CYBERGUARD CORPORATION

Buffer Overrun Attacks

Paul A. Henry MCP+I, MCSE, CISSP
CyberGuard Corp.

The threat was first seen widely in 1988 and it is still an active attack methodology in 2000. A buffer overrun attack was one of the mechanisms reportedly utilized to deploy the malicious agents used on the Solaris-based servers in the recent DDoS attacks. To see just how common buffer overrun attacks are, all you need to do is search the archives of CERT, CIAC, BugTraq, RootShell or X Force for the term "buffer overrun." The results are disturbing.

The problem of buffer overruns in C programs has been recognized since the early '70s as one possible consequence of the C language-data integrity model. The C programming language does not automatically support bounds-checking internally when initializing, copying or moving data between or into variables.

One of the first widely publicized buffer overrun attacks occurred in 1988 as part of the infamous Internet worm incident:

> A vulnerability that was exploited by the famous Internet worm involved a buffer overrun in "fingered." Using the "gets() call" function, fingered would read a line of information. The buffer allocated for the string was 512 bytes long, but fingered did not check to see if the read was greater than 512 bytes before exiting the subroutine. If the line of information read was greater than 512 bytes, the data was written over the subroutine's stack frame return address location. The stack could effectively be rewritten by the intruder to create a new shell and allow the intruder to execute commands from root.
>
> The Internet worm wrote 536 bytes to the "gets() call" function. The 24 bytes overrun contained Vax machine language instructions that, upon return from the main() call, tried to execute a shell by calling execve("/bin/sh",0,0).

More recently, it has been reported that in the devastating Stacheldraht DDoS attacks, buffer flows played a large part in the installation of the malicious agents used to facilitate the attacks:

"Stacheldraht agents were originally found in binary form on a number of Solaris 2.x systems, which were identified as having been compromised by exploitation of buffer overrun bugs in the RPC services "statd," "cmsd" and "ttdbserverd." They have been witnessed "in the wild" as late as the writing of this analysis."

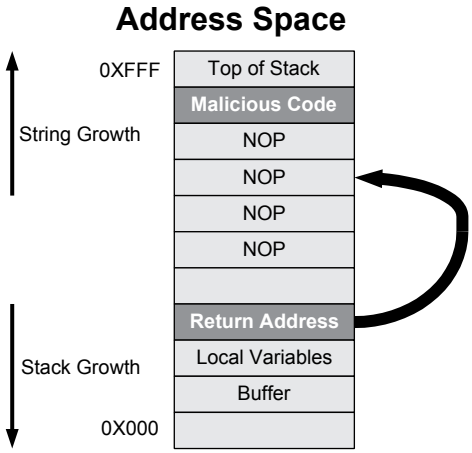
Despite the fact that fixing individual buffer overrun vulnerabilities is fairly simple, buffer overrun attacks continue to be a common problem in network security today. New programs are being developed with more care, but are often still developed using unsafe languages such as C, where simple errors can leave serious vulnerabilities. Many legacy systems are still running today that utilize thousands of lines of code with privileged daemons that contain numerous software errors.

Buffer Overrun Attack Methodology

Buffer overrun attacks exploit a lack of bounds-checking on the size of input being stored in a buffer array. By writing data past the end of an allocated array, the malicious user can make arbitrary changes to the program state stored adjacent to the array. Many C programs have buffer overrun vulnerabilities, both because the C language lacks array bounds-checking, and because the culture of C programmers encourages a performance-oriented style that avoids error-checking where possible to enhance performance. For instance, many of the standard C library functions such as "gets" and strcpy (???) do not perform any bounds-checking by default.

There are two definitive types of buffer overrun attacks:
1. Stack Buffer Over Flow 2. Heap Buffer Overrun

The Stack Buffer Overrun



The most common form of buffer overrun exploitation is to attack buffers allocated on the stack. Stack buffer overrun attacks are designed to achieve two mutually dependent goals:

1. Insert Malicious Code:

The malicious user provides an input string that is actually executable binary code that is native to the machine being attacked. Typically this code is simple, and does something similar to exec("sh") to produce a root shell.

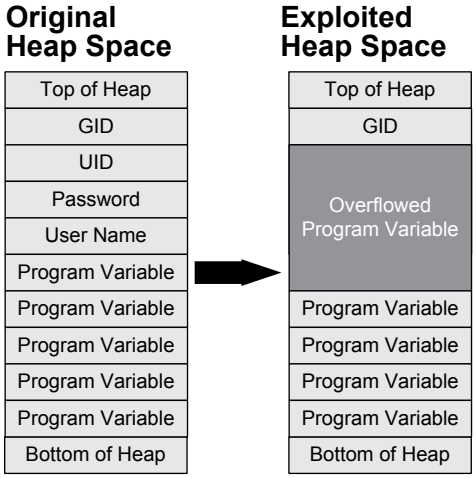
2. Change the Return Address :

There is a stack frame for a currently active function above the buffer on the stack that is being attacked. The stack buffer overrun changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps to the attack code. In many cases the malicious code is preceded by a block of NOP instructions which reduces the accuracy required to guess the exact return address for the malicious code. If the return address lands anywhere within the block of NOP instructions the malicious code will be executed.

The programs that are attacked using this technique are almost always privileged daemons– programs that run at root to perform some specific service. The injected attack code is usually a short sequence of instructions that spawns a shell to give the malicious user a shell with root privileges. If the program input comes from a network connection, it may allow any user anywhere on the network the ability to become root on the targeted host.

The Heap Buffer Overrun :

Memory that is dynamically allocated by an application for variable storage is called the heap. In the typical heap buffer overrun attack, variables such as passwords, file names and a saved uid in the heap are overwritten by the malicious user. Heap overrun attacks are not as common as stack buffer overrun attacks but they can be effective in providing unauthorized privileged access for the intruder.



An early heap buffer overrun vulnerability was found in BSDI crontab in 1996. This heap buffer overrun involved passing a long file name which overran its buffer in the heap. The overrunning data wrote over the fields in the heap which held the user's user name, password, uid, gid, etc. When used maliciously, one could easily change the privileges associated with the user / application by changing the uid/gid to 0.

Principle differences

While there are many similarities to the methodologies involved in the stack buffer and heap buffer overruns, there is one principle differentiator. In the stack buffer overrun, machine level commands are placed on the stack and are subsequently executed by overwriting the return address on the stack. While in a heap buffer overrun, dynamically stored application variables are overwritten in effort to increase the level of system privilege.

Defending your network :

A prudent security policy should include risk mitigation of protocol header buffer overruns. The static packet filter, dynamic (stateful) packet filter and many circuit gateway-based firewalls simply do not provide a mechanism to prevent protocol header-based buffer overrun attacks on your critical servers behind the firewall. Simply put, with these protection methodologies the malicious packet is allowed to pass to the critical server unchallenged. Currently only the strong application proxy provides the level of inspection necessary to verify all protocol header lengths are in compliance with RFC requirements to mitigate this broadly used attack methodology.

References :

"The stacheldraht distributed denial-of-service attack tool"
David Dittrich University of Washington

"How To Write Buffer Overflows"
by mudge@l0pht.com 10/20/95
http://www.pmf.ukim.edu.mk/~damjanev/sec/buffer.txt
http://www.insecure.org/stf/mudge_buffer_overrun_tutorial.html

"Smashing The Stack For Fun And Profit"
Phrack 49 Volume Seven, Issue Forty-Nine, File 14 of 16
By Aleph One
http://www.pmf.ukim.edu.mk/~damjanev/sec/stack-smash.txt