Buffer Overflows

Eugene Tay Kostas Zafiris Loizos Pallaris
Dirk Rösler Andrew McDonald Brendan Barnes
Philippe Lottmann Dayo Adewoye

December 1999

Abstract

The buffer overflow problem has plagued programmers for many years. This common programming mistake can lead to problems such as program failure, allowing a user to increase his access rights or allowing an outside attacker to break into a computer system.

We describe this problem, look at its history, including specific examples and also some possible solutions.

1 Introduction

According to a recent paper [1] the computer vulnerability of the decade is not the Y2K bug, but a security weakness known as buffer overflow.

A buffer overflow occurs when a program tries to write more data into a space in memory than can fit. This can result in corruption to the running program, which may cause program failure. However, by carefully crafting the contents of the data that overflows the buffer an attacker may be able to break into a system or increase his system privileges.

2 Background and History

Buffer overflow attacks are nothing new. Besides of representing one of the most prominent and first malicious software attacks, the infamous Internet Worm, written and released by Robert T. Morris, Jr. in 1988, employed the first well-known buffer overflow attack. The Worm, which infected thousands of systems on the Internet, exploited a buffer overflow in the finger daemon to gain access to VAX machines running BSD Unix. Even before the Worm's release, buffer overflow attacks may have been known in some circles and there is anecdotal evidence of buffer overflow attacks dating back to the 1960's (an example given was an OS/360 problem).

Even so, prior to 1997, buffer overflow vulnerabilities were only periodically discovered and never in such numbers as to attract too much attention. In recent years, however, buffer overflows have become the most commonly discovered class of vulnerability the most widely exploited, cynically labelling them "the year's most fashionable attack". To illustrate this: 24 out of 44 CERT advisories issued during 1997 involved some type of buffer overflow attack. Buffer overflow vulnerabilities therefore represent a class of security problem of great importance

to computer users, whether they be programmers, system administrators or users.

Since the Morris Worm first illustrated the so-called method of stack smashing literally thousands of buffer overflow vulnerabilities have been discovered in security-sensitive code, and continue to be discovered to this day. They appear to be the most common problems reported, with denial-of-service problems a distant second, bearing in mind that the latter is also a possible result of an stack-smashing attack.

Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors, before releasing the software, if the vendors had performed elementary testing or code reviews along the way, i.e. had it been audited for vulnerabilities. Some critics ultimately blame software vendors such as Sun Microsystems or Microsoft for failing to apply appropriate, admittedly expensive, quality control particularly when writing in a language such as C and its derivatives that are considered to contain error-prone idioms. For instance, many standard C/C++ library functions such as `gets()` and `strcpy()` do not do bounds checking. They also cite that programmers have let down their guard against a long-recognized hazard and a culture that favours performance over correctness and avoids error checking, a typical dilemma in computer security as higher security often is a trade-off in system performance. In the 1960s and '70s the solutions typically either used hardware or were implemented within the program itself. But when it was felt that it made the program go too slow and a lot of programs went out there without buffer checks.

The base problem is that, while individual buffer overflow vulnerabilities are simple to patch, the vulnerabilities overall are plentiful. Millions of lines of legacy code are still running as privileged daemons (root) that contain numerous software errors. New programs may be built with more care, but are often still written in languages such as C, where simple errors can leave serious vulnerabilities. To rectify this all of these programs that run as root either because they are setuid root, or because root runs them would need re-compiling, an almost impossible task.

Often the attacks are based on reverse-engineering the attacked program to determine the exact offset from the buffer to the return address in the stack frame, and the offset from the return address to the injected attack code. However, the reverse-engineering has been reduced to a cook book. The publication of "Smashing The Stack For Fun And Profit" [3] written by "Aleph One" in the hacking e-zine Phrack in November 1996 represents a milestone in the history of buffer overflow exploitation attacks. Detailed descriptions like these have made building buffer-overflow exploits easy as the only remaining work for a would-be attacker is to find a poorly protected buffer in a privileged program, and construct an exploit. Hundreds of such exploits have been reported in recent years. The emergence of the Internet has greatly increased the opportunities for an attack as more and more machines get connected. On one hand this means that these "recipes for disaster" have become available for everyone and on the other it also increased the odds of finding machines or systems which are inadequately protected, be it due to ignorance or simple naivete.

The continued success of these attacks is also due to the "patchy" way we protect against such attacks. In the life cycle of a buffer overflow attack a malicious user discovers the vulnerability in a highly privileged program and

someone else implements a patch to defend against that particular attack, on that privileged program. Fixes to buffer overflow attacks attempt to solve the problem at the source (the vulnerable program) instead of at the destination (the stack being overflowed). The implication for system administrators is that they must keep up with system patches as closely as possible. When a stack smashing vulnerability is announced, it is very shortly followed by "script kids" trying out their new toy on various hosts to see who has and has not applied the patch. This has the effect of turning patches and upgrades from a casual "when it's convenient" task into an urgent task for system administrators.

The experience to date by no means implies that Unix-based systems have a monopoly on this class of attack or even that they are more susceptible. Rather, due to the relative complexity of the Win32 API compared to Unix system calls, relatively few people have a sufficient understanding of the intricacies of the Windows API at an assembly level to exploit a buffer overflow in a controlled fashion. Thus, while it is trivial to exploit a buffer overflow so as to make a Windows program or service crash which in some cases would at least accomplish a denial of service, it is not so trivial to exploit a buffer overflow in order to attain access and/or increased privileges on a Windows system.

It is a simply a matter of time before buffer overflows are widely exploited on Windows systems and possibly other more recent systems. After all, buffer overflows existed in Unix-like operating systems for many years before they were well understood and exploited. Already, an attempt at creating a generalized framework for identifying and exploiting buffer overflows in Windows operating systems exists. But it seems that for now, many buffer overflow vulnerabilities in Windows are considered "purely theoretical". As history tells us, however, in time more and more people will be making the theoretical practical. In this instance a recent edition of Phrack includes an article on creating such buffer overflows. [5]

2.1 Terminology

A buffer is a data structure, used for the temporary storage of information, containing multiple instances of the same data type that are consecutively stored in a contiguous memory block. The most obvious implementation of a buffer is an array, such as a character array (string).

Overflowing a buffer involves attempting to store more data items than the buffer can accommodate. In the absence of an appropriate checking mechanism (bounds checking) the extra data can overflow to memory locations adjacent to the buffer and consequently overwrite whatever might be stored there.

2.2 Process Memory Structure

A running process utilizes a contiguous block of memory, which is structured in a predefined way. There are three distinct memory areas: text region, data region and stack. The code, as well as any read only data are stored in the text region. This is a read only area residing at the lowest end of the memory block utilized by the process. Any attempts to write in the text region result to a segmentation violation. Initialized and un-initialized program variables are contained in the data region, which resides directly above the text region. This area can be expanded at run time to accommodate any additional memory

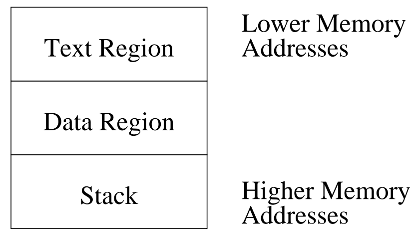

Figure 1: Process Memory Regions

requirements. The temporary memory requirements of any function calls by the process are satisfied by the stack.

2.3 Function Call Mechanism

A function is a piece of code performing a particular operation. One of the principal motivators for using functions is code reuse. The function usually knows little or nothing about the calling process and obtains its data through a clearly defined interface rather than by reference to the variable of the calling process. This interface regulates the flow of data to the function called via the specification of function parameters and a function result to the calling process.

When a function call is made, memory needs to be set aside for use by the called function. This includes memory for the storage of the function parameters, as well as for the storage of the local variables of the function called. After the function completes its job, the memory set aside for the function call is no longer necessary and is therefore released. However, function calls can be nested, i.e. a called function can make function calls of its own. This means that an additional amount of memory will need to be set aside for the function call at the inner level, in addition to the amount of memory set aside for the function call at the outer level. This characteristic of function calls makes a stack the ideal data structure for the handling of their memory needs. Memory areas for nested function calls can be set aside, accessed and released in a Last In First Out (LIFO) fashion.

The memory to be set aside for a function call does not only store the parameters passed to the function and the local variables of the function. A number of other important variables need to be stored for every function invocation, and these are also stored on the stack.

A function call involves a change in the execution flow of the program. This means that execution of the code of the calling process will stop at the point of the function call and execution will continue at the start of the code of the function called. After the called function has finished, execution will need to continue within the code of the calling process at the point just after the function call. This makes it necessary to keep track of the so called return address for the function call, i.e. the point where execution will resume after the function call. Therefore, the return address needs to be stored on the stack for every function invocation.

In the case of a single function call, the local variables of the function called can be referenced relative to the stack pointer. This is no longer the case when nested function calls have taken place. As a result, a pointer serving as a local

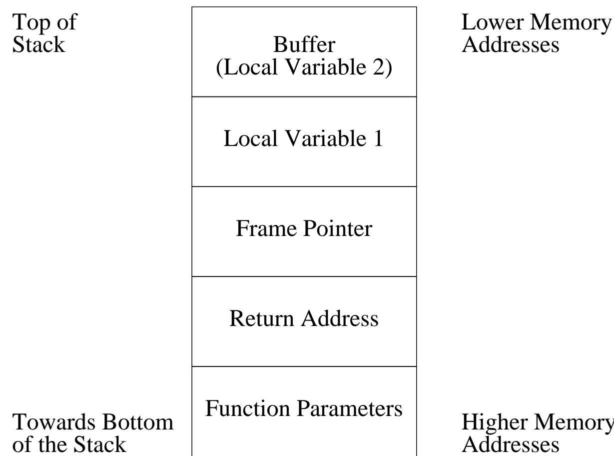

Figure 2: Stack Architecture

point of reference for each function call will also need to be saved on the stack. This will be used to allow for relative addressing of the local variables of each one of the nested function calls. This pointer is called the frame pointer (FP).

Whenever a function call is made, the parameters of the function called are the first data to be put on the stack. These are stored on the stack, in reverse order by the calling process. These are followed by the return address and the frame pointer, which are placed on the stack by the calling process. The local variables, defined by the function called are the last to be put on the stack. (See Figure 2).

2.4 Buffer Overflow Mechanism

In a stack implementation where the stack grows downwards, the local variables of the function called are in a lower memory address than the rest of the information stored on the stack for the particular function invocation. Hence, overflowing a buffer in the local variable area of the memory set aside for a specific function invocation will cause overwriting of higher memory locations. As a result one or more of the other variables stored on the stack for this particular function invocation may become corrupted. These can be other local variables, the frame pointer or even the return address.

Overwriting crucial variables stored on the stack in an arbitrary way will most probably cause a process to fail. This for example will happen if a corrupted return address falls outside the code space of the calling process. The above scenario can form the basis of a Denial of Service attack.

However, an attacker can afford to be more precise than that. By targeting the return address with specific data he can force execution flow to be transferred to the address of his choice. In this scenario, the code that will meet the attacker's objectives will need to pre-exist in an accessible memory location known to the attacker.

The easiest way to ensure this is by including the malicious code in the data used to overflow the buffer and forcing the return address to point back into the buffer, to the beginning of the malicious code.

Another approach might be to store the malicious code within an environmental variable, and make the return address for the attacked function call point to the memory area holding this environmental variable.

2.5 Attack Objectives

The most generic piece of code an attacker might wish to execute is a system shell, which would allow him to directly interact with the attacked computer system. However not all system shells are created equal. Like all processes a shell has a privilege level, which is consulted by the operating system so as to enforce proper access control to the resources of the computer system. It is obvious that it should not be possible for arbitrary called processes to run with higher privilege levels than the process or user that initiated them, otherwise the access control system would not work. However, a number of trusted programs with known functionality are allowed to do exactly that, because otherwise they would not be able to perform their jobs. In a UNIX system, programs that are marked SUID can run with higher privileges than their initiators.

If an attacker is able to cause a program with higher privileges than himself to run code forcing it to open a system shell, the attacker would be able to assume the privileges of this program and exploit them through this spawned system shell. As a matter of fact, some processes have root privileges, which means they have complete control of the computer system. Attacking such processes through buffer overflow or otherwise is an obvious way to completely compromise a computer system. It is therefore not surprising that processes with root privileges are actively targeted by attackers.

3 Examples

3.1 The Internet Worm

3.1.1 Introduction

The 'worm', written by Robert T. Morris, Jr, was unleashed onto the Internet on the 2nd of November 1988. Within hours of release, it infected thousands of computers throughout the United States of America. It consisted of a small 99 line bootstrap program, written in C language, that infected machines running 4.2 or 4.3 BSD Unix or derivatives like SunOS. Exploiting vulnerabilities in these OS's to bypass login authentication it self-propagated from one machine to another.

3.1.2 Chronology

It is generally thought that the Worm spread far faster than its creator had intended. This chronology highlights the speed at which it infected machines and ultimately rendered them unusable.

11/2	1800	First infection recorded at the MIT intelligence Laboratory (*prep.ai.mit.edu*).
11/2	1824	First known West Coast infection: *rand.org*.
11/2	1954	University of Maryland (*mimsy.umd.edu*) attacked through its finger server.
11/2	2049	University of Utah (*cs.utah.edu*) is attacked.
11/2	2206	cs.utah.edu reaches maximum runnable processes. Machine unusable.
11/2	2328	NASA posts warning about mysterious virus that has hit UC Berkeley.
11/3	0254	first fix for sendmail posted.
11/3	1918	first fix for finger fixed.
11/4	1236	MIT/Berkeley announce they have fully disassembled the Worm. It is fully decompiled and commented seven days later.
11/8		National Computer Security Center meeting centered around the Worm.

3.1.3 The Worm

The worm used different strategies to pass unnoticed and run processes on one machine until it can find suitable means for propagation. Upon arrival onto a new host the worm is just a 99-line text and a compiling command that will build the worm executable using local libraries found on site. Once it is compiled it assumes a seemingly harmless name and extension and renames itself from time to times to reduce the chances of being discovered. It switches off any programs susceptible to locate, intercept or hamper it and deletes any log where its actions would have been entered. Once it considers the infected machine secure, it looks inside the user account to find addresses of network hosts and clients to attack. The worm is given a limited time of life to reduce chances of being discovered after infecting other computers. It allows re-infecting computers permanently, including the one where it died.

The Worm used three methods to attach to and infect remote machines. The Worm prefers to exploit: *rsh/rexec*, a buffer overflow in the finger program and a bug in the sendmail program. These exploits, in particular the buffer overflow bug, are described below.

1. The first means of attack are via *rsh* and *rexec*. *rexec* is exploited through the likelihood that the machine under attack will have an account for a local user where both have the same password, thus allowing authenticated login. *rsh* is exploited through the likelihood that a local account will be included as a host in the remote machines *rsh* permissions files. Taking advantage of both these exploits requires the worm to crack the passwords of local accounts. This is carried out by native password cracking routines that first try simple 'trivial' passwords, but then move on to a full dictionary attack. Relevant information in the users `.forward` and `.rhosts` file is then used to select potential remote hosts for attack.

2. Sendmail is a character-oriented mail service for Unix systems, which can sent messages to mail boxes and processes. It contains a bug if compiled in debug mode (as were most versions at that time), which allows a command

line to be entered instead of a recipient's address. The worm sends a message which contains code and a command line to direct and compile the code. The local host (from where the message was sent) then waits for the executable built on the remote host to establish a connection and complete the infection process.

3. The *finger* service is intended to provide remote users with information on local user accounts and system details. The effected version of *finger* contained a buffer overflow that led to system security being compromised. On receipt of a *finger* request, the *finger* program is run where the variables set in the request are used as arguments. The C library routine `gets()`, which is pretty old and has been written for convenience more than security, is used to read the request, but unfortunately it carries out no bounds checking for an overflow of the servers 512 byte request buffer on the stack. Therefore, the worm provides a 536 byte request which contains machine code asking the system to run the command interpreter *sh* and the extra 24 bytes write over the main routine's stack frame. On exit of the main stack routine, the calling functions program counter is supposed to be restored from the stack, but the worm writes over this with one that points to the VAX code in the request buffer. As a result, the program jumps to the worms code and runs the command interpreter which is used to enter it's 99-lines of 'boot-strap' code. From here, the code is compiled into an executable as normal and the infection process begins.

The C code below is the `worm_oflow.c` routine that allows it to exploit the finger bug and penetrate into a remote machine.

```
static try_finger(host, fd1, fd2)
     struct hst *host;
     int *fd1, *fd2;
{
    int i, j, l12, l16, s;
    struct sockaddr_in sin;
    char unused[492];
    int l552, l556, l560, l564, l568;
    char buf[536];
    int (*save_sighand)();
```

(The section above is the initialisation part of the routine, preparing variables and sockets.)

```
    save_sighand = signal(SIGALRM, justreturn);

    for (i = 0; i < 6; i++) {
        if (host->o48[i] == 0)
            continue;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0)
            continue;
        bzero(&sin, sizeof(sin));
```

```
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = host->o48[i];
        sin.sin_port = IPPORT_FINGER;
```

(The section above sets up the socket's port and address in preparation for the buffer overflow)

```
        alarm(10);
        if (connect(s, &sin, sizeof(sin)) < 0) {
            alarm(0);
            close(s);
            continue;
        }
        alarm(0);
        break;
    }
```

(Checks if connection to another computer has been made)

```
    if (i >= 6)
        return 0;
    for(i = 0; i < 536; i++)
        buf[i] = '\0';
    for(i = 0; i < 400; i++)
        buf[i] = 1;
    for(j = 0; j < 28; j++)
        buf[i+j] = "\335\217/sh\0\335\217/bin\320^Z\335\0\335\0"
"\335Z\335\003\320^\\\274;\344\371\344\342\241\256\343\350\357"
"\256\362\351"[j];
```

(This is the most interesting section. This is where the program actually inserts the contents of the buffer overflow, and enters the command to launch sh, the command interpreter. The numbers are command codes.)

```
    l556 = 0x7fffe9fc;
    l560 = 0x7fffe8a8;
    l564 = 0x7fffe8bc;
    l568 = 0x28000000;
    l552 = 0x0001c020;

#ifdef sun
    l556 = byte_swap(l556);
    l560 = byte_swap(l560);
    l564 = byte_swap(l564);
    l568 = byte_swap(l568);
    l552 = byte_swap(l552);
#endif sun

write(s, buf, sizeof(buf)); write(s, XS("\n"), 1);
    sleep(5);
```

(The part above rewrites the stack frame. For Sun machines words need to be reversed. But this routine doesn't work for sun machines anyway, which just dump the core).

```
    if (test_connection(s, s, 10)) {
        *fd1 = s;
        *fd2 = s;
        return 1;
    }
    close(s);
    return 0;
}
```

(Tests if the overflow has been done correctly and closes down the routine)

If none of these tactics work the Worm drops the case and switches to another target. If one of them works, the bootstrap code is sent, a mirror of it is created so that the first worm can exit and continue its program. The new program launches, hides and start running the necessary processes to infect other computers. It can in that way affect several machines and then terminate.

3.2 Microsoft Exchange's LDAP exploitation

Microsoft Exchange Server version 5.5 supports LDAP v3. The Lightweight Directory Access Protocol (LDAP), described in RFC 1777, is an industry-standard protocol used to perform directory operations such as Read, Search, Add, and Remove on a directory database. Out of the four operations, Exchange only supports the Read function as specified in the RFC.

Microsoft Exchange Server incorporates the LDAP protocol into its directory service primarily to allow interoperability. Third party LDAP compliant clients are now able to perform read operation on the Exchange's directory database.

On 15 March 1999, a buffer overflow exploit on Microsoft Exchange's LDAP service was discovered. This exploit made use of a malformed bind request to overflow the buffer during the LDAP binding process. LDAP binding simply means logging in or authenticating to a directory, and it consists of sending a username, password, and a binding method. There are two methods in which to use this vulnerability against an Exchange Server. The first consists of sending a particular type of invalid LDAP bind packet that could cause an overflow to occur. This potentially crashes the LDAP service causing a Denial of Service. The second uses a large malformed LDAP bind packet that is carefully crafted to take advantage of the buffer overflow and can be used to execute arbitrary code.

Two recommendations have been made to rectify this exploit. Firstly, as with many buffer overflow exploits, it requires applying Microsoft's patch against the LDAP attack. Alternatively, a firewall policy could deny all incoming TCP packets with destination port of 389 to protect the Exchange LDAP service from external attack.

3.3 Buffer Overflow in Netscape Enterprise and FastTrack Authentication

Netscape Enterprise Server and Netscape FastTrack Server is basically an Internet web server. Both server products contain an administration server and the HTTP server. The administration server is a lightweight HTTP server that is used for administering the administration server itself and other Netscape Servers (eg. Enterprise Server, FastTrack Server). As such, the Administration Server will run as root in Unix, System in Windows NT and the Web server runs as user 'nobody' by default.

It was discovered that Netscape Enterprise Server, Netscape FastTrack Server as well as Netscape Administration Server has buffer overflow vulnerability. This vulnerability affects all supported platforms of Enterprise and FastTrack web servers. Enterprise 3.5.1 through 3.6sp2 and FastTrack 3.01 were found to be vulnerable. The exploit is present if HTTP Basic Authentication is used. It happens when accessing a password protected environment of the Administration or Web server, entering a username or password that is longer than 508 characters will cause the server to crash with access violation error. An attacker could then utilize the Base64 encoded Authorization string to execute arbitrary code as SYSTEM on Windows NT, or as root on Unix. With this exploit, the attackers can use these privileges to gain full access to the server without requiring authentication.

It is recommended to upgrade to the latest version of Netscape Enterprise Server, version 4.0sp2 to avoid this exploit totally.

3.4 Others

As a sample, below are listed the CERT advisories from 1997 which covered some form of vulnerability from a buffer overflow together with some of the operating systems known to be impacted. This is only a small selection of all the vulnerabilities which have been seen.

CA-97.02	HP-UX newgrp Buffer Overrun Vulnerability	HP-UX 9.x and 10.x
CA-97.04	talkd Vulnerability	BSDI, FreeBSD, HP-UX, AIX, Linux, SunOS
CA-97.05	MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4	BSDI, HP-UX
CA-97.06	Vulnerability in rlogin/term	BSDI, DG/UX, Digital UNIX, Digital ULTRIX, FreeBSD, HP-UX, AIX, SunOS
CA-97.09	Vulnerability in IMAP and POP	BSDI, AIX, Linux, SunOS
CA-97.10	Vulnerability in Natural Language Service	UNICOS, AIX, Linux
CA-97.11	Vulnerability in libXt	BSDI, DG/UX, DEC, FreeBSD, HP-UX, AIX, EWS-UX/V, UP-UX/V, UX/4800, IRIX, SunOS
CA-97.13	Vulnerability in xlock	BSDI, DG/UX, FreeBSD, HP-UX, AIX, Debian Linux, SuSE Linux, IRIX, SunOS
CA-97.17	Vulnerability in suidperl (sperl)	BSDI, Linux
CA-97.18	Vulnerability in the at(1) program	HP-UX, AIX, NCR MP-RAS SVR4, SCO, IRIX, SunOS
CA-97.19	lpr Buffer Overrun Vulnerability	some BSDI, some FreeBSD
CA-97.21	SGI Buffer Overflow Vulnerabilities	IRIX
CA-97.23	Buffer Overflow Problem in rdist	AIX, IRIX, SunOS
CA-97.24	Buffer Overrun Vulnerability in Count.cgi cgi-bin Program	
CA-97.26	Buffer Overrun Vulnerability in statd(1M) Program	Digital UNIX, AIX, SunOS, IRIX

4 Countermeasures

The buffer overflow problem is severe, but there are a number of ways in which it can be tackled.

The most obvious, and also the best solution, is to write good code in the first place. The programmer should make sure that buffer boundaries are checked, especially when manipulating strings of arbitrary length. In C, the most significant problem functions are `gets()`, `strcpy()` and `strcat()` which should be replaced by `fgets()`, `strncpy()` and `strncat()` respectively. The improved solutions here require that the size of the buffer is specified as an argument, and so ensure that the buffer cannot be overflowed.

The choice of programming language can also be significant. Most problems in the wild are seen in programs written in the C language. This is a powerful and very useful programming language, but is also noted for giving the programmer enough rope to hang himself by. Some languages have defenses against this problem, such as Java and Ada which do static and runtime buffer boundary checking. Interpreted languages are often less susceptible since they are also able to do more dynamic run-time bounds checking which makes buffer overflows more difficult. Java has type-safety which should ensure that buffer overflows cannot happen, however, this still leaves room for buffer overflow problems in the type checking system itself. These have been seen in practice. Type safety ensures that data can only be written to a memory address if it is of the 'right' type, e.g. only an `int` can be written into a variable that is declared as an `int`.

C compilers have also been created that do some static bounds checking. Run-time bounds checking can create a major performance loss.

C compiler patches that do array bounds checking have also been created. This method can stop all buffer overflow problems, but at the cost of a major loss of performance.

There are a couple of other possible solutions. One option is to make the stack non-executable. For Linux a kernel patch has been created by "Solar Designer" to do this. There are some cases, however, where an executable is required. One is for trampolines, that is, simply redirecting the execution flow on function exit somewhere other than where it was called from. On Linux, signal handlers also require an executable stack. The kernel patch mentioned does allow these exceptions, though these could be used as part of an attack that would breach it. This technique has the main advantage of very little performance degradation.

The other is to provide a way to check the integrity of the stack before returning from a function. This is the method used by the StackGuard [2] patch to the gcc/egcs C compiler. It only requires alterations to the `function_prolog()` and `function_epilog()` routines. When functions start a value, called a 'canary', is placed on the stack next to the return address. When the function exits the integrity of this canary is checked to make sure that the return address on the stack has not been corrupted. In order to alter the stack pointer, the attacker would also have to overwrite the contents of the canary.

StackGuard offers a couple of ways in which this can be set. The canary may be a random value, so that the attacker cannot guess it and ensure that it does not change. Another alternative is to make it a zero. In order to preserve it the attacker would have to write bytes containing zeroes (nulls) at that location, however, string functions such as `gets()` usually terminate on a null character and so would not continue and overwrite the stack pointer.

The StackGuard techniques do add an overhead which can be seen in some loss of performance. These depend on the application but may be between 40–80%.

StackGuard does not remove the need to fix buffer overflow vulnerabilities, but does turn a root vulnerability into a milder degradation of service attack.

However, not all buffer overflow attacks involve corruption of the stack, though these are the most common. PointGuard extends the technique used in StackGuard to all code pointers (function pointers and longjmp() buffers).

5 Conclusion

Buffer overflows are a very common problem, and can have significant security related consequences.

Attacks on such vulnerabilities are not easy to create without a fair degree of technical knowledge. However, generic attack tools and cook books that are now available have contributed to a great increase in the number of incidents.

There are a number of tools which can help reduce the effect of the problem. However, The major countermeasure against such problems is to make sure that programs are written such that they do proper bounds checking in the first place.

References

[1] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. To appear at DARPA Information Survivability Conference and Expo (DISCEX). This paper will also appear as an invited talk at SANS 2000. http://www.cse.ogi.edu/DISC/projects/immunix/discex00.pdf

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In the Proceedings of the 7th USENIX Security Conference, San Antonio, TX, January 1998. http://www.cse.ogi.edu/DISC/projects/immunix/stackguard_usenix98.ps.gz

[3] Aleph One: Smashing the Stack for Fun and Profit, Phrack 7(49), November 1996, http://www.phrack.com/search.phtml?view&article=p49-14

[4] klog: The Frame Pointer Overwrite, Phrack 9(55), September 1999, http://www.phrack.com/search.phtml?view&article=p55-8

[5] dark spyrit AKA Barnaby Jack: Win32 Buffer Overflows (Location, Exploitation and Prevention), Phrack 9(55), September 1999, http://www.phrack.com/search.phtml?view&article=p55-15

[6] Evan Thomas: Attack Class: Buffer Overflows, http://helloworld.ca/1999/04-apr/attack_class.html

[7] W. Olin Sibert: Malicious Data and Computer Security, InterTrust Technologies Corporation, http://www.fish.com/security/maldata.html

[8] H. Lorin, H.M. Deitel: The Systems Programming Series Operating Systems, Addison-Wesley

[9] A.J. Van De Goor: Computer Architecture and Design, Addison-Wesley

[10] William Stallings: Operating Systems: Internals and Design Principles, Prentice Hall

[11] Andrew S. Tanenbaum: Operating Systems: Design and Implementation, Prentice Hall

[12] Donn Seeley: A Tour of the Worm, http://www.alw.nih.gov/Security/FIRST/papers/virus/tour.ps

[13] Gene Spafford: worm_oflow.c, http://www.worm.net/worm-src/worm_oflow.c