TECHNICAL FEATURE

Unix Shell Scripting Malware

Marius van Oers
McAfee AVERT, The Netherlands

Unix/*Linux* binary malware can be very dependent upon distribution flavour and kernel version. Furthermore, the use of binary files as a starting point for virus infection may not always be very successful – starting off with a coredump will result in a rapid failure.

In the past we have seen worms (for example Linux/Adore) make use of a combination of ELF binary files and scripts. Usually scripts are independent of distribution flavour and kernel version, and most are likely to have few problems running on the target machine.

For some worm packages, scripts act as the 'fire-starters' on the target PC. The scripts may execute directly, or may call other script files and binaries. Sometimes local files are replaced by compromised ones that are included in the worm package.

So what are the possibilities in the Unix world for malicious code using scripting?

Unix Scripting

There are a number of different Unix/*Linux* distributions, the majority of which support scripting. Similarly, there are a number of different forms of scripting.

Javascript is supported on both *Windows* and on most Unix/*Linux* systems. Therefore, the creation of Javascript malware that will work in both operating system environments is technically possible, and it should be relatively easy to accomplish.

The binary infector W32/Lindose was a 32-bit PE *Windows*-based infector that searched the system for binary ELF files to infect, however Lindose does not operate in the opposite direction (i.e Unix to *Windows*). Technically, this should have been achievable – consider, for example, that emulator programs exist on Unix systems to run Win32 code. However, a considerable level of technical expertise would be needed to achieve this and, more importantly, it would be a significantly time-consuming process. It would be both quicker and easier to write a Javascript virus that can run natively in both the *Windows* and Unix environments.

The powerful Perl scripting is supported on a lot of Unix systems, either installed directly or using an add-on package. A sample file might be called 'runme.pl'.

Unix shell scripting is very powerful too; it may control program configuration and start/kill services. Unix shell scripting has many flavours, for example Bourne (sh), Bourne Again (Bash), Korn, C and Tops C shell scripting. Also it is possible to create a completely new shell interpreter. However, the most common is the Bourne Again shell scripting, using the '/bin/sh' interpreter. A sample file might be called 'runme.sh'.

A virus writer making the assumption that Bourne Again is the default shell interpreter runs the risk, should this not be the case, of the virus producing errors and crashing. A simple way to avoid this situation is to insert a '#!/bin/sh' line at the start of the file.

On *Linux* systems '#!/bin/sh' will act as a redirect to Bash, but on other Unix systems there are differences between sh and Bash. An alternative shell interpreter can be specified, for example using '#!/bin/csh'. On *Solaris* systems the Korn shell, ksh, is used widely.

Now let's take a closer look at Bourne shell scripting and the malware making use of it.

Unix Shell Malware

Creating malware using shell scripting is relatively easy. Simple viruses may be very short, consisting of only a few lines, and even less code is needed to construct a Trojan.

Another aspect is that, unlike binaries, the Bourne shell scripts will (usually) work on a large number of different Unix flavours (or will do so with a few very minor modifications).

By examining some samples that were distributed in the latest publication of a well-known virus-writing group, we can take a look at what possibilities and techniques exist for shell viruses.

Determining Which Files to Infect

With the support of 'if-then-else' and 'for-do' loops it is easy to create viruses that search files for suitable targets. The search can be carried out both in the current directory and in others, using directory walking loops.

Suppose we have a simple Bourne shell virus; without filtering the viral shell code could be added to binary files. So, in order to prevent unexpected results, proper filtering is required.

Grep

Usually Bourne shell scripts start with a reference to the interpreter, '/bin/sh', in the file header. So a quick check for files that start with '#!/bin/sh' would provide a good subset of initial target files for infection.

This search is possible using the 'grep' command. For example:

```
'grep –s #!/bin/sh $targetfiles'
```

In this case the '–s' option is used to suppress any error messages.

Head

Instead of examining the complete file, the head of the file can provide useful information for faster filtering.

The 'head' command returns information on the beginning of a file. For example, 'head –20 $file' returns the first 20 lines of the file, while 'head –c20 $file' returns the first 20 characters of the file.

File

Using the command 'file' it is possible to determine whether the filetype of a target file is of Bourne shell format. However, this technique is rarely used; it is not a perfect technique, as it reads file headers to determine the file type.

In some cases, for example with .sh scripts, it is not necessary for shell scripts to have lines such as ' #!/bin/sh' at the beginning of the file. Although this command interpreter line is encountered frequently, it is not mandatory. Files without the expected command interpreter line could be judged by 'file' to be regular ASCII files rather than shell script files.

Find

Unix systems have a wide range of protection techniques, so, in addition to this file checking, a virus should investigate the target file's permissions – for example, determine whether these are set to read (-r-) , write (-w-) and/or executable (-x-).

Some viruses walk through directories/folder trees but upon infection fail to check whether the target is a file or directory, which may result in crashes.

The 'find' command can be used to search for specific target files. And, not only can 'find' filter on files with specific attributes (-r -w -x etc.), but it can also execute a command on the target files that are found. However, making use of 'find' may result in a noticeable decrease in the speed of the system.

To prevent an early discovery by a user, it is possible to launch processes in the background, using the '&' shell script symbol.

Temp Files

To avoid speed reduction, script viruses may create temporary files. The viral code can be copied to these and any time-consuming routines can be run from there in the background. This way the process remains transparent to the user – there is no obvious decrease in the speed of the host application.

Another reason for making use of temporary files is to differentiate between the pure viral body and those files that are being infected. Some viruses copy the target file to the temp folder, modify it, and write back, replacing the now infected target file.

If there are errors, or corrupted files, it's easier to hide them by using a central, temporary, location than it is when working directly in the target file directory. Although error messages can be caught and redirected to null.

Bash allows redirection of the standard output to other files, by making use of '>'. Redirecting standard error output is possible also by using the '2>' symbol – for example, '2>/dev/null' (for sh and similar shells).

So a specific search selector could resemble the following:

```
… if [ "$(head –c9 $F 2>/dev/null)" = "#!/bin/sh"] …
```

This translates as: find files ($F), examine the first nine characters of the file and verify whether it is #!/bin/sh (the Bourne shell command line interpreter), while redirecting error messages to null.

To mark an infection, a simple, yet specific, marking can be used. Searching for the presence of an infection marker can also be done by using 'grep' or a similar technique as described above.

Infection Spectra

Unix shell viruses can:

- Prepend the viral code. Prepending viral code is pretty easy to do, the viral code is always executed. However, the drawback is that prepending viruses are easy to spot.
- Append the viral code. A simple tail –n 25 $0 >> target file will append 25 lines of the viral code to the target file. However, appending viral code might not always be called. If there's an error in the 'host' program, or it terminates with an exit code, the appended viral code won't be called. Usually script file code is executed from the beginning to the end of the file though, so both the host and the viral code will be called by the interpreter.
- Overwrite the target file with the viral code. Overwriting target files is, as such, already a rudimentary method but without proper file-type checking it may replace ELF-type binary files with ASCII-type script code.
- Insert the viral code somewhere inside the target file. This is more difficult for a user to detect, and might result in errors if certain host program code can't

complete (due, for example, to a crash by the inserted viral code).

- Create companion files. Usually the original file becomes hidden, the viral code takes the original host file name, while maintaining the same file attributes as the original host.
- Insert a call in the target file, in so doing leaving the real viral code in another file.
- Have encrypted/polymorphic code. Encrypting files can be easy. A simple ASCII-HEX conversion would make the code unreadable for most end users. ASCII to Hex conversions are possible using the 'printf' (\x123) command. Creating 'polymorphic' script viruses is pretty easy to do. One can insert random comments, or change variable names. Usually the random generator is supported, but other variables such as current date can be used as well.
- Use Sendmail. So far the use of Sendmail in Unix shell scripting malware is limited. In fact, this is quite remarkable as Unix systems can control mail programs often and are easy to call. A single line of code could call the program. Luckily, no successful Unix shell scripting mass-mailing worm has yet been encountered in the wild.
- Use another shell interpreter and recompile its code on a current system to avoid the incompatibility between its binaries and the operating system (see Unix/Cliph).
- Exploit security vulnerabilities in order to compromise the root account (see Unix/Cliph).

Sample 1: Unix/Zerto

This sample (filename elfo.sh) was included in a recent publication by a popular viral group.

The elfo.sh file starts with its identifier, marker (#;P) and Bourne shell interpreter, #!/bin/sh. Then the code performs a search on suitable files to check for the infection marker using:

```
[ -f $F ] && [ -x $F ] && [ "$(head -c3 $F)" != "#;P" ]
```

It searches for (-f) present, normal files that are flagged as (-x) executable and whose first three characters are not '#;P', thus checking that the specific file hasn't been infected already.

The virus takes the prepended viral code, the top 27 lines of the file, and copies the code to a tmp file. It then marks the file as executable/runnable and starts it.

Possible errors are redirected to null, thus hiding any error messages. The infector process runs in the background, this is mainly for speed considerations.

Host files are copied to the tmp directory and infected. Then the virus moves the tmp file back to the (now infected) host, and deletes the tmp file.

At this stage the viral script code should be prepended to an executable file, for example a shell script or ELF binary file.

However, when the virus sample that was provided was run on a *Linux RedHat 7.0* test system, a number of errors were produced.

Sample 2: Unix/Cliph

This backdoor sample (filename smlix.sh) came from a virus collection site and was discovered in August 2001. It is a *Linux* kernel 2.2.X (X<=15) & sendmail <= 8.10.1 local root exploit.

The malicious code starts with a reference to the shell command line interpreter '#!/bin/sh'. However, the code uses another shell interpreter in addition, namely tcsh: SHELL=/bin/tcsh.

The virus creates an anti-noexec library called 'capdrop.c' and attempts to compile it into a binary called 'capdrop.so'. Local recompilation is used to prevent problems that could be encountered when running binaries on different *Linux* distributions.

However, when the virus sample that was provided was run on a *Linux* test system, a number of errors were produced.

General Issues with Infecting

Creating a shell script virus sounds straightforward, but in practice a lot can go wrong during execution.

Apart from access rights, the viral file itself can sometimes be tricky to run successfully. One of the items that is overlooked sometimes is the exact end of the file. Without the new line symbol some viruses may fail to execute properly, resulting in errors.

Conclusion

Unix shell script viruses are relatively easy to create, yet powerful enough to create big problems.

Power users are likely to be alerted to malicious changes to their systems pretty quickly, but as more novice users migrate to popular *Linux* distributions such as *RedHat*, shell script malware may go unnoticed. More importantly, the novice users provide the less secure environments for malware to exploit.

At this stage, Unix shell script malware as such is more targeted at the specific machine – currently it doesn't spread its code to other machines natively. So far, it couldn't survive on its own.

Unix viral packages that have been successful have consisted of both binaries and scripts. However, there is no technical reason why Unix shell script malware cannot be successful in the future – it is a matter of proper coding combined with suitable (less secure) environments.