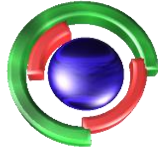
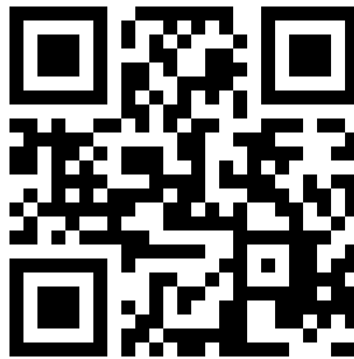
# FUTURE VISION BIE

## One Stop for All Study Materials & Lab Programs




By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

**Visit : https://hemanthrajhemu.github.io**

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

**Join Telegram to get Instant Updates:** https://bit.ly/VTU_TELEGRAM

**Contact: MAIL:** futurevisionbie@gmail.com

**INSTAGRAM:** www.instagram.com/hemanthraj_hemu/

**INSTAGRAM:** www.instagram.com/futurevisionbie/

**WHATSAPP SHARE:** https://bit.ly/FVBIESHARE

13.2 IPSec in Action *187*
13.2.1 IPSec Security Associations *188*
13.2.2 IPSec Protocols: AH and ESP *188*
13.2.3 Tunnel versus Transport Mode *189*
13.2.4 Incompatibility with NAT *190*
13.3 Internet Key Exchange (IKE) Protocol *192*
13.3.1 Preliminaries *192*
13.3.2 IPSec Cookies *192*
13.3.3 IKE Phase I *193*
13.3.4 IKE Phase 2 *196*
13.4 Security Policy and IPSec *197*
13.5 Virtual Private Networks *197*
*Selected References 198*
*Objective-Type Questions 198*
*Exercises 199*
*Answers to Objective-Type Questions 200*

**14. Security at the Transport Layer** **201**
14.1 Introduction *201*
14.2 SSL Handshake Protocol *201*
14.2.1 Steps in the Handshake *201*
14.2.2 Key Design Ideas *203*
14.3 SSL Record Layer Protocol *205*
14.4 OpenSSL *205*
*Selected References 205*
*Objective-Type Questions 206*
*Exercises 206*
*Answers to Objective-Type Questions 208*

**15. IEEE 802.11 Wireless LAN Security** **209**
15.1 Background *209*
15.2 Authentication *211*
15.2.1 Pre-WEP Authentication *211*
15.2.2 Authentication in WEP *211*
15.2.3 Authentication and Key Agreement in 802.11i *211*
15.3 Confidentiality and Integrity *215*
15.3.1 Data Protection in WEP *215*
15.3.2 Data Protection in TKIP and CCMP *217*
*Selected References 220*
*Objective-Type Questions 220*
*Exercises 221*
*Answers to Objective-Type Questions 221*

**16. Cellphone Security** **222**
16.1 Preliminaries *222*

https://hemanthrajhemu.github.io

16.1.1 Entities Involved 222
16.1.2 Security Goals 223
16.2 GSM (2G) Security 224
16.2.1 Entity Authentication and Key Agreement 224
16.2.2 Encryption 226
16.2.3 Problems and Drawbacks 226
16.3 Security in UMTS (3G) 227
16.3.1 Security Enhancements 227
16.3.2 Authentication and Key Agreement 227
16.3.3 Integrity Protection and Encryption 229
*Selected References* 230
*Objective-Type Questions* 231
*Exercises* 231
*Answers to Objective-Type Questions* 232

**17. Non-Cryptographic Protocol Vulnerabilities** 233
17.1 DoS and DDoS 233
17.1.1 Attack Types 233
17.1.2 Impact of SYN Flooding 234
17.2 Session Hijacking and Spoofing 236
17.2.1 Impersonation and Session Hijacking 236
17.2.2 ARP Spoofing 237
17.3 Pharming Attacks 239
17.3.1 Preliminaries 239
17.3.2 Attacks on DNS 239
17.3.3 DNSSEC 242
17.4 Wireless LAN Vulnerabilities 243
17.4.1 Frame Spoofing 244
17.4.2 Violating MAC Etiquette 245
*Selected References* 246
*Objective-Type Questions* 246
*Exercises* 247
*Answers to Objective-Type Questions* 249

**18. Software Vulnerabilities** 250
18.1 Phishing 250
18.2 Buffer Overflow 251
18.2.1 Stack-related Preliminaries 251
18.2.2 Exploiting Stack Overflows 254
18.2.3 Defences 257
18.2.4 Heap Overflows 258
18.3 Format String Attacks 260
18.4 Cross-site Scripting (XSS) 262
18.4.1 XSS Vulnerabilities 262
18.4.2 Overcoming XSS 264

https://hemanthrajhemu.github.io

18.5 SQL Injection 265
18.5.1 The Vulnerability 265
18.5.2 SQL Injection Remedies 268
*Selected References 268*
*Objective-Type Questions 268*
*Exercises 269*
*Answers to Objective-Type Questions 271*

**19. Viruses, Worms, and Other Malware** **272**
19.1 Preliminaries 272
19.2 Virus and Worm Features 273
19.2.1 Virus Characteristics 273
19.2.2 Worm Characteristics 274
19.3 Internet Scanning Worms 277
19.3.1 Case Studies: Code Red and Slammer 278
19.3.2 Worm Propagation Models 279
19.4 Topological Worms 281
19.4.1 E-mail Worms 281
19.4.2 P2P Worms 282
19.5 Web Worms and Case Study 284
19.6 Mobile Malware 286
19.6.1 Introduction 286
19.6.2 Bluetooth 287
19.6.3 Examples 290
19.7 Botnets 290
19.7.1 Basics 290
19.7.2 Case Study: The Storm Botnet 291
*Selected References 293*
*Objective-Type Questions 293*
*Exercises 294*
*Answers to Objective-Type Questions 295*

**20. Access Control in the Operating System** **296**
20.1 Preliminaries 296
20.2 Discretionary Access Control — Case Studies 299
20.2.1 Unix 299
20.2.2 Windows 301
20.3 Mandatory Access Control 308
20.3.1 Multi-level Security (MLS) 308
20.3.2 Security Policies 311
20.4 Role-based Access Control 312
20.5 SELinux and Recent Trends 313
*Selected References 315*
*Objective-Type Questions 315*
*Exercises 316*
*Answers to Objective-Type Questions 318*

https://hemanthrajhemu.github.io

**21. Firewalls** 319

21.1 Basics *319*

21.1.1 Firewall Functionality *319*

21.1.2 Policies and Access Control Lists *320*

21.1.3 Firewall Types *321*

21.2 Practical Issues *323*

21.2.1 Placement of Firewalls *323*

21.2.2 Firewall Configuration *325*

21.3 Personal Firewalls: A Case Study *326*

21.3.1 Chains and Tables *326*

21.3.2 Commands *327*

*Selected References 330*

*Objective-Type Questions 330*

*Exercises 330*

*Answers to Objective-Type Questions 331*

**22. Intrusion Prevention and Detection** 332

22.1 Introduction *332*

22.2 Prevention Versus Detection *333*

22.2.1 Prevention *333*

22.2.2 Detection *333*

22.2.3 Case Study: Unauthorized User Logins *334*

22.3 Types of Intrusion Detection Systems *335*

22.3.1 Anomaly versus Signature-Based IDS *335*

22.3.2 Host-based versus Network-based IDS *336*

22.4 DDoS Attack Prevention/Detection *337*

22.4.1 DDoS Prevention *337*

22.4.2 DDoS Detection *339*

22.4.3 IP Traceback *343*

22.5 Malware Defence *347*

22.5.1 Worm Defence *347*

22.5.2 Worm Signature Extraction *348*

22.5.3 Virus Detection *351*

*Selected References 352*

*Objective-Type Questions 353*

*Exercises 354*

*Answers to Objective-Type Questions 355*

**23. RFIDs and E-Passports** 356

23.1 RFID Basics *356*

23.2 Applications *357*

23.3 Security Issues *358*

23.4 Generation 2 Tags: A Case Study *359*

23.4.1 Features and Resources *359*

23.4.2 Operations *360*

https://hemanthrajhemu.github.io

Chapter 15

# IEEE 802.11 Wireless LAN Security

Wireless networks present formidable challenges in the area of security. The open nature of such networks makes it relatively easy to sniff packets or even modify and inject malicious packets into the network. The ease with which such attacks are launched necessitates careful design and deployment of security protocols for wireless networks.

In this chapter, we focus on wireless local area networks (WLANs). In particular, we examine the provision for security in *IEEE 802.11* networks. Authentication and key management in 802.11*i* are dealt with in Section 15.2, while the provision for confidentiality and integrity are dealt with in Section 15.3. We also highlight flaws in the widely used pre-802.11*i* security protocol, Wired Equivalent Privacy (WEP).

We first introduce WLANs and present a brief history of security protocols in WLANs.

## 15.1 BACKGROUND

There are two principal types of WLANs – ad hoc networks, where stations (possibly mobile) communicate directly with each other, and *infrastructure WLANs*, which use an *access point* (AP) (Fig. 15.1). In the latter, a station first sends a frame to an AP and the AP then delivers it to its final destination. The destination may be another wireless station. Alternatively, it may be a station on the wired network that the AP is connected to. The AP thus serves as a bridge between the WLAN and the existing wired network. In this chapter, we confine ourselves to infrastructure WLANs.

The wired network is often an ethernet LAN with an existing security infrastructure that includes an *authentication server* (AS). In many organizations, AAA (Authentication/Authorization/Accounting) functionality is provided by a RADIUS (Remote Authentication Dial in User Service) server. The challenge then is to develop protocols that seamlessly integrate the WLAN with the security infrastructure of the wired network.

A network of wireless stations associated with an AP is referred to as a *basic service set*. Such a network may be adequate for a home or small enterprise. However, in a large building or campus, all stations may not fall in the range of a single AP. It will be necessary to have several APs to cater to the stations dispersed over a set of buildings, for example. The APs in the different basic service sets are often connected over a wired network.

https://hemanthrajhemu.github.io

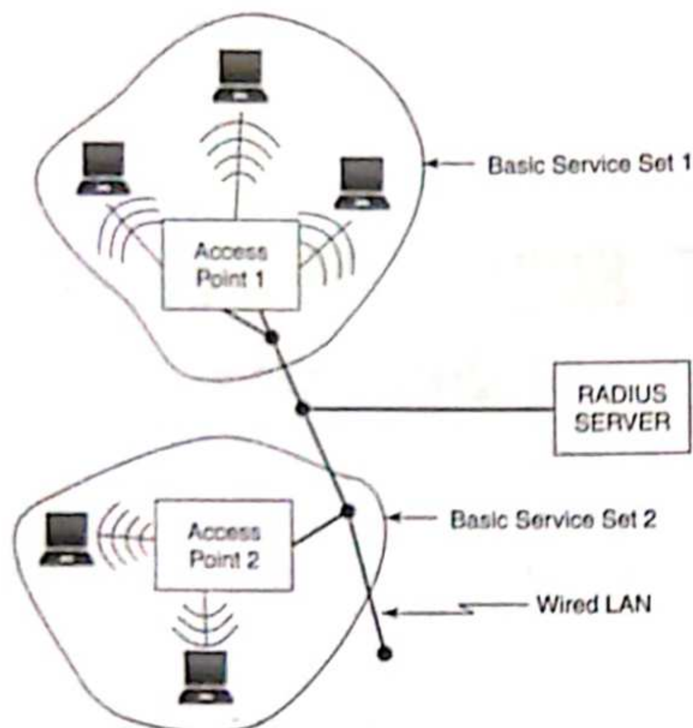



Figure 15.1 Infrastructure wireless LAN

The union of the basic service sets comprises an *extended service set* (ESS). As in wired LANs, each station and AP in the ESS is uniquely identified by a MAC address – a 48-bit quantity. In addition, each AP is also identified by an *SSID* (service set ID), which is a character string of length at most 32 characters.

A wireless station, on power-up, needs to first discover an AP within its range. This can be done by monitoring the wireless medium for a special kind of frame called a *Beacon*, which is periodically broadcast by the AP. The Beacon usually contains the SSID of the broadcasting AP. Alternatively, a station may send a *Probe Request* frame, which probes for APs within its range. An AP, on hearing such a request, responds with a *Probe Response* frame. Like the Beacon, the Probe Response frame contains the SSID of the AP and also information about its capabilities, supported data rates, etc.

To become part of the WLAN, a station will have to *associate* with an AP. At any point in time, a station can associate with only one AP. This relationship is recorded by the wired network. That way, if a frame from the wired network is destined for a wireless station, it can be dispatched to the AP with whom the wireless station is associated.

A station that wishes to associate with an AP sends it an *Associate Request* frame. The AP replies with an *Associate Response* frame if it accepts the request for associating with it. Before association, 802.11 requires the station to authenticate itself to the AP. This is addressed in detail in Section 15.2.

Authentication and other security functionalities in IEEE 802.11 LANs have seen much evolution in the last several years. The earliest protocol that incorporated security in WiFi was *WEP*. Designed to provide authentication/access control, data integrity, and confidentiality, it failed on all three counts. Explanations of the flaws in WEP are included in Section 15.3.1. WiFi Protected Access (WPA) was intended to fix the shortcomings of WEP without requiring new wireless network

https://hemanthrajhemu.github.io

cards. But WPA is not perfect – it too is susceptible to attacks on its cryptographic algorithms.

All the deficiencies in WEP have been addressed in the *IEEE 802.11i* standard ratified in June 2004. 802.11*i* (implemented in *WPA2*) makes sweeping changes in the way authentication, key management, integrity, and confidentiality are handled.

## 15.2 AUTHENTICATION

### 15.2.1 Pre-WEP Authentication

Early versions of 802.11 used naive approaches for authentication. For example, mere knowledge of the SSID sufficed for a station to be authenticated to the AP. However, an attacker could easily sniff the value of SSID from frames such as the beacon or probe response and then use it for authentication.

Another approach was to restrict admission to the WLAN by MAC address. The AP would maintain a list of MAC addresses (access control list) of stations permitted to join the WLAN. Here again, valid MAC addresses could be obtained by sniffing the wireless medium. The attacker could then modify his network card to spoof a valid MAC address. So, neither of these approaches helped.

### 15.2.2 Authentication in WEP

In WEP, the station authenticates itself to the AP using a challenge–response protocol Basically, the AP generates a challenge (nonce) and sends it to the station. The station encrypts the challenge and sends it to the AP. The stream cipher, RC4, is used for encryption. For this purpose, the station computes a keystream, which is a function of a 40-bit shared secret, *S*, and a 24-bit Initialization Vector (IV). The latter is pre-configured by the manufacturer of the wireless card. The challenge is then XORed with the keystream to create the response.

$$\text{RESPONSE} = \text{CHALLENGE} \oplus \text{KEYSTREAM}(S, IV)$$

The response together with the *IV* is sent by the station to the AP. In most implementations, the shared secret, *S*, is common to all stations authorized to use the WLAN.

All an attacker needs to do is to *monitor a challenge–response pair*. From this, he can compute the keystream. To authenticate himself to the AP, he needs to XOR the challenge from the AP with the computed keystream.

It may also be possible for an attacker to obtain *S* itself. By eavesdropping on several challenge–response pairs between the AP and various stations, an attacker could launch a ***dictionary attack*** and eventually obtain *S*.

One final note on WEP authentication is that there is no support for authenticating the AP to a station. This makes it possible for a rogue AP to masquerade as the authentic one opening the door to man-in-the-middle attacks.

### 15.2.3 Authentication and Key Agreement in 802.11i

#### *Authentication*

802.11i uses *IEEE 802.1x* – a protocol that supports ***authentication at the link layer***. Three entities are involved:

(i) Supplicant (the wireless station)
(ii) Authenticator (the AP in our case)
(iii) Authentication server

https://hemanthrajhemu.github.io

Different authentication mechanisms and message types are defined by IETF's *Extensible Authentication Protocol* (EAP). EAP is not really an authentication protocol but rather a framework upon which various authentication protocols may be supported. EAP exchanges are mostly comprised of requests and responses. For example, one party requests the ID of another party. The latter responds with its user_name or e-mail address. EAP also defines messages that may contain challenges and responses used in authentication protocols.

The AP broadcasts its *security capabilities* in the Beacon or Probe Response frames. The station uses the Associate Request frame to communicate its security capabilities. 802.11i authentication takes place after the station associates with an AP. This differs from earlier versions of 802.11 where authentication precedes association. Still, for backward compatibility with pre-WEP 802.11, the Authentication Request and Authentication Response messages that precede association are retained.

The generic authentication messages in IEEE 802.11i are shown in Fig. 15.2. The protocol used between the station and the AP is EAP but that used between the AP and the authentication server depends upon the specifics of the latter. For example, the authentication server is often a *RADIUS* server which uses its own message types and formats. (RADIUS stands for Remote Authentication Dial in User Service. It is a client–server protocol used for authentication, authorization, and accounting.)

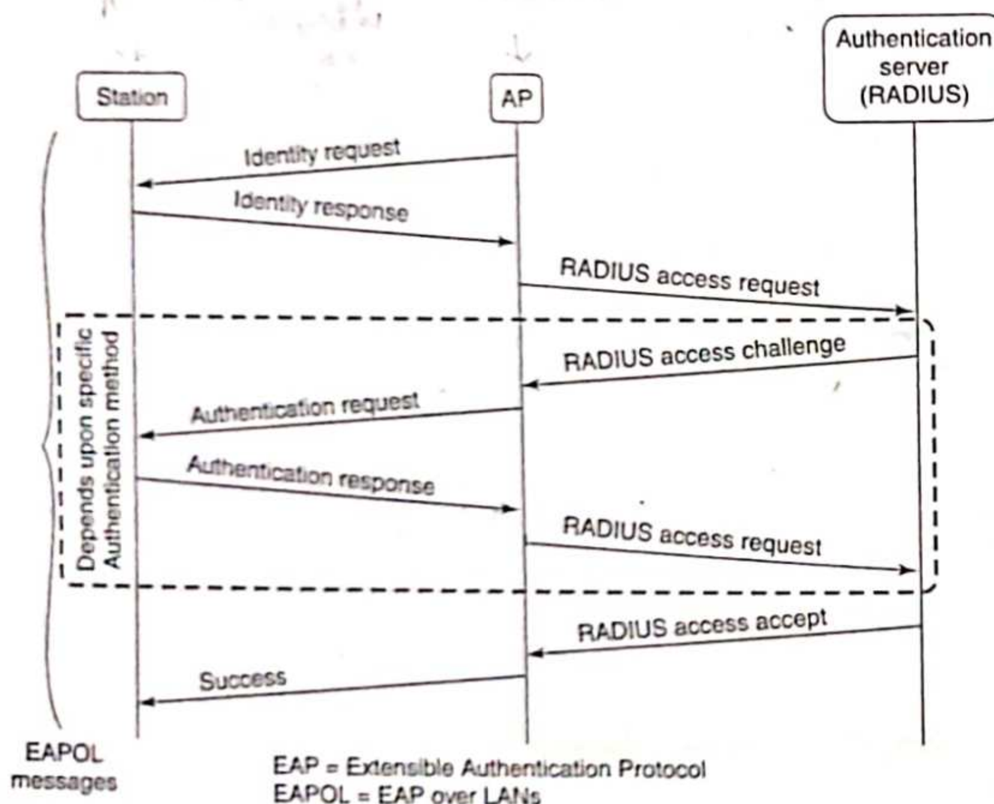



**Figure 15.2** *Authentication and master session key exchange in 802.11i*

The main authentication methods supported by EAP include the following:

EAP-MD5

EAP-TLS

https://hemanthrajhemu.github.io

EAP-TTLS
EAP-PEAP

*EAP-MD5* is the most basic of the EAP authentication methods. Here, the authentication server challenges the station to transmit the MD5 hash of the user's password. The station prompts the user to type his/her password. It then computes the hash of the password and sends this across. This method is insecure since an attacker could eavesdrop on such a message exchange and then replay the hashed password thus impersonating the owner of the password. Also, this method does not support authentication of the AP to the station.

*EAP-TLS* is based on the SSL/TLS protocol discussed in Chapter 14. Of all the EAP methods, it is the most secure and provides mutual authentication and agreement on a master session key. It requires the AP as well as the user (station) to have digital certificates. It is relatively straightforward to equip each AP with a digital certificate and a corresponding private key but extending the PKI to each user of the WLAN may not be feasible.

*EAP-TTLS* (tunnelled TLS) requires certificates only at the AP end. The AP authenticates itself to the station and both sides construct a secure tunnel between themselves. Over this secure tunnel, the station authenticates itself to the AP. The station could transmit attribute–value pairs such as

user_name = ramesh
password = 4rP#mNaS&7

Note that the station really authenticates itself to the RADIUS server – the AP merely forwards the authentication information (such as user_name and password in the above case) to the RADIUS server. Thus, the existing "legacy" security infrastructure of the organization can be leveraged to authenticate wireless clients.

*Protected EAP (PEAP)*, proposed by Microsoft, Cisco, and RSA Security, is very similar to EAP-TTLS. In PEAP, the secure tunnel is used to start a second EAP exchange wherein the station authenticates itself to the authentication server.

The enhanced security offered by EAP-TLS, EAP-TTLS, and PEAP does, however, come at a steep price in performance measured by the message and computational overheads incurred during authentication.

### *Key Hierarchy*

There are two types of keys used in WLANs. The first are *pairwise keys* used to protect traffic between a station and an AP. The second type of key is the *group key* intended to protect broadcast or multicast traffic between an AP and multiple stations. We describe the hierarchy of 802.11i keys next.

The root of the key hierarchy is the *Pairwise Master Key* (PMK). This is obtained in one of two ways. The station and the authentication server may agree on a Master Session Key (MSK) as part of the authentication procedure described earlier. The authentication server then communicates this key to the AP. The AP and station then derive the PMK from the MSK.

An alternative to computing a fresh PMK for each session is the Pre-Shared Key (PSK), which is used as the PMK. One possibility is for each station to be configured with the PSK. While the PSK approach is easier to administer, it is also much less secure than generating fresh master keys for each new session.

The 256-bit PMK is used to derive a 384-bit *Pairwise Transient Key* (PTK). The PTK is a pseudo-random function of the PMK, two nonces chosen by the AP, and the station and their MAC addresses. By deriving the PTK in this fashion, key refreshing can take place without the overhead

https://hemanthrajhemu.github.io

of negotiating a new PMK. Instead, the old PMK with two fresh nonces are all that is needed to refresh the PTK.

Three 128-bit chunks are extracted from the 384-bit PTK for the following purposes:

- A *Temporal Key* (TK) is used for both encryption and integrity protection of data between the AP and the station.
- A *Key Confirmation Key* (KCK) is used to integrity-protect some of the messages in the four-way handshake discussed next. Integrity protection is supported by a MAC computed as a function of the message and the KCK.
- A *Key Encryption Key* (KEK) is used to encrypt the message containing the group key.

The key hierarchy in 802.11i is summarized in Fig. 15.3.

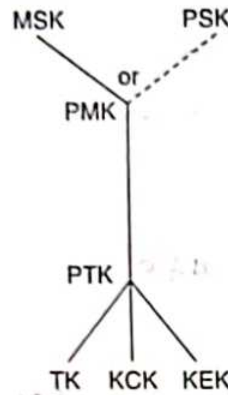



**Figure 15.3** *Key hierarchy in 802.11i*

### *Four-way Handshake*

The main goals of the four-way handshake are to

(a) derive the PTK from the PMK,

(b) verify the cipher suites communicated in the Beacon and Associate Request Frames and

(c) communicate the group keys from the AP to the station.

Figure 15.4 shows the messages comprising the four-way handshake.

1. The AP first sends a nonce, $N_A$, to the station.
2. The station chooses a nonce, $N_S$. The station computes the PTK as follows

   $$PTK = \text{prf}\,(PMK, N_A, N_S, MAC_A, MAC_S) \quad \text{... (15.1)}$$

   The PTK is a pseudo-random function (prf) of the PMK, the MAC addresses of the station and AP and nonces contributed by the station and the AP. The two nonces help prevent replay attacks. As mentioned earlier, three 128-bit keys – TK, KCK, and KEK are extracted from the 384-bit PTK (Fig. 15.3).

   The station sends its nonce together with its choice of cipher suite to the AP. It uses the KCK to compute a message integrity check (MIC). Such protection thwarts a possible man-in-the-middle attack intended to replace cryptographic algorithms in the cipher suite for possibly weaker options (e.g., shorter key sizes).

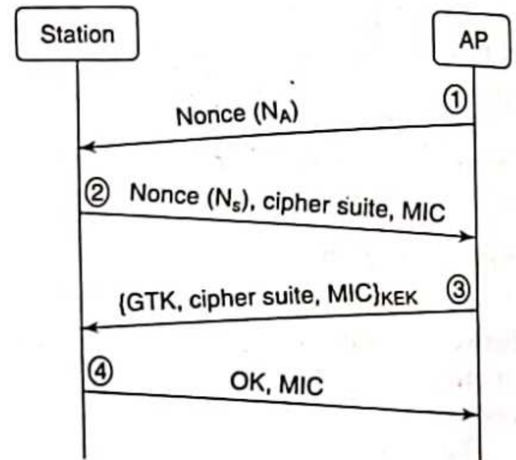



**Figure 15.4** *Four–way handshake in 802.11i*

   On receiving the message containing $N_S$ (Message 2), the AP computes the PTK from the above expression used by the station. It then extracts TK, KCK, and KEK. In addition, the AP verifies the integrity and source of Message 2 using the key, KCK.

3. Message 3 from the AP to the station contains the current *Group Transient Key* (GTK). This is the key used by the AP and all stations to integrity protect (and optionally encrypt) all

https://hemanthrajhemu.github.io

multicast or broadcast messages. Message 3 also contains the cipher suite chosen by the AP. The message is encrypted using the KEK and is integrity-protected using KCK.

4. Message 4 is an acknowledgement from the station that it has received the previous messages without error. It is a signal to the AP that henceforth all messages will be integrity-protected and encrypted with the TK.

## 15.3 CONFIDENTIALITY AND INTEGRITY

We first study how and why WEP failed to provide the desired level of security. We then study how 802.11i provides message confidentiality and integrity

### 15.3.1 Data Protection in WEP

WEP was designed to provide message confidentiality, integrity, and access control but it failed on all three counts. In Section 15.2.2, we exposed flaws in WEP's authentication mechanism. In this section, we show how plaintext can be recovered and messages can be modified due to flawed design decisions in WEP. There are many lessons to be learned from WEP – the most important being how not to design protocols for security.

#### *WEP Encryption and Integrity Checking*

WEP uses the stream cipher, RC4, for encrypting messages. It generates a pseudo-random keystream, *KS*, which is a function of a static secret shared between the two communicating parties. In order to have *KS* vary from message to message, a random *per-message initialization vector*, *IV*, is also used to generate *KS*. Early implementations of WEP used a 40-bit secret, *S*, concatenated with a 24-bit *IV* to create, in effect, a "64-bit key."

KS is ⊕'ed with the plaintext, $\mathcal{P}$, to obtain the ciphertext, C or

$$C = \mathcal{P} \oplus KS(S, IV) \tag{15.2}$$

The plaintext, $\mathcal{P}$, above includes the message to be sent and also an integrity check. The latter is a *32-bit CRC checksum* computed on the message. As shown in Fig. 15.5, the *IV* chosen by the sender is included in each frame.

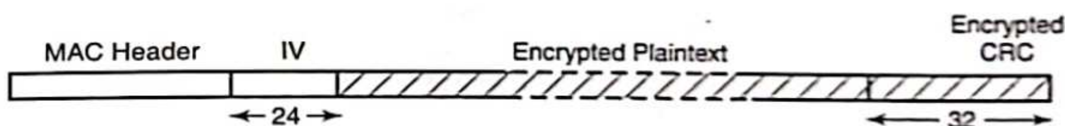



**Figure 15.5** *WEP frame*

To decrypt the message, the receiver generates *KS* from the shared secret, *S*, and the *IV* retrieved from the received frame. It recovers the plaintext from the following equation:

$$\mathcal{P} = C \oplus KS(S, IV) \tag{15.3}$$

#### *Known Plaintext Attack*

The first problem with WEP is the possibility of *keystream re-use*. Since the *IV* is 24 bits in length, there are only $2^{24}$ distinct keystreams that could be constructed given a secret *S*. Suppose an attacker finds two frames which were encrypted using the *same IV*. (Note that the per-message *IV* is sent in the clear.) Let their ciphertexts be C and C'. Let the corresponding plaintexts be P and P'. Using

https://hemanthrajhemu.github.io

Eq. (15.2), it follows that:

$$\mathcal{P} \oplus \mathcal{P}' = C \oplus C'$$

So

$$\mathcal{P}' = \mathcal{P} \oplus C \oplus C'$$

Thus, knowing C, C', and $\mathcal{P}$, we can obtain $\mathcal{P}'$ (a known plaintext attack). Note that even if a part of $\mathcal{P}$ is known (or can be guessed), then the corresponding bits in $\mathcal{P}'$ can be deduced.

How *frequently* would an attacker encounter frames XORed with the *same keystream*? A simple calculation reveals that an attacker, who eavesdrops on frames transmitted in a WLAN, will detect frames using the same keystream in less than 4 hours on a 10-Mbps channel used continually. This assumes an average frame size of 1000 bytes. It also assumes that the sender increments the *IV* after transmitting each frame as occurs in some implementations. If a WLAN interface card randomly chooses IVs for each frame, the detection time decreases further (see Exercise 15.3). In most WEP implementations, the same key is shared by all stations and the AP. This makes it even easier for the attacker to detect collisions in the *IV*.

### *Message Modification*

Consider an attacker who wishes to modify a message sent by a legitimate user. Let the sender's plaintext (not including the CRC checksum) be $M_1\ F\ M_2$ where $M_1$, $F$, and $M_2$ are each binary strings. The attacker wishes to substitute the substring, $F$, with another substring, $F'$, so that the decrypted message seen by the receiver is $M_1\ F'\ M_2$. The attacker does not need to know the values, $M_1$ and $M_2$. However, we assume that he knows $F$ and $F'$. Ideally, the *message integrity check* should detect any modification to an existing message. Can the attacker modify the message (including checksum) in such a way so that the modification is undetected at the receiver end?

For the above plaintext, the ciphertext computed by the sender is

$$((M_1\ F\ M_2) \parallel CRC(M_1\ F\ M_2)) \oplus KS$$

The attacker intercepts the ciphertext and performs the following operations:

1. He first constructs the string, $0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|}$. Here, $0^{|M_1|}$ is a string of $|M_1|$ zeros where $|x|$ is the length of the substring $x$.
2. He then computes the CRC on this string, $CRC(0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|})$.
3. He finally XORs the original ciphertext with $0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|} \parallel CRC(0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|})$.

These computations yield

$$\begin{aligned}
&((M_1\ F\ M_2) \parallel CRC(M_1\ F\ M_2)) \\
&\quad \oplus KS \\
&\quad \oplus ((0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|} \parallel CRC(0^{|M_1|} \parallel (F \oplus F') \parallel 0^{|M_2|})) \\
&\quad = ((M_1 \oplus 0^{|M_1|}) \parallel (F \oplus (F \oplus F')) \parallel (M_2 \oplus 0^{|M_2|})) \\
&\quad \parallel (CRC(M_1\ F\ M_2) \oplus CRC(0^{|M_1|} \parallel F \oplus F' \parallel 0^{|M_2|})) \\
&\quad \oplus KS \\
&\quad = ((M_1\ F'\ M_2) \parallel CRC(M_1\ F'\ M_2)) \oplus KS
\end{aligned}$$

The last step follows from the fact that the CRC is a linear operation, i.e.,

$$CRC(m_1 \oplus m_2) = CRC(m_1) \oplus CRC(m_2)$$

The receiver, on decrypting the ciphertext, obtains

$$(M_1\ F'\ M_2) \parallel CRC(M_1\ F'\ M_2)$$

https://hemanthrajhemu.github.io

The modified message has a valid CRC and so passes the integrity check at the receiver. Hence, the receiver accepts the message, unaware that it has been modified by an attacker.

### 15.3.2 Data Protection in TKIP and CCMP

Earlier versions of 802.11 used the stream cipher RC4. In addition to the attacks described in the previous section, there are many more attacks on RC4 as used in WEP. A well-known example is the FMS attack named after Fluhrer, Mantin, and Shamir [FLUH01]. By collecting a *sufficient number of frames* "over the air" bearing specific IVs, the encryption key used in WEP can be deduced.

The weaknesses in RC4 prompted the IEEE 802.11i standards committees to seek a replacement. The new standard for secret key cryptography, AES was an obvious choice. However, the use of AES necessitates use of new hardware. A more economical alternative is a firmware upgrade which retains existing 802.11 hardware (including RC4) but with many changes in overall design that eliminate the vulnerabilities that plague WEP. Thus was born *Wireless Protected Access* (WPA). The technical name for WPA is *Temporal Key Integrity Protocol* (TKIP).

The implementation of 802.11i that uses AES is referred to as *WPA-2*. Its technical name is *Counter Mode with CBC MAC Protocol* (CCMP). We study TKIP and CCMP in the next two subsections.

#### *TKIP*

The RC4 encryption key is used to generate the RC4 keystream. One of the most serious flaws in WEP was the way in which the RC4 key itself was created. Recall that the WEP encryption key is a simple concatenation of the 24-bit IV and the 40-bit shared secret to provide effectively a 64-bit key. (The next version used a 24-bit IV and a 104-bit shared secret.) The problem is that the variable part of the WEP key is too small, so the per-frame keystream repeats frequently.

By contrast, the encryption key in TKIP is 128 bits. More importantly, the method used to generate it is much more sophisticated. Basically, the designers of TKIP made sure that there was much randomness in most of the 128 bits of the key and that the probability of keystream collisions was negligible.

TKIP generates a random and different encryption key for each frame sent. It employs a process called *two-phase key mixing* (Fig. 15.6). The inputs to this process are the 128-bit temporal key, TK, computed as part of the four-way handshake (discussed in Section 15.2.3.2), the sender's MAC address and the four most significant bytes of a 48-bit *frame sequence counter*. The randomizing capabilities of the key mixing function and the large size of the key space virtually guarantee that "keystream collisions" never occur. Thus, known plaintext attacks that could be successfully launched on WEP have no chance of success with TKIP.

The sequence counter is incremented for each frame sent. It is also carried in the header of each frame. It is extracted by the receiver and used to compute the RC4 key for decryption. Both sender and receiver keep track of the sequence number of the last frame sent/received. The receiver accepts a fresh frame only if the frame's sequence number is greater than that of the previous frame received from the same sender. This helps protect the receiver from *replay attacks*.

Figure 15.6 shows the two phases used in generating the RC4 key. Two pseudo-random functions (PRF1 and PRF2) are employed in the two phases. The least significant 16 bits of the sequence counter are inputs to PRF2. So, the output of PRF2 changes for each frame sent. The 32 most significant bits of the sequence counter are input to PRF1. This input changes after every $2^{16}$ = 65,536 frames sent. Hence, PRF1 is executed very rarely and overall computation time is saved.

https://hemanthrajhemu.github.io

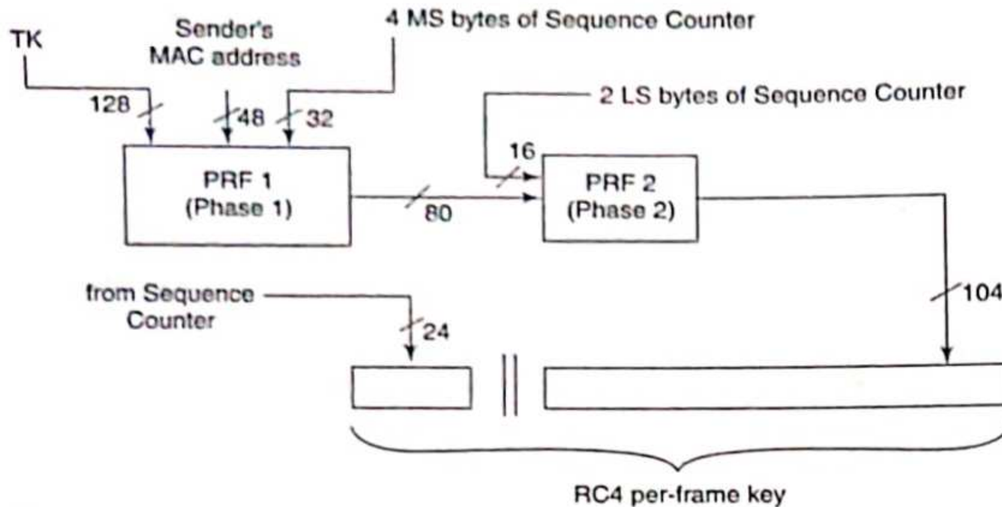



**Figure 15.6** *Two-phase key mixing in TKIP*

One of the most serious vulnerabilities in WEP was the use of the CRC checksum as an integrity check. The 64-bit message integrity check in TKIP, called MIC or Michael, is a great improvement. Unlike the CRC, MIC is non-linear, i.e.,

$$\text{MIC}(m_1 \oplus m_2) \neq \text{MIC}(m_1) \oplus \text{MIC}(m_2)$$

MIC is computed as a function of the data in the frame and also some fields in the MAC header such as the source and destination addresses. It also uses as input a key derived from the PTK which was computed during the four-way handshake.

Due to design constraints on WEP cards, MIC's implementation uses simple logical functions, shifts, etc. Hence, it is not as secure as a keyed cryptographic hash. On the other hand, it is much better compared to the CRC checksum used in WEP.

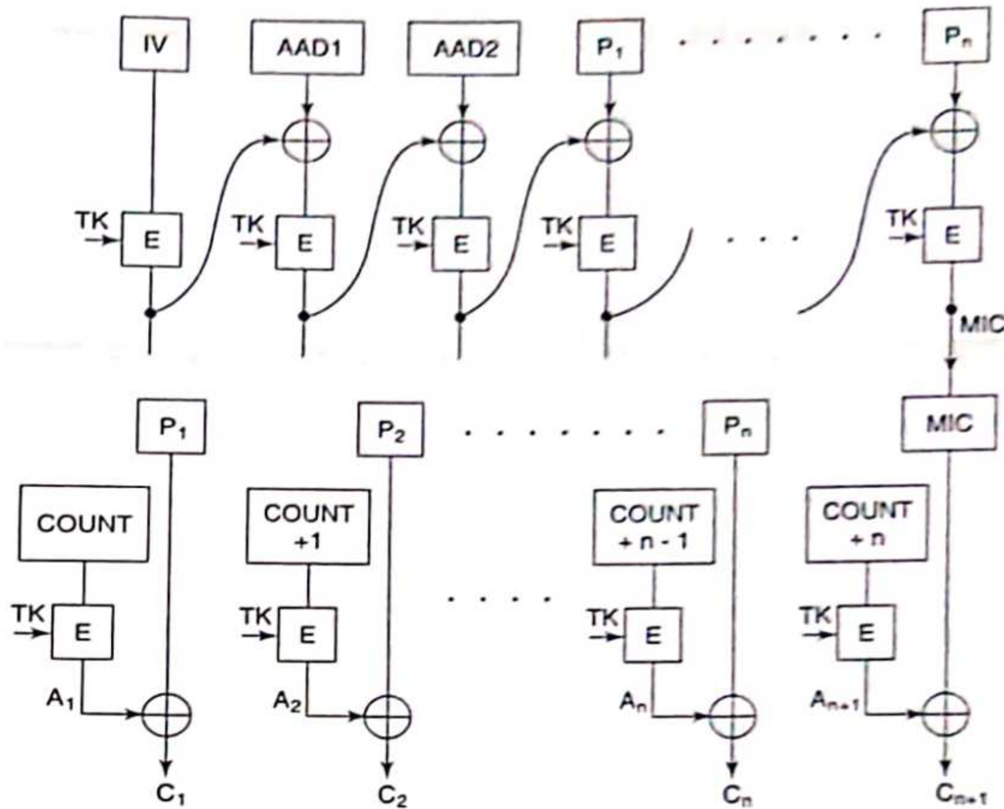
### *CCMP*

The most significant feature of CCMP vis-a-vis WEP and TKIP is the use of AES for both encryption and for providing message source authentication/integrity. Unlike TKIP, the same key is used for encryption and MAC computation. This is the 128-bit temporal key, TK, that was introduced as part of the 802.11i key hierarchy (see Section 15.2.3.2). Because AES is a block cipher, there is no need to re-compute a fresh key for each frame as was the case in WEP and TKIP.

As in the case of TKIP, a 48-bit sequence counter is initialized when a session is started between a sender and receiver. In CCMP terminology, this count is referred to as a *packet number* (PN). The count is maintained at both sender and receiver ends. The PN is included in a special *CCMP header* field in a CCMP frame. The PN is incremented by the sender after each frame is sent. Upon receipt of a fresh frame in that session, the receiver compares the value of PN in the CCM header versus the value stored by it. If the former is less than the stored value, the frame is likely to be a replayed frame and is hence discarded.

The first task in preparing a frame for transmission is to compute a MIC. The scope of the MIC is the frame data and several immutable fields in the MAC header. The MIC is computed using *AES* in *Cipher Block Chaining (CBC) mode* with block size = 128 bits (Fig. 15.7). The key for

https://hemanthrajhemu.github.io




IV = Initialization Vector (includes 48-bit Packet Number)

AAD1, AAD2 = Additional Authentication Data (includes certain immutable fields of the MAC header)

COUNT is a function of the Packet Number

**Figure 15.7** *MAC generation and encryption in CCMP*

performing encryption in each stage of Fig. 15.7 is TK. The IV for the MIC computation is a "nonce," which includes the 48-bit PN. The second and third blocks used in the MIC computation are specific fields in the frame header such as the MAC addresses, sequence control, and frame type. Next, the blocks in the frame data are sequentially processed resulting in an 8-byte MIC.

The next step is encryption. The frame data and the MIC are concatenated and then encrypted using *AES in counter mode* (Fig. 15.7). Let $n$ be the total number of blocks in the frame body + MIC. The procedure for encrypting the $i$-th block is:

1. Compute $A_i = E_{TK}(PN + i * j)$. Here, PN is the packet number and $j$ is a constant known to both sender and receiver.
2. Compute $i$-th block of ciphertext = $A_i \oplus P_i$. Here, $P_i$ is the $i$-th block of plaintext.

The frame now includes two new fields – the CCMP header and the MIC. Upon receipt of the frame, the receiver reverses the operations performed by the sender. It performs decryption followed by MIC verification.

https://hemanthrajhemu.github.io

## SELECTED REFERENCES

[802.11STD] contains the technical details of the IEEE 802.11i standard. A thorough analysis of the 802.11i authentication and handshake appears in [HE04]. [BORI01] exposes a number of flaws in WEP communications such as unauthorized frame modification and frame insertion. [FLUH01] exposes weaknesses in the RC4 key scheduling algorithm (the Fluhrer, Mantin, Shamir attack). This is further studied in the exercises at the end of this chapter.

## OBJECTIVE-TYPE QUESTIONS

**15.1** The RADIUS server
- (a) is part of the wireless infrastructure within a typical organization
- (b) is part of the wired infrastructure within a typical organization
- (c) is responsible for authentication, authorization, and accounting (AAA functionality)
- (d) communicates fluently using EAPOL messages

**15.2** Which of the following is/are true of EAP authentication methods?
- (a) EAP-TLS requires both station and AP to have digital certificates
- (b) EAP-TTLS requires both station and AP to have digital certificates
- (c) EAP-MD5 is vulnerable to a replay attack
- (d) PEAP involves user authentication through passwords

**15.3** The Key Confirmation Key (KCK) is used to
- (a) integrity-protect data between the station and the AP
- (b) integrity-protect messages in the four-way handshake
- (c) encrypt data between the station and the AP
- (d) encrypt the message containing the Group Key

**15.4** The four-way handshake is used to
- (a) authenticate the station to the AP
- (b) authenticate the AP to the station
- (c) agree on a pairwise master key (PMK)
- (d) agree on a pairwise transient key (PTK)

**15.5** Which of the following is/are true regarding the IV used in WEP?
- (a) The IV is randomly chosen by the sender
- (b) The IV is mutually agreed upon by the sender and receiver
- (c) The IV is included in each frame transmitted
- (d) The IV is part of the 64-bit (or 128-bit) key used to compute a keystream

**15.6** Inputs to the key-mixing function that produces the per-frame encryption key in TKIP include
- (a) the PMK
- (b) a frame sequence number
- (c) the sender's MAC address
- (d) the receiver's IP address

**15.7** The message integrity check in CCMP is computed using
- (a) two-phase key mixing
- (b) AES in CBC mode
- (c) a keyed hash
- (d) a CRC

https://hemanthrajhemu.github.io

## EXERCISES

**15.1** Explain how each key in 802.11i was derived and where it is used.

**15.2** Can the four-way handshake in 802.11i be accomplished with fewer messages? If yes, show how. If no, explain why not.

**15.3** Assume that 24-bit initialization vectors are generated randomly in a WLAN using WEP. Calculate the number of frames an attacker should collect before the first occurrence of a repeated keystream with the following probability:
(a) 50% (b) 90%

**15.4** Consider any two attacks on WEP. Explain how these attacks are mitigated in TKIP and in CCMP.
Is 802.11i vulnerable to any of the attacks that plague WEP? Explain.

**15.5** Given the large number of problems with WEP, your organization wishes to migrate to either TKIP or CCMP. Identify the tradeoffs between migrating to CCMP versus migrating to TKIP.

**15.6** How would you detect the presence of a rogue station on the wireless LAN?
How would you detect the presence of a bogus AP on the wireless LAN?

**15.7** Do you have a WLAN on your campus or within your organization? Does it support any encryption at all? If so, does it use WEP or 802.11i? If the WLAN is unencrypted or uses WEP, you may wish to alert the system administrators and IT department of your organization regarding the risks such a network is exposed to.

**15.8** Is RC4 a problem wherever it is used? The discussion in Section 15.3.1 indicates that RC4 as used in WEP is vulnerable to attack. However, RC4 has been and continues to be used in the SSL protocol without a problem. Investigate why RC4 is a problem in WEP but not in SSL.

**15.9** Is RC4 the only problem? Suppose that RC4 encryption in WEP were replaced by AES (as in CCMP). Would the modified WEP be secure? Explain why or why not.

## ANSWERS TO OBJECTIVE-TYPE QUESTIONS

**15.1** (b)(c) **15.2** (a)(c)(d) **15.3** (b) **15.4** (d)
**15.5** (a)(sometimes true), (c)(d) **15.6** (b)(c) **15.7** (b)

https://hemanthrajhemu.github.io

# Chapter 19

# Viruses, Worms, and Other Malware

No book on Computer Security would be complete without a detailed study of malware. In this chapter, we study various kinds of malware including *worms*, *viruses*, *trojans*, and *bots*. We identify several novel features of malware and their many vectors of propagation. Some malware are activated by human intervention – the click of an infected e-mail attachment or a visit to a website hosting malicious code. On the other hand, Internet scanning worms spread automatically without human intervention.

Recently, another type of malware called bots has received much press. An army of compromised computers or bots under the control of a single master is referred to as a *botnet*. Botnets have been used for sending spam mail, launching Distributed Denial of Service (DDoS) attacks, and in identity theft. We study botnets in the last section of this chapter after investigating different kinds of viruses and worms.

## 19.1 PRELIMINARIES

Of all malware types, worms and viruses have probably received the maximum attention. They both refer to *malware* that *replicate* themselves. A virus latches itself on to an executable *file* or program while a worm is typically a stand-alone *program*. A virus infects a file and uses it as a host from which to infect other files while a worm spreads from one computer to another.

The term "computer virus" is of earlier coinage – its earliest usage was in the context of malware that resided in the boot sector of a floppy disk. This is usually the first sector on the disk and contains code for bootstrapping, i.e., loading the operating system from disk. Since boot floppies were often exchanged and copied by people, *boot viruses* spread within user communities.

The rate of spread of viruses, in general, is relatively slow. For example, the rate of spread of the boot sector virus was limited by the rate at which boot floppies were exchanged. By contrast, worms use the network (the Internet, cellphone networks, etc.) to propagate. Many of the recent worms propagate at *extraordinary speeds*. Some of them are activated and propagate without human intervention. They may spread half way around the globe before systems administrators could be alerted to take remedial action.

There are no universally agreed upon definitions for worm and virus. For example, "e-mail worms" straddle the grey area between worms and viruses. On one hand, like the more typical worms, they propagate over networks such as the Internet. On the other hand, they share virus

https://hemanthrajhemu.github.io

characteristics in that many of them are activated by *human action* such as clicking on an attachment.

A *trojan horse* (or simply trojan) is a program with a malicious component masquerading as a useful piece of software. Unlike viruses or worms, trojans do not replicate. A trojan is typically activated by action on the part of the victim.

Trojans can enter a system in several ways – through e-mail attachments, through file-sharing software, from a website, or through cellphone downloads. For example, a user may download an exotic calculator program from a website. While the calculator may function as expected, it may also perform a more insidious task such as reading a file on disk and setting up a network connection to a hacker over which it communicates the file.

Section 19.2 highlights developments in virus/worm design over the last few years. We then introduce four categories of malware – Internet scanning worms, topological worms (including e-mail worms), web worms, and mobile malware. Sections 19.3–19.6 are, respectively, dedicated to each of these malware types. We also present case studies of relevant worms. Finally, we study botnets in Section 19.7 including a case study of the Storm botnet.

## 19.2 VIRUS AND WORM FEATURES

### 19.2.1 Virus Characteristics

When a virus-infected program is run, the virus code is executed first. One of the first tasks of virus code is to seek other programs not yet infected and then pass on the infection to one or more of them. A truly malicious virus may then perform actions such as deleting certain files. An innocuous virus may attempt something benign like printing a "hello world" message. Execution of the virus code is usually followed by execution of the host's original program.

All the virus code need not be located at the start of the infected file. In some cases, virus code is both prepended and appended to the host file. Virus code could be split into several segments and interspersed throughout the infected file using JUMP statements at the end of each virus segment. In most of these cases, the size of the infected program is larger than the original host program. This helps anti-virus software to detect infected code.

To evade detection, some viruses modify the *file service interrupt handler* that returns attributes of files. By so doing, the service handler may be programmed to return the uninfected length of the file. Another technique is to use *compression* so that the length of an infected file remains the same as the length of its original version. The virus writer includes a compression routine in the viral code. To infect another file, the virus first compresses that file and then prepends the virus code to the compressed file. The infected file must be uncompressed just prior to execution.

One of the characteristic features of many viruses is the set of system calls they make. System calls are used by application programs to request services of the operating system. They are made to read/write files, spawn new processes, establish TCP connections, etc. Some viruses make calls to copy their own code to other files, create/modify entries in the Windows registry, or search for e-mail. Such "suspicious" calls are often used to distinguish malicious from benign code.

In the next section, we highlight many novel features of worms such as polymorphism and metamorphism. It should be understood, however, that some of the worm features may also be exhibited by viruses and other malware.

https://hemanthrajhemu.github.io

### 19.2.2 Worm Characteristics

#### *Classes and Features*

Worms are most commonly classified based on their *vector of propagation*. The main categories include

- Internet scanning worms
- E-mail worms
- P2P worms
- Web worms and
- Mobile worms

We describe each of these categories in the next few sections. We exclude from this discussion many features of bots which are exclusively covered in Section 19.7.

Over the years, worm writers have brought many ingenious techniques to worm design. Table 19.1 lists selected malware including innovative aspects of each. In the remainder of this section, we present a limited categorization of "achievements" in worm design and explore sample features in each category.

**Table 19.1** *Selected malware with their innovative features*

| Malware name | Year unleashed | Type of malware/ vectors of propagation | Claim to fame |
|---|---|---|---|
| *Code Red* | July 2001 | Internet scanning | First worm that spread rapidly causing billions of dollars in damage |
| *Nimda* | September 2001 | E-Mail, HTTP, file sharing | One of the first worms to use multiple vectors of propagation |
| *Slammer* | January 2003 | Internet scanning | First Internet scanning worm to spread through UDP, not TCP. Hence much faster – infected 90% of vulnerable hosts in just 10 min |
| *Sobig* | August 2003 | E-mail | Worm updated itself at specific points in time. The updated worm code was obtained from specific URLs |
| *Witty* | March 2004 | Internet scanning | One of the first worms to carry a destructive payload, which wiped out part of its victims' disks |
| *Cabir* | June 2004 | Bluetooth | One of the first worms to target cellphones |
| *Santy* | December 2004 | Web | One of the first worms to use a web search engine to locate new targets. Exploited a generic vulnerability in PHP |
| *Commwar* | March 2005 | Bluetooth, MMS | One of the first mobile worms to use two vectors of propagation |
| *Samy* | October 2005 | Web | In 24 hours, infected over 1 million users' profiles on MYSpace – a social networking site |
| *Storm* | January 2007 | Bot. E-mail, infected websites | Multiple infection stages, URLs for secondary infections communicated via encrypted links in P2P network |
| *Conficker* | September 2008 | Bot. Random scans, USB drives, network file systems | Dynamically generated URLs for code updates. Updates digitally signed by botnet controller |

https://hemanthrajhemu.github.io

### *Enhanced Targeting*

The most important attribute of a worm is that it spreads its infection to other computers. But how does a worm know who to target next?

Many target selection strategies have been proposed and implemented. Worms that spread through e-mail, for example, have an easy way to figure out their targets. All they need to do is look into their victim's mailbox or *e-mail address book* to find a set of targets. A mobile worm obtains phone numbers of its potential victims from the phone book in the cellphone hosting the worm. Some web worms use search engines to harvest URLs of potentially vulnerable targets.

Internet scanning worms, on the other hand, scan the IP address space for vulnerable machines. The most straightforward approach is *random scanning* – choosing IP addresses at random. This was adopted by Code Red Version-I. However, Code Red Version-II adopted *localized scanning*. Over 80% of the time, it attempted to connect to victims with whom it shared the network address (most significant 8 or 16 bits of the IP address). This strategy was more successful since hosts in the same network are likely to be closer and be running the same software.

Worms like Nimda, unleashed in September 2001, spread aggressively thanks to its *five different vectors of propagation*. Propagation through HTTP and e-mail were particularly successful in penetrating the perimeter of the enterprise. Once inside, it exploited the Windows file-sharing feature to spread within the enterprise.

### *Enhanced Speed*

To enhance the infection rate, some worms are designed to spawn *multiple threads*. Each thread is responsible for setting up connections to a different subset of hosts, thus increasing the rate at which infection is spread.

Some worms reduce infection latency by targeting a buffer overflow vulnerability on an application that employs *UDP* rather than TCP. TCP connection establishment involves a three-way handshake and is time-consuming. UDP, by contrast, is connectionless. This sharply reduces infection latency.

A steep increase in the number of infected machines at the very outset of a worm epidemic has a multiplicative effect on spreading rate. For this purpose, the attacker could create one or more *hit-lists* carrying addresses of several thousand vulnerable machines. The first worms to be let loose could carry one such list. As a worm infects each new machine, it splits its list between itself and the machine it has just infected. Given that most of the machines on the hit-lists are vulnerable, the worm spreads rapidly during the initial stage of the epidemic. Thereafter, the infected machines could spread the infection using random scanning or some other spreading method.

### *Enhanced Capabilities*

Most worms (and viruses) have unique and distinct *signatures* – a pattern of bits, usually assembly language code, which appears in all instances of the worm. Worm and virus signatures are the key to detecting them. However, there are sophisticated code obfuscation techniques to evade detection. One such technique is the use of encryption for disguising worm code. Different instances of the worm may use different keys for encryption. Thus, they might fail to match any existing worm signatures. Such worms are said to be *polymorphic*.

https://hemanthrajhemu.github.io

A polymorphic worm would have to be decrypted before being executed. This suggests that a decryptor routine "in the clear" would have to be part of the worm code. Decryptors themselves may be very simple, involving XOR operations or trivial shift-based substitutions. However, detecting a worm on the assumption that the decryptor routine is invariant would not always succeed. As it turns out, there are several freely available "mutation engines" that can create hundreds of variations in the code of a given decryptor that can be used by wannabe virus writers.

While encryption is one way of disguising malware, another is the creation of several code "versions" that are superficially different but functionally identical. Tricks to create multiple versions include

- use of dummy instructions (NOPs, ADDing 0 to a register, etc.)
- use of extraneous operands (variables)
- changing the flow of control without disturbing the existing logic

Figure 19.1 shows two versions of assembly code that look different but perform the same function. The second version is inefficient with spurious instructions. The second version also has a spurious branch instruction to confuse worm code detection software that relies on control flow analysis. Worms that have multiple such versions with or without relying on encryption are referred to as *metamorphic* worms.

*Assembly Pseudo-code*

```
if R5 > 0
    R4 ← R1 + R2
else
    R4 ← 4 × R2 + 3 × R3
```

*Assembly Code: Version 1*

```
        CMP   R5, #0
        BLE   Second
First:  ADD   R1, R2, R4
        BRA   Finish
Second: ADD   R2, R3, R4
        SLA   R4, #2, R4
        SUB   R4, R3, R4
Finish: . . .
```

*Assembly Code: Version 2*

```
        XOR   R6, R6, 0
        ADD   R5, R6, R6
        SUB   R6, #0, R6
        BG    First
Second: ADD   R2, R2, R4
        ADD   R4, R4, R4
        ADD   R4, R3, R4
        ADD   R4, R3, R4
        ADD   R4, R3, R4
        BNE   Finish
        CMP   R4, R4
        BE    Finish
First:  ADD   R1, R2, R4
Finish: . . .
```

**Figure 19.1** *Polymorphic assembly language versions of same pseudo-code*

Some worms need to be *time-aware*. They obtain the current date and time from a network time protocol (NTP) server and can initiate specific actions at specified points of time. This capability

https://hemanthrajhemu.github.io

allows worms to remain dormant for extended periods of time and then strike in a concerted fashion by, for example, launching a denial of service attack at the same time. Some worms can *update themselves* by downloading code from given URLs. Alternatively, they may access a URL which in turn provides a set of URLs from which updated worm code may be downloaded. Finally, early e-mail worms used the host's e-mail services to spread infection while some more recent worms have been designed with a built-in SMTP engine which they use to send mail.

#### *Enhanced Destructive Power*

It is estimated that worms such as Code Red and Nimda caused *billions of dollars in damage*. How are these costs estimated? Analysts estimate costs based on lost productivity, clean-up costs, and system downtime which affects business and revenues. Fast-spreading worms also caused severe network congestion problems disrupting normal Internet traffic and contributing to system downtime.

Nevertheless, most worms thus far have been relatively benign. Some worms contributed attack packets to a DDoS attack or caused website defacement. The *Witty* worm which appeared in March 2004, however, was qualitatively different. It was the first worm to carry a destructive payload. It deleted a random section of the victim's hard disk leading to a system crash. It is not hard to imagine worms carrying far more destructive payloads that could crash many more systems.

The harm caused by a worm is not just destructive power measured by downtime, lost productivity, and system crashes. There are more sinister and subtle goals such as the stealing of sensitive personal and corporate information, which could remain undetected.

In the next couple of sections, we study different classes of worms and attempt to answer the following questions:

- What vulnerabilities does a worm exploit?
- How does it select its targets?
- How is it carried?
- How does it infect other hosts?
- How is it activated?
- How fast does it spread?
- What harm does it cause?
- Are there preventive/detective measures against such attacks?
- If so, how effective are these measures?

## 19.3 INTERNET SCANNING WORMS

One characteristic of Internet scanning worms is that they are *self-activated*. The ability to spread without human intervention distinguishes them from most types of e-mail, P2P, and web worms.

This category of worms is so called because they scan the Internet looking for vulnerable machines. The vulnerability could be a buffer overflow problem in a commonly used service provided by a particular version of an OS. The worm communicates with and delivers its malicious payload to the victim using standard transport protocols such as *TCP* or *UDP*. Once installed on the victim, it could erase local files, steal secrets, or deface webpages, but above all it seeks new victims to infect. The principal goal of these worms seems to be to spread as rapidly as possible and, as we will soon see, many of them have been eminently successful.

https://hemanthrajhemu.github.io

### 19.3.1 Case Studies: Code Red and Slammer

#### *Code Red*

One of the best known examples of Internet scanning worms is the Code Red worm, which received a lot of press because of the considerable damage it wrought (upward of US$ 2.5 billion and more than any at the time). We first describe this worm and then contrast it with the Slammer worm.

It all started on June 18, 2001, when a *buffer overflow vulnerability* was discovered in the Microsoft *IIS Web Server*. A patch for this vulnerability was developed a few days later. It is estimated that there were several million IIS servers in active deployment. Even assuming that a large percentage of these were patched, that still left plenty of room for the spread of the worm, which was unleashed on July 12, 2001. The worm itself was carried in HTTP request messages targeted at IIS servers.

The first version of the worm used a random number generator to generate new addresses of machines to infect. However, the same seed was used for the random number generator in every instance of the worm – this resulted in the same machines being infected over and over again. However, on July 19, 2001, a variant of Code Red I was launched wherein a *random seed* was generated in a worm. This had a dramatic effect on worm propagation – about 360,000 machines were infected in just 14 hours after the launch of this variant.

The infection phase continued until July 20th at which point the worms moved to attack phase. At this point until July 28th, they launched a denial of service attack on www.whitehouse.gov. They also defaced webpages with the phrase "Hacked by Chinese."

A few weeks after the launch of Code Red I, Code Red II was released. Code Red II installed a backdoor providing an attacker remote, administrator-level access to the victim. Unlike Code Red I, which was memory-resident and was destroyed by re-booting the system, Code Red II *persisted on disk*. It spawned several hundred threads – each thread sent probe packets to several targets. Unlike Code Red I, only about 12% of the addresses it generated were random, the rest were biased towards machines within its own subnet.

#### *Slammer*

The SQL Slammer was launched on 25 January, 2003, and targeted a *buffer overflow vulnerability* on the Microsoft *SQL server 2000*. The worm sent packets on UDP port 1434 – the database software's resolution service. It used simple random scanning to propagate.

Slammer's payload was a mere *384 bytes* in length – far smaller than the 4 kb payload of Code Red. Also, *UDP*, being a connectionless protocol, there is no overhead of connection establishment. By contrast, each thread in Code Red spent much time in waiting as it participated in TCP's three-way handshake protocol for connection establishment. Thus, Slammer's propagation speed was not limited by network latency but rather by the maximum link bandwidth. The number of *machines infected* by Slammer *doubled every 8.5 sec* compared to a doubling once in 37 min in the case of Code Red.

Slammer was not designed to erase files, install backdoors, etc. Nevertheless, because of the considerable traffic it generated (55 million scans per second after only 3 min), it severely disrupted Internet traffic. Among the many casualties were several switches and routers that crashed. In hindsight, Slammer was probably a proof-of-concept worm alerting us to the destructive potential of fast-spreading worms. It spread around the world in just 10 min before most humans could respond. During those first 10 min, it is estimated to have infected most of the 75,000 or so vulnerable servers. One can only imagine its impact if, in addition, its payload were actually malicious!

https://hemanthrajhemu.github.io

### 19.3.2 Worm Propagation Models

One of the most alarming aspects of some of the worms developed since 2001 is the speed at which they infect their victims. Modelling worm propagation is important for several reasons – it helps us obtain insights into the factors that govern its speed. It also helps in studying the efficacy of different schemes designed to retard the spread of a worm.

#### *The Simple Epidemic Model*

The Simple Epidemic Model used to study the spread of infectious diseases among humans is an appropriate starting point. The model assumes that there are only two types of entities in the population. Either an individual is *susceptible* or he is *infected*. An infected individual can infect a susceptible person. Once infected, a person remains infected and does not recover.

Let $N$ be the size of the total population. Let $I(t)$ be the number of infected individuals at time $t$. The number of susceptibles at time $t$ is then $N - I(t)$. $\beta$ is the initial infection rate, i.e., each infected person attempts to pass on the infection to $\beta$ susceptibles in 1 time unit. The following differential equation captures the number of infected persons at time $t$.

$$dI = \beta I(t)\left(1 - \frac{I(t)}{N}\right)dt \tag{19.1}$$

or

$$\beta\, dt = \left(\frac{dI(t)}{I(t)\left(1 - \frac{I(t)}{N}\right)}\right) \tag{19.2}$$

In an infinite population, each infected person infects $\beta\, dt$ susceptibles in time interval $dt$. However, in a finite population of size $N$, the probability that the target of an infective is already infected is $\frac{I(t)}{N}$. Such targets do not add to the population of newly infected. The factor $\left(1 - \frac{I(t)}{N}\right)$ in the above equations ensures that only previously uninfected entities are added to the count of the freshly infected in time interval $dt$.

Integrating both sides of Eq. (19.2) yields

$$I(t) = \frac{I_0 N}{I_0 + (N - I_0)e^{-\beta t}} \tag{19.3}$$

A number of organizations, research labs, etc. independently monitor incoming packets into their networks (typically Class B or Class C). They reported their observations covering the entire 24-hour period on July 19th, 2001, when Code Red-I (version 2 with the randomly seeded random number generator) was unleashed. Of interest are the *worm scan traffic* (on port 80) and also the number of unique IP addresses from where this traffic emanated.

A crucial observation is that the Simple Epidemic Model is fairly accurate in determining the number of infected machines in the first 15 hours following the outbreak of Code Red-I (Fig. 19.2). Thereafter, the observed data and the model diverge. There are several explanations for this. First, some administrators applied *patches* so that their systems, whether infected or susceptible, ceased

https://hemanthrajhemu.github.io

to be either after being patched. Also, the large amount of scan traffic caused congestion on the Internet causing a reduction in infection rate.

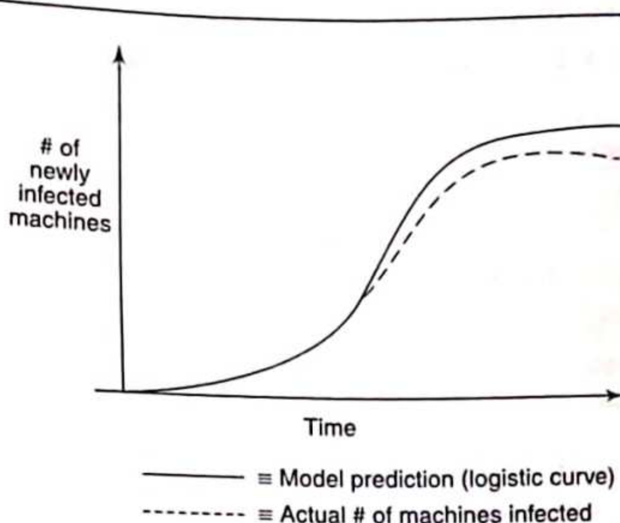



**Figure 19.2** *Number of machines infected by Code Red versus time*

### *Kermack-McKendrick Model*

The Kermack–McKendrick (K–M) model more accurately models the spread of human infectious disease by considering three (instead of two) categories of people:

- those who are susceptible (state $S$)
- those who are infectious (state $I$) and
- those who are neither, i.e. individuals who are cured or those who have succumbed to the disease (terminal state $\mathcal{T}$)

Initially, all individuals in the population are susceptible. It is possible to go from state $S$ to $I$ but not vice versa. An infectious person may or may not be cured. If cured, however, he is never again vulnerable to the disease (see Fig. 19.3). The transition from states $I$ to $\mathcal{T}$ corresponds to an infected machine being patched. Also, such a machine is never again vulnerable to a Code Red infection.

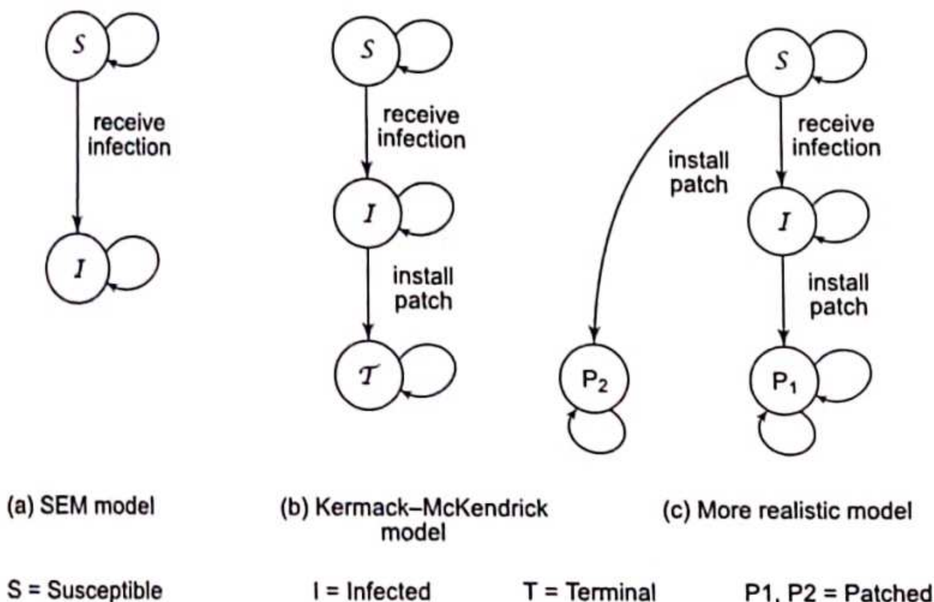



**Figure 19.3** *State machine representation of vulnerable machines during a worm epidemic*

As before, let $I(t)$ be the number of infectious machines at time $t$, let $N$ be the total number of machines, and let $\beta$ be the infection rate. Let $S(t)$ be the number of susceptibles. So, the number of machines in the terminal state is $N - S(t) - I(t)$. The K–M set of equations is

$$\frac{dS}{dt} = -\beta I(t)\left(\frac{S(t)}{N}\right) \tag{19.4}$$

https://hemanthrajhemu.github.io

and

$$\frac{d(N - S(t) - I(t))}{dt} = \gamma I(t) \quad (19.5)$$

Equation 19.4 describes the rate at which the susceptibles decrease (due to the transition to the infectious state). Equation (19.5) captures the rate at which machines in the terminal state increase. Note that the machines in the terminal state are all those that are neither susceptible nor infectious. Analogous to $\beta$, $\gamma$ is the rate at which the infectious machines transit to the terminal state.

While the K–M model better explains the spread of Code Red compared to the Simple Epidemic Model, it still falls short. Its implicit assumptions are at variance with the spread of Internet scanning worms. In particular:

- Machines that are *susceptible but not infectious may also be patched*. Thus, there ought to be a transition from state S to state $\mathcal{T}$. This is not factored into the model.
- The *infection rate, $\beta$, is network-dependent*. As the worm continues to spread, it will consume network resources such as bandwidth leading to traffic congestion. This in turn will slow down the infection rate. However, both the models considered here assume $\beta$ to be constant.
- The K–M model assumes that the rate of transitioning from the infectious to the terminal state is a constant, $\gamma$. During the early stages of the worm epidemic, not much is known about it. So few machines will be patched. However, once the epidemic sets in and there is more public awareness, more administrators will apply patches. This rate will decrease after most worm-aware administrators have applied their patches. Thus, the rate at which machines are patched, and hence $\gamma$, *is far from constant*.

A state diagram of a model that factors the patching of susceptible machines is shown in Fig. 19.3(c).

## 19.4 TOPOLOGICAL WORMS

Topological worms are so called because the machines vulnerable to such a worm can be represented as a *graph* with the nodes representing the vulnerable machines. An edge between Machine A and Machine B exists if A knows/stores the address of B and is capable of *directly* infecting B by sending it a malicious payload.

Topological worms have *focused targets*. Their immediate targets are their neighbours who, in turn, spread the infection to their neighbours and so on. Thus, their rate of spreading is potentially faster than Internet scanning worms, which typically scan the IP address space randomly incurring many futile probe attempts.

In this section, we study two types of topological worms – e-mail worms and P2P worms.

### 19.4.1 E-mail Worms

E-mail worms are among the most common types of malware that have received a lot of attention. As its name suggests, the e-mail worm propagates through infected e-mail. The victim receives e-mail that appears to have come from a trusted or familiar source. On other occasions, the e-mail recipient is lured by an attachment with a catchy caption. In either case, the worm is *activated* when the user *clicks on the attachment*.

A classic example of an e-mail worm is the *"I love you"* worm unleashed in 2000. It appears in the victim's inbox sporting the catchy title – "I love you." The victim sees an innocent text file

https://hemanthrajhemu.github.io

attached to the e-mail. In reality, the file named loveletter.text.vbs contains *Visual Basic script*. By clicking on this attachment, the embedded VB script executes, sending a copy of itself to every person in the victim's contact list. Many e-mail worms exploit the fact that documents created by certain word processors embed *software macros* in them. The macros execute when the document is opened. For example, *Melissa* was a macro worm (or "macro virus") that propagated by sending copies of itself to the first 50 persons in the victim's address book.

A careful study of malware reveals that the space of exploitable vulnerabilities is immense. Who would have thought that a worm could be designed to exploit a vulnerability in the *HTML rendering engine* of a very popular web browser? Yet, that is part of the story of *Nimda* whose claim to fame was that it was one of the earliest worms with *multiple vectors of propagation*. E-mails can carry different MIME types such as image/jpeg, audio/mpeg, or text/html. MIME is short for multipurpose internet mail extension. An attachment is opened by a program appropriate to its MIME type. A flaw in the rendering program was exploited by an attacker who created an executable attachment of that MIME type in the case of Nimda.

One of the best-known e-mail worms of more recent vintage is *Sobig*, which was let loose in 2003. It spread by communicating malicious e-mail or copying itself to an open network share. There were several versions of SoBig. One version could update itself by *downloading code from certain websites*. The URLs of these sites were contained in a file that itself was downloadable from geocities.com (this site allows users to host their own free webpages besides providing tools in support of building dynamic webpages). Some of the malicious code received installed a keystroke logger and stole passwords from its victims.

### 19.4.2 P2P Worms

A P2P network is a *massively distributed* system of computers where each peer or node plays the *role of both client and server*. They are used principally for *sharing files*, which may contain songs, images, videos, etc. Each peer maintains within itself a shared folder of files that it is willing to share with others. Users do not download files from a central server but from their peers located across the globe. They are immensely popular as evidenced by the fact that a very large proportion of Internet traffic is comprised of P2P packets. Examples of P2P systems include *Gnutella* and *KaZaA*.

P2P networks are scalable and resilient. But some of its attractive characteristics may also be serious vulnerabilities. To see how P2P worms spread, it is important to understand how a P2P network operates. Most P2P networks use an *overlay network*, which is a logical network of peers. Two peers are said to be neighbours at any given point of time if there is an active TCP connection between them. For example, a node in the Gnutella network has about 10 neighbours on the average.

A peer, A, that wishes to obtain a file, say *abc*, creates a "Query Request" message for *abc* which it sends to all its neighbours. Each neighbour that does not have the file, in turn, queries its neighbours and so on. In Fig. 19.4, links labelled Q1 are the first set of links over which the query from A propagates. Links labelled Q2 are the next set of links, etc. Many peers are thus hard at work independently trying to locate a peer who has *abc* and is willing to share it. If a peer has *abc*, it returns a "Query Hit" message. This message traces the path of the "Query Request" message in reverse.

The requesting node may receive multiple positive responses. For example, Node A (the requester in Fig. 19.4) receives two positive responses from nodes B and C. It chooses one of them (say node B) and directly contacts it. Node A sends its IP address to node B with a request that *abc* be downloaded to that IP address. The file is then downloaded using FTP or HTTP.

https://hemanthrajhemu.github.io

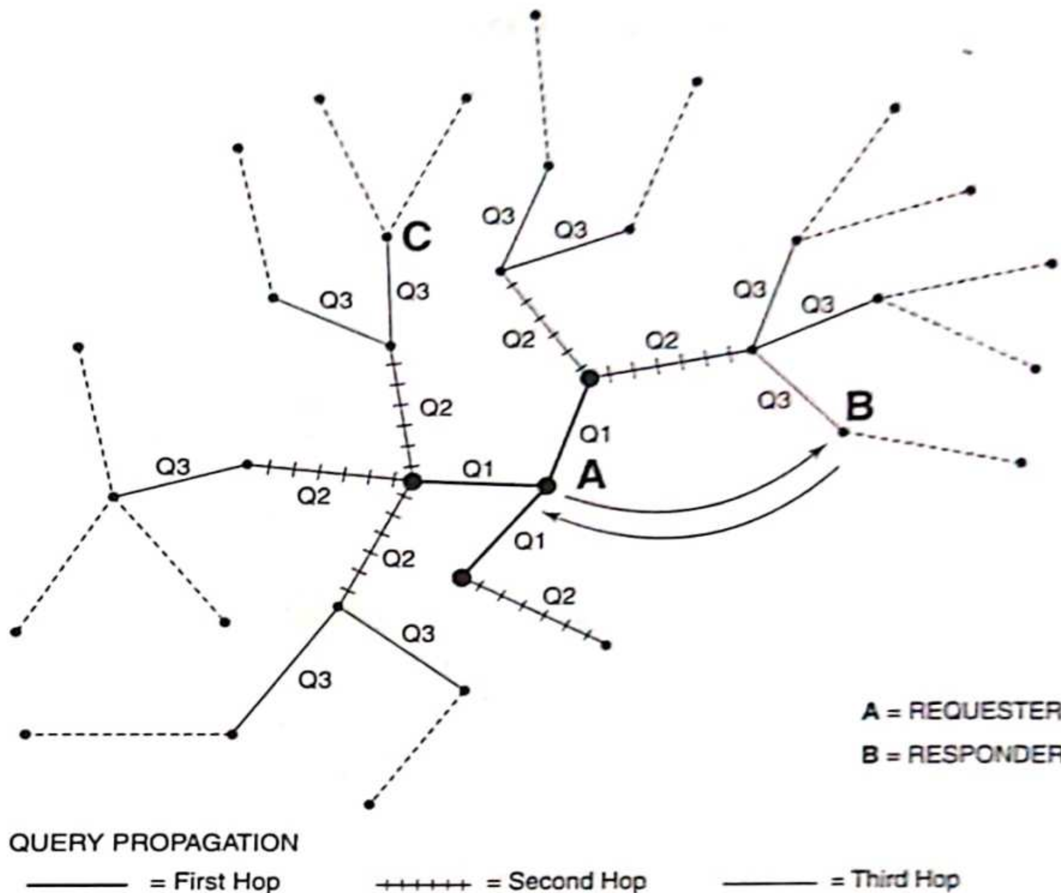



**Figure 19.4** *Querying peers for a file in a P2P network*

Here are potential ways in which P2P worms may spread:

- One of the simplest is for a malicious peer to respond positively to *any* query. If the requester then chooses to download the file from the malicious peer, the latter sends it an infected file whose name is changed to match that of the requested file. The infected file contains a worm which passes on its infection to the requester. Once infected, the requester mimics the behaviour of the malicious peer thus helping to propagate the infection. Alternatively, various "popular" files stored in the shared folder of a peer may be infected. When any of them is downloaded, the infection spreads to the shared folder of the requesting peer.
- Peers in a given P2P network run the same P2P protocol. There are usually few different implementations of this protocol leading to little software diversity. An exploitable buffer overflow vulnerability in one popular implementation is a familiar starting point. This, coupled with the fact that a peer maintains a list of neighbours, implies that a worm has ready targets and does not need to perform random scanning as in the case of Internet scanning worms.

The first type of worm is said to be *passive* since it propagates only when requested to download a file. The second type of worm is *active* since it propagates on its own without receiving requests from its peers. One aspect of P2P worms is that they may result in no apparent traffic anomaly, so an intrusion detection system monitoring network traffic is unlikely to raise an alert.

https://hemanthrajhemu.github.io

## 19.5 WEB WORMS AND CASE STUDY

Web worms differ from malware such as the Internet scanning worms in several ways. Many web worms are executed in browsers which run on *diverse hardware/OS platforms*. Web worms are written in a *high-level language* making it easy to perform complex operations but difficult to execute low-level operations. On the other hand, many other worms are written in assembly language.

One type of web worm is the *XSS worm* – so called since it exploits cross-site scripting vulnerabilities in web servers (see Section 18.4 in Chapter 18 for a discussion of cross-site scripting vulnerabilities). The first step in creating an XSS worm is to inject attack code into a vulnerable web server. When a user accesses the infected website through his/her browser, the malicious code (usually Javascript) is downloaded on to the browser. As in any XSS vulnerability, malicious code executes on the browser. Given that a key function of a worm is to *propagate*, the challenging question then is "How does an XSS worm propagate?" A partial answer to this question may be found through our next case study of the Samy worm.

### XSS Worm Case Study

The XSS worm, *Samy* was unleashed in October 2005. Authored by Samy Kamkar, it infected the social networking site, *Myspace*. Social networking sites typically allow users to create and edit their *profiles* (Fig. 19.5), which are stored on the site and are accessible to other members of the social networking group. A user profile may contain information about him including his hobbies, photographs, etc. A user profile also contains a list of the user's friends with hyperlinks to their profiles.

Samy added a bunch of carefully crafted *Javascript* to his profile. When a visitor to Samy's website, say $V_1$, downloaded Samy's profile on to his browser, the Javascript in Samy's profile executed. This caused Samy to be added as a friend in $V_1$'s profile and also to include the message "*but most of all, Samy is my hero.*"

Within 20 hours of the first visit to Samy's profile, Samy had been added as a friend to more than a *million* user profiles. This rate of spread was even faster than that of Code Red. How did the worm spread and why did it spread so fast?

Any MySpace member can update his profile after logging in. After $V_1$ logged in and viewed Samy's profile, the malicious Javascript embedded in it began to execute. The Javascript uploaded itself on to $V_1$'s profile on the MySpace server, thus infecting it. This is done by an HTTP Post-request sent from the browser to the server. However, that would cause the screen to freeze between sending the request and receiving the HTTP response from the server.

→ *To ensure that the viewer had a normal screen experience, Samy's Javascript created an XMLHttpRequest object which was used to send the malicious Javascript to the Myspace server.*

Unlike the regular HTTP request, the message from an XMLHttpRequest object is *asynchronous* and runs in the background. Consequently, it was not noticed by the user – in this case, $V_1$. The XMLHttpRequest object sent a Post message to update $V_1$'s profile with the malicious Javascript. How does the server know where the XMLHttpRequest has come from?

Because HTTP is a stateless protocol, a *session ID* is created by the server upon user log-in. It is included in all subsequent messages exchanged between client and server during that session. In particular, the session ID is included in the HTML content of each webpage sent by the server to the client. The malicious Javascript in Samy's profile contained code that retrieved the session ID and included it in the XMLHttpRequest.

https://hemanthrajhemu.github.io

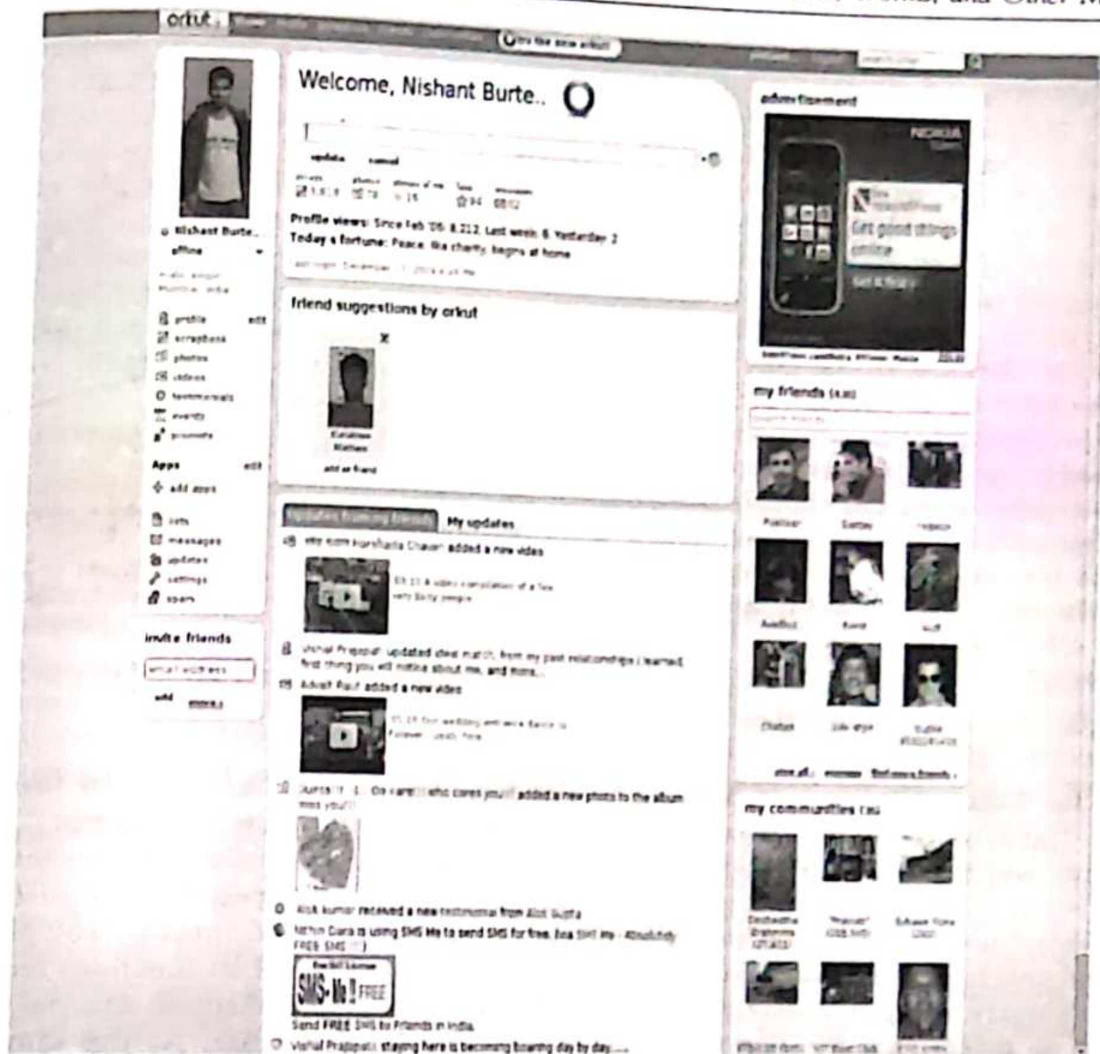



**Figure 19.5** *A social networking site*

From the sessionID in the XMLHttpRequest, the server was able to deduce that it was $V_1$'s profile that needed to be updated. The malicious Javascript that was included in the XMLHttpRequest was then added to $V_1$'s profile and stored on the server. When one or more friends of $V_1$ (say $V_3$) downloaded the profile of $V_1$ on to his browser, his profile was similarly infected by the malicious Javascript. The infection thus propagated as shown in Fig. 19.6.

It is pertinent to ask whether the Myspace server and the browsers that executed the malicious code could have prevented the spread of the Samy worm. For example, why did the Myspace server permit a user to add Javascript to his profile? The fact of the matter is that Myspace was very cautious in *filtering* many suspicious tags. It did filter out tags such as <script> and <body>.

→ *However, Samy included Javascript within the <div> tag* as shown below.

```
<div expr = " * Javascript here * " style="background:url('java
script:eval(document.all.mycode.expr)')">
```

https://hemanthrajhemu.github.io

The <div> tag is used to define separate divisions or segments within a document. (It is often used in the context of *Cascading Style Sheets* (CSS) – a mechanism that allows the content of a webpage to be separated out from aspects of its presentation. CSS allows each segment to be styled differently using style information stored separately in a .css file.) Samy created a Javascript expression as part of the expr attribute of the <div> tag. It instructed the browser to execute the expression using eval(document.all.mycode.expr). Not all browsers execute Javascript contained in the <div> tag but some did cooperate with Samy and thus aided the spread of the worm.

Note that the keyword "javascript" was filtered by Myspace. So how did it manage to persist in infected user profiles on the Myspace server?

→ *Samy split "javascript" on two lines as shown in the <div> tag above.*

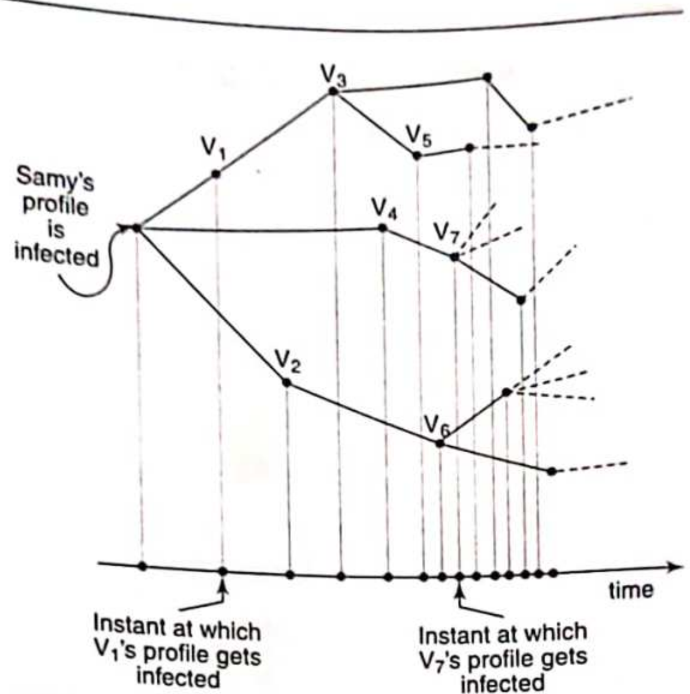



**Figure 19.6** *Illustrating spread of the Samy worm*

Fortunately for Samy, this was not detected by the Myspace server. Second, most browsers were able to fuse "java" and "script" from the two successive lines to create "javascript."

While Samy was written in Javascript and executed on the client, another web worm, *Santy* was written in *Perl* and executed on the server. It targeted websites developed using *phpBB* – an open source software package used to create bulletin boards, newsgroups and forums. The phpBB application did not carefully check for inputs received from its clients. One of its functions received input from a URL's query string. Cleverly disguised worm code was passed through this parameter. The server failed to detect that the input parameter was actually Perl code. At the same time, the function had the capability to treat the input parameter as code and execute it. Thus the malicious code was executed. Santy was novel in another respect. It attempted to identify other websites running the phpBB application by contacting *web search engines* such as Google to locate its targets.

## 19.6 MOBILE MALWARE

### 19.6.1 Introduction

New-generation smartphones combine the functionality of a cellphone and a low-end PC. They may be used for storing confidential documents, communicating via e-mail/SMS/MMS, and taking photographs. They support feature-rich applications that run on top of a complete OS. The most common OS on smartphones is the Symbian followed by Windows Mobile, Linux, and recently the Mac OS X (on the iPhone). They provide a rich set of APIs to access the phone book and other files, send SMS/MMS messages, etc. Unfortunately, these very APIs can also be used by malware to, for example, read a confidential document on the smartphone and ship it to the attacker as an MMS attachment.

https://hemanthrajhemu.github.io

In the case of the Symbian operating system, users can download new applications onto their smartphones. New applications as well as updates of existing applications are packaged in Symbian Installation Source (SIS) files. Some of the earliest mobile malware was packaged in "well-formed" SIS files. The installer was tricked into believing that this was an update of an existing application. The installer replaced the existing application with its new "version" containing the malicious code. So, whenever the application was invoked, it was the malware also that ran.

Another feature that many smartphones (and indeed laptops) have these days is *Bluetooth* connectivity. Bluetooth supports a host of tasks such as the exchange of e-business cards between two smartphone users, communication with point-of sale terminals, headset to smartphone communication and "sync"ing between a smartphone and a PC. But, as we will soon see, Bluetooth has turned out to be an important vector of mobile worm propagation.

## 19.6.2 Bluetooth

Bluetooth is both a communication technology and a protocol stack. As a communication technology, it supports short-range wireless communication – a maximum of between *10 and 100 meters* between devices. Like Wi-Fi, Bluetooth uses 2.4 GHz shortwave radio technology. However, its *power* requirements are considerably *lower* than Wi-Fi. Correspondingly, its transmission speed is limited to *5 Mbps*.

Bluetooth is a complex, multi-layered protocol. The implementation of Bluetooth on Linux systems exceeds 25,000 lines of kernel code. Not surprisingly, there are many examples of buffer overflow, integer underflow, and other *vulnerabilities* in its *implementation* that have been exploited. In addition, there are a number of subtle vulnerabilities in the Bluetooth *protocol* itself.

### *Discovery and User Authorization*

Besides the vulnerabilities mentioned above, *social engineering* is a key factor in the spread of mobile malware. Many PC users do not hesitate to click hot links in their e-mail or open attachments even when the sender of the e-mail is unknown. This behaviour carries over to the mobile world where many users have no hesitation in accepting a file from an unknown source. The combination of Bluetooth implementation vulnerabilities, rich feature set of the smartphone, and unthinking user behaviour has exposed the smartphone to various strains of malware.

To investigate the spread of malware in smartphones, it is necessary to understand the basics of how smartphones exchange files using Bluetooth. To *discover* other Bluetooth-enabled devices in its neighbourhood, a device initiates an *inquiry* procedure, which includes broadcasting an inquiry request. All devices in the range of the initiator that are in *discoverable mode* respond sending their bluetooth device address (BD_ADDR). This is a 48-bit MAC address – the first 24 bits identify the device manufacturer/model and the last 24 bits specify a particular instance of that model.

Bluetooth is a connection-oriented protocol. A device, A, can set up a connection to any other device, B. But to set up such a connection, A should know the BD_ADDR of B. One way of obtaining B's BD_ADDR is through the discovery procedure. A large percentage of users keep their phones in discoverable mode, so this is an attractive way of harvesting device addresses especially in crowded areas such as railway stations or malls. However, it should be noted that even with the phone in non-discoverable mode, there are a number of brute force techniques to extract its BD_ADDR.

Knowing B's BD_ADDR, A could attempt to exchange files with it using the OBEX (object exchange) protocol. A session protocol that resembles HTTP, OBEX is used to transfer images, business cards, and other files between Bluetooth devices. An attacker, A, could use OBEX Push to

https://hemanthrajhemu.github.io

transfer a file containing malicious code to user, B. *User authorization* is usually required before a file can be accepted by his smartphone. Each user selects a *PIN* which varies between 4 and 16 characters long but 4 characters are typically used. The smartphone usually prompts a user to enter his/her PIN as a way to confirm whether an external file, for example, should be accepted.

Some OS versions accept file transfers without user authorization. And some smartphones allow users to disable the "Authorization Required" option for file transfers. Unfortunately, that is not the end of the story – it has been estimated that between 7% and 25% of users indiscriminately accept files or MMS attachments. It has been found that this percentage increases sharply if the file or attachment title has connotations of sex, for example. With the growth in smartphone usage to less computer savvy segments of the population, these percentages are likely to be higher in the coming years.

In summary, by keeping his smartphone in discoverable mode, a user risks revealing his smartphone's BD_ADDR to an attacker. Further, by disabling authorization or by careless acceptance of external files/attachments, a user keeps his smartphone open to trojans and other malware.

### *Link Level Security*

The level of security provided by user authorization alone is generally inadequate. Another level of security provided by Bluetooth is *link-level authentication* and *encryption*. For this purpose, both sides compute a common secret called the *link key*. Bluetooth uses a procedure called *pairing* wherein this key is computed by two participating devices. Pairing is preceded by discovery/inquiry and paging. The latter is a procedure whereby the discovering device, A, establishes a connection with the discovered device, B.

*Computing the Link Key*

- The first step in deriving the common link key between A and B is to compute an *initialization key*, $K_{init}$. This is a function of the *BD_ADDR* of B and a nonce, *IN_RAND*, generated by A as shown in Fig. 19.7(a). Before the pairing procedure, the owners of A and B must agree in an off-line manner on a temporary *PIN* to be used specifically as part of the pairing procedure. Both users then type in the temporary *PIN*. $K_{init}$ is also a function of this temporary *PIN* agreed to by both parties. $K_{init}$ is computed from *IN_RAND*, $BD_ADDR_B$, and the *PIN* using an algorithm, $E_{22}$, based on a block cipher called SAFER+.
- To compute the *link key*, A and B each generate a random number ($LK_RAND_A$ and $LK_RAND_B$, respectively). Each party then performs an XOR of its random number with $K_{init}$ and they each transmit this across. Each side recovers the other party's random number by performing an XOR of the received value with $K_{init}$ (see Fig. 19.7(b)). Now, each side has $LK_RAND_A$, $LK_RAND_B$, and the two device addresses, $BD_ADDR_A$ and $BD_ADDR_B$. They then perform identical operations on these to obtain the link key, $K_{AB}$ as shown in Fig. 19.7(b). The operations involve the use of an algorithm, $E_{21}$, which, like $E_{22}$, is based on the cipher, SAFER+ .
- Thereafter, each device stores the pair, *BD_ADDR*, of the other device and the newly computed link key in a database. Each device maintains such a database of *BD_ADDR*, link key pairs, one pair per device it is paired up with.

*Using the Link Key*

The link key is used for both authentication and encryption. We discuss authentication next. Suppose A and B were already paired. Let $K_{AB}$ be their common link key. Now suppose A wishes to authenticate B. For this purpose, it generates a random number, $RAND_A$ (a challenge) and sends

https://hemanthrajhemu.github.io

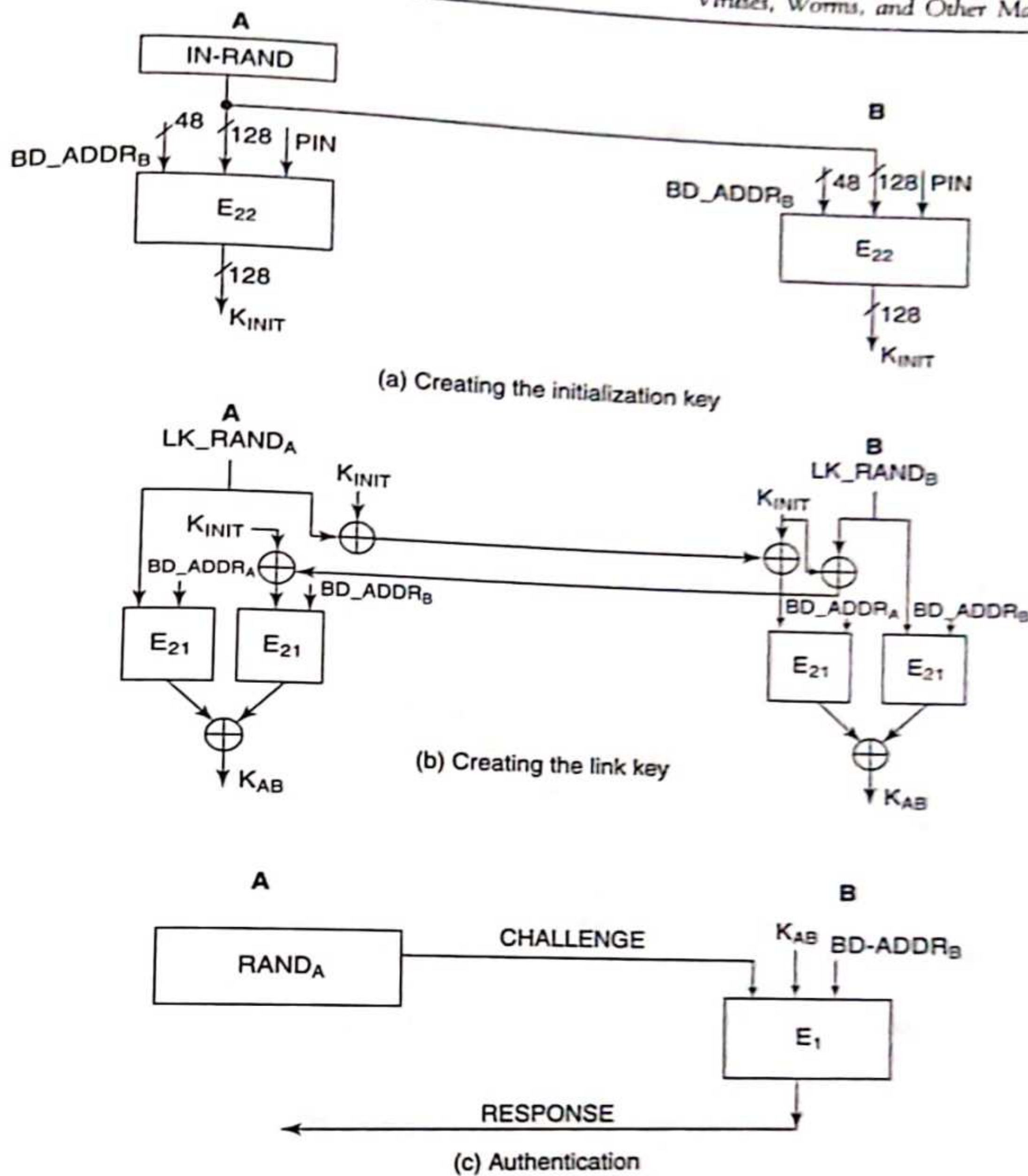



**Figure 19.7** *Generation and use of the Link key*

it to B. The response computed by B is $E_1(K_{AB}, RAND_A, BD_ADDR_B)$. Here again, $E_1$ is a function that is based on the block cipher, SAFER+. A also computes $E_1(K_{AB}, RAND_A, BD_ADDR_B)$ since it has $K_{AB}$. It can then verify whether the response from B tallies with what it computed.

*Hacking the Link Key*

It is possible to launch a *dictionary attack* by sniffing each message involved in pairing and authentication (Exercise 19.10). These attacks enable an eavesdropper to obtain the link key, $K_{AB}$ making it possible for the attacker to impersonate B to A. The latest version of Bluetooth – version 2.1 – gets rid of this problem by using Elliptic Curve Diffie–Hellman (ECDH) key exchange. The

https://hemanthrajhemu.github.io

idea is similar to that in the EKE protocol (Chapter 12). Recall that the latter was designed to thwart off-line dictionary attacks on weak passwords. In the case of smartphones, the PIN, which could be just four digits long, is analogous to the weak password. Once the PIN is guessed, the link key can easily be obtained.

There are social engineering attacks that enable an attacker to pair his device with that of a careless user. Imagine an attacker who has obtained the *BD_ADDR* of a user during the discovery procedure. The attacker's device then makes a pairing request to the victim. Pairing requests contain the *common name* of requester such as Bobby. Note that one can change one's common name but not the BD_ADDR of one's smartphone. In this case, the attacker sports a fancy name such as *Your_Sexy_Admirer*. As in phishing attacks, the victim is lured to respond to the pairing request. But there is one caveat. Don't the attacker and victim have to agree on a Pairing PIN? The answer is *Yes*. However, many people pay scant attention to the choice of a PIN and frequently use simple PINs such as 1234. Thus, with a combination of luck and some planning, it is not difficult for an attacker to pair his device with the device of a careless user.

*Paired devices* are sometimes given the exalted status of *trusted devices*. This means that in some implementations, two paired devices can exchange files without the need for user authentication. This again opens the door to the receiving of mobile malware on to the cellphone of unsuspecting users.

### 19.6.3 Examples

*Cabir* was one of the earliest proof-of concept worms that targeted the *Symbian Series 60 OS*. Unleashed in June 2004, it was authored by the International Virus writing group 29A. The worm attempts to discover other *Bluetooth-enabled phones* set in discoverable mode. When it finds such a phone, it sends the worm payload in a SIS file. The receiver needs to accept and install the file. Its payload was mostly benign typically displaying "Caribe" on the screen. However, the continuous scanning for new victims by an infected phone depletes battery power.

*Commwarrior*, which appeared in March 2005, was the first worm to spread through, *both Bluetooth and MMS*. Like Cabir, it targeted *Symbian* smartphones. It used MMS to spread to different contacts in the smartphone's address book. It requires user interaction to be installed. It entices the user with catchy subject headers such as "Happy Birthday" or "Fee SEX! . . .." Once it infects a smartphone, it attempts to discover Bluetooth-enabled smartphones and pass on the infection as a SIS file to them.

While Symbian phones have been the most widely targeted, Windows and J2ME (Java) phones have also been the target of attacks. *WinCE.Duts* was one of the first worm to target the *Windows CE* operating system while the *Redbrowser* trojan was the first to target *J2ME* phones. Finally, proof-of-concept "crossover" worms have also been observed. These are worms that spread from a PC to a smartphone and vice versa when the two are being "synced."

## 19.7 BOTNETS

### 19.7.1 Basics

In the last few years, there has been a trend away from the headline-grabbing, fast-spreading Internet worms studied in Section 19.3 to botnets. A botnet is an army of compromised computers or *bots* connected to the Internet and *remotely controlled* by a "botmaster." The earliest botnets were a collection of zombies that participated in DDoS attacks as explained in Section 17.1 of Chapter 17. Today's botnets may comprise tens of thousands or even millions of bots.

https://hemanthrajhemu.github.io

The emergence of botnets is closely linked to the motive of financial gain that is behind many recent cyber attacks. They are often used to send *spam mail* on behalf of third parties, for example. Bot programs may contain *keyloggers* and other forms of spyware that capture sensitive personal information such as passwords and credit card numbers and send these to the botmaster. Botnets have also been used as an extortion tool – "Pay up or your website will be bombarded by a *DDoS* attack".

How does a computer become a bot? Bots are created in ways similar to many of the traditional trojan/worm/virus infections. A common *vector of propagation* is e-mail that contains an infected attachment. Another is through downloading a malicious webpage containing scripts that exploit vulnerabilities in certain browsers or application software. A bot infection may also be propagated by bots themselves by scanning the Internet for vulnerable machines. Finally, open file shares and IRC (Internet Relay Chat) multicast messages have also been widely used to spread infections.

One important difference between a bot and a computer infected by a traditional worm/virus/Trojan is that a bot needs to communicate with specific nodes in the botnet to receive fresh commands. A bot may be ordered to send spam or to "Launch a DDoS attack on site abc.com beginning 14:00 hours on 01-12-10." Some of the nodes in the botnet play the role of *Command and Control (C&C) servers*. They receive commands from the botmaster and disseminate these to the rest of the bots.

In addition, some botnets need to inject their bots with *new or updated executables*. Often, the new executables must be downloaded by bots from a site on the Internet. The URL of this site needs to be disseminated to the faithful. Thus, one of the key design issues in a botnet is *communication and control*. The two principal botnet architectures are *centralized and distributed*. These are described next.

Early botnets used an *IRC server* as a C&C server. A channel on such a server was used to convey command information. However, once the C&C server was detected, it was easy for IRC server operators to bring down the botnet channel and thus disrupt all or part of the botnet. Examples of IRC-based botnets are *Agobot* and *SDBot*, which were first seen in 2002.

A more recent trend has been distributed and decentralized botnet architectures, which leverage existing *P2P networks*. P2P networks are highly *scalable* and *robust*. The connectivity of P2P networks ensures that even if a large number of bots are disabled, the rest of the bots continue to stay connected. Moreover, there are no fixed C&C servers making it hard to detect and incapacitate a P2P-based botnet (see Fig. 19.8).

### 19.7.2 Case Study: The Storm Botnet

The Storm botnet was first detected in *January 2007*. Its other names are Peacomm, Nuwar, and Zhelatin.

Bots in the Storm botnet are infected in stages. The most common vectors for propagating the primary infection appear to be *e-mail* or *infected websites*. E-mail was sent with sensational subject lines like "230 die as Storm batters Europe." Likewise, users were lured into downloading free but infected files from websites containing music of various pop artists.

The primary infection instructed the victim to join the Storm botnet embedded in the *Overnet P2P network*. Once part of the botnet, the bot was programmed to receive the second and subsequent injections of malicious code. One of the injections instructed the bot to propagate e-mail viruses. Another injection received some days later instructed the bot to launch a DDoS attacks on a target specified by the botmaster.

https://hemanthrajhemu.github.io

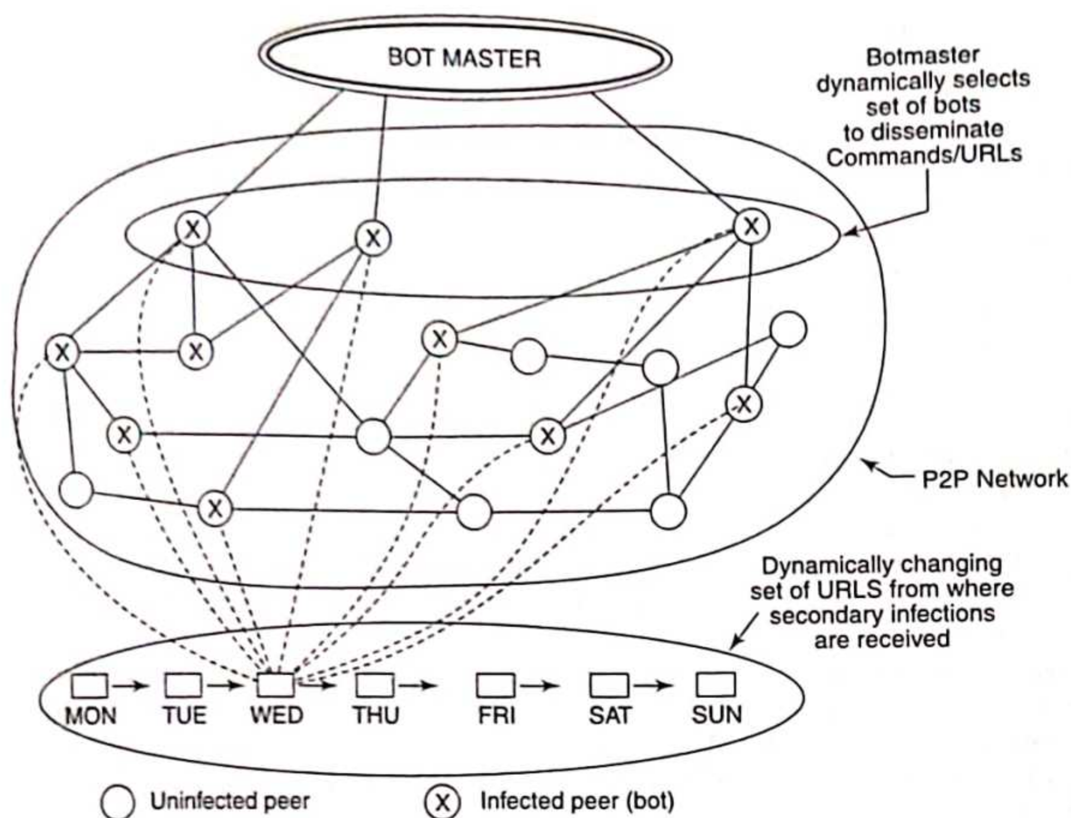



**Figure 19.8** *Second-generation botnets using P2P networks*

The Overnet P2P network is based on the *Kademlia* protocol. The latter employs a distributed *hash table-based routing protocol*, which efficiently locates a value corresponding to a given *search key*. Suppose a peer, X, needs to access a file. It would perform a search based on a search key, which could be the hash of a file (or the hash of the file name). The result of the search is the IP address and port number of the peer hosting that file, say Y. Then X would directly contact Y to obtain a copy of the file. Both node IDs and search key are drawn from the same space of 128-bit numbers.

The initial infection had a *hard-coded list of 146 peers* used for *bootstrapping* (this is a procedure that makes a newly infected machine part of the Storm botnet). Each entry in the peer list is made up of a 128-bit MD-4 peer hash followed by the IP address and port number of the peer. When searching for a key, a bot first consults these peers. But what does a bot search for and with what search key?

The bot is programmed to fetch updated code for its subsequent infections. To confuse security analysts, the designers of the Storm botnet not only updated the malicious code but also changed the URLs from which the code was to be downloaded. The P2P network is not used to communicate this code but rather encrypted URLs from which this code may be downloaded. Thus the *search value was the encrypted URL*, while the *search key* was computed by each bot as a *function of the date and a random integer*,

$$f(date, rand)$$

where *rand* is between 0 and 31.

https://hemanthrajhemu.github.io

In response to the search query, a bot receives the encrypted URL and also a *partial decryption key*. This partial key in conjunction with a hard-coded key in the bot is used to decrypt the encrypted URL. The bot then proceeds to fetch the malicious code from that URL. Thus, the URLs from which malicious code are downloaded changes daily (or several times a day). As law enforcement authorities attempt to shut down one such site, another comes up.

In September, 2008, another widely dispersed botnet was discovered based on the *Conficker worm*. Its design was in many ways similar to that of the Storm worm. A bot, in this case, used a *domain generation algorithm* to dynamically generate domain names from where malicious code could be downloaded. In addition to "domain flux," it used another DNS technique called "fast flux" in which a single domain name was mapped to hundreds of IP addresses. Another aspect of Conficker is that it attempted to disable anti-virus and other detection software on its victims.

## SELECTED REFERENCES

[WEAV03] presents a taxonomy of computer worms. [MOOR02] is a case study of the Code Red worm. The classical epidemic model was first applied to Internet scanning worms in [STAN02]. [MOOR03] is a case study of the Slammer worm. A good summary of the vulnerabilities exploited by web worms is contained in [HOLZ06] and a technical analysis of the Samy worm appears in [SAMY(a)]. Mobile worms are introduced in [LEAV05]. Hacking into mobile phones is the subject of [FADI06].

[DAGO07] is a good taxonomy of botnet architectures. [STEW07], [GRIZ07], and [HOLZ08] are useful case studies of the Storm worm. [PORR09] provides valuable insights into the working of the Conficker worm.

## OBJECTIVE-TYPE QUESTIONS

**19.1** Which of the following is/are true of various forms of malware:
- (a) a worm attaches itself to a file or program
- (b) a trojan is a stand-alone program
- (c) a virus does not necessarily replicate
- (d) a worm uses the network to spread

**19.2** Early viruses used the following technique to evade detection
- (a) they were encrypted and decrypted only during execution
- (b) they updated themselves by downloading code from an FTP site
- (c) they were hidden in the payload of TCP packets carrying regular traffic
- (d) they used compression so that the length of the infected and original files matched

**19.3** One of the reasons for Slammer's much faster spread compared to Code Red was
- (a) it employed multi-threading
- (b) it was transported using UDP, not TCP
- (c) it employed multiple vectors of propagation
- (d) it employed hit-list scanning

**19.4** During the initial stages of an attack by a new Internet scanning worm, the number of infected machines increases
- (a) exponentially with time
- (b) logarithmically with time
- (c) polynomially with time
- (d) at a constant rate

https://hemanthrajhemu.github.io

**19.5** Which of the following is/are true of topological worms
(a) they include e-mail and P2P worms
(b) they only infect their immediate neighbours in a given topology (graph)
(c) they usually have at least two vectors of propagation
(d) they are typically activated by some human action

**19.6** Which of the following is/are true of the SoBig worm?
(a) it had multiple vectors of propagation
(b) it wiped out large sections of its victim's hard disk
(c) it updated itself by downloading code from certain websites
(d) it obtained its targets' addresses using a web search engine

**19.7** Cabir's claim to fame is that
(a) it was one of the first worms to target cellphones
(b) it was the first worm to use web search engines to locate new targets
(c) it was one of the first worms to carry a destructive payload
(d) it was one of the first XSS worms

**19.8** The Storm Bot uses a P2P network to
(a) spread a primary infection
(b) spread a secondary infection
(c) disseminate a URL from which to obtain the next stage of infection
(d) obtain the address of their next target

## EXERCISES

**19.1** Which is the more appropriate term – e-mail worm or e-mail virus?
Explain your answer.

**19.2** An attacker performs OS-fingerprinting to determine what OS and which version is running on a potential target. How is this information obtained and how is it used?

**19.3** State three reasons why Code Red (and especially its later versions) was so successful in spreading.
It has been stated that Code Red caused billions of dollars in damage. What do you think this cost is due to?

**19.4** Are the following variables captured by (i) the Simple Epidemic Model and (ii) the Kermack-McKendrick Model? If so, how?
(a) communication latency
(b) network bandwidth
(c) payload size
(d) number of never-infected machines that were patched
(e) number of infected machines that were subsequently patched

**19.5** Using the Simple Epidemic Model ($dI/dt = \beta I(t)\,(1 - I(t)/N)$), we derived an expression for the total number of machines infected up to time $t$. Derive an expression for the total number of worms created up to time $t$.

**19.6** Some worm/virus detection software identify distinctive patterns of bits in the worm payload (called worm signatures) and use these to detect other instances of the worm. Now consider a worm that has just been unleashed. Such worms that have never been seen before are

https://hemanthrajhemu.github.io

referred to as zero-day worms. Explain how you would design a high-speed detection system for zero-day

— Internet scanning worms

— web worms

— mobile malware

**19.7** What, under each of the following headings, did the Samy worm exploit?

- Vulnerability in server-side software
- Vulnerability in certain browsers
- Certain features of web technologies such as CSS and AJAX

**19.8** A web worm always executes on the user's browser. Yes or no? Explain.

**19.9** A Bluetooth-enabled device set in discoverable mode sends its Bluetooth Device address (MAC address) in response to an Inquiry Request message. What is the danger, if any, in revealing this information?

**19.10** The pairing procedure between two Bluetooth-enabled devices is used to compute the common link key. Explain how this pairing procedure may be attacked to obtain the link key.

**19.11** Design an improved Bluetooth protocol that is not vulnerable to the attack in the previous exercise.

**19.12** State and explain the pros and cons of a distributed botnet architecture vis-à-vis a centralized one.

**19.13** Suggest a way to determine if a particular machine is bot-infested.
Devise a scheme to count the number of bot-infested machines in your organization or campus.

## ANSWERS TO OBJECTIVE-TYPE QUESTIONS

| | | | |
|---|---|---|---|
| **19.1** (b)(d) | **19.2** (d) | **19.3** (b) | **19.4** (a) |
| **19.5** (a)(b) | **19.6** (a)(c) | **19.7** (a) | **19.8** (c) |

https://hemanthrajhemu.github.io

Chapter 21

# Firewalls

A firewall acts as a *security guard* controlling access between an internal, protected network and an external, untrusted network based on a given *security policy*. Besides preventing intruders getting in, a firewall also helps prevent confidential inside data from getting out.

A firewall may be implemented in hardware as a stand-alone "firewall appliance" or in software on a PC. In addition, many routers support basic firewall functionality. A single firewall may be adequate for small businesses and homes. However, in several large enterprises, multiple firewalls are deployed to achieve *defence in depth*.

We first address basic firewall functionality and then the different types of firewalls in Section 21.1. In Section 21.2, we study the security architecture of a mid-size organization with a focus on firewall placement and configuration. Finally, in Section 21.3, we introduce the personal firewall through a case study of Linux IPTables/NetFilter.

## 21.1 BASICS

### 21.1.1 Firewall Functionality

The main functions of a firewall are listed as follows:

*Access Control*. A firewall filters incoming (from the Internet into the organization) as well as outgoing (from within the organization to the outside) packets. A firewall is said to be *configured* with a *ruleset* based on which it decides which packets are to be allowed and which are to be dropped.

*Address/Port Translation*. Network Address Translation (NAT) was introduced in Chapter 2. NAT was initially devised to alleviate the serious shortage of IP addresses by providing a set of private addresses that could be used by system administrators on their internal networks but that are globally invalid (on the Internet). Publicly accessible machines within an organization, such as web servers, may or may not have public Internet addresses. However, in the latter case, it is possible to conceal the addressing schema of these machines from the outside world through the use of NAT. Through NAT, internal machines, though not visible on the Internet, can establish a connection with external machines on the Internet. NATing is often done by firewalls.

*Logging*. A sound security architecture will ensure that each incoming or outgoing packet encounters at least one firewall. The firewall can log all anomalous packets or flows for later study. These logs are very useful for studying attempts at intrusion together with various worm and DDoS attacks.

https://hemanthrajhemu.github.io

*Authentication, Caching, etc.* Some types of firewalls perform authentication of external machines attempting to establish a connection with an internal machine. A special type of firewall called a *web proxy* authenticates internal users attempting to access an external service. Such a firewall is also used to cache frequently requested webpages. This results in decreased response time to the client while saving communication bandwidth.

### 21.1.2 Policies and Access Control Lists

High-level policies for access to various types of services are formulated within an organization or campus. Examples of these include the following:

- All received e-mail should be *filtered for spam* and viruses.
- All *HTTP requests by external clients* for access to authorized pages of the organization's website should be permitted.
- *DNS queries* made *by external clients* should be allowed provided they pertain to addresses of the organization's publicly accessible services such as the web server or the external e-mail server. However, queries related to the IP addresses of internal machines should not be entertained.
- The organization's employees should be allowed to *remotely log into* authorized internal machines. However, all such communication should be authenticated and encrypted.
- Only two types of outgoing traffic are permitted. First, all e-mail from within the organization to the outside world are permitted. Second, requests emanating from within the organization for external webpages are permitted. However, requests for pages from certain "inappropriate" websites should be denied.

High-level policies are translated into a set of rules that comprise an *Access Control List*. A rule specifies the action to be taken as a function of

(i) the packet's *source IP address* and *port number*
(ii) the packet's *destination IP address* and *port number*
(iii) the *transport protocol* in use (TCP or UDP)
(iv) the packet's *direction* – incoming or outgoing

The Access Control List for the high-level policies enunciated earlier appears in Table 21.1.

Policies can, in general, be either permissive or restrictive. A *permissive policy* is defined as follows:

*Permit all packets except those that are explicitly forbidden.*

A *restrictive policy*, on the other hand, is defined as follows:

*Drop all packets except those that are explicitly permitted.*

The ACL in Table 21.1 implements a restrictive policy – the default action is Deny as expressed in rules 5 and 8.

The rules are scanned top to bottom. As soon as a rule is found that matches the packet's attributes (IP addresses, port numbers, etc.), the action in that rule (usually permit or deny) is taken and no further rules are processed for that packet. The scanning order is important. For example, if rules 4 and 5 in Table 21.1 are interchanged, then IPSec traffic will be dropped. Also, from a performance perspective, it makes sense to put the most frequently acted upon rule earlier on. By so doing, we can expedite the decision on what to do with a packet. Finally, it is important to include the *default deny* rule at the end of the rule set – this prevents ambiguity over what action to take for a packet that has not been matched against the attributes in any of the previous rules.

https://hemanthrajhemu.github.io

**Table 21.1** *Example access control list*

| *No.* | *In-bound (I) or out-bound (O)* | *Transport protocol* | *Src. IP addr.* | *Src. port* | *Dest. IP addr.* | *Dest port* | *Action* | *Comment* |
|---|---|---|---|---|---|---|---|---|
| 1 | I | TCP | Any | Any | MS | 25 | Permit | Allow incoming e-mail |
| 2 | I | TCP | Any | Any | WS | 80 | Permit | Allow requests for organization's webpages |
| 3 | I | UDP | Any | Any | NS | 53 | Permit | Allow DNS queries |
| 4 | I | IPSec | Any | Any | * | * | Permit | Allow incoming VPN traffic |
| 5 | I | Any | Any | Any | Any | Any | Deny | Forbid all other incoming traffic |
| 6 | O | TCP | Any | Any | Any | 25 | Permit | Allow outgoing e-mail |
| 7 | O | TCP | Any | Any | * | 80 | Permit | Allow requests for external webpages |
| 8 | O | Any | Any | Any | Any | Any | Deny | Forbid all other outgoing traffic |

*Note:* MS, WS, and NS are the IP addresses of the organization's e-Mail Server, Web Server, and DNS server (Name Server), respectively. * depends on configuration

### 21.1.3 Firewall Types

Firewalls can be classified into the categories described in rest of this section.

#### *Packet Filters and Stateful Inspection*

Processing the ruleset in Table 21.1 involves checking for matches in the IP, TCP, or UDP headers. For example, it may be necessary to check whether a packet carries a certain specific source or destination IP address or port number. The earliest firewall designed to perform this task was referred to as a *packet filtering firewall*. It is often performed by the *border router* or *access router* that connects the organization's network to the Internet. In effect, the border router becomes the first line of defence against malicious incoming packets. We next explain why the packet filtering firewall is inadequate.

Consider an external mail server (IP address = ABC) that wishes to deliver mail to an organization. For this purpose, it should first establish a TCP connection with the organization's mail server, MS. Consider the arrival of a packet with the following attributes:

Source IP address = ABC
Destination IP address = MS
TCP destination port = 25 (SMTP port)
ACK flag set

Such a packet would be part of a normal flow provided a connection between ABC to MS has been established. But suppose such a connection has not yet been established. Should the packet still be allowed in?

The simple packet filter will allow the packet to enter even if no prior connection between ABC and MS was established. It should be noted that such packets are often used to perform *port scans*. Without an existing connection, the MS would send an RST in response to receipt of such a packet. Thus, such packets provide information to the sender regarding which ports are open on various hosts within the organization.

https://hemanthrajhemu.github.io

A simple packet filter merely inspects the headers of an incoming packet in isolation. It does not view a packet as part of a connection or flow. Hence, it will not be able to filter out such packets arriving from ABC. What is needed is a *stateful packet inspection firewall*. Such a firewall uses a packet's TCP flags and sequence/acknowledgement numbers to determine whether it is part of an existing, authorized flow. If it is participating in the establishment of an authorized connection or if it is already part of an existing connection, the packet is permitted, otherwise it is dropped. In the above example of the packet from ABC, the stateful packet inspection firewall will realize that it has not encountered the first two packets in the three-way handshake and will hence drop this packet.

### *Application Level Firewalls*

A packet-filtering firewall, even with the added functionality of stateful packet inspection, is still severely limited. It understands the network and transport layer headers but is indifferent to the application being run. What is needed is a firewall that can examine the *application payload* and scan packets for worms, viruses, spam mail, and inappropriate content. Such a device is called a *deep inspection firewall*.

A special kind of application-level firewall is built using proxy agents. Such a "proxy firewall" acts as an intermediary between the client and server. The client establishes a TCP connection to the proxy and the proxy establishes another TCP connection with the server as shown in Fig. 21.1. To a client, the proxy appears as the server and to the server, the proxy appears as the client. Since there is no direct connection between the client and the server, worms and other malware will not be able to pass between the two, assuming that the proxy can detect and filter out the malware. Hence, the presence of the proxy enhances security.

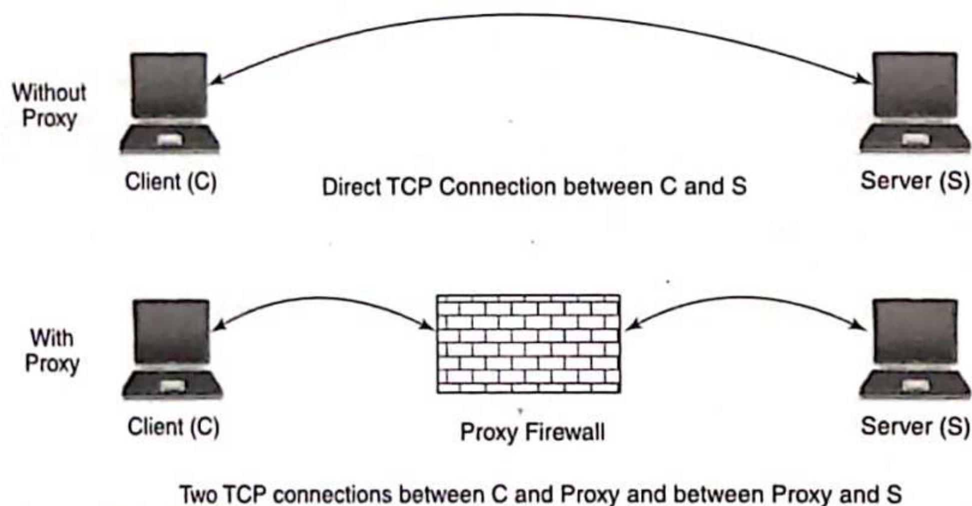



**Figure 21.1** *Proxy firewall*

There are proxy agents for many application layer protocols including HTTP, SMTP, and FTP. In addition to filtering based on application layer data, proxies can perform client authentication and logging. An HTTP proxy can also cache webpages. Caching has a major impact on performance. If the webpage is cached in a web proxy server located in the client's organization, the response time could be greatly reduced compared to that where the page has to be fetched from the external web server. Also, caching reduces the demand on external communication bandwidth while easing the load on the web server.

https://hemanthrajhemu.github.io

Firewalls are a necessary element in the security architecture of an organization that permits access to/from the external world. In the next section, we study firewall deployment.

## 21.2 PRACTICAL ISSUES

The security architecture of a medium size or large organization includes firewalls, proxy servers, VPN terminators, and intrusion detection/prevention (IDS/IPS) devices. In this section, we concentrate on the placement and configuration of firewalls and proxy servers.

### 21.2.1 Placement of Firewalls

We first note that firewalls help segregate or isolate the network into multiple *security zones*. Each firewall in the organization enforces rules that control the transfer of packets between different security zones. At the very least, there are three zones – the Internet, the region containing the publicly accessible servers, and the internal network.

Figure 21.2 depicts a four-zone layout using three firewalls. Of the three firewalls, the first is really a router (the Border Router) with some packet-filtering capability. This is the access router that interfaces with the Internet. It is connected to a stateful firewall, FW–1, which has three interfaces (firewalls that have more than two interfaces are referred to as *multi-homed*). The zone connected to the right interface of FW–1 is referred to as a *screened subnet* though it is more commonly (though somewhat inaccurately) referred to as a *De-Militarized Zone* (DMZ). It is labelled *DMZ–1* in Fig. 21.2.

A DMZ, in the true sense, is the area between two firewalls. In Fig. 21.2, the zone between firewalls FW–1 and FW–2 is a real DMZ labelled *DMZ–2*. Demilitarized zones are so called because they often host servers that are accessible to the Internet and also to the internal network. Because they are accessible to the public, they are the most likely machines to be compromised in the entire network. For this reason, machines in the DMZ often run "hardened" versions of operating systems. Also, unnecessary services on these machines should be removed and newly available patches for these machines should be deployed expeditiously. In addition, an IDS/IPS should be placed in the DMZ to generate alerts in the event of an intrusion.

Once a machine in the DMZ is compromised, other machines in the DMZ could get infected. The next step could well be compromise of the internal machines. This is more likely if a compromised machine in the DMZ shares a trust relationship with an internal machine. Hence, the firewall that separates the DMZ from the internal network is a critical component of the security architecture.

*DMZ–1* contains the *publicly accessible servers*. These include the web server, the external e-mail server, and the DNS server. All incoming mail from the Internet is received by this e-mail server, which checks for virus signatures and spam mail. The DNS server resolves names of publicly accessible servers. However, care should be taken to ensure that it does not contain address records of any of the internal machines.

*DMZ–2* contains the *internal e-mail server*. This is the server that hosts the mailboxes of the company employees. It handles the sending and receiving of all mail between internal parties. It periodically establishes a connection to the external mail server (in *DMZ–1*) to retrieve all incoming mail. Outgoing mail (from the internal network to the Internet) can be handled in several ways. The internal mail server can set up an SMTP connection to a remote mail server to transfer mail.

https://hemanthrajhemu.github.io

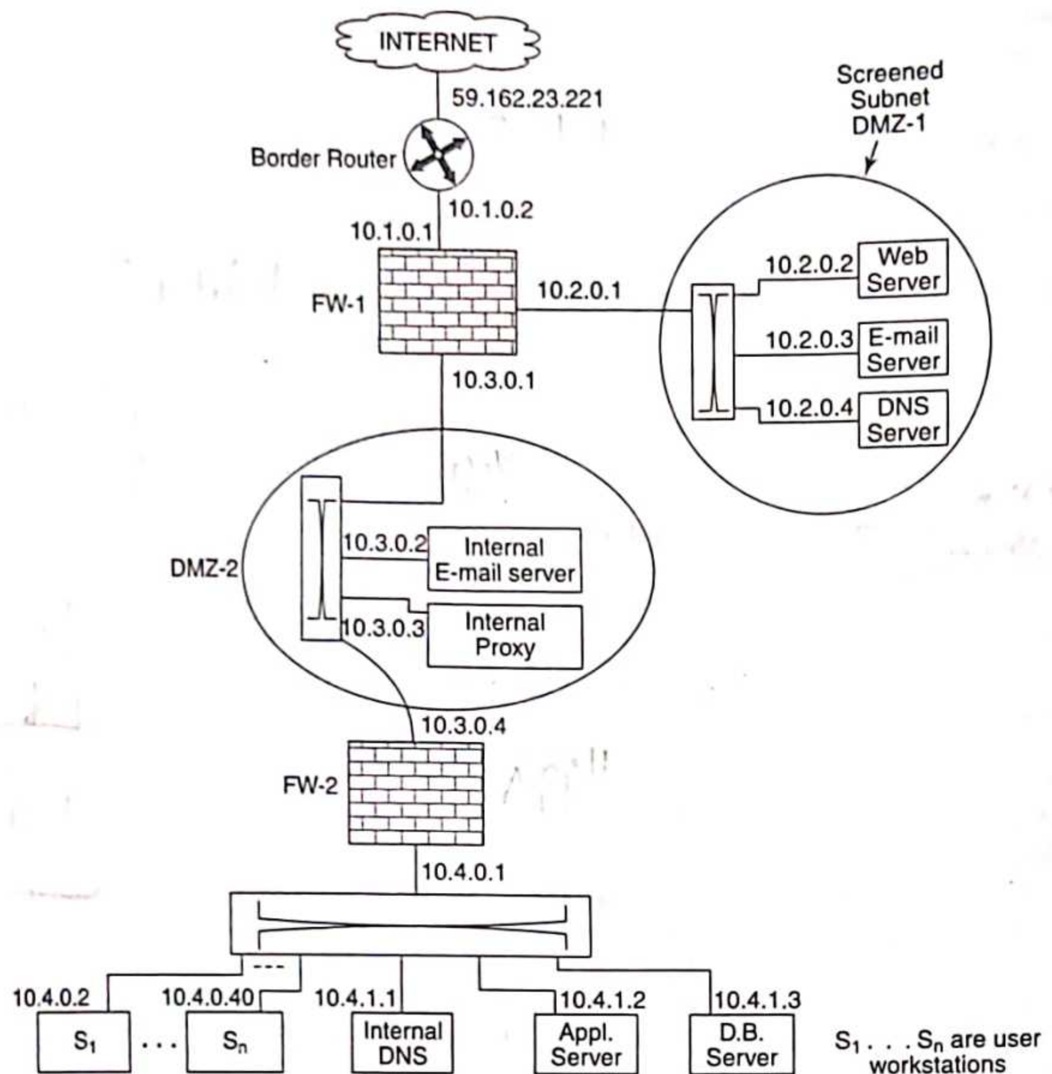



**Figure 21.2** *Firewall placement*

Alternatively, it can connect to the external mail server (in *DMZ–1*) and use it to relay all outgoing mail.

*DMZ–2* also contains an *Internet proxy server*. All internal users who wish to access external webpages connect to the proxy. The proxy authenticates the internal user and decides whether a page can be accessed (different restrictions might apply to different classes of users). The proxy scans incoming webpages for virus signatures and objectionable content. Finally, the proxy also performs caching of webpages.

The *internal network* contains application servers, database servers, and user workstations. It also has an internal DNS server. This DNS server is different from the external DNS server in that it provides mappings between the domain names of the internal machines and their IP addresses. The internal machines all have private addresses. It is neither necessary nor desirable for third parties on the Internet to be aware of the private addresses of the internal machines. Hence, this DNS server is placed in the internal network.

https://hemanthrajhemu.github.io

A feature of the security architecture in Fig. 21.2 is that services such as DNS and e-mail are *split*; that is, there is an internal DNS server as well as an external one. Likewise, there is an internal e-mail server and an external one. Generally, no external connection should be allowed to the internal servers. Connections in the reverse direction from the internal servers to hosts on the Internet should either be forbidden or severely restricted.

### 21.2.2 Firewall Configuration

In the previous subsection, we listed the servers in different security zones and specified their functionality. In order to design the ruleset of a firewall, we need to identify all the possible authorized connections that might be set up between pairs of machines in two different zones adjacent to the firewall. We first present a simplified version of the ruleset for firewall FW–2 (Table 21.2).

**Table 21.2** *Simplified ruleset for firewall, FW–2*

| *No.* | *From IP Addr.* | *From Port* | *To IP Addr.* | *To Port* | *Protocol* | *Action* |
|---|---|---|---|---|---|---|
| 1 | * | * | Internal | * | * | Drop |
| 2 | User | * | Int_Mail_S | 25 | SMTP | Accept |
| 3 | User | * | Proxy | 80 | HTTP | Accept |
| 4 | * | * | DMZ-2 | * | * | Drop |

*Wildcard

The first rule states that no machine from any other security zone is permitted to establish a TCP connection to any internal machine. Rules 2–4 assert that, other than connections from internal stations to the internal mail server (on port 25) and web proxy (on port 80), no other connections are permitted to *DMZ–1*, *DMZ–2*, or the Internet.

Table 21.3 shows the ruleset for firewall FW–1.

**Table 21.3** *Simplified ruleset for firewall, F1*

| *No.* | *From IP Addr.* | *From Port* | *To IP Addr.* | *To Port* | *Protocol* | *Action* |
|---|---|---|---|---|---|---|
| 1 | * | * | DMZ-2 | * | * | Drop |
| 2 | Int_Mail_S | * | Ext_Mail_S | 25 | SMTP | Accept |
| 3 | Internet | * | Ext_Mail_S | 25 | SMTP | Accept |
| 4 | Internet | * | Web_S | 80 | HTTP | Accept |
| 5 | Internet | * | DNS-S | 53 | UDP | Accept |
| 6 | * | * | DMZ-1 | * | * | Drop |
| 7 | Proxy | * | Internet | 80 | HTTP | Accept |
| 8 | Ext_mail_S | * | Internet | 25 | SMTP | Accept |
| 9 | * | * | Internet | * | * | Drop |

Rule 1 in Table 21.3 states that no TCP connection is to be established to any machine in *DMZ–2* from any machine in *DMZ–1* or the Internet. Rule 2 states that the external mail server can accept connections from the internal mail server to receive incoming mail or to send outgoing mail. Rule

https://hemanthrajhemu.github.io

3 allows connections to the external mail server from mail servers on the Internet to deposit incoming mail. Rules 4 and 5 permit connections from the Internet to the organization's web server and external DNS server, respectively. Rule 6 states that no other connections may be set up to any machines in *DMZ-1* for any other purpose.

The Internet proxy in *DMZ-2* and the external mail server are permitted to make connections to machines on the Internet to access webpages and to send outgoing mail (Rules 7 and 8). Rule 9 confirms that no other connection from the organization's machines to the Internet for any other purpose is allowed.

## 21.3 PERSONAL FIREWALLS: A CASE STUDY

*Netfilter* [which comes bundled with Linux (version 2.4 and higher)] is an inexpensive alternative to a special-purpose firewall appliance. *IPTables*, as Netfilter is more commonly called, is a utility employed to configure Netfilter kernel modules. It includes a repertoire of expressive user-space commands used to build firewall rules. IPchains, the predecessor of IPTables, supported packet filtering and NAT. IPTables also performs stateful packet inspection, packet mangling and can be programmed to limit the rates of various flows.

### 21.3.1 Chains and Tables

With IPTables, rules are grouped together on the basis of *functionality* into three *tables*.

*Filter*. This table includes the standard rules for filtering traffic based on source/destination IP address, source/destination port, type of protocol, MAC address, etc. Rules related to connection tracking are also part of this table.

*NAT*. The rules in this table perform source or destination IP address translation. This table may also contain rules that perform port translation.

*Mangle*. Rules for changing certain fields in the IP header belong to this table. The fields include TTL (Time to Live) and ToS (Type of Service).

Not all rules in each of the three tables residing in host, H, are applicable to every packet seen by H. There are two questions to be answered for every packet – (a) whether a rule is applicable and (b) when is it applicable. The answers to these questions partly depend upon which of the three categories a packet belongs to:

- Packet entering H destined for H
- Packet leaving H whose source is H
- Packet entering H which must be forwarded to another host

Consider, for example, a packet entering H. A routing decision within H is taken whether the packet is destined for H or whether and which interface it should be forwarded through. For this packet, a rule involving Destination NAT (DNAT) should be taken *before* the routing decision. A filtering rule, on the other hand, should be processed *after* the routing decision. There is thus sanctity in the order of the three operations: D-NATing, routing, and filtering.

IPTables also groups firewall rules into *chains*. Chains are comprised of subsets of rules from different tables. Chains define the order in which the different subsets of rules should be processed for the three classes of packets enumerated above. The principal chains are as follows:

- PREROUTING
- INPUT

https://hemanthrajhemu.github.io

- FORWARD
- OUTPUT
- POSTROUTING

The order of chain traversal for the three different categories of packets is shown in Fig. 21.3. Each chain also shows the order in which subsets of rules from the three tables are executed.

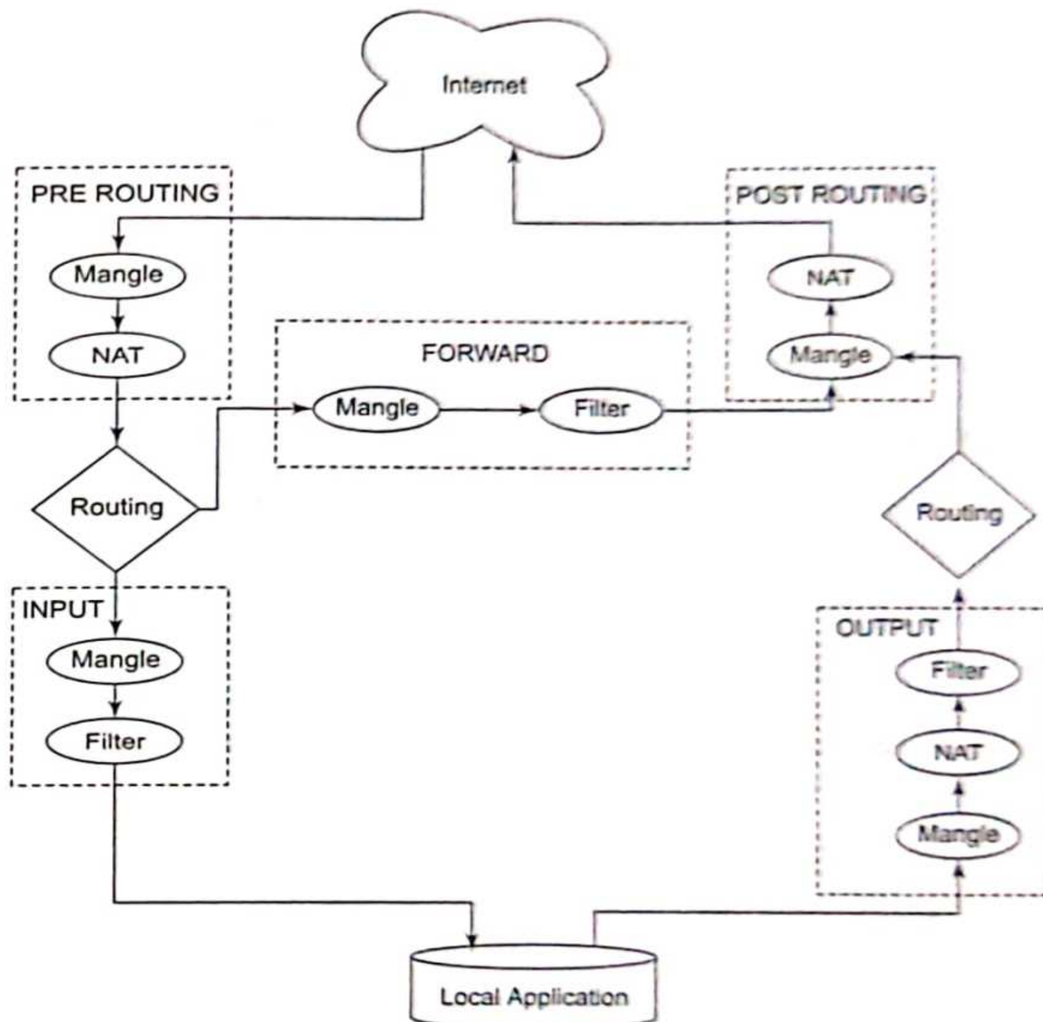



**Figure 21.3** *Chains and tables in Netfilter/IPtables*

## 21.3.2 Commands

The command syntax for creating a rule in IPtables is:

*iptables* [-t `table`] command [packet-attributes] [-j action]

The *command* field, for example, allows the user to specify that a given rule should be inserted or appended to a particular chain. It can also specify that a particular rule should be deleted or replaced. In addition, there are commands to create new chains or delete existing ones.

Packet attributes such as source/destination IP address or port number, the network interface a packet arrives on, protocol, etc. may be included in the *packet-attributes* field. If an incoming

https://hemanthrajhemu.github.io

packet matches this set of attributes, the *action* field of the command specifies what should be done with the packet.

In the event of a match, possible actions to be taken on a packet are ACCEPT, DROP, or REJECT. The difference between DROP and REJECT is that, in the latter, an explicit error message is sent back to the source. The action field may also specify a chain (or sub-chain) to be branched to.

### *Example 21.1(a)*

```
iptables –t filter –A INPUT –s 240.01.02.03 –j ACCEPT
```

The above command only *accepts* packets with IP source address = 240.01.02.03. This rule is appended to the *filter* table and is executed for packets traversing the INPUT chain. As shown in Fig. 21.3, this rule is only applicable to incoming packets that are not forwarded.

### *Example 21.1(b)*

```
iptables –A INPUT –s !240.01.02.03 –p udp –j DROP
```

The above rule is also inserted into the filter table (the default table is filter) and will be executed in the INPUT chain. According to this rule, all UDP packets from an IP source address other than 240.01.02.03 should be dropped.

### *Example 21.2*

Consider a LINUX IPTables-based firewall behind which is a small LAN. The firewall is connected to the Internet through a router (Fig. 21.4). The internal hosts include a web server(WS). The external IP address of the access router together with the internal IP addresses of the web server, firewall, and access router are all shown. The network interfaces of the firewall connected to the internal network and to the access router are designated *int* and *ext*, respectively. Two policies are to be implemented by the firewall. We explore how the policies are translated into rules using IPTables.

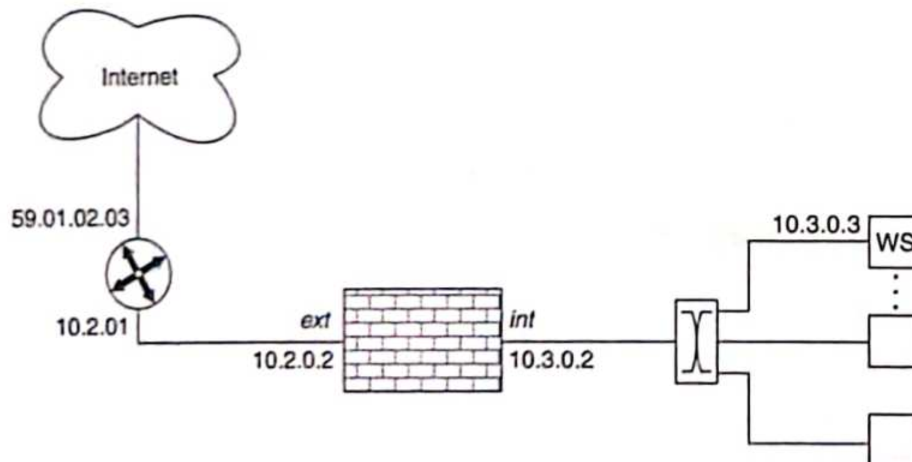



**Figure 21.4** *IP tables protecting a small LAN*

*Policy 1.* Any client on the Internet can set up a TCP connection to the web server but to no other internal machine.

https://hemanthrajhemu.github.io

*Policy 2.* All internal machines behind the firewall should have private addresses and the addressing schema should not be visible to clients on the Internet.

The following rule is composed to implement Policy 1:

```
iptables –A FORWARD –p tcp –i ext –o int –d WS –dport 80
         –sport 1024:65535 –m state –state NEW, ESTABLISHED
         –j ACCEPT
```

This rule permits the entry of external packets provided their TCP destination port number is 80 (port 80 corresponds to HTTP). The source port number in the TCP header of the packet should be a number between 1024 and 65535 (these are the so-called ephemeral port numbers). This rule implicitly permits all packets (handshake and data) that are part of such connections.

The " –m state" option in the above rule specifies that a packet should be *m*atched on attributes related to the state of a connection. A connection is recognized as being in one (or more) of the following states: ESTABLISHED indicates that the packet is part of an already established and not yet terminated connection. (RELATED has a similar connotation except that the incoming packet is associated with an already existing connection such as is the case with FTP data transfer). NEW indicates that the current packet is participating in the establishment of a new connection even though the connection has not yet been completely established. If the connection is not in one of these three states, it is considered to be INVALID.

To communicate with the web server, clients on the Internet use destination IP address = 59.01.02.03 and destination port = 80. On receipt of such packets, the firewall realizes that since TCP port 80 is the HTTP port, the packet should be directed to the web server (internal IP address = 10.3.0.3). Hence, the firewall needs to change the destination IP address of this packet from 59.01.02.03 to 10.3.0.3. This is accomplished by the following IPTables command for creating a rule in the NAT table.

```
iptables –t nat - A PREROUTING - p tcp -o int - i ext
         –d 59.01.02.03 -- dport 80 -j DNAT --to-destination 10.3.0.3
```

Likewise, the source IP address of packets from the internal web server to destinations on the Internet needs to be changed. In this case, the source IP address should be changed from the web server's internal source IP address = 10.3.0.3 to the externally visible IP address of the firewall which is 59.01.02.03.

In addition to the five system-provided chains, administrators may create their own chains for handling special types of packets. For example, a special chain, ICMP_PACKETS may be defined for handling ICMP packets. Such a chain might be called from the INPUT chain by using the following rule:

```
iptables - A INPUT - p icmp - j ICMP_PACKETS
```

Thus, when the INPUT chain encounters an ICMP packet, it jumps to the ICMP_PACKETS chain. The rules in the ICMP_PACKETS chain are executed. If a packet is accepted by the ICMP_PACKETS chain, it "returns" to the rule in the INPUT chain just after the rule that "called" the ICMP_PACKETS chain. In this sense, calling and returning from another chain are analogous to calling a function and returning from it in a program.

https://hemanthrajhemu.github.io

## SELECTED REFERENCES

[WACK02] is a useful introduction to many aspects of firewalling including different types of firewalls, placement, configuration issues, etc. [WOOL04] enumerates some of the most common errors in firewall configuration and their actual frequency of occurrence in the real world. [LIU08] studies subtle issues in configuring firewall rulesets including features such as completeness, consistency, and compactness. [IPTAB] is a comprehensive tutorial on IPTables.

## OBJECTIVE-TYPE QUESTIONS

**21.1** Tasks performed by firewalls include
- (a) access control
- (b) packet integrity checking
- (c) IP address spoofing detection
- (d) port address translation

**21.2** Which of the following is/are performed by a web proxy?
- (a) webpage caching
- (b) authentication
- (c) spam filtering
- (d) malware signature detection

**21.3** Which of the following make(s) filtering decisions based on application payload?
- (a) packet filter
- (b) stateful packet inspection firewall
- (c) deep inspection firewall
- (d) reverse proxy

**21.4** Which of the following is/are usually placed in the outer DMZ of an organization?
- (a) web server
- (b) database server
- (c) application server
- (d) authentication server

**21.5** Which of the following may be performed by the Linux firewall, IPTables?
- (a) traffic rate limiting
- (b) filtering on MAC address
- (c) authentication
- (d) stateful packet inspection

**21.6** Which of the following operations is/are performed in the INPUT chain of IPTables?
- (a) NAT
- (b) Filter
- (c) Route
- (d) Mangle

**21.7** A packet passes through a host running IPTables. (The host is neither a source nor a sink for that packet.) Such a packet encounters the following chains in that host
- (a) INPUT
- (b) OUTPUT
- (c) PREROUTING
- (d) POSTROUTING
- (e) FORWARD

## EXERCISES

**21.1** Can a firewall be configured to defend against DDoS attacks? Can it be configured to defend against different kinds of worm attacks? In each case, explain why or why not.

**21.2** Consider the security/firewall architecture of an organization depicted in Fig. 21.2. Suppose the organization allows its customers and suppliers the ability to transact with it over the net.
- (a) State three distinct reasons why the organization's web server in DMZ 1 might wish to initiate a connection with any of the internal machines.
- (b) How would the rules in Tables 21.2 and 21.3 change to accommodate such traffic?

https://hemanthrajhemu.github.io

**21.3** An organization allows its employees a remote log-in facility through an IPSec-based VPN. With the help of neat sketches, show different possible placements of a VPN terminator with respect to the organization's firewalls. Discuss the pros and cons of each placement.

**21.4** The servers in the (external) DMZ are typically publicly accessible over the Internet. Hence it is recommended that they be "hardened." Suggest any three ways of hardening these servers and their effectiveness.

**21.5** You will need a Linux machine to run this experiment (kernel version 2.4× and above). Most such machines have the IPTables package already installed. For further installation details visit http://www.netfilter.org/projects/iptables/index.html.

The purpose of this exercise is to experiment with IPTables commands. You will need super-user privileges to run these commands. Try out some of the commands contained in this chapter.

Also enter each of the following and examine what effect they have.

```
iptables   – L
iptables   – A INPUT    – p tcp --dport 22        – j DROP
iptables   – A OUTPUT   – d x.x.x.x               – j DROP
iptables   – A INPUT    – p icmp --icmp-type 8    – m limit -- limit 20/minute
           – j ACCEPT
```

**21.6**

| *Rule number* | *From IP address* | *To* | | *Protocol* | *Action* |
|---|---|---|---|---|---|
| | | *IP address* | *Port* | | |
| 1 | Any | Any | 80 | TCP | Accept |
| 2 | BadGuy | Any | Any | Any | Reject |
| 3 | GoodGuy1 | Any | Any | TCP | Accept |
| 4 | GoodGuy1 | Any | Any | UDP | Accept |
| 5 | GoodGuy2 | Any | 25 | TCP | Accept |
| 6 | Any | Any | Any | UDP | Reject |

The above rules were written to reflect a given security policy.

(a) Would the same policy be implemented by interchanging rules 1 and 2. Explain.

(b) Is any of these rules redundant? If so, which?

(c) A set of rules is incomplete if there exists at least one packet that does not satisfy any of the predicates (columns 2–5) in the rule set. Is the above set of rules complete? Justify your answer.

**21.7** We had earlier studied protocol tunnelling in the context of IPSec (Chapter 13). Another example is that of an application protocol tunnelled within an HTTP message. HTTP/web traffic on port 80 is usually allowed by many firewalls. So, an application protocol blocked by a firewall may be tunnelled in an HTTP message.

How would a firewall be designed to filter traffic that is part of a forbidden protocol tunnelled through HTTP or any other protocol accepted by the firewall?

## ANSWERS TO OBJECTIVE-TYPE QUESTIONS

**21.1** (a)(d) **21.2** (a)(b)[sometimes(d)] **21.3** (c)
**21.4** (a) **21.5** (a)(b)(d) **21.6** (b)(d) **21.7** (c)(d)(e)

https://hemanthrajhemu.github.io

Chapter 22

# Intrusion Prevention and Detection

## 22.1 INTRODUCTION

An intrusion is the act of gaining unauthorized access to a system so as to cause loss or harm. Examples of intrusions include the following:

- Unauthorized login to a system by illegally acquiring a password (through, for example, a password guessing attack).
- Worm infections that use the system as a launch pad to spread and infect other machines.
- Injection of spyware that passively monitors the activities of the user and relays this information back to the attacker (over the Internet, for example).
- Flooding the host with spurious connection requests that attempt to exhaust the target's resources – processing power, memory, or communication bandwidth.

Two ways of handling attempted intrusions are *intrusion prevention and intrusion detection.* The latter stems from the realization that despite our best preventive efforts, intrusions nevertheless do take place.

An analogy with human disease might help clarify the two approaches. Modern day ailments like diabetes and high blood pressure are known to be linked to lifestyle choices. Preventive "treatment" for these ailments includes regular exercise, a stress-free life, and a diet rich in fruits and vegetables. However, an individual who strictly follows these three practices cannot be guaranteed freedom from these medical problems.

We know that regular health check-ups (especially for those beyond 50 years of age) can help prolong life. In the current context, periodic monitoring of blood sugar levels and blood pressure are strongly recommended so that timely action can be taken if there is deviation from the norm. The regular health check-ups are analogous to intrusion detection, while the positive lifestyle choices are analogous to intrusion prevention.

In Sections 22.2 and 22.3, we look at different kinds of intrusion detection systems. We also present a case study that helps clarify the difference between measures intended to provide intrusion detection and intrusion prevention. Sections 22.4 and 22.5 detail how two of the principal attacks are handled: DDoS and malware attacks.

https://hemanthrajhemu.github.io

## 22.2 PREVENTION VERSUS DETECTION

### 22.2.1 Prevention

Intrusion prevention anticipates various kinds of attacks and takes steps to forestall their occurrence. Take the case of software vulnerabilities discussed in Chapter 18. On the one hand, *programmers* should adopt practices that help reduce or eliminate software vulnerabilities. The use of safe string manipulation functions in C/C++ and the use of parameterized SQL queries are some of the practices recommended to protect against buffer overflow and SQL injection attacks, respectively. Likewise, sanitizing user input from HTML forms is one preventive measure against cross-site scripting attacks.

Another set of preventive measures may be taken by the *computing system* (hardware, compiler, or operating system) to provide a second line of defence. Here again, the buffer overflow vulnerability is a prime example. Making the stack non-executable prevents one class of buffer overflow problems. On the other hand, using a canary value (Chapter 18) on the stack detects a buffer overflow and thus helps prevent its possible exploitation.

Many attacks can be prevented by properly configuring firewalls and timely application of software patches. Extensive training should be imparted to system administrators on this and related tasks. Finally, users should be trained to adhere to sound security practices such as password protection and be educated on the variety of social engineering attacks.

One final aspect of intrusion prevention is *deterrence*. Hacking, whether for fun or profit, is a criminal offence. The penalty for perpetrating cyber attacks can far outweigh the "thrill" of successful hacking or the financial gain – disseminating messages such as these can be serious deterrents to wannabe hackers and cyber criminals.

### 22.2.2 Detection

An intrusion detection system (IDS) (Fig. 22.1) performs the following three tasks:

1. First, it *monitors* "events of interest" occurring in the target system or in the network. An event of interest may be a system call (a call made to the operating system) to, for example, open a file containing sensitive data. Another event of interest may be the attempted establishment of a TCP connection from a specific IP address to a certain port.
2. An IDS generates a large amount of data which it then *analyzes* and converts into valuable information to be used by system administrators. The information gleaned may lead to conclusions such as "the source addresses in 90% of the packets received in the past 5 min are never-seen-before IP addresses."
3. The IDS creates a database of "interesting events." It raises an *alert* each time it observes any such event. For example, it might be programmed to raise an alert if more than 70% of the packets received within the last 10 min arrive from never-seen-before IP addresses. These are examples of *thresholds* and parameters set by a human. On the other hand, it would be highly desirable if the IDS were capable of learning what is normal behaviour, detecting anomalous events when they occur, and flagging such events.

There are a number of key questions related to IDS functioning and deployment:

- What are the variables that the IDS should monitor?
- When should an alert be raised? When should an alarm be sounded?
- Where should the IDS be placed?

Before attempting to answer these questions, we present a case study related to password abuse. This is intended to illustrate the difference between intrusion prevention and detection.

https://hemanthrajhemu.github.io

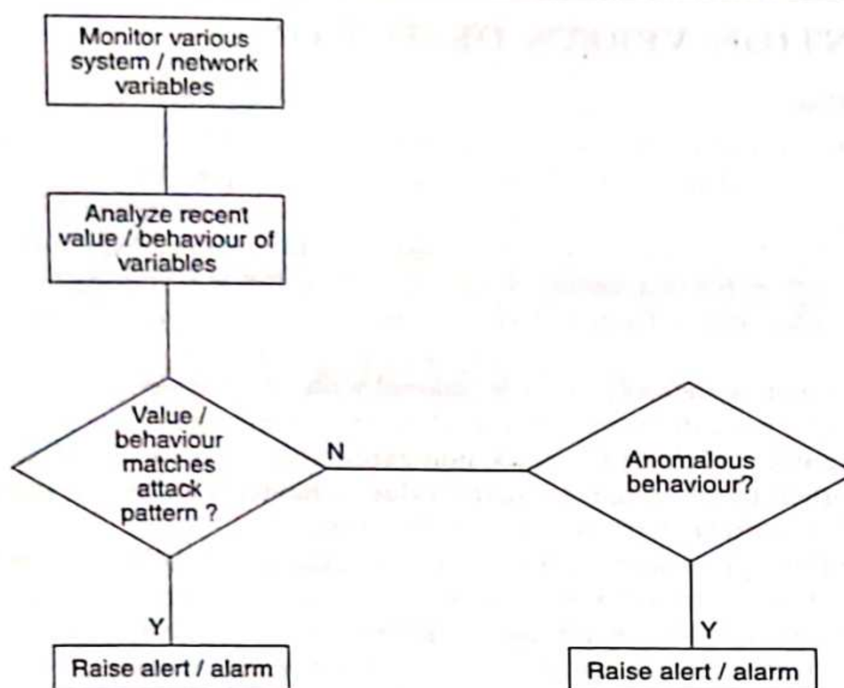



**Figure 22.1** *Tasks performed by an IDS*

### 22.2.3 Case Study: Unauthorized User Logins

To prevent unauthorized logins owing to compromised passwords, the following should be adhered to:

1. A password should be at least eight-characters long, hard to guess, and include at least one non-alphanumeric character.
2. A password should be changed at least once in two months.
3. Passwords should be stored securely (not written on sticker pads) and should not be communicated to friends, relatives, and co-workers.
4. After three consecutive unsuccessful attempts to a specific account, the system should be designed to disable all further log-in attempts for the next 20 minutes. (The spirit of this rule is important though the actual numbers may vary.)

Rules 1 and 2 should be observed by the user though they can and must be enforced by the system. Rule 3 involves the user alone, while rule 4 involves the system alone. These rules are all measures intended to *prevent* intrusion. As a further preventive measure, a high-security organization may mandate two-factor authentication – passwords in conjunction with biometrics.

In addition to prevention, an IDS may also be deployed to monitor suspicious log-ins. For example, consider an employee who, for 5 years, has never logged in outside of office hours (between 9 am and 5 pm). If now, after 5 years, the same user were to log in at 4 am, the IDS should raise an alert.

Consider that 10 unsuccessful logins have been made to the same login account over a 2-hour period. An IDS should be "smart" enough to sense a password guessing attack even if these login attempts were sufficiently spaced out. It should respond by disabling all further log-in attempts until the system administrator has been alerted.

https://hemanthrajhemu.github.io

The last two mechanisms are reactive – they monitor the system for anomalous activity and trigger an alert when suspicious activity is observed. The password protection approaches, on the other hand, are preventive and designed to minimize the possibility of a break-in.

## 22.3 TYPES OF INTRUSION DETECTION SYSTEMS

A real-world IDS monitors and mines hundreds of variables for interesting patterns. Table 22.1 shows a sample of variables together with a condition that may trigger an alert. Some of the variables are mere *bit patterns* in the packet header or the packet payload. Other variables are counts of a certain occurrence within a *time interval*. We next classify intrusion detection systems based on their functionality.

**Table 22.1** *Events of interest to an IDS*

| *Variable monitored* | *Event of interest* | *Possible attack* |
|---|---|---|
| No. of accesses to specific file | Tenfold increase over norm | DoS attack or flash event |
| Login frequency to particular account | Unusually high | Attempted break-in |
| No. of distinct source IP addresses of arriving packets | Very high | Worm attack |
| Ratio of ARP request packets to ARP response packets | >> 1 | Network scan to identify local active hosts |
| Ratio of TCP SYN packets to TCP FIN packets | >>1 | Possible DoS attack |
| Percentage of half-open TCP connections | Sudden surge | Possible DoS/DDoS attack |
| TCP header flags | Invalid combination | Port scan, OS fingerprinting |
| TCP connection establishment | Unused destination port | Attempt to find which services are open |
| Payload of incoming packet | Specific byte sequence present | Specific worm attack |
| O.S. calls | Particular sequence of calls | Specific virus attack |

### 22.3.1 Anomaly versus Signature-Based IDS

*Anomaly based* intrusion detection involves making a determination whether the behaviour of the system is a *statistically significant departure* from *normal*. The IDS will have to learn, over time, what constitutes normal activity, usage, and behaviour. Moreover, the definition of what is normal may vary as a function of time of day or day of the week. What is normal may also vary from one host to another.

The first six conditions in Table 22.1 are examples of what an anomaly based IDS would monitor. Consider monitoring the number of TCP *SYN packets* (with the SYN flag set) and *FIN packets* (with the FIN flag set) in each successive 10-second interval. A disproportionate number of SYN packets vis-a-vis FIN packets indicate several half-open TCP connections and possibly the onset of a SYN flooding attack. We discuss the prevention and detection of these attacks in greater detail in the next section.

*Signature-based* intrusion detection (also called misuse detection) works by identifying specific patterns of events or behaviour that portend or accompany an attack. A signature-based IDS

https://hemanthrajhemu.github.io

maintains a *database of known signatures*. It attempts to obtain a match between the currently observed behaviour of the system and an entry in this database. A real-world signature-based IDS will have thousands of attack signatures against which to compare. An example of an attack signature is a *specific byte sequence* in a worm payload.

To an anomaly based IDS, it is the *absence* of normal behaviour that presages an attack while to a signature-based IDS it is the *presence* of a specific signature that raises an alert. Detecting a zero-day worm (one whose signature is not included in the database of an IDS) is a challenge. On the other hand, it is possible that the spread of the worm has caused much network traffic congestion and greatly increased CPU utilization on infected machines. In that case, it is more likely that an anomaly based IDS will detect the attack. But the latter may be unable to pin-point the exact cause of the anomaly.

### 22.3.2 Host-based versus Network-based IDS

An IDS that captures information about packets flowing through the network is referred to as a *network-based IDS*. For reasons of performance, it is common to have stand-alone appliances that perform network-based intrusion detection. These typically run only the IDS and are hence not vulnerable to various worm and virus attacks. They may be deployed at multiple points in a large organization.

A *host-based IDS* is typically implemented in software and resides on top of the host's operating system. Its main job is to monitor the internal behaviour of the host such as the sequence of system calls made, the files accessed, etc. For this purpose, it makes use of *system logs, application logs, and operating system audit trails* to identify events related to an intrusion.

Operating system logs, for example, keep track of when users log in, the number of unsuccessful login attempts, the commands executed, network connections made, etc. Application logs keep track of which files have been opened or which registry keys have been accessed during the run of an application. They may also monitor what system calls were made and in what order.

Keeping track of modified files and file attributes can play a key role in host-based intrusion detection. For example, a change in the contents of core system libraries should arouse suspicion since these are rarely modified, if ever. *File system integrity checkers*, for example, compute a cryptographic hash on the contents of each file. They detect file changes by comparing the computed hash of a file to its stored hash.

Two desirable features of an IDS are *speed* and *accuracy*. Speed is especially important in fast-spreading Internet worms, for example. Early worm detection and an early response mechanism such as automated system shutdown can help reduce the number of infected machines.

The IDS should be able to detect every instance of an intrusion. An undetected intrusion is referred to as a *false negative*. Given the importance of not missing out on any intrusion, an IDS may go overboard and react to even innocuous events. We say that an IDS generates a *false positive* if it raises an alarm even though there is no intrusion currently occurring or about to occur. Two aspects of IDS accuracy are *sensitivity* and *selectivity* – high sensitivity implies a low false negative rate, while high selectivity implies a low false positive rate.

***Example 22.1***

Suppose packets containing a certain worm need to be filtered out by a deep inspection firewall. The worm accesses a file called *run.exe*. The worm payload includes this filename.

https://hemanthrajhemu.github.io

The firewall could be configured to filter out packets containing the string *run.exe*. However, in the process, innocuous packets containing strings such as *xrun.exe*, *rerun.exe*, *newrun.exe*, etc. will all be filtered out. Such false positives can be eliminated by enhancing the *specificity* of the search string. So, in addition to the file name, the directory and subdirectory names could also be specified as in

/winXP/system32/run.exe

The more specific search string prevents the occurrence of the *false positives* mentioned above. Now however, the attacker can defeat the system by using the following equivalent pathname in the body of the worm.

/winXP/system32/ . . /system32/run.exe

A packet carrying this string is not filtered out by the firewall, creating a false negative. One solution to this problem is to parse a filename and create an equivalent name in "canonical form" that can be compared with strings in a blacklist of suspicious filenames.

## 22.4 DDoS ATTACK PREVENTION/DETECTION

In Chapter 17, we saw how the denial of service attack could be made more potent by co-opting multiple agents to launch a coordinated attack on a given victim. Much research has been conducted on DDoS defence mechanisms which range from the preventive to the reactive. We explore some of these in this section.

### 22.4.1 DDoS Prevention

#### *Preventive Measures at the Host*

One possible way of handling SYN attacks is to drop requests for TCP connections. But this could result in *collateral damage* if the victim is unable to distinguish between SYN packets that are part of the attack and those from its legitimate clients. One way to reduce collateral damage is to categorize IP addresses as "almost certainly genuine", "probably spoofed", etc. The "almost certainly genuine" addresses are those with whom normal connections were established and terminated in the past. Under moderate load conditions, all incoming SYN requests are entertained. However, under rapidly increasing load, packets with unfamiliar source addresses are discarded with high probability.

Another strategy under high-load conditions is to allocate a full buffer of about 300 bytes for a given TCP connection request only upon completion of the three-way handshake. While the connection is still half-open, *minimal information* about it is stored in a hash table called the *SYN cache*. This information includes the TCP sequence numbers and source/destination addresses and ports.

An alternative to the SYN cache is the *SYN cookie*, which stores no state information at all for each half-open connection. Instead, the *responding machine* places a cookie within the Sequence Number field of the second handshake message. The cookie is computed as a hash function of the source address, destination address, source port, destination port, and a secret. The initiator of the connection dispatches the cookie it just received in its ACK message (third message of the three-way handshake). Upon receiving the ACK, the responder re-calculates the cookie and verifies that it matches the value enclosed in the received ACK. Only then does it reserve buffer space for the connection.

https://hemanthrajhemu.github.io

If the source IP address in the first message of the handshake were spoofed, the cookie in the second message would not be received by the initiator but by the machine corresponding to the spoofed IP address. The initiator would not be able to complete the three-way handshake since it does not know the cookie value. Hence, its connection request would not be granted buffer space.

The above schemes help prevent memory exhaustion at the victim's machine. However these schemes do not help reduce incoming attack traffic, which could cause the victim's network link to saturate. In the next section, we investigate approaches that throttle traffic through the core of the network well before it enters the victim's network.

### *Preventive Measures inside the Network*

An intuitively appealing approach to frustrating DDoS attacks is to implement measures closer to the source of the attack. One such measure is *egress filtering*. Most DDoS attack packets use spoofed source IP addresses. Address spoofing is employed to confuse cyber sleuths making it hard to pinpoint the true source of the attack. The perpetrator hopes to continue the attack for as long as desired and perhaps even resume it at a later point without being traced.

The egress router is the last router encountered by any packet generated inside the network before it exits that network and enters the Internet. Let $\mathcal{A}$ be the set of all externally visible IP addresses within the network (*behind* the egress router). The egress router examines the source address of each packet leaving it. If the address does not match any address in $\mathcal{A}$, it drops the packet. By thus *detecting* and *filtering spoofed packets*[1] it helps *prevent* DDoS attacks.

It should be noted that such a mechanism, while being effective, is unlikely to be universally deployed. There is not always sufficient motivation for an ISP to implement egress filtering since he sees no immediate incentive/gain in forestalling a DDoS attack on someone else's server albeit with zombies residing in his own network.

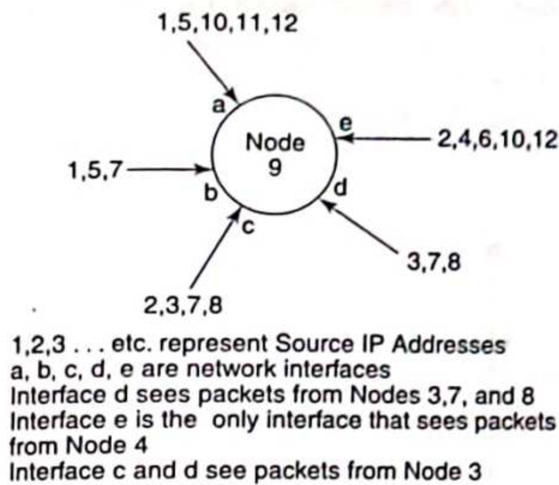
The idea of egress filtering has been extended to routers in the core of the Internet. Call a core router that performs filtering of spoofed packets a filtering router or simply *filter*. Internet routing is done based on the destination address contained in a packet. A filter, on the other hand, uses the packet's source address to make a decision on whether or not to discard the packet. To implement *Distributed Route Filtering* (DRF) [PARK01], a filter maintains, for each of its interfaces, the set of all source addresses from which packets arrive *en route* to some destination. The router uses BGP routing information to obtain the latest mapping between each of its interfaces and the subset of source addresses using that interface. The filtering decision is straightforward – if a packet with source IP address = S arrives via an interface that it should not have, that packet is assumed to be spoofed and is hence discarded.

Figure 22.2(a) shows an example of a router implementing DRF. Each of its interfaces is marked with the source addresses that use that interface en route to some destination. Note that packets from the same source may enter the router through different interfaces. For example, packets from source address 7 may arrive through interfaces *b*, *c*, or *d*. By way of clarification, two packets arriving from source address 7 via the same interface may have different destinations. Also, two packets bearing the same source–destination pair may arrive on two different interfaces if there exist multiple shortest paths between that source–destination pair.

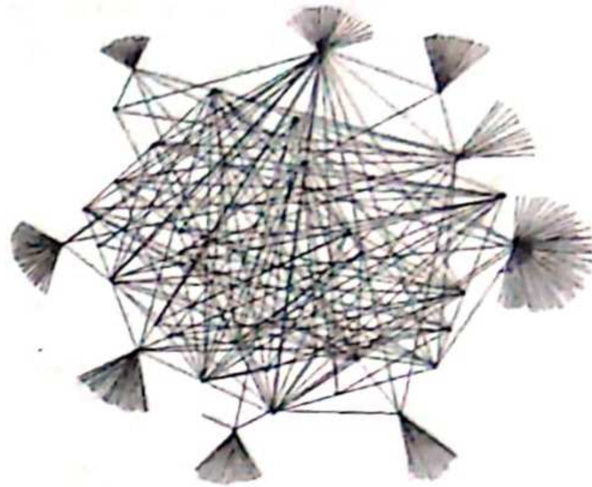
In the simplest implementation of the filter, the router checks whether a packet has arrived on one of its "acceptable" interfaces based *only* on the packet's source IP address. For example, a packet bearing source address = 7 arriving on interface *c* would be forwarded. However, another

[1]Not all spoofed packets will be detected.

https://hemanthrajhemu.github.io




(a) Router implementing DRF



(b) Internet Topology (Power Law Graph)

**Figure 22.2** *Distributed route filtering*

packet with the same source address but arriving on interface *e* would be suspected of having a spoofed source address and would be discarded [see Fig. 22.2(a)].

One of the issues in distributed router-based solutions is estimating the percentage of core routers that need to be retro-fitted with a filter for DRF to prevent DDoS attacks. Simulation results in [PARK01] indicate that excellent coverage against DDoS attacks is obtained if only about 18% of core routers are DRF-enabled. The reason for such an optimistic cost estimate is the peculiar "power-law" topology of the Internet. As shown in Fig. 22.2(b), a few nodes in the graph are connected to a large number of nodes and many nodes are connected to just a few – such a topology is called a power law graph.

Given that the Internet is made up of thousands of core routers, the number of core routers that have to be retro-fitted to support DRF is still a high number in an absolute sense. Moreover, the scheme depends on how fast BGP (Border Gateway Protocol) route updates are disseminated. Routing information changes in response to failed nodes or congested links. This information, in turn, decides whether an incoming packet at a router should be filtered out or not. A wrong decision could discard legitimate packets in addition to spoofed ones.

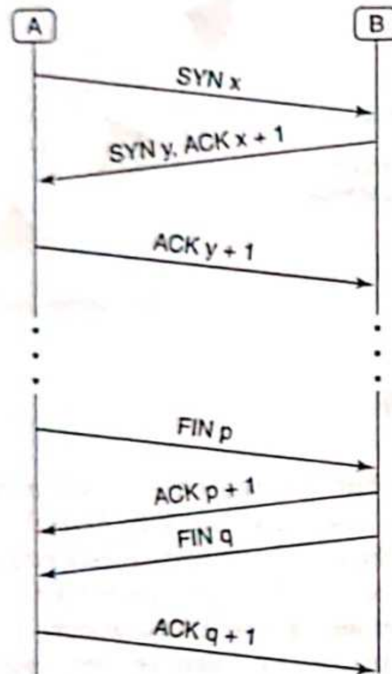
## 22.4.2 DDoS Detection

Egress filtering and DRF are preventive mechanisms. Another approach is to detect the onset of DoS and then take remedial action. We discuss detection next.

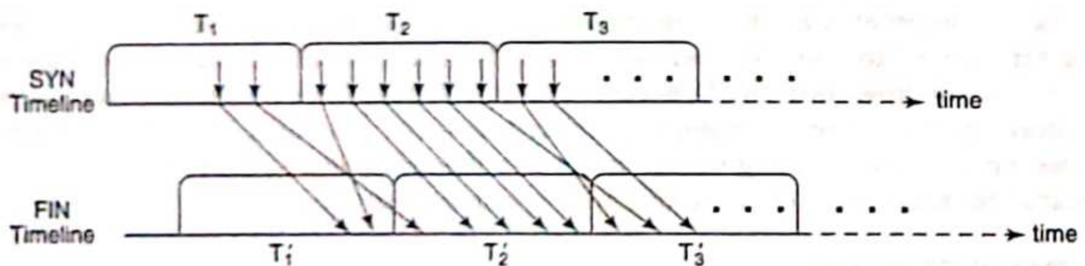
In a SYN flood attack, the victim sees a disproportionate number of SYN packets compared to FIN packets. By a SYN packet, we mean any incoming packet with the SYN flag set. Recall that such a packet is a TCP connection request packet. Likewise, FIN and RST packets are, respectively, those with the FIN and RST flags set. A FIN packet is sent by the side that wishes to terminate the TCP connection. If the other party agrees to termination, it responds with its own FIN packet. Thus, SYN and FIN packets usually occur in pairs.

https://hemanthrajhemu.github.io

TCP connections that *terminate normally* involve one SYN packet (from the client) and a corresponding FIN packet to initiate or confirm termination of a connection (see Fig. 22.3(a)). Thus, the total number of incoming SYN packets should equal the number of incoming FIN packets. In the event of a connection being aborted, an RST packet takes the place of the FIN. In most cases, the number of RST packets is a small fraction of the number of FIN packets, so we ignore them in this analysis.




(a) TCP connection establishment and termination




(b) SYN packets and their corresponding FIN packets

**Figure 22.3** *TCP SYNs and matching FINs*

Figure 22.3(b) shows two horizontal timelines – the top line shows the times of SYN packet arrivals and the bottom line shows the corresponding FIN arrivals. Time is slotted into fixed-length "observation intervals," $T_1$, $T_2$, ... , during which we record the number of SYN arrivals. The corresponding observation intervals for FINs, $T_1'$, $T_2'$, ... are shifted to the right by the average duration of a TCP connection.

https://hemanthrajhemu.github.io

To construct an anomaly detection system, we define the following variables as in [WANG04].

$S_i$ ≡ # of SYN packet arrivals in the $i$-th observation interval

$F_i$ ≡ # of FIN packet arrivals in the $i$-th observation interval

$D_i$ ≡ normalized difference between # of SYN and FIN packets in the $i$-th observation interval, i.e.,

$$D_i = \frac{S_i - F_i}{F_i}$$

$\mathcal{T}$ ≡ threshold for detection

Consider the *time series*,

$$D_1, D_2, D_3, \ldots$$

We now present different algorithms that attempt to detect the onset of a SYN Flood Attack by monitoring the above series.

*Algorithm 1.* Raise an alert if the most recently computed detection variable $D_i$ exceeds the threshold, i.e., $D_i > \mathcal{T}_1$.

Figure 22.4(a) shows $D$ versus time with the threshold set at $\mathcal{T}_1 = 90$. Some of the problems with this approach are as follows:

(i) The IDS may raise many *false alarms* since it bases its decision on point values. So, for example, at *time* = 16 in Fig. 22.4(a), the value of $D$ rises to 102 triggering an alarm. However, this alarm is unwarranted since the $D$ values at neighbouring points (around *time* = 16) are well below the threshold, $\mathcal{T}_1$. A modest spike in $D$ at just one point is very unlikely to result in memory exhaustion but it does cause the IDS to raise an alarm.

(ii) The values of $D$, between *time* 28 and 33 are just below the threshold, $\mathcal{T}_1$ (Fig. 22.4). The *cumulative* effect of the attack packets across the interval will cripple the system but this algorithm will not raise an alarm. Thus, SYN Flood attacks may evade detection by flying below the radar as shown in Fig. 22.4(a).

Our next two solutions, consider the cumulative effect of previous values of $D$.

*Algorithm 2.* Raise an alert if the "smoothed average" of the previous values of $D$ exceeds the threshold.

This approach uses the well-known technique of *exponential smoothing*. The decision variable at the end of the $i$-th observation interval is the smoothed average, $S_i$ computed using

$$S_i = \alpha D_i + (1 - \alpha) S_{i-1} \qquad 0 < \alpha < 1 \text{ and } S_0 = 0$$

The above recursive expression for $S_i$ can be expressed iteratively by repeated substitution of $S_{i-j}$ in terms of $S_{i-j-1}$. This yields

$$S_i = \alpha D_i + \alpha(1 - \alpha) D_{i-1} + \alpha(1 - \alpha)^2 D_{i-2} \ldots$$

For example, for $\alpha = 0.4$, we get

$$S_i = 0.4 D_i + 0.24 D_{i-1} + 0.144 D_{i-2} \ldots$$

The decision variable, $S_i$, is thus a *weighted sum* of $D_i$, $D_{i-1}$, $D_{i-2}$... with decreasing weights assigned to earlier values of $D$. Thus, "earlier" values of $D$ count less.

An alarm will be raised if $S_i$ exceeds a threshold $\mathcal{T}_2$. The value of $\mathcal{T}_2$ is set based on empirical data. If it is too low, it will result in many *false positives*. If it is set too high, it will result in *false negatives*. Another design parameter is the "smoothing constant," $\alpha$. If a value close to 1 is selected, it will give disproportionate importance to the most recent value of $D_i$. In the limiting case of

https://hemanthrajhemu.github.io

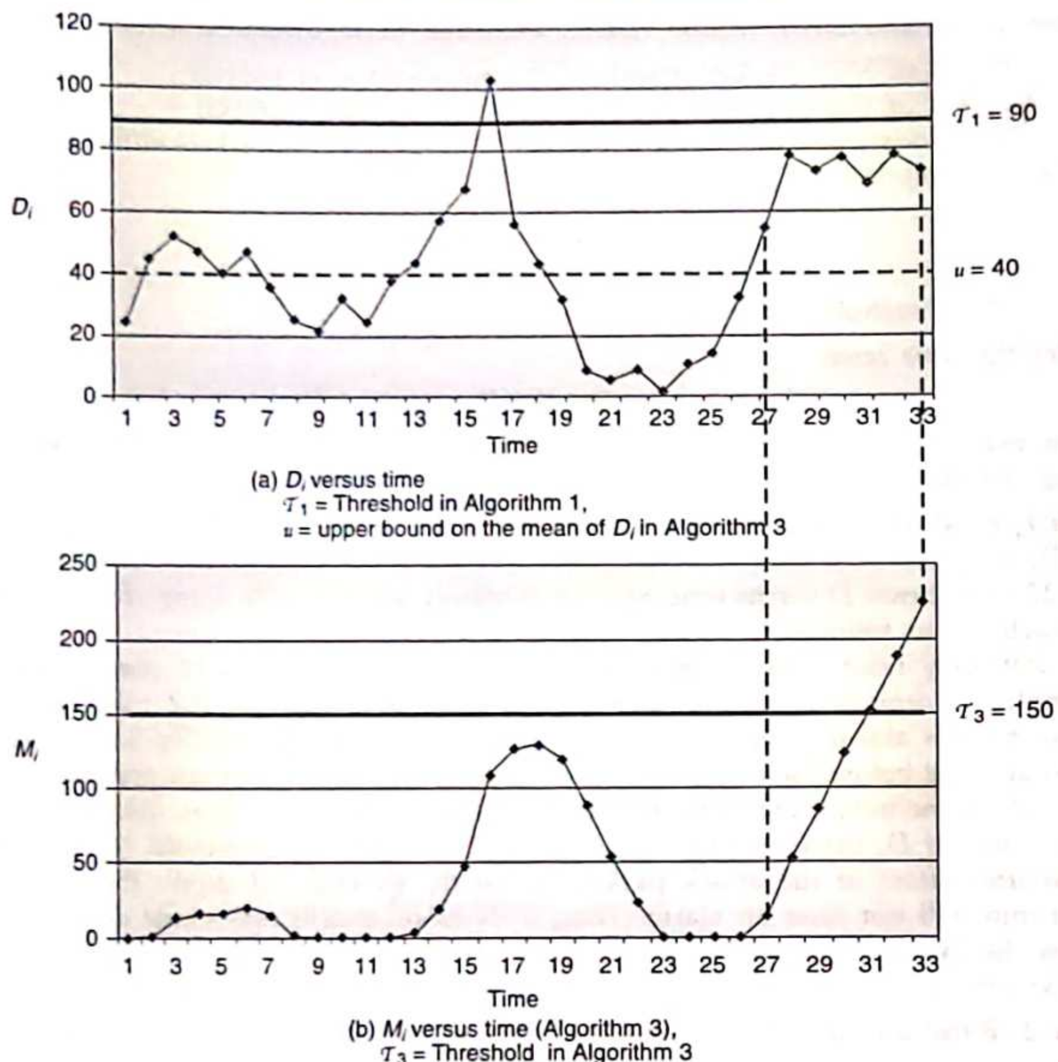



**Figure 22.4** *Variables monitored in SYN flood attack detection*

$\alpha = 1$, this reduces to Algorithm 1. On the other hand, the closer to zero $\alpha$ gets, the more even are the weights assigned to all values of $D_i$.

*Algorithm 3*. Define a *modified cumulative sum* of previous values of $D$. Raise an alert if this value exceeds a threshold.

This method makes use of a technique called *sequential change point detection* [BASS93]. Its application to SYN flooding is found in [WANG04].

During normal operation, the number of FINs will balance out the number of SYNs and hence $D_i$ will be close to 0.

Let $u$ be an upper bound on the mean of $D_i$ during normal operations. Let $D_i'$ be a shifted version of $D_i$, i.e., $D_i' \equiv D_i - u$.

The *decision variable*, $M_i$, used here is defined as

$$M_i = (M_{i-1} + D_i')^+$$

with

$$M_0 = 0$$

Here, the notation $x^+$ is defined as follows. $x^+ = x$ if $x > 0$, otherwise it is 0.

https://hemanthrajhemu.github.io

The IDS sounds an alarm at the end of the $j$-th interval if $M_j > \mathcal{T}_3$, where $\mathcal{T}_3$ is a threshold determined empirically. Figure 22.4(b) shows the value of $M_i$ versus time with a threshold of $\mathcal{T}_3$ = 150. At *time* = 1, $D_i < u$, so $M_i = 0$. Between *time* 2 and 6, $D_i$ is slightly above $u$, so $M_i$ *increases* monotonically. Between *time* 7 and 12, $D_i'$ falls below $u$, so $M_i$ decreases to 0 and remains there until *time* 12.

The interesting interval is between *time* = 27 and 33. Here, $D_i$ is consistently well above $u$ (though it is below the threshold in Algorithm 1). This causes $M_i$ to increase and it *overshoots the threshold* of $\mathcal{T}_3$ = 150. This causes an alarm to be sounded due to a cumulative build-up of SYN attack packets.

We thus see that with Algorithm 3 (cumulative sum method), the false positive and false negative encountered with Algorithm 1 are both avoided.

We have studied mechanisms for DDoS attack prevention and detection. We next turn our attention to IP traceback – attempting to trace the attack back to its source.

### 22.4.3 IP Traceback

The source IP addresses in the attack packets are typically spoofed – hence we cannot rely on them to identify the subnet from where the attack packets emanate. Instead, we attempt to identify the path (or paths) traversed by the attack packets.

There are two principal approaches to IP traceback – either the packet keeps track of the routers it has visited or each router keeps track of the packets passing through it. Solutions under the first approach use *packet marking*. The second approach is commonly referred to as *packet logging*. Both these methods require the co-operation of core routers. In addition, *hybrid approaches* using a combination of packet marking and packet logging have been proposed. In the rest of this subsection, we study a version of packet marking called *probabilistic packet marking* (PPM) followed by an implementation of packet logging.

#### *Probabilistic Packet Marking*

Consider, for a moment that every intermediate router were to append its 32-bit IP address to each packet it forwards. A packet on the Internet traverses about 10 hops on the average, so an extra 40 bytes would be needed to keep track of its path from source to destination. This is an unacceptable per-packet overhead. Instead, existing but infrequently used fields in the IP header are used to keep track of the routers visited.

The IP header has a *16-bit ID field*. This field provides support for packet fragmentation and re-assembly. Different networks have different restrictions on the size of the datagrams they can carry. They may split a datagram into two or more fragments and send each fragment separately. The router at the destination end has the responsibility for re-assembling the fragments to create the original packet. All the fragments carry the same number in their ID fields, so they can be identified for re-assembly.

On the assumption that the ID field is often unused, traceback schemes employing PPM use the ID field to store partial information on intermediate routers. But, given that the length of each IP address is 4 bytes, how can a packet store router address information in a 16-bit ID field?

The answer lies in computing a global fingerprint for each router – this is, say, 16 or fewer bits of the hash of a router's IP address. An intermediate router writes its fingerprint value into the ID field of a packet with probability $p$. Note that it could *over-write* a previously written fingerprint of a router closer to the source of the attack. To identify the perpetrator of the attack, the ingress router at the victim end will need to collect a sufficient number of packets that are all part of the same flooding attack.

https://hemanthrajhemu.github.io

We assume that each ingress router has a map of all upstream routers from it. Consider an attack path as shown in Fig. 22.5(a). Here, V is the victim and S the source of the attack. Since all packets have been *probabilistically* marked with the fingerprint of intermediate routers, an ensemble of attack packets will reveal the identities of various intermediate routers, thus helping to re-construct the attack path. Figure 22.5(b) shows two packets – Packet 1 was first marked by D and not over-written by any downstream router. Packet 2, on the other hand was first market by Router E and then its ID field was overwritten by Router C.

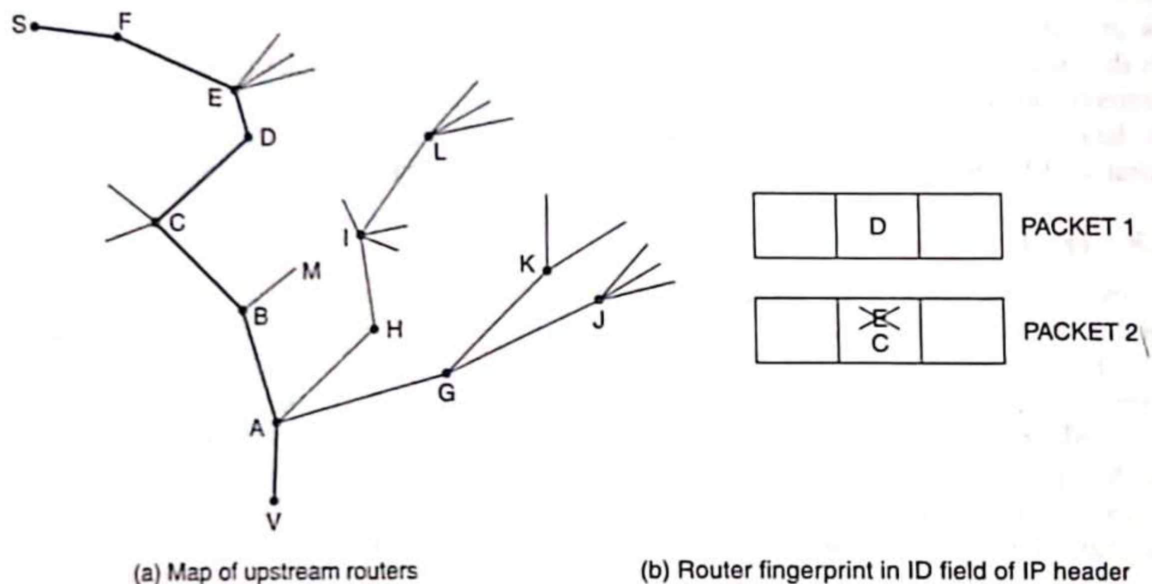



**Figure 22.5** *Probabilistic packet marking*

One problem with this approach is that it is more probable that upstream routers have their marks overwritten. For example, the probability that the mark made by Router F on a packet to V survives is $(1 - p)^5$. The probability that a packet arriving at V is marked by Router F and that the mark survives is $p(1 - p)^5$.

In general, the probability of an incoming packet at V having the mark of a router $h$ hops away is

$$p(1 - p)^{h-1}$$

An important consideration is the number of packets needed at V to reconstruct the attack path. This is closely related to $p$ and the distance between V and the attack source. (Exercise 22.7).

### *Packet Logging*

An alternative to packet marking is packet logging. Here, each router attempts to keep track of every packet that passes through it. While packet marking made use of the idea of a router fingerprint, packet logging makes use of the idea of a *packet fingerprint* or digest. This is computed using a well-designed hash function – one that distributes the hash values uniformly across all possible hash inputs.

An interesting feature of packet logging is that it can help *track even a single rogue packet*. First, assume that each router stores each packet received by it in the last 5 minutes. Suppose the victim wishes to obtain the exact path followed by a packet received by it. The idea is that the victim's

https://hemanthrajhemu.github.io

ingress router, A, queries each of its adjacent routers whether they have seen the packet. In Fig. 22.5(a), A would query B, H, and G. The router that responds positively, say B, then queries its neighbours, C and M. The one that responds positively then contacts its neighbours and so on until the source of the packet is traced.

The storage requirements at each router implementing this scheme could be prohibitive. Consider, for example, a router with six links. Assume that the link speeds are 1 Gb per second and that the router is operating at peak capacity. First, assume that a copy of each packet traversing the router is to be maintained for a period of 5 minutes. So the required amount of storage required in a router is about 1 Terabyte. (We assume that a DoS attack can be detected and traced back to its source in about 5 minutes). At the expense of some computation, we can bring down the storage requirements by storing only the digest or hash of a packet instead of its entire content.

The storage requirements can be further reduced by the use of a space-efficient data structure called the *Bloom Filter*. Here's how it works. We first introduce the required notation.

- Let $n$ be the maximum number of packets to be stored in a router in a given interval, say 7 minutes.
- Each time an element has to be inserted, one or more hash functions on that element need to be computed. Let $k$ be the number of distinct and independent hash functions used. $k$ is a design parameter.
- The output of each hash function returns a $w$-bit quantity.
- The Bloom Filter is basically a bit array. Let $m = 2^w$ be the size of this array.

*Packet "Insertion."* When a packet enters the router, the $k$ hashes are computed on its content. To speed up the computation, the hashes are only computed on the invariant parts of the IP header and a small part of the payload, say 10 bytes. Suppose the $k$ hash computations yield the values $i_1, i_2, i_3, \ldots i_k$. These $k$ hash outputs are used as indices into the bit array. To "insert" a packet, the bits in those positions are all set to 1. (If one or more of them were already set, they remain set.) Note that neither the packet itself nor the computed hash values are stored. Instead, only the relevant bits are set.

*Packet Presence Check*. To check if a packet, $\mathcal{P}$, is present in the Bloom Filter, compute the $k$ hashes on it as done during packet insertion. Suppose the $k$ hash computations yield the values $i_1, i_2, i_3, \ldots i_k$. Then, check whether each of the $i_1^{th}, i_2^{th}, \ldots i_k^{th}$ elements of the Bloom Filter are set. If even one of these elements = 0, $\mathcal{P}$ has not been encountered by this router.

If, however, all the $k$ elements are set, it is not necessarily true that the router has encountered $\mathcal{P}$ during the current time interval. Suppose that the output of one of the hash functions for at least one packet stored in the Bloom Filter is $i_1$ and the output of a hash function of at least one packet stored in the Bloom Filter is $i_2$ and so on. Then, since the values at positions $i_1, i_2, i_3, \ldots i_k$ are all set to 1, we will deduce that $\mathcal{P}$ is stored in the Bloom Filter even though it may not. We will incorrectly conclude that the router has encountered $\mathcal{P}$ in the time interval under observation, resulting in a *false positive*. On the other hand, there is no chance of a false negative with the Bloom Filter.

Figure 22.6 clarifies how a false positive may occur. A router has been presented with Packet $P_7$ and is being asked whether it has encountered this packet. So, it computes $h1(P_7)$, $h2(P_7)$ and $h3(P_7)$ and it looks to see if the corresponding bits in the Bloom Filter are set. As it turns out, these bits have been set respectively by the application of h2 on Packet $P_{1054}$, h1 on Packet $P_{309}$ and h1 on Packet $P_7$. So the system will conclude, rightly or wrongly, that it has encountered $P_7$ before. We next derive an expression for the probability of a false positive.

https://hemanthrajhemu.github.io

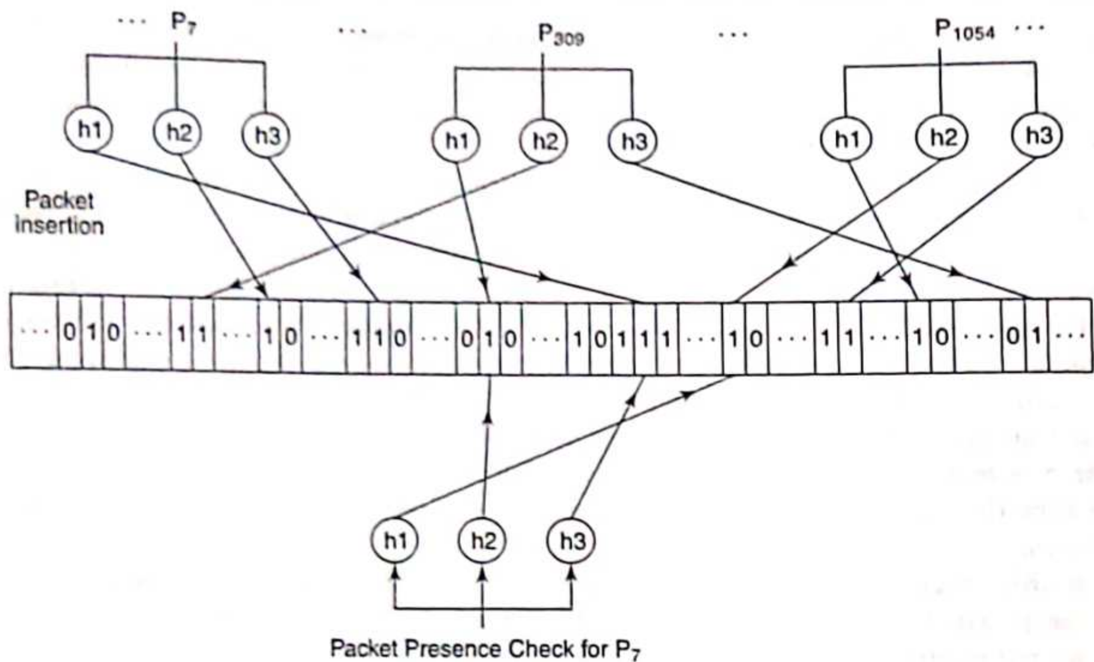



**Figure 22.6** *Illustrating false positive in a Bloom Filter*

Intuitively, for a given $n$, the chance of a false positive decreases with larger $m$. On the other hand, a larger value of $m$ incurs a greater storage overhead. We next derive an expression for the probability of a false positive as a function of $m$ and $k$.

Probability [a packet hashes to $i_1$] = $\frac{1}{m}$

Probability [a packet does not hash to $i_1$]

$$= \left(1 - \frac{1}{m}\right)$$

Probability [none of the $n$ packets hash to $i_1$ with any of the $k$ hash functions]

$$= \left(1 - \frac{1}{m}\right)^{kn}$$

Probability [at least one of the $n$ packets hashes to $i_1$ with at least one of the $k$ hash functions]

$$= 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Now the probability of a false positive is given as:

Probability [at least one of the $n$ packets hash to $i_1$ with at least one of the $k$ hash functions *and* at least one of the $n$ packets hash to $i_2$ with at least one of the $k$ hash functions

...

https://hemanthrajhemu.github.io

*and* at least one of the $n$ packets hash to $i_k$ with at least one of the $k$ hash functions]

$$= \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

It turns out that a reasonably acceptable false positive probability of 1% is achievable with $k = 3$ and $m = 12 \times n$. This translates to a storage cost of only 12 bits per packet at the expense of performing three hash computations per packet. This is considerably less than storing each IP datagram (about 500 bytes on average) or even storing just a 32-bit hash of a packet.

## 22.5 MALWARE DEFENCE

### 22.5.1 Worm Defence

In this section, we focus on the defence and detection strategies in the context of fast-spreading Internet worms. To the extent that many of these worms exploit the buffer overflow vulnerability, the precautions and techniques discussed in Chapter 18 will help prevent worm infections. Software vendors should create timely *patches* following the detection of a vulnerability in their software. Expeditious dissemination and application of patches are necessary to ensure a greatly reduced population of susceptible machines.

Preventive techniques alone are seldom adequate where security is concerned. Detecting the onset of a worm epidemic and then taking appropriate measures is an important second line of defence. We next enumerate challenges in the detection of worms and worm-based attacks.

*Speed.* In Section 22.3.2, we had mentioned speed and accuracy as two important attributes of an IDS. Speed is especially important in the detection of epidemics caused by Internet scanning worms. Efforts by humans to detect such attacks could take hours – this is clearly unacceptable in the case of worms such as Slammer which spread across much of the Internet in 15 minutes. Timely detection in conjunction with semi-automated deployment of patches could greatly help slow down the pace of the epidemic. For example, simulation studies have shown that if patching begins before 5% of the vulnerable machines are infected, then it is possible to confine the epidemic to less than 50% of the vulnerable machines.

*Non-monomorphic Worms.* Detecting *polymorphic* and *metamorphic* worms is not straightforward since multiple instances of the same worm may not contain common substrings. This is less true of polymorphic worms which might contain decryption routines that are invariant across multiple instances of a worm.

*Zero-day/Zero-hour Worms.* How does one detect a new worm that has never been seen before? A database of worm signatures will not contain the signature of a newly unleashed worm. So, anti-virus software may be incapable of protecting a system in the face of zero-day worms. What is needed is a quick and efficient way to detect zero-day worms and the ability to rapidly disseminate this information.

We next list certain characteristics that an IDS should look for in network traffic.

- Monomorphic worm instances share *common substrings*. So, it is necessary that our IDS should have built-in string matching algorithms that can process large volumes of Internet traffic.

https://hemanthrajhemu.github.io

- Second, a particular worm targets one vulnerable application. Hence, the ***destination port numbers*** in different instances of that worm are identical. In the case of worms with multiple vectors of propagation, a limited number of vulnerable applications are targeted. Even in that case, only a small number of destination port numbers will dominate worm traffic.
- A third characteristic is based on the scanning technique employed by worms such as Code Red and Slammer. These worms select their targets randomly. Hence, the ***IP addresses*** (both source and destination) in a sample of such worms would tend to exhibit much ***diversity***.

Before studying the techniques used for worm detection, we address the issue of where should the IDS be deployed. One possibility is a *host-based IDS* for worm detection. Such an IDS would report anomalous events such as a surge in connection attempts from an infected host to never-seen-before IP addresses. Another possibility is to place the IDS just behind the ingress router in the *DMZ* of an edge network (at the edge of the Internet). Finally, co-locating an IDS with routers in the *core* of the Internet is another option. An alternate strategy is to deploy artefacts such as network *telescopes* and *honeypots*.

Network telescopes are set up to monitor large portions of the address space of the Internet. Traffic directed at unused portions of this address is deemed suspicious. This traffic could be the result of port scans or of worms attempting to locate potential targets by randomly scanning the address space of the Internet.

Honeypots detect malicious traffic by appearing as an attractive target to attackers. To the outside world, they appear to run vulnerable applications. Worms that exploit these applications are attracted to the honeypot just as nectar attracts bees. Once malware enters the honeypot, carefully designed software in the honeypot inspects the malware. The honeypot attempts to study which applications are being targeted, where are the attack sources and what are the likely worm signatures.

### 22.5.2 Worm Signature Extraction

A key step in network-based worm detection is to identify *worm signatures* – patterns of substrings that occur in the payloads of all/most instances of a particular monomorphic worm.

For reasons of efficiency, arriving packets should undergo some form of pre-processing to identify *suspicious flows*. For example, packets from never-seen-before IP addresses may be deemed suspicious. Thereafter, only suspicious traffic is parsed in search of common substrings. The question is "How large are the common substrings?"

There is a clear tradeoff in choice of *length of the common substrings*. Choosing small substrings will increase the number of false positives. On the other hand, by attempting to obtain a match on large substrings, we may fail to detect worms that are essentially identical though superficially different. For example, the two worms shown in Fig. 22.7 are both created out of a common parent. They each have superfluous instructions such as NOPs strewn about them making it hard to capture large common substrings. Thus, trying to identify copies of a worm based on large common substrings may increase the chance of false negatives.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 35 | 7F | 29 | 00 | A1 | 2B | 65 | 00 | 98 | D3 | 4C | 18 | 2C | 00 | 90 |
| 35 | 7F | 29 | 00 | A1 | 00 | 2B | 65 | 00 | 98 | D3 | 4C | 00 | 18 | 2C |

**Figure 22.7** *Two instances of the same worm*

https://hemanthrajhemu.github.io

Commonly occurring strings that are parts of protocols such as HTTP or SMTP should be handled appropriately by including them in a *white list*. The underlined substrings in the HTTP request message shown below are examples of innocuous strings that should not arouse suspicion.

GET /favourites/greatpage.html HTTP/1.1
Host: www.iitb.ac.in
Connection: close
Accept-language: es

Many schemes such as those in [SING04], [KIM04], and [CAI07] make use of modified *Rabin fingerprints* to identify common substrings in different packets. A Rabin fingerprint of a block is basically an easily computable hash of that block To see how this works, consider the payload of an incoming IP packet. Suppose its length is 1000 bytes and that we choose to search for common substrings of length 15 bytes. We then compute the Rabin fingerprint for each 15-byte block (contiguous sequence of bytes) as follows.

Let

$$b_{i+14}\, b_{i+13}\, b_{i+12} \dots b_{i+1}\, b_i$$

be the bytes of a 15-byte block. The Rabin fingerprint of this block, $RF(i)$ is

$$(b_{i+14} \times p^{14} + b_{i+13} \times p^{13} + b_{i+12} \times p^{12} \dots b_{i+1} \times p + b_i) \bmod m$$

where $p$ and $m$ are suitably chosen integer constants.
The overlapping block with rightmost byte $b_{i-1}$ is

$$b_{i+13}\, b_{i+12}\, b_{i+11} \dots b_i\, b_{i-1}$$

The Rabin fingerprint of the above block, $RF(i - 1)$, is

$$(b_{i+13} \times p^{14} + b_{i+12} \times p^{13} + b_{i+11} \times p^{12} \dots + b_i \times p + b_{i-1}) \bmod m$$

It is easy to verify that

$$(p \times RF(i) - RF(i\text{-}1)) \bmod m = (b_{i+14} \times p^{15} - b_{i-1}) \bmod m$$

or

$$RF\ (i\text{-}1) = (p \times RF(i) - b_{i+14} \times p^{15} + b_{i-1}) \bmod m$$

The above expression simplifies the computation of $RF(i\text{-}1)$ if $RF(i)$ is already known. In addition, $p^{15}$ can be pre-computed thereby further reducing computation time.

We can compute the Rabin fingerprints for each of the 986 15-byte blocks in the packet starting with the leftmost block in Fig. 22.8. We thus compute the sequence

$$RF(985),\ RF(984),\ \dots\ RF(0)$$

using the above optimization.

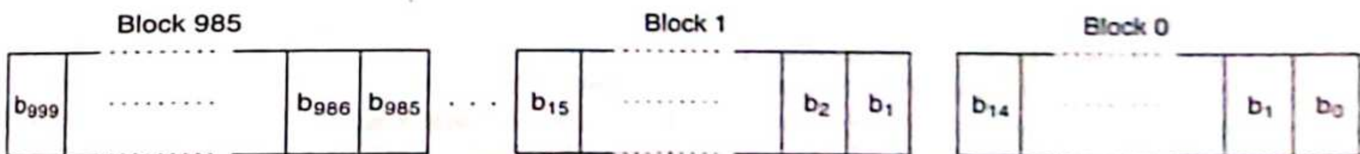



**Figure 22.8** *Blocks upon which Rabin fingerprint is computed*

For ease of understanding, we next present a simplified version of the worm detection schemes. To detect the outbreak of a worm epidemic, we keep track of the *prevalence* of different substrings and

https://hemanthrajhemu.github.io

also the *address dispersion* of incoming packets. For this purpose, we create a table, $\mathcal{T}$, with four columns. The columns are

- the destination port number (Port #),
- the Rabin Fingerprint value (RF),
- the prevalence count (PC) and
- a set of IP source–destination pairs (IP).

The exact roles of each column are explained next.

For each incoming packet, the Rabin fingerprint of each block of length, $w$ is computed. Each block is checked against the white list introduced earlier. If a block appears in the white list, it is ignored. For each distinct block encountered within a packet, one of two updates is made in $\mathcal{T}$. If either the port number of the packet or a fingerprint is encountered for the first time, a new row is created in $\mathcal{T}$. The port# and *RF* columns are initialized appropriately. The *PC* column is set to 1 while the *IP* column is set to NULL. If both, the port number and fingerprint, were encountered for a previous packet, no new row is created. Instead the value in the *PC* column of that row is incremented. Note that the *PC* column keeps track of the number of packets having a particular value of destination port and containing at least one block with a particular Rabin fingerprint value.

If the prevalence count corresponding to a given port and fingerprint exceeds a threshold, *Threshold_PC*, the IDS *suspects* that a worm may have been unleashed. Only then, do we begin the process of tracking the *distinct source–destination pairs* of packets carrying such strings. Thereafter, for each new packet containing the given destination port and fingerprint, we insert its source–destination address pair to the *IP* column of that row, if not already there.

```
Let p be the destination port # of an incoming packet, X.
Let s be the source IP address in X and
Let d be the destination IP address in X.

for each block of width w in the payload of X
        compute the Rabin Fingerprint
    Let F be the set of Rabin Fingerprints of the blocks in X

    for each fingerprint, f ∈ F {
        if (f ∈ WhiteList)
            quit
        if T has a row with (port# = p AND RF = f)
            increment PC for that row
        else create a new row with
            port# = p, RF = f and PC = 1

        if (PC > Threshold_PC) {
            if (s, d) ∉ IP column of this row
                    save (s, d) in the IP column of this row
            if (# of IP address pairs in the IP column > Threshold_IP)
                    sound "Worm Alert"
        }
    }
}
```

https://hemanthrajhemu.github.io

If the address dispersion column in a row with Port# = $p$ and RF = $f$ exceeds a threshold, *Threshold_IP*, we conclude that there is strong likelihood that

- an Internet scanning worm is on the prowl
- that it is targeting a vulnerable application on destination port, $p$ and
- that its payload contains a substring with Rabin Fingerprint = $f$.

A negative feature of this algorithm is that the table, $\mathcal{T}$, will soon get very large. One possibility to reduce the number of table entries, is to save fingerprint values that satisfy a certain predicate such as

Store only those fingerprints ending in '11111'

This effectively cuts down the size of $\mathcal{T}$ to 1/32th of its original size.

How long is such a table maintained? Assuming that a worm epidemic spreads in minutes, a single table should capture packets arriving in the last 10 minutes, for example. After 10 minutes, highly suspicious fingerprints in the current table are saved in another table for further analysis. The current table is then purged and the algorithm re-started for the next 10-minute interval.

The efficacy of the above IDS can be greatly magnified by having multiple IDS' at geographically dispersed sites. The IDS' exchange suspicious strings that they have encountered. By so doing, they achieve faster convergence on detecting signatures of newly unleashed Internet scanning worms.

### 22.5.3 Virus Detection

The previous subsection dealt with detection of fast-spreading Internet scanning worms. These worms have generally been of the non-polymorphic, non-metamorphic variety. In this subsection, we focus on host-based detection of polymorphic/metamorphic viruses. Given the importance of virus scanners, there are numerous anti-virus products out in the market including brands such as Kaspersky, Avast, Symantec, Avira, etc.

*Virus scanners* attempt to detect and identify malicious executables in files, e-mail messages and network packets. Over time they create and update a database of virus signatures. The earliest signatures were strings of bytes or instructions that would faithfully appear in every instance of a particular virus. But these attempts at virus detection were soon defeated by virus writers who created tens of *mutants* of common viruses. Bagle, Netsky, and MyDoom are examples of viruses whose variants successfully evaded detection.

Newer strains of polymorphic and metamorphic viruses were soon out. This forced researchers to invent virus signatures based on *malware behavioural characteristics* that are invariant across different strains of a given virus despite the many *obfuscation techniques* employed. But what does malware behaviour really mean? Could the behaviour of a virus family be captured by a set (or subset) of operating system calls made?

The *system call interface* is the primary means across which a user program interacts with the operating system. Modern operating systems support hundreds of system calls for file creation/access, process creation and synchronization, etc. Are system calls made by an infected program different from those made by a normal program and, if so, how? It is not easy to assert that certain calls are only made by malicious executables. Nevertheless, the latter do often exhibit suspicious behaviour – calls to copy their code to another file, creating and deleting entries in the Windows registry, etc. should certainly arouse suspicion.

It is tempting to conclude that a piece of code is infected if it contains certain suspicious sequences of system calls. However, virus writers could easily intersperse innocuous system calls within such black-listed sequences. So, instead of just sequences, it is more appropriate to look for dependencies between calls. This is illustrated for the Netsky virus.

https://hemanthrajhemu.github.io

Figure 22.9 shows the flow of data between a subset of dependent calls made by the Netsky virus [KOLB09]. Mutants of this virus create a file with an innocent sounding name and copy themselves to this file. This is accomplished by the following sequence of system calls.

- The memory-resident Netsky virus attempts to find the name of the file containing its executable. For this purpose, it makes the GetModuleFileName system call. As shown in Fig. 22.9, this call returns the file name, *netsky.exe*, for example.
- The virus invokes the CreateFile system call to open the file *netsky.exe*. The call returns a handle, *F1*, to this file.
- The virus creates a file with a deceptive name like *AntiVirus9x.exe*. Here again its uses the CreateFile system call which returns handle, *F2*, to this file.
- The virus reads its own code into a block of main memory using the CreateSection call. The call uses the file handle, *F1*, as input which was returned by the first CreateFile call. This dependency is shown by an arc in Fig. 22.9. The CreateSection call returns a handle to a memory block, *S1* which is input to the call, MapViewOfSection (see Fig. 22.9).
- Finally, the virus copies the memory buffer containing its code to the newly-created file, *AntiVirus9x.exe*. The WriteFile system call used here takes as input the file handle, *F2*, returned by the second CreateFile call in Fig. 22.9.

| System Call | Inputs (Name, Value pairs) | Output (Name, Value pairs) |
|---|---|---|
| GetModuleFileName ( ) | – | FileName = netsky.exe |
| CreateFile ( ) | FileName = netsky.exe<br>mode = open | FileHandle = F1 |
| CreateFile ( ) | FileName = AntiVirus9x.exe | FileHandle = F2 |
| CreateSection ( ) | FileHandle = F1 | SectionHandle = S1 |
| MapViewOfSection ( ) | SectionHandle = S1 | |
| WriteFile ( ) | FileName = F2<br>buffer = . . . | |

**Figure 22.9** *System call dependencies in netsky.exe virus*

Virus behavioural models offer much hope in detecting seemingly different strains of a virus. In [KOLB09], for example, known viruses are executed in a controlled environment and signatures in the form of behaviour graphs are generated. To detect whether an unknown program is infected, its run-time behaviour is checked against a database of accumulated signatures.

## SELECTED REFERENCES

An excellent reference on IDS is the NIST publication, [NIST07]. Protection from DDoS attacks through Distributed Route Filtering in the core of the network was proposed in [PARK01]. SYN flood detection using sequential change point detection techniques were studied in [WANG04].

https://hemanthrajhemu.github.io

Probabilistic marking schemes for IP traceback appear in [SAVA00], [SONG01], etc. Hash-based IP traceback was first proposed in [SNOE01].

Techniques to detect high-speed Internet scanning worms appear in [SING04] (Earlybird), [KIM04] (Autograph), and [CAI07] (Wormshield). Virus detection using behavioural analysis appears in [CHRI05] and [KOLB09].

## OBJECTIVE-TYPE QUESTIONS

22.1 Which of the following is/are true?
- (a) An anomaly-based IDS uses OS-based audit trails to detect intrusions
- (b) A signature-based IDS identifies patterns of behaviour that accompany an attack
- (c) A network-based IDS identifies whether the behaviour of the network is a statistically significant departure from normal
- (d) A host-based IDS alerts the administrator if it sees a disproportionate number of malformed TCP packets entering the organization

22.2 The possible goal of an attacker in sending packets with invalid combinations of TCP header flags is to
- (a) launch an SYN flood attack
- (b) find which services are open
- (c) perform OS fingerprinting
- (d) determine the addressing schema within an organization

22.3 A preventive measure against DDoS is
- (a) packet logging
- (b) distributed route filtering
- (c) ingress filtering
- (d) egress filtering

22.4 The probability of false positives in the Sequential Change Point Detection Algorithm can be reduced by
- (a) increasing the length of each observation interval
- (b) decreasing the length of each observation interval
- (c) increasing the threshold
- (d) decreasing the threshold

22.5 Probabilistic Packet Marking is a technique used in support of
- (a) DDoS prevention
- (b) IP traceback
- (c) DDoS detection
- (d) Worm detection

22.6 The most appropriate variable that must be monitored to detect the outbreak of an epidemic involving an Internet scanning worm is
- (a) the distribution of source IP addresses of incoming packets
- (b) the distribution of destination IP addresses of incoming packets
- (c) the distribution of source ports of incoming packets
- (d) the distribution of destination ports of incoming packets

22.7 The main goal of computing Rabin fingerprints on incoming packets is to
- (a) determine the presence of a particular worm signature
- (b) determine the absence of a particular worm signature
- (c) determine the distinct substrings in incoming packets
- (d) determine the common substrings in incoming packets

https://hemanthrajhemu.github.io

## EXERCISES

**22.1** Whatever is accomplished by a firewall can also be accomplished by an IDS. Yes or No? Explain.

**22.2** Answer the following for a host-based IDS. Justify your answer to each part.
(a) What variables should it monitor?
(b) Should it be signature-based, anomaly-based, or both?
(c) On which host or hosts within an organization should such a host-based IDS be installed?

**22.3** Repeat Exercise 22.2 for a network-based IDS. Where in the organization should such an IDS be placed?

**22.4** Which kind of intrusions and attacks (including different worm/virus attacks and DDoS attacks) would a network-based IDS be especially suited for? And which intrusions/attacks would a host-based IDS be suited for? Explain your answers.

**22.5** How does an IDS distinguish between the occurrence of a DoS/DDoS attack and a flash event? (In the latter, for example, a website could see a surge in HTTP traffic in response to an advertisement displaying the site's URL on TV during a cricket match.)

**22.6** In the context of SYN flooding, is it possible for an attacker to defeat the Sequential Change Point Detection technique? If so, how?
Can you think of a better variable to monitor than the one discussed in this chapter? If so, what is it and why is it better?

**22.7** Consider the probabilistic packet marking approach to the IP traceback problem studied in this chapter.
(a) How many attack packets at the victim end would the IDS need to inspect to trace back the location of the attacker?
(b) The problem with this approach is that it is much less likely that an upstream router's fingerprint survives in attack packets. Can you think of ways to improve this algorithm? Explain your approach(es).
(c) Is it possible to design a scheme wherein attack packets have a more even distribution of fingerprints of intermediate routers between the source of the attack and the victim? If so, elaborate on your design.

**22.8** What is the role of a Bloom Filter in packet logging?
Consider a router with 7 links, each having a speed of 1 Gbps. Assume that the router needs to "store" all packets received in the last 5 min into a Bloom Filter. Assume that two hash functions are used to determine the positions a packet must be inserted into. Calculate the size of the Bloom Filter if a maximum false positive rate of 2% may be tolerated.
How would your answer change if three hash functions were employed instead?

**22.9** Compare the packet marking versus packet logging schemes for IP traceback in respect to probability of success, cost, ease of deployment, performance overheads, and any other relevant measure.

**22.10** Can the worm detection technique (with slight modifications) described in this chapter (Section 22.5.2) be used for detecting
(a) peer-to-peer worms
(b) web worms
(c) e-mail worms?
Explain why or why not.

https://hemanthrajhemu.github.io

22.11 In the section on malware detection, we first studied a way to detect a zero-day fast-spreading Internet scanning worm. We then looked at ways to detect polymorphic/metamorphic malware. How would you handle the double-challenge of detecting zero-day malware that is, in addition, obfuscated using polymorphism/metamorphism?

## ANSWERS TO OBJECTIVE-TYPE QUESTIONS

22.1 (a)(b)(c) 22.2 (c) 22.3 (b)(d) 22.4 (c)
22.5 (b) 22.6 (a)(d) 22.7 (d)

https://hemanthrajhemu.github.io

# Chapter 25

# Web Services Security

## 25.1 MOTIVATION

### 25.1.1 Introduction

The emergence of the web in 1993 has helped change the way millions of users shop, bank, and pay their bills. Round-the-clock (24×7) availability of the web, the interactive nature of web applications, and the personalized nature of the experience on some websites have all played a role in providing unprecedented convenience to the customer.

The earliest web applications used Java servlets/JSP or ASP (on the Microsoft platform). Scalability and reusability together with support for transaction processing, security, etc. was provided by the next generation of component-based web technologies such as J2EE and .NET. Securing the communication link was made possible through the use of the SSL protocol (studied in Chapter 14).

Many of the earlier web applications (such as Internet banking) involved human-to-program interaction. However, applications such as supply chain management differ from traditional web applications in several significant respects:

- Programs communicate with each other over the web with little or no human intervention.
- Services might have a composite nature. Such "composite services" necessitate the involvement of multiple providers, each providing an "atomic service."
- There are potentially a large number of "atomic service" providers offering a given service. So, clients have a choice and can dynamically change providers.
- Clients and providers may be running applications using different web technologies on diverse computing platforms with different operating systems. Inter-operability is thus an important issue.

*Web-based travel planning* is an application that possesses many of the above features. A user might visit the website of his/her travel agent to book an airline ticket. It would be convenient if the user could also reserve a hotel room and rent out a car in the same login session. In this case, there are four atomic service providers – the travel agent, the airline, the hotel chain, and the car rental company.

For a given travel destination, the travel agent's site also offers the user a host of hotel sites and car rental options. There are thus multiple atomic service providers a user may choose from that best suit his/her needs and budget. As part of the travel service application, the user should be able to seamlessly visit/exit the four websites and perform transactions with no other human interaction. The travel site would have business partnerships with the airlines, hotels, and car rentals listed on

https://hemanthrajhemu.github.io

its web page for the computing platforms and the software that power their applications might be very different from one another.

The common term used to identify web applications that share some or all of the above features is a web service. The World Wide Web Consortium, W3C, defines a web service as

> "A software system identified by a URI whose public *interfaces* and *bindings* are defined and described using XML. Its definition can be *discovered* by other software systems. These systems may then interact with the web service in a manner prescribed by its definition using XML-*based messages* conveyed by *Internet protocols*."

### 25.1.2 Entities Involved

An atomic web service involves three entities: the requester (or client), the provider (or server), and a registry. As shown in Fig. 25.1:

- Providers register or *publish* their services in a public registry.
- Requesters *discover* services by querying the registry for services that match certain criteria.
- Once a requester has identified a provider whose services it needs, it *binds* to and *invokes* the services of that provider.

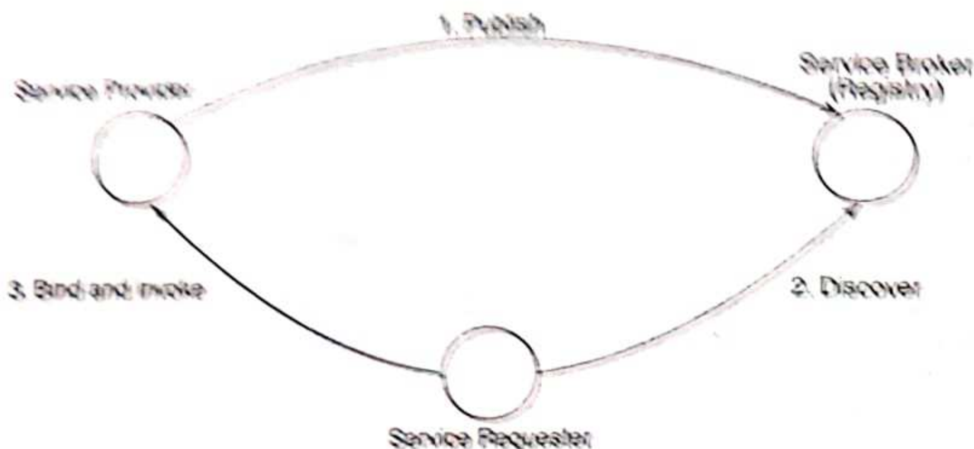



**Figure 25.1** *Entities involved in a web service*

The technologies to support web services are all based on XML (Extended Markup Language) which has become the *lingua franca* for electronic documents. XML and the technologies that support web services are introduced in the next section. We then look at the need for security in web services and explain why SSL is either unsuitable or inadequate in providing it. Various security standards in support of web services, such as WS-Sec, the XML Encryption Standard, the XML Signature Standard, Security Assertions Markup Language (SAML), etc., are then introduced in subsequent sections.

## 25.2 TECHNOLOGIES FOR WEB SERVICES

### 25.2.1 XML

A *markup language* uses *tags* as a mechanism to identify *structures* in a document or to specify presentation style/format. For example, a chapter in a text book is made up of one or more sections.

https://hemanthrajhemu.github.io

Each section in turn is made up of zero or more subsections and each subsection is made up of one or more paragraphs. These facts may be represented using a markup language as follows

```
<chapter>
    <section>
        <subsection>
            <paragraph>
            </paragraph>
        </subsection>
    </section>
</chapter>
```

One of the most common markup languages in widespread use is HTML. Tags in HTML are used to tell the browser how to display the content of a web page. *XML tags*, on the other hand, are used to describe data – in particular, the *structure of the data*. Unlike HTML, XML does not have a pre-defined tag set. Instead XML is a *meta language* – it provides a facility to define tag sets in diverse fields such as business, medicine, mathematics, and law.

The most basic kind of markup found in an XML document is an *element*. The start of an element within a document is indicated by a *start-tag* which contains the name of the element within angular brackets. Figure 25.2(a) shows a Purchase Order in XML. The tag on Line 3 of the document indicates the start of element *shipTo*. The end-tag, </ *shipTo*> on Line 9 in Fig. 25.2(a) indicates the end of the element, *shipTo*. An *end-tag* can be recognized by the "/" to the immediate right of the opening angular bracket.

An element may contain only data or it may contain other *sub-elements* or it may contain both, data, and other sub-elements. In Fig. 25.2(a), the element, *shipTo* contains sub-elements *name*, *street*, *city*, *state* and *PIN*. The sub-elements, in this case, contain only text. For example, the name element (Line 4) contains the customer's name.

An element may contain zero or more *attributes*. An attribute is a name value pair which appears after the element name in the element's start tag. For example *shipTo* (Line 3) contains a single attribute whose name is *country* and whose value is INDIA.

The Purchase Order in Fig. 25.2(a) is highly structured – the name and address of the person receiving the shipment occurs first. This is followed by the items ordered. Each item includes the *productName*, *quantity*, and *UnitPrice* in sequence. The correct sequencing and nesting of elements is necessary since computers are expected to process such documents. But how does a computer know what to expect in a document such as a Purchase Order?

A *Document Type Definition* or the more recently standardized *XML schema* contains rules to interpret the document's content. The rules include information such as

- What is the element type, e.g., string, decimal, complex type, etc.?
- Is an element optional? If not, how many times should it occur (once, one or more times, etc.)?
- Does an element have any attributes? If so, what are their names and types?
- What is the content of an element (other sub-elements or text)?
- What is the sequence of elements and how are they nested?

Figure 25.2(b) shows the XML schema representation of a purchase order. The purchase order document of Fig. 25.2(a) is an instance of this schema. We consider here a toy example: a real-world schema of a purchase order may include hundreds of elements and attributes.

https://hemanthrajhemu.github.io

```
1   <?xml version="1.0"?>

2   <purchaseOrder orderDate="2009-01-05">

3      <shipTo  country="INDIA" >
4         <name>  Kiran Kumar  </name>
5         <street> 63 M.G. Road </street>
6         <city>  New Chicago  </city>
7         <state> Gujarat </state>
8        <PIN> 123456 </PIN>
9      </shipTo>

10     <items>

11        <item partNum="129BZ" >
12           <productName> Electric Toaster </productName>
13           <quantity> 1 </quantity >
14           <UnitPrice> 1412.00 </UnitPrice >
15        </item>

16        <item partNum = "798RD" >
17           <productName>  Dinner Set </productName>
18           <quantity> 1 </quantity>
19           <UnitPrice> 2142 </UnitPrice>
20        </item>

21     </items>

22  </purchaseOrder>
```

Purchase order conforming to schema definition in (b)

**Figure 25.2(a)** *Purchase order in XML*

### 25.2.2 SOAP

*Simple Object Access Protocol* (SOAP), standardized by $W^3C$, is a framework for exchanging structured information over the Internet. The SOAP protocol defines a one-way message transfer between two entities. (A two-way message transfer as employed in typical client–server applications is easily synthesized using multiple one-way messages.)

A SOAP message is an XML document made up of an *envelope*, which includes an optional *header* and a mandatory *body*. Most of the information in the message is contained in its body. The header is used to extend the message and may include security meta-information such as the encryption algorithm used, a digital signature computed on the message, etc. A SOAP message fits snugly inside the body of an HTTP request or response packet. The MIME type field in the HTTP header of a SOAP message is set equal to text/xml.

In lieu of the document-style message format, the body of a SOAP message may contain remote procedure calls (RPCs) in XML format. The example below shows a SOAP request message from a client to a provider of current stock prices. Note that it is encapsulated in an *HTTP request* packet. Also shown is the SOAP response message which is encapsulated in an *HTTP response* packet.

https://hemanthrajhemu.github.io

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://xyz.org/po.xsd"
xmlns="http://tempuri.org/po.xsd" elementFormDefault="qualified">

 <xs:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xs:element name="comment" type="xs:string"/>

 <xs:complexType name="PurchaseOrderType">
  <xs:sequence>
   <xs:element name="shipTo" type="Address"/>
   <xs:element ref="comment" minOccurs="0"/>
   <xs:element name="items"  type="Items"/>
  </xs:sequence>
  <xs:attribute name="orderDate" type="xs:date"/>
 </xs:complexType>

 <xs:complexType name="Address">
  <xs:sequence>
   <xs:element name="name"   type="xs:string"/>
   <xs:element name="street" type="xs:string"/>
   <xs:element name="city"   type="xs:string"/>
   <xs:element name="state"  type="xs:string"/>
   <xs:element name="PIN"    type="xs:decimal"/>
  </xs:sequence>
  <xs:attribute name="country" type="xs:NMTOKEN"/>
 </xs:complexType>

 <xs:complexType name="Items">
  <xs:sequence>
   <xs:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="productName" type="xs:string"/>
      <xs:element name="quantity">
       <xs:simpleType>
        <xs:restriction base="xs:positiveInteger">
         <xs:maxExclusive value="200"/>
        </xs:restriction>
       </xs:simpleType>
      </xs:element>
      <xs:element name="UnitPrice"    type="xs:decimal"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>

</xs:schema>
```

XML schema definition of a purchase order

**Figure 25.2(b)** *Purchase order in XML*

https://hemanthrajhemu.github.io

The mapping between a SOAP message and an underlying transport protocol is referred to as a *SOAP binding*. SOAP may be run on top of either HTTP or SMTP but it is more commonly used over HTTP. In the case of an HTTP binding, either a GET request or a POST request may be used. In the latter case, a SOAP message may be encapsulated in the body of the HTTP POST packet as well as in the HTTP response packet as shown in Fig. 25.3.

```
POST /InStock HTTP/1.1
Content-Type: application/soap+xml; charset=utf-8
. . .
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap = "http://www.w3.org/2001/12/soap-envelope . . .>
  <soap:Body xmlns:X="http://www.stockQuote.com/price">
    <X:GetPrice xmlns:X = "http://www. . . " >
      <X:StockName> MyStartUp </X:StockName>
    </X:GetPrice>
  </soap:Body>
</soap:Envelope>
```

(a) SOAP message in HTTP POST request

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
. . .
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap = "http://www.w3.org/2001/12/soap-envelope . . . >
  <soap:Body xmlns:X="http://www.stockQuote.com/price">
    <X:GetPriceResponse xmlns:X = "http://www. . . " >
      <X:Price> 3847 </X:Price>
    </X:GetPriceResponse>
  </soap:Body>
</soap:Envelope>
```

(b) SOAP message in HTTP response

**Figure 25.3** *SOAP messages in HTTP packets*

A SOAP message between two end-points may be routed through several intermediaries. A node in the SOAP message path may require that a particular header element, say <block1>, be processed by the ultimate destination node or its immediate successor node.

This information is conveyed by two attributes – *role* and *mustUnderstand* that are included in <block1>. Processing a header might involve modifying values within the given header, removing the header or inserting a new header.

https://hemanthrajhemu.github.io

### 25.2.3 WSDL and UDDI

*Web Services Definition Language* (WSDL) is a language for describing web services. It exposes the operations and communication protocols used by the web service. A complete *WSDL service description* will include definitions of various elements – *types*, *messages*, *operations*, *portTypes*, and *bindings*.

At the heart of a web service description is a *portType*, which specifies one or more operations within its scope.

An *operation* is an abstract definition of an action. It involves one or more messages. For example, an operation can receive a message that needs no response or it can send a "notification" message – one that expects no response. More commonly, an operation receives a request and sends a response.

A *message* is an abstract definition of data being exchanged as part of an operation. Messages may have multiple parts; each part has an associated type.

Figure 25.4 shows a portType, which comprises a single operation involving two messages: a request and a response.

```
<message name="message1">
   <part name=" . . . " type=" . . . "/>
</message>

<message name="message2">
   <part name=" . . . " type=" . . . "/>
</message>

<portType name="portType1">
  <operation name=" . . . ">
      <input message="message1"/>
      <output message="message2"/>
  </operation>
</portType>
```

**Figure 25.4** *Port type that includes one operation comprising two messages*

In addition to specifying the messages within an operation and the operations within a portType, WSDL permits the web service developer to indicate the specific communication protocols to be used in support of each operation. This is referred to as a *binding*. Permissible bindings include SOAP, HTTP POST, and HTTP GET.

*Universal description, discovery, and integration* (UDDI) is a *registry* or catalogue that allows businesses across the globe to list themselves on the Internet. Prospective clients discover web services by querying the registry for specific services using SOAP messages. In response, they are provided access to the WSDL that describes the operations, messages, and protocols for the desired web service. The registry includes the equivalent of *white*, *yellow*, and *green* pages of a telephone directory. White pages provide address and contact information of a service. Yellow pages provide an industrial categorization of the services and the green pages provide information about services that the business exposes.

https://hemanthrajhemu.github.io

### 25.2.4 Why not SSL?

SSL, studied in Chapter 14, is a widely used protocol for securing web communications. However, there are a number of reasons why SSL is not suitable for use in web services.

Messages created or received by web services may involve several intermediaries. Since SSL is a point-to-point protocol, messages will have to be decrypted and re-encrypted by each intermediary. What web services require is not point-to-point but *end-to-end security*.

Another limitation of SSL is that it is a transport-level protocol that performs encryption and integrity protection of transport-level entities. It does not understand XML – the lingua franca of web services. Encryption and signing of *specific elements* of a message are required by web services. SSL, on the other hand, encrypts and integrity-protects messages in their entirety. What web services require is *fine-grained message protection*.

To address the specific security requirements of web services, a number of standards have been developed. The most important of these is *WS-Sec*, which defines the structure of security tokens and how these are applied to the SOAP header. Under the WS-Sec umbrella are standards for XML encryption and XML signatures. SAML (security assertion markup language) provides a way to exchange authentication, attribute, and authorization information. SAML is one way of providing a *single sign-on* facility whereby a user logs in and authenticates himself/herself just once during a transaction and his/her credentials are transferred to other partner sites participating in the transaction.

*WS-Trust* provides a framework for creating various kinds of trust relationships between communicating entities. *WS-SecureConversation* creates a secure session, which is analogous to the security association in IPSec or the SSL connection. *WS-Federation* helps create trust relationships across multiple trust domains. Most of these standards have appeared since 2002 and some have undergone several revisions to date. The principal players in developing the standards are $W^3C$, IETF (Internet Engineering Task Force), and OASIS (Organization for the Advancement of Structured Information Systems).

We discuss some of these standards in this chapter beginning with WS-Sec.

## 25.3 WS-SECURITY

### 25.3.1 Token Types

WS-Security (also called WS-Sec) addresses some of the most basic problems in securing messages used in web services. Its main functions are

- It defines XML elements that are used to communicate *security tokens* (defined below) in the header of a SOAP message.
- Together with the XML Encryption Standard it defines the syntax and processing rules used to *encrypt* one or more parts of a SOAP message.
- Together with the XML Signature Standard, it defines the syntax and processing rules used to create and represent a *digital signature* on one or more parts of a SOAP message.

The first version of the WS-Security standard was created by the Organization for the Advancement of Structured Information Standards (OASIS) in 2004 with input from IBM, Microsoft and Verisign. Version 1.1 (the latest version) was released in February 2006.

Security-related information is contained within a <Security> element in a SOAP header. It specifies what operations are performed (such as signing, encryption, etc.) and in what order. The <Security> element includes security tokens, keys, signatures, timestamps, and security meta-information.

https://hemanthrajhemu.github.io

A security *claim* is a statement made about a subject's identity, signing key, a subject's privileges, etc. A claim may be made by the subject himself or by another party on behalf of the subject. One or more claims is/are represented by a security *token*. Common examples of security tokens are a username + password, an X.509 certificate or a Kerberos ticket.

The username + password is one of the most commonly used security tokens. The default is to send the password in the clear but this is not a very secure option. Alternatively, the password (*pw*), a nonce (*n*), and the timestamp (*t*) may be concatenated and then hashed using a cryptographic hash function such as SHA-1

$$\text{SHA-1}\ (n, t, pw)$$

The user name, hash, nonce, and timestamp are sent in a <UsernameToken>

```
< UsernameToken >
        < Username > John < /Username >
        < Password Type = "PasswordDigest" >
                4u%h&+q:L
        < /Password >
        < Nonce > . . . < /Nonce >
        < Created > . . . </Created >
< /UsernameToken>
```

Security tokens containing usernames are pure text. Some security tokens may contain binary data such as signatures or keys. The *BinarySecurityToken* element is used in that case. Examples of binary security tokens include X.509 certificates and Kerberos tickets. The binary content in such tokens is rendered readable by encoding it using Base64 encoding or using hexadecimal notation.

The following is the representation of an X.509 certificate within the <Security> element of a SOAP header.

```
< Security >
. . .
        < BinarySecurityToken
                ValueType = " . . . X509v3"
                EncodingType = " . . . Base64Binary" >
                        Lp9tba4Pc7G . . .
            < / BinarySecurityToken >
. . .
< /Security >
```

### 25.3.2 XML Encryption

The XML Encryption Standard was developed by W³C in 2002. It defines XML elements for representing *encrypted data* and *keys* used for encryption. One of its key attributes is that it allows encryption at different levels of *granularity*:

- an entire document or
- a complete XML element within the document or
- the content of an XML element

The standard permits any combination of elements within the body and/or the header of a SOAP message to be encrypted.

https://hemanthrajhemu.github.io

The <EncryptedData> element is used to represent encrypted data in SOAP messages. Figure 25.5 shows encrypted elements in the body of a SOAP message. The actual ciphertext of each encrypted element is enclosed in a <CipherValue> sub-element (lines 32 and 43). Included in <EncryptedData> is the encryption algorithm used (256-bit AES in CBC mode on lines 29 and 40). Information on the key used for encryption may be placed within <EncryptedData>. Alternatively, the key may be placed in the header as described below.

The encryption key may be a pre-shared secret between the communicating parties. Alternatively, it may be a short-term or session key chosen by the sender. In the latter case, it should be transmitted in encrypted form. For this purpose, it is enclosed in an <EncryptedKey> element and placed in the SOAP header (lines 6–21 in Fig. 25.5). It may also be the case that many segments of the SOAP message are encrypted using the short-term key. In that case, a <ReferenceList> containing a manifest of encrypted segments (lines 18 and 19) is included in <EncryptedKey >.

Line 8 indicates that the algorithm used to encrypt the key is RSA. The <KeyInfo> element (lines 9–11) identifies the key used to encrypt the short-term key. In particular, line 10 informs us that the encrypted short-term key can be decrypted by the private key corresponding to the certificate belonging to Rajiv Singhvi. Finally, the ciphertext of the encrypted short-term key appears on line 14.

### 25.3.3 XML Signatures

The XML Signature Standard was developed jointly by W[3]C and IETF in 2002. It specifies the *syntax* for signatures and signature keys while offering a rich set of options for signing XML documents. For example, parts of a document can be signed by an entity. One or more intermediaries may attach their signatures to the document. Two entities may sign overlapping or disjoint parts of the document. Like standard RSA signatures, XML signatures involve computing the hash of a document followed by encryption using the signer's private key. There is, however, a caveat associated with signing XML documents which is explained below.

XML allows a lot of leeway in syntax. For example, extraneous white spaces are liberally permitted. Thus, two documents may be syntactically identical despite superficial differences in appearance resulting in different binaries. (Their binaries could be simply their UNICODE or Base64 representations.) Consequently, the cryptographic hash, applied separately to the two documents, will, in general, be distinct. Thus, syntactically identical documents signed by the same individual may have different digital signatures.

The first step in generating an XML signature is canonicalization. The parts of a document that need to be signed are first transformed into a *canonical form* before computing their hashes. Canonicalization guarantees that syntactically identical documents produce the same *serialized representations*.

Signatures are included in the header of the SOAP message containing the document. More specifically, they are contained in a <Signature> element in the header. The major elements and sub-elements contained in <Signature> shown in Fig. 25.6 are

<SignedInfo> Within this element is included information about the *canonicalization algorithm* and *signature algorithm* employed. An example of the signature algorithm is RSA-SHA1, i.e., the use of SHA-1 to perform the hash followed by an RSA private key operation on the hash value. A "signature" in the form of a Message Authentication Code (MAC) is also supported.

https://hemanthrajhemu.github.io

```
1        <?xml version="1.0" encoding="utf-8"?>
2        <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
3                                         xmlns:xenc="..." xmlns:ds="...">
4         <S11:Header>
5            <wsse:Security>
6              <xenc:EncryptedKey>
7                    <xenc:EncryptionMethod Algorithm =
8                            "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
9                    <ds:KeyInfo>
10                     <ds:KeyName> CN= Rajiv Singhvi, C=IN </ds:KeyName>
11                   </ds:KeyInfo>
12                   <xenc:CipherData>
13                       <xenc:CipherValue>
14                           aKLj89qwhMZsaRiDutagbx78bigxb...
15                        </xenc:CipherValue>
16                   </xenc:CipherData>
17                   <xenc:ReferenceList>
18                       <xenc:DataReference URI="#Id2"/>
19                       <xenc:DataReference URI="#Id3"/>
20                   </xenc:ReferenceList>
21             </xenc:EncryptedKey>
22           < / wsse:Security>
23        </ S11:Header>
24        < S11:Body wsu:Id = "#Id1" >
25            <xenc:EncryptedData
26            Type = "http://www.w3.org/2001/04/xmlenc#Element"
27                   Id="Id2">
28              <xenc:EncryptionMethod
29                     Algorithm = "http://www.w3.org/xmlenc#aes256-cbc"/>
30              <xenc:CipherData>
31                  <xenc:CipherValue>
32                           tdaqUsjXipJ09jlkjh5oinlkdsn...
33                  </xenc:CipherValue>
34              </xenc:CipherData>
35           </xenc:EncryptedData>

                          This is text sent in the clear. The previous segment
                          of text and the next segment of text are both
                          encrypted with the key contained in the header.

36           <xenc:EncryptedData
37                  Type = "http://www.w3.org/2001/04/xmlenc#Element"
38                  Id = "Id3">
39              <xenc:EncryptionMethod
40                  Algorithm = "http://www.w3.org/xmlenc#aes256-cbc"/>
41              <xenc:CipherData>
42                  <xenc:CipherValue>
43                           tdaqUsjXipJ09jlkjh5oinlkdsn...
44                  </xenc:CipherValue>
45              </xenc:CipherData>
46           </xenc:EncryptedData>
47        </ S11:Body >
48       </ S11:Envelope >
```

**Figure 25.5** *Illustrating XML encryption*

https://hemanthrajhemu.github.io

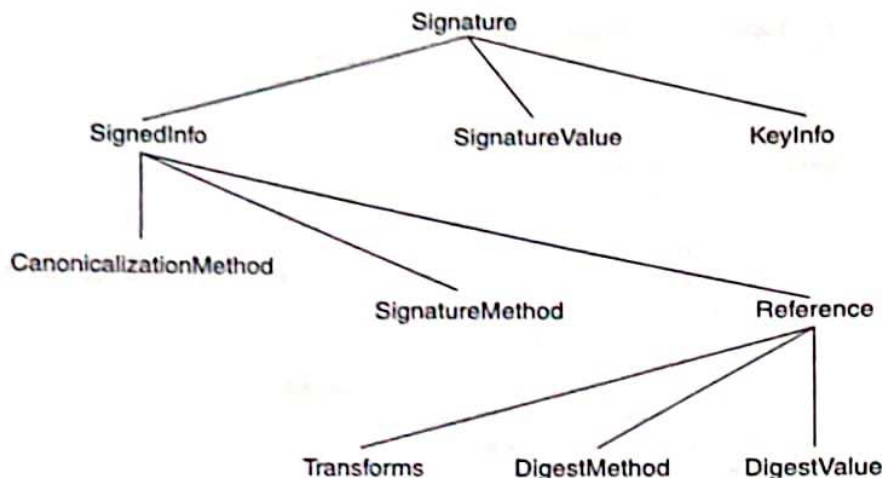



**Figure 25.6** *Tree for signature element*

A single signature is computed over possibly multiple elements in the document. Some of the elements may be in the header and others may be in the body of the SOAP message. *References* to all these are included within <SignedInfo>. Each reference is followed by the digest algorithm used in computing its digest and the digest value.

<SignatureValue > This element contains the *digital signature*. Note that the signature is computed on the entire <Signature> element. This is done by canonicalizing the <Signature> element using the canonicalization algorithm specified in the <SignedInfo> sub-element. The signature is then generated using the signature algorithm specified in the <SignedInfo> sub-element.

<KeyInfo > This element typically includes reference to *key* material that is needed for *verifying the signature* at the receiver end. For example, it may reference an X.509 certificate. The public key in the certificate would then be used to verify the signature.

The following points are worthy of note regarding the digital signature of Fig. 25.7:

- The digital signature covers three elements. Two of these are timestamps in the SOAP header – the document creation date/time (line 6) and the document expiration date/time (line 9). The third element is in the body of the SOAP envelope (lines 52 and 53). The three elements are referred to within the <SignedInfo> subelement (lines 20, 27, and 34) by their IDs: ID1, ID2, and ID3.
- The three elements are canonicalized using the canonicalization algorithm specified on lines 22, 29, and 36. The digests of the three elements appear on lines 25, 32, and 39 using digest algorithms specified on lines 24, 31, and 38.
- The entire <Signature> element is then canonicalized using the canonicalization algorithm specified on line 18.
- Finally, the canonicalized <Signature> element is signed. The signature algorithm is RSA-SHA1 (indicated on line 19). To perform signature verification, the receiver needs to know the public key corresponding to the private key used for signing. The <KeyInfo > element (line 43) contains this information. Note that it contains a reference to an element with ID = DigCert

https://hemanthrajhemu.github.io

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
                                  xmlns:xenc="..." xmlns:ds="...">
3   <S11:Header>
4         <wsu:Timestamp>
5           <wsu:Created  wsu:Id="Id1" >
6                20090418T15:35:27Z
7           </wsu:Created>
8           <wsu:Expires wsu:Id="Id2" >
9                20090418T15:37:00Z
10          </wsu:Expires>
11        </wsu:Timestamp>

12      <wsse:Security>
13        <wsse:BinarySecurityToken
                   ValueType = "...#X509v3"
                   wsu:Id = "DigCert"
                   EncodingType = "...#Base64Binary">
14             ysG7FeWSQ31pK9JhNMM...
15        </wsse:BinarySecurityToken>
16        <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
17          <SignedInfo>
18              <CanonicalizationMethod
                     Algorithm="http://www.w3.org/2001/10/xmlexcc14n#" />
19              <SignatureMethod
                     Algorithm="http://www.w3.org/2000/xmldsig#rsasha1" />

20              <Reference URI="#Id1">
21                 <Transforms>
22                    <TransformAlgorithm =
                        "http://www.w3.org/2001/xmlexcc14n#" />
23                 </Transforms>
24                <DigestMethodAlgorithm=
                       "http://www.w3.org/2000/09/xmldsig#sha1"/>
25                 <DigestValue>  Yhsl  . . .  pKl  </DigestValue>
26              </Reference>

27              <Reference URI="#Id2">
28                 <Transforms>
29                    <Transform Algorithm=
                                "http://www.w3.org/2001/10/xmlexcc14n#" />
30                 </Transforms>
31                <DigestMethodAlgorithm=
                     "http://www.w3.org/2000/09/xmldsig#sha1"/>
32                 <DigestValue>  ts7Q  . . .  0KB  </DigestValue>
33              </Reference>

34              <Reference URI="#Id3">
35                 <Transforms>
36                    <Transform Algorithm=
                                "http://www.w3.org/2001/10/xmlexcc14n#" />
37                 </Transforms>
38                 <DigestMethod Algorithm=
                     "http://www.w3.org/2000/09/xmldsig#sha1"/>
39                 <DigestValue>  m5hl  . . .  xTv  </DigestValue>
40              </Reference>
41          </SignedInfo>
42          <SignatureValue>  hMB...uaW  </SignatureValue>
43          <KeyInfo>
44               <wsse:SecurityTokenReference>
45                    <wsse:Reference URI ="#DigCert" />
46               </wsse:SecurityTokenReference>
47           </KeyInfo>

48        </Signature>
49     </ wsse:Security>
50   </ S11:Header>

51   < S11:Body wsu:Id = "#Id3" >
52          Include the body too in the computation
53          of the digital signature.
54   </ S11:Body >
55  </ S11:Envelope >
```

**Figure 25.7** *Illustrating XML signatures*

https://hemanthrajhemu.github.io

which is a BinarySecurityToken on line 13. The ValueType of this token indicates that it is an X.509 certificate. The certificate is encoded in Base64 and is attached.

## 25.4 SAML

### 25.4.1 Motivation

Consider a principal, C, who happens to be a long-term client of a service provider, SP1. Each time C requests a service from $SP_1$, he needs to be authenticated. This can be done through the standard login name–password route.

Another possibility is for SP1 to store a cookie in C's browser which would be transparently dispatched to SP1 each time C visits SP1's website. The *browser cookie* could store, in possibly encrypted form, information about C's identity. Relevant attributes of C may be also stored in browser cookies or at the server.

Now, suppose C wishes to use the services of another provider, SP2. If C is a total stranger to SP2, on what basis might SP2 trust C? If, however, SP1 and SP2 share a trust relationship, SP2 could read the cookies stored in C's browser created by SP1. The cookie could include information such as

"SP1 trusts C".

If SP2 knows that SP1 trusts C, then SP2 might also be willing to trust C.

Such a solution has serious problems. In particular, browsers do not allow cookies created by one server to be dispatched to a server in a different domain. So, for example, a cookie created by the web server at www.cse.iitb.ac.in can be read by a web server at www.ee.iitb.ac.in but not by the web server at www.cse.iitk.ac.in.

### 25.4.2 Assertion Types

The *Security Assertions Markup Language* (SAML) developed in May 2002 by OASIS (Organization for Advancement of Structured Information Systems)[1] is a standard that addresses such problems. Basically, SAML provides XML schema for expressing assertions about a principal. For example, SP1 might make the following assertion:

SP1 authenticated C
using password-based authentication
on 1st February 2010
at 09:25:15 hours.

In the example above, SP1 is the *asserting party*. In SAML terminology, it performs the role of an *Identity Provider* (I). SP2 is a *consumer of assertions* and is referred to as the *relying party*. There are some key questions to be addressed here regarding when and how assertions are transmitted from the asserting party to the relying party. Before that, we look at the three types of SAML assertions and their XML syntax.

- An *authentication statement* is an assertion by Identity Provider, I, that it did authenticate a principal, C, using a particular authentication method at a particular point of time.
- An *attribute statement* is an assertion by Identity Provider, I, that the value of attribute *A* for principal C is *a*.

[1]The current version is 2.0, released in March 2005.

https://hemanthrajhemu.github.io

- An *authorization statement* is an assertion by Identity Provider, I, that a principal, C, is permitted to perform an action or operation *O* on resource *R*.

An example of a SAML assertion is shown in Fig. 25.8. It is an authentication statement containing the *identities* of the *issuer* and *principal*. A URL is used to identify the issuer and an e-mail address is used to identify the principal (lines 2 and 5). The statement indicates the date/time at which the principal was authenticated (line 12). It asserts that the principal was authenticated using a *password* transmitted across a *protected channel* (using SSL). It also includes an explicit condition that the authentication is valid for the next 26 hours.

```
1    <saml:Assertion  xmlns:saml = . . .    Version = "2.0"
                                           IssueInstant = "2010-02-01T08:25:15Z">

2            <saml:Issuer Format= . . . :entity>
                         http://www.admin.iitb.ac.in
3            </saml:Issuer>

4            <saml:Subject>
5                    <saml:NameID   Format = " . . . :emailAddress">
                             rajeshX@cse.iitb.ac.in
6                    </saml:NameID>
7            </saml:Subject>

8            <saml:Conditions
9                    NotBefore = "2010-02-01T08:26:00Z"
10                   NotOnOrAfter = "2010-02-02T10:30:00Z"
11           </saml:Conditions>

12           <saml:AuthnStatement  AuthnInstant = "2010-02-01T08:25:15Z"
                                                 SessionIndex = "1234">
13                   <saml:AuthnContext>
14                           <saml:AuthnContextClassRef>
                                     . . . :PasswordProtectedTransport
15                           </saml:AuthnContextClassRef>
16                   </saml:AuthnContext>
17           </saml:AuthnStatement>
18   </saml:Assertion>
```

**Figure 25.8** *SAML assertion – Authentication statement*

### 25.4.3 Creating/Communicating Assertions

A useful application of SAML is in *single sign-on*. We now consider a usage scenario of single sign-on over the web.

A user, Sandeep, visits the website of his familiar travel agent, SmartTravels in order to book a ticket to say, Rio. Sandeep logs in and authenticates himself at the SmartTravels site. He indicates travel preferences including date/time of departure, budgetary constraints, etc. He is presented with

https://hemanthrajhemu.github.io

a choice of airlines satisfying his requirements. After clicking on his preferred airline, Jet Air, he is seamlessly directed to the website of that airline where he makes a reservation.

Now suppose that Sandeep is a gold customer of SmartTravels – a status conferred on all customers of SmartTravels who have done business in excess of Rs. 4,00,000 over the last 4 years. SmartTravels has business relationships with several airlines, including Jet Air, which provide varying discounts to all of its gold customers. How is Jet Air expected to know that Sandeep is a gold customer of SmartTravels and is eligible for the discounted price?

For the sake of completeness, we enumerate all the steps involved in Sandeep's transaction.

1. Sandeep logs in to the SmartTravels website and is *authenticated*. He indicates the destination city, date and time of travel, and price of ticket he is willing to pay.
2. SmartTravels determines that Sandeep is a gold customer and presents a list of airlines that satisfy Sandeep's requirements.
3. Sandeep clicks on the airline of interest, say JetAir.
4. SmartTravels creates *SAML assertions* indicating that
   a. Sandeep has been authenticated using a login name–password mechanism (authentication assertion)
   b. Sandeep is a gold customer (attribute assertion)
5. SmartTravels creates an HTML form with two hidden inputs. The first, named *SAMLResponse*, contains the signed SAML assertion. The second hidden input, called RelayState, contains the *URL of the resource* required by Sandeep. The relevant portion of the form is

```
<html>
<body onload = "document.forms.SAMLform.submit( );">
<form name = SAMLform method = post
        action = "www.JetAir.com/Reservations/SAMLConsumer">
<input type = hidden name = "SAMLResponse"
                        value = " . . . signed assertion . . . "/>
<input type = hidden name = "RelayState"
                        value = "encodedTargetURL"/>
</form>
</body>
</html>
```

   This form is sent to Sandeep's browser.
6. When Sandeep's browser receives the form, it is immediately re-directed to www.JetAir.com/Reservations/SAMLConsumer.
7. The assertions are consumed by the JetAir web server. It is now aware that Sandeep is a gold customer of its business partner – SmartTravels. It returns Sandeep a page containing information on its travel schedules and special fares.

It is possible that Sandeep's travel plans include booking accommodation at a hotel and also renting a car at his travel destination. The above example could then be extended so that Sandeep *logs in just once* – at the travel agent's website. From there he is able to visit the other websites and make reservations – for travel by air, for accommodation at the hotel, and at a car rental agency.

https://hemanthrajhemu.github.io

# 25.5 OTHER STANDARDS

## 25.5.1 WS-Trust

Two end points of a web service may have never interacted with each other. To build trust between themselves, they could use an intermediary known to both parties who could create a SAML token on behalf of the party that needs to be authenticated. This is just one way of establishing trust. We need a framework that is more powerful and general. Here is a concise wish list.

- Our framework should encompass different kinds of trust – direct trust and indirect trust brokered by one or more intermediaries.
- Besides SAML tokens, we need to support other token types – simple password-based tokens, digital certificates, Kerberos tickets, etc.
- Our framework should be able to define messages for requesting security tokens from a Security Token Service. We also need to define messages that communicate the security tokens. Finally, we need to validate tokens received from a client.

The WS-Trust standard is exactly the response to our wish list.

*Example 25.1*

Consider an importer, I, who wishes to import goods from an exporter, E, in another country. I and E are not known to each other and so need to establish a trust relationship with each other. I and E have trust relationships with their "local" banks, IE and IB, respectively. IB and EB do not have a direct trust relationship with each other. They each do, however, have a trust relationship with an intermediary – a well-known international bank called the Correspondent Bank, CB (see Fig. 25.9). The trust relationships are summarized below.

$$I \leftrightarrow IB \qquad IB \leftrightarrow CB$$
$$E \leftrightarrow EB \qquad EB \leftrightarrow CB$$

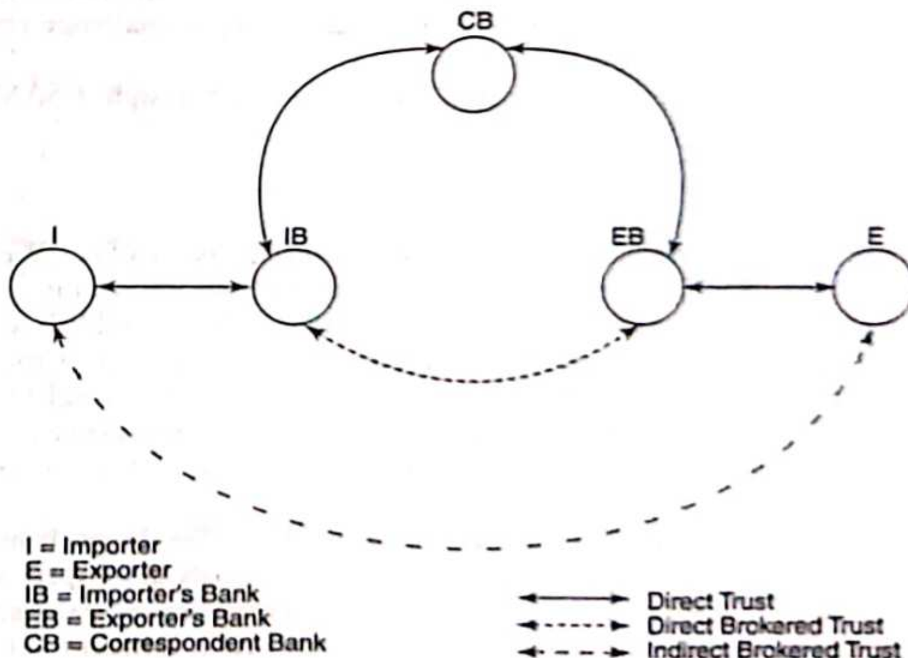



**Figure 25.9** *Trust relationship between entities involved in international trade*

https://hemanthrajhemu.github.io

Before the importer and exporter can transact securely, they need to establish a relationship of mutual trust. Basically, I needs to securely communicate its credentials to E. The credentials are security tokens acceptable to E (i.e., verifiable by E). WS-Trust could be used in the following manner:

- I authenticates himself to IB.
- Since IB and CB trust each other, IB requests CB to issue a security token to be used by I.
- CB creates a security token and includes information such as the maximum credit amount extended to I. CB acts as a Security Token Service Provider. CB communicates this token to IB who forwards it to I.
- I dispatches the token to E.
- E requests EB to validate the token. EB may be able to validate the token on its own. If not, it sends it to CB for validation.
- The success or failure of the validation process is communicated by CB to E through EB.

### 25.5.2 WS-SecurityPolicy

WS-SecurityPolicy enables a web service to specify the security tokens it will accept for authentication and access control. For example, it might state that it accepts either X.509 certificates or Kerberos tickets. It conveys information about whether it requires all or part of the client's message to be encrypted. If encryption is required, it will indicate the encryption algorithms supported and key size. Also, the security policy may require that all or part of the message be integrity-protected. In that case, it will also specify the integrity check algorithm.

WS-SecurityPolicy assertions are communicated as part of the web service's WSDL. Alternatively, it may be included in entries in the UDDI registry related to the web service provider.

Returning to the example of Fig. 25.9, the policies regarding authentication laid out by the relevant entities may be as follows:

- An importer must authenticate himself to his local bank via a login name and password.
- A local bank must authenticate itself to the global bank using a challenge-response protocol in conjunction with a digital certificate.
- An importer must authenticate himself to the given exporter through a SAML token signed by a global bank.

### 25.5.3 The Big Picture

The most basic standard in providing security for web services is WS-Sec. We discussed *WS-SecurityPolicy*, *SAML*, and *WS-Trust* because we feel these are also important.

*WS-SecureConversation* is a standard that enables two principals to establish a *session context*. The state of a session includes a secret key that they share for the duration of the session. Setting up such a session is especially efficient if two parties need to exchange several messages during a session. This reduces the overhead of creating a shared secret for every message exchanged. The idea of a session context is the application layer analogue of the session in SSL or the IPSec security association.

WS-Trust builds on WS-Sec. Similarly, *WS-Federation* builds on WS-Trust. It helps *broker trust* between entities in different security domains. Many of the standards developed for Web Services Security are complementary to each other. A single application may use several standards to realize its security goals. At the same time, different developers might use a different subset of standards to achieve their security goals for the same application.

https://hemanthrajhemu.github.io

## SELECTED REFERENCES

www.w3.org, the website of the World Wide Web Consortium ($W^3C$), is an excellent source of information for the many standards related to web services (for example, WSDL) and web services security. [XMLENC] and [XMLDSIG] are good sources for XML encryption and XML signatures.

The OASIS website (www.oasis-open.org) contains a wealth of information on many of the other standards related to web services security. For example, SAML assertions, protocols, bindings, etc. are at [SAML]. The specification of WS-Trust version 1.4 appears in [WS-TRUST], while WS-SecurityPolicy appears in [WSSECPOL].

## OBJECTIVE-TYPE QUESTIONS

**25.1** A SOAP binding refers to
- (a) the objects bound to a SOAP message
- (b) the XML schema of a SOAP message
- (c) the mapping between a SOAP message and the underlying transport protocol
- (d) the headers in a SOAP message

**25.2** A WSDL operation involves at least one of which of the following:
(a) ports (b) portTypes (c) methods (d) messages

**25.3** Which of the following is/are not example(s) of a WS-Sec security token:
- (a) a Kerberos ticket
- (b) a signature algorithm
- (c) a username + password
- (d) an X.509 certificate

**25.4** Which of the following statements is/are true of XML encryption?
- (a) The encryption key is always enclosed in the SOAP message
- (b) An entire element (including tags) needs to be encrypted, not just the content of the element
- (c) An encrypted SOAP message must include a ⟨Security⟩ element in its header
- (d) The ciphertext is contained in the ⟨CipherData⟩ sub-element

**25.5** The SignedInfo sub-element inside a Signature element contains
- (a) the name of the signature algorithm used
- (b) the digital signature that is computed
- (c) a reference to the key used for signature verification
- (d) the canonicalization algorithm employed

**25.6** Which of the following is not a SAML assertion?
- (a) An Identification statement
- (b) An Authentication statement
- (c) An Attribute statement
- (d) An Authorization statement

**25.7** Which of the following is/are true of WS-Trust?
- (a) It handles only direct, not indirect trust
- (b) It defines messages for requesting and validating security tokens
- (c) It specifies the tokens that a web service accepts for access control
- (d) It defines various schemes for key management

https://hemanthrajhemu.github.io

## EXERCISES

**25.1** Identify two web-based applications for which SSL is appropriate and two applications for which it is not appropriate. In each case, explain clearly why it is appropriate or why it is not.

**25.2** All of the standards for web services security that we have studied are based on XML. State and explain three main reasons why XML is employed. Also, is there any drawback in using XML? If so, what?

**25.3** Figure 25.6 shows the XML tree for the Signature element. Show the XML trees for the EncryptedKey and EncryptedData elements used for XML encryption.

**25.4** The body of a certain SOAP message comprises 10 lines of text. The first four lines of the body need to be signed and then encrypted. The sender generates an RSA signature using her private key. She encloses her certificate in the message so that the recipient, Jignesh can verify the signature. Assume that encryption uses a session key that is itself encrypted and sent as part of the SOAP message. The session key is encrypted with the public key of Jignesh.

(a) Show the complete SOAP message. Make sure you include the elements containing the encrypted text, the signature, the encrypted key and certificates. The SOAP message should also indicate the algorithms used for encryption, signing, hash computation, canonicalization, etc.

(b) How would the SOAP message change if encryption were to precede signing?

**25.5** In the example of Section 25.4.3, Sandeep is a Gold Customer of SmartTravels. By virtue of this status, he is able to receive attractive air fare discounts from airlines which have a business relationship with SmartTravels. After authentication at the website of SmartTravels, Sandeep is presented with a list of flight schedules and airlines. By clicking on the airline of choice, say JetAir, he is directed to the website of JetAir.

(a) List the SAML assertions that SmartTravels makes to JetAir on behalf of Sandeep so that he can avail of JetAir's special discounts.

(b) How are these assertions communicated to JetAir?

(c) Can these assertions be spoofed? If so, why? How can you prevent such spoofing?

## ANSWERS TO OBJECTIVE-TYPE QUESTIONS

**25.1** (c) **25.2** (d) **25.3** (b) **25.4** (c)
**25.5** (a)(d) **25.6** (a) **25.7** (b)

https://hemanthrajhemu.github.io