

- There are four levels at which a system must be protected:
 1. **Physical** - The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
 2. **Human** - There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via ***social engineering***, which basically means fooling trustworthy people into accidentally breaching security.
 - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
 - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)
 - **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)
 3. **Operating System** - The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.
 4. **Network** - As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

15.2 Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

15.2.1 Trojan Horse

- A ***Trojan Horse*** is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with ***viruses***, (see below.)
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to

commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)

- **Spyware** is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of **covert channels**, in which surreptitious communications occur.) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

15.2.2 Trap Door

- A **Trap Door** is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

15.2.3 Logic Bomb

- A **Logic Bomb** is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the **Dead-Man Switch**, which is designed to check whether a certain person (e.g. the author) is logging in every day, and if they don't log in for a long time (presumably because they've been fired), then the logic bomb goes off and either opens up security holes or causes other problems.

15.2.4 Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:
 - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
 - (The problem could be avoided using strncpy, with a limit of 255 characters copied plus room for the null byte.)

```
#include
#define BUFFER_SIZE 256

int main( int argc, char * argv[ ] )
{
    char buffer[ BUFFER_SIZE ];

    if( argc < 2 )
        return -1;
    else {
        strcpy( buffer, argv[ 1 ] );
        return 0;
    }
}
```

Figure 15.2 - C program with buffer-overflow condition.

- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
 - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
 - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
 - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

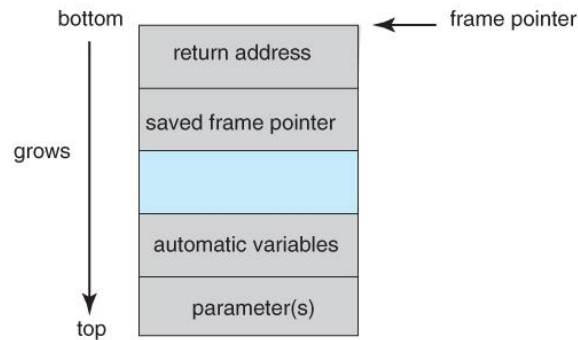


Figure 15.3 - The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec(/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. (Note, however, that neither the bad code or the padding can contain null bytes, which would terminate the strcpy.)
- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.

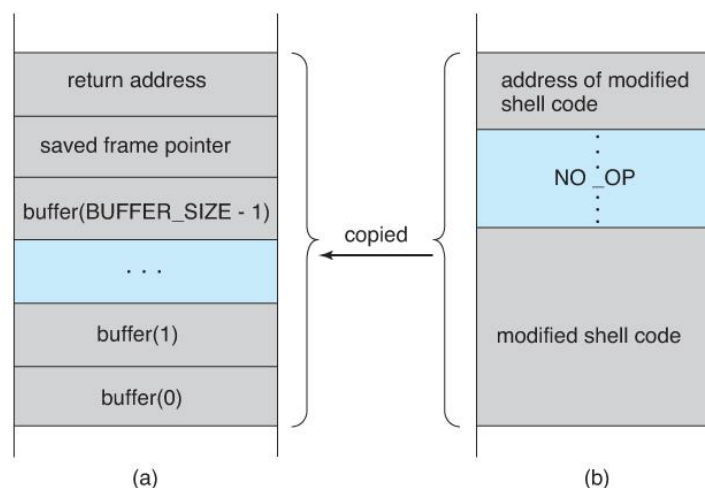


Figure 15.4 - Hypothetical stack frame for Figure 15.2, (a) before and (b) after.

- Unfortunately famous hacks such as the buffer overflow attack are well published and well known, and it doesn't take a lot of skill to follow the instructions and start attacking lots of systems until the law of averages eventually works out. (*Script Kiddies* are those hackers with only rudimentary skills of their own but the ability to copy the efforts of others.)
- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.
- (More details about stack-overflow attacks are available on-line from <http://www.insecure.org/stf/smashstack.txt>)

15.2.5 Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a *virus dropper*, usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure 15.5 shows typical operation of a boot sector virus:

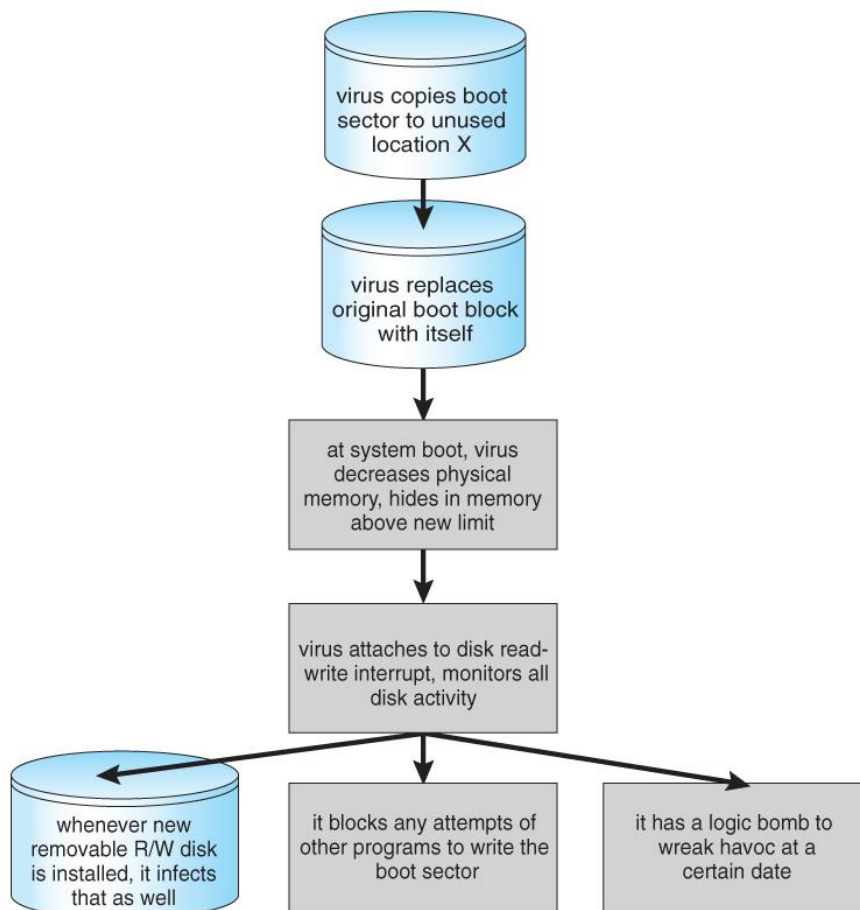


Figure 15.5 - A boot-sector computer virus.

- Some of the forms of viruses include:
 - **File** - A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed *parasitic*, because they do not leave any new files on the system, and the original program is still fully functional.

- **Boot** - A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as *memory viruses*, because in operation they reside in memory, and do not appear in the file system.
- **Macro** - These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
- **Source code** viruses look for source code and infect it in order to spread.
- **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their *signature*, by which virus checkers recognize them.
- **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
- **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read() system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
- **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
- **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
- **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a *keystroke logger*, which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a *monoculture*, in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

15.3 System and Network Threats

- Most of the threats described above are termed *program threats*, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

15.3.1 Worms

- A *worm* is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
 1. A small program called a *grappling hook*, which was deposited on the target system through one of three vulnerabilities, and
 2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

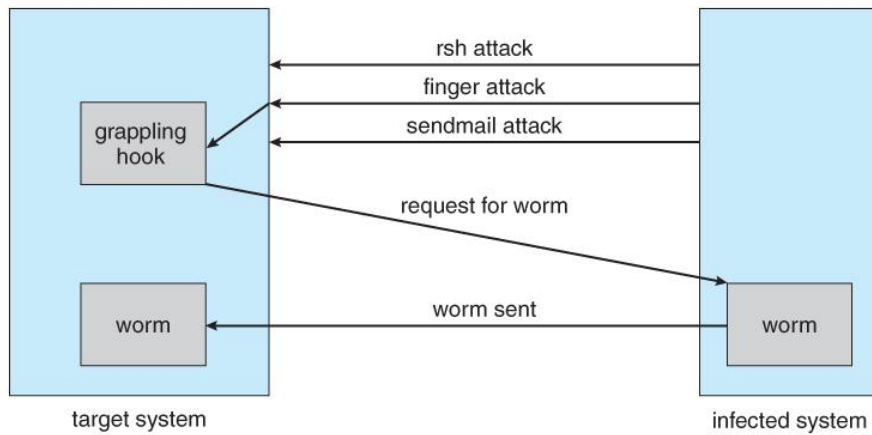


Figure 15.6 - The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:
 1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that **any** user (except root) on system A could access the same account on system B without providing a password.
 2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
 3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.
- Once in place, the worm undertook systematic attacks to discover user passwords:
 1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
 2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
 3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.
- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in

August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

15.3.2 Port Scanning

- **Port Scanning** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from **zombie systems**, i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.
- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are **nmap** (<http://www.insecure.org/nmap>) and **nessus** (<http://www.nessus.org>). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

15.3.3 Denial of Service

- **Denial of Service (DOS)** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
 - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
 - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
 - CS students given their first programming assignment involving `fork()` often quickly fill up process tables or otherwise completely consume system resources. :-)
 - (Please use `ipcs` and `ipcrm` when working on the inter-process communications assignment !)