5. Symbolic links

Symbolic links, or "soft" links, were introduced in the BSD system to overcome some of the limitations of classic Unix links, now renamed "hard" links to distinguish them from the new ones:

- hard links cannot point to directories
- hard links cannot point to a file on another device.

Symbolic links (symlinks from now on) do not have these limitations, but they have other problems of their own.

Hard links are created and removed using the `link()` and `unlink()` system calls. These calls simply add or remove "(name, inode number)" pairs in directories. The kernel keeps track of all the hard links pointing to an inode and only frees the inode (and the disk space allocated for its data) if there are no hard links (and no open file descriptors) pointing to it.

The decision to disallow links to directories came very early in Unix development, even before V1[a]. However, in early revisions, the superuser could still create them: the system call that created new directories didn't add the the special "`.`" and "`..`" entries and a set-uid root `mkdir` command created them using privileged calls to `link()`. Later, this functionality was built into the system call, and now even root cannot create links to directories.

[a] https://www.tuhs.org/Archive/Distributions/Research/McIlroy_v0/UnixEditionZero-Threshold_OCR.pdf (page 5, last paragraph)

R You can see the number of existing hard links to any file or directory in the second column of the long listing output of `ls`.

There is no way to free an inode, other than removing all the hard links pointing to it. In particular, the `rm` command uses the `unlink()` system call.

A symlink is not just an entry in a directory. Instead, it is a special kind of file with its own inode. The *contents* of the symlink are a filesystem path, pointing to a file or directory (or anything else) which is called the *target* of the symlink. There is no "hard" relationship between the symlink and its target, which is in no way affected by the creation of the symlink. In particular, symlinks to the same target are not reference-counted, and the target of a symlink does not even have to exist.

Most of the system calls that take a path as an argument (e.g., `open()`, `execve()`, `chdir()` and so on) will "follow" a symlink when parsing the path. That is, if one component of the path turns out to be a symlink, they will first follow the path to the symlink target and then continue with the rest of the original path. Note that this can be repeated recursively if they find more symlinks in the target path, up to a maximum "nesting level" which is a system constant. Exceptions are `link()` and `unlink()`, which do not follow symlinks in the last component of their paths. This is especially important in the case of `unlink()`, which will then remove the *symlink itself* instead of the symlink target.

5.1 Exploiting symlinks

A few properties of links and `open()` are particularly interesting from an attacker's point of view.

- Both hard links and symlinks can be created by any user to any file (or directory, in the case of symlinks). The link creator only needs the usual permissions in the directory where she creates the link, but she needs no special permissions on the target.

  R In the case hard links, however, the link creator still needs the search permission (`x` flag) on all the directories in the path leading to the target.

  The idea is that permissions to read, write or execute the target file are still determined by the target inode, and so the user creating the link is not gaining any new permissions.
- Programs that want to create a file usually pass the `O_CREAT` flag to `open()`. This flag tells `open()` to create the file if it does not exist. However, if the file already exists, no error is returned and the existing file is opened. In particular, this is the behaviour of the `fopen()` function in the stdio C library, when using the `"w"` or `"a"` open modes.
- The behavior of `open()` with `O_CREAT` on symlinks with non-existent targets may be surprising. The system call transforms the path, by following the symlink, before doing anything else. Once it has the transformed path, now pointing to the target, it checks whether the file exists and creates it if necessary. So the system call will create the missing target!

The combination of these features can enable attack vectors when privileged programs create files in directories writable by less privileged users. A typical example is temporary files created in the `/tmp` directory, which is writable by everyone. If the privileged program creates temporary files with predictable names, without checking that they don't already exist, it may unintentionally open a symlink created by an attacker. The effect is that the privileged program will operate on the target of the symlink, which could be anything the attacker wants. Due to the `open()` behavior on symlinks with non-existent targets, the program may even create files of the attacker's choosing.

This type of bug is very common. A search for the keyworkd "symlink" on https://www.cve.org/ returns more than 1590 entries, starting in 1999 and ending in 2024. Figure 5.1 shows the number of symlink advisories per year.

In many cases, the symlink attack can be used to cause a denial of service, by writing garbage into an important system file. In some cases, however, it can be used to escalate privileges. Let us consider some examples from the CVE list.

5.1.1 CVE-1999-1187

Details about this bug can be found here:

https://marc.info/?l=bugtraq&m=87602167419803&w=2

The bug is in a version of PINE, an old mail reader. The program creates a temporary file in `/tmp` with a random name, but always the same for the same user. It also creates the file with write permissions for everyone. The idea is to trick PINE into creating a strategic file that belongs to a victim user and is writable by the attacker.

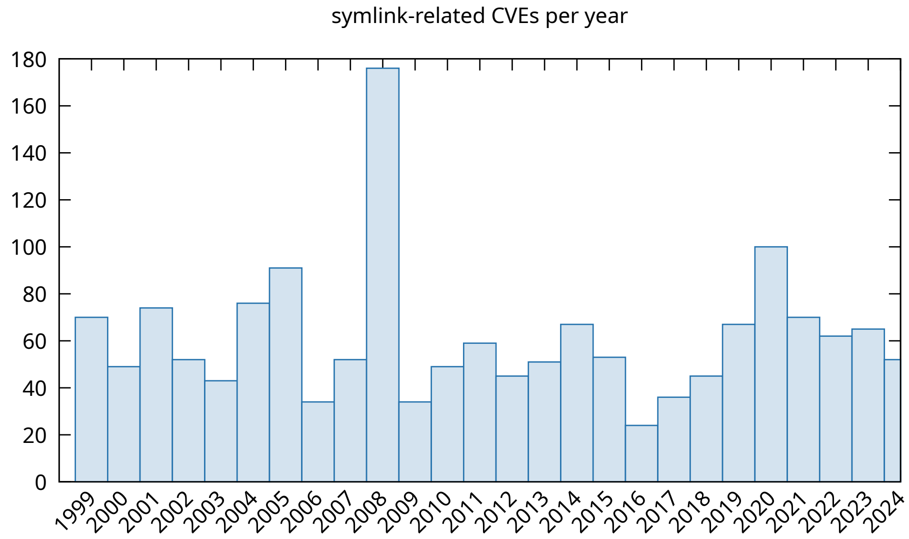

Figure 5.1 – The graph was obtained by searching for "symlink" or "symbolic link" in the CVE database.

The attacker can learn the temporary file names of the other users (by looking at `/tmp`), create a symlink with the same name while the victim is not using PINE, and wait. When the victim opens PINE again, the program will open the attacker's symlink with the victim's credentials.

PINE stands for Pine Is Not Elm, a play on an older mailreader called ELM, for ELectronic Mail.

In the proof of concept shown in the link above, the attacker creates a symlink to a non-existent `.rhosts` file in the victim's home directory. This file was used in the old (pre `ssh`) system for remote access between networked Unix machines. It essentially played the role of the modern "`authorized_keys`", but without the keys: hosts listed in `.rhosts` are allowed access without passwords. When the victim opens PINE, a world-writable `.rhosts` file is created in the victim's home directory. The attacker can later write whatever she wants to the file and gain access to the victim's account.

5.1.2 CVE-1999-1091

This is a similar problem in the `tin` newsreader. The attack in this case is even easier because this program creates a word-writable log file in `/tmp` with a fixed name. More specifically, it "created" a `/tmp/.tin_log` file without checking that it did not already exist, and then changed its permission to `0666` (read-write permissions for everyone). The attacker can exploit this vulnerability in much the same way as the one above, but this time it can even be used to gain write permission on existing `.rhosts` files.

5.1.3 CVE-2020-8019

Now let us look at a more recent one. Newer vulnerabilities tend to be harder to exploit, but they still exist. This vulnerability can only be exploited by users who can log in as the `news` user, or users who belong to the `news` group. The bug is in the install script of the `syslog-ng` package, which tries to create a /var/log/news/news.err file without checking that it doesn't already exist, and then changes the owner and group of the file to news:news. Since the /var/log/news directory is writable by the news user and group, the attacker can create a news.err symlink and wait for root to reinstall the package. The installation script will change the owner and group of the file symlinked by the attacker. The proof-of-concept suggests symlinking the /etc/shadow file, which stores all the encrypted passwords and is normally only readable by root. A write permission to this file allows the attacker to change the passwords of other users, while a read permission can be used to start a dictionary attack and learn any weak passwords contained therein.

5.1.4 CVE-2021-25321

Here is an example of an unfortunately not-so-rare occurrence: a vulnerability introduced in an attemp to improve security. It is a bit unusual in that it involves a chown() operation rather than a file creation, but it still involves symlinks.

The arpwatch daemon can be used to monitor changes in the ARP tables (relating MAC and IP addresses). It opens a network interface in raw mode to monitor ARP messages, and maintains a database of current MAC-IP pairs in a file (by default, /var/lib/arpwatch/arp.dat). Opening the interface in raw mode requires root privileges, but once opened the privilege is no longer needed. As a standard mitigation strategy, the daemon drops privileges (using setuid()) to another non-root user, call it runtime. The SUSE Linux distribution of this daemon included code to also change the ownership of the /var/lib/arpwatch directory and the arp.dat file within it, so that the runtime user can access them. This leads to an easy exploit if anyone can log in as runtime: once the ownership of /var/lib/arpwatch has changed, remove the arp.dat file and create a link to, say, /etc/shadow in its place . The next time time arpwatch is started, the daemon will go through the change-owner code again, this time allowing runtime to become the owner of /etc/shadow.

5.1.5 CVE-2022-35631

Classic bugs like predictable temporary file names are still beeing introduced today. This CVE addresses a bug in Velociraptor, a forensics and incident response tool that security professionals should use to monitor and inspect systems that may have been compromised by attackers. The systems being monitored run Velociraptor clients (with high privileges) talking to a remote server. Unfortunately, the client created a temporary file with a name written in its configuration file, placing too much trust in the system it was supposed to be monitoring.

Exercise 5.1 — bad4suid. Solve the *bad4suid* challenge. You can draw inspiration from Section 5.1.1: the rsh command will connect to a local rshd daemon. The daemon will ask for a login name, then check for an .rhosts file in the home directory of the user; if the .rhosts file doesn't exist or doesn't grant you passwordless access, the daemon will ask for a password. Also note that a + in the .rhosts file is a wildcard that grants access from any host. ■

Exercise 5.2 — bad5suid. Solve the *bad5suid* challenge. You can use rsh as in Ex. 5.1. ■

5.2 Countermeasures

Programs should always check that the files they intend to create do not already exist. However, code like this is not sufficient:

BUG

```
if (stat(file, &sb) < 0 && errno == ENOENT) {
  open(file, O_CREAT ...);
}
```

The problem with the above code is that there is a *race condition* between the `stat()` system call used to check that the file does not exist, and the subsequent `open()` used to create it. An attacker could create the symlink in the window between the two calls. The correct way to create the file is to use the `O_EXCL` flag in addition to the `O_CREAT` flag. That way, `open()` will atomically check that the file does not exist before creating it, and will return an error if it does not. Note that it will also return an error if the path is actually a symlink, even if the target of the symlink does not exist. Thus, this technique closes all known attack vectors.

If you are trying to create a temporary file, you should use the `mkstemp()` library function, which uses the above technique and also automatically choses an hard-to-guess random name.

> **Exercise 5.3 — bad6suid.** Solve the *bad6suid* challenge. You can use `rsh` as in Ex. 5.1 and 5.2. ■

5.2.1 Linux-specific countermeasures

Linux implements a number of non-standard countermeasures against symlink attacks.

The first is a kind of "safety net" for buggy programs that create temporary files insecurely. It can be enabled by typing

```
# echo 1 > /proc/sys/fs/protected_symlinks
```

as root (your distribution may have already enabled this for you). It works like this: if the sticky bit is set on a world-writable directory (such as `/tmp`), Linux will only allow a process to follow a symlink if either the process owns the symlink, or the symlink and the directory have the same owner (which is usually root for `/tmp`). This way, even if the attacker has installed a symlink, the victim process will not be able to follow it and will receive an error. This turns a potential privilege escalation attack into a more benign denial of service for that particular process. There is also an option to protect hardlinks, but it works differently: if `protected_hardlinks` is 1, users can only create hardlinks to files they own, or to files to which they have both read and write access. The restriction is more severe in this case because hardlinks cannot be detected once they have been created, and also because hardlinks can be used to prevent files from being deleted (think of a superuser trying to remove a vulnerable suid program: an attacker can create a hardlink to the program, thus keeping a copy around). Modern kernels also have options to protect FIFOs and regular files in world-writable sticky directories.

The second countermeasure Linux implements is the `O_TMPFILE` flag for the `open()` primitive. This flag causes `open()` to create a file with no links in the file system, i.e. it just allocates an inode without giving it a name. The primitive is guaranteed to always allocate a new inode, so symlink attacks are impossible. It also saves the programmer the trouble of inventing a unique name for the temporary file. Finally, since the inode has a link count of 0, the temporary file is automatically removed when all its open file descriptors are closed. In the normal case this happens automatically when the process exits, even if it does not exit cleanly (e.g., because it is killed). If you are writing an application that needs to run only on Linux, you should consider using this flag whenever you create a temporary file.

> **Exercise 5.4 — bad4suid2.** Solve the *bad4suid2* challenge. ■

Exercise 5.5 What happens if we set protected_symlinks in Exercises 5.1, 5.2, or 5.3? Are the exploits affected by the status of protected_hardlinks? ■