NAMA : ALIFATURROHMAH

NIM : 2121700002

**CAPTURING APPLICATION TRAFFIC**

Surprisingly, capturing useful traffic can be a challenging aspect of protocol analysis. This chapter describes two different capture techniques: passive and active. Passive capture doesn’t directly interact with the traffic. Instead, it extracts the data as it travels on the wire, which should be familiar from tools like Wireshark. Active capture interferes with traffic between a client application and the server; this has great power but can cause some complications.

**Passive Network Traffic Capture Passive**

capture is a relatively easy technique: it doesn’t typically require any specialist hardware, nor do you usually need to write your own code. Figure 2-1 shows a common scenario: a client and server communicating via Ethernet over a network.

**Quick Primer for Wireshark**

Wireshark is perhaps the most popular packet-sniffing application available. It’s cross platform and easy to use, and it comes with many built-in protocol analysis features. In Chapter 5 you’ll learn how to write a dissector to aid in protocol analysis, but for now, let’s set up Wireshark to capture IP traffic from the network. To capture traffic from an Ethernet interface (wired or wireless), the capturing device must be in promiscuous mode. A device in promiscuous mode receives and processes any Ethernet frame it sees, even if that frame wasn’t destined for that interface. Capturing an application running on the same computer is easy: just monitor the outbound network interface or the local loopback interface (better known as localhost). There are three main view areas. Area ➊ shows a timeline of raw packets captured off the network.➋ provides a dissected view of the packet, separated into distinct protocol layers that correspond to the OSI network stack model. Area ➌ shows the captured packet in its raw form. n. Due to the nature of networks and IP, there is no guarantee that packets will be received in a particular order. Therefore, when you are capturing packets, the timeline view might be difficult to interpret. Fortunately, Wireshark offers dissectors for known protocols that will normally reassemble the entire stream and provide all the information in one place. For example, highlight a packet in a TCP connection in the timeline view and then select Analyze ▸ Follow TCP Stream from the main menu.

**Alternative Passive Capture Techniques**

Sometimes using a packet sniffer isn’t appropriate, for example, in situations when you don’t have permission to capture traffic. You might be doing a penetration test on a system with no administrative access or a mobile device with a limited privilege shell. You might also just want to ensure that you look at traffic only for the application you’re testing. That’s not always easy to do with packet sniffing unless you correlate the traffic based on time.

**System Call Tracing**

Many modern operating systems provide two modes of execution. Kernel mode runs with a high level of privilege and contains code implementing the OS’s core functionalityWhen an application wants to connect to a remote server, it issues special system calls to the OS’s kernel to open a connection. The app then reads and writes the network data. Depending on the operating system running your network applications, you can monitor these calls directly to passively extract data from an application.

**The strace Utility on Linux**

In Linux, you can directly monitor system calls from a user program without special permissions, unless the application you want to monitor runs as a privileged user. Many Linux distributions include the handy utility strace, which does most of the work for you. If it isn’t installed by default, download it from your distribution’s package manager or compile it from sourceThe first entry ➊ creates a new TCP socket, which is assigned the handle 3. The next entry ➋ shows the connect system call used to make a TCP connection to IP address 192.168.10.1 on port 5555. The application then writes the string Hello World! ➌ before reading out a string Boo! ➍. The output shows it’s possible to get a good idea of what an application is doing at the system call level using this utility, even if you don’t have high levels of privilege.

**Monitoring Network Connections**

with DTrace DTrace is a very powerful tool available on many Unix-like systems, including Solaris (where it was originally developed), macOS, and FreeBSD. It allows you to set systemwide probes on special trace providers, including system calls used for IPv4 connections at ➊; in many cases these structures can be directly copied from the system’s C header files. The system call to monitor is specified at ➋. At ➌, a DTrace-specific filter is used to ensure we trace only connect calls where the socket address is the same size as sockaddr\_in. At ➍, the sockaddr\_in structure is copied from your process into a local structure for DTrace to inspect. At ➎, the process name, the destination IP address, and the port are printed to the console. The output shows individual connections to IP addresses, printing out the process name, for example 'Google Chrome', the IP address, and the port connected to. Unfortunately, the output isn’t always as useful as the output from strace on Linux, but DTrace is certainly a valuable tool.

**Process Monitor on Windows**

In contrast to Unix-like systems, Windows implements its user-mode network functions without direct system calls. The networking stack is exposed through a driver, and establishing a connection uses the file open, read, and write system calls to configure a network socket for use. Even if Windows supported a facility similar to strace, this implementation makes it more difficult to monitor network traffic at the same level as other platforms. Column ➊ shows the name of the process that established the connection. Column ➋ shows the operation, which in this case is connecting to a remote server, sending the initial HTTP request and receiving a response. Column ➌ indicates the source and destination addresses, and column ➍ provides more in-depth information about the captured event. Although this solution isn’t as helpful as monitoring system calls on other platforms, it’s still useful in Windows when you just want to determine the network protocols a particular application is using.

**Advantages and Disadvantages of Passive Capture**

The greatest advantage of using passive capture is that it doesn’t disrupt the client and server applications’ communication. It will not change the destination or source address of traffic, and it doesn’t require any modifications or reconfiguration of the applications. Passive capture might also be the only technique you can use when you don’t have direct control over the client or the server. You can usually find a way to listen to the network traffic and capture it with a limited amount of effort. After you’ve collected your data, you can determine which active capture techniques to use and the best way to attack the protocol you want to analyze. One major disadvantage of passive network traffic capture is that capture techniques like packet sniffing run at such a low level that it can difficult to interpret what an application received. Tools such as Wireshark certainly help, but if you’re analyzing a custom protocol, it might not be possible to easily take apart the protocol without interacting with it directly. Passive capture also doesn’t always make it easy to modify the traffic an application produces. Modifying traffic isn’t always necessary, but it’s useful when you encounter encrypted protocols, want to disable compression, or need to change the traffic for exploitation.

**Active Network Traffic Capture Active capture**

differs from passive in that you’ll try to influence the flow of the traffic, usually by using a man-in-the-middle attack on the network communication. As shown in Figure 2-7, the device capturing traffic usually sits between the client and server applications, acting as a bridge. This approach has several advantages, including the ability to modify traffic and disable features like encryption or compression, which can make it easier to analyze and exploit a network protocol.

**Network Proxies**

The most common way to perform a man-in-the-middle attack on network traffic is to force the application to communicate through a proxy service. In this section, I’ll explain the relative advantages and disadvantages of some of the common proxy types you can use to capture traffic, analyze that data, and exploit a network protocol. I’ll also show you how to get traffic from typical client applications into a proxy.

**Port-Forwarding Proxy**

Port forwarding is the easiest way to proxy a connection. Just set up a listening server (TCP or UDP) and wait for a new connection. When that new connection is made to the proxy server, it will open a forwarding connection to the real service and logically connect the two, as shown in Figure 2To create our proxy, we’ll use the built-in TCP port forwarder included with the Canape Core librariThis very simple script creates an instance of a FixedProxyTemplate ➊. Canape Core works on a template model, although if required you can get down and dirty with the low-level network configuration. The script configures the template with the desired local and remote network information. The template is used to create a service instance at ➎; you can think of documents in the framework acting as templates for services. The newly created service is then started; at this point, the network connections are configured. After waiting for a key press, the service is stopped at ➏. Then all the captured packets are written to the console using the WritePackets() method ➐. Running this script should bind an instance of our forwarding proxy to the LOCALPORT number for the localhost interface only. When a new TCP connection is made to that port, the proxy code should establish a new connection to REMOTEHOST with TCP port REMOTEPORT and link the two connections together.

**Redirecting Traffic to Proxy**

With our simple proxy application complete, we now need to direct our application traffic through it. For a web browser, it’s simple enough: to capture a specific request, instead of using the URL form http://www.domain.com/resource, use http://localhost:localport/resource, which pushes the request through your port-forwarding proxy. Other applications are trickier: you might have to dig into the application’s configuration settings. Sometimes, the only setting an application allows you to change is the destination IP address. But this can lead to a chicken-and-egg scenario where you don’t know which TCP or UDP ports the application might be using with that address, especially if the application contains complex functions running over multiple different service connections. This occurs with Remote Procedure Call (RPC) protocols, such as the Common Object Request Broker Architecture (CORBA). This protocol usually makes an initial network connection to a broker, which acts as a directory of available services. A second connection is then made to the requested service over an instance-specific TCP port. In this case, a good approach is to use as many network-connected features of the application as possible while monitoring it using passive capture techniques. By doing so, you should uncover the connections that application typically makes, which you can then easily replicate with forwarding proxies.

**Advantages of a Port-Forwarding Proxy**

The main advantage of a port-forwarding proxy is its simplicity: you wait for a connection, open a new connection to the original destination, and then pass traffic back and forth between the two. There is no protocol associated with the proxy to deal with, and no special support is required by the application from which you are trying to capture traffic.

**Disadvantages of a Port-Forwarding Proxy**

Of course, the simplicity of a port-forwarding proxy also contributes to its disadvantages. Because you are only forwarding traffic from a listening connection to a single destination, multiple instances of a proxy would be required if the application uses multiple protocols on different ports. For example, consider an application that has a single hostname or IP address for its destination, which you can control either directly by changing it in the application’s configuration or by spoofing the hostname. The application then attempts to connect to TCP ports 443 and 1234. Because you can control the address it connects to, not the ports, you need to set up forwarding proxies for both, even if you are only interested in the traffic running over port 1234.

**SOCKS Proxy**

Think of a SOCKS proxy as a port-forwarding proxy on steroids. Not only does it forward TCP connections to the desired network location, but all new connections start with a simple handshake protocol that informs the proxy of the ultimate destination rather than having it fixed. It can also support listening connections, which is important for protocols like File Transfer Protocol (FTP) that need to open new local ports for the server to send data to. Figure 2-9 provides an overview of SOCKS proxy. As an example, a client will send the request shown in Figure 2-10 to establish a SOCKS connection to IP address 10.0.0.1 on port 12345. The USERNAME component is the only method of authentication in SOCKS version 4 (not especially secure, I know). VER represents the version number, which in this case is 4. CMD indicates it wants to connect out (binding to an address is CMD 2), and the TCP port and address are specified in binary form.

**Simple Implementation**

The Canape Core libraries have built-in support for SOCKS 4, 4a, and 5. Place Listing 2- 6 into a C# script file, changing LOCALPORT ➋ to the local TCP port you want to listen on for the SOCKS proxy.

**Redirecting Traffic to Proxy**

To determine a way of pushing an application’s network traffic through a SOCKS proxy, look in the application first. For example, when you open the proxy settings in Mozilla Firefox, the dialog in Figure 2-12 appears. From there, you can configure Firefox to use a SOCKS proxy. application, the Java Runtime accepts command line parameters that enable SOCKS support for any outbound TCP connection. For example, consider the very simple Java application in Listing 2-7, which connects to IP address 192.168.10.1 on port 5555.

**Advantages of a SOCKS Proxy**

The clear advantage of using a SOCKS proxy, as opposed to using a simple port forwarder, is that it should capture all TCP connections (and potentially some UDP if you are using SOCKS version 5) that an application makes. This is an advantage as long as the OS socket layer is wrapped to effectively push all connections through the proxy. A SOCKS proxy also generally preserves the destination of the connection from the point of view of the client application. Therefore, if a client application sends in-band data that refers to its endpoint, then the endpoint will be what the server expects. However, this does not preserve the source address. Some protocols, such as FTP, assume they can request ports to be opened on the originating client.

**Disadvantages of a SOCKS Proxy**

The main disadvantage of SOCKS is that support can be inconsistent between applications and platforms. The Windows system proxy supports only SOCKS version 4 proxies, which means it will resolve only local hostnames. It does not support IPv6 and does not have a robust authentication mechanism.

**HTTP Proxies**

HTTP powers the World Wide Web as well as a myriad of web services and RESTful protocols. Figure 2-14 provides an overview of an HTTP proxy. The protocol can also be co-opted as a transport mechanism for non-web protocols, such as Java’s Remote Method Invocation (RMI) or Real Time Messaging Protocol (RTMP), because it can tunnel though the most restrictive firewalls. It is important to understand how HTTP proxying works in practice, because it will almost certainly be useful for protocol analysis, even if a web service is not being tested. Existing web application–testing tools rarely do an ideal job when HTTP is being used out of its original environment. Sometimes rolling your own implementation of an HTTP proxy is the only solution.

**Forwarding an HTTP Proxy**

The HTTP protocol is specified in RFC 1945 for version 1.0 and RFC 2616 for version 1.1; both versions provide a simple mech anism for proxying HTTP requestsThe method ➊ specifies what to do in that request using familiar verbs, such as GET, POST, and HEAD. In a proxy request, this does not change from a normal HTTP connection. The path ➋ is where the proxy request gets interesting. As is shown, an absolute path indicates the resource that the method will act upon. Importantly, the path can also be an absolute Uniform Request Identifier (URI). By specifying an absolute URI, a proxy server can establish a new connection to the destination, forwarding all traffic on and returning data back to the client. The TCP connection to the proxy now becomes transparent, and the browser is able to establish the negotiated TLS connection without the proxy getting in the way. Of course, it’s worth noting that the proxy is unlikely to verify that TLS is actually being used on this connection. It could be any protocol you like, and this fact is abused by some applications to tunnel out their own binary protocols through HTTP proxies. For this reason, it’s common to find deployments of HTTP proxies restricting the ports that can be tunnele.

**Simple Implementation**

Once again, the Canape Core libraries include a simple implementation of an HTTP proxy. Unfortunately, they don’t support the CONNECT method to create a transparent tunnel, but it will suffice for demonstration purposes.

**Redirecting Traffic to Proxy**

As with SOCKS proxies, the first port of call will be the application. It’s rare for an application that uses the HTTP protocol to not have some sort of proxy configuration. If the application has no specific settings for HTTP proxy support, try the OS configuration, which is in the same place as the SOCKS proxy configuration. For example, on Windows you can access the system proxy settings by selecting Control Panel ▸ Internet Options ▸ Connections ▸ LAN Settings. Many command line utilities on Unix-like systems, such as curl, wget, and apt, also support setting HTTP proxy configuration through environment variables.

**Advantages of a Forwarding HTTP Proxy**

The main advantage of a forwarding HTTP proxy is that if the application uses the HTTP protocol exclusively, all it needs to do to add proxy support is to change the absolute path in the Request Line to an absolute URI and send the data to a listening proxy server. Also, only a few applications that use the HTTP protocol for transport do not already support proxying.

**Disadvantages of a Forwarding HTTP Proxy**

The requirement of a forwarding HTTP proxy to implement a full HTTP parser to handle the many idiosyncrasies of the protocol adds significant complexity; this complexity might introduce processing issues or, in the worst case, security vulnerabilities. Also, the addition of the proxy destination within the protocol means that it can be more difficult to retrofit HTTP proxy support to an existing application through external techniques, unless you convert connections to use the CONNECT method (which even works for unencrypted HTTP). Due to the complexities of handling a full HTTP 1.1 connection, it is common for proxies to either disconnect clients after a single request or downgrade communications to version 1.0 (which always closes the response connection after all data has been received). This might break a higher-level protocol that expects to use version 1.1 or request pipelining, which is the ability to have multiple requests in flight to improve performance or state locality.

**Reverse HTTP Proxy**

Forwarding proxies are fairly common in environments where an internal client is connecting to an outside network. They act as a security boundary, limiting outbound traffic to a small subset of protocol types. (Let’s just ignore the potential security implications of the CONNECT proxy for a moment.) But sometimes you might want to proxy inbound connections, perhaps for load-balancing or security reasons (to prevent exposing your servers directly to the outside world). However, a problem arises if you do this. You have no control over the client. In fact, the client probably doesn’t even realize it’s connecting to a proxy. This is where the reverse HTTP proxy comes in.

**Simple Implementation**

Unsurprisingly, the Canape Core libraries include a built-in HTTP reverse proxy implementation, which you can access by changing the template object to HttpReverseProxyTemplate from HttpProxyTemplate.

**Redirecting Traffic to Your Proxy**

The approach to redirecting traffic to a reverse HTTP proxy is similar to that employed for TCP port-forwarding, which is by redirecting the connection to the proxy. But there is a big difference; you can’t just change the destination hostname. This would change the Host header, shown in Listing 2-10. If you’re not careful, you could cause a proxy loop.1 Instead, it’s best to change the IP address associated with a hostname using the hosts file. But perhaps the application you’re testing is running on a device that doesn’t allow you to change the hosts file. Therefore, setting up a custom DNS server might be the easiest approach, assuming you’re able to change the DNS server configuration.

**Advantage of a Reverse HTTP Proxy**

The advantage of a reverse HTTP proxy is that it doesn’t require a client application to support a typical forwarding proxy configuration. This is especially useful if the client application is not under your direct control or has a fixed configuration that cannot be easily changed. As long as you can force the original TCP connections to be redirected to the proxy, it’s possible to handle requests to multiple different hosts with little difficulty.

**Disadvantages of a Reverse HTTP Proxy**

The disadvantages of a reverse HTTP proxy are basically the same as for a forwarding proxy. The proxy must be able to parse the HTTP request and handle the idiosyncrasies of the protocol.