The following paper was originally published in the
Proceedings of the Sixth USENIX UNIX Security Symposium
San Jose, California, July 1996.

Murphy’s law and computer security

Wietse Venema
Eindhoven University of Technology
The Netherlands

Murphy's law and computer security

Wietse Venema

Mathematics and Computing Science
Eindhoven University of Technology
The Netherlands
wietse@wzv.win.tue.nl

Abstract

This paper discusses lessons learned from a selection of computer security problems that have surfaced in the recent past, and that are likely to show up again in the future. Examples are taken from security advisories and from unpublished loopholes in the author's own work.

1. Widely-known passwords

Imagine that you choose a password to protect your systems and then advertise that password in big neon signs for all to see. That would not be a very responsible thing to do. Surprisingly, this is almost exactly what some programs have been doing for a long time.

Passwords are the traditional method to authenticate users to computer systems. With todays computer networks, large pseudo-random numbers are being used as password tokens or as cryptographic keys for communication between computer programs. Examples that will be discussed in this paper are Kerberos tickets, X11 magic cookies, and NFS file handles.

These password tokens or cryptographic keys are not chosen by people. Instead, a pseudo-random value is generated programmatically whenever one is needed. Unfortunately it is easy to end up with a predictable result.

1.1. Predictable Kerberos keys

Recent advisories allude to a problem that allows an attacker to impersonate users of the Kerberos version 4 [Stei88, Kohl93] authentication system. In order to understand the problem it is not necessary to go into the details of how Kerberos works. The problem is with the generation of encryption keys.

In the Kerberos system, temporary encryption keys are used to protect authentication information. One of these encryption keys is generated by the authentication server at the very beginning of a login session: it is part of the ticket-granting ticket that allows an authenticated user to obtain service through the network without having to provide a password each time.

Unfortunately, the key generation algorithm used by the version 4 authentication server was predictable. Ultimately, all encryption keys were derived from *known* information (the time of day), from *constant* information (a process ID and a machine identity), and from *predictable* information: a counter that was incremented with each call. The result: a cryptographer's nightmare.

Armed with this knowledge, an intruder could impersonate other users without even having to know their password. The code fragment below illustrates the problem:

```
p = getpid() ^ gethostid();
gettimeofday(&time, (struct timezone *) 0);
/* randomize start */
srandom(time.tv_usec ^ time.tv_sec ^ p ^ n++);
```

The fragment shows a fine example of a comment that lies: feeding predictable input into a deterministic routine produces a predictable result. Curiously, the Kerberos version 4 source code already contains an improved key-generation algorithm that does use secret data as input, but the improved algorithm was not put into actual use until the release of Kerberos version 5[1].

1. Several vendors were already aware of the problem and had taken measures in their own version of the software.

1.2. Only 256 different magic cookies

The X Window system [Schei86, Schei92], comes with the XDM graphical login tool. Some vendors provide their own alternative, but the programs all perform the same basic task: display a logo on the screen and prompt for a login name and a password. When a correct name and password are given, an X session is started. This can be a default desktop that is provided by the system, or it can be a desktop as specified by the commands in a *.xsession* file in the user's home directory.

When an X application program connects to the X server program (i.e., to the user's keyboard, screen and mouse), it typically authenticates according to one of three methods:

- No authentication: every user on the network has access to the user's keyboard, screen and mouse. This is the default on too many systems.
- Client network address: all users on specific hosts have access to the user's keyboard, screen and mouse. The client network address is provided by the client host.
- Magic cookie: all users that know a 128-bit secret value have access to the user's keyboard, screen and mouse. The secret is typically kept in a file *.Xauthority* in the user's home directory. The secret is sent in the clear over the network.

Other authentication methods exist, based on data encryption techniques, but their use is less common. Authentication methods differ in strength. Authentication with magic cookies is more secure than authentication by client network address. Authentication by network address, in turn, is a lot more secure than no authentication at all.

The XDM program suffers from a problem much like the one described earlier for Kerberos version 4: magic cookies are generated from non-secret data (time of day and process ID). Such cookies can be guessed in a small amount of time if one has access to the victim's machine.

Cookie guessing becomes harder, but not impossible, without access to the victim's machine: it is still possible to find out the approximate time of login by fingering the host. However, until recently, some XDM implementations suffered from an even worse flaw that made them vulnerable to arbitrary users on the network. A fix was announced in November 1995.

The problem was than many implementations of the UNIX random number generator are truly horrible: the low 8 bits of its result repeat with a cycle of length 256. These low 8 bits are exactly what some XDM implementations use when they generate a magic cookie:

```
for (i = 0; i < len; i++) {
    value = rand();
    auth[i] = value & 0xff;
}
```

With a cycle of length 256, there can be only 256 different magic cookies! It takes only a fraction of a second to try them all and to find out which of the 256 magic cookies an X server is using. It is as if every other house has the same key to the front door.

Fortunately, many XDM implementations are not vulnerable to this particular problem: they either use a better random number generator or they use an algorithm that involves cryptography.

1.3. Identical NFS file handles

In a discussion of security loopholes, the Network File system [Call95] cannot remain unmentioned. The *fsirand* flaw that I describe here was found and fixed in SunOS 4 years ago. In order to explain the problem I will describe the NFS protocols in a nutshell.

When an NFS client host wants to access a remote file or directory, it sends a request to the file server's NFS daemon. The request includes an NFS file handle that identifies the object being accessed.

How does an NFS client obtain an NFS file handle? Honest clients use the NFS mount protocol. When an NFS client host wants to access a remote file system for the very first time, the client host sends a mount request to the file server's mount daemon. The mount daemon verifies that the client host has permission to access the file system. When the mount daemon grants access, it sends an NFS file handle back to the client.

Once an NFS client has a file handle, it can send file access commands directly to the file server's NFS daemon. In fact, *any client* that is in the possession of a valid NFS file handle can use it. Export restrictions are primarily enforced by the mount daemon. In the most common cases the file server's NFS daemon does not care what client is talking to it.

How, then, does an NFS server protect itself against malicious clients that make up their own NFS file handles? SUN's solution was to make NFS file handles hard to guess. When a file system is created, the *fsirand* program initializes all NFS file handles with pseudo-random numbers. The program is seeded with a process ID and with the time of day. Unfortunately, early fsirand implementations did not initialize the time of day variable. Because of this missing initialization, the time of day variable contained fixed garbage values, depending on the system architecture. Only the process ID was being set [Dik96].

Many sites run the same *suninstall* procedure to install the operating system onto their disks. This procedure is highly automated, and by implication, the *fsirand* process ID is very predictable. Unknowingly, many sites initialized their file systems with the same NFS file handles world wide.

Thus, in order to access a file system there was no need to use the NFS mount protocol at all. Every other house in the street did have the same keys to the front door.

While researching this paper I noticed that many systems do not even have an *fsirand* command or its moral equivalent. Some systems simply use the time of day when allocating a filesystem inode.

1.4. Moral

Why all this attention to problems with the generation of pseudo-random numbers? The answer is cryptography. Whether we like it or not, cryptography is becoming more and more important for the protection of data and systems. Pseudo-random numbers are essential for the generation of hard-to-guess cryptographic keys. The best encryption in the world is of no use when encryption keys are taken from a predictable source, or when the keys are taken from a too small domain.

In order to generate a secret password you need a secret to begin with. The time of day and the process ID are often not secret. Kerberos is just one system that has suffered from key-generation problems. Another example is the Netscape navigator. Until September 1995, this software generated session keys with only about 30 bits of randomness [Net95]. This amount is even less than the 40 bits that the US government presently allows for exportable cryptography.

In RFC 1750, Eastlake *et al* recommend the use of external sources for randomness [East94]. Even inexpensive computers provide excellent opportunities: for example, keystroke timings, mouse event timings, or disk seek times. The UNIX kernel gives convenient access to all this information and more. By running various incantations of the *ps* command you get a look at information that changes rapidly. When you are attached to a network (and who isn't these days?), statistics from the *netstat* command can be a good source, too, with the caveat that network traffic can be manipulated and monitored. The UNIX kernel is exactly what I used as source of randomness for the SATAN [Farm93] cookie generator: I never even considered the use of a random-number generator.

2. Burning yourself with malicious data

Only a few years ago, Eindhoven University was a peaceful site. Some may remember its name from the days of the great Professor Dijkstra who did fundamental computing science work on structured programming, deadlock avoidance, and so on. And of course, Eindhoven is the place where Philips began making light bulbs and thus started its electronics imperium.

The peace came to an end when Eindhoven University was connected to the global Internet. The university computer systems became immensely popular with data travelers. The facilities had become an excellent starting point to get onto 'the net'. Most visitors were careful not to break things. One visitor, however, had a problem. Every month or so he would break into one of the university computer systems, acquire system privileges, and wipe the machine completely clean:

```
# rm -rf / &
```

The TCP Wrapper [Ven92] began as a simple tool to maintain a log of the intruder's network access attempts. In the course of time I extended the program to learn more and more about the opponent. A powerful extension was the *booby trap*. It allowed us to automatically execute shell commands whenever a suspicious connection attempt was made to our systems. The typical application was to finger the host that connected to our site, in order to get information about the user who was trying to break into our systems.

```
ALL: .bad.domain: \
  finger -l @%h | /usr/ucb/mail root
```

The booby trap feature is very flexible: before the command is given to the shell, it is subjected to substitutions. For example, the sequence *%h* is replaced by the name of the remote host, or by its network address when the name is unavailable.

The TCP Wrapper development process is a tedious one: because so many systems depend on it, everything needs to be tested very extensively. I had been using booby traps for almost a whole year before I included the feature into the public TCP Wrapper release. Only a few months later I received an email message from a kind Mr. Icarus Sparry [Spar92]. He informed me that the booby trap feature could introduce a security loophole.

The problem was as follows. The booby trap feature substitutes host names into shell commands. Host names are looked up via the Domain Name System (DNS), which is a distributed database. The reply to a DNS query can literally come from anywhere on the

Internet. With the booby trap, I was substituting untrusted data into shell commands that were being executed with *root* privileges. This was not nice.

I quickly ran a few tests and, indeed, the DNS server would accept almost anything for a hostname. With an unmodified DNS server I was able to generate hostnames containing various shell metacharacters. In the DNS server data base files, only a few characters have a special meaning (dollar, semicolon, white space and a few others). Anything else can be put into a hostname, for example something as destructive as:

    >/etc/passwd

The attack is not as easy as it seems, though. The TCP Wrapper attempts to expose malicious DNS servers by asking for a second opinion. The program compares the name and address results from a reverse lookup (by host address) with the name and address results from a forward lookup (by host name) and detects discrepancies. Thus, an attacker would have to manipulate the results of both forward and reverse lookups. Nevertheless, it is a bad idea to pass uncensored data from the network into, for example, commands that are given to a shell.

When a program has to defend itself against malicious data, there are two ways to fix the problem: the right fix and the wrong fix. The right fix is to permit only data that is known to give no problems: letters, digits, dots, and a few other symbols. This is the approach that I took with the TCP Wrapper: when doing substitutions on shell commands it replaces characters outside the set of trusted characters by underscores.

Unfortunately, many people choose the wrong fix: they allow everything except the values that are known to give trouble. This approach is an invitation to disaster. Only months ago, part of the WWW (world-wide web) community discovered that CGI (common gateway interface) scripts and other WWW server helper applications can be manipulated by sending data containing newline characters, especially when that data is passed on to shell commands.

2.1. Moral

Recent advisories point out that programs can be manipulated by malicious data from name servers. Together with the *sendmail* program, TCP Wrappers and CGI servers are not the only programs that must defend themselves against malicious data. For example, if you do automated logfile analysis with Swatch [Hans92] or with other tools, you'd better be careful when passing data from logfiles on to other programs: untrusted systems can often send data directly to the *syslog* daemon.

3. Secrets in user-accessible memory

In a previous life I was a nuclear physicist. Working with delicate equipment was a matter of daily routine. Of course things would stop working at the most inconvenient hour of the day. One golden rule that I learned very quickly: if you're fixing a problem, better be sure that you're not breaking something else in the process.

In the information technology world, working with fragile pieces of software is a matter of daily routine. Of course programs stop working at the most inconvenient hour of the day - hardware failures are becoming rare. One golden rule that a system administrator learns very quickly: if you're fixing something, better be sure that the solution does not break something else. Of course we never break something while fixing a problem.

The programmer is faced with the same problems. All too often something breaks while a security or non-security problem is being fixed[2]. Shadow passwords are a good example: they were invented to fix one problem and at the same time opened up a hole.

Shadow passwords attempt to cure a real problem. By the end of the eighties, computers had become fast enough that large-scale password cracking became practical. Alec Muffett's crack program [Muff92] gave everyone the best available tool at the time. Like many UNIX password cracking programs, Alec's program takes encrypted passwords from a password file and tries to guess them in a systematical manner. It is amazing what it found when I first ran it on our own password files.

Some people stated that such programs should not be made available because they might help intruders to break into systems. History repeats. We had a similar discussion again about the release of SATAN, only this time the opposition was much stronger.

Instead of applying censorship to the distribution of software, a more practical solution is to get to the root of the problem: prevent users from choosing weak passwords in the first place, and make encrypted password information less easily accessible. The usual approach is to move the encrypted passwords to a so-called shadow password file that is accessible only to privileged programs. With shadow passwords an attacker can no longer run a password cracker on a stolen copy of the regular password file. Instead, one must connect to the machine to try a password. This makes the risk of detection much larger.

2. Not to mention the things that break as a side effect of fixing a non-problem.

Shadow passwords can give a false sense of security, though. It seems as if there is less need to be careful when choosing a password. After all, the encrypted password is protected from password cracking programs. However, it is easy to see how shadow passwords can actually weaken a system's defenses. Here is a simplified description of how the login program works:

- read username and password from user
- look up password and shadow file entry
- validate username and password
- drop privileges and execute login shell

In the second step, the login program is still privileged, so it can read the secret shadow password file. Every practical login implementation will read a lot more data than just the entry for the user logging in: most likely it reads a whole kilobyte of data or even more. This excess data lingers on in memory buffers somewhere in the process address space.

The fourth step implies a race condition: in-between the time the login program switches to the user and the time it executes the login shell, the login program can be signaled by the user. Shadow password file information that was read in step 2 can be found in the core dump.

It is easy to fall into the trap and keep secret data in the memory of unprivileged programs. I fixed my logdaemon [Ven96] utilities years ago when I ported them to Solaris 2, which has shadow passwords. The fix prevents programs from dumping core.

The SSH [Ylo96] utilities suffered from a similar problem. SSH is a re-implementation of rlogin, rsh and of a few other utilities. It aims to improve host and/or user authentication by the use of strong cryptography. Some of this protection was lost when it was discovered that cryptographic keys remain in memory after a process has switched privilege to the user.

3.1. Moral

The moral of this story is that one should avoid carrying secret information in the memory of unprivileged programs. Preventing core dumps does not give total protection, though; it takes kernel support to protect a process against manipulation with a debugger program[3]. It is therefore essential that a process wipes secrets from memory before switching user privileges. Part of the solution is to use a modified memory manager (malloc) that wipes memory upon return to the free memory pool.

3. In particular, the UNIX kernel should not allow unprivileged users to debug a process with an *effective*, *real*, or *saved* user or group ID that belongs to another user [Ellis95].

4. Depending on other programs

Being the author of a popular security tool is much like walking a wire thirty feet above the ground without a safety net. The bright side of such an elevated position is that you get a nice view of the world. The dark side is that an error can have painful consequences.

A case in point is the booby trap example of a few sections ago: the example where we finger a host whenever a connection comes from a suspicious source.

```
ALL: .bad.domain: \
  finger -l @%h | /usr/ucb/mail root
```

The booby trap depends on two programs: *finger* and */usr/ucb/mail*. The finger program is relatively harmless: after all, it just connects to the host and reads the reply across the network. Unfortunately, it is not as harmless as it appears to be at first sight. Imagine what happens when the remote finger demon just keeps sending data forever: sooner or later the disk fills up with an enormous temporary file. A judicious link from */dev/zero* to a user's *.plan* file is sufficient to effect a denial of service attack.

The bigger problem is with the dependence on /usr/ucb/mail. This is a complex program with many features. One feature is that it has so-called shell escapes: the mail program recognizes commands in its input when they are preceded by a tilde character. The tilde-command feature was useful when composing a message at the terminal. Nowadays, few people still use the raw /usr/ucb/mail command in interactive mode, but the feature is still there. By inheritance, tilde-command is also supported by the System V *mailx* program.

Three years ago, Borja Marcos pointed out in private email that these tilde commands are also recognized when /usr/ucb/mail reads its input from a pipe [Marc93]. This was bad news: anyone could put shell escape commands into their .plan file and have them executed remotely by triggering a TCP Wrapper finger+mail booby trap.

I decided to silently fix the problem by flooding the market. Each year brought a new major TCP Wrapper release with enough useful features to make people upgrade their old version to the current one. With the *safe_finger* program I attempted to eliminate all known problems with the finger+mail booby trap. While I was at it, I took the opportunity to solve a few other problems, too: fingering a host is not a trivial activity.

This time, at least, I was in good company: a year later it became known that the widely-used INN news transport software [Salz92] had fallen into the same trap. It was possible to execute shell commands world-wide on news servers running INN, by posting just one single news article.

The problem with /usr/ucb/mail shell escapes is going stay with us for quite a while: I have found that many web sites run CGI helper scripts that send data from the network into /usr/ucb/mail, without censoring of, for example, newline characters embedded in the data.

4.1. Moral

Insecurity by depending on other programs is an old problem. The problem becomes even worse when security depends on programs that were not designed for security purposes. Both finger and /usr/ucb/mail are unprivileged programs. They were not designed to defend themselves against malicious inputs from the network. The finger+mail loophole is just one type of accident that can happen. As the author of the TCP Wrapper, it is not possible for me to protect every user against every possible accident. The TCP Wrapper booby trap feature is a sharp tool and it should be used with care.

5. Concluding remarks

The problems discussed in this paper are only the tip of a large ice berg of recurring security problems. There was a lot of material to choose from, and the selection is far from representative. UNIX environment settings are just one example of a major source of recurring trouble that could not be discussed.

All examples in this paper were taken from the UNIX world. Is UNIX such a bad system? Of course not. UNIX is a platform where much pioneering work was done and still is being done. It is therefore not surprising that many errors were made first in the UNIX environment. On the positive side, these experiences should give UNIX users an advantage. For example, now that PC operating systems become capable enough to provide network services, we can expect to see a lot of familiar problems coming back.

As the examples in this paper show, the path of the security programmer is riddled with land mines. The author has had the questionable privilege to meet Mr. Murphy several times in person. Murphy is a cruel teacher.

6. References

[Call95]
B. Callaghan, B. Pawlowski, P. Staubach: NFS version 3 protocol specification. RFC 1813, June 1995.

[Dik96]
Casper H.S. Dik. Private communication, May 1996.

[East94]
D. Eastlake, S. Crocker, J. Schiller: Randomness recommendations for security. RFC 1750, December 1994.

[Ellis95]
James T. Ellis. Private communication, May 1995.

[Farm93]
Dan Farmer, Wietse Venema: Improving the security of your site by breaking into it. ftp.win.tue.nl:/pub/security/admin-guide-to-cracking-101.Z, December 1993.

[Hans92]
Stephen E. Hansen, E. Todd Atkins: Automated system monitoring and notification with Swatch. Proc. UNIX security III, Baltimore, September 1992.

[Kohl93]
J. Kohl, C. Neuman: The Kerberos network authentication service (V5). RFC 1510, September 1993.

[Marc93]
Borja Marcos. Private communication, June 1993.

[Muff92]
Alec D.E. Muffett: Crack version 4.1 - a sensible password checker for UNIX. Part of the Crack distribution, ftp://cert.org/pub/tools/crack/

[Net95]
Netscape technical documents: Potential vulnerability in Netscape products. http://www.netscape.com/newsref/std/random_seed_security.html, September 1995.

[Salz92]
Rich Salz: InterNetNews: Usenet transport for Internet sites. Proc. Usenix conference, San Antonio, June 1992.

[Schei86]
Robert W. Scheifler, Jim Gettys: The X window system. ACM transactions on graphics vol. 5, no. 2, April 1986.

[Schei92]
Robert W. Scheifler, Jim Gettys: X window system (third ed.). Digital Press, 1992.

[Spar92]
Icarus Sparry. Private communication, August 1992.

[Stei88]
Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller: Kerberos: an authentication service for open network systems. Proc. winter Usenix conference, Dallas, 1988.

[Ven92]
Wietse Z. Venema: TCP WRAPPER, network monitoring, access control and booby traps. Proc. UNIX security III, Baltimore, September 1992.

[Ven96]
Wietse Z. Venema. The logdaemon utilities provide a login program and several network daemons with enhanced logging and authentication. ftp.win.tue.nl:/pub/security/logdaemon_XX.tar.gz.

[Ylo96]
Tatu Ylönen: SSH - secure login connections over the internet. Proc. UNIX security VI, San Jose, July 1996.