

Unit II

What is a Firewall?

A Firewall is a network security device that monitors and filters incoming and outgoing network traffic based on an organization's previously established security policies. At its most basic, a firewall is essentially the barrier that sits between a private internal network and the public Internet. A firewall's main purpose is to allow non-threatening traffic in and to keep dangerous traffic out. A firewall is a network security device that monitors incoming and outgoing network traffic and decides whether to allow or block specific traffic based on a defined set of security rules.

Types of Firewalls

A firewall can either be software or hardware. Software firewalls are programs installed on each computer, and they regulate network traffic through applications and port numbers. Meanwhile, hardware firewalls are the equipment established between the gateway and your network.

Additionally, you call a firewall delivered by a cloud solution as a cloud firewall.

There are multiple types of firewalls based on their traffic filtering methods, structure, and functionality.

A firewall welcomes only those incoming traffic that has been configured to accept. It distinguishes between good and malicious traffic and either allows or blocks specific data packets on pre-established security rules. These rules are based on several aspects indicated by the packet data, like their source, destination, content, and so on. They block traffic coming from suspicious sources to prevent cyberattacks. Firewalls are designed with modern security techniques that are used in a wide range of applications. In the early days of the internet, networks needed to be built with new security techniques, especially in the client-server model, a central architecture of modern computing. That's where firewalls have started to build the security for networks with varying complexities. Firewalls are known to inspect traffic and mitigate threats to the devices. Firewalls are appliances that protect networks against external intrusion by screening incoming data and admitting or excluding traffic. Packet filtering firewalls achieve this goal by applying security rules to data packets.

A packet filtering firewall

A packet filtering firewall is a network security device that filters incoming and outgoing network packets based on a predefined set of rules. **Packet filtering relies on the IP packet header information and information about the firewall appliance, such as the following:**

- Source IP address.Destination IP address.Source port.Destination port.Network protocol.
- IP flags.Firewall interface.Direction, ingress or egress.

Rules are typically based on IP addresses, port numbers, and protocols. By inspecting packet headers, the firewall decides if it matches an allowed rule; if not, it blocks the packet. The process helps protect networks and manage traffic, but it does not inspect packet contents for potential threats.

This type of firewall operates at a fundamental level by applying a set of predetermined rules to each network packet that attempts to enter or leave the network. These rules are defined by the network administrator and are critical in maintaining the integrity and security of the network.
--

Packet filtering firewalls use two main components within each data packet to determine their legitimacy: the header and the payload.

The packet header includes the source and destination IP address, revealing the packet's origin and intended endpoint. Protocols such as TCP, UDP, and ICMP define rules of engagement for the packet's journey. Additionally, the firewall examines source and destination port numbers, which are similar to doors through which the data travels.
--

Certain flags within the TCP header, like a connection request signal, are also inspected. The direction of the traffic (incoming or outgoing) and the specific network interface (NIC) the data is traversing, are factored into the firewall's decision making process.

Packet filtering firewalls can be configured to manage both inbound and outbound traffic, providing a bidirectional security mechanism. This ensures unauthorized access is prevented from external

sources attempting to access the internal network, and internal threats trying to communicate outwards. A primary packet filtering firewall use case is the prevention of IP spoofing attacks, where the firewall examines the source IP addresses of incoming packets. By ensuring the packets originate from expected and trustworthy sources, the firewall can prevent attackers from masquerading as legitimate entities within the network. This is particularly important for perimeter defences. In addition to security, packet filtering firewalls are used to manage and streamline network traffic flow. By setting up rules that reflect network policies, these firewalls can limit traffic between different subnets within the enterprise. Limiting traffic between different subnets helps contain potential breaches and segment network resources according to departmental needs or sensitivity levels.

A **firewall** works as a network gateway and provides a protective wall over a network. Access through a firewall requires an authorization to then access the network. The initial process for most firewalls involves installing a software to a user computer or over a network which initiates an access control policy on that network. Some firewalls are primarily for prevention of the flow of traffic, while some are primarily for easing the flow of traffic. Firewalls are of great importance in our present digital world, due to the surge of cyber-security concerns.

What can a firewall do?

A firewall can protect a network from unauthorized interactive access from the outside world. A network is guarded from strangers and unauthorized users.

A firewall can protect a network from external traffic, but can let authorized users have access with unhindered communication. In hardware terms, a firewall can protect the network against network-borne attacks when unplugged.

A firewall can protect users from an attacker who tries to connect a new application to the device while in use on the internet. It queries the user's consent on this particular scenario. It also protects a user from an attacker trying to connect to an application that has not been captured on the filter rule stored on the firewall.

What a firewall protects a user from

Default setting

Some user systems come with certain pre-installed applications which can be accessed remotely. These applications could be seen as factory set applications. The remote access possibility of these applications creates room for intrusion of malicious codes into the application, which could cause harm on the user side. A firewall can be installed for applications to go through packet filtering each time data packets arrive remotely into the user systems.

Cyber attacks

Cyber-security attacks like *denial of service (DOS)* are very popular in current times. Most strangers can make a network server inaccessible by flooding the server with multiple access requests. This would reduce the server performance and could cause a breakdown. But when a firewall is installed, such attacks could be detected and hindered so as to stop a breakdown of the server.

Malicious spam

Firewalls can prevent users from falling prey to spam links, mail, and messages. The spam messages create an open door for hackers to steal important information from users. Firewalls sieve this spam mail and blocks them so that users will not fall prey to them.

Viruses

Viruses could be very dangerous to a user, as they tend to compromise the user data and storage. Most times, the viruses keep increasing until the user is locked out of the system. Firewalls protect against viruses, and using anti-virus alongside firewalls can protect a computer totally.

Macros

Macros exist, like scripts, which help applications make complicated processes simple. A hacker can possibly inject their own macro to run in their own preference, so as to gain control of user applications. This can be observed and stopped by a firewall if put in place to defend the system.

Packet Filter vs. Firewall

Throughout this book, the terms *firewall* and *packet filter* are used rather interchangeably. Firewalls and packet filters generally perform the same function. Packet filters inspect traffic based on

characteristics such as protocol, source or destination addresses, and other fields in the TCP/IP (or other protocol) packet header. Firewalls are packet filters, but application layer firewalls may examine more than just packet headers; they may examine packet data (or payloads) as well. For example, a packet filter may monitor connections to ports 20 and 21 (FTP ports), whereas a firewall may be able to establish criteria based on the FTP port numbers as well as FTP payloads, such as the **PORT** command or filenames that include the text *passwd*. A web application firewall (WAF) watches incoming connections for tell-tale signs of SQL injection attacks and outbound traffic for sensitive information being leaked from the web app.

Normally, the term *packet filter* refers to software that makes decisions based on protocol attributes: addresses, ports, and flags. Packet filtering provides coarse (but effective) security to a network routing device. However, the software is simplistic because the access control is limited to a handful of protocols like TCP/IP, UDP, and ICMP. The term *firewall* is usually reserved for software or devices whose primary purpose is to apply security decisions to network traffic.

Sometimes you may also hear the phrase *intrusion-prevention system (IPS)*. This usually refers to hardware and software that combines packet filtering, content filtering, intrusion-detection system (IDS) capabilities, and other security functions. For example, alerts from an IDS would automatically trigger certain firewall rules. Before you resort to trying to tackle a commercial IPS, determine whether using a firewall, keeping your systems fully patched on a regular basis, and perhaps using an IDS such as Snort (covered later in this chapter) provides sufficient protection for your system. If you find that you need extra security measures, then look into a commercial IPS.

How a Firewall Protects a Network

Firewalls are only as effective as the rules they're configured to enforce. As mentioned previously, firewalls examine particular characteristics of network traffic and decide which traffic to allow and deny based on some criteria. It is the administrator's job to define rules so that the firewall sufficiently protects the networks—and information—behind it without negatively impacting legitimate traffic. Most firewalls have three ways to enforce a rule for network traffic:

- *Accept* the packet and pass it on to its intended destination.
- *Deny* the packet and indicate the denial with an Internet Control Message Protocol (ICMP) message or similar acknowledgment to the sender. This provides explicit feedback that such traffic is not permitted through the firewall.
- *Drop* the packet without any acknowledgment. This ends the packet's life on the network. No information is sent to the packet's sender. This method minimizes the sender's ability to deduce information about the protected network, but it may also adversely impact network performance for certain types of traffic. For example, a client may repeatedly attempt to connect to a service because it hasn't received an explicit message that the service isn't available.

Most firewalls drop packets as their default policy for traffic that isn't permitted. When building a ruleset, start with the concept of *least privilege* or *deny all*. It's safer to start with a firewall that rejects every incoming connection and open only the necessary holes for services you want to expose, rather than to start with an open firewall that exposes all of your network's resources.

Packet Characteristics to Filter

Most firewalls and packet filters have the ability to examine the following characteristics of network traffic:

- Type of protocol (IP, TCP, UDP, ICMP, IPSec, etc.)
- Source IP address and port
- Destination IP address and port
- ICMP message type and code
- TCP flags (ACK, FIN, SYN, etc.)
- Network interface on which the packet arrives

For example, if you wanted to block incoming ping packets (ICMP echo requests) to your home network of 192.168.1.0/24, you could write something like the following rule. (Don't worry about the specific syntax yet—we'll get to that shortly.) The important components of the rule are the action (deny), the packet attributes (ICMP protocol, specifically "ping" types), the direction of the rule (packets "from" one source "to" another), and the type of source (a network address range like 192.168.1.0/24).

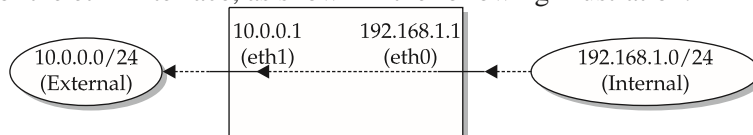
deny proto icmp type 8:0 from any to 192.168.1.0/24

Or if you wanted to allow incoming web traffic to 192.168.1.50 but deny everything else, you would create two rules. The first one would specify the direction of web traffic to a specific TCP port on a specific host. The second one would make sure all other traffic is denied. Those rules would look like the following:

**allow proto tcp from any:any to 192.168.1.50:80 deny proto
all from any to 192.168.1.0/24**

CAUTION You understand the order in which your firewall interprets rules. One firewall may use a "first match" approach that permits (or denies) a packet as soon as it encounters a rule. A "last match" firewall may traverse every rule and apply the final, most specific match to a packet. Consider how a rule like **allow any any** (unrestricted packet flow) might behave in these two scenarios.

You can also use a firewall to protect your network from IP spoofing. For example, imagine your firewall's external interface (called eth1) has an IP address of 10.0.0.1 with a netmask of 255.255.255.0. Your firewall's internal interface (called eth0) has an IP address of 192.168.1.1 with a netmask of 255.255.255.0. Any traffic from the 192.168.1.0 network destined to the 10.0.0.0 network will come *in* to the eth0 interface and go *out* of the eth1 interface, as shown in the following illustration.



Typically, traffic from the 10.0.0.0/24 network destined for the 192.168.1.0/24 network will come *in* to the eth1 interface and go *out* of the eth0 interface. Therefore, you should never see traffic with a source address in the 192.168.1.0/24 range coming *inbound* on the eth1 interface.

If you do, it means someone on the external 10.0.0.0/24 network is attempting to spoof an address in your local IP range. Your firewall can stop this kind of activity by using a rule like the following:

deny proto any from 192.168.1.0/24 to any on eth1

This rule may seem ambiguous. Might it match legitimate traffic coming from the 192.168.1.0/24 network heading out to the external network? It could, but it depends on the firewall's interpretation of the syntax. Since we're using a fictional firewall rule syntax for these examples, this rule remains ambiguous and possibly ineffective. This illustrates an important lesson: You have to be very careful when writing firewall rules. Simply knowing what you are trying to block isn't sufficient; you must verify that the rule works as expected.

For example, a ruleset might be interpreted in a linear manner, a ladder of sorts in which a packet moves from one rule to the next in order until it is accepted or denied.

Another firewall might merge rules into a set of overlapping controls in the manner of a Venn diagram. You have to make sure that you understand how the firewall applies rules and what its default or assumed behavior might be. In the antispoofing example, we rewrite the rule with less ambiguity by specifying the network interface on which it should be applied:

deny proto any from 192.168.1.0/24 to any in on eth1 allow proto any from 192.168.1.0/24 to any out on eth1

The combination of these two rules clearly indicates our intention. We'll talk more about writing good firewall rules later in this chapter.

Stateless vs. Stateful Firewalls

Chapter 9 showed you how tools such as Nmap can be used to determine whether a firewall is stateful or not. We'll review those concepts here. A *stateless* firewall examines individual packets in isolation from each other; it doesn't track whether related packets have arrived before or are coming after. A *stateful* firewall places that packet in the context of related traffic and within a particular protocol, such as TCP/IP or FTP. This enables stateful firewalls to group individual packets together into concepts like connections, sessions, or conversations. Consequently, a stateful firewall is able to filter traffic based not only on a packet's characteristics, but also on the context of a packet according to a session or conversation. For example, a TCP ACK packet will be denied if the protected service hasn't set up the SYN and SYN-ACK handshake to establish a connection.

Stateful firewalls also allow for more dynamic rulesets. For example, suppose a system on the internal 192.168.1.0/24 network wanted to connect to a web server on the Internet. The following steps demonstrate the drawbacks of trying to apply simple packet inspection to the traffic. The first step establishes a rule for TCP traffic from the 192.168.1.0/24 network to port 80 on any IP address. **allow proto tcp from 192.168.1.0/24: any to any:80 out on eth1**

So far, so good. But what happens when the web server responds? We need to make sure the response packet gets accepted by our firewall. Unfortunately, since the web browser's system chooses a port at random to receive traffic, we won't know which destination port to open for the response until after the connection starts. The only thing we know for certain is that the web server's response packet will have a source port of 80. Consequently, we might try a rule that allows any web traffic (e.g., TCP port 80) from the Internet to reach our internal network: **allow proto tcp from any:80 to 192.168.1.0/24: any in on eth1**

This allows the web server's response to reach any host on the internal network at the expense of opening a gaping hole in the firewall. The rule assumes that only return web traffic would be using a source port of 80. However, as we have seen in other chapters, TCP services and connections use specific port numbers by common agreement, not by technical restrictions.

If a hacker were aware that any packet with a source port of 80 could pass through the firewall, they could use port redirection to set up a tunnel (see Chapter 8) to do something as simple as scan for ports or as (only slightly less) simple as tunnel traffic for a remote shell. The tunnel would forward any traffic it received to a machine on the 192.168.1.0 network, substituting 80 for the packet's source port in order to traverse the firewall rule. For a stateless firewall, a rather weak protection against this scenario is to restrict incoming traffic to the ephemeral ports used by TCP clients, as follows: **allow proto tcp from any:80 to 192.168.1.0/24:1024-65535 in on eth1**

The operating system's network stack chooses a random port as the source of its traffic, whereas the destination port for something like an HTTP service is 80 by default. This rule improves on the stateless protection, but it still leaves a large, unnecessary hole in the firewall.

Wouldn't it be better if the firewall could instead remember the details of our outgoing connection? That way, we could say that if the initial outgoing packet is allowed by the firewall, any other packets that are part of that session should also be allowed. This dynamic rule prevents us from having to poke potentially exploitable holes in our firewall. This is the advantage of stateful firewalls. Some of this concept was demonstrated in Chapter 10 in the review of the hping tool.

Network Address Translation (NAT) and Port Forwarding

Networking devices, whether a consumer-level wireless access point or an enterprise-grade firewall, are the gateways between networks. They separate external networks like the Internet from private networks like those used by the systems in your home. Systems on the Internet must have unique, public (i.e.,

“routable”) IP addresses. This ensures that packets for a web site or a gaming server always go to the right destination. If the same public IP address were permitted to be used for different, unrelated servers, then traffic control would be a nightmare of congestion and security problems.

NOTE

This chapter focuses on IPv4, which remains the predominant IP protocol on the Internet despite efforts over the past decade to move large networks onto IPv6. A major difference with IPv6 is that the address space is so large that there is no need for an equivalent RFC 1918 address space—we’ll run out of humans, devices, and planets before we exhaust the IPv6 address space. Regardless of protocol differences, the fundamental concepts of firewalls and monitoring remain similar enough to avoid the need for protocol nuances in this chapter.

Internal networks, on the other hand, use “nonroutable” IP addresses, referred to as *private* or *RFC 1918* addresses. RFC 1918 refers to the document that explicitly defines the address space of the following networks:

- 192.168.0.0 through 192.168.255.255 (written 192.168.0.0/16 or 192.168.0.0/255.255.0.0)
- 172.16.0.0 through 172.31.255.255 (written 172.16.0.0/12 or 172.16.0.0/255.240.0.0)
- 10.0.0.0 through 10.255.255.255 (written 10.0.0.0/8 or 10.0.0.0/255.0.0.0)

The Internet Assigned Numbers Authority (IANA) reserved those IP address blocks for private networks. This enables organizations large and small to build networks whose traffic will not leak onto the Internet unless it passes through a gateway device like a router or firewall. Internet traffic should never accommodate packets whose source contains an RFC 1918 address. It also means that organizations are free to use addresses within these networks without worrying about whether other networks are using the same IP addresses. (That is, until they start trying to tie networks together with VPNs or similar links—but those are network design problems for a different book.)

TIP

RFC 5737 defines network address ranges for use in documentation. These are guaranteed to be “empty” and nonroutable more so than the RFC 1918 ranges. Should you write about networking but wish to avoid using the overly familiar private IP ranges in examples, consider the networks 192.0.2.0/24 (TEST-NET-1), 198.51.100.0/24 (TEST-NET-2), or 203.0.113.0/24 (TEST-NET-3).

The ability for organizations to independently use the same private network addresses reduces the risk of running out of unique addresses for the millions and millions of devices on modern networks. This address scarcity problem will be solved when IPv6 is more universally adopted because IPv6 exponentially expands the available address space. (IPv4 supports about 4 billion devices theoretically due to its 32-bit address field, but much of that space cannot be used for practical addressing. IPv6 uses a 128-bit address field, enough for roughly 3.4×10^{38} unique devices. We’ll run out of funny cat videos and *Doctor Who* episodes long before we need to worry about running out of IPv6 addresses.) The “nonroutable” nature of private address spaces poses a problem once a device needs to access the Internet. The addresses are fine for syncing your stereo with your music collection stored on the local network, but they won’t work when your device with address 10.0.1.42 needs to retrieve music from storage on the Internet. The music storage service needs to know the difference between your device using the 10.0.1.42 address and someone else’s device using the same private IP address on their private network.

Network Address Translation (NAT) solves this routing problem by translating packets from private to public addresses. NAT is usually performed by a networking device on its external interface for the benefit of the systems on its internal interface. A NAT device allows machines on its private, internal network to masquerade as the IP address assigned to the NAT device. Private systems can communicate with the Internet using the routable, publicly accessible IP address on the NAT device’s external interface.

When a NAT device receives traffic from the private network destined for the external network (Internet), it records the packet’s source and destination details. The device then rewrites the packet’s header such that the private source IP address is replaced with the device’s external, public IP address.

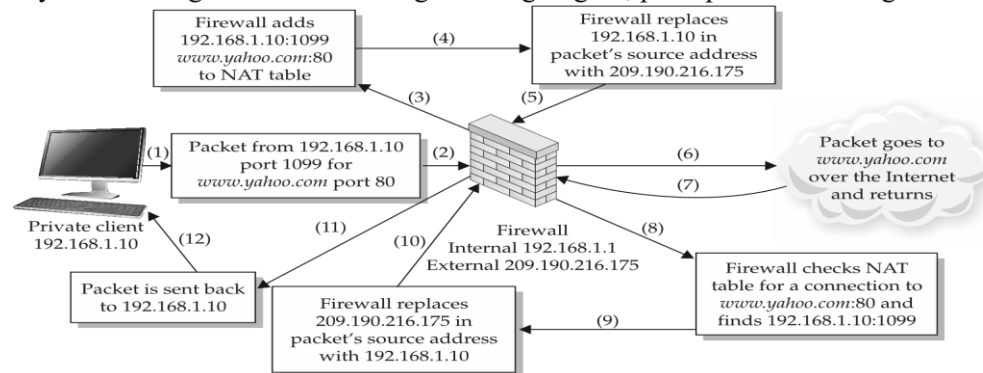
Then the device sends the packet to the destination IP address. From the destination system's point of view, the packet appears to have come directly from the NAT device. The destination system responds as necessary to the packet, sending it back to the NAT device's IP address.

When the NAT device receives the response packet, it checks its address translation table to see if the address and port information of the packet match any of the packets that had been sent out. If no match is found, the packet is dropped or handled according to any firewall rules operating on the device. If a match is found, the NAT device rewrites the packet's destination IP address with the private IP address of the system that originally sent the packet.

Finally, the NAT device sends the packet to its internal destination. The network address translation is completely transparent to the systems on the internal, private IP address and the Internet destination.

The private system can access the Internet, but an Internet system cannot directly address it.

If you're having trouble visualizing what's going on, perhaps the following illustration will help:



NAT has a few limitations with regard to the kinds of traffic it may successfully translate. The packet header manipulation will interfere with any protocol that requires the use of true IP addresses, such as IPSec. Also, any protocols that require a separate, reverse incoming connection, such as active mode FTP, will not work. The outgoing FTP control connection to the FTP server will make it through the NAT device just fine, but when the FTP server attempts to establish the data connection, the NAT device won't know what to do because it doesn't have a corresponding entry in its translation table. NAT's prevalence has influenced people to create workarounds to resolve these limitations.

In the end, NAT has become integral to firewalls and network security. It provides an added layer of security to a firewall appliance, as it not only protects machines behind its internal interface, but also hides them. But what happens if you decide you'd like to expose a particular service on your private network to the Internet? What if you wanted someone across the country to be able to look at something you had posted on your internal web server?

For this, you can use a technique called **port forwarding** (just like the concepts covered in Chapter 8). The NAT device may forward traffic received on a particular port on the device's external interface to a port on a system on the private, internal network. A remote system on the Internet that connects to the NAT device on this port effectively connects to the port on the internal system and only needs to know the public IP address of the NAT device.

This is all well and good, but now you've made your private network a little less private by exposing the service listening on that port. Now anyone on the Internet can access your internal web server by connecting to the port on your NAT device. If your NAT device is a firewall, you can use firewall rules to limit which IP addresses are allowed to access it. While this is more secure, you're still relying solely on IP-based authentication. On many occasions, users who have built fortified, private networks may find it necessary to open up internal network resources to another remote facility.

There are many ways to restrict access from that remote facility and prohibit the rest of the Internet. But do we really want to forward dozens of ports and open dozens of holes in our firewall, or dozens of rules and exceptions? This is where Virtual Private Networks come into play.

The Basics of Virtual Private Networks

Virtual Private Networks (VPNs) are a complex subject in terms of identity, authentication, and encryption. We touch on them here because so many firewall and networking devices provide some

degree of VPN capability. In essence, a VPN establishes an encrypted channel between two networks (or single systems, or a combination thereof) that is overlaid on a public network. It's designed to mitigate the impact of using a hostile network like a public Wi-Fi connection where data may be sniffed or intercepted by an attacker. The VPN's encrypted traffic is meant to be opaque to anyone who tries to monitor or interfere with it. The VPN provides *confidentiality* and *integrity*.

A VPN server requires a remote client to authenticate to it before it will connect the remote client to the protected network. A VPN extends the boundaries of a network, which creates a mixed sense of security. On the one hand, the client now has a protected channel into another network. On the other hand, the network has a new ingress point and must trust that the client neither has malicious intent itself nor is compromised by an attacker and used as a relay to the network. Authentication at least creates a barrier to access and helps provide an audit log for access. As with any authentication point, it's important to use strong credentials (complex passwords or token-based solutions) to prevent brute-force guessing attacks from successfully compromising an account.

VPNs usually forward all traffic (or as much traffic as desired) between the networks over a single set of ports. Imagine how many port forwards and firewall rules you'd have to write if you had to open up several internal network resources to a remote location? File sharing, printer sharing, code repositories, web sites, and other services would create a NAT and port forwarding configuration nightmare.

By combining the capabilities of a firewall, a NAT device, and a VPN in one network device, you can greatly improve the external security of your internal network without losing convenience or productivity.

Linux System Firewall

All Linux distributions rely on the kernel's netfilter software to provide firewall capabilities (plus NAT and other network wrangling activities). Netfilter is part of the kernel. You can find the project's home at <http://netfilter.org>. The command-line interface for administering netfilter rules is the **iptables** command. The following example shows what may be the default rules for your system:

```
$ sudo iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
```

The **iptables** command fits in very well with the Linux philosophy of empowering the user to make their own decisions and give them complete control over their system. The drawback for novice users is that commands may be difficult to learn and clunky to use depending on the syntax they expect. Netfilter builds tables of rules based on *chains*. As you saw in the previous example, the **iptables** command lists the three default chains for netfilter: INPUT, FORWARD, and OUTPUT. These chains reflect the direction of traffic into or out of the network interface monitored by netfilter. The FORWARD chain is a special case for supporting NAT.

The following example shows how to change the default stance of the firewall from accepting all connections to limiting incoming connections to the SSH service on port 22. The key points to notice are the **-A** option that appends a rule to the INPUT chain and the **-j** option that tells netfilter which rule target to jump to. In this case, we tell netfilter to jump to accepting any traffic with a destination port of 22. This way it doesn't have to spend time checking the packet against other rules that might be redundant. The final rule in the INPUT chain rejects all traffic.

```
$ sudo iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
$ sudo iptables -A INPUT -j REJECT
```

A quick review of the man page might be all you need from here on out to set up simple rules to protect your laptop. Netfilter and iptables support many types of complex chains (rulesets) and

interactions. The complexity you need varies depending on whether you're protecting a laptop you use for browsing the web or protecting a web server you've deployed for the Internet to see. Good luck updating chains from a remote shell. More often than not you'll run into a conflicting rule (or typo...) that shuts down your remote connection and leaves the system in a state that rejects all incoming traffic. Should you mess up a chain and wish to restart from scratch, keep the following command handy. Remember to rebuild your rulesets after flushing them—otherwise you may leave the system unintentionally unprotected.

```
$ sudo iptables --flush INPUT
```

If you love Linux but find the iptables syntax frustrating, check out the Shorewall project at <http://shorewall.net>. It keeps you on the command line, but with a more user-friendly interface to managing rules.

Windows System Firewall

The Microsoft Windows operating system has evolved significantly over the past few decades. In its early days the system exposed important services and sensitive information (like remote Registry access) by default. Today, Microsoft has adapted the OS to a more hostile environment and has learned from many of its mistakes in the past. One important feature it has included is a system firewall. The interface for enabling and configuring the firewall is in the Control Panel, as shown in Figure 11-3. Note that it allows you to define different trust levels based on your network location. Since it's likely a laptop will roam among several different networks during its lifetime, or perhaps even daily, this interface helps you to determine when and what to share for a particular environment. Figure 11-4 shows the basic options available to you for these different locations. Follow the recommendations unless you have a compelling reason for disabling the firewall.

Snort: An Intrusion-Detection System

Firewalls block traffic that we know beforehand shouldn't be traversing a protected network. However, we have to let *some* traffic into the network, and, of course, traffic needs to go out. A competent administrator creates a robust ruleset to prevent malicious traffic from bypassing a firewall. A savvy administrator prepares for scenarios in which malicious traffic manages to bypass the firewall. This is where network monitoring comes in.

Snort (www.snort.org) is a network monitoring tool that watches traffic for signs of malicious activity (e.g., buffer overflows being executed against a service, command and control traffic from malware), suspicious activity (e.g., port scans and service enumeration), and anything else that you wish to look out for.

At its core, an intrusion-detection system (IDS) is a sniffer like tcpdump or Wireshark, but with specialized filters that attempt to identify malicious activity. A good IDS can find anything from a buffer overflow attack against an SSH server to the transmission of /etc/passwd files over FTP. Network administrators place these systems where they can best monitor traffic, such as a point where they can see all traffic through a firewall or see all traffic between network segments with different security contexts (e.g., production servers and developer systems). The IDS examines packets, looking for particular signatures or patterns that are associated with suspicious or prohibited activity. The IDS then reports on all traffic that matches those signatures.

Snort is a robust IDS that runs on Unix-based and Windows systems. It is also completely free. In this section, we focus on version 2.9.

Snort is one of the more complex tools covered in this book. In fact, entire books have been dedicated solely to Snort. We'll cover some of the basic concepts that make Snort a superior IDS. You can view the online documentation at www.snort.org/docs/ for full details on configuration and usage.

Snort Rules: An Overview

Snort rules are similar to the kind of packet-filter expressions that you create in tcpdump or Wireshark. They can match packets based on IP, ports, header data, flags, and packet contents. Snort has several types of rules that affect how it handles traffic:

- **Alert rules** Log packets whose characteristics match a predefined suspicious pattern (e.g., generated by a common hacking tool, or contain a string indicative of a buffer overflow or web attack) or custom rules that monitor packets you determine to be prohibited or undesirable on your network (e.g., file sharing, gaming, etc.).
- **Pass rules** Explicitly ignore packets. Traffic that matches these rules will not be logged.
- **Log rules** Record packets but do not generate rules. This would be useful for diagnosing network problems, storing traffic for audits, or monitoring sensitive systems so that traffic can be analyzed in case a compromise is detected.
- **Activate rules** Generate an alert for traffic that matches this rule's trigger, then activate a subsequent dynamic rule. (Until it is activated, a dynamic rule will not generate an alert even if traffic matches it.)
- **Dynamic rules** Triggered by activate rules. This enables you to chain rules together in a way that makes inspection more efficient (don't run rules needlessly) and more effective (create complex chains). These are great mechanisms for gathering more information during an attack.

Snort comes with a standard ruleset that checks for such activity as Nmap stealth scans, vulnerability exploits, attempted buffer overflows, anonymous FTP access, and much more.

By default, Snort checks the packet against alert rules first, followed by pass rules, and then log rules. This setup is perfect for the administrator who is just learning Snort and plans on using the default config file and ruleset. Snort's default ruleset doesn't include any pass rules or log rules. However, running Snort without performing any kind of customization or configuration is usually a bad idea, as you'll no doubt be inundated with false positives.

As you become more familiar with the Snort rule syntax, you'll be able to write rules to ignore certain traffic. For example, imagine a network that has been receiving a flood of DNS queries forwarded to its own DNS server from other DNS servers on the Internet. (In other words, the traffic was legitimate, just arriving in spikes.) It may happen that Snort would falsely alert on the traffic as UDP port scans and DNS probes. Obviously, it's not helpful to clutter Snort's logs with false positives. Consequently, we could create a custom rule to handle this specific scenario. To do so, you could start the rule with a variable definition called

Snort Rules Syntax

For details on the syntax of Snort rules, you should go to www.snort.org/snort-rules/.

This section provides a brief description of how rules are put together.

Basic Snort rules consist of two parts: the header and the options. The first part of the header tells Snort what type of rule it is (such as **alert**, **log**, **pass**). The rest of the header indicates the protocol (**ip**, **udp**, **icmp**, or **tcp**), a directional operator (either **->** to specify source to destination or **<>** to specify bidirectional), and the source and destination IP address and port. The source and destination IP address can be written using the syntax *aaa.bbb.ccc.ddd/yy*, where *yy* is the number of network bits in the netmask. This allows you to specify networks and single hosts in the same syntax (single hosts have a netmask of 32 bits). To specify several addresses, you can put them in brackets and separate them with commas, like this:

[192.168.1.0/24,192.168.2.4,192.168.2.10]

Port ranges can be specified using a colon (so that **:1024** means all ports up to 1024, **1024:** means 1024 and above, and **1024:6000** means ports 1024 to 6000).

Alternatively, you can use the keyword **any** to have all IP addresses and ports matched. You can also use the exclamation mark (!) to negate the IP or port (for example,

1:1024 and **!1025:** would be equivalent).

The rule options contain such things as the alert message for that rule and the packet contents that should be used to identify packets matching the rule. The options are always enclosed in parentheses and follow the syntax **keyword:value**, with each option pair separated by a semicolon (and optional whitespace before the **value**). Several keywords are available. Table 11-1 contains a sampling of the more important keywords taken directly from the documentation.

Scanning for Web Vulnerabilities

Only a few kinds of web servers drive the Web's traffic. Apache HTTP Server is the most recognizable in the open source category, while Microsoft's Internet Information Server (IIS) is the most recognizable commercial one. The nginx server, also open source, is a rising star for web administrators. All other web servers have mostly been left to the dustbin of web progress (for example, the previous edition of this book mentioned the iPlanet server, something most readers have probably never heard of nor encountered).

The web server is the most obvious component of a web application platform; something has to deliver pages to web browsers. But the platform may also comprise data stores, load balancers, and the programming framework used to write pages. There are even efforts such as Node.js (<http://nodejs.org/>) to take a client-side language like JavaScript onto the server.

It's a testament to the quality of web server development that very few high-impact vulnerabilities have been reported for Apache, IIS, and nginx over the past few years. However, this doesn't imply that these servers will remain secure or continue to be configured correctly. A vulnerability scanner contains a knowledge base of all vulns reported for different components of a web platform. It uses this knowledge to probe a target for indicators that one of the vulns is present. A web application must start out with a secure foundation.

You can use a web vulnerability scanner to test the basic security of a web application. Chapter 4 covers OpenVAS and Metasploit, which are scanners that check for the presence of known vulnerabilities in web sites in addition to vulns in network devices and operating systems. This section covers a web-specific scanner called Nikto. As you become more familiar with web security testing, you might try other open source tools like w3af (<http://w3af.org>).

Nikto

Nikto, by Chris Sullo and David Lodge, is a Perl-based scanner that searches for known vulnerabilities in common web applications, looks for the presence of common files that have the potential to leak information about an application or its platform, and probes a site for indicators of common misconfigurations. It is an outgrowth of the Whisker and LibWhisker tools created by Rain Forest Puppy, which were based on his influential work on web security as documented in *Phrack* Issue 55 from 1999 (www.phrack.org/issues.html?issue=55&id=7#article).

Use Nikto for assessing the security of a web application's deployment. The tool focuses on identifying vulns in commercial and open source web application frameworks. It won't be as helpful for assessing the security of a custom web application. For example, it may tell you that a site uses an outdated (and insecure) version of WordPress, but it won't be able to tell you if the blogging application you wrote from scratch is secure or not.

Implementation

Nikto is written in Perl, so it will run on any platform that Perl runs on. In practice, this means Nikto will run on Windows and any of the Unix-based operating systems. Clone the Git repository from <https://github.com/sullo/nikto.git>. You shouldn't need to install any Perl libraries that aren't already present in a default installation.

Scanning Nikto is uncomplicated, but not unsophisticated. Use the **-host** option to start scanning a single target for the presence of default files, pages that might expose sensitive information, or pages with known vulnerabilities. The following example shows Nikto's output when run against a blogging site running on the open source WordPress framework:

Nikto Components Nikto uses the `nikto.conf` file for settings that may be used less often (and don't have a command-line option) or that apply to every scan (and would be annoying to have to set on the command line every time). Review these settings to make sure they match values you desire. For example, a trivial server configuration might reject any request with the word "Nikto" in the User-Agent header—it wouldn't make the server any more secure, but it would frustrate naive script kiddies who don't understand the tools they run.

Nikto uses the files in the **database** subdirectory to determine what kinds of test it performs and how it categorizes responses from a server. The most important file is the `db_dictionary` file that contains a manifest of common directories found on web servers. These directories correspond to hierarchies from common web applications, design patterns that developers typically use (e.g., `/admin/`), and locations known to have files useful from a hacker's perspective. Add (or remove) any entries you wish to tune the time and comprehensiveness of a scan.

Running w3af

w3af has two user interfaces, the console user interface and the graphical user interface. This user guide will focus on the console user interface where it's easier to explain the framework's features. To fire up the console UI execute:

```
$ ./w3af_console ,w3af>>>
```

From this prompt you will be able to configure framework and plugin settings, launch scans and ultimately exploit a vulnerability. At this point you can start typing commands. The first command you have to learn is **help**

The main menu commands are explained in the help that is displayed above. The internals of every menu will be seen later in this document. As you already noticed, the **help** command can take a parameter, and if available, a detailed help for that command will be shown, e.g. **help keys**.

Other interesting things to notice about the console UI is the ability for tabbed completion (type 'plu' and then TAB) and the command history (after typing some commands, navigate the history with the up and down arrows).

To enter a configuration menu, you just have to type it's name and hit enter, you will see how the prompt changes and you are now in that context:

```
w3af>>> http-settings
```

```
w3af/config:http-settings>>>
```

All the configuration menus provide the following commands:

- **help**
- **view**
- **set**
- **back**

To summarize, the **view** command is used to list all configurable parameters, with their values and a description. The **set** command is used to change a value. Finally we can execute **back** or press CTRL+C to return to the previous menu. A detailed help for every configuration parameter can be obtained using **help parameter**

The **http-settings** and the **misc-settings** configuration menus are used to set system wide parameters that are used by the framework. All the parameters have defaults and in most cases you can leave them as they are. **w3af** was designed in a way that allows beginners to run it without having to learn a lot of its internals. It is also flexible enough to be tuned by experts that know what they want and need to change internal configuration parameters to fulfill their tasks.

HTTP Utilities

The following tools serve as workhorses for making connections over HTTP or HTTPS. Alone, they do not find vulnerabilities or secure a system, but their functionality can be put to use to extend the abilities of a web vulnerability scanner, peek into SSL traffic, or encrypt client/server communication to protect it from network sniffers.

Curl

Where Netcat deserves bragging rights for being a flexible, all-purpose network tool, curl deserves considerable respect as a flexible tool for HTTP connections. It consists of a command-

line tool (which is the focus of this section) and a high-performance, crossplatform, open source library. Its home page, <http://curl.haxx.se>, contains links to source code, documentation, and mailing lists. You'll find that the curl mailing lists are helpful, active lists regardless of whether you're trying to understand the command line or using one of the library's APIs.

Implementation

The **curl** command is a default tool on most Unix-based systems. If it's not present, then it's likely available as a package for your system or you can install it from source.

To connect to a web site, specify the URL on the command line, like the following example:

```
$ curl http://antihackertoolkit.com
```

The power and helpfulness of curl is best demonstrated by the scripting you can build around it. The **curl** command could be used to crawl a web site, repeat requests for a brute-force guessing attack, or replay requests to exploit a vulnerability. Table 14-3 lists some of its most useful options. Note that the **curl** command accepts a long (started by two dashes) and a short (started by one dash) form for most of its options.

Option	Description
-H --header	Sets a client request header. Use this to imitate many scenarios. For example:
You're likely	User-Agent: Mozilla/5.0 spoofs a particular browser. Referer: http://localhost/admin bypasses poor authorization that checks the Referer.
-b --cookie	X-Forwarded-For: http://localhost/admin bypasses poor authorization that checks a proxy header. -b uses a file that contains cookies to send to the server. For example, -b cookie.txt includes the contents of cookie.txt with all HTTP requests. Cookies can also be specified on the command line in the following form: -b ASPSESSIONID=INEIGNJCNDEECMNPCPOEEMNC; -c uses a file that stores cookies as they are set by the server.
-c --cookiejar	For example, -c cookies.txt holds every cookie from the server. Cookies are important for bypassing form-based authentication and spoofing sessions.
-d --data	Submits data with a POST request. This includes form data or any other data generated by the web application. For example, to set the form field for a login page, use -d login=arha&passwd=tenar . This option is useful for writing custom brute-force passwordguessing scripts. The real advantage is that the requests are made with POST requests, which are more tedious to craft with a tool such as Netcat. Use the --data-ascii , --data-binary , or --dataurlencode variant to affect how the data is encoded.
-G --get	Forces the data sent via the --data option to be submitted with the HTTP GET method instead of the default POST method.
-u --user	Sets the credentials for server-based authentication. For example: --user arha:tenar

OpenSSL for web hacking because you can perform most of the necessary activities from a browser or through an interactive proxy like ZAP (covered later in this chapter). However, one important use of OpenSSL is to generate a certificate for an SSL/TLS service. The OpenSSL library provides a Perl script (CA.pl) and a shell script (CA.sh) that automate the basic steps for creating a self-signed certificate (cert). The following example provides a verbose description of the steps involved for generating and signing certs.

The first step is to generate a Certificate Authority (CA) cert. The CA cert represents an ultimate authority in terms of a cert's validity. The act of signing another cert by the CA connotes that the signed cert has been "approved" or "verified." In other words, the CA attests that a cert should be trusted (with the implication that you trust the CA). Use the **req** and **ca** actions to generate a cert and establish it as your local CA:

Stunnel

OpenSSL is excellent for one-way SSL conversions. Unfortunately, you can run into situations in which the client sends out HTTPS connections and cannot be downgraded to HTTP. In these cases, you need a tool that can either decrypt SSL or sit between the client and server and watch traffic in clear text. Stunnel provides this functionality. Install this tool with your system's package manager or download it from [https:// www.stunnel.org](https://www.stunnel.org).

You can also use stunnel to wrap SSL around any network service. For example, you could set up stunnel to manage connections to an Internet Message Access Protocol (IMAP) service to provide encrypted access to e-mail (you would also need stunnel to manage the client side as well). Fortunately, modern operating systems and services recognize the importance of encrypting connections with SSL/TLS. Stunnel is now needed less as a “patch” for plaintext services and more as a tool for redirecting traffic in order to manipulate it for security testing.

Implementation

Stunnel has two major versions, 3 and 4. The majority of this section relates to the command-line options for the stunnel 3 version because the command line tends to be easier to deal with in rapidly changing environments and one-off testing of services. Check out the end of the section for configuration differences in version 4, the biggest of which is its bias for relying on a configuration file instead of command-line options to control its activity. Both versions provide the same capabilities, and all of the following techniques can be applied to either version.

If you're already familiar with stunnel version 3 command-line options, check out the Perl script provided by the project at [https://www.stunnel.org/downloads /stunnel3](https://www.stunnel.org/downloads/stunnel3). The script wraps the new command-line syntax with the options used by version 3.

SSL communications rely on certificates. The first thing you need is a valid PEM file that contains encryption keys to use for the communications. Stunnel comes with a default file called `stunnel.pem`, which it lets you define at compile time.

If you wish to use a different cert, use the following `openssl` command. This is slightly different from the command covered in the previous “OpenSSL” section of this chapter. A notable difference is the inclusion of the `-nodes` option, which skips the encryption of the cert's private key.

```
$ openssl req -new -out stunnel.pem -keyout stunnel.pem -nodes -x509 \  
> -days 365  
...follow prompts...  
$ openssl dhparam 2048 >> stunnel.pem
```

In the next section, we'll provide this cert to stunnel with its `-p` option to enable stunnel to receive SSL connections. Note that the command is going to complain if the file's “world” permissions are set. Suppress this complaint with the following command:

```
$ chmod o-rwx stunnel.pem
```

Intercept Traffic One use of stunnel is to intercept traffic by downgrading client connections from HTTPS to HTTP, inspect or manipulate the traffic, and then upgrade the connection back from HTTP to HTTPS for the server. The concept is similar to using an interactive proxy (see the upcoming “Zed Attack Proxy” section) to be able to view the plaintext form of HTTPS traffic.

Run stunnel in normal daemon mode (`-d`). This mode accepts SSL traffic and outputs traffic in clear text. The `-f` option forces stunnel to remain in the foreground. This is useful for watching connection information and making sure the program is working. Stunnel is not an end-point program. In other words, you need to specify a port on which the program listens (`-d port`) and a host and port to which traffic is forwarded (`-r host:port`).

Instead of using both commands to forward traffic directly to the server, we could have a client (like a web browser) connect to the listener on port 443 (the first example) and forward that plaintext traffic over port 80 to the other listener on port 80 (the second example), which in turn would forward the traffic over SSL/TLS to its final destination. In this way we can peek into an encrypted connection.

Redirecting traffic through stunnel may also require spoofing the server's IP address so that the client connects to the first stunnel listener. Depending on the security configuration of the client, it may reject a connection if the stunnel.pem cert is invalid. This problem can be solved if you can find a way to install the stunnel.pem cert as a trusted cert or sign it with a CA cert and install the CA cert as a trusted signer in the client. This is straightforward with desktop browsers, but it's more difficult for mobile or embedded devices.

Stunnel is a robust way to wrap SSL/TLS protection around an otherwise unencrypted service. Use the `-l` option to specify the full path to a service daemon. Then launch stunnel (or create a service for it on a Unix-based service manager like xinetd or rinetd):

```
$ sudo stunnel3 -p stunnel.pem -f -d 443 -l /path/to/daemon
```

Most services natively support SSL/TLS connections. This is more useful for setting up redirects in order to inspect traffic between a client and server. For example, some clients either don't provide HTTP proxy settings (otherwise you could use a tool like the Zed Attack Proxy discussed a bit later) or run some protocol other than HTTP over the SSL/TLS connection. In these cases, it's necessary to use host spoofing tricks and redirection so that you can "downgrade" the client's connection from SSL/TLS in order to manipulate it, then "upgrade" the connection back to SSL/TLS when sending traffic on to the server.

NOTE

The client mode setting is needed when stunnel connects to another host. It controls whether the remote service expects an SSL/TLS connection (`client = yes`) or not (`client = no`).

If the path names correspond to the correct location of the certificate files, you're ready to go. Otherwise, change the paths and define the services you wish to use. Table 14-4 lists some additional directives for the stunnel.conf file. This is not an exhaustive list, but it is representative of the most useful directives for getting stunnel started and debugging problems.

The **TIMEOUTxxx** directives are useful for minimizing the impact of some kinds of denial of service attacks that attempt to keep connections open for a long time or open lots of connections to exhaust resources. They can also be increased or decreased depending on the expected latency of a connection.

These previous examples may seem familiar if you've read about Netcat in Chapter 7. The primary difference is that stunnel establishes encrypted channels whereas Netcat just deals with "normal" plaintext TCP connections. You're less likely to need stunnel if you're interacting with a web site from a browser. However, it comes in handy for dealing with HTTP clients in embedded devices or clients that use non-HTTP protocols.

Directive	Description
Foreground	Values: yes or no Available only for Unix-based stunnel execution. It will print activity to stderr, which is an excellent way to troubleshoot connectivity problems.
TIMEOUTbusy	Value: time in seconds Time to wait for data. Available only as part of a specific service definition.
TIMEOUTclose	Value: time in seconds Time to wait for close_notify socket messages. The stunnel developers recommend a value of 0 when using the Internet Explorer browser. Available only as part of a specific service definition.
TIMEOUTidle	Value: time in seconds Time to keep an idle connection before closing it. Available only as part of a specific service definition.

Table 14-4 Additional stunnel.conf Directives

Application Inspection

The previous tools in this chapter focused on the platform beneath the code that drives a web application. The platform needs to start out secure so that it doesn't weaken the code above. But the platform is usually a small part of the application—at least from the end user's perspective. A web application's platform may consist of tens of thousands of web servers connected to massive data stores, but if it only exposes ports 80 and 443 to the user, and the application's document root (the location of its web pages) is locked down, then there's very little of the platform for an attacker to target.

So, the attacker targets the application's behavior instead. This is where we discover vulnerabilities that attackers exploit with techniques like SQL injection, HTML injection (aka cross-site scripting), account hijacking, logic flaws, and more. Many of these attacks require no tools other than a web browser. But some tools make the process easier.

This section covers tools that assist with the manual analysis of and interaction with a web application. For this section we care much less about whether the application is running on Apache or IIS, or whether the source code is Ruby or Java. Knowing those details informs and influences some of the attacks that we might try against the web application, but in this section we care more about how the web application handles cookie values, or how it responds to different values for a URL parameter, or what kinds of data it accepts from a form submission.

These tools help record, analyze, and manipulate the requests and responses to a web site in order to see how securely it's written. We'll be focusing more on how to use each tool rather than how to find specific kinds of vulnerabilities. But don't worry, many web app vulns are easy to understand and even easier to exploit. You can find many web security resources at <http://deadliestwebattacks.com>.

Zed Attack Proxy

The browser is as much a tool for hacking web applications as it is for interacting with them. Many web application attacks require a meager knowledge of HTML and no other tool than a browser's address bar. Manipulating links is a primary way of testing a site's security. But the browser alone is a cumbersome attack platform for conducting security tests.

Zed Attack Proxy (ZAP) is a premier example of an *interactive proxy*. An interactive proxy provides the means to inspect, alter, and manipulate web traffic in order to probe a web application for the presence of vulns. ZAP does this and more. It is able to passively inspect traffic for indicators of poor (and good!) security practices. It may also run active attacks against a web application, such as automatically crawling pages or fuzzing parts of a request in order to elicit errors (or exploits) against the site.

The ZAP project is part of the OWASP Foundation's efforts to improve the knowledge, tools, and skills related to web application security. The tool's project page is at https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. This tool is in fact a resurrection of the defunct Paros Proxy that served web hackers well in the early 2000s. The development home for ZAP is at <https://code.google.com/p/zaproxy/>.

You cannot effectively understand and explore web security without the capability provided by an interactive proxy. You may never need a tenth of the features that ZAP provides, but you do need to understand how a tool like this is used to hack web applications.

NOTE

The Burp Proxy tool provides similar capabilities to Zed Attack Proxy. It is a commercial tool available at <http://portswigger.net/burp/proxy.html>.

Installation

The easiest way to get started with ZAP is to download an installer for your operating system of choice. ZAP is written in Java, so your experience in using it doesn't noticeably change between systems. Some developers have also started to extend ZAP with Python, which is also cross-platform. Web applications are not tied to operating systems; it's a good sign that web hacking tools are not either.

Following are the basic steps for downloading and building the latest source code.

Note that you'll need to set up your environment correctly for building Java source code (e.g., class files). ZAP requires a JDK (available from www.java.com) and the **ant** command (available from your system's package manager, or at <http://ant.apache.org>).

```
$ svn co https://zaproxy.googlecode.com/svn/trunk zap
$ cd zap
$ cd build
$ ant
$ cd zap
$ sh zap.sh
```

ZAP has a relatively fast-paced development cycle. But it has also managed to maintain stability in the trunk. You're unlikely to run into problems using the latest version of the source code.

Sqlmap

SQL injection is a class of web security vulns and exploits that affects the datastore used by a web app. The programming flaws that lead to SQL injection are similar to the ones that produce the kind of HTML injection vulns we briefly covered in the previous "Zed Attack Proxy" section. A web application takes a piece of data received from the browser (and therefore a value that can be manipulated by an attacker) and uses string concatenation to piece together a database query (or snippet of text in equivalent HTML injection scenarios) based on the received data, but the app neglects to prevent the received data from changing the meaning of the SQL statement (or HTML page). That's the long-winded explanation; we'll look at some clarifying examples using the sqlmap tool.

Sqlmap automates the detection and exploitation of SQL injection vulns. The project's home page is <http://sqlmap.org>. It brings together attack techniques that have been improving for more than a decade and ties them to specific exploit methods for most of the possible SQL-based databases that a web app might use.

The following examples demonstrate the basic way that SQL injection vulns occur within a web app and a simple way they can be exploited. The examples include PHP code to show how the server mishandles a request, but this vuln is not specific to PHP. It happens in any programming language in which a developer constructs a SQL statement via string concatenation on user-supplied data that hasn't been properly validated.

To begin, recall the search link referenced in the "Zed Attack Proxy" section:

```
https://web.site/search?q=tardis+repair
```

When the web app receives the request, it might place the value of the "q" parameter into a database query like the following. An important thing to notice is how the term is treated as a string within the query by placing it within single quotes (i.e., apostrophe characters).

```
SELECT info FROM howto_guides WHERE topic = 'tardis repair';
```

Any results that match the search term would be returned in the web page. Now, consider what might happen if the search term includes SQL syntax characters. The following link includes a terminating single quote followed by a semicolon (which denotes the end of a SQL statement), followed by a SQL comment delimiter (dashdash-space, which causes the SQL query interpreter to ignore whatever follows the delimiter). Remember that most web servers will interpret the plus symbol as a space character when it's part of the URL. Use the percent encoding of %2b to represent a literal plus symbol. **`https://web.site/search?q=tardis+repair' ;--+`**

The web app constructs a new SQL statement whose behavior is identical to the previous example, but whose syntax has been modified:

```
SELECT info FROM howto_guides WHERE topic = 'tardis repair';-- ';
```

At this point we have a vulnerable web app. The next step might be to exploit the vulnerability by extracting additional data into the statement's original result set. The following link demonstrates one way this might happen by using SQL's **UNION** keyword to combine the results for "tardis repair" with results from the database's **USERS** table:

```
https://web.site/search?q=tardis+repair'+UNION+SELECT+password+FROM+USERS;--+
```

Since the web app has done nothing to prevent this kind of attack, it builds a statement like the following, the results of which would dump passwords alongside the “normal” results expected for this query:

```
SELECT info FROM howto_guides WHERE topic = 'tardis repair' UNION SELECT password
FROM USERS;-- ';
```

And the reason the web app would construct a query like this is if its source code used string concatenation. Again, this example uses PHP, but the vuln can be re-created in almost any programming language used by web apps:

```
$table = 'howto_guides';
$sql = "SELECT info FROM {$table} "
      . "WHERE topic = '{$_GET['q']}'";
```

In this PHP code, the SQL statement is built with two different variables. The `$table` variable is a constant value that can't be manipulated by a visitor to the web site. Such usage could be considered safe, albeit ill-advised since it relies on a poor programming pattern. The web app takes the second variable, `$_GET['q']`, directly from the “q” parameter of the link. This variable is completely under the control of an attacker, which is how the string can be manipulated to contain arbitrary SQL statements. The web app has failed to properly separate code (the grammar of the SQL query) from data (the values to insert into the query's grammar).

Password OpSec

We can't control what happens to our passwords once they leave our phone, laptop, or other device. In fact, we can rarely control what happens to them once they leave our fingertips and are typed on a keyboard or touchscreen. But we can follow some basic Operations Security (OpSec) in choosing, managing, and using passwords. The following list of recommendations is biased toward users of web applications, but the principles should be applicable to using passwords in general:

- Keep your system up to date. This reduces your exposure to compromise by malware and viruses.
- Do not use the unique password of your primary e-mail account for any other account you create. Most web apps rely on e-mail for password reset and recovery mechanisms. E-mail accounts are a prime target for theft. Losing access to your e-mail account (or unwittingly divulging the account's password to someone else) means not only losing contact with friends and family via that account, but an attacker may be able to leverage the e-mail to access other accounts.
- Enable multifactor authentication whenever a web app offers support for it. This helps protect your account from compromise even if your password is weak (and easily guessed) or disclosed (by a server-side hack).
- Avoid entering your credentials on public or shared computers. The security of such systems cannot be guaranteed and they are excellent targets for hackers to install keyloggers.
- Avoid authenticating to web apps when using public Wi-Fi networks. Or at least restrict your activity to apps that use HTTPS for all communication. See Chapter 10 for reasons why this matters.
- Avoid any web site whose password recovery mechanism e-mails your original password rather than a new, temporary one. Sending an e-mail with your original password means the site does not hash passwords (against all recommended security practices) and its developers are ignorant of secure programming.
- Choose a password that isn't based on easily discoverable personal information such as school names, demographic details, a favorite topic you always blog about, or pets. If you're a pet, don't use any of this information about your human. On the Internet, no one knows you're a dog. Make sure they don't know your password either.
- If you use your social media account (e.g., Facebook or Twitter) as the ID for other apps, follow the same advice given for your e-mail password. Plus, always make sure the login prompt you receive points to the correct domain for the social media site.

Application developers and system administrators bear different responsibilities for protecting passwords. The majority of this chapter is predicated on the assumption that a password hash—or an entire password store—has been compromised. Preventing such a compromise in the first place must be a goal of secure application deployment. This entails efforts like keeping software updated with its latest patches, using encrypted channels for communication, and establishing network monitoring to enable quick reaction and analysis in the event of a compromise.

John the Ripper

John the Ripper (www.openwall.com/john/) remains one of the fastest, most versatile, and most popular password crackers available. It supports password hashing schemes used by many systems, including most Unix-based systems (like OpenBSD and various Linux distributions) and the various Windows hashes, as well as proprietary password hashing functions used by several database and software packages for user account management. John’s cracking modes include specialized wordlists, the ability to customize the generation of guesses based on character type and placement (useful when targeting a specific password policy), raw brute force, and statistically guided brute force that uses successfully cracked passwords to influence future guesses. And John runs on just about any operating system.

Implementation

First, you need to obtain and compile John. The following examples use the John-1.7.9 version with the “jumbo-7” patch. The “jumbo” patches include code from contributors who have added support for more esoteric password file formats or hash algorithms. Download and extract the tarball. John may be compiled on any Unix-based system or Windows. When you type the **make** command, the compilation step will complain that you haven’t specified a target. Don’t worry; that’s okay.

```
$ tar zxvf john-1.7.9-jumbo-7.tar.gz
$ cd john-1.7.9-jumbo-7
$ cd src
$ make
```

John has hard-coded many compilation flags and optimization settings for dozens of specific operating systems and CPU architectures. It should be easy to guess which is the most appropriate for your own system. Plus, look for the “(best)” annotation for your system.

Cracking Passwords

John is compiled and awaits our command. Let’s crack a password. John automatically recognizes common password formats extracted from operating system files like `/etc/shadow` or dumped by tools like `pwdump` (we’ll get to that tool in a moment). In practice, John supports close to 150 different hashing algorithms; you’ll find them listed by running the benchmark with the **-test** option. The following example shows John’s ability to guess the correct format for password entries. First, create a text file named `windows.txt` with the following two lines containing an entry for “Ged” and “Arha.” They represent passwords taken from a Windows system.

THC-Hydra

THC-Hydra (aka simply Hydra) easily surpasses the majority of brute-force tools available on the Internet for two reasons: it is fast, and it targets authentication mechanisms for several dozen protocols. Its source code and documentation are available from <https://www.thc.org/thc-hydra/>. The Hacker’s Choice web site (<https://www.thc.org>) contains many security tools, although some of them have not been maintained for several years. Even so, its tools, papers, and information are informative. Compile Hydra from source or look for it in your system’s package manager. It will compile under any Unix-based system and Cygwin.

Implementation

Hydra compiles on BSD and Linux systems without a problem; the Cygwin and OS X environments have been brought to equal par in the most current version. Follow the usual **./configure, make, make install** method for compiling source code. Once you have successfully compiled it, check out the command-line arguments detailed in Table 15-2.

Hydra Option	Description
-R	Restores a previous aborted/crashed session from the hydra.restore file (by default this file is created in the directory from which Hydra was executed).
-S	Connects via SSL.
-s n	Connects to port <i>n</i> instead of the service's default port.
-l name	Uses <i>name</i> from the command line or from each line of <i>file</i> as the -L file username portion of the credential.
-p password	Uses <i>password</i> from the command line or from each line of <i>file</i> as -P file the password portion of the credential.
-C file	Loads user:password combinations from <i>file</i> . Each line contains one combination separated by a colon.

Table 15-2 Hydra Command-Line Options (*continued*)

Hydra Option	Description
-e nsr	Also tests the login prompt for a null password (n), a password equal to the username (s), or a password of the login name reversed (r).
-M file	Targets the hosts listed in each line of <i>file</i> instead of a single host.
-o file	Writes a successful username and password combination to <i>file</i> instead of stdout.
-f	Exits after the first successful username and password combination is discovered for the host. If multiple hosts are targeted (-M), then Hydra will continue to run against other hosts until the first successful credentials are found.
-t n	Executes <i>n</i> parallel connects to the target service. The default is 16. The performance gain from this option is affected by both your system's resources and the target's resources.
-w n	Waits no more than <i>n</i> seconds for a response from the service before assuming no response will come.
-v	Reports verbose status information.
-V	
-4	Connects over IPv4 (-4) or IPv6 (-6).
-6	
server	Specifies the target's IP address or hostname. For multiple targets, use the -M option to load targets from a text file (with each target on a single line).
service	Specifies the target's service to brute force.

Table 15-2 Hydra Command-Line Options

The target is defined by the **server** and **service** arguments. The type of service can be any one of the applications in the following list, which contains some of the more interesting services that Hydra is able to brute force. Note that for several of the services, a port for SSL access has already been defined. The first number in the parentheses is the service's default port; the second number is the service's port over SSL. Make sure to use the **-s** option if the target service is listening on a different port.

- **cisco (23)** Telnet prompt specific to Cisco devices when only a password is requested.
- **cisco-enable (23)** Entering the enable, or superuser, mode on a Cisco device. You must already know the initial login password and supply it with the **-m** option and without the **-l** or **-L** options (there is no prompt for the username). **hydra -m access_password -P password.lst 10.0.10.254 cisco-enable**

- **http, http-head, http-get (80,443)** HTTP Basic Authentication schemes on the web service. Note that this technique expects the server to send particular HTTP response codes; otherwise, the accuracy of this module may suffer. Use the **https** version for SSL-enabled services.
- **http-get-form, http-post-form (80,443)** HTML login forms over HTTP. Specify the target path with the **-m** option. Use the **https** version for SSL-enabled services. Run the following command for instructions on specific usage: **hydra -U http-post-form**
- **http-proxy (3128)** Web proxies such as Squid.
- **imap, imaps (143, 993)** E-mail access.
- **irc (194 or 6667, 6697)** Chat software.
- **mssql (1433)** Microsoft SQL Server. Remember that SQL Server may use integrated authentication. Try the default SQL accounts, such as sa, and Windows accounts.
- **mysql (3306)** MySQL database server.
- **oracle-listener (1521)** Oracle database server.
- **pop3, pop3s (110, 995)** E-mail access.
- **postgres (5432)** PostgreSQL database server.
- **rdp (3389)** Remote Desktop Protocol.
- **smb/cifs (139 or 445)** Windows SMB services such as file shares and IPC\$ access.
- **snmp (161 or 1993)** UDP-based network management protocol.
- **socks5 (1080)** Proxy.
- **ssh (22)** Secure Shell, remote command-line administration.
- **svn (3690)** Source code versioning system.
- **teamspeak (8767)** Distributed voice chat system, often used by gamers.
- **vnc (5900 and 5901)** Remote administration for GUI environments.

Running Hydra is simple. The biggest problem you may encounter is the choice of username/password combinations. Here is one example of targeting a Windows SMB service. If port 139 or 445 is open on the target server and an error occurs, then the

Windows *Server* service might not be started—the brute-force attack will not work.

If you really do wish to have an optimum test, as opposed to an exhaustive test, then you may wish to consider the **-C** option instead of supplying a file each for **-L** (users) and **-P** (passwords). The **-C** option takes a single file as its argument. This file contains username and password combinations separated by a colon (:). This is often a more efficient method for testing accounts because you can populate the file with common default username/password combinations or combinations you expect to be more likely to succeed (perhaps based on passwords cracked from a tool like John the Ripper). Using this option reduces the number of unnecessary attempts when a username does not exist. This is more useful for situations where you only wish to test for default passwords and the most common passwords.

Do not forget to use the **-e** option when auditing your network's services. The **-e** option turns on testing for the special cases of no password (**-e n**) or a password equal to the username (**-e s**). Specify an **r** (**-e r**) to submit the reverse of the login name as the service's password.

Note that Hydra writes a state file (hydra.restore) to the current directory from which it is executed. You can use the **-R** option to restart an interrupted scan. This also means that if you wish to run concurrent scans against different servers or different services, then you should do so in different directories. From a forensics perspective, the hydra.restore file might be a good addition to the list of common “hacker” files to search for on suspect systems—just remember that a one-line change to the source code can change this filename.

Hydra now also includes a GUI based on the open source GTK library. This version, called xHydra, provides all of the functionality of the command line.