Security

Chapter 9

The Security Environment
Threats

Goal	Threat
Confidentiality	Exposure of data
Integrity	Tampering with data
Availability	Denial of service

Figure 9-1. Security goals and threats.

Can We Build Secure Systems?

Two questions concerning security:

1. Is it possible to build a secure computer system?
2. If so, why is it not done?

Trusted Computing Base

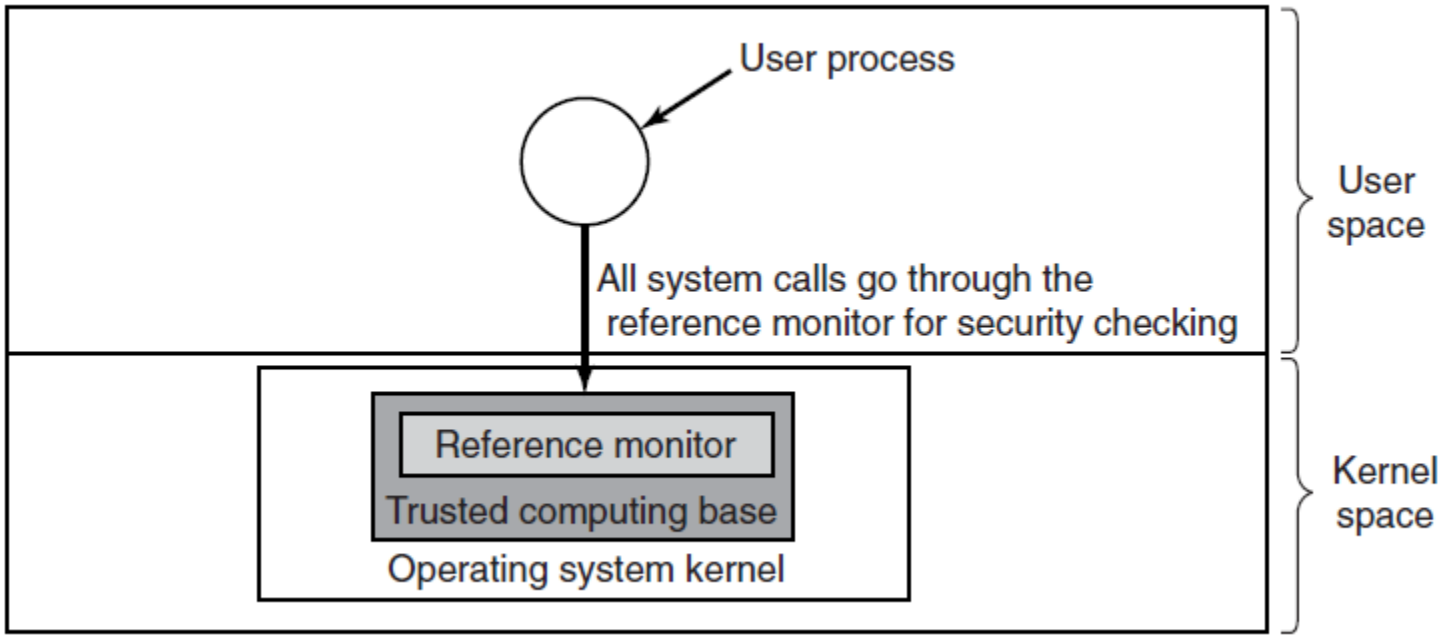


Figure 9-2. A reference monitor.

Protection Domains (1)

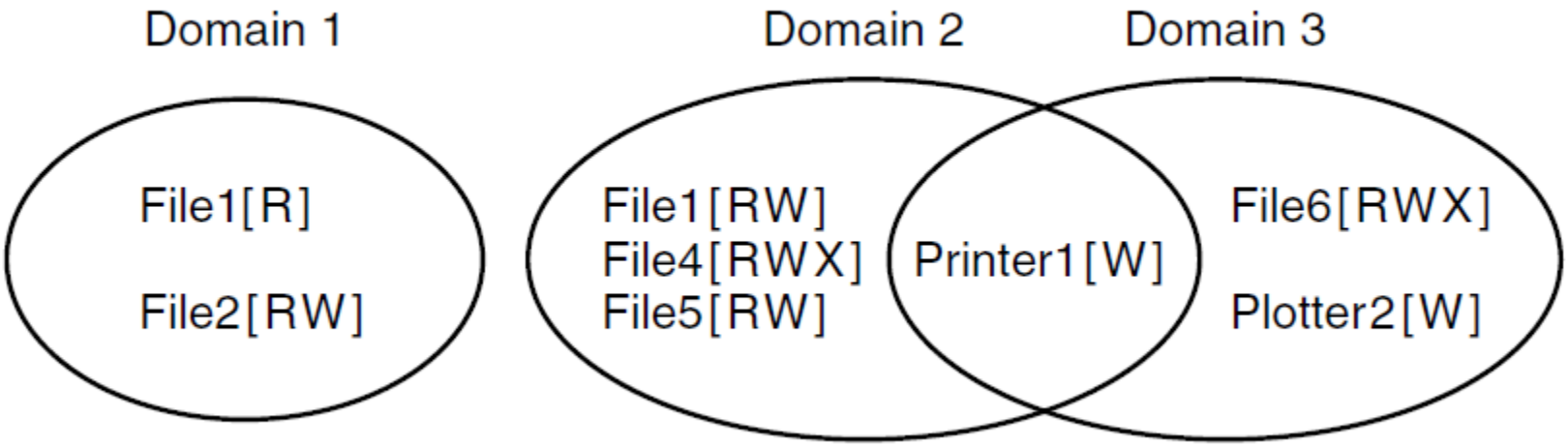


Figure 9-3. Three protection domains.

Protection Domains (2)

	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain 1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 9-4. A protection matrix.

Protection Domains (3)

Domain	Object: File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 9-5. A protection matrix with domains as objects.

Access Control Lists (1)

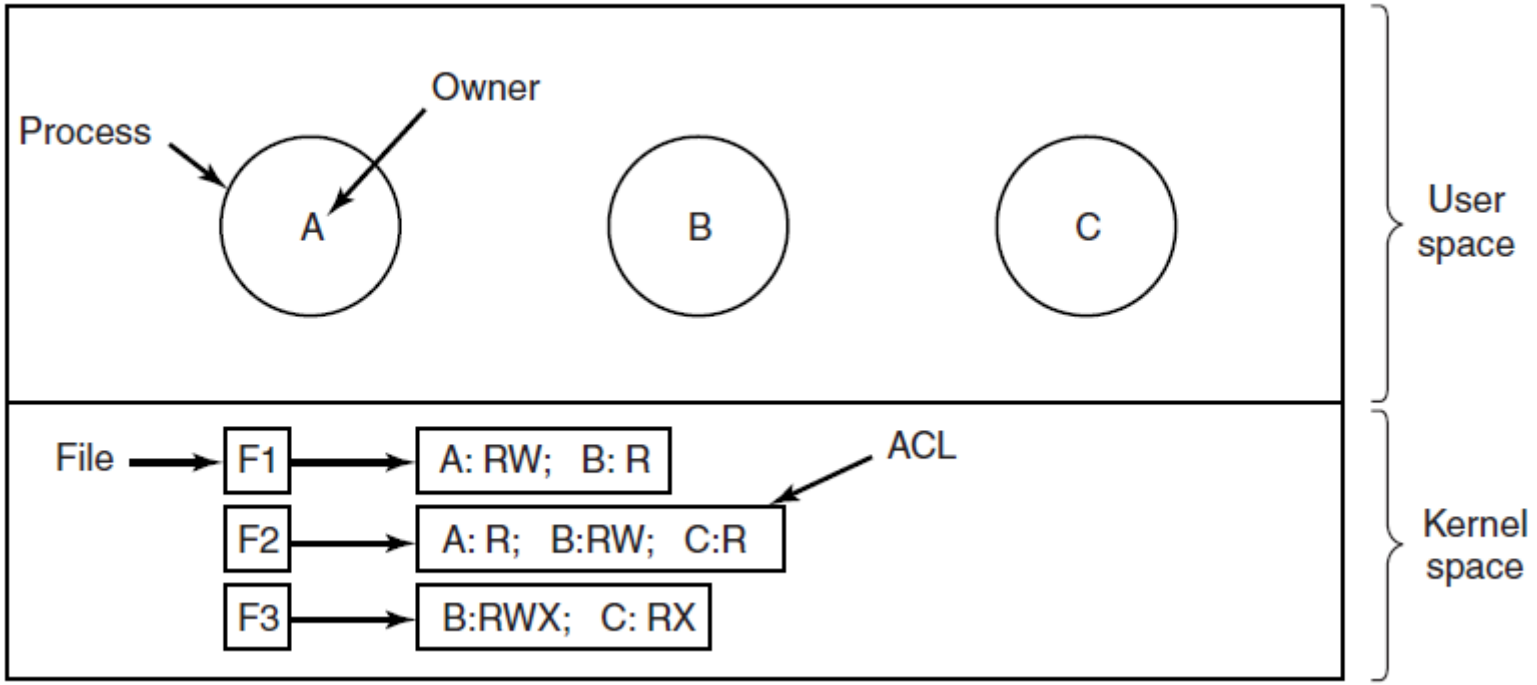


Figure 9-6. Use of access control lists to manage file access.

Access Control Lists (2)

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 9-7. Two access control lists.

Capabilities (1)

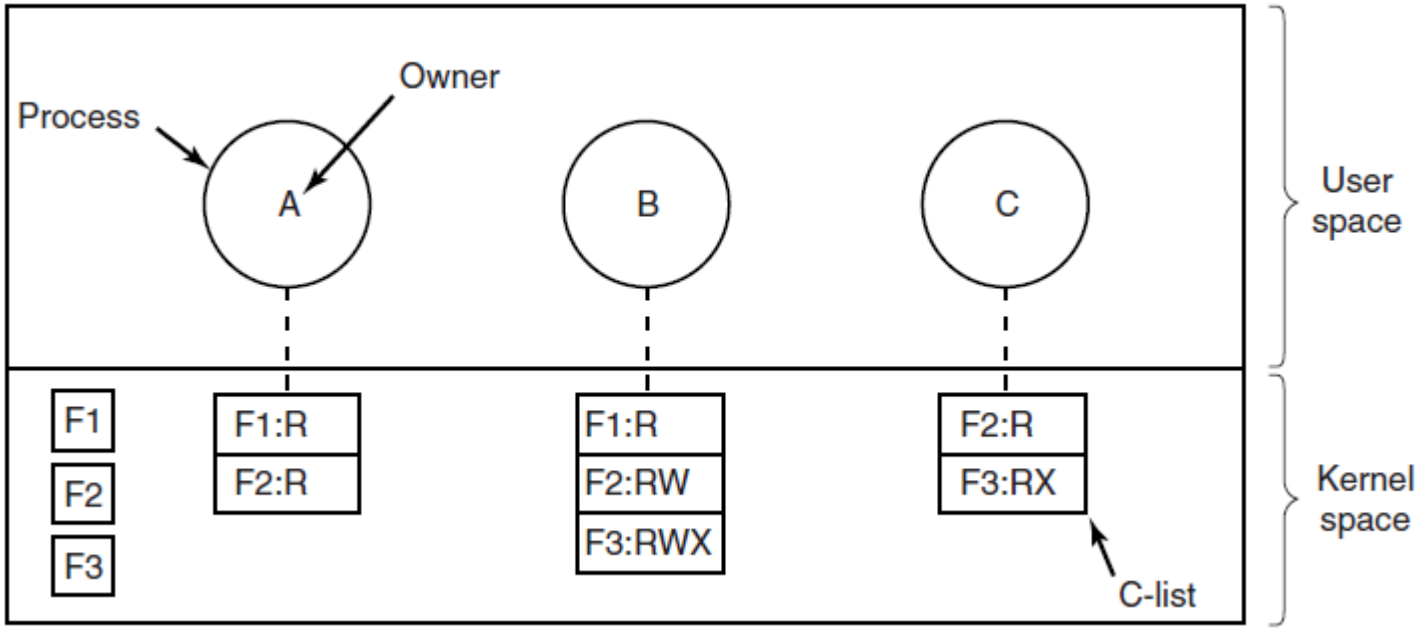


Figure 9-8. When capabilities are used, each process has a capability list.

Capabilities (2)

Figure 9-9. A cryptographically protected capability.

Capabilities (3)

Examples of generic rights:

1. Copy capability: create new capability for same object.
2. Copy object: create duplicate object with new capability.
3. Remove capability: delete entry from C-list; object unaffected.
4. Destroy object: permanently remove object and capability.

Formal Models of Secure Systems

Objects

	Compiler	Mailbox 7	Secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute		Read Write

(a)

Objects

	Compiler	Mailbox 7	Secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute	Read	Read Write

(b)

Figure 9-10. (a) An authorized state.
(b) An unauthorized state.

Multilevel Security
Bell-LaPadula Model

Bell-LaPadula Model rules for information flow:

1. The simple security property
   - Process running at security level k can read only objects at its level or lower
2. The * property
   - Process running at security level k can write only objects at its level or higher

Bell-LaPadula Model

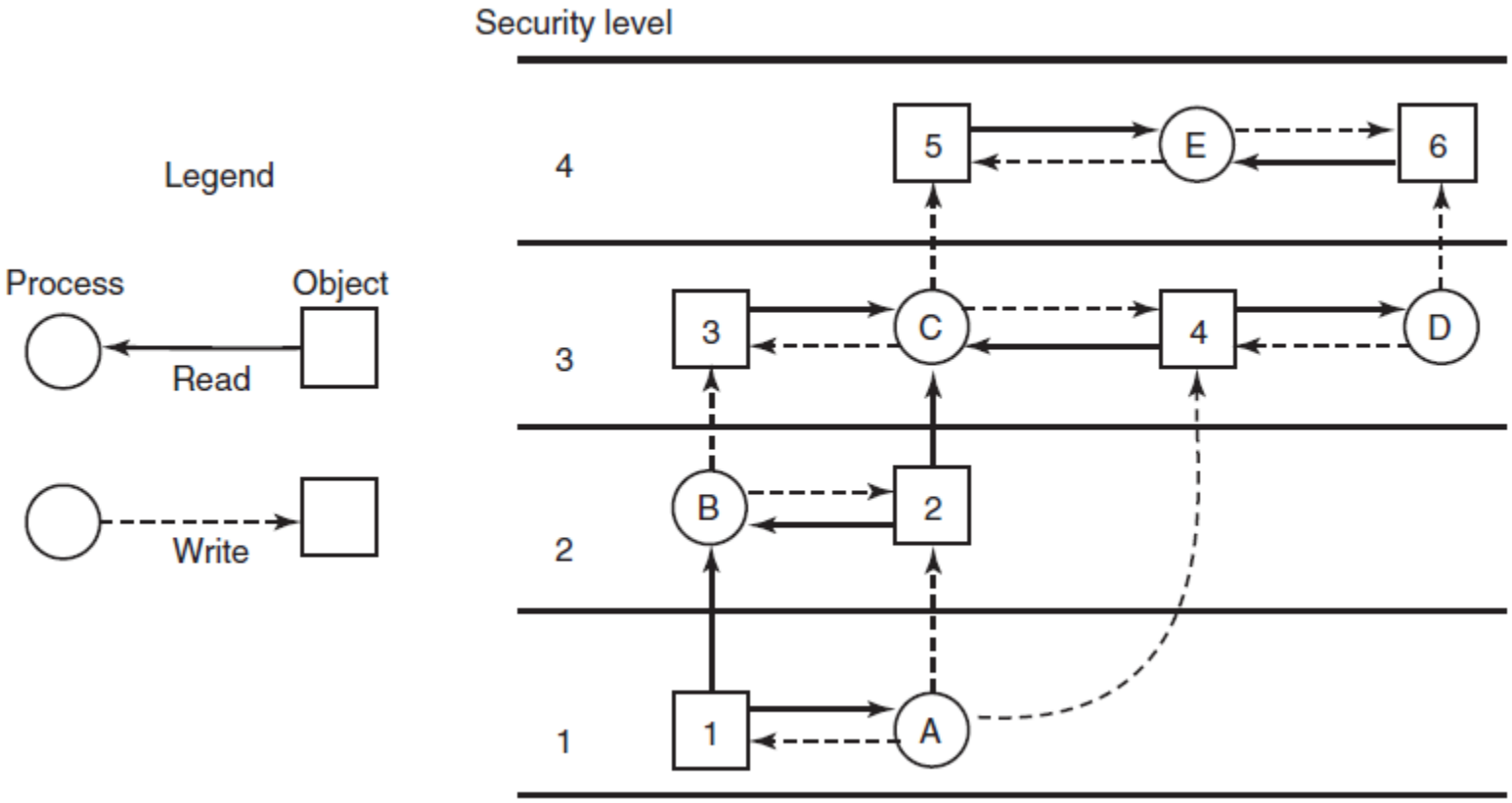


Figure 9-11. The Bell-LaPadula multilevel security model.

The Biba Model

To guarantee the integrity of the data:

1. The simple integrity principle
   - process running at security level k can write only objects at its level or lower (no write up).
2. The integrity * property
   - process running at security level k can read only objects at its level or higher (no read down).

Covert Channels (1)

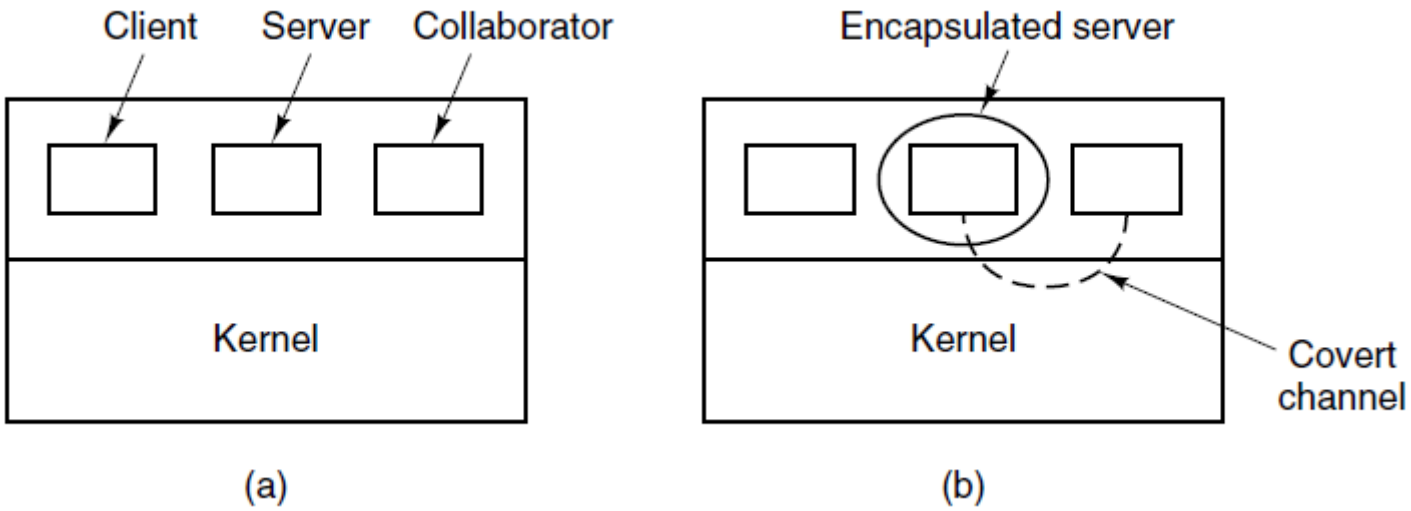


Figure 9-12. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

Covert Channels (2)

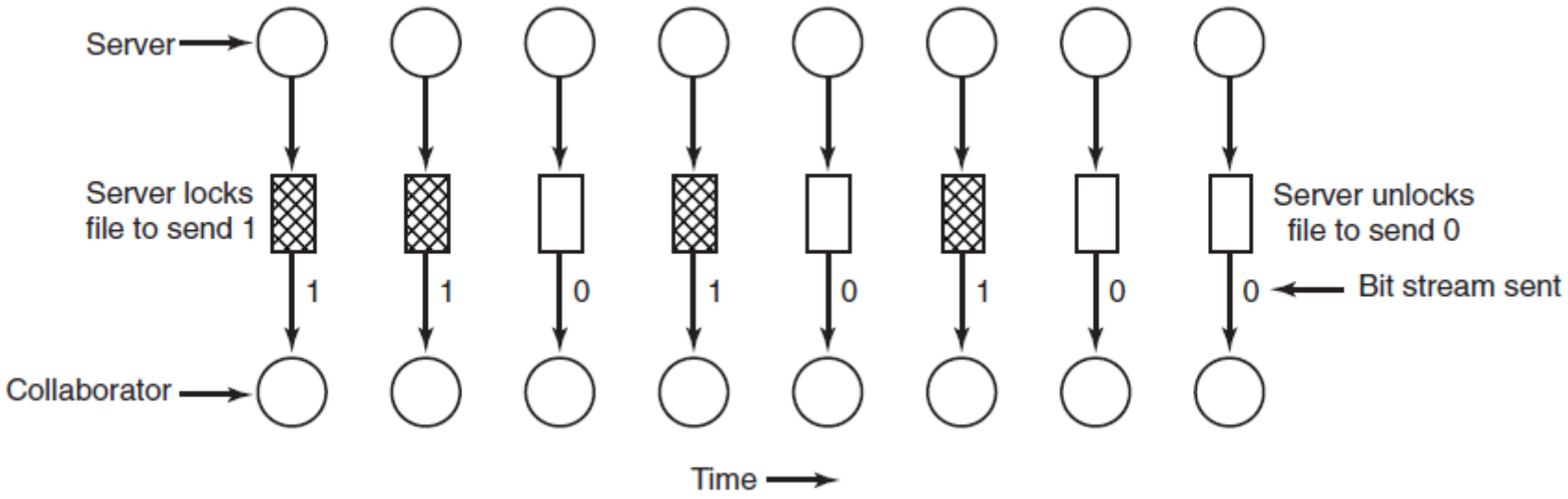


Figure 9-13. A covert channel using file locking.

Steganography

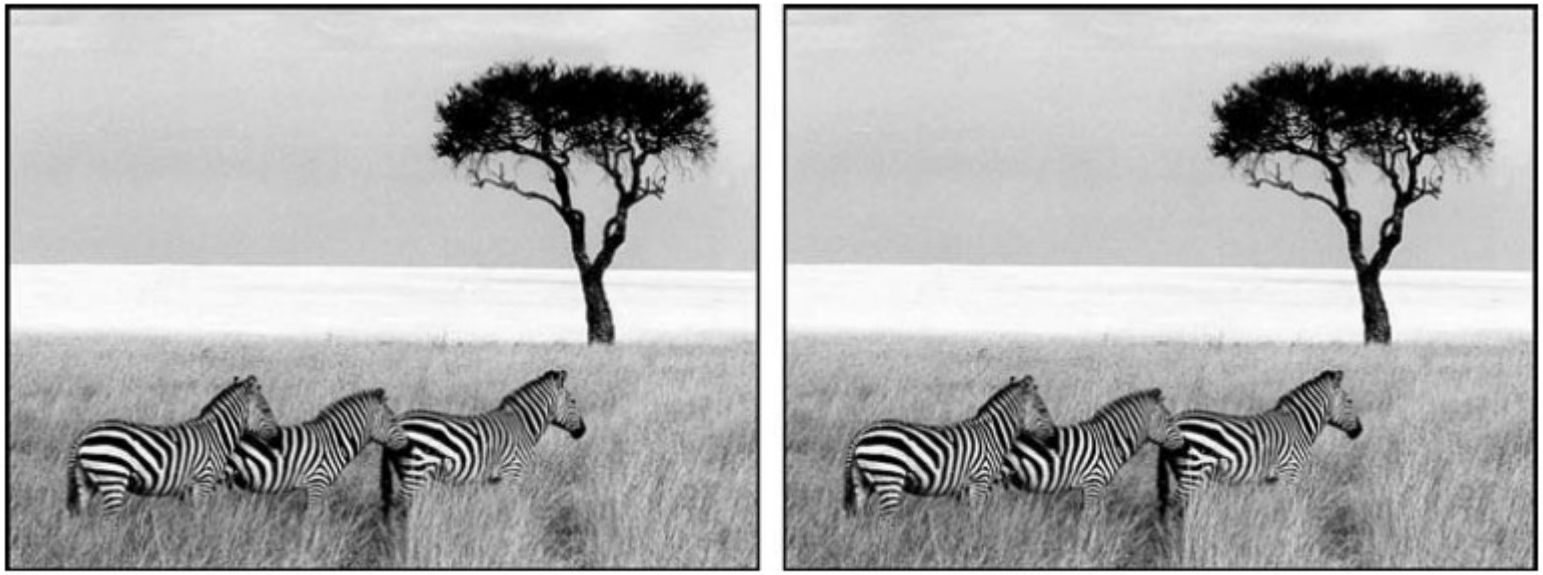


Figure 9-14. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

Basics of Cryptography

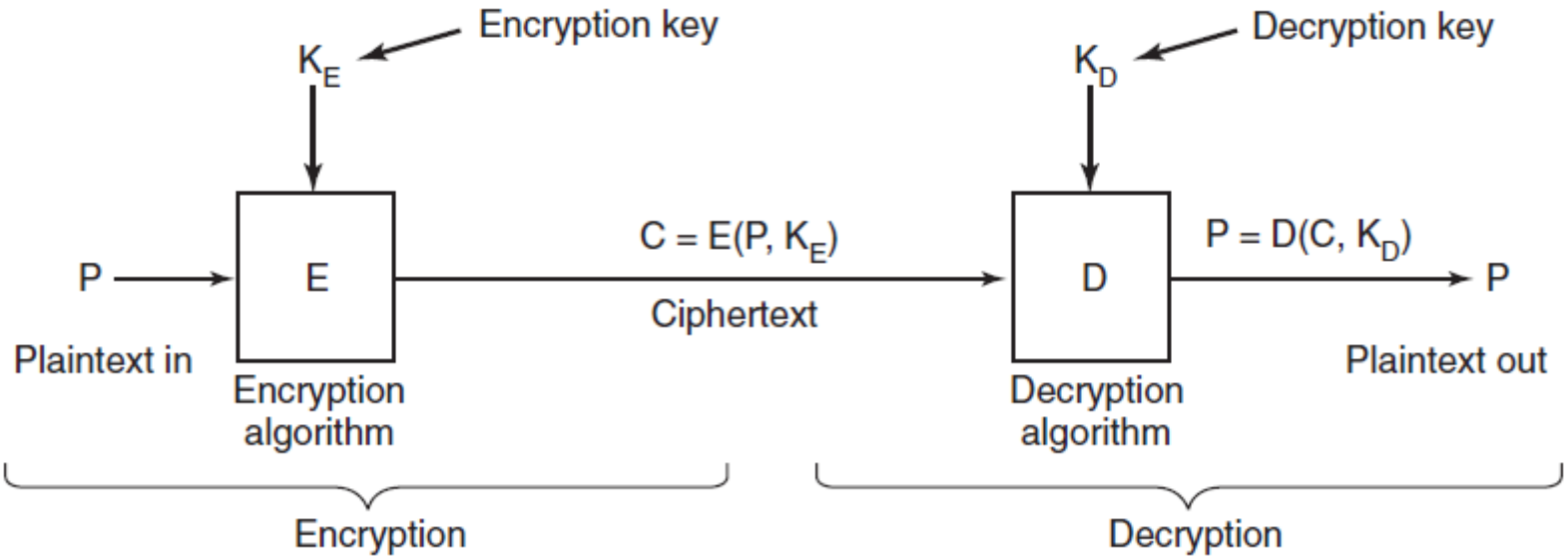


Figure 9-15. Relationship between the plaintext and the ciphertext.

Secret-Key Cryptography

plaintext:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
ciphertext:	Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

An encryption algorithm in which each letter is replaced by a different letter.

Digital Signatures

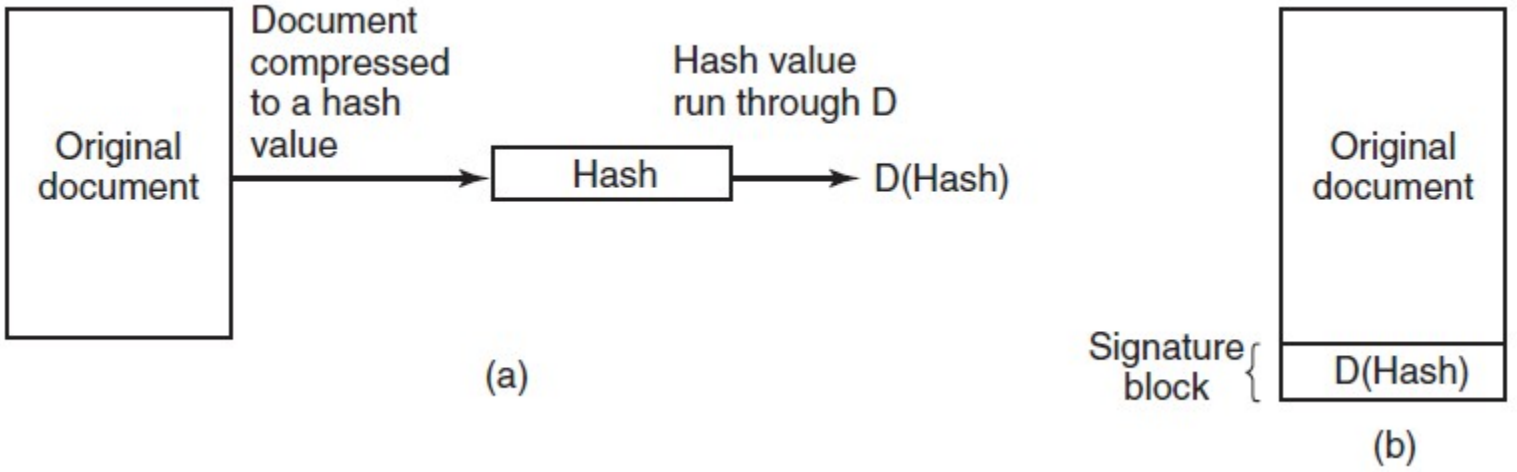


Figure 9-16. (a) Computing a signature block. (b) What the receiver gets.

Authentication (1)

Methods of authenticating users when they attempt to log in based on one of three general principles:

1. Something the user knows.
2. Something the user has.
3. Something the user is.

Authentication (2)

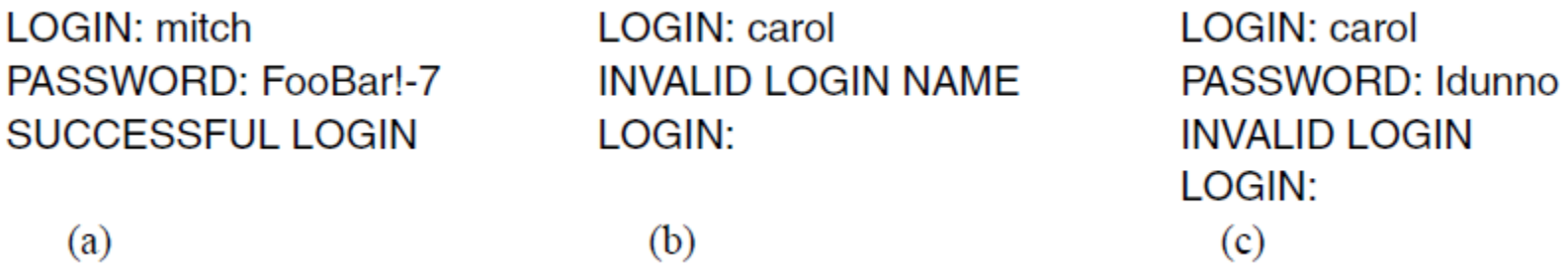


Figure 9-17. (a) A successful login. (b) Login rejected after name is entered. (c) Login rejected after name and password are typed.

UNIX Password Security

Bobbie, 4238, e(Dog, 4238)
Tony, 2918, e(6%%TaeFF, 2918)
Laura, 6902, e(Shakespeare, 6902)
Mark, 1694, e(XaB#Bwcz, 1694)
Deborah, 1092, e(LordByron,1092)

Figure 9-18. The use of salt to defeat precomputation of encrypted passwords.

Challenge-Response Authentication

Questions should be chosen so that the user does not need to write them down.
Examples:

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Ellis teach?

Authentication Using a Physical Object

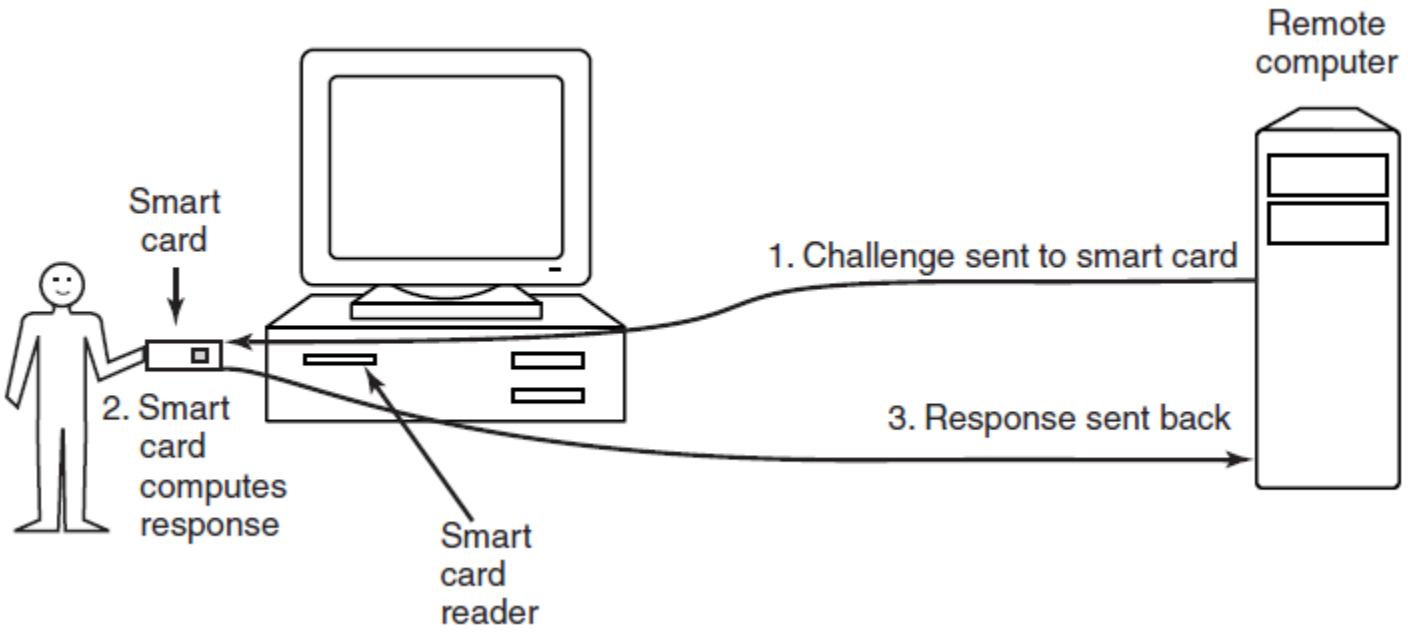


Figure 9-19. Use of a smart card for authentication.

Authentication Using Biometrics

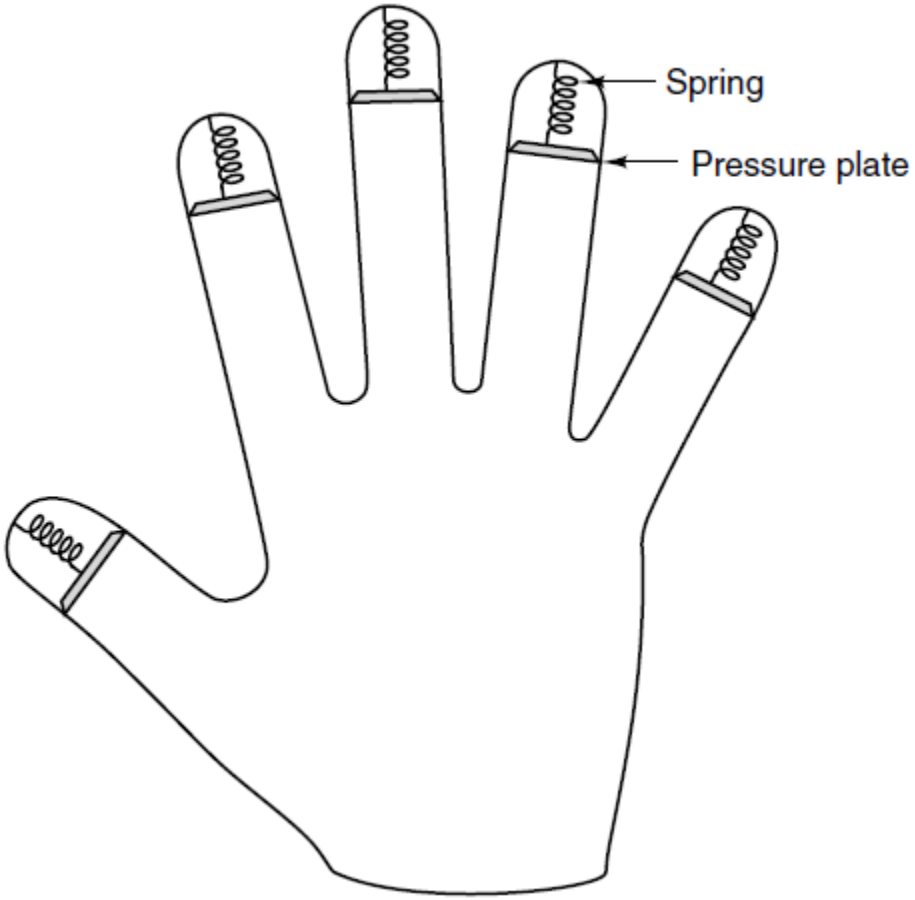


Figure 9-20. A device for measuring finger length.

Buffer Overflow Attacks

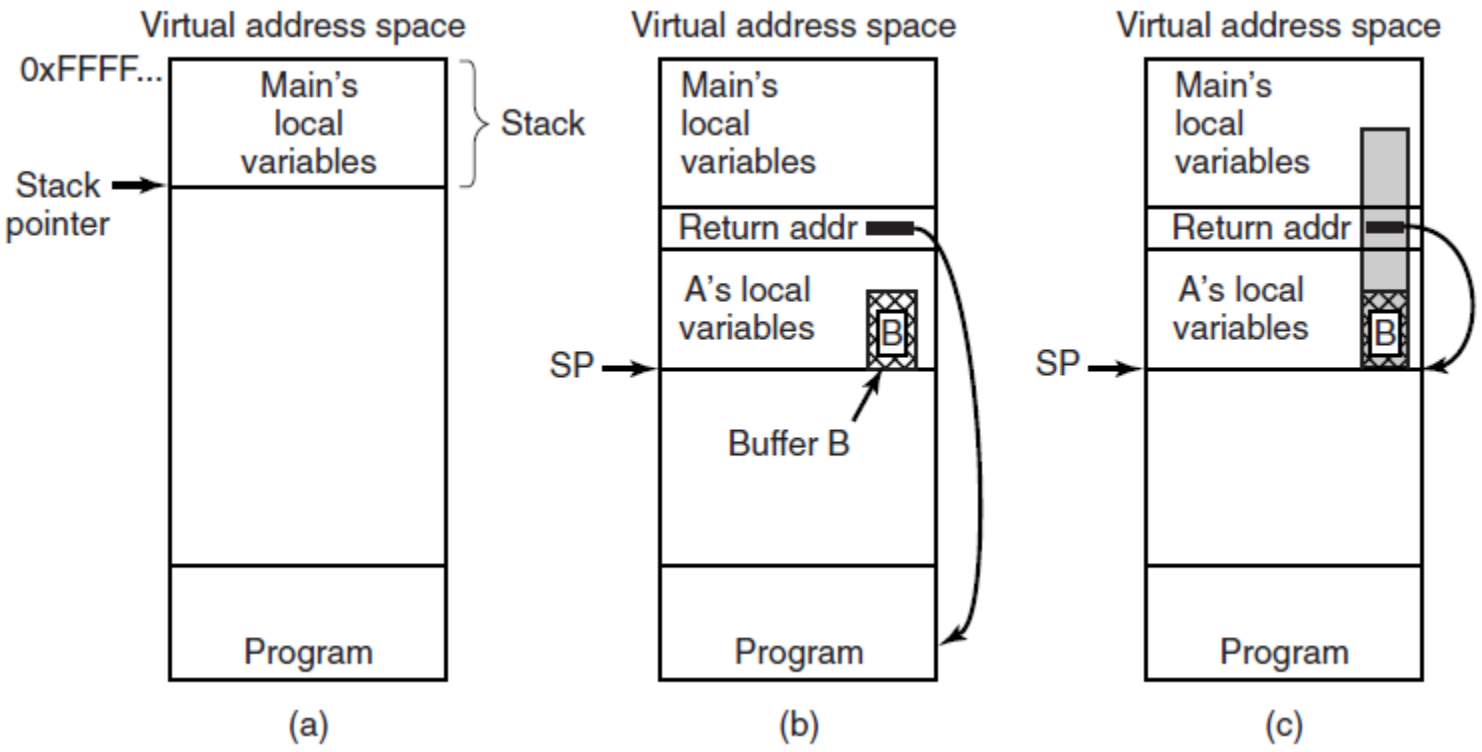


Figure 9-21. (a) Situation when the main program is running. (b) After the procedure A has been called.
(c) Buffer overflow shown in gray.

Avoiding Stack Canaries

```
01. void A (char *date) {
02.     int len;
03.     char B [128];
04.     char logMsg [256];
05.
06.     strcpy (logMsg, date);   /* first copy the string with the date in the log message */
07.     len = strlen (date);     /* determine how many characters are in the date string */
08.     gets (B);                /* now get the actual message */
09.     strcpy (logMsg+len, B);/* and copy it after the date into logMessage */
10.     writeLog (logMsg);       /* finally, write the log message to disk */
11. }
```

Figure 9-22. Skipping the stack canary: by modifying *len* first, the attack is able to bypass the canary and modify the return address directly.

Code Reuse Attacks

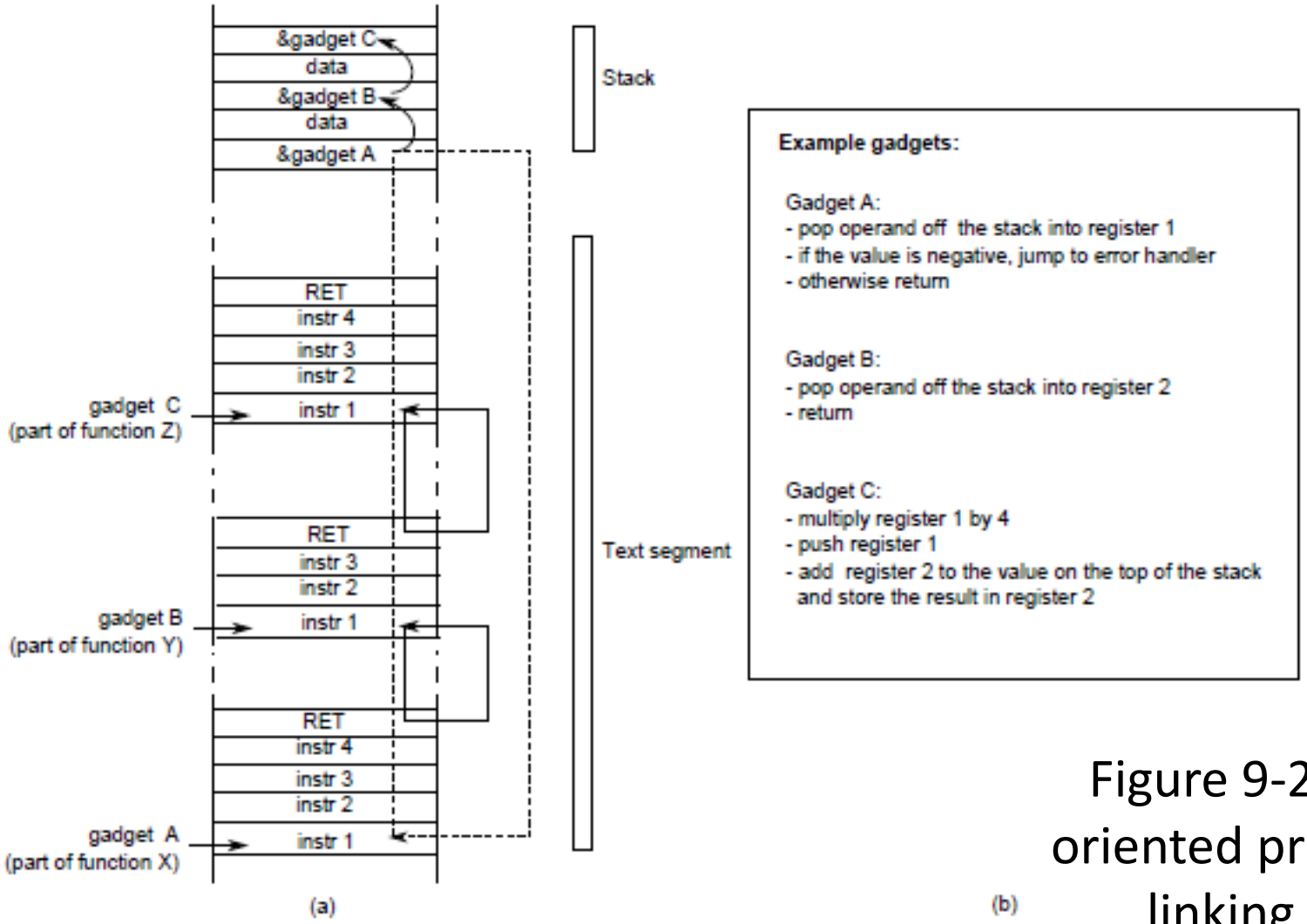
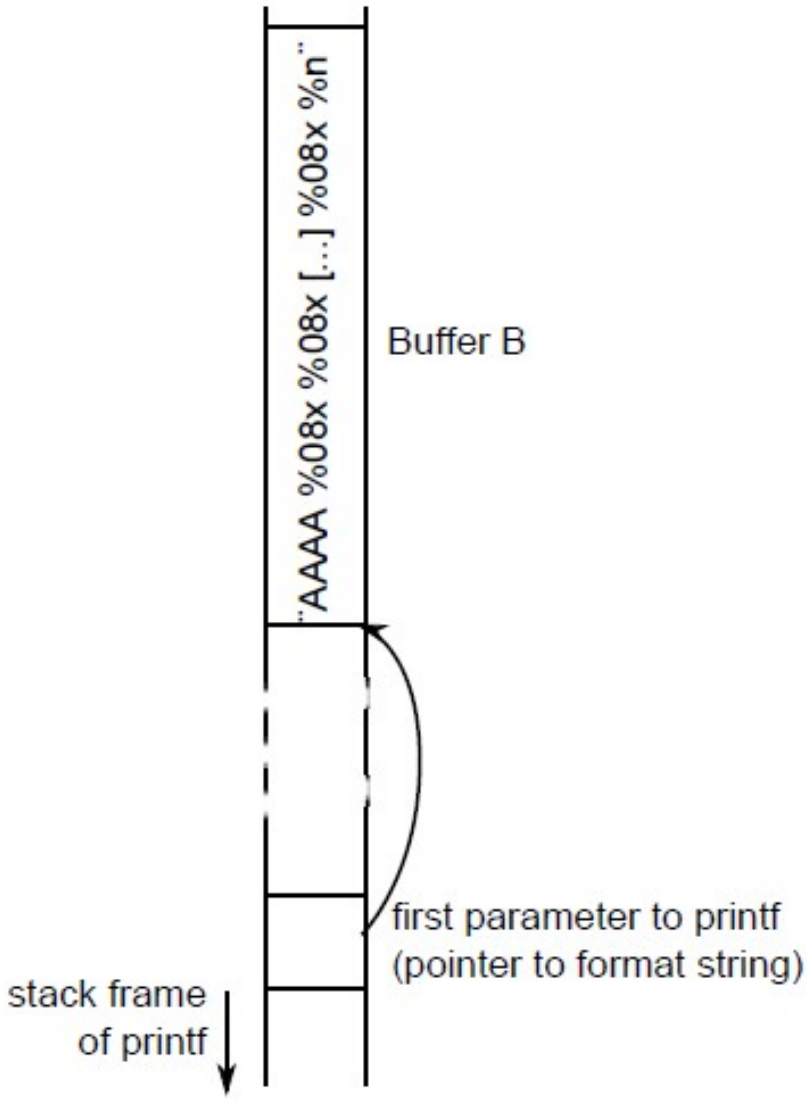


Figure 9-23. Return-oriented programming: linking gadgets

Format String Attacks

Figure 9-24. A format string attack. By using exactly the right number of %08x, the attacker can use the first four characters of the format string as an address.



Command Injection Attacks

```
int main(int argc, char *argv[])
{
  char src[100], dst[100], cmd[205] = "cp ";          /* declare 3 strings */
  printf("Please enter name of source file: ");       /* ask for source file */
  gets(src);                                          /* get input from the keyboard */
  strcat(cmd, src);                                   /* concatenate src after cp */
  strcat(cmd, " ");                                   /* add a space to the end of cmd */
  printf("Please enter name of destination file: ");  /* ask for output file name */
  gets(dst);                                          /* get input from the keyboard */
  strcat(cmd, dst);                                   /* complete the commands string */
  system(cmd);                                        /* execute the cp command */
}
```

Figure 9-25. Code that might lead to a command injection attack.

Back Doors

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

(b)

Figure 9-26. (a) Normal code.
(b) Code with a back door inserted.

Login Spoofing

Figure 9-27. (a) Correct login screen. (b) Phony login screen.

Executable Program Viruses (1)

```
#include <sys/types.h>                  /* standard POSIX headers */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                       /* for lstat call to see if file is sym link */

search(char *dir_name)
{                                       /* recursively search for executables */
    DIR *dirp;                          /* pointer to an open directory stream */
    struct dirent *dp;                  /* pointer to a directory entry */

    dirp = opendir(dir_name);           /* open this directory */
    if (dirp == NULL) return;           /* dir could not be opened; forget it */
    while (TRUE) {
        dp = readdir(dirp);             /* read next directory entry */
        if (dp == NULL) {               /* NULL means we are done */
```

Figure 9-28. A recursive procedure that finds executable files on a UNIX system.

Executable Program Viruses (2)

```
    if (dirp == NULL) return;                          /* dir could not be opened; forget it */
    while (TRUE) {
        dp = readdir(dirp);                            /* read next directory entry */
        if (dp == NULL) {                              /* NULL means we are done */
        chdir ("..");                                  /* go back to parent directory */
        break;                                         /* exit loop */
    }
    if (dp->d_name[0] == '.') continue;                /* skip the . and .. directories */
    lstat(dp->d_name, &sbuf);                          /* is entry a symbolic link? */
    if (S_ISLNK(sbuf.st_mode)) continue;               /* skip symbolic links */
    if (chdir(dp->d_name) == 0) {                      /* if chdir succeeds, it must be a dir */
        search(".");                                   /* yes, enter and search it */
    } else {                                                  /* no (file), infect it */
        if (access(dp->d_name,X_OK) == 0) /* if executable, infect it */
            infect(dp->d_name);
    }
    closedir(dirp);                                    /* dir processed; close and return */
}
```

Figure 9-28. A recursive procedure that finds executable files on a UNIX system.

Executable Program Viruses (3)

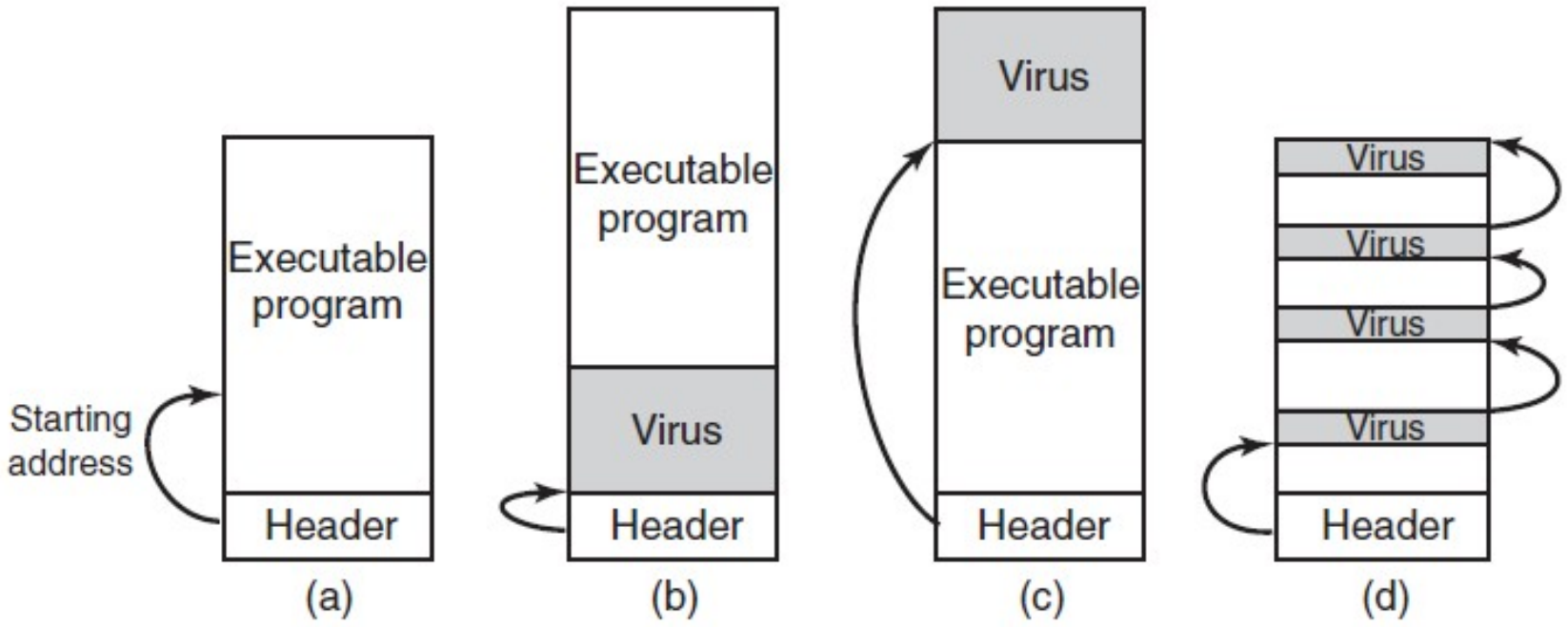


Figure 9-29. (a) An executable program. (b) With a virus at the front. (c) With a virus at the end. (d) With a virus spread over free space within the program.

Boot Sector Viruses

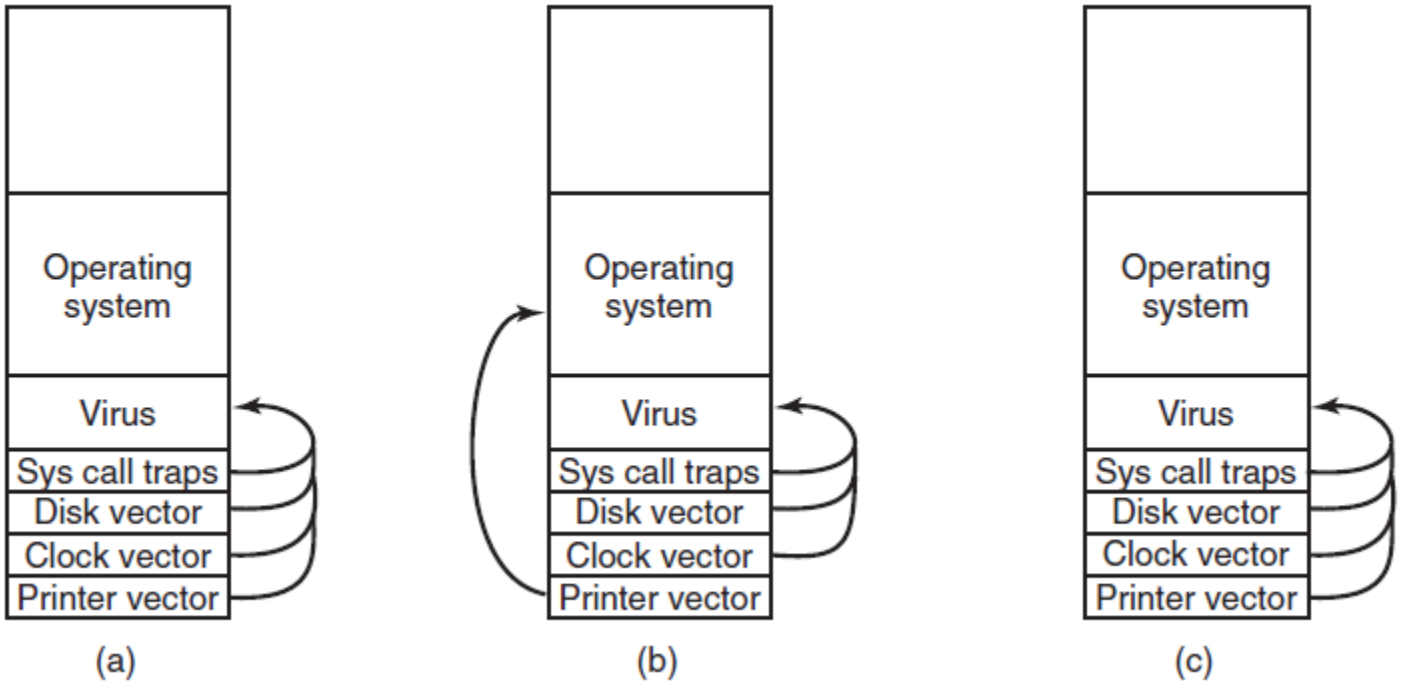


Figure 9-30. (a) After the virus has captured all the interrupt and trap vectors. (b) After the operating system has retaken the printer interrupt vector. (c) After the virus has noticed the loss of the printer interrupt vector and recaptured it.

Actions Taken by Spyware (1)

1. Change the browser's home page.
2. Modify the browser's list of favorite (bookmarked) pages.
3. Add new toolbars to the browser.
4. Change the user's default media player.
5. Change the user's default search engine.

Actions Taken by Spyware (2)

6. Add new icons to the Windows desktop.
7. Replace banner ads on Web pages with those the spyware picks.
8. Put ads in the standard Windows dialog boxes
9. Generate a continuous and unstoppable stream of pop-up ads.

Types of Rootkits (1)

Five kinds of rootkits – issue is where do they hide?

1. Firmware rootkit
2. Hypervisor rootkit
3. Kernel rootkit
4. Library rootkit
5. Application rootkit

Types of Rootkits (2)

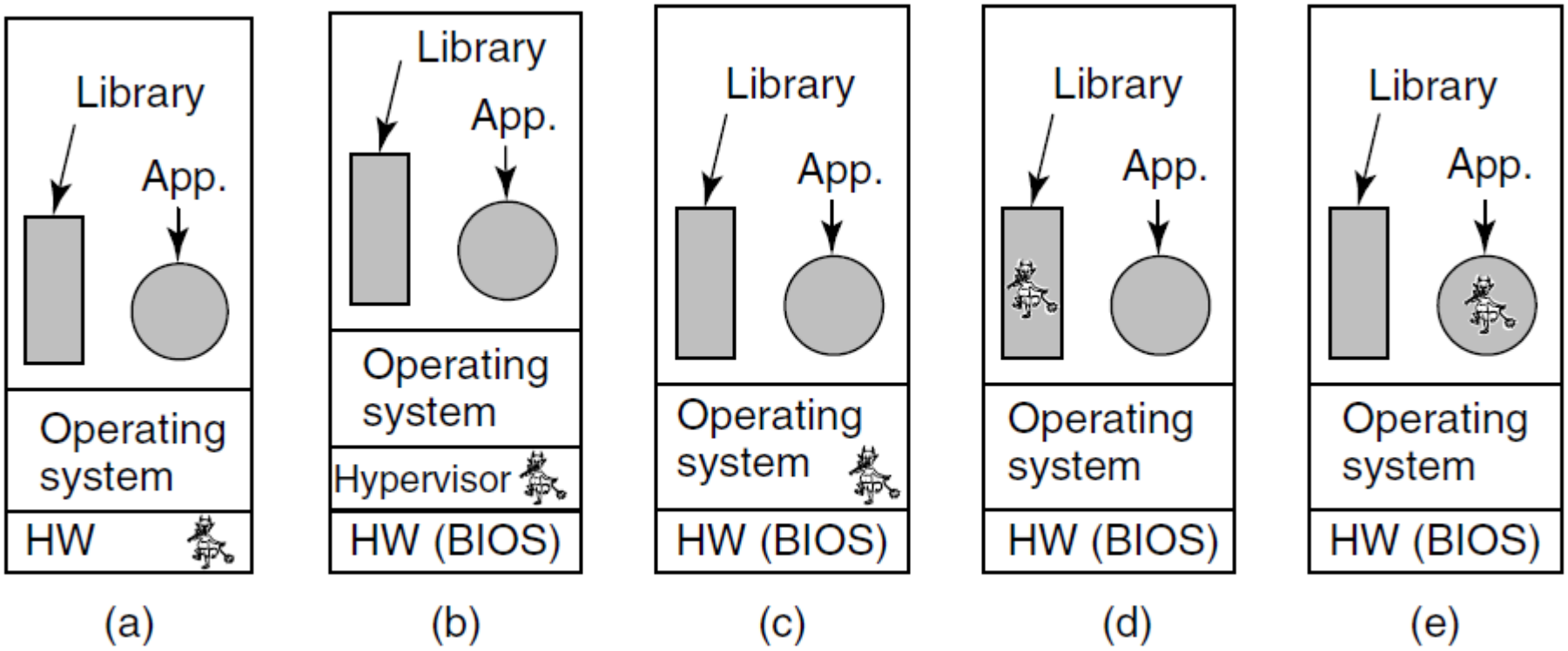


Figure 9-31. Five places a rootkit can hide.

Firewalls

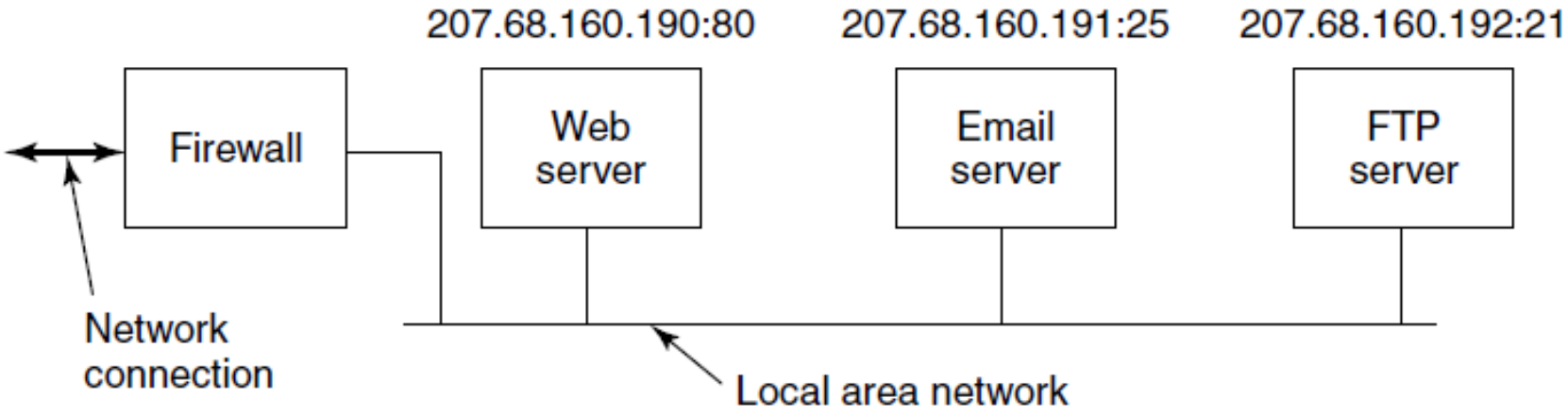


Figure 9-32. A simplified view of a hardware firewall protecting a LAN with three computers

Virus Scanners (1)

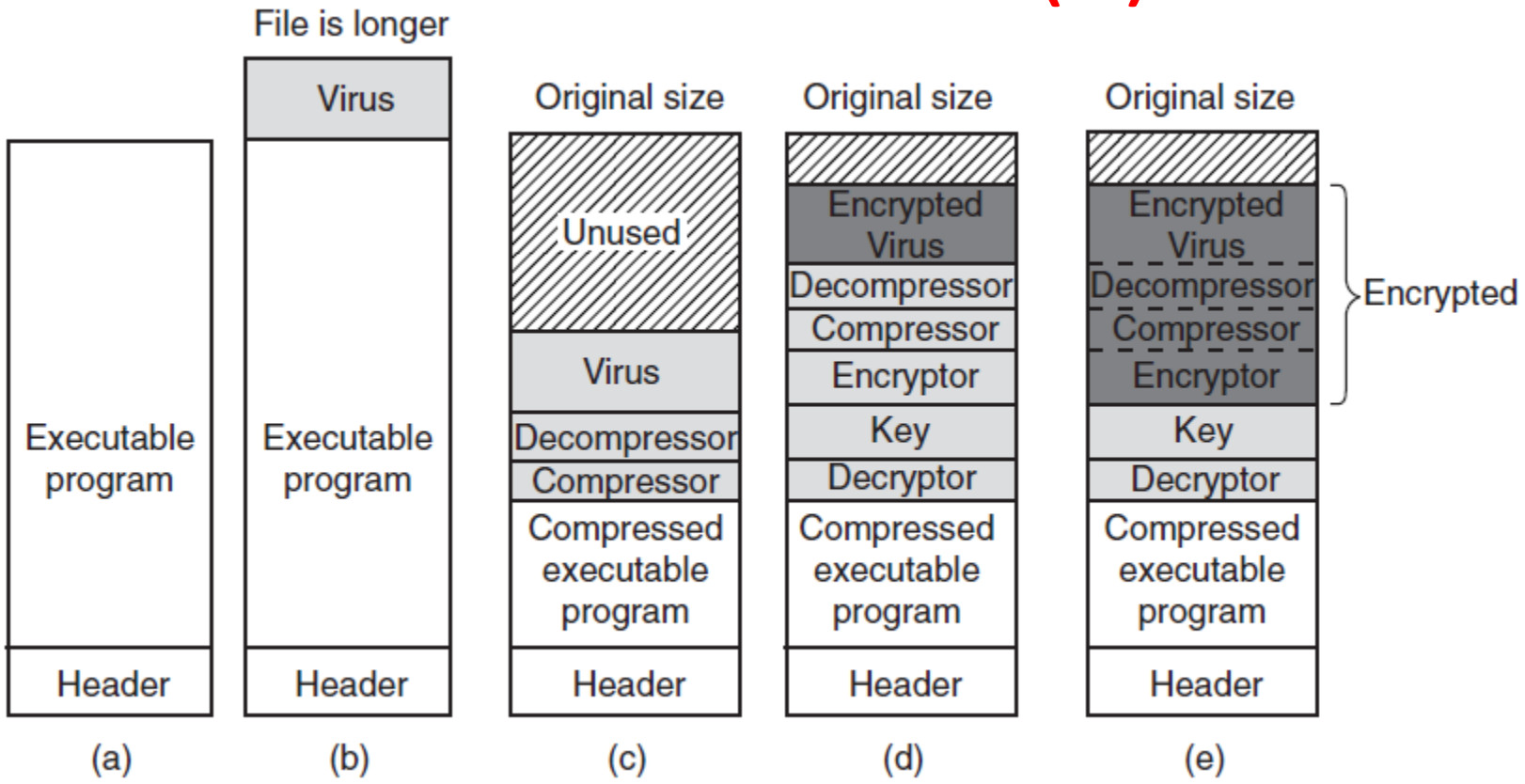


Figure 9-33. (a) A program. (b) An infected program. (c) A compressed infected program. (d) An encrypted virus.
(e) A compressed virus with encrypted compression code.

Virus Scanners (2)

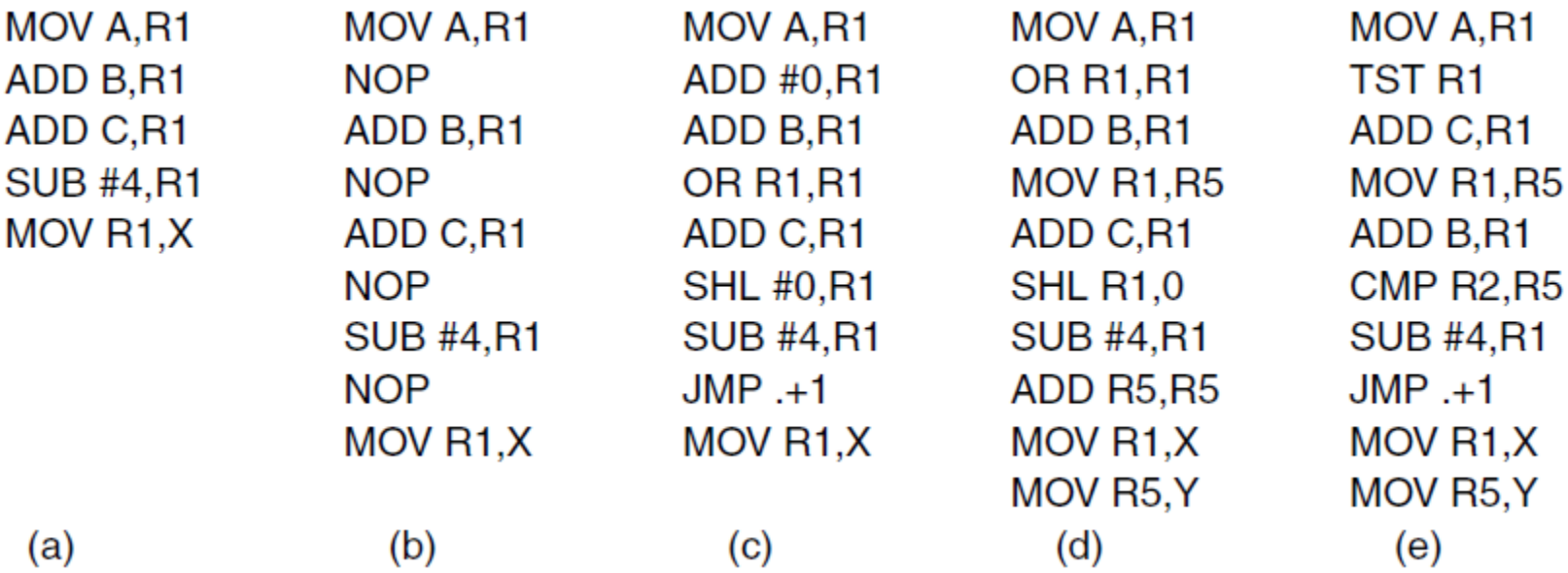


Figure 9-34. Examples of a polymorphic virus.

Code Signing

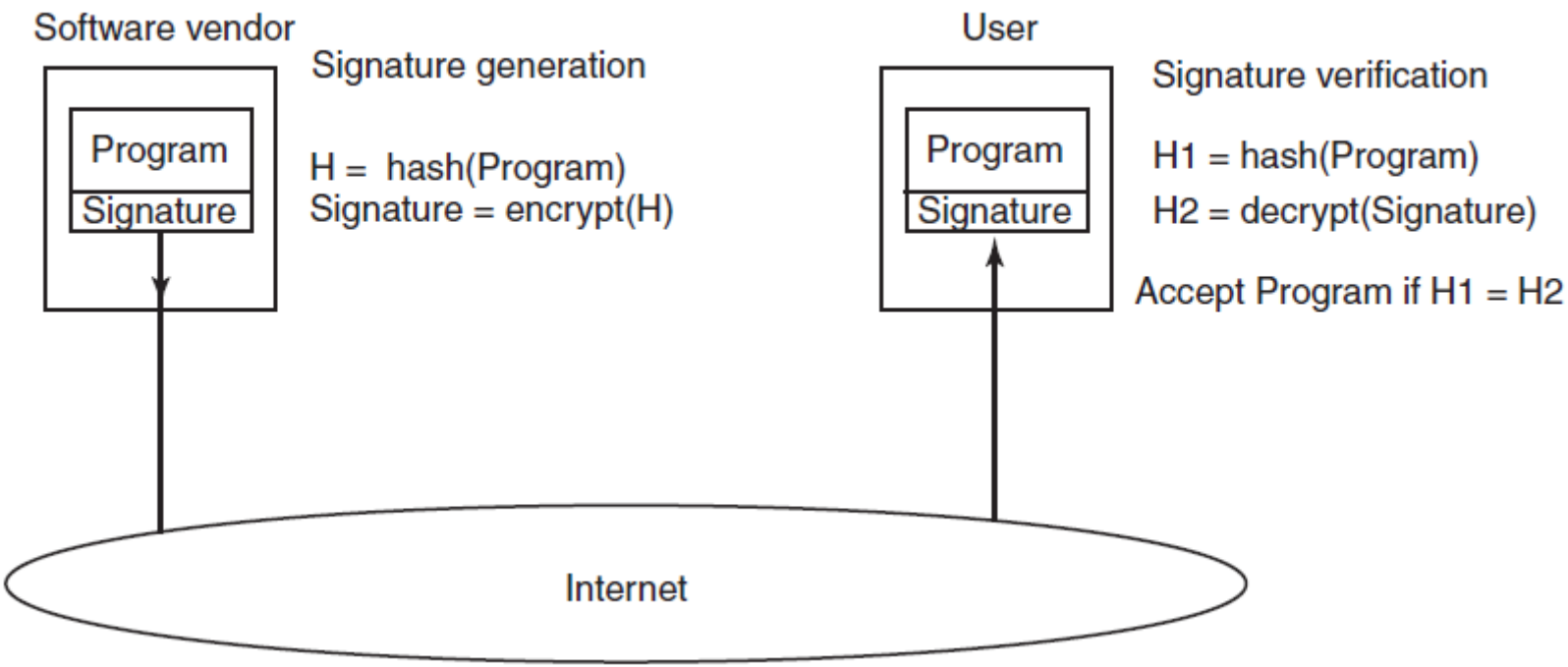


Figure 9-35. How code signing works.

Jailing

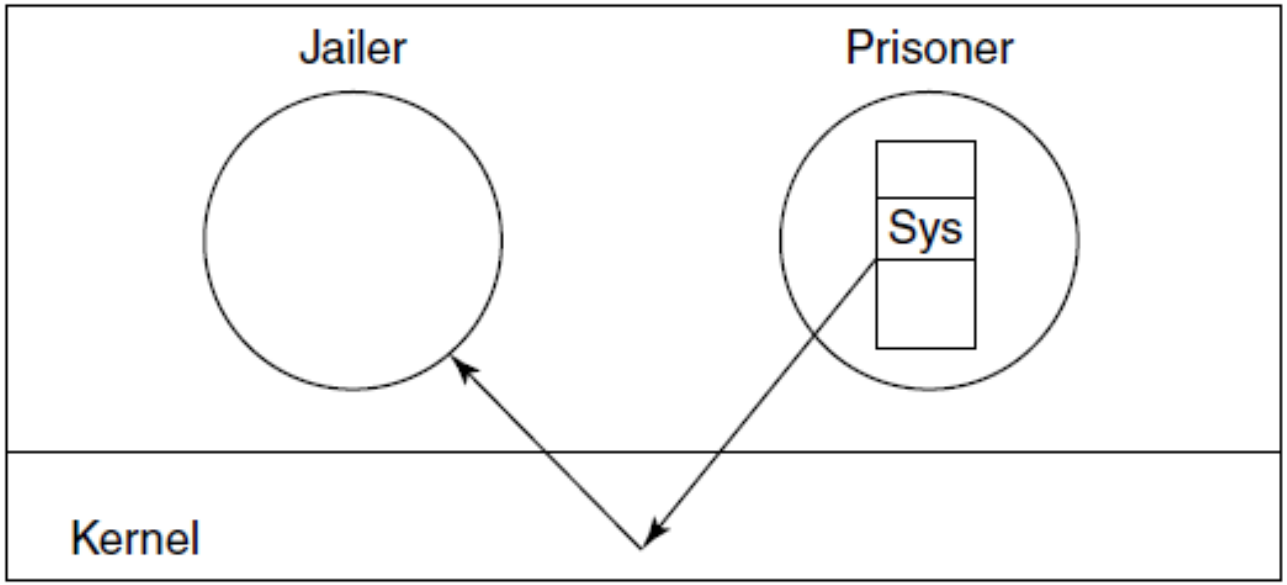


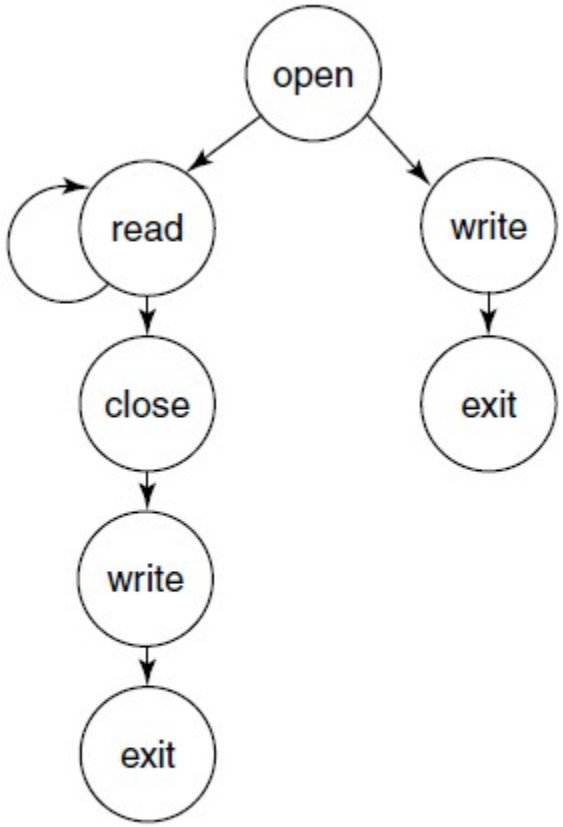
Figure 9-36. The operation of a jail.

Model-Based Intrusion Detection

```
int main(int argc *char argv[])
{
  int fd, n = 0;
  char buf[1];

  fd = open("data", 0);
  if (fd < 0) {
      printf("Bad data file\n");
      exit(1);
  } else {
      while (1) {
          read(fd, buf, 1);
          if (buf[0] == 0) {
            close(fd);
            printf("n = %d\n", n);
            exit(0);
          }
          n = n + 1;
      }
  }
}
```

(a)



(b)

Figure 9-37. (a) A program. (b) System call graph for (a).

Sandboxing

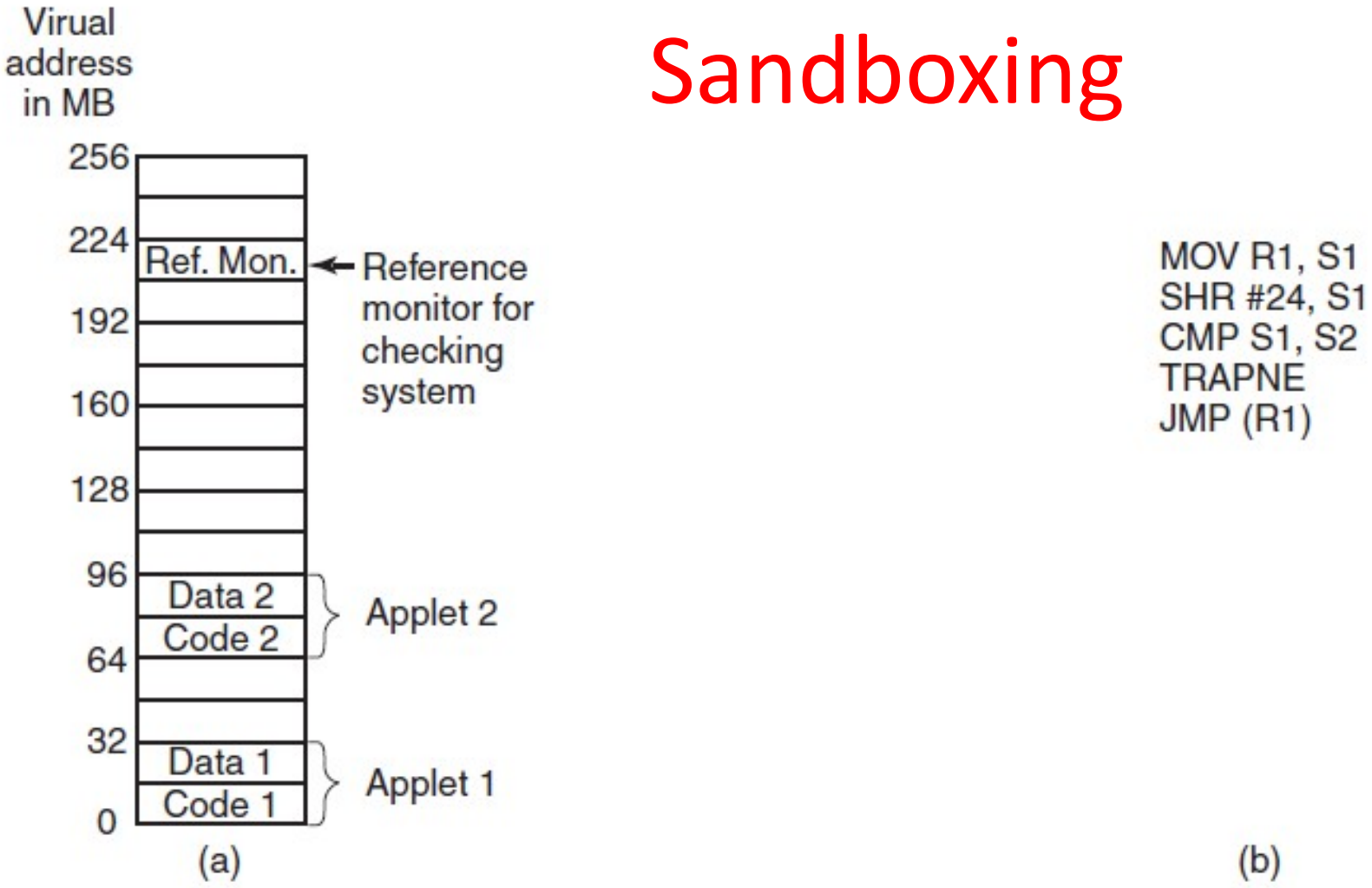


Figure 9-38. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

Interpretation

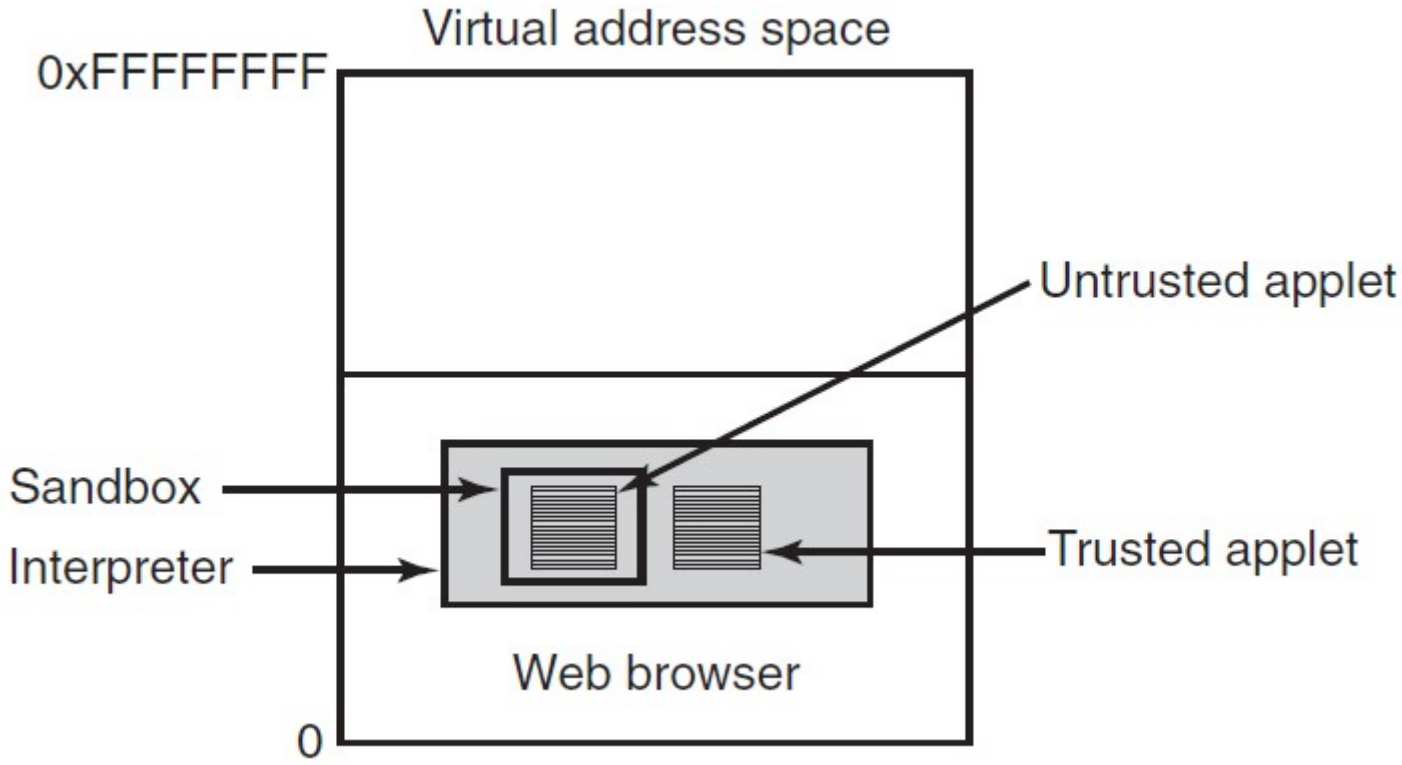


Figure 9-39. Applets can be interpreted by a Web browser.

Java Security (1)

Checks on applets include:

1. Does applet attempt to forge pointers?
2. Does it violate access restrictions on private-class members?
3. Does it try to use variable of one type as another?
4. Does it generate stack overflows or underflows?
5. Does it illegally convert variables of one type to another?

Java Security (2)

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

Figure 9-40. Some examples of protection that can be specified with JDK 1.2.

End

Chapter 9