network to customize processing of user and/or application specific data. This chapter discusses active networks in general and a prototype implementation of an active network called ANTS [23].

3.1 Architecture

There are two approaches to building an active network, a discrete and an integrated one [21]. In the following sections, we present both approaches with more emphasis on the integrated approach used by ANTS.

3.1.1 A Discrete Approach

The discrete approach separates the mechanism for injecting programs into a "programmable" node from the actual processing of packets as they flow through a node. Users send a program to a node as they would to a host. This program would then be stored at the node. When a packet arrives at the node, the corresponding program is selected using some header information and then executed. When a new version of the program is necessary, or if a different type of processing is required, the user can send the new program to the node to replace the old one.

This approach is definitely preferable when programs are relatively large compared to the packets. This approach also maintains a modularization between user data and program, which may be useful for network management tasks.

3.1.2 An Integrated Approach

A different approach, the one used by ANTS, is to integrate the program with every packet. In this manner, every packet that is sent contains not only data, but also a program. The term *capsule* is used to describe these new types of packets. When a capsule arrives at a node, its contents are evaluated using the program in the capsule.

The traditional router or switch, which is responsible for routing and header processing, is replaced by an active node. In addition to capsule routing, an active node consists of three major components: a code loading mechanism, a transient execution environment, and a more permanent storage area. When a capsule arrives at an active node, the code associated with the capsule is loaded by the code loading mechanism. The capsule and the program loaded are then passed on to a transient execution environment. There the program is allowed limited access to node resources such as memory and other storage components. There is a restricted set of capsule primitives that comprise all the actions that a capsule can perform at a node. Capsules are also allowed to retrieve or store information in a more permanent storage area, the node cache. The result of processing may involve zero or more capsules being sent out into the network and/or a modification of the node cache. When processing is completed, the transient environment is destroyed and resources held by the capsule are released.

3.2 ANTS – An Active Network and Toolkit

ANTS is an active network toolkit developed at the Massachusetts Institute of Technology (MIT) Laboratory for Computer Science (LCS) by D. J. Wetherall [23]. ANTS is toolkit to help build and maintain active network applications. ANTS also includes an implementation of an integrated active network. ANTS is written in Java and runs as a user-level application on the Linux operating system.

The entire ANTS environment consists of three major classes, as shown in Table 3.1. The `Application` class is the user interface part of the environment. It is also responsible for sending and receiving capsules, which are implemented by the `Capsule` class. The `Node` class is the implementation of an active node.

Class	Methods
`Application`	`send`, `receive` (upcall), `node`
`Capsule`	`evaluate`, `length`, `register`, `serialize`, `deserialize`
`Node`	`address`, `get`, `put`, `routefornode`, `delivertoapp`

Table 3.1: Key Classes and Methods [23]

3.2.1 Applications

Active network applications are written by specializing the `Application` class. This class provides some basic methods that interact with the node it runs on to send and receive capsules. Before an application can send a capsule, however, it must first register the capsule type with the local node using the `Capsule.register` method. This is a requirement imposed by the code distribution protocol [see Section 3.2.3]. Once the capsule type is registered, an application can send instances of the capsule type into the active network environment using the `send` method. In addition, an application must have a `receive` method, which will be executed when a capsule arrives at the node. Currently, only one application can be running on a node. In the future, multiple applications can run on a node, and incoming capsules will be directed to their respective applications.

3.2.2 Capsules

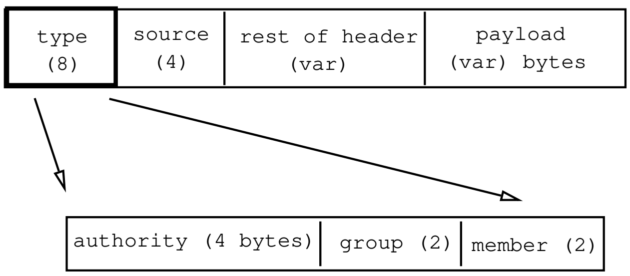


Figure 3-1: Capsule Format [23]

In ANTS, capsules are implemented by the `Capsule` class. Capsules have a specific format as shown in Figure 3-1. The first 8 bytes of a capsule is the capsule ID, which is used to identify the capsule type and where the code for the capsule resides. It is represented in the `Capsule` class as the MethodID (MID), which must be unique for each capsule type[1]. The MID is divided into three parts, an authority (4 bytes), a group (2 bytes), and a subgroup or member (2 bytes). As each capsule type may have unique and customized routines, there must be a way to locate the code and transport it to a particular node when necessary. The authority is the identifier of the node that is the repository of the processing routines associated with this capsule type. The group and member value provide a shallow hierarchy for capsules. If two capsules are of the same group, they are considered as part of a single protocol, and their associated routines are transported around the network as a single unit. This is useful for applications that spawn capsules in the middle of the network.

The next 4 bytes indicate the source address of the capsule; currently, the system uses IPv4 style addresses [7]. The rest of the capsule contains a variable header and payload, depending on the type of capsule. The variable header field contains, among other things, information necessary for routing, e.g., the destination address. The payload field may contain data that is necessary for capsule processing at a node.

There are actually two different types of capsules: user capsules and system capsules. Normal applications use user capsules. A system capsule has more privileges and is used by ANTS itself for special purposes. For example, `DLRequestCapsule` and `DLResponseCapsule` are system capsules used by the code distribution mechanism. The capsule class hierarchy is shown in Figure 3-2.

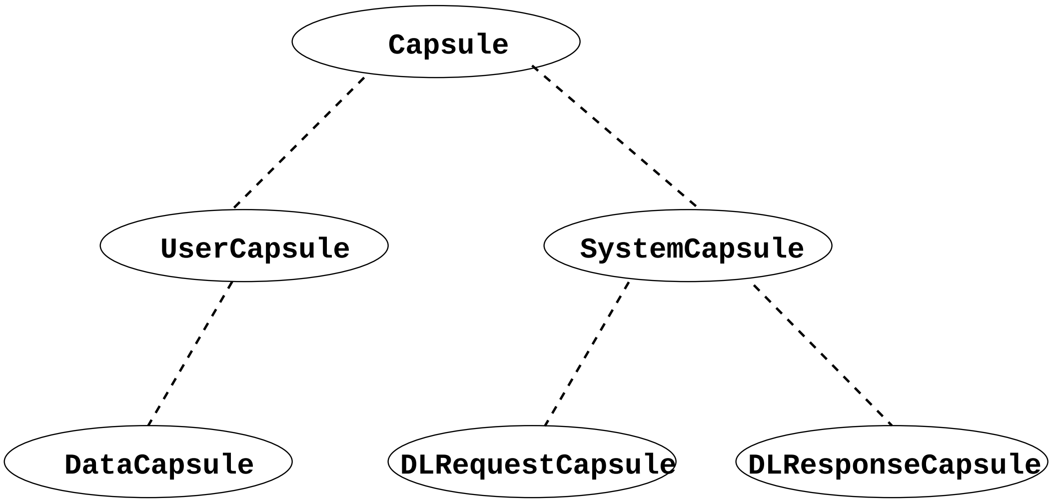


Figure 3-2: Capsule Class Hierarchy

Each capsule type must also provide a certain set of required methods. For

[1]Currently, ANTS does not provide a mechanism to generate unique MethodIDs. Users must generate their own MethodID and hope there is no collision within the network. A future version will use a server hash of the code as the MethodID.

marshaling purposes, the developer must provide a `length` method which returns the length of the capsule in bytes. In addition, a `serialize` and `deserialize` method are required to marshal objects into data streams for transmission and back. For capsule processing, an `evaluate` method is required. This is the heart of the capsule; this method dictates all the processing that will be done when a capsule arrives at a node. Finally, before a capsule can be used (sent into a network) by an application, it must first be registered. Registering a capsule indicates that the capsule routine resides at a particular node. This is necessary for the dynamic code loading mechanism explained in the next section.

3.2.3 Active Nodes

Active nodes are the processing engines that receive capsules, load capsule routines, process the routines, and schedule capsules for further transmission if necessary. They are basically the environment in which capsules run. An active node, including the runtime, object cache, and demand loading protocol, is implemented by the `Node` class. This class provides a set of "primitives" used by capsules. The `address` method returns the address of the current node. The `get` and `put` methods give capsules the ability to store and retrieve objects in the node cache. The cache stores objects for a period of time, as requested by capsules, but not exceeding a maximum time dictated by the node itself. The cache uses a least-recently-used (LRU) strategy to evict items. The node offers no guarantee that an object is cached for the duration of time specified by the capsule or for the maximum cache time since the object may be evicted before it times out. The `routefornode` method is used to route packets to their destination node. If the current node is the destination node, the `delivertoapp` method is used to deliver the capsule to the application.

The demand loading protocol, the mechanism that loads capsule programs into a node, works as shown in Figure 3-3. When a capsule arrives at a node (1), the node first checks its cache to see if the code is already present. If so, no further code loading is necessary. If not, the node will query the last node visited by the capsule for the code (2). If the previous node has the code, it will respond, and the code will be loaded (3). It is unlikely that a node cannot retrieve the necessary code from the previous node because either (a) the previous node just processed the capsule before routing it to the current node, which means the code is in the node cache of the previous node, or (b) the previous node is the originator of the capsule, which means an application running on that node must have registered the capsule [see Sections 3.2.1 and 3.2.2]. When an application registers a capsule, it indicates that it has the processing routines associated with the capsule or, at least, knows how to find them. Hence, the node that injected the capsule into the network will always be able to produce the capsule routines. On the other hand, it is possible for a previous node (not the originator) that just processed the capsule to not be able to produce the code.

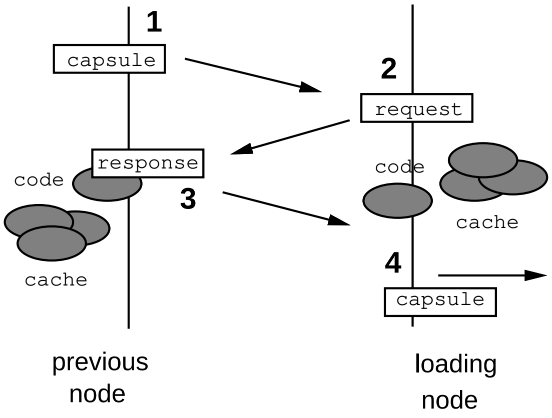


Figure 3-3: Demand Loading Protocol [23]

Due to a great number of capsules going through that node at a particular time, the capsule routines may have been flushed from the cache before the request arrived. If this is the case, the querying node will use the authority field in the MethodID of the capsule to retrieve the code. This is less efficient, because the code may reside on a node far away, and therefore, is only used when the first method fails to yield code.

3.2.4 Routing

Routing in ANTS works much the same way as routing in the Internet today. It uses an algorithm known as "next-hop" routing [7]. Because the Internet is large, it is not feasible to keep track of all paths from every possible source to every possible destination. Not only would this consume too much space, but it would take too much time to search through the tables to route a packet. As a result, routing tables do not have information about the exact location of all nodes. Instead, they keep track of some nodes, usually those within the same local area network. The routing tables contain a default "next-hop" address for addresses that the router does not know how to get to. When a packet arrives and needs to be routed, the routing table is queried and the result is the address of the next node to which the packet should be sent along the way to its final destination. The list of routers, or nodes, from the source to the destination is the *route*, or path.

In a TCP/IP network, as well as ANTS, each packet is routed independently [see Section 1.1.1]. This means that different packets from source A may take different paths to destination B depending on network conditions, e.g., congestion or broken links. In addition, routing in TCP/IP networks and ANTS is *asymmetric*, which means that routes from A to B may not be the same as routes from B to A. This is especially true in the Internet today. Asymmetric routing, however, adds complexity to the network and makes network troubleshooting difficult [16]. To reduce complexity, we have added the constraint of *symmetric routing* to our active network. This means that routes from A to B are the same, but opposite, as routes from B to A. This is actually necessary for our filtering algorithm to work.

3.2.5 Security Issues

ANTS does not provide any type of authentication or traditional security schemes to ensure the safety of running foreign code on an active node [23]. Instead, it relies on the safety mechanisms of Java (e.g., byte-code verification) to execute untrusted code. This, of course, is not sufficient to ensure a robust network. For example, it is difficult to guarantee the uniqueness of a MethodID for individual capsule types, because different users writing different capsule classes can accidentally use the same MethodID. Additionally, there is no safety mechanism to prevent capsules from corrupting the objects stored in the node cache by other capsules. There is also no safeguard against capsules monopolizing node resources. It is clear that additional security mechanisms are necessary to ensure secrecy, integrity, and availability for all the users in the network. These mechanisms have not yet been implemented and are currently being investigated.

These security concerns have a great impact on our application as well as other active network applications. We depend on the unique MethodID to make sure that our capsule routines are loaded instead of another user's capsule routines. We assume that an adversary is unable to forge a capsule type, i.e., make a capsule that looks like our capsule but has a different `evaluate` method. We also trust the active nodes to safeguard the objects we store within the node. Finally, we trust the active nodes to fairly ration processing time so that we get a "fair" share. Although these mechanisms are not in place, we (and other active network applications) assume that they exist. Without these assumptions, our model of the network breaks down, and our solution will not work.

Chapter 4

Dynamic Filtering

The leading proposal to the problem of address spoofing is Ingress Filtering. This proposal, however, has its drawbacks. It fails to prevent address spoofing within a domain, and it is heavyweight. It needs to be implemented everywhere and must filter continuously. It also requires a concerted effort by all ISPs to reconfigure and/or update their routers to accommodate these changes, which is a long and arduous task. On the other hand, an active network allows dynamic deployment of new programs into the network, including routers and switches, without the need for a community effort[1]. Instead, an individual user or system can inject programs that can perform user and/or application specific processing, into the network. With an active network in place, we can dynamically deploy a filter into the network whenever an attack is suspected to filter out packets that have spoofed source addresses.

4.1 System Overview

We have implemented a system within the ANTS environment that can detect a SYN-Flooding attack and dynamically deploy a defense mechanism to filter out spoofed capsules within the network [see Figure 4-1]. The main application is the `TCPApplication`, which uses the `TCPCapsule` as a part of its protocol. It implements both TCP and a defense mechanism that detects attacks and dynamically deploys a filter, the `DefendObj`, using a `DefendCap`. The filter will ensure that source addresses of capsules destined for the host are not forged. As soon as forgery is detected, forged capsules are dropped. This filter is progressively *pushed-back* into the network so as to get as close to the adversary as possible. This

[1] It is true, however, that a community effort is required to install an active network. In this case, we assume that an active network is already in place.

way, the forged capsules will be dropped as soon as possible. This helps thwart the attack and reduce the network congestion it causes. To test our implementation, we use the `FloodApp`, a program that simulates a TCP SYN-Flooding attack similar to the ones mounted in today's networks.

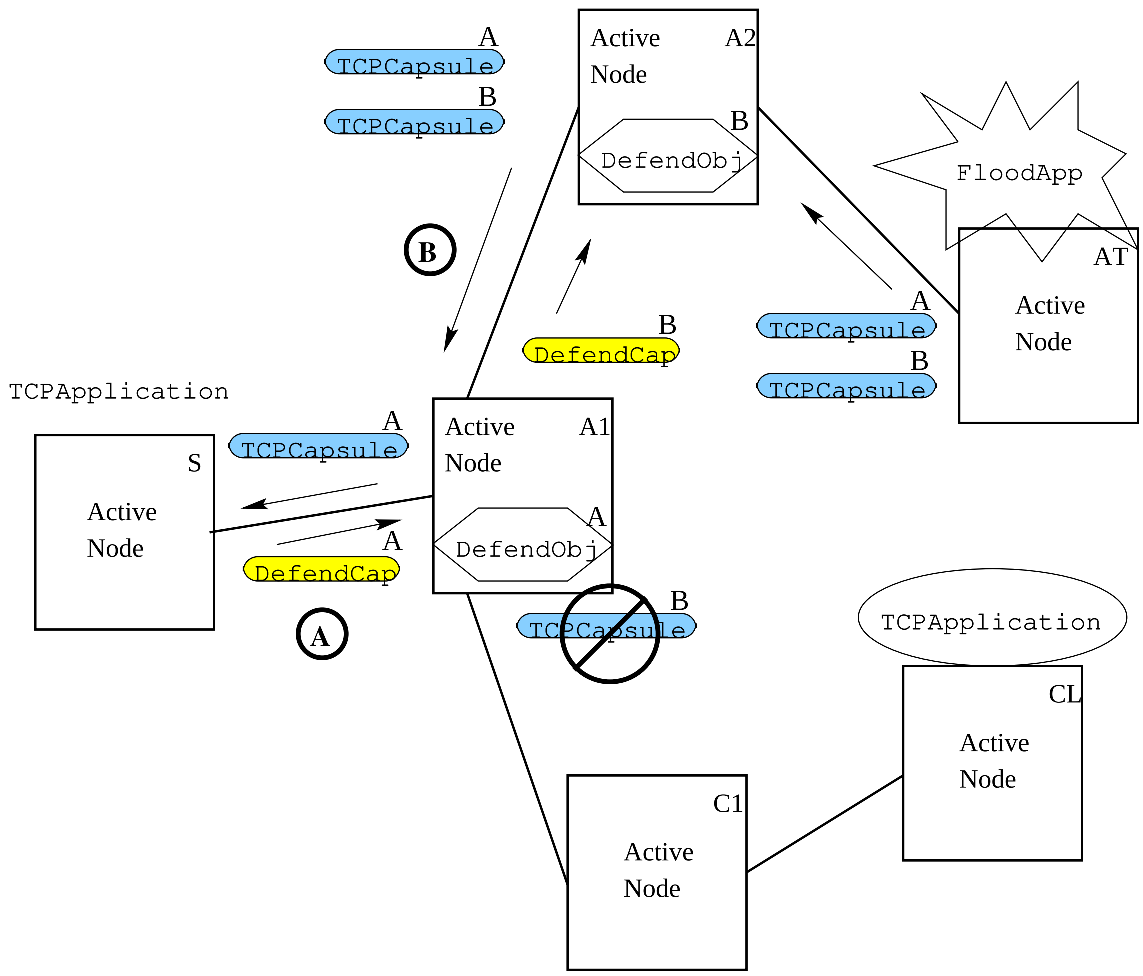
Our filter is different from an Ingress Filter in a number of ways. First, we filter all forged packets, even those within the same domain. Second, our filter is a lightweight filter deployed only when necessary, i.e., during an attack. Ingress Filtering, on the other hand, needs to be running continuously. Third, Ingress Filtering needs to be implemented everywhere (all points of connections). Our filter automatically fans out into the network only to where it is necessary. Fourth, Ingress Filtering filters every packet going through the router. Our filter implements *selective filtering*, which is more efficient [see Section 4.4].

4.2 The `TCPApplication`

The `TCPApplication` is similar to TCP in today's networks. Because we are only concerned with how adversaries use address spoofing to exploit TCP, the `TCPApplication` only simulates the 3-way handshake used by TCP to initiate a connection between a client and a server. The `TCPApplication` is a subclass of the `Application` class (Section 3.2.1). It uses the `TCPCapsule` (Section 4.3) for communication.

This application works much the same way as TCP in a traditional network. The application remains idle until a `TCPCapsule` destined for it arrives. When such a capsule arrives at the node on which the application is running, the capsule is delivered to the application, by calling the `receive` method. The capsule is then processed by the `TCPApplication`. If it is a SYN packet, an appropriate pending connection is created and stored in the pending connections stack, which has a maximum stack size of 5, similar to those in traditional networks. An appropriate SYN-ACK is then returned to the source address indicated in the capsule header, and the application returns to idle as it waits for the ACK from the client. Finally, when it receives an ACK, it removes the pending connection from the stack and creates a `connection` object, and places it in the `connections` table which is substantially larger than the half-open connections stack.

The `TCPApplication` is also capable of initiating a connection by sending a SYN to another server and replying with an ACK when the appropriate SYN-ACK is received. It also keeps track of the number of SYN capsules received, half-open connections, and full connections. The number of SYN capsules is incremented whenever a SYN capsule arrives and decremented whenever a connection is established. The half-open connections time-out and are removed from the table if no



(A) During an attack, when a `TCPCapsule` arrives, the `TCPApplication` will send a `DefendCap` to create a `DefendObj` at the previous node, A1, to filter out forged packets.

(B) When the node A1 receives a spoofed `TCPCapsule`, the `DefendObj` will filter it out and also "push" the filter back by sending a `DefendCap` to create a new `DefendObj` at the previous active node, A2.

Figure 4-1: Dynamic Filtering System

ACK is received in 75 seconds. Our application is also able to send and receive RSTs when necessary.

In addition to the regular TCP protocol, our `TCPApplication` has a mechanism to deploy a `DefendObj`, when an attack is suspected. This can be done in two ways: 1) by user intervention or 2) automatically. An administrator of the system can deploy the `DefendObj` by manually instructing the system to do so. On the other hand, the system itself can deploy the filter automatically when it detects an attack. It detects an attack by a simple mechanism based on the number of SYN capsules arriving after the pending connections stack is full. In order to flood a system, an adversary must continually send SYN capsules to a host, even after the stack is full, to ensure that the stack remains full. To detect an attack, the system checks to see the number of incoming SYN capsules after the stack is full. If this number rises above a certain watermark, which in practice would be determined experimentally based on the load distribution of the particular host, an attack is suspected. Each individual system can be configured to a different watermark that is appropriate to its specific environment. We used a value of 10 in our experimental environment.

4.3 The `TCPCapsule`

A single capsule type is used for communication between client and server. The `TCPCapsule` is implemented as a subclass of the `DataCapsule` type defined in ANTS. Information regarding the MethodID is inherited from the `DataCapsule`, while other information is found in the variable header field. Our TCP header in the variable header field includes source address, source port, destination address, destination port, capsule type, capsule time, and previous node. The capsule type field is not to be confused with the MethodID. This field indicates whether the capsule is a SYN, SYN-ACK, ACK, or RST. The most important field is the previous node, which indicates the address of the last node that transmitted the capsule. This may not be the original sender. This information is vital to the filtering algorithm which is explained in Section 4.4.2.

The reason we use our own source address header field instead of the one provided by ANTS is that we assume the adversary can spoof the source address. The source address header field provided by ANTS is very difficult, albeit not impossible, to spoof. It is used by the dynamic code loading protocol to locate the processing routines of the capsule. This research, however, is not an exercise in how to make it hard for an adversary to spoof addresses. We assume that an adversary is capable of doing this, and we make it simpler for the adversary to spoof the address by providing a different easily accessible field that is much easier to tamper with.

When designing the `TCPCapsule`, our goal was to make it as lightweight as possible so as not to affect performance. When a TCPCapsule arrives at a node, it first checks if it has arrived at its destination. If it has, it delivers itself to the TCPApplication and capsule processing is completed. If not, it checks to see if a DefendObj is present in the node cache. If a DefendObj does not exist, the TCPCapsule continues on its route to the destination after updating the prevNode field. If a DefendObj does exist, the object's checkCap method is called to check the capsule. If the TCPCapsule passes the check, it is routed, and processing is completed. If it does not pass the checks, it is dropped, and processing ends. [See Figure 4-2]

```
public boolean evaluate(Node n)
{
  TCPCapsule c = this;

  if ( n.address == c.dstadd )
    return n.delivertoapp(c, c.dstprt);
  else
    if ( c.capType == SYN )
      {
        DefendObj defender =
          castDefendObj(n.softstate.get(DefendCap.DEFEND_KEY));
        if ( defender != null )
          c = defender.checkCap(c, n);
      }
  if ( c != null )
    {
      // Stamp capsule before sending
      c.prevNode = n.address;
      return n.routefornode(c, c.dstadd);
    }
  else
    // Capsule did not pass the filter; capsule dropped
    return false;
}
```

Figure 4-2: TCPCapsule Program

4.4 The DefendObj

The DefendObj is a filter that is stored in the cache of an active node. Normally, this object does not exist anywhere within the network. It is deployed only when a host suspects an attack. Dynamic deployment avoids taking up unnecessary space within node caches when filtering is not necessary. The DefendObj is created within a node by a DefendCap [see Section 4.4.1]. Once created, the object is then responsible for filtering out forged capsules and for pushing the filter back as close to the adversary as possible.

The DefendObj is designed for *selective filtering*. It only filters TCP SYN capsules destined for servers that have requested filtering. TCP SYN capsules destined for other servers are not filtered. All other capsule types pass the node as if the filter does not exist. This is so that the network does not experience a degradation in performance when filtering is in place.

4.4.1 The `DefendCap`

The `DefendCap` is used to create a `DefendObj` at a node and/or request that a `DefendObj` begin filtering for a specific server. When this capsule arrives at a node, it checks to see if a `DefendObj` already exists. If a `DefendObj` does not, the object is created and added to the node cache. The capsule adds the address of the server under attack to the object as one of the servers for which it is filtering. If the object already exists at a node, the capsule just adds the server address to the list of servers that are filtering capsules. This means that more than one server can share the same `DefendObj`, thereby conserving resources at the active node. The `evaluate` method of the `DefendCap` is shown in Figure 4-3.

```
public boolean evaluate(Node n)
{
  boolean b;

  if ( n.address == dstadd )
    // Capsule arrived at its destination
    {
      // Checks to see if filter object already exists in node cache
      DefendObj filter = castDefendObj(n.softstate.get(DEFEND_KEY));
      if ( filter == null )
        // If Object does not exist,
        // create one and place it in the node cache
        {
          filter = new DefendObj();
          n.softstate.put(DEFEND_KEY, filter, DURATION);
        }
      // Add host to the list of filtering hosts
      b = filter.addfHost(host);
      // b = true if host successfully added
      // b = false otherwise
    }
  else
    // Capsule at intermediate node; forward capsule
    b = n.routefornode(this, dstadd);
    // b = true if capsule successfully forwarded
    // b = false otherwise
  return b;
}
```

Figure 4-3: `DefendCap` Program

4.4.2 Filtering Algorithm

The DefendObj uses a simple algorithm to determine whether a capsule has a spoofed source address [see Figure 4-4]. First, it checks to see if the TCP SYN capsule is destined for a host that has requested filtering. If not, the capsule is forwarded. If so, it retrieves the capsule's source address from the header field. Using the capsule's source address as the destination address, it queries the routing table for the "next hop" address. If a "next hop" address does not exist, the query will result in a null return value, meaning that the capsule's source address does not exist in the network. The DefendObj recognizes this and discards the capsule. If a "next hop" address is returned, this address is compared to the capsule's prevNode field. By symmetric routing, we know that if a capsule travels from M to N on its route from source A to destination B, it must travel from N to M on its return trip. If the capsule's source address is not forged, then the "next hop" address returned by the query should match the prevNode field in the capsules header field. If the addresses do not match, the source address is probably forged, and the capsule is dropped. If they do match, the capsule passes the filter and is forwarded to the next node on its path to the destination.

```
private boolean filter(TCPCapsule c, Node n)
     /* This is the filter mechanism that when given a TCPCapsule
      * and a Node, will return true if the capsule passes the
      * filter and false otherwise.
      */
{
  // Using the capsule's source address as the destination, query
  // for the next hop address
  RouteEntry r = n.routes.get(c.srcadd);
  if ( r == null || r.nxt != c.getPrevNode() )
    // either nextHop does NOT exist when heading back to the
    // source node or nextHop exists but is NOT the prevNode.
    // This means the capsules source address may be forged;
    // drop capsule
    return false;
  return true;
}
```

Figure 4-4: filter method

4.4.3 Push-back

In addition to filtering out forged capsules, the DefendObj also has a push-back mechanism that moves the filter closer to the adversary one node at a time. Whenever a capsule calls the checkCap method, the capsule's prevNode field is retrieved. The DefendObj then checks to see if it has already sent this node a

DefendCap. If not, it creates a DefendCap on behalf of the server and sends it off to the capsule's previous node, thereby, creating a DefendObj at that node. It keeps track of all the nodes to which it has sent a DefendCap in a table. If it has already sent a DefendCap to the node, it does nothing in terms of push-back. In this manner, the filtering object is pushed back into the network as close to the adversary as possible. The code for the push-back algorithm is shown in Figure 4-5.

```
private void pushback(TCPCapsule c, Node n, boolean toFilter)
     /* This method pushes the Defend Object closer to the
      * adversary within the network using the DefendCap.
      * This is one of the private methods of the Defend Object.
      */
{
  Address host = new Address(c.dstadd);
  Address prevNode = new Address(c.getPrevNode());

  // Keep track of nodes to which Defend Capsules have been sent
  prevfHosts.addElement(prevNode);
  DefendCap dc = new DefendCap(c.dstadd, c.srcprt, c.dstprt,
                               n.address, c.getPrevNode(),
                               toFilter, null);
  n.send(dc);
}
```

Figure 4-5: pushback method

4.5 The FloodApp

The attack application, FloodApp, is a very simple implementation of the TCP SYN Flooding attack. When activated, the application will continuously generate random numbers to serve as source addresses. As each address is generated, the application sends a SYN capsule to the system it is trying to flood. The victim host can be specified by the user of the attack application. In this particular implementation, the source addresses are merely generated at random. We do not make any restrictions in terms of what the spoofed addresses should be. This is, however, sufficient for our purposes. Recall that it is only necessary to spoof addresses that either do not exist or cannot respond with an RST. Since our network is small compared to the number of addresses available using IPv4 addressing, it is improbable that a randomly generated address belongs to a "real" host in our network; even if it does, a few RST's sent by "real" hosts will not be enough to thwart an attack.

4.6 Assumptions

In order for our filtering algorithm to work, there are a few assumptions we have made regarding our network environment.

- We trust the active nodes to properly execute the capsule routines. We assume that an adversary has control over specific hosts, but not over any active nodes. This is analogous to today's Internet. We trust routers in the Internet to route packets properly.
- We assume that an adversary can forge header information of capsules, but cannot forge a capsule program. This means that an adversary cannot create a `TCPCapsule` that has the same MethodID as our `TCPCapsule`, but has a different `evaluate` method. Although the current implementation of ANTS has no safeguards against spoofing capsule programs, it can be achieved if the MethodID was a server hash of the capsule code. Investigation is currently underway to build such a mechanism, which will be in place before a commercial active network is deployed.
- We assume that the network uses symmetric routing. Although there are protocols, such as Network Time Protocol (NTP) [11], a clock synchronization protocol, and IP Multicast (MBONE) [14], that depend on symmetric routing [5], asymmetric routing is more prevalent in today's networks due to its flexibility. A possible extension to this thesis is to build a more versatile filter that works in an asymmetric routing environment as well.

Chapter 5

Conclusion

Since the inception of the TCP SYN-Flooding attack in 1996, this problem has plagued the Internet community. As yet, there is no solution in today's network environment. On the other hand, we have successfully built a dynamic filter in an active network, ANTS, that filters out spoofed capsules, thereby stopping the TCP SYN-Flooding and other similar denial-of-service attacks.

5.1 Evaluation

Our goal in this research was to find a solution to the TCP SYN-Flooding attack that meets the criteria presented in Section 1.2. We feel we have met our goal with some success.

- Stops the attack: Unlike many of the fixes presented in Chapter 2, our dynamic filter effectively filters out forged packets within an active network. The only weakness in our filter is that it is dependent on symmetric routing. More work needs to be done in order to make our solution work in a more general network environment.

- User transparency: Our solution does have this property to some extent. Regular users will not need to do anything different when filtering is occurring. They can continue to send and receive TCP Capsules just as if the network was not filtering. The protocol and capsules are not changed, and everything functions exactly as before, except that forged capsules will be dropped.

- Easy deployment: Since active networks were designed for easy, dynamic deployment of network applications, our solution definitely has this property.

Deployment is on a per server basis, independent of other hosts within a network. Deployment can be automatic or manual, but does not require a cooperation from the entire network community. This is a big improvement over today's networking environment.

- Minimal system impact: Since most of the work is done within the network, the performance on end systems is not affected by our solution. Some SYN packets may reach the server during an attack, but this is necessary in order to activate the filter. These few half-open connections will affect the server in that they do take up space in the server's stack, but we believe this is minimal [see Section 5.3].

- Minimal network impact: It would seem that if most of the work is done within the network, network performance would be affected dramatically. Our design of the system, however, precludes this. We only deploy the filter object during an attack. Therefore, the network is not affected when everything is running normally, which is most of the time. Our filter is designed to be lightweight, so even during an attack, it does not take up a great amount of system resources to filter. Our selective filtering scheme makes the filter transparent to other applications and protocols running on the same network. The only capsules affected are the TCP SYN capsules. Hence, we believe the amortized cost of filtering is low [see Section 5.3].

5.2 Issues

During this research, there were a few difficult problems that we faced. When designing the filter, one of the main questions was where to put the filter to be most effective in filtering out forged capsules. The key, of course, is to place the filter as close to the adversary as possible so that forged capsules are dropped as soon as possible. This is so that the rest of the network will not suffer the effects of an adversary flooding the network. The optimal node is what we call an "edge" node, because it lies on the outer edge of the network, connecting individual hosts to the rest of the network. In today's network, a firewall or a gateway would be an edge node. The problem lies in identifying these "edge" nodes in an active network. A domain firewall or gateway is very different from a router within the Internet. In an active network, however, each active node is virtually indistinguishable from other active nodes.

To circumvent this problem, we use a push-back mechanism to continuously move towards the adversary, but this leads to yet another problem. Continuously pushing back causes us to overshoot and "notify" the adversary that we are filtering packets. We send a `DefendCap` to the immediate predecessor node whenever a capsule arrives and the server is filtering. Eventually, however, the immediate predecessor will be the adversary host itself. If we send a `DefendCap` to the adversary, he will know that countermeasures have been taken and may employ other methods of attack. Perhaps if there were distinctions between hosts and active nodes, we could avoid this problem, but our active network makes no distinction between a host and an active node.

Another interesting problem is determining a filtering algorithm. Ingress Filtering is the leading proposal in today's network environment and can also be implemented in active networks; however, there is the problem of making all domain administrators implement Ingress Filtering. To bypass this problem in an active network, we could implement Dynamic Ingress Filtering, but this is easier said than done. Ingress Filtering in today's network is usually implemented by domain administrators who know their subnets. In our model, filtering is initiated by a server. To implement Dynamic Ingress Filtering, we would need to be able to determine dynamically what subnet a node is responsible for. This information is not readily available in our active network, nor is it readily available in the Internet today. In addition, routing in our active network is not based on hierarchy as it is in today's networks. For example, it is easy to determine that a host with an IP address of 18.X.X.X is within the `mit.edu` domain because it is a Class A address and 18 is assigned to the MIT. Within MIT, different subnets correspond to different parts of the Institute, and routing to individual hosts goes through this hierarchy. In our active network, this hierarchy does not exist. Any host can have any IP address and so routing does not depend on hierarchy. This makes implementation of Dynamic Ingress Filtering difficult.

5.3 Future Work

As stated in Section 5.1, we have met our goal only with partial success. To further evaluate the effectiveness of our solution, we need to take performance measures on our filter in an actual or simulated active network environment. Future investigation would be needed to determine, quantitatively, the impact our dynamic filter has on system and network performance.

The filter we have built only works in a network environment where routing is symmetric; however, in today's Internet, as well as many subnets, routing is asymmetric and, usually, unpredictable [16]. Under these circumstances, our filter would fail. Another possible extension to this thesis would be to build a filter that works in a more general network environment.

Another question worth investigating is how robust is our solution? Availability is difficult to maintain because it is a *backwards* goal. The idea is to make sure that people who have legitimate access do. This is different from maintaining secrecy and integrity where the idea is to keep those who do not have legitimate access out. It is difficult to test all possible cases. To make our filter more robust, we need to analyze the filter for any flaws in design and/or implementation that makes it vulnerable to attacks.

As noted in previous chapters, Ingress Filtering is one of the most effective countermeasures against address spoofing attacks. The difficulty in implementing such a mechanism has been described, but it by no means precludes Dynamic Ingress Filtering from ever being implemented. If our active network was modeled more like today's Internet, e.g., with hierarchical addressing and routing, we may be able to extract domain information dynamically from the routing tables and perform Ingress Filtering. We believe that further investigation may lead to an answer.

But beyond filtering, the active network may prove to be even more useful than just stopping an attack. Stopping an attack is analogous to healing a symptom of a disease. The symptom no longer exists, but it does not mean that the illness is gone. In our case, we are able to stop an attack, but the adversary remains at large. A more effective solution to our problem is to track down our adversary by tracing packets. In today's network architecture, it is extremely difficult (though not impossible) to trace a packet back to its originator. This is because routers forward packets towards the destination; there is no concept of going "backwards" to the sender except through the use of the IP header which, of course, can be forged. In an active network, the same is true, but perhaps with the increased processing power of the network, a tracer can be implemented.

5.4 Summary

The security concern of maintaining availability in computer systems has long been a neglected area of research. With the rise of denial-of-service attacks, such as the TCP SYN-Flooding attack, more attention is drawn towards this aspect of security. Current network technology offers little help in defending against such attacks. Although several fixes have been proposed, no solution has emerged. We propose to solve this problem in a new context, active networks. We have shown that we can thwart the TCP SYN-Flooding attack and other such address spoofing attacks using active network technology. Our solution is a lightweight, flexible, dynamic filter, which is capable of filtering out spoofed capsules within the network without significant impact on system or network performance. Our solution works only in a network that uses symmetric routing. Nonetheless, we feel our work provide many insights into solving the more general problem of address spoofing. It also shows that active networks are powerful tools that can be used in the future to build dynamic defenses against security attacks.

References

[1] Berkeley Software Design, Inc. (BSDI) "BSDI Releases Defense For Internet Denial-of-Service Attacks." http://www.bsdi.com/press/19961002

[2] CERT Advisory CA-95:01. "IP Spoofing Attacks and Hijacked Terminal Connections." Issue date: January 23, 1995. Revision date: September 24, 1996. ftp://info.cert.org/pub/cert_advisories/CA-95%3A01.IP.spoofing

[3] CERT Advisory CA-96.21. "TCP SYN Flooding and IP Spoofing Attacks." Issue date: September 19, 1996. Revision date: December 10, 1996. ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding

[4] Check Point Software Technologies, Ltd. "TCP SYN Flooding Attack and the FireWall-1 SYNDefender." October, 1996. http://www.checkpoint.com/fw21/syndefender/syndefender-white.html

[5] E. Chen. "Symmetric Routing in a Multi-Provider Internet." *North American Network Operators' Group May 1995 Meeting Notes*. Academ Consulting Services. http://www.academ.com/nanog/may1995/symmetric.html

[6] Cisco Systems. "Defining Strategies to Protect Against TCP SYN Denial of Service Attacks." http://www.cisco.com/warp/public/707/4.html

[7] D. E. Comer. *Internetworking with TCP/ IP*. Prentice Hall Inc., Englewood Cliffs, NJ, 1995.

[8] daemon9, route, and infinity. "Project Neptune." *Phrack Magazine*. Vol. 7, Iss. 48, File 13 of 18. July 1996 Guild Productions. http://www.fc.net:80/phrack/files/p48/p48-13.html

[9] V. D. Gligor. "On Denial of Service in Computer Networks." *Proc. IEEE Int'l. Conf. on Data Engineering*, pp. 608-617, Feb. 1986.

[10] J. Gosling and H. McGilton. "The Java Language Environment: A White Paper." http://java.sun.com/doc/language_environment/ May 1996, Sun Microsystems, Inc.

[11] ICAST Corporation. "The MBONE Information Web." http://www.mbone.com/techinfo/

[12] K. James. "Game 3 chess match a draw; IBM site up." *USA Today*, February 14, 1996. Sec. A, p.1.

[13] J. Markoff. "A New Method of Internet Sabotage is Spreading." *New York Times*, September 16, 1996.

[14] D. L. Mills. "The Network Time Protocol (NTP)." http://www.eecis.udel.edu/ ntp/

[15] R. M. Needham. "Denial of Service: An Example." *Comm. ACM*, Vol. 37, No. 11, pp. 43-46, Nov. 1994.

[16] V. Paxson. "End-to-End Routing Behavior in the Internet." *ACM SIGCOMM '96*, Stanford University, USA. August 1996. http://www.acm.org/sigcomm/sigcomm96/papers/paxson.html

[17] C. P. Pfleeger. *Security in Computing*. Prentice Hall Inc., Englewood Cliffs, NJ, 1989.

[18] J. Santos, V. Tardif, and D. Weber. "Foiling the TCP SYN Attack." December 10, 1996. MIT Computer Security Course (6.857). Will appear in http://web.mit.edu/course/6/6.857/www

[19] J. Shiver. "Outage at AOL Pulls the Plug on 6 Million Subscribers." *Los Angeles Times,* August 8, 1996. Sec. D, p.1.

[20] W. R. Stevens. *TCP/ IP Illustrated, Volume 1 – The Protocols*. Addison-Wesley Publishing Co., Reading, MA, 1994.

[21] D. L. Tennenhouse and D. J. Wetherall. "Towards an Active Network Architecture." *Computer Communication Review,* Vol. 26, No. 2, April 1996.

[22] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. "A Survey of Active Network Research." *IEEE Communications Magazine,* Vol. 35, No. 1, pp80-86. January 1997.

[23] D. J. Wetherall. "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols'." Submitted to *ACM SIGCOMM'97*, Cannes, France. September 1997.

[24] C. Yu and V. D. Gligor. "A Specification and Verification Method for Preventing Denial of Service." *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, pp. 581-592, June 1990.