2022-11-10
: From the Linux command-line you want to search a directory structure of files and return the paths to all files that contain a particular string. What command do you use? (Because we all need daily #Linux #DFIR #CommandLine #Trivia)

2022-11-11
: Props to @feld for the first correct answer to yesterday's #Linux #DFIR #CommandLine #Trivia: "grep -rl string /path/to/files"

2022-11-11
: If you're enjoying my #Linux #DFIR #CommandLine #Trivia, check out our old content at https://blog.commandlinekungfu.com

2022-11-11
: From the Linux command line, give a count of the number of active processes per user, sorted in descending order by count. #Linux #DFIR #CommandLine #Trivia

2022-11-12
: Props to @istar_nil@twitter.com for a great answer to yesterday's #Linux #DFIR #CommandLine #Trivia:

  ```
  ps -ef | tail +2 | awk '{ print $1 }' | sort | uniq -c | sort -nr
  ```

  First off, *chef's kiss* to the "tail +2" to skip the initial header line in the ps output. I thought that was going to trip some people up!

  I liked this question because it also requires one of my favorite shell idioms, namely the "command line histogram": "...| sort | uniq -c | sort -nr". The first sort gets all the matching lines next to each other so that "uniq -c" can count them. The second "sort -nr" sorts the counts in reverse ("-r") numeric ("-n") order-- so descending by count numerically.

  The command line histogram is useful in so many situations. I guarantee it will come up in the answers to future daily trivia questions!

2022-11-12
: For today's #Linux #DFIR #CommandLine #Trivia I want a command to produce a sorted list of the executable paths for all running processes on the system.

2022-11-13
: The first and best answer to yesterday's #Linux #DFIR #CommandLine #Trivia comes from @Linkavych@twitter.com:

  ```
  readlink -f /proc/*/exe | sort
  ```

  The first lesson here is never be afraid to get the answers you need directly from /proc. This saves a lot of heartache compared to solutions that try to use "ps" or "lsof".

  Also, leverage your wildcards! You don't need "find" or a loop when your targets are at a predictable spot in a directory tree.

  Why did I want the output sorted? It's a whole lot easier to spot suspicious outliers when the output is sorted:

```
# readlink -f /proc/*/exe | sort
/dev/shm/.rk/src.sh (deleted)
/usr/bin/VGAuthService
/usr/bin/Xorg
...
```

Why hello there! How did you get to be on my system?

2022-11-13

Today's #Linux #DFIR #CommandLine #Trivia asks where does the command name "awk" come from? Shout out to @tliston who thinks you young whipper snappers don't even know what awk is.

2022-11-14

Despite pessimism from @tliston, lots of people checked in with the correct answer to yesterday's #Linux #DFIR #CommandLine #Trivia. The command name "awk" comes from the initials of its creators—Al Aho, Peter Weinberger, and Brian Kernighan.

@williballenthin had the best flex—he actually had a compiler design class taught by Aho.

Turns out Peter Weinberger and I went to the same undergrad. He's '69 and I'm '89 so I've met him a couple of times at CS reunion events. I was a ridiculous fanboy but he was very gracious.

2022-11-14

True or false: There is no way to arbitrarily set ctime or btime on an existing file in an EXT or XFS file system. #Linux #DFIR #CommandLine #Trivia

2022-11-15

Apparently the most confusing part of yesterday's #Linux #DFIR #CommandLine #Trivia was my wording of the question. Sorry, didn't mean to turn this into a CISSP exam!

The basic question was can you arbitrarily set ctime and btime in an EXT or XFS file system? The common wisdom is that you can't do this, but common wisdom is WRONG in this case as @djanatyn@twitter.com correctly pointed out.

If you want your EXT-based file to party like it's 1999, try this as root:

```
debugfs -w -R 'set_inode_field /path/to/file crtime 199901010000' /dev/disk/device
echo 2 > /proc/sys/vm/drop_caches
```

Change the file path and the disk device path as appropriate. You can also use "ctime" (or "mtime" or "atime") instead of "crtime" depending on which file timestamp you want to tweak.

The echo into /proc/sys/vm/drop_caches is necessary to actually see your change. Essentially debugfs is doing an end run around the usual file system calls, so the in-memory file cache isn't picking up your change. Do the echo command to flush all cached file data and force the inode data to be re-read from disk.

Btw, if you want to manipulate timestamps in XFS, use the "write" command in "xfs_db -x".

2022-11-15

Here's a fun #Linux #DFIR #CommandLine #Trivia question with a little #RedTeam flavor.

You find the following commands in /root/.bash_history:

```
lsof -c /ssh/ | awk '$5 == "unix" && $NF != "socket" {print $3, $NF}'
export SSH_AUTH_SOCK=/tmp/ssh-iU9kC9BVaa5c/agent.1296
ssh-add -l
cat ~shelby/.ssh/known_hosts
ssh shelby@foo.example.com
```

What happened and what do you do about it?

2022-11-16

Props to @ilikepi for checking in with the first correct answer to yesterday's Linux DFIR command line trivia.

Let's break this down line by line:

1. The lsof pipeline grabs the username and path name of all ssh-agent sockets as well as those supported by agent forwarding through sshd.

2. SSH_AUTH_SOCK is the environment variable where your SSH clients find the socket path name to communicate with ssh-agent. The root user can read/write any user's SSH_AUTH_SOCK.

3. Does the ssh-agent you're talking to contain keys? "ssh-add -l" will tell you.

4. Which machines is this user normally SSHing into? Check their known_hosts file for machines that are likely to be accepting your target user's key.

5. Try to SSH as that user into one of the machines from the user's known_hosts file.

We've already seen successful attackers using automated scripts to pillage the victim system for SSH identity certificates and known_hosts files. Leveraging ssh-agent processes is likely to be the next escalation.

What do you do in this case? You can certainly consider the local machine compromised-- spin up your Linux IR playbook. You'll need to check the logs on foo.example.com to see if the shelby login was successful. If yes, then that machine is compromised too and you'll be looking hard for successful privilege escalation.

While this attack does not actually compromise shelby's private key(s), I would still recommend that shelby burn any keys in this ssh-agent process and create new ones. shelby should also look for attackers adding their own keys into shelby's authorized_keys files.

Also, please enable the HashKnownHosts option on all your systems. This will make it harder for attackers to get a list of machines to target.

#Linux #DFIR #CommandLine #Trivia

2022-11-16

Give me a Linux pipeline to output the 10 longest lines in a file.

#Linux #DFIR #CommandLine #Trivia

2022-11-17

Yesterday's Linux DFIR command line trivia asked for a Linux pipeline to output the 10 longest lines in a file.

You're going to need a loop to do this. You could go with an explicit loop, like @secshoggoth did:

```
cat filename | while read line; do echo $(echo $line | wc -c) $line; done | sort -nr | head
```

Or you could implicitly loop over each line of the file using awk, which is what @ajk, @knarphie, and @jwmwi@twitter.com suggested:

```
awk '{print length(), $0}' filename | sort -nr | head
```

Add "cut -d' ' -f2-" to the end of either pipeline to remove the line lengths from the output.

Aside from being shorter to type, the awk solution is MUCH faster than the explicit loop. The main culprit here is the "$(echo line | wc -c)" inside the loop. That construct requires spawning a new shell and the "wc" command each time through the loop, which is a big performance drag.

So why do I care about this pipeline from a DFIR perspective? Because I find it very useful for finding web exploit attempts in my web server logs:

```
# awk '{print length(), $0}' access.log | sort -nr | head | cut -f2- -d' '
192.168.210.131 - - [05/Oct/2019:13:01:29 +0200] "POST /jabc/?q=user/
password&name%5b%23post_render%5d%5b%5d=passthru&name%5b%23markup%5d=php%20-
r%20%27eval%28base64_decode%28Lyo8P3BocCAvKiov[...snip...]
```

Bonus points if you have read this far and can identify the exploit being used based on the snipped output provided above.

#Linx #DFIR #CommandLine #Trivia

2022-11-17

Give me Linux command to locate files larger than 500MB in a directory structure.

#Linx #DFIR #CommandLine #Trivia

2022-11-18

Lots of folks chimed in with the correct answer to yesterday's Linux DFIR command line trivia, but @xuf was quickest off the mark.

```
find /path/to/dir -type f -size +500M
```

This answer demonstrates a couple of "find" conventions. "+" and "-" in front of numbers generally mean "greater than" and "less than", respectively. Also, "find" recognizes units like "M" for megabytes, "k" for kilobytes, and so on.

Large files are pretty uncommon in Linux. So this "look for large files" idiom is useful for finding collections staged for exfiltration, hidden logs and PCAPs, and other things that "just don't belong".

#Linux #DFIR #CommandLine #Trivia

2022-11-18

You find a string of interest at a specific byte offset in your XFS file system image. How can you convert this byte offset into an XFS block address using only the command line?

#Linux #DFIR #CommandLine #Trivia

2022-11-19

Wow, yesterday's Linux DFIR command line trivia scared everybody off!

Let's start with this string:

```
6240111554 # I love bash!
```

Because, after all, who doesn't?

```
xfs_db -r -c "convert daddr $((6240111554 / 512)) fsblock" /path/to/image
```

Let's start in the middle where the "$((...))" syntax converts the byte offset into the 512 byte sector offset. XFS refers to the sector offset as the "daddr" (direct address). The "convert" function in xfs_db lets you switch between daddr and any other address type, but "fsblock" is probably most useful. "fsblock" is the standard XFS packed address format.

Note that you must include the image file (or disk device) in the command. That's because XFS uses variable sized allocation groups and address lengths based on the size of file system. You can't do the conversion without specific data fields from the file system superblock.

Allocation groups? Variable length addresses? Packed addresses? Oh yeah, XFS is all kinds of fun. For more details see my series of articles breaking down XFS at https://righteousit.wordpress.com/tag/xfs/

#Linux #DFIR #CommandLine #Trivia

2022-11-19

Linux audit.log files contain lines like:

```
type=PROCTITLE msg=audit(1584141754.020:42):
proctitle=2F7362696E2F6D6F6470726F6265002D71002D2D0069707461626C655F66696C746572
```

The "proctitle=" value is a hex-encoded command line with null-terminated strings. What command line will let you see the original command line?

#Linux #DFIR #CommandLine #Trivia

2022-11-20

I admit that I was curious which way people would go on yesterday's Linux DFIR command line trivia.

One possible answer (props to @linkavych for chiming in first) is to simply use "ausearch -i" to decode the hex strings. ausearch has plenty of other useful features as well, including representing the epoch timestamps as human-readable dates and gathering related messages using audit ID numbers.

Another approach, however, is to decode the hex more directly (@SaThaRiel went in this direction):

```
$ echo 2F7362696E2F6D6F6470726F6265002D71002D2D0069707461626C655F66696C746572 |
xxd -r -p | tr \\\\000 ' '; echo
/sbin/modprobe -q -- iptable_filter
```

"xxd -r" is the reverse hex dumper, taking hex ASCII and converting it back into it's binary representation. In this case, that's a string of characters. Then I'm using "tr" to convert the null-terminated strings to space-terminated strings. The extra "echo" at the end adds a newline and makes the results much more readable.

Shout out to @timb_machine for sending us off in to Perl land. I'll let you read the replies for that zaniness. I couldn't possibly do it justice here.

#Linux #DFIR #CommandLine #Trivia #Perl

2022-11-20

How can you get information about a user's last login to the system, even if that login pre-dates the current wtmp and security/audit logs on the system?

#Linux #DFIR #CommandLine #Trivia

2022-11-21

The answer to yesterday's Linux DFIR command line trivia is the "lastlog" information. It is stored in /var/log/lastlog and can be read with the "lastlog" command. This is also the source of the information about your last login that you see whenever you log into the system.

From a DFIR perspective, this information is sometimes interesting if a user's last login was so long ago that it predates all of the normal logging information on the system. Syslog, wtmp, and audit.log info all has a limited lifetime depending on your log retention/rotation policies.

/var/log/lastlog can be a tricky artifact to interpret from a forensic drive image. It's a sparse file with the last login records being found at offsets that are based on the numeric UIDs of the users from the image's /etc/passwd file. Also the format of the lastlog records has changed often over the history of Linux, so you need to know which record version you are dealing with. Also the records are of different sizes depending on whether you are dealing with a 32-bit or 64-bit system.

So if you can get the lastlog output from the live system, it definitely makes your life easier. Failing that, try a generic lastlog parser like https://github.com/tigerphoenixdragon/lastlog_parser

#Linux #DIFR #CommandLine #Trivia

2022-11-21

You discover a suspicious process when conducting an investigation on a live Linux system. How can you extract strings from the memory of the running process?

#Linux #DFIR #CommandLine #Trivia

2022-11-22

Wow, when I posed yesterday's Linux DFIR command line trivia, I had forgotten what a pain this was to actually accomplish. Let me give the solution first and then explain what the heck is going on:

```
cat /proc/<pid>/maps | tr - ' ' |
while read start end junk; do
start=$((16#$start / 4096));
dd if=/proc/<pid>/mem bs=4096 skip
=$start count=$((16#$end / 4096 - $start))
done 2>/dev/null | strings -a >strings.txt
```

/proc/<pid>/maps is a breakdown of the mapped memory regions in a given process. Each line begins with the starting and ending addresses (in hex) of the mapped memory region.

Normally you would just seek() to the start address and dump the requisite number of bytes. Unfortunately, there's no seek() operation built into bash. So we use "dd" instead.

We know the memory regions will be page-aligned and memory pages are 4K (4096 bytes) big. So I'm converting all the addresses so that dd will read in 4K chunks ("bs=4096") for efficiency. Trust me, dumping one byte at a time is truly painful.

So reading from /proc/<pid>/mem, "skip=$start" becomes the equivalent of seek(). We calculate how much data to dump ("count=") by subtracting the start address from the end address.

Note the "2>/dev/null" at the end of the loop, to throw away all the noise from the "dd" commands. This includes errors when I try to read non-readable memory regions, because I'm not bothering to pay attention to the readability flag in the /proc/<pid>/maps output.

The non-error output gets thrown into "strings -a" and ... done! Easy-peasy lemon squeezy, right?

Oy.

#Linux #DFIR #CommandLine #Trivia

2022-11-22

Why does the "dd" command have such a different command line syntax compared to other Unix/ Linux commands?

#Linux #DFIR #CommandLine #Trivia

2022-11-23

Damn, some of you all are old school mainframe types! Yesterday's Linux DFIR command line trivia asked why the "dd" command has such a different syntax from other Unix/Linux commands.

@z0vsky and @knu jumped on board with the correct answer. "dd" was deliberately paying homage to (some would say spoofing on) the IBM JCL DD command.

Don't believe me? Here's an old USENET posting where Dennis Ritchie lays down the law. It must be considered canon at this point.

https://groups.google.com/g/alt.folklore.computers/c/HAWoZ8g-xYk/m/HDUVxwTVLKAJ

#Linux #DFIR #CommandLine #Trivia #Unix #Mainframe #History

2022-11-23

You've got a bash_history file which includes timestamps. Write a Linux shell pipeline to output a human readable time/date stamp followed by the corresponding command on a single line, e.g.

yyyy-mm-dd hh:mm:ss command line

#Linux #DFIR #CommandLine #Trivia

2022-11-24

Did anybody have to do research for yesterday's Linux DFIR command line trivia? If you've never seen a timestamped .bash_history file before the format is:

```
#<epoch timestamp>
command
```

Each command is preceded by a comment line that contains the Unix epoch time timestamp (seconds since Jan 1, 1970) for when the command was executed. History timestamping is enabled when you set the HISTTIMEFORMAT environment variable in bash. I would recommend globally setting this variable in /etc/profile for all servers you control. You will thank me the next time you have to do investigations on your systems!

Here's a solution that outputs a human readable timestamp on the same line as the corresponding command:

```
cat .bash_history | while read line; do
[[ $line =~ ^# ]] && \\
echo -ne $(date -d @${line/\\#/} "+%F %T\\t") || \\
echo $line
done
```

The "[[...]] && ... || ..." is a quick and dirty way of doing an "if ... then ... else ..." on the command line. Here I am using the bash regex matching operator (did you know bash could do that?) to see if the line starts with a hash mark ("#").

If yes, then it's a date we need to convert and reformat (otherwise it's just a command line that we output with "echo $line"). You can convert epoch dates with "date -d @<epoch>" but first we have to remove the leading "#".

I'm using another bash trick here-- the variable substitution operator-- to remove the "#". The syntax is "${variable/match/substitute}". So I am matching the "#" and replacing it with nothing, getting rid of it entirely. By the way, I have to backwhack the "#" here because "${var/#pattern/sub}" normally means "match pattern at the beginning of $var" (like "s/^pattern/sub/" in sed). But in this case I want to match the literal "#".

OK, so we can strip out the "#" and convert the remaining epoch date with "date -d @<epoch>". The "+%F %T\\t" specifies the output format we want: "YYYY-MM-DD hh:mm:ss\\t". That trailing "\\t" is going to become a tab space to give some separation between the date and the command.

I've wrappered this up in "echo -ne $(date ...)". "-n" means don't output a newline, so the command output I print out next will be on the same line as the date. "-e" means interpret the "\\t" as a tab character to provide our spacing between the date and the command.

Short-circuit logical operators, regex matching, and variable substitution. Bash is fun!

#Linux #DFIR #CommandLine #Trivia

2022-11-24

Give me a Linux command line that outputs the path names of any regular files found under /dev. Why is this useful in an investigation?

#Linux #DFIR #CommandLine #Trivia

2022-11-25

Apparently the relative simplicity of yesterday's Linux DFIR command line trivia threw some people off. Hey, it's a holiday here in the USA and we all deserve a bit of a break!

How do you find regular files under /dev? How about:

```
find /dev -type f
```

Why is this useful?

```
# find /dev -type f
/dev/shm/.udevd/src.sh
```

With a few exceptions depending on your distro and version, files under /dev should be device files. So when an attacker stages their files under /dev, this little find command makes them pop right out.

Shouts to @1e0aaab9ec6c49027d7c2282fcc8d4 and @pauldokas who chimed in at nearly the same time with the correct answer.

#Linux #DFIR #CommandLine #Trivia

2022-11-25

The primary superblock in your EXT file system image is corrupted. How can you use the command-line to locate a backup superblock and how can you mount the file system using that information?

#Linux #DFIR #CommandLine #Trivia

2022-11-26

The first part of yesterday's Linux DFIR command line trivia asked how to find an alternate superblock if the primary superblock had become corrupted. @oliverwiegers chimed in with my preferred method, "mkfs -n":

```
# mkfs -n linux.raw
mke2fs 1.42.9 (28-Dec-2013)
linux.raw is not a block special device.
Proceed anyway? (y,n) y
[... snip ...]
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
4096000
```

The "-n" means "show what you would to but don't actually make the file system". At the end of the output, you get a nice listing of the backup superblock locations.

As for mounting the file system, the "mount" command has a "sb=" option to specify an alternate superblock. However, "sb=" wants the superblock offset in 1K units rather than the normal 4K blocksize. So we need to do a little math:

```
mount -t ext4 -o ro,noexec,noload,loop,sb=$((4096000*4)) linux.raw /mnt/test
```

"noexec" helps prevent you accidentally running malware from the mounted image. "noload" allows you to ignore an underplayed records in the file system journal.

#Linux #DFIR #CommandLine #Trivia

2022-11-26

A couple of days ago @nf3xn was riffing on some other "find" command ideas and that suggested one of my favorite "find" hacks...

Over the holiday weekend, you get an IDS alert about beaconing activity from one of your Linux servers starting on Nov 25, 2022 at 11:32pm. How can you use "find" to locate all files that have been modified since this precise date and time?

#Linux #DFIR #CommandLine #Trivia

2022-11-27

One of the best parts of running these daily Linux DFIR command line trivia questions is when I learn something new!

Yesterday's question asked how to find all files that have been modified since a specific date and time. Old school me would do this by:

```
touch -t 202211252332 /tmp/myfile
find / -newer /tmp/myfile
```

First create a file using touch that has the specific mtime we want. Then use "find ... -newer ..." to locate all files with newer mtimes. This is a great DFIR hack that has served me well countless times. @crash0ver1d3 and @ilikepi checked in with this solution too.

But then I heard about the "-newermt" option from @barubary and @linkavych which lets you specify the specific timestamp as part of the find command:

```
find / -newermt '2022-11-25 23:32:00'
```

Awesome!

#Linux #DFIR #CommandLine #Trivia #TIL

2022-11-27

Give me a command-line for locating all immutable files in a file system.

#Linux #DFIR #CommandLine #Trivia

2022-11-28

Yesterday's Linux DFIR command line trivia asked for a command line to locate all immutable files in a file system. Interestingly, the answer does not involve the "find" command!

Turns out there is no "find" option to look for extended attributes like immutable. Fortunately, "lsattr" has a recursive ("-R") option. @timb_machine checked in with one correct answer:

```
lsattr -Ra / 2>/dev/null | grep '^....i'
```

The immutable flag is listed in the fifth position in the standard "lsattr" output.

@t_pageflt suggested adding the "-l" flag to "lsattr" so we could just search on the word "Immutable":

```
lsattr -laR / 2> /dev/null | grep Immutable
```

Cool idea, but will match on file paths containing the word "Immutable". Not a likely false-positive, so good solution.

@furicle suggested just doing "rm -rf /" and looking at whatever was left over. While this might be effective, it's a bit hard on your operating system! 🙂

#Linux #DFIR #CommandLine #Trivia

2022-11-28

Extract all unallocated blocks from an XFS file system into a single file.

#Linux #DFIR #CommandLine #Trivia

2022-11-28

@xabean Use "ls -l /proc/PID/fd" to spot the link pointing to the deleted file. Then "cp /proc/PID/fd/LINKNO /tmp/recovered-file". Note that this will make a "point in time" copy of the file-- if more data is added to the file later, your copy won't get that.

It's interesting to note, however, that "tail -f" works on these links as well, so you could do "tail -f -n +0 /proc/PID/fd/LINKNO >/tmp/recovered_file"

#Linux #DFIR #CommandLine #Trivia

2022-11-29

Yesterday's Linux DFIR command line trivia was looking to extract unallocated blocks from an XFS file system. The trick here is to figure out that you can get a list of unallocated extents from the "freesp -d" command in "xfs_db".

Here's what the output looks like:

```
xfs_db> freesp -d
agno agbno len
0 4 1
0 5 1
0 6 1
0 7 1
0 756 4
[... snip ...]
3 331529 1
3 336728 299
3 337109 318251
from to extents blocks pct
1 1 62 62 0.00
2 3 30 76 0.01
4 7 36 198 0.02
[... snip ...]
```

The first part of the output is the list of free extents. Each line gives the Allocation Group (AG) number, the block offset within the AG, and the number of blocks in the free extent. After the free extents list, there is a histogram which we can ignore.

So we need to extract the numeric data between the two header lines. But in order to dump the extents, we also need to know the size of each AG. This is data we can get from the file system superblock:

```
agblocks=$(xfs_db -r -c 'sb 0' -c 'print agblocks' xfs_image.raw | cut -f3 -d' ')
```

And with that data, we can then extract the unallocated extents:

```
xfs_db -r -c 'freesp -d' xfs_image.raw | while read agno agbl len; do
[[ $agno == "agno" ]] && continue
[[ $agno == "from" ]] && break
```

```
dd if=xfs_image.raw bs=4096 skip=$(($agblocks * $agno + $agbl)) count=$len
done 2>/dev/null | gzip >unallocated_blocks.gz
```

Why is this useful? Many times we need to go looking for deleted data. No point in searching allocated blocks for deleted file data. So speed up your file carving, string searching, or whatever by first extracting just the unallocated collection like we are doing here.

#Linux #DFIR #CommandLine #Trivia #XFS #FileSystem

2022-11-29

Sort a file of IPv4 addresses numerically by octets and remove duplicate IPs.

#Linux #DFIR #CommandLine #Trivia

2022-11-30

Yesterday's Linux DFIR command line trivia asked you to sort and uniquify a file of IPv4 addresses. And if "uniquify" is wrong, I don't want to be right!

The old school way to numerically sort on the octets is:

```
sort -nu -t. -k1,1 -k2,2 -k3,3 -k4,4 addrfile >addrfile.sorted+uniq
```

"-n" means numeric sorting, "-u" is output only unique lines, "-t." specifies "." as the field delimiter. Then we sort on each of the octets. Props to @ajk for being the first old Unix hand to check in with this solution.

Of course those crazy Linux kids and their new-fangled thinking have a different way of dealing with this:

```
sort -V -u addrfile >addrfile.sorted+uniq
```

"-V" is the "version sort" option, intended to sort software version numbers like 6.0.10 and 5.15.80. But it also works great on IPv4 addresses. @xabean and @cybeej checked in with the "-V' option, but curiously both chose to pipe into "uniq" rather than using "-u". Clearly, they're conflicted.

#Linux #DFIR #CommandLine #Trivia

2022-11-30

Output lines from /etc/passwd that have duplicate UIDs.

#Linux #DFIR #CommandLine #Trivia

2022-12-01

Oh my but yesterday's Linux DFIR command line trivia generated lots of different solutions! Lots of awk and even some sqlite3!

Let's break this down into pieces. First, we can figure out the duplicate UIDs in the passwd file:

```
cut -f3 -d: passwd | sort | uniq -d
```

Split out the third colon-delimited field from the passwd file, sort the lines, and then use "uniq -d" to output one copy of each of the duplicates.

But what to do with this information? We could write it out to a temporary file, which is the approach taken by some of the suggested solutions. But bash has a handy little "<(...)" syntax to avoid having to do this:

```
grep -f <(cut -f3 -d: passwd | sort | uniq -d) passwd
```

"<(...)" allows me to take the output of one pipeline and make it "file input" in another command. Here I'm using my duplicate UID list as a "file" of patterns for "grep -f".

Of course there's a problem here. Suppose that one of my duplicate UIDs was UID zero. My grep statement above would be matching zeroes anywhere in each passwd file line, not just the UID field. For example, it would match UIDs or GIDs that merely contained a zero, like 1000.

So we need to tweak up our input patterns a little bit:

```
grep -f <(cut -f3 -d: passwd | sort | uniq -d | awk '{print "^[^:]*:[^:]*:"$0":"}') passwd
```

Or if you prefer an awk-free solution:

```
grep -f <(cut -f3 -d: passwd | sort | uniq -d | xargs -I{} echo "^[^:]*:[^:]*:{}:") passwd
```

Either way, we're creating a pattern using our duplicate UID list that forces each UID to match against the full UID field in the passwd file. Thanks to @smix@fosstodon.org for pointing out I need the "^" in my regex!

Whee! Fun!

#Linux #DFIR #CommandLine #Trivia

2022-12-01

You have a directory of photos with file names like "0105.jpg"-- all files end in .jpg but the numbers are unique. Some of the photos have been deleted, leaving gaps in the numbering. Output the numbers of the missing photos.

#Linux #DFIR #CommandLine #Trivia

2022-12-02

Props to @smix@fosstodon.org for checking in with the solution that I was going to suggest for yesterday's Linux DFIR command line trivia:

```
for n in {0000..9999}; do [ -f $n.jpg ] || echo $n; done
```

"{0000.9999}" is an example of bash's range operator. Because the leading term starts with a zero, you will get zero-filled numbers like "0105", "0106", and so on.

Inside the loop, we're using the short-circuit "||" operator like an "if" statement. If "$n.jpg" exists then move on to the next number, otherwise output the missing file name.

#Linux #DFIR #CommandLine #Trivia

2022-12-02

Sort a nested directory structure of files by file size, outputting the file paths in descending order by size.

#Linux #DFIR #CommandLine #Trivia

2022-12-03

Here I am getting schooled again by @darkvinge's answer to yesterday's Linux DFIR command line trivia question.

How do you output an entire directory structure of file paths, sorted in descending by size? Old-school me would have told you to do something like this:

```
find /some/path -type f -exec wc -c {} \\; | sort -nr
```

It's an interesting example of one time when you want to use "find ... -exec" rather than "find ... | xargs ...".

But then @darkvinge pointed out that you can just use "-printf" instead of "-exec":

```
find /some/path -type f -printf "%s\\t%p\\n" | sort -nr
```

"-printf" has a huge number of handy shortcuts for different file parameters. Here we see "%s" for file size and "%p" for file path.

This is vastly more efficient than executing "wc -c" on each file. Don't believe me? Check out how long the different find commands take on the same directory structure:

```
# time find /usr -type f -exec wc -c {} \\; >/dev/null
real 4m20.615s
user 2m47.897s
sys 1m37.520s
```

```
# time find /usr -type f -printf "%s\\t%p\\n" >/dev/null
real 0m0.306s
user 0m0.068s
sys 0m0.238s
```

It's not even remotely close.

#Linux #DFIR #CommandLine #Trivia

2022-12-03

Output the directory name of any directory that contains files with image extensions like .jpeg, .jpg, .gif, .png, .tiff, etc. Extension matching should be case-insensitive but you must output the file paths without case changes.

#Linux #DFIR #CommandLine #Trivia

2022-12-04

Looks like we're continuing the theme of "Hal gets schooled in the finer points of GNU find" in our daily Linux DFIR command line trivia. But I am all about the learning, so let's get down to it!

Yesterday's question was basically to print out all directories that contain files with image extensions like .jpeg, .jpg, .gif, and so on. My old-school pipeline-based solution would be:

```
find /some/path -type f | grep -Ei '\\.(jpe?g|gif|png|tiff|bmp)$' | xargs dirname | sort -u
```

Find outputs the file paths. We use "grep -Ei" (aka "egrep -i") to match the file extensions in a case-insensitive manner. "xargs dirname" pulls out the directory paths efficiently. "sort -u" to just get the unique directory paths.

But then here comes @LinuxAndYarn to take me back to school:

```
find /usr/share -regextype posix-egrep -type f -iregex '.*\\.(jpe?g|gif|png|tiff|bmp)$' -printf "%h\\n" |
sort -u
```

The "-iregex" operator replaces "grep -Ei", or at least it does once I specify "-regextype posix-egrep". The tricky part is that "-iregex" wants to match the ENTIRE PATH, so we add a ".*" at the front of the regex.

The "-printf" statement means that we don't need "xargs dirname". "%h\\n" means just output the file paths without the file names.

If you're on a non-Linux system that doesn't have GNU find, then my pipeline solution is still useful. But, wow, GNU find has some fun little features built in.

#Linux #DFIR #CommandLine #Trivia

2022-12-04

Ping a /24 network and output the IP addresses of all currently live systems. Some hosts on the network do not respond to broadcast pings.

#Linux #DFIR #CommandLine #Trivia

2022-12-05

Yesterday's Linux DFIR command line trivia asked how you can ping out a /24 network.

@apgarcia weighed in with:

```
nmap -sP <subnet>/24
```

And @damien contributed

```
fping -s -g <prefix>.1 <prefix>.254
```

Both of these are correct. However, my rules are that I limit myself to tools that I am likely to find on any Linux system I go to. Nmap is pretty common, but not always there.

@apgarcia and @darkvinge were both circling around a solution that only used ping and shell built-ins. Putting the best parts of their solutions together with a little extra sed flourish of my own and we get:

```
for o in {1..254}; do ping -c1 -w1 </24 prefix>.$o; done | grep from | sed 's/.* from \\([^:]*\\):.
*/\\1/'
```

"ping -c1 -w1" is the quickest way to ping out the hosts-- send one ICMP packet and wait a max of one second for the reply.

Output from a successful ping looks like this:

```
PING 192.168.10.131 (192.168.10.131) 56(84) bytes of data.
64 bytes from 192.168.10.131: icmp_seq=1 ttl=64 time=0.111 ms
```

```
--- 192.168.10.131 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

So we can key in on the "from" lines. The sed shortens the line to just the responsive IP address.

#Linux #DFIR #CommandLine #Trivia

2022-12-05

Output the start date and time for all processes on the system in 'YYYY-MM-DD hh:mm:ss' format.

#Linux #DFIR #CommandLine #Trivia

2022-12-06

The answers to yesterday's Linux DFIR command line trivia remind me to always check my assumptions.

The problem was to output the start date/time for all processes on the system in 'YYYY-MM-DD hh:mm:ss' format. I realize now that I didn't specify that I also wanted the process name and/or command-line. Assumption number one for me-- don't you people read my mind by now?

Early responses focused on using the "lstart" output from "ps". After all, this is how you get the full start time of each process, so it's a reasonable assumption that this is what you should use. However, lstart gives a very unhelpful output format: "Tue Dec 6 07:23:32 2022". People who went down this rabbit hole spent way too much time trying to reformat that date into something close to what the question asked for.

What if we went a different way. "ps" also supports "etimes" output-- elapsed seconds since the process started. If we get the current time in Unix epoch format, and then do a little subtraction with etimes, we can figure out the process start time:

```
epoch=$(date +%s)
ps -eo etimes,cmd | while read sec cmd; do
echo -e $(date -d @$(($epoch - $sec)) "+%F %T")\\\\t$cmd
done
```

I was feeling pretty good about that solution until @stass came along and suggested an entirely different solution that used /proc instead of "ps". And I realized I had been assuming that "ps" was the only way to get this done. @stass' solution output PIDs instead of command lines, so here's a slightly modified version of the original idea:

```
for file in /proc/[0-9]*/cmdline; do
echo -e $(stat -c '%y' $file | cut -f1 -d.)\\\\t$(cat $file | tr \\\\000 ' ');
done
```

This isn't perfect, because some of the command lines are empty. But we can add some logic to clean that up:

```
for file in /proc/[0-9]*/cmdline; do
cmd=$(cat $file | tr \\\\000 ' ')
[[ -z "$cmd" ]] || echo -e $(stat -c '%y' $file | cut -f1 -d.)\\\\t$cmd
done
```

[Late edit: since posting this answer, further discussion seems to indicate that timestamps on objects in

/proc are not reliable markers for process start time in all versions of Linux. Neat idea, possibly not reliable-- go with the "ps" solution above.]

That's the thing about Linux/Unix-- there's always more than one way to do it. And maybe the way you thought of first isn't always the best way to go about it.

#Linux #DFIR #CommandLine #Trivia

2022-12-06

Today's Linux DFIR command line trivia comes to us from @funkybuddha!

Two directories are "duplicates" if they both contain the same file/directory names (file content, timestamps, etc don't matter). Output all such duplicate directories in a given directory structure-- only the directory names!

#Linux #DFIR #CommandLine #Trivia

2022-12-07

Yesterday's Linux DFIR command line trivia was a request from @funkybuddha.

Two directories are "duplicates" if they both contain the same file/directory names (file content, timestamps, etc don't matter). We want to output the names of all duplicate directories.

We need some way to quickly compare two directories, and my impulse here is to use the file names in each directory as input to a hashing function. Which hashing function you use doesn't really matter, but I went with md5:

```
$ ls -A /etc | md5sum
baa275801f1097e83e827dc771914274 -
```

Now we need to do that for a whole bunch of directories:

```
find /usr/share -type d | while read dir; do echo -e $(ls -A "$dir" | md5sum)\\\\t$dir; done | sort | uniq -D
-w32
```

The trick here is that "uniq" doesn't have to use the entire line. "-w32" means only consider the first 32 characters when deciding if we have a duplicate. "-D" means show all duplicate lines.

In this case I think leaving the hashes in the output is useful because it lets you see the groupings of different duplicate directories. You could throw a "... | cut -f2" at the end of that pipeline to get rid of them.

Or we could throw some more shell code at the problem:

```
find /usr/share -type d | while read dir; do echo -e $(ls -A "$dir" | md5sum)\\\\t$dir; done | sort | uniq -D
-w32 |
while read hash junk dir; do
[[ $hash != $lasthash ]] && echo =====
lasthash=$hash
echo $dir
done
```

That will output a line with "=====" between each group of duplicate directories.

I could do this all day!

#Linux #DFIR #CommandLine #Trivia

2022-12-07

Looking at SSH private keys in a user's .ssh directory, how can you tell which ones are protected with a passphrase and which are not?

#Linux #DFIR #CommandLine #Trivia

2022-12-08

Yesterday's Linux DFIR command line trivia asked how you can tell if an SSH key file is encrypted or not. While some key formats have a header that declares the file is encrypted, you can't generally count on that across all key formats and all versions of OpenSSH.

@linearchaos was the first to check in with a correct answer:

ssh-keygen -y -P '' -f keyfile

"ssh-keygen -y" was designed to allow people to check their passphrases. The "-P """ means we're passing in an empty passphrase. If "keyfile" is not encrypted, then ssh-keygen will print out the public key associated with the file. If the file is encrypted, then you get an error message.

If you want to use this in a script, running this command against an encrypted keyfile exits with a non-zero value, which is typical for commands that exit with an error. Running it against an unencrypted file returns the usual zero exit code for a successful command.

#Linux #DFIR #CommandLine #Trivia

2022-12-08

/tmp/evil-bash is a set-UID copy of the bash shell owned by the root user. From your normal user account you "exec /tmp/evil-bash". What is your privilege level now?

#Linux #DFIR #CommandLine #Trivia

2022-12-09

Yesterday's Linux DFIR command line trivia asks what is your privilege level after executing a set-UID root copy of the bash shell? @deeseearr checked in with a very complete answer, which I encourage you to read, but allow me to summarize.

The short answer is that you will not end up getting root privileges. The bash shell knows the difference between your real UID (that you logged in with) and your effective UID (granted by the set-UID bit) and puts you right back to your real UID, refusing to operate at the higher privilege level.

The reason for this behavior backs down to a race condition when executing set-UID scripts. This race condition is unsolvable given the current way Unix/Linux execute interpreted code. Since it's unsafe, bash and AFAIK every other interpreted language in Unix/Linux refuse to operate set-UID under normal circumstances.

However, since any feature you can't turn off is a bug, you can execute a set-UID bash shell with the "-p" (privilege) option and it will honor the set-UID bit and give you elevated access.

#Linux #DFIR #CommandLine #Trivia

2022-12-09

Give me a Linux command line to locate all set-UID files that are NOT in a directory named bin, sbin, or libexec.

#Linux #DFIR #CommandLine #Trivia

2022-12-10

Yesterday's Linux DFIR command-line trivia asked how to locate set-UID files that were not in directories named bin, sbin, or libexec. I use this during investigations to look for set-UID programs attackers may have staged in non-standard directories.

Finding the set-UID programs is straightforward:

```
find / -type f -perm -04000
```

The set-UID bit is the 4000 bit in octal notation, and the leading "-" means the file must match on this bit pattern but may also have other bits set.

By the way, you can also do this match with symbolic notation:

```
find / -type f -perm /u+s
```

Now the question is how do we eliminate the set-UID programs in the bin, sbin, and libexec directories? Replies were pretty evenly split. From an expediency standpoint, I probably would go with the "pipe to egrep" route myself:

```
find / -type f -perm /u+s | grep -Ev '/(s?bin|libexec)/'
```

The alternative is to use the "-prune" operation in find to eliminate the unwanted directories:

```
find / -regextype posix-egrep -regex '.*/(s?bin|libexec)' -prune -o -type f -perm /u+s -print
```

Note that your regex needs to match the whole path, so we put a ".*" at the front. Also, you must have a "-print" at the end there or else you will end up printing both the set-UID files and the directories you pruned. Find is weird.

#Linux #DFIR #CommandLine #Trivia

2022-12-10

Since we were talking about set-UID executables in the last trivia challenge, let's keep the theme going...

What does set-GID mean when set on a directory?

#Linux #DFIR #CommandLine #Trivia

2022-12-11

Yesterday's Linux DFIR command line trivia asked what is the effect of the set-GID bit on a directory? Props to @rory for checking in first with the correct answer!

Set-GID on a directory means that new files created in that directory will be group-owned by the group owner of the directory by default. This is different from the normal default where new files will end up inheriting the default group ownership of the user creating the file.

Set-GID on directories was defined this way to help group projects share files easily via a common group. Folks have mostly forgotten about this behavior because these days much of our file sharing for group projects is done remotely via source code management systems like git.

#Linux #DFIR #CommandLine #Trivia

2022-12-11

One more directory permissions question and I promise I'll stop...

What is the meaning of the "sticky bit" on a directory? What was the original meaning of the sticky bit when applied to files?

#Linux #DFIR #CommandLine #Trivia

2022-12-12

Yesterday's Linux DFIR command line trivia asked about the meaning of the "sticky bit" on directories. Props to @timb_machine for checking in first with the correct answer.

Normally, anybody with write access to a directory would be allowed to delete files in that directory. But that would mean in /tmp or any other world-writable directory that anybody could remove anybody else's files.

Setting the "sticky bit" on the directory means that only the owner of the file is allowed to remove that file. You'll see the "sticky bit" set on /tmp and /var/tmp on your Unix/Linux systems:

```
$ ls -ld /tmp /var/tmp
drwxrwxrwt 5 root root 4096 Dec 11 06:45 /tmp
drwxrwxrwt 2 root root 4096 Jul 6 09:53 /var/tmp
```

The "sticky bit" is the "t" at the end of the permissions vector, or the 01000 bit in absolute mode.

However, in the early Unix era, "sticky" was applied to executables, and not directories. This is why you see it represented symbolically as a "t" in the "execute for other" spot.

In those olden times, "sticky" indicated to the kernel that the executable should "stick around" in memory rather than being immediately cached out. For commonly run programs, this meant not having to reload them from disk each time. External storage was much slower in those days and having to reload the "ls" command every time was pretty painful.

Of course as disks got faster this wasn't so much an issue anymore. So 4.3 BSD decided to introduce the current directory semantics to guard against race conditions and other bad behavior in globally writable directories.

#Linux #DFIR #CommandLine #Trivia

2022-12-12

Last week we did a trivia question about looking for rogue set-UID programs that weren't installed in bin, sbin, or libexec directories. @cross pointed out that an evil attacker could install their rogue set-UID program in /tmp/.evil/sbin and successfully hide.

So today's Linux DFIR command line trivia question is, how can you get a list of the directories on your (hopefully uncompromised) system that currently contain set-UID executables? Then how could you use that list of directories to check another system for rogue set-UID programs?

#Linux #DFIR #CommandLine #Trivia

2022-12-13

Yesterday's Linux DFIR command line trivia asked how can you get a list of directories containing set-UID executables and then use that list to scan another system for rogue set-UID programs outside of those directories?

Getting the initial directory list is straightforward, and similar to some other challenges we've done recently:

```
find / -type f -perm /u+s -printf "%h\\n" | sort -u
```

But I think I might spruce up that output a little to make it more useful:

```
find / -type f -perm /u+s -printf "^%h/\\n" | sort -u >set-uid-dirlist.txt
```

Now my lines have a leading "^" and a trailing "/". This makes for better matching when I do my filtering for rogue directories:

```
find / -type f -perm /u+s | grep -vf set-uid-dirlist.txt
```

Now, as several folks pointed out, there's nothing that stops an attacker from placing rogue set-UID executables in one of the directories in my list, or even flipping on the set-UID bit on an existing executable. Nothing is perfect.

You could detect changes to existing binaries by running a package check on the target system ("rpm -Va" or "debsums" depending on your Linux flavor). And of course there are File Integrity Assessment (FIA) tools like Tripwire, AIDE, OSSEC, etc that will spot both new files in existing directories and changes to existing files. The problem with FIA tools is that the best use case is where you have them installed and configured BEFORE the compromise.

#Linux #DFIR #CommandLine #Trivia

2022-12-13

You have access to the vim text editor via sudo, but shell escapes are blocked. How do you escalate privileges to get an unfettered root shell without sudo?

#Linux #DFIR #CommandLine #Trivia

2022-12-14

Yesterday's Linux DFIR command line trivia asked what you can do to escalate privilege if you have sudo access to the vim text editor. The constraints are that shell escapes are disabled (see the "noexec" option to sudo) and your final privilege escalation path must not use sudo (because logging, y'all). Several people checked in with good ideas!

@steve and @millert (who knows a little something about sudo-- look it up) jumped in with a classic. Simply edit /etc/passwd and make your regular account UID 0 (or make yourself a new UID 0 account you can su into). Any account with UID 0 has root privs. You will need to log out and log back in again after making this change.

@millert and @timb_machine suggested setting up a root cron job to execute whatever commands you want-- just drop a new script into /etc/cron.hourly. For example, you could run commands as root to give you a set-UID copy of the shell:

```
cp /bin/bash /tmp/evil-bash
chown root:root /tmp/evil-bash
chmod 4555 /tmp/evil-bash
```

@rkervell went for editing a file like /etc/ld.so.conf and setting up an LD_PRELOAD style rootkit. Google "Linux LD_PRELOAD rootkit" for more background and some working examples.

@timb_machine checked in with a bunch of good ideas. For example, adding your own SSH public key to /root/.ssh/authorized_keys. You might also need to modify the "PermitRootLogin" setting in /etc/ssh/sshd_config, but once you have your key in authorized_keys you should be able to HUP the SSH server remotely to pick up the config change.

He also suggested making changes to other start-up files for the root user. For example, /root/.bashrc which will execute on every root shell execution (like the commands suggested for the evil cron job above). You might have to wait a bit for this to trigger though.

Tim also suggested using vim to overwrite an existing set-UID binary. For example, once you run "sudo vim" you could:

```
:r /bin/bash
:w! /usr/bin/chfn
:q
```

Then you should be able to execute "/usr/bin/chfn -p" and get your root shell.

That's a bunch of good ideas so far. One other idea I can think of is to modify the system PAM configuration. I'd have to fully research this idea, but you should be able to modify /etc/pam.d/su to remove the authentication requirement.

So the takeaway here is never give anybody root access to a text editor. Even if they don't directly shell escape, there's a lot of evil they can do!

#Linux #DFIR #CommandLine #Trivia

2022-12-14

Give me a Linux command line to search your logs and output all commands executed via sudo for user hal along with the date and time of each command.

#Linux #DFIR #CommandLine #Trivia

2022-12-15

Yesterday's Linux DFIR command line trivia asked for a shell pipeline to extract commands executed via sudo for a given user along with the date and time of each command.

One approach is to filter through the normal sudo Syslog messages that look like this:

```
Dec 13 17:07:00 LAB sudo: hal : TTY=pts/1 ; PWD=/home/hal ; USER=root ; COMMAND=/bin/vim
```

@spitecho and @koblas checked in with variations on this theme:

```
grep 'sudo: *hal' /var/log/auth.log | sed 's/ [^ ]* sudo:.*COMMAND=/ /'
```

I'm using grep to match on the sudo log lines related to user "hal", and then the sed pattern removes everything between the timestamp and the "COMMAND=" (including the "COMMAND="), replacing it with a single space.

@smlx suggested using journalctl rather than going directly at the logs in /var/log, but I find you get more historical information by grep-ing the raw logs.

@johntimaeus chimed in with the interesting idea of using ausearch on the audit.log data instead. Unfortunately, at least on my CentOS test system, the default audit logs don't capture the actual sudo events in enough detail to see the commands being executed.

#Linux #DFIR #CommandLine #Trivia

2022-12-15

You have a collection of images from multiple different sources. How can you find the duplicate images in a nested directory of image files?

#Linux #DIFR #CommandLine #Trivia

2022-12-16

Yesterday's Linux DFIR command line trivia asked you to find duplicate image files in a nested directory structure.

Several folks suggested using fdupes or jdupes. These would work well, but my person rules for these challenges are that I'm not allowed to use any commands that are not commonly installed as part of a base Linux OS. None of my test systems include these packages by default, although they are easily installed by your favorite package manager.

So that leaves the old-school approach:

```
find /some/path -type f -print0 | xargs -0 md5sum | sort | uniq -D -w32
```

Use find to get the file names, and then xargs to efficiently process checksumming all the files. Note I'm using "-print0 | xargs -0" in case we have to deal with files that have spaces in the name.

After that we sort the files by checksum and display all files with duplicate checksums.

If you want to remove the checksums from the output, we can use a little trick I employed in a previous posting:

```
find /some/path -type f -print0 | xargs -0 md5sum | sort | uniq -D -w32 |
while read hash file; do
[[ $hash != $lasthash ]] && echo =====
lasthash=$hash
echo $file
done
```

This will output the duplicate files in groups, separated by "=====" markers.

#Linux #DFIR #CommandLine #Trivia

2022-12-16

Today's Linux DFIR command line trivia is a suggestion from @apgarcia!

Given a stream of text or a collection of text files, output a sorted list of unique words similar to /usr/share/dict/words.

#Linux #DFIR #CommandLine #Trivia

2022-12-17

Yesterday's Linux DFIR command line trivia from @apgarcia asked to convert a stream of text into a unique list of words, one per line, similar to /usr/share/dict/words. This seems straightforward at first, but the devil truly is in the details on this challenge.

@koblas checked in with a sed pipeline:

```
cat textstream | sed 's/[^A-Za-z]/\\n/g' | sort -u | grep -v '^$'
```

I'd actually use tr instead of sed here:

```
cat textstream | tr -c A-Za-z \\\\n | sort -u | grep -v '^$'
```

"tr -c" means take the compliment of the first set. It's like the "[^A-Za-z]" pattern David used in the sed expression.

Either way, we're converting all non-alphabetic characters to newlines and then hitting that with sort/uniq. There will be a lot of blank lines in the output which will get squashed to a single blank line by "sort -u", but the final grep gets rid of the blank line from the final output.

@georgewherbert points out that because there will be so many blank lines coming out of the tr command, we're actually better off removing the blank lines earlier in the output so that sort doesn't have to do so much work:

```
cat textstream | tr -c A-Za-z \\\\n | grep -v '^$' | sort -u
```

This solution is a decent first approximation, but there are some difficulties.

Capitalization really is a hassle. For example, if your input is English text you will probably end up with both "the" and "The" (starts a sentence) in your output. We could just lower case everything:

```
cat textstream | tr -c A-Za-z \\\\n | grep -v '^$' | tr A-Z a-z | sort -u
```

Of course, this causes problems for proper nouns and the like. But if you're generally planning on doing case-insensitive matching against your word list anyway, this is probably the way to go.

Hyphenated words are another issue, as are contractions ("can't" etc). Frankly, I don't have a world-beating solution for this (you see what I did there?). Also, try using this command line against a Linux manual page (I was using the lsof manual page as my test input). All of those command line switches turn into single letter lines in your output.

So maybe we should establish a minimum word length for our output:

```
cat textstream | tr -c A-Za-z \\\\n | grep -Ev '^.{,3}$' | tr A-Z a-z | sort -u
```

Notice that the grep expression has changed. We've moved to extended regular expressions ("grep -E") and now we are eliminating lines with three or less characters on them.

Frankly, I think this is the best you are going to do without turning this into an actual script or program.

#Linux #DFIR #CommandLine #Trivia

2022-12-17

After a root compromise, you find these commands in /root/.bash_history:

```
dd if=/dev/urandom of=/junk bs=1M; rm -rf /junk
```

What did these commands accomplish?

#Linux #DFIR #CommandLine #Trivia

2022-12-18

Several people chimed in with the correct answer to yesterday's Linux DFIR command line trivia. Well done!

The "dd" command creates a file called "/junk" filled with pseudo-random data from /dev/urandom. The command will exit when all unused space is consumed. Then the rm command will remove /junk, freeing all of the space that file was using.

The upshot is that all of the file system's unused blocks are now overwritten with random data. Any evidence in unallocated is now gone forever.

Also, if you're in a virtual environment that doesn't pre-allocate disk space, your virtual disks have now been expanded to their maximum size. This is going to make taking a forensic image substantially more costly and could even result in denial of service conditions if you run out of back-end storage for your private cloud services.

@NegativeK asked about the impact of this command on SSD media. I'm speculating that some fragments of data will still exist in cells that are currently being reserved for wear-leveling. But you would have to conduct a chip-off to read that data.

By the way, the shitposting in response to the original question was first-rate. Please keep up the good work!

#Linux #DFIR #CommandLine #Trivia

2022-12-18

In a similar vein to yesterday's Linux DFIR command line trivia, you find the following commands in /root/.bash_history:

```
dd if=/dev/urandom of=/junk bs=1M &
tail -f /junk >/dev/null 2>&1 &
rm -rf /junk
```

Describe what is happening on this system and why you are most likely investigating.

#Linux #DFIR #CommandLine #Trivia

2022-12-19

To answer yesterday's Linux DFIR command line trivia, let me break down each of these commands in order:

```
dd if=/dev/urandom of=/junk bs=1M &
```

```
tail -f /junk >/dev/null 2>&1 &
rm -rf /junk
```

Just like the previous question, the dd command is going to fill up the root volume with a large file of pseudo-random data named /junk. The ampersand ("&") at the end of the command puts it in the background, freeing up the command prompt for the next command.

tail will start reading from /junk, but we're just throwing the output away by sending it to /dev/null, along with the standard error-- "2>&1" means "send standard error to the same place as standard output". The "-f" option means keep reading from the file and outputting any new data. Again we put this command in the background ("&") to get our shell prompt back.

Lastly, we remove /junk. What happens to the previous two commands? No a lot really. dd is still happily shoving data into the file and consuming disk blocks and tail is happily reading the data and throwing it away in /dev/null. The fact that the file has become unlinked from the file system doesn't matter to any commands that already have the file open.

Eventually dd will fill the root volume and exit. But because we invoked tail with the "-f" option, it will keep the file open looking for more data to be written to the file.

And this is the critical difference from the previous trivia question. As long as tail keeps the file open, the disk blocks allocated to the file cannot be freed back into the file system. So now all the disk space in the root volume has been consumed by a file that is no longer visible in the file system. And this is probably why the SOC woke you up in the middle of the night-- your disk space is gone and nobody can find where it went!

There are a couple of ways to locate these "open but unlinked" files in your file system. You could leverage the /proc/<pid>/fd directories:

```
ls -l /proc/*/fd 2>/dev/null | grep deleted
```

But personally I prefer to just use "lsof +L1". The "+L1" option means "show files with link count less than one", or link count zero in other words.

To free up the space occupied by the unlinked file, just kill any process(es) that have the file open. So once you find and terminate the tail process, your free space should come back. Of course unallocated has still be overwritten with random data, which is not ideal.

#Linux #DFIR #CommandLine #Trivia

2022-12-19

After breaking root on a system you don't want your commands to be logged in root's bash history file. But you do want to be able to refer to your command history during your current session. How can you accomplish these goals?

#Linux #DFIR #CommandLine #Trivia

2022-12-20

Yesterday's Linux DFIR command line trivia asked how you can use your command history in a shell session, but not have the commands logged to the .bash_history file.

Lots of responses to this one! First let's break down some of the ones that won't work for you:

-- There were multiple variations on "export HISTCONTROL=ignorespace" and then preceding all your commands with spaces. That will prevent the commands being logged to the bash history, but it will also mean that you can't use your own command history during your shell session. No bueno.

-- Doing "rm /root/.bash_history" isn't going to work out for you either. Your shell history is written to disk when your shell exits. If you remove the history file and then exit your shell, you're going to end up with a new history file that contains just the shell commands from your most recently completed session.

-- "chmod -w /root/.bash_history" doesn't work. You're root, so you can write to any file regardless of permissions.

-- "set +o history" isn't what we want either, since it will disable history for the rest of your current session. Also any commands you entered up to and including "set +o history" will end up in the /root/.bash_history file.

OK then, how about some of the other choices?

"export HISTFILESIZE=0" meets the conditions of the challenge (this suggestion came from @fakejazz). You can still use your history in your current shell session, but you will have an empty history file when you exit. The only downside here is that empty history will make it obvious that somebody has been messing around. "rm /root/.bash_history; ln -s /dev/null /root/.bash_history" is similar.

@jtk suggested exiting your shell with "kill -9 $$". This works. Shell history is only written on a normal exit. "kill -9" prevents the shell from writing history the way it usually would.

@sternecker suggested doing "history -c" to clear your history before exiting the shell. This one also works.

Finally we got multiple responses suggesting variants of "unset HISTFILE" or "export HISTFILE=/dev/null" (giving credit to @dot0 for being first off the mark with this solution). This was actually the solution I had in mind when I proposed the challenge. I prefer this solution because I can set the variable at any time during my shell session and I don't have to remember to exit my shell in some special fashion.

#Linux #DFIR #CommandLine #Trivia

2022-12-20

Write a "find" expression to locate directories whose names begin with a dot (".") and which are not located in a user's home directory.

#Linux #DFIR #CommandLine #Trivia

2022-12-21

Yesterday's Linux DFIR command line trivia ended up being another learning experience for me. The question was how to find directories whose names begin with a dot (".") which are not located in a user's home directory. This is useful for finding hidden directories where attackers may be staging tools without being distracted by the typical directories like /root/.ssh and so on.

My solution looks like this:

```
find / \\( -path /root -o -path /home/\\* \\) -prune -o -type d -name .\\* -print
```

Filter out the /root and /home/* directories with -prune and then print the remaining directories with leading dots. The -print at the end ensures you only print out the directories you want and not the pruned directories.

Then @tfkhdyt checked in with an alternate approach:

```
find / -type d -name .\\* ! \\( -path /root/\\* -o -path /home/\\* \\)
```

Filtering on the back end of the command eliminates the need for -prune and -print, and overall I think the intent of the command is clearer. From a performance perspective, there doesn't seem to be any significant difference.

So thanks for changing my thinking on this!

#Linux #DFIR #CommandLine #Trivia

2022-12-21

Happy holidays to all, no matter how you celebrate! I've got a lot of family responsibilities through the holiday season so my Linux DFIR command line trivia is going on hiatus until the new year.

I will be back in 2023 with more command line goodness! Wishing you all the best until then!

#Linux #DFIR #CommandLine #Trivia

2023-01-02

Write a Linux command line to extract only a single line from a text file-- let's say line number 25 for purposes of this challenge.

#Linux #DFIR #CommandLine #Trivia

2023-01-03

Answers to yesterdays Linux DFIR command line trivia fell into two distinct camps. There's the folks (like @fw) who liked head and tail:

```
head -25 myfile.txt | tail -1
```

Note that if you prefer you could pipe tail into head instead and get the same result:

```
tail -n +25 myfile.txt | head -1
```

The other camp were the sed users. @lamitpObuS got in first with:

```
sed -n '25p' myfile.txt
```

The "-n" option tells sed not to print out every line (normally the default). And then '25p' says to explicitly print out line 25 (and I am having "ed" flashbacks).

However, @deeseearr pointed out that even though we are only printing the 25th line here, sed goes on and reads the entire file, which could be painful if you're dealing with a large logfile. So we'll tweak the original sed solution ever so slightly:

```
sed -n '25p; 25q' myfile.txt
```

In other words, "print line 25 and then quit on line 25".

#Linux #DFIR #CommandLine #Trivia

2023-01-03
I have a tab-separated file of census data that gives the top 100 male and female names. The fields on each line are:

rank malename population femalename population

For example:

1 James 4,700,229 Mary 3,196,385
2 Robert 4,455,696 Patricia 1,558,407
[...]

I want to create a file with just the names, one name per line, sorted alphabetically.

#Linux #DFIR #CommandLine #Trivia

2023-01-04
Yesterday's Linux DFIR command line trivia asked you to parse names out of fields two and four in a tab separated files and create a new file that is the list of names, one per line, in alphabetical order.

First up to the plate was @lamitpObuS with a solution using cut:

```
cut -f2,4 names | tr \\\\t \\\\n | sort
```

Then @derekmceachern wandered in with an awk solution:

```
awk '{print $2"\\n"$4}' names | sort
```

These are both good solutions.

I could also do it with a loop, though I'm not sure why you'd want to give the awk solution:

```
while read num n1 pop n2 junk; do
echo -e $n1\\\\n$n2
done <names | sort
```

#Linux #DFIR #CommandLine #Trivia

2023-01-04
Create an empty directory structure that exactly duplicates an existing set of directories, but without any of the files from the original directory structure.

#Linux #DFIR #CommandLine #Trivia

2023-01-05
Yesterday's Linux DFIR command line trivia again saw folks showing up with multiple different approaches.

First is the straightforward "find ... | xargs mkdir" or "find ... -exec mkdir" approach. I'll give @prograhamer credit for being the first to check in with this idea.

```
cd /orig/directory
```

```
find . -type d -print0 | (cd /target/dir; xargs -0 mkdir)
```

The subshell here lets us take the directory list output from the find command and execute "xargs mkdir" in the target directory location. We use "-print0" and "xargs -0" just in case we're dealing with directory names that contain spaces.

Then @apgarcia went all old-school and suggested "find ... | cpio":

```
cd /orig/directory
find . -type d | cpio -pdm /target/dir
```

cpio in passthru mode ("-p") takes a list of paths, one per line (no need for -print0) and copies them to the target destination. "-d" means make directories as needed and "-m" preserves last modified times.

@uriy jumped in with the rsync version:

```
rsync -r --dirs -f'+ **/' -f'- *' /orig/directory/ /target/directory/
```

rsync is recursively ("-r") copying the entire directory contents, making directories as needed ("--dirs"). However, the first filter matches just the directory names (paths ending in slash) and the second filter suppresses the other directory contents. The rsync solution has the advantage that you don't need to do a cd command at any point.

Finally, @silverwizard suggested simply doing a "cp -r" and then going back and deleting everything that wasn't a directory. I shall treat this solution with the seriousness it deserves...

#Linux #DFIR #CommandLine #Trivia

2023-01-05

You have a file of output from the "host" command:

```
207.251.16.10.in-addr.arpa domain name pointer server1.srv.mydomain.net.
208.251.16.10.in-addr.arpa domain name pointer server2.srv.mydomain.net.
16.254.16.10.in-addr.arpa domain name pointer www.mydomain.net.
17.254.16.10.in-addr.arpa domain name pointer mydomain.com.
```

Change it into /etc/hosts format:

```
10.16.251.207 server1.srv.mydomain.net
10.16.251.208 server2.srv.mydomain.net
10.16.254.16 www.mydomain.net
10.16.254.17 mydomain.com
```

#Linux #DFIR #CommandLine #Trivia

2023-01-06

Yesterday's Linux DFIR command line trivia seems to have hit a nerve with some folks. Some people wanted me to not use the "host" command, we got a Perl solution, and folks were concerned about the need to parse IPv6 addresses. I tell you it's pandemonium out there!

The goal is to turn a file of lines like this:

```
16.254.16.10.in-addr.arpa domain name pointer www.mydomain.net.
```

into this:

```
10.16.254.16 www.mydomain.net
```

@lamitpObuS checked in with a sed solution that I can optimize somewhat:

```
sed -r 's/([0-9]+)\\.([0-9]+)\\.([0-9]+)\\.([0-9]+)\\..*pointer/\\4.\\3.\\2.\\1/; s/\\.$//' inputfile
```

We're using "sed -r" here which turns on extended regular expression syntax. The main advantage there is that we don't have to backwhack all the parens, which makes things considerably more readable.

The first substitution matches the four octets in the in-addr.arpa address as individual subexpressions along with all of the other "domain name pointer" text. All of that noise gets replaced with the four octets in normal network order.

The second substitution just removes the trailing dot from each line.

Now was that so hard? Settle down people, it's almost the weekend!

#Linux #DFIR #CommandLine #Trivia

2023-01-06

Attackers staged files and malware in /tmp and then later deleted everything. The full disk is too large to image so you want to extract just the blocks for the EXT block group associated with /tmp so that you can do some file carving. How can you do this using only standard Linux command line tools?

#Linux #DFIR #CommandLine #Trivia

2023-01-07

Yesterday's Linux DFIR command line trivia definitely seems to have been a stumper. Let's start by reviewing some EXT basics.

By default, EXT file systems use 4K blocks. These blocks are divided into block groups of 32K (32768) blocks. Each block group has 8K (8192) inodes associated with it.

By default, new files are put into the same block group as their parent directory. We can figure out which block group /tmp belongs to by looking at its inode:

```
# ls -id /tmp
57346 /tmp
# echo $((57346 / 8192))
7
```

OK, inode 57346 falls into block group seven. We can use dd to extract the blocks in that block group (use your root drive device in place of /dev/sda2):

```
dd if=/dev/sda2 bs=$((4096*32768)) skip=7 count=1 | gzip >block-group07.gz
```

I'm setting dd's block size argument to be the entire block group. This makes using the "skip" and "count" arguments more intuitive. The actual read/write buffer size for dd is going to end up being less than 128MB. Make sure your output is going into a different volume than the one you are imaging from!

By the way, when a block group fills up, sometimes blocks slop over into the next block group. So you might want to capture some extra block groups (say "count=4") rather than just to give yourself a better shot at collecting all of /tmp.

#Linux #DFIR #CommandLine #Trivia

2023-01-07

Your web logs document multiple attackers trying a directory traversal attack originating from /cgi-bin/. So the exploit indicator is "/cgi-bin/../". However, either one or both of the dots may be encoded as "%2e". Output all IPs that attempted the exploit against this web server and the number of attempts per IP. Sort the output by the number of attempts. Assume the source IP is the first space-separated field of the log file.

#Linux #DFIR #CommandLine #Trivia

2023-01-08

I thought I was giving you all a break with yesterday's Linux DFIR command line trivia. But it seems to have been more challenging than anticipated. Perhaps we need more regular expression based challenges to practice on...

In this case we're trying to match the pattern "/cgi-bin/../" where one or both of the dots might be encoded as "%2e". When we match, we need to output the first space-delimited field of the matching line. This is clearly a job for awk:

```
awk '/\\/cgi-bin\\/(\\.|%2e){2}\\// {print $1}' access_log | sort | uniq -c | sort -nr
```

We use awk's pattern matching ("/.../") to find the lines we want and then output the first field ("{print $1}"). The rest of the pipeline is the "command line histogram" idiom I've used before-- "uniq -c" counts the number of times each IP occurs in the output and the find "sort -nr" sorts them in descending numerical order by count.

So let's talk about the regular expression. You'll note that I have to carefully backwhack every literal slash ("/") in the pattern so that awk doesn't confuse these literal characters with its own pattern matching operator ("/.../"). "(\\.|%2e)" means match either a literal dot or the string "%2e", and we require this to happen twice ("{2}").

Both awk and regular expressions are their own particular domain specific languages. But if you spend a lot of time on the Linux command line, both are extremely useful to learn!

#Linux #DFIR #CommandLine #Trivia

2023-01-08

During a live response, you find suspicious accounts SSHing into a system. Write a loop that continuously kills all SSH processes that don't belong to your user ID and which are not the master SSH daemon.

#Linux #DFIR #CommandLine #Trivia

2023-01-09

Before we get to the shellicious answer to yesterday's Linux DFIR command line trivia, a shout out to @truekonrads. He points out that if you really want to block certain users from SSHing into your host, use the AllowUsers option to specify a list of the users who are allowed to log in and then everybody else will be blocked.

But I was trying to get you to think about filtering "ps" output in various clever ways. Here's some sample output:

```
# ps -ef | grep sshd
root 1220 1 0 2022 ? 00:00:00 /usr/sbin/sshd -D
root 59568 1220 0 Jan08 ? 00:00:00 sshd: hal [priv]
hal 59579 59568 0 Jan08 ? 00:00:00 sshd: hal@pts/0
root 59707 1220 0 Jan08 ? 00:00:00 sshd: evil [priv]
evil 59727 59707 0 Jan08 ? 00:00:00 sshd: evil@pts/1
```

Each interactive SSH session is actually two processes because of privilege separation. One will be root owned and one will be owned by the unprivileged user. The goal is to destroy all sshd processes that are not the master process or associated with user "hal".

Some folks suggested solutions based on "pgrep", but frankly I'd just use awk:

```
# ps -ef | awk '/sshd/ && !/sshd: hal/ && $3 != 1 {print $2}'
59707
59727
```

Match sshd lines in the "ps" output, but then ignore lines associated with user hal or which have a PPID of 1 (the master sshd). When we have a matching line, output just the PID from the second field.

With that basic logic working, now all we have to do is feed the PIDs to "kill" inside of a loop:

```
while :; do
kill -9 $(ps -ef | awk '/sshd/ && !/sshd: hal/ && $3 != 1 {print $2}')
sleep 1
done
```

"while :; do" is a convenient idiom for creating an infinite loop. Inside the loop we "kill -9" the offending PIDs and then sleep for one second so we don't completely slam the CPU.

Note that loops like this can also be fun during your next CTF if you're trying to keep opponents away from a flag.

#Linux #DFIR #CommandLine #Trivia

2023-01-09

You want to incorporate raw audit.log data into a database or other collection. But before you do that you need to convert the epoch timestamps on each line into human readable dates.

Create a Linux command line that reads raw audit.log data and outputs each line starting with an additional "YYYY-MM-DD hh:mm:ss" timestamp and then the rest of the original line.

#Linux #DFIR #CommandLine #Trivia

2023-01-10

Yesterday's Linux DFIR command line trivia provoked one "install logstash" answer and one Perl answer. Let's try an up our shitposting game here, people!

The goal is to take an audit.log entry like this:

```
type=LOGIN msg=audit(1673352061.960:35854): pid=8213 uid=0
subj=system_u:system_r:crond_t:s0-s0:c0.c1023 old-auid=4294967295 auid=0 tty=(none) old-
ses=4294967295 ses=4223 res=1
```

And output the same line but with a human-readable time and date stamp at the front of the line.

The epoch date is the number that appears in parentheses after "msg=audit(...". I'm going to rewrite each line to put a copy of this epoch date at the front of each line:

```
sed -r 's/(.*msg=audit\\()([0-9]*)(.*)/\\2 \\1\\2\\3/' /var/log/audit/audit.log
```

I'm breaking the line into three chunks: everything up to and including "...msg=audit(", the epoch date, and then everything after that. Replace that with the date followed by the original line in order.

With the epoch date at the front of the line, we can now use a partial awk solution sent in by @lamitpObuS:

```
sed -r 's/(.*msg=audit\\()([0-9]*)(.*)/\\2 \\1\\2\\3/' /var/log/audit/audit.log | awk '$1 = strftime("%F
%T", $1)'
```

The awk nicely converts the epoch date into the "YYYY-MM-DD hh:mm:ss" format we want.

Teamwork! That's how we do it!

#Linux #DFIR #CommandLine #Trivia

2023-01-10

You can see the list of processes using network resources with "netstat -peanut" or "lsof -i". Output the full executable path for each process.

#Linux #DFIR #CommandLine #Trivia

2023-01-11

Yesterday's Linux DFIR command line trivia asked us to get a listing of the full pathnames of executables for processes that are currently using network sockets. "netstat" and "lsof" can give us a list of PIDs, and then we will need to convert that to executable paths.

Let's start with "netstat":

```
netstat -peanut | tail -n +3 | sed -r 's/.* ([0-9]*)\\/.*/\\1/' | sort -un
```

"-peanut" is a helpful mnemonic for all processes ("-a") using TCP ("-t") and UDP ("-u") sockets, giving process info ("-p") but not converting IP addresses and ports into names ("-n"). "-e" just gives more detailed info and also allows us to spell "peanut" with the options.

We need "tail -n +3" to skip the header lines in the output. And then some crazy "sed" to extract just the PIDs. A final "sort -un" to give us just the unique PIDs in numerically sorted order.

And then @polyna dropped in to remind me that Linux deprecated "netstat" a long time ago. Sigh. Here's the same thing using "ss" like we all should be (but which my fingers generally refuse to type the first time):

```
ss -natup | sed -r 's/.*pid=([0-9]*).*/\\1/' | sort -un
```

At least there are no header lines to skip in the "ss" output, but it's still some nasty "sed" to extract the PIDs.

Which is why I generally prefer the "lsof" option:

```
"lsof -i -t"
```

Yep, that's it. "-i" means show me processes using "internet sockets" and "-t" is "terse mode" which just outputs the PIDs. Terse mode is designed to be used with "kill" but we want to do something else with the PID list:

```
lsof -i -t | xargs -I{} readlink /proc/{}/exe
```

I could also do this with a loop, but "xargs" gives us a quick way to call "readlink /proc/<pid>/exe" for each PID from "lsof". The "-I{}" lets us create a marker for where we want "xargs" to put the PID in each command.

Why is this useful at all? See if you can spot the suspicious process in the sorted list of executables:

```
# lsof -i -t | xargs -I{} readlink /proc/{}/exe | sort -u
/dev/shm/.rk/lsof (deleted)
/dev/shm/.rk/xterm (deleted)
/usr/lib/systemd/systemd
/usr/sbin/dnsmasq
/usr/sbin/NetworkManager
/usr/sbin/sshd
```

#Linux #DFIR #CommandLine #Trivia

2023-01-11

The admin wants to prevent users from creating their own authorized_keys files. How could this be accomplished?

#Linux #DFIR #CommandLine #Trivia

2023-01-12

Yesterday's Linux DFIR command line trivia asked how an admin could prevent users from creating their own authorized_keys files.

@DavidJBianco suggested disabling Public Key authentication entirely. I have to disallow this answer because while it moots the authorized_keys file entirely, I believe the question implied that we wanted to keep Public Key authentication available. I'll be clearer next time.

Multiple people suggested changing the location of the authorized_keys file by setting AuthorizedKeysFile in sshd_config (looks like @llutz was the first to chime in on this idea). The admin could set the file path to a directory that was root-owned and populate the file with whatever keys they felt appropriate.

@mjharmon checked in with the command line solution I was expecting: using "chattr +i" to render users' authorized_keys files immutable. So we might do something like:

```
rm -f /home/*/.ssh/authorized_keys
for dir in /home/*; do
```

```
mkdir -p -m 700 "$dir/.ssh"
touch "$dir/.ssh/authorized_keys"
chattr +i "$dir/.ssh/authorized_keys"
done
```

The first "rm" command removes any existing authorized_keys files, or even symlinks if the user decided to get tricky. Then the loop ensures we have a ".ssh" directory for each user, which gets a new empty authorized_keys file, which then gets set immutable so it can't be changed.

Some suggested using file and directory permissions to solve the problem, but it simply isn't workable. Even if the root user created root-owned ".ssh" directories for all users (problematic because of the known_hosts file among other reasons) with root-owned authorized_keys files, the user could simply rename the root-owned ".ssh" directory because they have write permissions on their own home directory. Once the root-owned directory was moved out of the way, the user would be free to create their own directory with their own authorized_keys file.

EDIT: Actually @bbaugh points out the "chattr" solution above fails for exactly the same reason-- the user can rename the directory with the immutable file and then start a new directory with their own authorized_keys. You could make the ".ssh" directory immutable, but then you have problems with things like known_hosts again. On the whole, I like the sshd_config file change better.

#Linux #DFIR #CommandLine #Trivia

2023-01-12

You want to extract both ASCII and Unicode strings from a large disk image, but you don't want to have to read the disk image more than once because it is so large. How could you do this on the Linux command line?

#Linux #DFIR #CommandLine #Trivia

2023-01-13

Yesterday's Linux DFIR command line trivia asked how you could extract both ASCII and Unicode strings from a large disk image without having to read the disk image twice.

@steve chimed in with the way I had always done this, using a FIFO and "tee":

```
mkfifo /tmp/myfifo
cat /tmp/myfifo | strings -a -el >strings.unicode &
cat disk.img | tee /tmp/myfifo | strings -a >strings.ascii
```

A FIFO is a disk-based interprocess communication channel kind of like a pipe. We create the FIFO and then create a background process that will read any data put into the FIFO and pass that data into "strings -a -el" to extract Unicode strings. Then we push our disk image into the "tee" command that writes one copy to the FIFO and dumps a second copy to standard out. We have a second "strings" process taking the standard out from "tee" and grabbing the ASCII strings.

Steve and I were feeling pretty good about life until @apgarcia reminded us of some useful bash syntax:

```
cat disk.img | tee >(strings -a -el >strings.unicode) | strings -a >strings.ascii
```

The ">(...)" syntax in bash lets you create a shell pipeline that gets treated like a file handle. This way we don't need to mess around with the FIFO at all, just let bash handle everything.

So TIL anytime you're thinking about using a FIFO, maybe you ought to consider ">(...)" instead!

#Linux #DFIR #CommandLine #Trivia

2023-01-13

I'm going to be traveling for the MLK weekend, and the daily Linux DFIR command line trivia will be going on hiatus until Tuesday. Everybody have a great weekend and we will resume our daily Linux nerdfest next week!

#Linux #DFIR #CommandLine #Trivia

2023-01-17

You have a traditional Syslog style log file with dates in the "Jan 17 08:10:11" format, written in the default local time zone of the machine. Output the log file with the dates converted to "YYYY-MM-DD hh:mm:ss" format in UTC.

#Linux #DFIR #CommandLine #Trivia

2023-01-18

Yesterday's Linux DFIR command line trivia provoked a Python solution, a Ruby solution, and a pretty scripty awk solution. And that's OK, because previously I had generally solved this problem with Perl myself.

But with the GNU "date" command as powerful as it is, I always wondered if there was a pure bash solution. So I spent a little time with the "date" manual page and one of the examples suggested the following solution:

```
cat /var/log/messages |
while read mon day time log; do
echo $(date -u "+%F %T" -d "TZ=\\"US/Eastern\\" $mon $day $time") $log
done
```

Not only does "-d" allow you to specify a date/time string, but you can even include a time zone specifier as we're doing here. Then the "-u" option forces the "date" command to convert the output to UTC.

Now of course the classic problem with classic Syslog timestamps is that they don't include the year. "date" will infer the current year when converting the date/time string, but suppose our logs were from an earlier year? Well, we can just include the appropriate year in our date/time string:

```
cat /var/log/messages |
while read mon day time log; do
echo $(date -u "+%F %T" -d "TZ=\\"US/Eastern\\" $mon $day 2022 $time") $log
done
```

"date" will recognize the date/time string format here and output the logs from 2022.

And it turns out we can generalize this idea to include other information in the date/time string as @ilikepi demonstrated in his solution:

```
export LTZ=$(date '+%Z')
cat /var/log/messages |
while read mon day time log; do
```

```
echo $(date -u "+%F %T" -d "$mon $day $LTZ $time") $log
done
```

James is using "date" to first get the common name of the current timezone and storing it in $LTZ. We then simply include $LTZ in the date/time string.

While this solution definitely reads better, I'm not honestly certain what would happen if you used this approach on a log that spans the Standard/Daylight time change.

#Linux #DFIR #CommandLine #Trivia

2023-01-18

What Linux commands could you use to add an entry to a traditional "Vixie cron" type crontab and have them be executed by cron at the next appropriate interval?

#Linux #DFIR #CommandLine #Trivia

2023-01-19

Yesterday's Linux DFIR command line trivia asked how you can add an entry to the traditional "Vixie cron" style crontabs from the command-line. This one is a feature of many attacker post-exploitation scripts and it pays to be on the lookout for it. The problem is that you can't just append an entry onto an existing crontab file, because the running cron daemon won't see the new entry until it is restarted or HUPed.

@jtk and @deeseearr checked in with the classic strategy for inserting new cron entries:

```
(crontab -l; echo '* * * * * echo Excellent cron job, would run again!') | crontab -
```

"crontab -l" outputs the current cron entries for the user, and then we can append whatever additional jobs we want. "crontab -" replaces the user's crontab with the standard input and notifies the cron daemon of the change.

Sometimes I'll see code like this:

```
crontab -l >/tmp/cronfile
echo '* * * * * echo Sneaky cron job' >>/tmp/cronfile
crontab /tmp/cronfile
rm -f /tmp/cronfile
```

This works, but it leaves a copy of the modified cron file in unallocated (smart attackers will use "shred -u /tmp/cronfile" instead of "rm"). I prefer the "crontab -" approach that writes no intermediate files to disk.

But what always makes me wonder is why the attackers bother with Vixie cron at all. It seems to me that the easiest and sneakiest way to get your code executed regularly would be to just append a line to one of the scripts already in /etc/cron.hourly. That strategy seems more likely to be overlooked to me.

#Linux #DFIR #CommandLine #Trivia

2023-01-19

In honor of folks traveling to ShmooCon this weekend...

The airport WiFi network gives you 30min of free WiFi. Write a Linux command line to change your ethernet address randomly every 30 minutes. Bonus points if it also automatically requests a new DHCP lease.

#Linux #DFIR #CommandLine #Trivia

2023-01-20

Yesterday's Linux DFIR command line trivia asked how to bypass those annoying "free internet" portals that limit access by time or bandwidth. Simply changing your MAC address generally works.

So first let's talk about generating random MAC addresses. You could generate all six octets like this:

```
printf "%02x:%02x:%02x:%02x:%02x:%02x" $(($RANDOM % 255)) $(($RANDOM % 255))
$(($RANDOM % 255)) $(($RANDOM % 255)) $(($RANDOM % 255)) $(($RANDOM % 255))
```

Kind of gross, but it works.

However, in my experience, this sometimes generates invalid MAC addresses. So I usually prefer to steal the first three octets (the "Organizational Unit Identifier" or "OUI") from my primary MAC address and then generate the last three octets randomly:

```
oui=$(ip link show ens33 | awk '/link/ {print $2}' | cut -d: -f1,2,3)
printf "%s:%02x:%02x:%02x" $oui $(($RANDOM % 255)) $(($RANDOM % 255)) $(($RANDOM
% 255))
```

Replace "ens33" with the name of your primary NIC in the first command above.

OK, cool, now that we can make random ethernet addresses, all we need is a little looping action:

```
oui=$(ip link show ens33 | awk '/link/ {print $2}' | cut -d: -f1,2,3)
while :; do
ip link set dev ens33 addr $(printf "%s:%02x:%02x:%02x" $oui $(($RANDOM % 255))
$(($RANDOM % 255)) $(($RANDOM % 255)))
dhclient -r
dhclient
sleep 1730
done
```

Set our primary interface with a random ethernet address, then release the current DHCP lease, and then ask for another. The sleep statement waits for just under 30 minutes before the whole cycle starts again.

Note that this approach is generally fine for web browsing, but it's definitely going to interrupt persistent connections like SSH. So be sure you're using tmux on all those interactive login sessions!

#Linux #DFIR #CommandLine #Trivia

2023-01-20

You're capturing the image of a system that uses Linux Software RAID. How can you dump the current RAID configuration before you shut the machine down in order to more easily rebuild the RAID in your lab?

#Linux #DFIR #CommandLine #Trivia

2023-01-24

Wow, it's been a crazy few days! Last Friday's daily Linux DFIR command-line trivia asked how to capture the current Linux Software RAID configuration of a system before shutting it down and imaging the drives.

@cephurs checked in with the classic answer:

```
mdadm --detail --export >raid-config.txt
```

This will give you a nice breakdown of configured RAID devices and save it to a file.

But let me show you a little trick that comes in handy every so often:

```
# mdadm --detail --export
mdadm: No devices given.
# mdadm --detail -s --export
MD_LEVEL=raid1
MD_DEVICES=2
MD_METADATA=1.2
MD_UUID=cccdcc9b:f86a0b98:98df7949:8f2f7bb8
MD_DEVNAME=localhost:boot
MD_NAME=localhost:boot
MD_DEVICE_dev_loop0_ROLE=0
MD_DEVICE_dev_loop0_DEV=/dev/loop0
MD_DEVICE_dev_loop1_ROLE=spare
MD_DEVICE_dev_loop1_DEV=/dev/loop1
MD_LEVEL=raid1
MD_DEVICES=2
MD_METADATA=1.2
MD_UUID=d6454bae:a2d8ee0c:4b7ef9c3:c09ec51a
MD_DEVNAME=localhost:pv00
MD_NAME=localhost:pv00
MD_DEVICE_dev_loop2_ROLE=spare
MD_DEVICE_dev_loop2_DEV=/dev/loop2
MD_DEVICE_dev_loop3_ROLE=1
MD_DEVICE_dev_loop3_DEV=/dev/loop3
```

See the "-s" option in the second command? That's the "scan" option. In addition to looking for configured devices in mdadm.conf, the "-s" also tells the "mdadm" command to scan /proc/mdstat for RAID devices that may have been configured "on the fly" by admins.

Speaking of /proc/mdstat, it might also be a good idea to grab the contents of that as well:

```
cat /proc/mdstat >proc-mdstat.txt
```

/proc/mdstat gives you a better idea of the current running status of the system's RAID devices:

```
# cat /proc/mdstat
Personalities : [raid1]
md126 : active (read-only) raid1 loop2[0](F) loop3[1]
100882432 blocks super 1.2 [2/1] [_U]
bitmap: 0/1 pages [0KB], 65536KB chunk
```

```
md127 : active (read-only) raid1 loop1[1](F) loop0[0]
3904512 blocks super 1.2 [2/1] [U_]
bitmap: 0/1 pages [0KB], 65536KB chunk

unused devices: <none>
```

Hopefully between the two sources of information, you'll have enough clues to rebuild the RAID configuration in your lab.

#Linux #DFIR #CommandLine #Trivia

2023-01-24

You are looking at timestamps on a software package directory under /opt:

-- atime and mtimes are mostly all months in the past, with a few atimes that are current
-- creation dates are within the last 24 hours and clustered tightly together
-- ctime values are all equal to or greater than the creation date on the same file

What caused this timestamp pattern? Was it malicious timestomping or something else?

#Linux #DFIR #CommandLine #Trivia

2023-01-25

Yesterday's Linux DFIR command line trivia asked whether malicious timestomping or some other situation caused a particular timestamp configuration. atime and mtime values are generally months old, but btime and ctime values are recent and tightly clustered.

@deeseearr correctly suggested that one likely cause would be unpacking an archive like a tar or ZIP file. If you unpack an archive as root, it will generally set the atime and mtime values on files based on timestamps stored in the archive. But btime will be the actual creation date of the file on your system, and ctime will reflect the time when the atime and mtime values got set shortly after file creation.

@smlx suggested the other common reason for this timestamp pattern-- the files were copied onto the system using "rsync -a". It looks very similar to an archive unpack because "rsync -a" preserves atime and mtime values from the original source directory, but the btime and ctime values reflect the arrival time of the files on the local system.

Of course it is possible that the file timestamps were maliciously manipulated. Clues that it might be enemy action include things like zeroed fractional seconds, because the attacker specified times to one second granularity and forgot the fractional seconds. Another clue would be all of the mtime and atime values being identical, as if somebody did "touch ... *" on the directory.

#Linux #DFIR #CommandLine #Trivia

2023-01-25

Where can you find a record of USB devices that have been plugged into a Linux system?

#Linux #DFIR #CommandLine #Trivia

2023-01-26

I have to admit that sometimes I pose these daily Linux command line trivia questions because I'm hoping to learn something I didn't know. Yesterday's question asked how you can get historical information on USB devices plugged into a Linux system.

I got a couple of responses suggesting the "lsusb" command. "lsusb" is great for looking at USB devices currently plugged into the system, but unfortunately has no historical information about devices that have been connected and later removed.

And so we're left with the traditional method of looking at the kernel logs, whether through "dmesg" or just going and finding the raw logs themselves under /var/log. The problem here is that these logs don't last forever. Under standard Linux log rotation settings, they're only going to be around for about a month. After that, you might be able to pull some old logs out of unallocated, emphasis on "might".

As always, it's a good idea to keep copies of your logs in some central, searchable collection for longer than a month. In this case, it will help you go back historically and track any USB devices that may have been deployed during an incident.

#Linux #DFIR #CommandLine #Trivia

2023-01-26

What's the difference between the "auid" and "uid" values in a Linux audit log?

#Linux #DFIR #CommandLine #Trivia

2023-01-27

Yesterday's daily Linux DFIR command line trivia asked what is the difference between the "auid" and "uid" values in a Linux audit log. @SaThaRiel and @deeseearr checked in with correct answers, and @k8em0 checked in with a musical interlude. So thanks all around!

"auid" is short for "audit user ID". This is the user ID you log in as, and it's tracked consistently throughout your login session.

"uid" is the actual UID the event happened as. So if you're using "sudo" or "su", the "uid" field will reflect the user account that the command is actually running as.

Note that there are other fields including "euid" ("effective UID"), "suid" ("set UID"), and "fsuid" ("file system UID"). I've tried different scenarios to affect these values but I've never gotten them to be different from the "uid" value in my audit logs.

For example, if I login as user "hal" (auid=1000) and then "sudo" to get a root shell (uid=0), and then "su" to run a command user nobody with UID 99, I would expect to see a log where "auid=1000 uid=0 euid=99". But that's not what I see. Instead I get an audit record where "auid=1000" and all of the other "*uid" fields are set to 99. Go figure.

Anyway, the distinction between "auid" and "uid" is a useful one since it lets you home in on the users that may be abusing escalated privileges.

#Linux #DFIR #CommandLine #Trivia

2023-01-27

Write a Linux command line to give the names of the sub-directories in just the current working directory (no recursion).

Also, I've decided to reclaim my weekends, so these daily Linux challenges will henceforth be a Mon-Fri affair. I'll check in with the answer to this question on Monday. Everybody enjoy your weekends!

#Linux #DFIR #CommandLine #Trivia

2023-01-30

Last Friday's daily Linux command line trivia asked for a command line to print out the names of the subdirectories of the current directory, but NOT to recurse and show all subdirectories below this point. There are lots of ways to accomplish this, and I was curious to see if I would learn any new ones. Turns out I did.

First, there's a classic solution using the "find" command (props to @barubary for being the first to mention it):

```
find . -maxdepth 1 -type d
```

Here we're using the "maxdepth" control to only show directories ("-type d") in the current directory.

There are also solutions using "ls". @beejjorgensen checked in with one of my favorites:

```
ls -lA | grep ^d
```

In the detailed listing format ("ls -l"), the file type is the first character on the line. Directories are denote with a "d".

@llutz checked in with another "ls" solution that I hadn't considered before:

```
ls -d .*/ */
```

@barubary points out that "echo" would work here too, instead of "ls -d". The trailing "/" after the wildcards matches only directories. Don't forget ".*/" to match hidden directories!

@barubary suggested a solution using the "tree" command. Strictly speaking, the "tree" package is not part of the core Linux OS, so it falls outside the rules I've set for myself. But here's the solution:

```
tree -daL 1
```

This means show all files ("-a") including hidden files, but display directories only ("-d"). "-L 1" is like the "-maxdepth" control on "find"-- only display output from the current directory.

#Linux #DFIR #CommandLine #Trivia

2023-01-30

Where does Linux store password expiration, aging, etc timeouts? How can you display this information on a live system?

#Linux #DFIR #CommandLine #Trivia

2023-01-31

Yesterday's Linux DFIR command line trivia was not as much of a mystery to folks as I assumed. Several people checked in with the correct answer, but it looks like @deeseearr was quickest off the mark.

Password expiration, aging, and other timeout values are stored in /etc/shadow. That's what all those extra fields after the password hash are for.

While you could set these values by editing /etc/shadow directly, the recommended user interface is the "chage" command. "chage -l hal" would list the current values for user "hal". But other "chage" options allow you to set these values as needed.

Note that other Unix systems may use the "passwd" command to set these values rather than having a separate "chage" command like Linux does.

#Linux #DFIR #CommandLine #Trivia

2023-01-31

You are looking for a string of interest in a collection of files in a nested set of directories. Write a Linux command to output the number of matching lines in each file and the file name.

#Linux #DFIR #CommandLine #Trivia

2023-02-01

Yesterday's Linux DFIR command line trivia asked how to walk a directory hierarchy looking for files containing a particular string and then output a count of matching lines and the name of each file.

These kinds of questions reveal just how old I am in "Unix years", because my natural approach would be something like this:

```
find /usr/include -type f |
while read file; do
echo -e $(grep -a PATH_MAX "$file" | wc -l)\\\\t$file
done
```

We use "find" to locate the regular files ("-type f") then loop over each file path. Inside the loop we calculate the matching lines with "grep ... | wc -l" and then use "echo" to output that number and the file name. Note that I am using "grep -a" here, just in case my collection of files includes binary data like executables (/usr/include should be only text files, but "-a" has become a habit for me).

And I learned to do it this way because I was raised in a (pre-POSIX) time when "grep" didn't have the "-c" argument to return a count of the matching lines. The easier and more modern approach is:

```
grep -Farc PATH_MATH /usr/include | sed -r 's/(.*):([0-9]+)/\\2\\t\\1/'
```

Props to @avuko for being the first to check in with the "grep -rc" solution. Note that I've added the "-F" option to "grep" since we're searching for a fixed string and not a regular expression. The extra "sed" expression is tacked onto the end there so that the output is "count<tab>file name" rather than the standard "file name:count" output from "grep".

Aside from being easier to read (IMHO), putting the counts first makes it easier to do things like:

```
grep ... | sed ... | grep -v ^0
```

This will eliminate output for files with no matching lines. Or we could sort by the number of matching lines with:

```
grep ... | sed ... | sort -n
```

Anyway, this is a neat little trick if you quickly need to find a string of interest in a collection of files.

#Linux #DFIR #CommandLine #Trivia

2023-02-01

The standard Apache "combined" log format is:

IPaddr - - [date] "method URI protocol" code bytes "referer" "user-agent"

Write a Linux command to output:

count IPaddr user-agent

Here "count" is the number of times each "IPaddr user-agent" combo appears. The output should be sorted by IP address.

#Linux #DFIR #CommandLine #Trivia

2023-02-02

Yesterday's Linux DFIR command line trivia asked how to count the number of unique [IP address, user-agent] pairs from the Apache combined log file format. Why is this useful? You can often spot malware going active on an endpoint when you start seeing new user-agent activity in your web proxy logs that differs from the user's normal web browser.

The tricky part is extracting the information we need from each line in the log file. The log file is space delimited, which would normally be a job for "awk". But user-agent strings usually contain spaces, which makes things difficult.

Ultimately, it seemed to me to be a job for "sed":

```
sed -r 's/([^ ]+) .* "([^"]+)"/\\1\\t\\2/' access.log
```

The IP address is the first clump of non-space characters ("[^]+") at the start of the line. The user-agent string is the last quoted string on the line. The " .* " in the middle eats up all the other characters. Then we just replace all that with the IP address, a tab character, and the user-agent string without the enclosing quotes.

@smlx checked in with an interesting solution that used "cut" instead:

```
cut -d' ' -f1,12- access.log
```

This is leveraging the fact that the user-agent is the last field on the line. So any space-separated string from field 12 onward is the user-agent. Scott's version ends up keeping the quotes around the user-agent string, but no big deal there.

Either way, the rest of the challenge is just using "sort" and "uniq" to get the output we want:

```
cut -d' ' -f1,12- access.log | sort -V | uniq -c
```

"sort -V" is a trick we discussed previously for sorting IP addresses. It's really intended for sorting version numbers like 2.4.55, but it works great on IPv4 addresses. "uniq -c" just counts the unique pairs once they are sorted.

#Linux #DFIR #CommandLine #Trivia

2023-02-02

Why do old fogey Sys Admins like me get twitchy when they see cron jobs scheduled to run between 2am and 3am? Why do modern Sys Admins not care?

#Linux #DFIR #CommandLine #Trivia

2023-02-03

Oh my. Yesterday's Linux DFIR command line trivia seems to have hit a nerve. I asked why do old-timey Sys Admins get twitchy when we see cron jobs scheduled between 2am and 3am?

The answer is Daylight Savings Time (DST). In olden times, that time shift would mean that your cron jobs got skipped in the Spring and run twice in the Fall. I can remember auditing our crontabs twice every year just to make sure that nobody had messed up and scheduled stuff during the magic hour.

The reality is that "Vixie cron" fixed this issue decades ago. @barubary even supplied the relevant section of the "cron" manual page:

"Local time changes of less than three hours, such as those caused by the Daylight Saving Time changes, are handled in a special way. This only applies to jobs that run at a specific time and jobs that run with a granularity greater than one hour. Jobs that run more frequently are scheduled normally.

If time was adjusted one hour forward, those jobs that would have run in the interval that has been skipped will be run immediately. Conversely, if time was adjusted backward, running the same job twice is avoided."

I'm still twitching though..

#Linux #DFIR #CommandLine #Trivia #History

2023-02-03

After an intrusion, you find the following line in /etc/rc.d/rc.local:

```
echo /tmp/.ICEd-unix/.start.sh | at now + 5 minutes
```

What does this line accomplish? Where would you look to see if these commands executed? Why do you find no evidence of the commands actually running?

I'll be back Monday with the answers!

#Linux #DFIR #CommandLine #Trivia

2023-02-06

Friday's Linux DFIR command line trivia asked about finding the following line in /etc/rc.d/rc.local:

```
echo /tmp/.ICEd-unix/.start.sh | at now + 5 minutes
```

Although deprecated, the rc.local mechanism is still part of the Linux boot process. The line above is how you would configure an "at" job to execute the script /tmp/.ICEd-unix/.start.sh about five minutes after the system boots. You might see attackers creating this little hook for persistence, so their malware is restarted when the system reboots.

While the job is still pending, you'll find a record under /var/spool/cron/atjobs (sometimes /var/spool/at, depending on your distro). However, the job file gets deleted once the job executes. You might be able to find the job file by searching unallocated for "# atrun uid=".

When the job executes, you will find a log in whatever logfile your "cron" logs go to-- often /var/log/cron.log. Unfortunately the log message just notes that a job was executed as a particular user, but gives no indication exactly what the job executed. If "auditd" is enabled on your system, it will also log the "at" job being executed but again without details of exactly what was run.

Actually, the attacker just adding the line to rc.local will not get the "at" job to fire. The rc.local mechanism has been deprecated to the point that the execute bits have been removed from rc.local. If you or the attacker want to use this mechanism, you also have to "chmod +x /etc/rc.d/rc.local" to get your commands to run at boot time.

#Linux #DFIR #CommandLine #Trivia

2023-02-06

On a live system, how can you get a list of which services are configured to start at boot time? How can you get a list of services that exist on the system, but which are not configured to start at boot?

#Linux #DFIR #CommandLine #Trivia

2023-02-07

Yesterday's Linux DFIR command line trivia asked how you could get a list of services that are configured to start at boot time and how to get a list of services which exist on the system but which are not configured to start at boot.

I'm going to give the win to @lamitpObuS for correcting my original (incorrect) solution:

systemctl list-units --state=[active|inactive]

Actually "--state=active" is the default if you run "systemctl list-units".

Note that by default "list-units" shows everything, including items like the file system mounting targets and network configuration, etc. If you really just want to see the normal Linux background services that are configured on the box, add the "--type" option:

systemctl list-units --type=service

That will get you a list of the active background services.

systemctl list-units --type=service --state=inactive

Will get you the ones that are not running.

Note that you can also run:

systemctl list-units --all

This will give you one listing showing not only active and disabled unit files, but even items that have not yet been installed.

#Linux #DFIR #CommandLine #Trivia

2023-02-07

If all you have is a disk image of a Linux system, how can you determine what services were configured to start at boot time?

(You saw this one coming after yesterday's question, didn't you?)

#Linux #DFIR #CommandLine #Trivia

2023-02-08

Yesterday's Linux DFIR command line trivia asked how you figure out which services were supposed to start at boot time given only a disk image to work from. This one seems to have scared everybody off. I guess systemd can be a little daunting.

The first step is to figure out the default state the machine is configured to boot into. You will find a link called "default.target" under /etc/systemd/system or /usr/lib/systemd/system that points to the default boot target. This will typically be either the "multi-user.target" or the "graphical.target". "graphical.target" is basically everything in "multi-user.target" plus starting up the graphical login on the system console.

Once we know the default boot target, we can work backwards to figure out the milestones that get met along the way. For example, here's the contents of /usr/lib/systemd/system/graphical.target on my lab system:

```
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

The graphical.target "Requires" that we execute the "multi-user.target" milestone first. Similarly, we can work backwards and look at the contents of /usr/lib/systemd/system/multi-user.target:

```
[Unit]
Description=Multi-User System
Documentation=man:systemd.special(7)
Requires=basic.target
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
AllowIsolate=yes
```

multi-user.target requires us to satisfy the basic.target milestone. If you keep following the trail of "Required" configuration backwards you end up with the following milestones: sysinit, basic, multi-user, graphical.

For each milestone, you will normally find a directory under /etc/systemd/system called "<target>.wants", e.g. /etc/systemd/system/multi-user.target.wants. Each of these directories contains links to the unit files that need to be activated to complete each milestone. The individual unit files generally live under /usr/lib/systemd/system, although some site-specific unit files may live under /etc/systemd/system.

It's worth noting, however, that there is no basic.target.wants directory. Let's look at /usr/lib/systemd/system/basic.target:

```
[Unit]
Description=Basic System
Documentation=man:systemd.special(7)
Requires=sysinit.target
Wants=sockets.target timers.target paths.target slices.target
```

```
After=sysinit.target sockets.target paths.target slices.target tmp.mount
RequiresMountsFor=/var /var/tmp
Wants=tmp.mount
```

basic.target triggers multiple other targets and also requires mounting /var and /var/tmp. But it doesn't trigger any services directly.

But our list of milestone targets has expanded to: sysinit, basic, sockets, timers, paths, slices, multi-user, graphical. Use the "<target>.wants" directories and the target files themselves to fully enumerate everything that executes during the boot process.

#Linux #DFIR #CommandLine #Trivia

2023-02-08

Give me a Linux command line to pull down a complete mirror copy of a (suspicious) web site.

#Linux #DFIR #CommandLine #Trivia

2023-02-09

Yesterday's Linux DFIR command line trivia asked how to mirror a web site from the Linux command line.

I admit that I posed this question hoping to learn something new. But the solutions I got all revolved around the tried and true "wget --mirror" option:

```
wget --mirror --page-requisites -o wget.log blog.commandlinekungfu.com
```

@deeseearr suggested the "--page-requisites" option to download embedded images, style sheets, Javascript, etc. DC also points out that this approach can be easily thwarted by a malicious web site that notices what you are trying to do and for example tar pits you into a dynamically generated hierarchy of infinite web pages. Ultimately you may want to use a headless browser framework to (more) safely scrape the site, but that gets us beyond simple Linux built-in tools.

#Linux #DFIR #CommandLine #Trivia

2023-02-09

Why is this command line useful?

```
ls -li /usr/bin | sort -n
```

#Linux #DFIR #CommandLine #Trivia

2023-02-10

Yesterday's Linux DFIR command line trivia asked why this command might be useful:

```
ls -li /usr/bin | sort -n
```

Several people correctly identified that this would produce a detailed directory listing sorted by inode number. But why should we care?

This is a quick and dirty form of "outlier analysis". When the operating system is installed and binaries are stuffed into /usr/bin, they're likely going to get nearly sequential inode numbers. Over the lifetime of the system, patches may add binaries in /usr/bin, and those new files will get their own tight clusters of inode values. When you sort the directory by inode number, you'll see those clusters of inode numbers, but in general the dates on the files should be increasing with the inode numbers.

Now suppose an attacker drops a bunch of malware into /usr/bin. Again, those new files are going to tend to get tightly clustered inode numbers, so it will be easy to see which files came in together. Also, if the attacker tries to set timestamps on the files to make them "blend in", sorting the directory by inode number will make the timestomped files stand out, because the dates will no longer be increasing with the inode numbers.

This isn't a perfect investigative mechanism by any means. For example, the attacker could overwrite an existing binary. The new binary would get the inode of the original file. The attacker could then timestomp the new binary to make the timestamps look like the original file. Though if the attacker fails to fix the creation date on the file, it will really make their change stand out... if the analyst thinks to check the creation dates and not just the last modified date that "ls -li" shows by default.

The other issue is that this sort of outlier analysis is generally only useful on directories like /usr/bin that are relatively static. Trying to do this on a busy user home directory or directory like /var/log or /tmp is going to be a mess.

#Linux #DFIR #CommandLine #Trivia

2023-02-10

Write a Linux command line to locate only pure ASCII text files in a directory and print the path names of these files. This would include things like scripts, startup files, regular text files, etc.

#Linux #DFIR #CommandLine #Trivia

2023-02-13

Friday's Linux DFIR command line trivia asked for a Linux command line to find all pure ASCII text files under a given directory hierarchy.

I got a number of solutions that combined "find" and "file", and looking for output from "file" that included "ASCII" and/or "text. And it's true that many text file types are reported by "file" with "ASCII text" somewhere in the output. But as @smoot points out, there are other common text file types that are reported by "file" as:

Certificate
CSV text
JSON data
Perl script text executable
PostScript document text

This caused some people to lose all faith in the "file" command. For example, @polyna checked in with this solution:

```
find /usr/share -type f \\( -exec env LC_ALL=C grep -Pq '[^[:ascii:]]' '{}' \\; -o -print \\)
```

This works, albeit a bit slowly, since it calls "grep" individually on each file. The "-q" option at least aborts the "grep" as soon as any non-ASCII characters are found. Also, "-P" is needed for Perl-style regular expressions so that the "[:ascii:]" character class is supported. Anyway, it's an interesting use of "-exec" as a negative conditional in "find".

@smoot went with the all "grep" solution:

```
grep -ErL '[^[:print:]]' /usr/share
```

The trick is the "-L" option which means "show the names of files that DO NOT match".

Steve is using "[^[:print:]]" here since extended regular expressions don't support the "[:ascii:]" character class. Unfortunately, "[^[:print:]]" doesn't include the tab character and some files end up being incorrectly eliminated. I believe this slight tweak is the best solution to the problem:

```
export LC_ALL=C
grep -ErL '[^[:print:][:space:]]' /usr/share
```

I'm borrowing the environment variable setting from @polyna's solution to force us into the most restrictive US ASCII locale. Then I add the "[:space:]" class to @smoot's original pattern. It's teamwork that makes the dream work!

Btw, before the others weighed in with their suggestions, I was working on a solution that leveraged "file -i":

```
find /usr/share -type f -print0 | xargs -0 file -i | grep -F charset=us-ascii | cut -d: -f1
```

But it turns out the all "grep" solution is both much faster and more accurate. "file -i" incorrectly identifies some short ASCII files of symbols as "charset=binary" rather than "charset=us-ascii". This is perhaps the only time my Linux life when the "file" command has been really underwhelming.

#Linux #DFIR #CommandLine #Trivia

2023-02-13

You have multiple directories of files from different pieces of evidence. You want to find any files that appear in two or more of the different evidence collections.

#Linux #DFIR #CommandLine #Trivia

2023-02-14

Yesterday's Linux DFIR command line trivia was clearly a case of hubris on my part. A little checksumming, a little "sort", a little "uniq -D", and we should be good to go, right? Um, yeah, about that...

The challenge is you have multiple directories of evidence. You are looking for files that are common to at least two different directories. I wanted to cover three basic test cases:

1) Same file, multiple different directories: REPORT ALL DUPLICATES FOUND

2) Same file, multiple different directories, including multiple instances in the same directory: REPORT ALL INSTANCES

3) Same file, same directory: DO NOT REPORT

This would have been a shorter lesson if I had a "sort" command that could report duplicates like "uniq -D" or a "uniq" command that could ignore later fields as opposed to only being able to skip initial fields. Alas, none of these things exists... so I kind of ended up writing my own.

First I need a sorted list of file checksums. This at least was straightforward:

```
find dir1 dir2 dir3 -type f -print0 | xargs -0 md5sum | sort
```

But what to do with this list. I realized that basically I was looking for the same checksum in different directories. So I wrote some code to handle that:

```
find dir* -type f -print0 | xargs -0 md5sum | sort |
while read sum file; do
dir=${file%%/*}
[[ $sum == $lastsum && $dir != $lastdir ]] && echo -e $lastsum\\\\t$lastfile\\\\n$sum\\\\t$file
lastsum=$sum; lastfile=$file; lastdir=$dir
done | uniq
```

Woof, there's a lot to unpack here. I get the directory name using a fairly obscure bash variable expansion. "${var%%pattern} strips whatever matches "pattern" off the right-hand side of the variable ("${var##pattern}" is the same bur from the left-hand side). So here we're stripping everything from the first "/" onwards.

If the current checksum matches the previous checksum and the directory names are different, then we output the previous checksum and filename along with the current checksum and filename. This can result in duplicate output lines when we have the same file in several different directories, which is why there is a trailing "uniq" statement after the loop to clean up the final output.

This solution handles test cases #1 and #3, but unfortunately fails on #2. I could have hacked together a complex solution using indexed arrays, but I've been burned too many times by arrays being non-portable in old versions of bash. So I decided to go with a multi-step solution, that's a little bit easier to comprehend:

```
find dir* -type f -print0 | xargs -0 md5sum | sort >checksums.txt
while read sum file; do
dir=${file%%/*}
[[ $sum == $lastsum && $dir != $lastdir ]] && echo $sum
lastsum=$sum; lastdir=$dir
done <checksums.txt | uniq > common-sums.txt
grep -Ff common-sums.txt checksums.txt
```

We dump the sorted hashes to a file called "checksums.txt". The loop outputs the unique checksums that appear in multiple directories to a file called "common-sums.txt". We use the "common-sums.txt" file as a list of patterns to pull the files we want out of the original "checksums.txt" file.

Meh. I guess if it was easy, everybody would be doing it.

#Linux #DFIR #CommandLine #Trivia

2023-02-14

Write a Linux command to output all users from the /etc/passwd file along with a list of all groups they are assigned to-- including their primary group from the /etc/passwd file and any additional groups they belong to in /etc/group. You only have a system image to work from, not a live system.

(Oh boy I hope this one doesn't suck as much as yesterday's challenge...)

#Linux #DFIR #CommandLine #Trivia

2023-02-15

I broke the solution for yesterday's Linux DFIR command line trivia up into three pieces.

The first piece was to parse the passwd file. This is relatively straightforward:

```
IFS=:
while read user junk uid gid rest; do
echo $user $gid
done <passwd
```

Setting "IFS" means the shell will tokenize each line for us. All we have to do is read the fields into the appropriate variables.

The second part of the challenge is to convert the numeric GID from the passwd file into a group name. You could do this with "grep", but I went with "awk" because I'm hard core like that:

```
awk -F: "\\$3 == $gid {print \\$1}" group
```

We want the "$gid" variable to be expanded by the shell, but have to be careful to backwhack the "$" signs in front of the "awk" variables so that they get passed through to "awk" as is. Quoting is such a pain sometimes.

The third part of the solution is to run through the group file and look for any other groups the user might have been added to:

```
awk -F: "\\$4 ~ /(^|,)$user(,|$)/ {print \\$1}" group | tr \\\\n ' '
```

The regular expression here is complicated because I want to match the exact user name and not accidentally do a substring match. We may get multiple lines of output if the user belongs to several groups, and the "tr" command at the end merges the output onto a single line.

Now all we have to do is put everything together:

```
IFS=:
while read user junk uid gid rest; do
echo $user $(awk -F: "\\$3 == $gid {print \\$1}" group) $(awk -F: "\\$4 ~ /(^|,)$user(,|$)/ {print \\$1}"
group | tr \\\\n ' ')
done < passwd
```

Being able to create these sorts of ad hoc reports is why I like having Linux as my DFIR platform!

#Linux #DFIR #CommandLine #Trivia

2023-02-15

Write a Linux command line to traverse a directory of files and summarize the number of instances of each file extension found (e.g., .docx, .jpg, .pdf, etc). Fold all file extensions to lower case-- ".JPG" and ".jpg" should be reported together. Files that have no extension should be reported as "other". List the extensions found in descending numeric order by the number of occurrences of each extension.

#Linux #DFIR #CommandLine #Trivia

2023-02-16

I took yesterday's Linux DFIR command line trivia from one of our old Command Line Kung Fu blog postings (Episode 99). It's an interesting challenge and useful for quickly summarizing types of files in a directory by simply collecting and counting the file extensions.

I also liked it because we get to use the "${var##pattern}" expansion, similar to my solution from a couple of days ago using "${var%%pattern}". So there's some nice symmetry.

If you read the original blog posting, my first solution used a funky "sed" substitution to whittle down to the file extensions. It was actually loyal reader and friend of the blog Jeff Haemer who suggested this much cleaner version (and to @barubary who reminded me to quote my variables) :

```
find Documents -type f |
while read f; do
echo "${f##*.}";
done |
sed 's/.*\/.*/other/' | tr A-Z a-z |
sort | uniq -c | sort -n
```

"find" gets us a list of files in the directory and then we feed that into a loop that uses "${f##*.}" to remove everything up until the final "." in the file path.

Some file names are not going to have a "." and so the original file path will be unchanged. The "sed" expression after the loop marks these files as being in category "other". Finally we shift everything to lowercase so "GIF" and "gif" are recognized as the same type. Then our usual command line histogram idiom rounds things out.

We could add some other fix-ups to make things nicer:

```
find Documents -type f |
while read f; do
echo "${f##*.}";
done |
sed 's/.*\/.*/other/' | tr A-Z a-z |
sed -r 's/^jpg$/jpeg/; s/(~|,v)$//' |
sort | uniq -c | sort -n
```

I've dropped in another "sed" expression before we start counting things. We make "jpg" and "jpeg" count the same and remove some trailing file extensions for backup files to reduce clutter in the output. This just shows that you can arbitrarily tweak the output to suit your needs.

Props to @apgarcia for getting very close to the final solution on this one!

#Linux #DFIR #CommandLine #Trivia

2023-02-16

An attacker has created a file whose name is all non-printable control characters. How can you remove this file from your file system?

#Linux #DFIR #CommandLine #Trivia

2023-02-17

Yesterday's Linux DFIR command line trivia asked how to deal with file names that contain only non-printable control characters. This is a classic Linux/Unix trivia question, but when I wasn't looking GNU coreutils made the answer a whole lot easier.

The classic solution for this problem is to delete the file by its inode number:

```
# ls -i
783401 "$'\\030\\024\\031'
# find -inum 783401 -delete
```

This one is such a classic that multiple people chimed in with the correct answer. Credit to @koblas for being quick off the mark and also giving several other possible solutions.

But hold on there. Back up a bit. What is that crazy quotes representation of the file name about? Well that's what GNU coreutils refers to as "shell-escape" style quoting. And it's been the default terminal output style for "ls" since coreutils v8.25 (released 2016-01-20). What you are looking at is the file name represented as a series of octal numbers in the quoting style that bash recognizes.

So for example you can do:

```
touch "$'\\030\\024\\031'
cp /some/file "$'\\030\\024\\031'
rm "$'\\030\\024\\031'
```

And that just made this question a whole lot easier, right? Before 2016, and on lots of non GNU userlands even today, "ls" outputs the raw control characters, which is why everybody learned the "delete via inode" trick.

@polyna points out that this is only the default format if the output is going to your terminal window. If you pipe the output into some other program like "less" or "grep" then you are going to see the raw control characters. Also your Linux distro may unhelpfully alias "ls" to something that reverses the standard quoting behavior. You can force the right thing to happen with “ls --quoting-style=shell-escape”.

Some folks also brought up a related trivia question, which is "How do you remove a file named '-i'?". If you try to naively "rm -i", the file name gets parsed as an argument and the wrong thing happens.

You can always use the "remove by inode trick" here. But there are other work-arounds in this case:

```
rm -- -i
rm ./-i
```

"--" tells the "rm" command that the argument switches are done with and everything else is a file name. Or you prefix the "-i" with "./" ("remove the file called '-i' in the current directory") so that it doesn't look like a command line argument.

#Linux #DFIR #CommandLine #Trivia

2023-02-17

Write a Linux command line to find all executables that are group or world writable or which are installed in group or world-writable directories.

#Linux #DFIR #CommandLine #Trivia

2023-02-20

Friday's daily Linux DFIR command line trivia asked you to find executables that were installed in an unsafe fashion-- where either the executable itself is group or world writable it is installed in a directory that is group or world writable.

I did this one in two parts. First is finding executables that have group or world write permissions:

```
find / -type f -perm /111 -perm /022 2>/dev/null
```

We're leaning heavily on an innovation in GNU find, which is the "-perm /..." construct. This means show me files that match any of these permission bits (yes, you can also use it with symbolic mode letters too).

So here we have a file ("-type -f") that has at least one execute bit set ("-perm /111") and is group and/or world writable ("-perm /022"). Add a little output redirection so that we don't get annoying error messages when our "find" command traverses directories like /proc and we are good to go.

I honestly puzzled about the second part of the challenge for a while and went in several failed directions. And then it occurred to me that group/world writable directories are just not that common in Linux. And so I went with:

```
find / -type f -perm /111 2>/dev/null | grep -Ff <(find / -type d -perm /022 2>/dev/null)
```

First we find all executables. Then we pipe that output into "grep -f" using the "<(...)" syntax to use command output as the pattern file to "grep". The command is the "find" command that gives us a list of group/world writable directories ("-type d -perm /022"). Note that I'm using "grep -F" here which treats all input lines as plain text without regular expression matching, just in case any of the directory names contain characters that are special in regular expressions.

As a first pass, this is pretty effective. But I realized there is a way this solution can produce incorrect results. We all know that the /tmp directory is both group and world writable, so "/tmp" is going to be one of our input patterns for "grep". Now supposed you had an executable installed as /root/tmp/myexe. The "/tmp" pattern would match against this directory even if all of the permissions on /root/tmp/exe were safe.

And so I ended up with this solution instead:

```
find / -type f -perm /111 2>/dev/null | grep -f <(find / -type d -perm /022 2>/dev/null | sed -r 's/
([^[:alnum:]])/\\\\\\1/g; s/(.*)/^\\1\\/[^\\/]*$/')
```

It's the same basic idea, but now we are turning each directory path into a regular expression (notice I drop the "-F" option from "grep). The first part of the "sed" expression "backwhacks" all non-alphanumeric characters. A path like

```
/tmp/.X11-unix
```

gets converted to

```
\\/tmp\\/\\.X11\\-unix
```

The second part of the "sed" expression adds "^" to the beginning of each expression and "/[^/]*$" to the end of each path:

```
^\\/tmp\\/\\.X11\\-unix/[^/]*$
```

So now our paths are rooted to the start of each line ("^") and are not allowed to match any additional subdirectories after the writable directory path ("/[^/]*$"). It's a little gnarly, but it works.

EDITED LATER - Well that was certainly a lot of "sed", but @barubary suggested a cleaner solution:

```
find / -type d -perm /022 -exec find '{}' -maxdepth 1 -type f -perm /111 \\; 2>/dev/null
```

First find all the group or world writable directories, then exec another find command to look for executables in those directories. Note the "-maxdepth 1" to prevent descending into subdirectories. This solution is nicer on a lot of levels-- not the least of which is that it provides a template for solving many similar problems.

#Linux #DFIR #CommandLine #Trivia

2023-02-20

As long as we are on the subject of unsafe executable installations, write a Linux command line to find all executables that are installed in directories that are not owned by the root user.

#Linux #DFIR #CommandLine #Trivia

2023-02-21

Yesterday's Linux DFIR command line trivia asked for a command line to find executables installed in non-root owned directories. This is another common misconfiguration that can lead to privilege escalation in some cases.

@barubary checked in with the winning answer:

```
find / -type d \\! -user root -exec find '{}' -maxdepth 1 -type f -perm /111 \\; 2>/dev/null
```

Mmmm, tasty! A "find" command that execs another "find" command. The first "find" locates directories not owned by the root user ("-type d \\! -user root") and then the second "find" looks for any executables in those directories. Note the use of "-maxdepth 1" to avoid descending into subdirectories.

@barubary also points out that this is a pattern for a cleaner solution for the previous challenge where we were looking for executables in group or world writable directories. Rather than the "sed" monstrosity I cooked up, we can just:

```
find / -type d -perm /022 -exec find '{}' -maxdepth 1 -type f -perm /111 \\; 2>/dev/null
```

First find the group or world writable directories and then look for executables in those directories. Easy peasy lemon squeezy!

#Linux #DFIR #CommandLine #Trivia

2023-02-21

Write a Linux command line to print every third line from a text file.

#Linux #DFIR #CommandLine #Trivia

2023-02-22

Yesterday's Linux DFIR command line trivia asked for a way to print every 3rd line in a file. I actually used this one in an investigation when I was extracting USB serial numbers from a data file, but it's generally useful with any sort of regularly formatted text file.

Also, I was hoping to see people weigh in with multiple solutions, and I was not disappointed.

"awk" and "sed" were the most popular, so let's just focus on those. @koblas checked in with:

```
awk 'NR % 3 == 0 { print $0 }' input.txt
```

In "awk", "NR" is the "record" or line number and "%" is modulus. So every third line this prints out the entire line ("$0").

"{ print $0 }" is actually the default action, which led to @barubary's more terse version:

```
awk 'NR % 3 == 0' input.txt
```

@davidpostill further reduced this to:

```
awk '!(NR%3)' input.txt
```

And I think that about does it for "awk".

On the "sed" side of the house, @barubary chimed in with the classic solution:

```
sed -n '3~3p' input.txt
```

"-n" changes the default behavior of "sed" so that it will print nothing by default. Then '3~3p' means "print every third line ("~3") starting with the third line (that's the first three in "3~3").

@davidpostill had an interesting alternative:

```
sed -n 'n;n;p;' inut.txt
```

Again we see "-n" to make the default not to print. In the quotes, the "n" means "read a line but don't print" and the "p" means read and print a line.

Linux means there's always more than one way to do it. If it gets the job done for you, then you do you!

#Linux #DFIR #CommandLine #Trivia

2023-02-22

You have a large directory of log files named:

"basename.YYYYMMDD[.gz]"

The base name of the log file varies and some files are gzipped while others are not.

You want to move all of these files to a new directory structure broken out by year and month:

newdir/YYYY/MM/basename.YYYYMMDD[.gz]

Write a Linux command line to make this happen.

#Linux #DFIR #CommandLine #Trivia

2023-02-23

Yesterday's Linux DFIR command line trivia asked you to move a directory full of files named "<something>.YYYYMMDD[.gz]" into hierarchy of directories like "<newdir>/YYYY/MM/<something>.YYYYMMDD[.gz]". @apgarcia checked in with a nice solution:

```
for f in *; do
YMD=$(echo $f | cut -f2 -d.)
YEAR=${YMD:0:4}
MONTH=${YMD:4:2}
mkdir -p /my/new/dir/${YEAR}/${MONTH}
mv $f /my/new/dir/${YEAR}/${MONTH}
done
```

Great use of the "${var:offset:length}" expansion to chop up the date elements. "mkdir -p" lets you create arbitrarily deep directories with a single command.

All the solutions I received focused on parsing the file names. But what if we did something more like this:

```
for y in {2020..2023}; do
for m in {01..12}; do
mkdir -p /my/new/dir/$y/$m
mv *.$y$m* /my/new/dir/$y/$m
done
done
```

You can pick the parameters for the outer loop based on the set of files that you have. If possibly ending up with some empty directories bothers you, you can always "find /my/new/dir -type d -empty -delete" afterwards to clean things up.

It's easy to get tunnel-visioned onto a particular solution. But maybe the first way the solution suggests itself is not always the best way.

#Linux #DFIR #CommandLine #Trivia

2023-02-23

How about another date-based trivia question? This time you have a text file where the first field is a date in the American style "MM/DD/YYYY" format (let me apologize to the rest of the world for what can only be described as a terrible mistake). How do you sort the lines in the file by date without changing the text on each line?

#Linux #DFIR #CommandLine #Trivia

2023-02-24

Yesterday's Linux DFIR command line trivia asked how to sort lines of a text file on an initial date field in the American date style, "MM/DD/YYYY", without rewriting the line.

Shout out to @mish for checking in with a solution to leverage the ability to sort on fields:

```
sort -t/ -n -k3,3 -k1,1 -k2,2 input.txt
```

"-t/" means break up the fields on "/" characters rather than spaces, and "-n" means we're going to sort numerically rather than alphabetically. "-k" lets us specify the "key fields" we want to sort on: year first, then month, then day.

Yay for "sort"!

#Linux #DFIR #CommandLine #Trivia

2023-02-24

For the weekend, let's do a little longer command line trivia challenge suggested by @trimyak. The challenge is validating the package binaries installed on your system. But of course there are multiple possible scenarios here:

-- RedHat vs Debian package managers
-- How about an externally supplied list of known good hashes?
-- How about an externally supplied list of known bad hashes?
-- Live system vs dead system image

How many of these different scenarios can you come up with solutions for?

#Linux #DFIR #CommandLine #Trivia

2023-02-27

Friday's Linux DFIR command line trivia asked about validating system executables under a variety of circumstances.

Probably the easiest case is validating packages on a live system. On Red Hat you can "rpm -Va" and on Debian derivatives you can "dpkg -Va".

Doing the same thing on a mounted image is not much more difficult since both "rpm" and "dpkg" support a "--root" option to specify an alternate root directory to perform operations on. Hmmm, it's almost as if they were striving to be feature-compatible or something.

The question also asked about checking executables against lists of "known good" and "known bad" hashes. First we need to hash our own executables, something we've covered in previous questions:

```
find / -type f -perm /111 -print0 | xargs -0 md5sum
```

Substitute whatever hashing algorithm you need for "md5sum".

If you want to filter out "known goods":

```
find / -type f -perm /111 -print0 | xargs -0 md5sum | grep -Fvf known-goods.txt
```

In other words, show me all the files that don't match one of the known goods in "known-goods.txt".

Looking for known bads means just dropping the "-v" argument:

```
find / -type f -perm /111 -print0 | xargs -0 md5sum | grep -Ff known-bads.txt
```

Show me anything that matches a hash in "known-bads.txt".

And if you want to do this on a mounted system image rather than the live system, just change the directory you specify at the front of the find command:

```
find /path/to/image ...
```

So although there are many possible scenarios, none of them are actually that difficult to solve.

#Linux #DFIR #CommandLine #Trivia

2023-02-27

List all files in a directory that have common base names but different file extensions after a ".". For example, {foo.c, foo.h, and foo.o} or {cron.allow and cron.deny}. List files only and not subdirectories-- so for example "modprobe.conf" and "modprobe.d" DO NOT count.

#Linux #DFIR #CommandLine #Trivia

2023-02-28

Why do I do these things to myself? Yesterday's Linux DFIR command line trivia asked for a command to find all files in a directory that shared a common base name. For example, if you're looking for evil.exe and evil.dll installed in the same directory in an image.

If I hadn't restricted myself to matching only files, we could do something like this:

```
ls -d $(ls | sed -r 's/(.*\\.)[^.]*/\\1*/' | sort | uniq -d)
```

Inside the "$(...)" I'm changing all "something.whatever" file names into "something.*" patterns. Then I only output the ones that have duplicates. The outer "ls" statement then lists all the matching patterns, but unfortunately doesn't discriminate regular files from other objects. In fact, I had to add the "-d" option to the outer "ls" command so that we wouldn't list the contents of matching directories.

So, back to the drawing board. I can get a list of the regular files in the directory with:

```
find . -maxdepth 1 -type f
```

That's going to give us an extra "./" at the front of our file names (e.g., "./something.whatever"). But I can take care of that by adding some "sed" tweakery:

```
ls $(find . -maxdepth 1 -type f | sed -r 's/.\\//; s/(.*\\.)[^.]*$/\\1*/' | sort | uniq -d)
```

This is really close but it ultimately fails in some cases. For example, when I run it in my /etc directory the command matches the files "ld.so.cache" and "ld.so.conf" but unfortunately also matches "ld.so.d" which is a directory.

So, fine then, here we go:

```
find . -maxdepth 1 -type f | grep -Ff <(find . -maxdepth 1 -type f | sed -r 's/(.\\/[^.].*\\.)[^.]*$/\\1/' | sort |
uniq -d) | sort | sed 's/.\\//'
```

Yes, I'm using the "find" command twice. First I get a list of the files in the current directory. That gets piped into a "grep -f" command where the pattern list for the "-f" comes from a modified version of our earlier "find ... | sed" nonsense. In this case I'm converting the "./something.whatever" file names into "./something." patterns and only outputting the ones that are duplicated. That will give us just the regular files that have duplicated base names.

However, the file names come out in the order that "find" pulls them from the directory, which is not necessarily alphabetical order. So we add a "sort" at the end... and I little more "sed" to remove the unsightly "./" from the front of each file name.

Honestly, my first solution is probably fine for most cases.

#Linux #DFIR #CommandLine #Trivia

2023-02-28

Write a Linux command line to output only the non-blank, non-comment lines from a Linux configuration file. "Blank lines" are defined to be any line that only contains whitespace. "Comment lines" are those lines that start with "#" (with optional leading whitespace) followed by anything else.

#Linux #DFIR #CommandLine #Trivia

2023-03-01

Whee! The answer to yesterday's Linux DFIR command line trivia turned into a regular expression explosion. And I am always down for that.

Generally the solution is going to be:

```
grep -Ev <pattern> inputfile
```

Where <pattern> is everybody's favorite regular expression to match blank lines and comments.

My preferred method was always:

```
'^[[:space:]]*(#.*)?$'
```

"For the whole line, match zero or more whitespace, with an optional hash mark followed by anything."

@barubary checked in with something similar:

```
'^[[:space:]]*(#|$)'
```

"From the beginning of the line, zero or more whitespace either to the end of the line or up to a hash mark." Clever!

And then @SaThaRiel checked in with:

```
'^\\s*$|^\\s*#'
```

And I said, "'egrep' doesn't do \\s!" And I was wrong, sort of. Extended regular expressions typically did not include sequences like "\\s" for matching whitespace. I confirmed this with GNU grep v3.4 on one of my older machines. But when I try it with GNU grep 3.6 "\\s" actually works with "grep -E"! I can find nothing in the release notes that documents this change-- what the heck?!

"\\s" should always work with "grep -P" ("-P" for Perl compatible regular expressions) regardless of version. So I'm going to go with this as my final answer:

```
grep -Pv '^\\s*(#|$)' inputfile
```

Teamwork is what makes the dream work!

#Linux #DFIR #CommandLine #Trivia

2023-03-01

Oh sure you could use "ssh-keygen -R" to remove outdated entries from your known_hosts file. But how would you remove single lines from other files using the Linux command line? Write a command line to delete the 25th line of the file.

#Linux #DFIR #CommandLine #Trivia

2023-03-02

Yesterday's Linux DFIR command line trivia asked how to delete the 25th line of a file. @barubary checked in with the standard answer:

```
sed -i 25d known_hosts
```

"-i" means "in place" editing of the file-- the file will be replaced with a new version after deleting line 25 ("25d"). You can have "sed" automatically make a backup copy by specifying a file extension for the backup after the "-i":

```
sed -i.orig 25d known_hosts
```

After this command runs, the original version will be saved as "known_hosts.orig" and the modified file becomes your "known_hosts" file.

Of course, you may not always know the line number you want to delete. You can also deleted based on pattern match:

```
sed -i.orig '/192.168.1.10/d' known_hosts
```

Any line(s) matching "192.168.1.10" will be deleted.

Shout out to @AMS and @darkvinge for jumping in with solutions that used "ed". You sick, sick people...

#Linux #DFIR #CommandLine #Trivia

2023-03-02

You're looking through your web content tree after an exploit for encoded web shells left behind by an attacker. These web shells installed as long lines of encoded commands. Locate all files in a directory that have any lines that are longer than 256 characters.

#Linux #DFIR #CommandLine #Trivia

2023-03-03

Yesterday's Linux DFIR command line trivia question asked you to find all files in a directory with a line longer than 256 characters.

I was all set to leverage "wc -L" which gives us the length of the longest line in the file. You know, something like:

```
find /some/dir -type f -exec wc -L {} + | awk '$1 > 256 {print $2}'
```

Note the use of "-exec command {} +" which replaces the old "find ... | xargs command" idiom.

And then @barubary slips in this gem:

```
grep -Erl '^.{257}' /some/dir
```

The pattern matches any line with 257 or more characters. "-r" searches the entire directory and "-l" means only output the file names.

Sigh. Maybe I should retire.

#Linux #DFIR #CommandLine #Trivia

2023-03-03

For a single directory, figure out the number of seconds since a file has been modified in that directory (regular files only, do not look into subdirectories). Now given a list of directories (include their subdirectories as separate elements of the list), sort the list by each directory's "seconds since last file modified" number.

EDITED for clarity

#Linux #DFIR #CommandLine #Trivia

2023-03-06

Friday's Linux DFIR command line trivia asked you to calculate for a single directory the number of seconds since a file has been modified in that directory. Then sort a list of directories by this "file last modified" number.

This one is definitely tricky, and would be impossible without "find ... -printf" to get us going:

```
# find /etc -type f -printf "%T@\\t%h\\n"
1164310390.0000000000 /etc
1599400266.0000000000 /etc/bluetooth
1599400266.0000000000 /etc/bluetooth
1623353696.0000000000 /etc/bluetooth
[...]
```

For each regular file we are outputting the mtime on the file in epoch time format (seconds since Jan 1, 1970) and the directory name.

We could do without the fractional seconds, but a little "sed" and a little "sort" and we're well on our way:

```
# find /etc -type f -printf "%T@\\t%h\\n" | sed -E 's/\\.[0-9]+//' | sort -k2,2 -k1,1nr
1677866431 /etc
1677787353 /etc
1677787353 /etc
[...]
```

"sed" gets rid of the fractional seconds for us. Then "sort" and we have the output grouped together by directory with the newest mtime per directory at the top of each group.

Now for some math:

```
# epoch=$(date "+%s")
# find /etc -type f -printf "%T@\\t%h\\n" | sed -E 's/\\.[0-9]+//' | sort -k2,2 -k1,1nr |
while read sec path; do
[[ "$path" == "$oldpath" ]] && continue
echo -e $(($epoch - $sec))\\\\t$path; oldpath="$path"
done | sort -n
43853 /etc/cups
229848 /etc/ssh
238269 /etc
[....]
```

We feed our sorted output into a "while" loop that looks for the first line of each new directory grouping-- the most recently modified file. We subtract the mtime from the current epoch time to get the "seconds since last modified" value. We output this value along with the directory name. A "sort -n" at the end of the loop sorts the output by the "seconds since last modified" value in the first column.

Thanks "-printf"! This would have been much more gnarly without you.

#Linux #DFIR #CommandLine #Trivia

2023-03-06

Given two directories with the same file and directory names, but with different file contents and perms/ownerships, copy the perms and ownerships from the "master" directory to the new dir while preserving the content that's different in the new directory (i.e., copy the ownerships and perms without overwriting any files).

#Linux #DFIR #CommandLine #Trivia

2023-03-07

Yesterday's Linux DFIR command line trivia asked you to copy the permissions and ownerships from files in one directory onto another directory without copying the file contents.

I'm old enough in Unix years to remember a time when the only solution to this problem was a loop like this:

```
# cd /your/source/dir
# find * -print0 | xargs -0 stat -c '%a %u %g %n' |
while read perms user group file; do
chown $user:$group "/path/to/target/dir/$file";
chmod $perms "/path/to/target/dir/$file";
done
```

But now thanks to the Linux "cp" command we can just do this:

```
cp -r --attributes-only /source /target
```

Shout out to @llutz for chiming in with the modern solution.

#Linux #DFIR #CommandLine #Trivia

2023-03-07

Sort a list of directories by their total size (including the total size of all files, subdirectories, etc), presenting the directory sizes in human-readable units ("3.5G", "228M", etc).

#Linux #DFIR #CommandLine #Trivia

2023-03-08

Yesterday's Linux DFIR command line trivia asked for a list of directories sorted by their total size with the sizes presented in human-readable units. Well @jtk and I are here with the good news that "sort" has a "-h" option for sorting human-readable units:

```
find /usr/share -type d -exec du -sh {} + | sort -h
```

I'm using /usr/share for my list of directories. "du -s" gives me the total size of each directory and "-h" makes the output in human-readable units. Just bang that output into "sort -h" and we're done!

#Linux #DFIR #CommandLine #Trivia

2023-03-08

Copy all files owned by user "hal" from one directory into a new directory, preserving the directory structure of the old directory.

#Linux #DFIR #CommandLine #Trivia

2023-03-09

Yesterday's Linux DFIR command line trivia asked for a command to copy all files owned by user "hal" from one directory to a new directory while preserving the original directory structure.

@barubary checked in with the "all find" solution:

```
cd srcdir
find . -type f -user hal -exec cp --parents -t /path/to/target {} +
```

Props for leveraging the "-t" (target directory) and "--parents" options (recreates the original directories in the target).

@regnil went with the "tar copy" version:

```
cd srcdir
find . -user hal | tar cf - --files-from - | (cd /path/to/target && tar xf -)
```

"--files-from -" has "tar" reading the file names from the standard input and spewing out a tar file with the named files. Then in a subshell we "cd" to the target directory and unpack the archived files being shoved at us on the standard input.

Of course old school me always returns to the classics:

```
cd srcdir
find . -user hal | cpio -pd /path/to/target
```

This is pretty much the only remaining use case for "cpio", but it's a good one. Now you kids get off my lawn!

#Linux #DFIR #CommandLine #Trivia

2023-03-09

How about a little Red Team flavor? Given a file of user names and a file of password guesses, output a merged file like:

```
user1 password1
user1 password2
user1 password3
user1 password4
user2 password5
user2 password6
...
```

In other words, each user gets associated with the next set of four passwords from the password input file. Output each user/password combo on a single line in the merged file.

#Linux #DFIR #CommandLine #Trivia

2023-03-10

Yesterday's Linux DFIR command line trivia asked you to create a file of possible username/password combos from a file of usernames and a file of passwords. Each username would be output four times and joined to the next four entries from the passwords file.

@barubary checked in with a nice, tight solution:

```
sed 'p;p;p' users.txt | paste - passwords.txt
```

"sed" loops over the usernames file and outputs each username three extra times, for a total of four times for each username. This is then used as input for "paste" which slams the usernames together with the passwords.

@barubary points out there will be trailing nonsense when one file is longer than the other. We can take care of that with a little extra "grep" action:

```
sed 'p;p;p' users.txt | paste - passwords.txt | grep -Pv '(^\\t|\\t$)'
```

Our "grep" expression just throws away lines where either the first or second field is empty, leaving a tab at the start of the line or at the end of the line.

#Linux #DFIR #CommandLine #Trivia

2023-03-10

Picking up where yesterday's Linux DFIR command line trivia left off, suppose even after repeating the usernames four times we have less usernames than passwords. Recycle the username list enough times to use up all of the passwords using each password only once but again using each username four times. Output a separate merged file for each trip through the username list.

#Linux #DFIR #CommandLine #Trivia

2023-03-13

At the end of last week, our Linux DFIR command line trivia was creating username/password lists from a file of usernames and a file of passwords. Each username was repeated four times with the next four passwords from the list. This lets us maximize the number of passwords tried without tripping the account lockout after five failed logins.

But suppose we had many more potential passwords than usernames? Friday's question asked how to recycle the usernames as many times as necessary to use up all the passwords while using each username four times.

@barubary bravely went with nested loops and some tricky input redirection to read the inputs in memory. I went with something a little more disk-based:

```
split -l $(( $(cat names | wc -l)*4 )) passwords
for file in x*; do
sed 'p;p;p' users | paste - $file >passwords.$file
done
```

```
sed -i '/\\t$/d' $(ls passwords.* | tail -1)
```

In the first line, I'm using "split" to break my passwords up into smaller files that are at most 4x the length of our names list. The default file names will be "xaa", "xab", and so on. For each on of those files from "split", I am using Thursday's solution to output a file with the username and password combos. However, the last file will likely have fewer passwords than usernames, so the final "sed" command deletes any line that just has "username<tab>" on the line.

Feel free to add "rm -f x*" to clean up the intermediate password files from "split".

#Linux #DFIR #CommandLine #Trivia

2023-03-13

Suppose you had a directory of files like:

```
foo-10006.img
foo-10009.img
foo-8899.img
foo-9998.img
bar-3235.img
bar-44328.img
bar-4433.img
bar-788.img
bar-6754.img
baz-981.img
baz-76543.img
```

You want to keep, for each base file name, the three files with the highest number in their file names. All other files should be deleted.

FORGOT TO MENTION: I will be on vacation the rest of this week. The answer to this question will be posted on Monday, March 20.

#Linux #DFIR #CommandLine #Trivia

2023-03-20

Last Monday's Linux DFIR command line trivia asked you to clean up a directory of files with names like:

```
foo-10006.img
foo-10009.img
foo-8899.img
foo-9998.img
bar-3235.img
bar-44328.img
bar-4433.img
bar-788.img
bar-6754.img
baz-981.img
baz-76543.img
```

For each base name you want to keep the three files with the highest number in their names. All other files should be removed.

My initial thought was similar to the submission by @jtk:

```
for f in $(ls | cut -f1 -d- | uniq); do
ls $f-*.img | sort -nr -t- -k2 | tail -n +4
done | xargs rm
```

First we figure out all the base names with "ls | cut -f1 -d- | uniq". Then we loop over those and sort them by the number in the file name with "sort -nr -t- -k2". Any file names after the first three get printed by "tail -n +4". We feed the unwanted file names into "xargs rm" at the end of the loop and we're done.

But then I got to wondering if I could do this with a single "ls" command. Hmmm, sort the entire directory by the number in each file name and then maybe use an array to track how many times I'd seen each base name?

```
ls | sort -nr -t- -k2 | awk -F- '++a[$1] > 3' | xargs rm
```

I'm using "awk" to autosplit the file names on "-" ("-F-") and then track an array of the base file names ("a[$1]"). Each time we hit a base name, we increment the counter in the array by one. When we get to the fourth and later instances ("a[$1] > 3") "awk" will output the full file name because the implicit action is always "{print $0}" if not specified. Again we use "xargs rm" to remove the unwanted files.

Using "awk" here because it saves me some typing over using a shell "while read ..." type loop. Also it avoids issues with older versions of bash that don't do text indexed arrays. That hasn't been an issue for some time in bash, but I've been burned so many times by older infrastructure that I'm a little paranoid about it.

#Linux #DFIR #CommandLine #Trivia

2023-03-20

Search a directory for all files where all timestamps are greater than a year old and move those files to a new directory, preserving the structure of the original directory.

I used "move" intentionally here. After the operation, the original directory should only contain files where at least one of the timestamps is less than one year old. The new directory should contain all of the "old files" only.

#Linux #DFIR #CommandLine #Trivia

2023-03-21

Yesterday's Linux DFIR command line trivia asked you to find files with any timestamps older than one year and move them to a new directory while preserving the directory structure.

@deeseearr checked in with an excellent solution and a great explanation. So good, in fact, that I am simply going to refer you to that toot for the answer: https://infosec.exchange/@deeseearr/110056258962644046

My only quibble would be that on systems where it is supported I would also add a "! -newerBt" test to check the file creation date.

#Linux #DFIR #CommandLine #Trivia

2023-03-21

One analyst created a file of checksums using "openssl md5 ..." which produces output like:

```
MD5(/etc/passwd)= 6d7dbb8bb133f5af42992a2e5c76bade
```

Another analyst used "md5sum" which produces output like:

```
6d7dbb8bb133f5af42992a2e5c76bade /etc/passwd
```

Write a shell command to convert the "openssl md5" output into the "md5sum" output format so that the hash lists can be compared more easily.

#Linux #DFIR #CommandLine #Trivia

2023-03-22

Yesterday's Linux DFIR command line trivia asked you to convert a file of lines like

```
MD5(/etc/passwd)= 6d7dbb8bb133f5af42992a2e5c76bade
```

The resulting output should look like:

```
6d7dbb8bb133f5af42992a2e5c76bade /etc/passwd
```

The most terse responses ended up using "sed" that looked something like

```
sed -r 's,MD5\\((.*)\\)= (.*),\\2\\t\\1,' input >output
```

Shout out to @cdp1337 for chiming in with the first working answer.

Match the file name inside "MD5(...)" and the hash after the equals sign as sub-expressions. Then replace the whole line with the hash, a tab, and the file name.

#Linux #DFIR #CommandLine #Trivia

2023-03-22

Output "ps" data for all interactive commands running on the system. This means processes where the "TTY" column is something other than "?".

#Linux #DFIR #CommandLine #Trivia

2023-03-23

Yesterday's Linux DFIR command line trivia asked you to find interactive commands running on the system. I was all geared up to use "awk" to filter on the TTY column:

```
ps -ef | awk '$6 != "?"'
```

And then I got one of the TIL moments that I appreciate about this series when people like @regnil (very kindly) pointed out that I should just use "ps a" instead. So here's the solution:

```
ps a
```

Yeah.

It's interesting to note that the output includes the "agetty" process on the text console(s) and any graphical login console you might have set up on what is usually "tty7". These are not strictly user interactive processes. But you don't want to ignore the stuff happening on the "tty*" entries just in case a user is doing something over a physical login device.

#Linux #DFIR #CommandLine #Trivia

2023-03-23

You are receiving uploads across a number of directories at the same time. You can tell if an upload is not finished because there will be at least one file in that directory with a ".inprogress" extension on it. As each upload finishes, you want to move the directory containing the upload to a new location. Write a command line to recognize and move the completed upload directories.

#Linux #DFIR #CommandLine #Trivia

2023-03-24

Yesterday's Linux DFIR command line trivia asked you to move directories that did NOT contain a file with a ".inprogress" extension.

We can find the directories with ".inprogress" extensions easily enough:

```
find * -type f -name \\*.inprogress | cut -f1 -d/
```

I might get repeats of a directory name if it has multiple ".inprogress" files inside of it, but that turns out to be not important because here's how I'm going to get the directories that DO NOT have ".inprogress" files:

```
cat <(ls) <(find * -type f -name \\*.inprogress | cut -f1 -d/) | sort | uniq -u
```

Sort the list of all directories together with the list of directories containing ".inprogress" files and then only output the unique directory names. This approach works even with craziness like directory names with spaces.

To actually move the directories, we'll tack on a bit of help from "xargs":

```
cat <(ls) <(find * -type f -name \\*.inprogress | cut -f1 -d/) | sort | uniq -u | xargs -d \\\\n mv -d /some/
dest/dir
```

I'm telling "xargs" to use newlines as the delimiter ("-d \\\\n") again to protect us from directory names with spaces in them.

@xabean correctly points out that solutions like this have race-condition issues and suggests some possible solutions leveraging hard links. Ultimately, solutions that avoid race conditions are going to veer sharply towards a full-blown shell script. This is fine, but I'm here for the command line stunts mostly.

#Linux #DFIR #CommandLine #Trivia

2023-03-24

You want to convert a split-raw image from files named "fileaa", "fileab", "fileac", ... and so on into files named "file.001", "file.002", "file.003", etc.

#Linux #DFIR #CommandLine #Trivia

2023-03-27

Friday's Linux DFIR command line trivia asked you to rename a directory of files from "fileaa", "fileab", "fileac", ... and so on into files named "file.001", "file.002", "file.003", etc.

Typically I've done this with a counting loop, similar to the solution provided by @barubary:

```
i=0; for f in file*; do let ++i; mv "$f" "$(printf 'file.%03d' $i)"; done
```

But then @lamitpObuS checked in with an interesting solution that uses "nl" to get the file numbers:

```
ls file* | nl -nrz | while read num file; do mv $file file.${num: -3}; done
```

The only downside here is that "nl" expands the numbering to six places (there needs to be an option implemented to control the feed width). So we end up needing to trim off the leading three zeroes in each file extension.

An alternative approach would use "sed" to strip the zeroes before the loop:

```
ls file* | nl -nrz | sed 's/000//' | while read num file; do mv $file file.$num; done
```

This is perhaps marginally faster, but it the real cost of the loop is renaming the files, not computing the extensions.

#Linux #DFIR #CommandLine #Trivia

2023-03-27

On Friday I asked how to rename a directory of files from "fileaa", "fileab", "fileac", ... and so on into files named "file.001", "file.002", "file.003", etc. Now do it in the other direction. Rename the "file.001", "file.002", "file.003", etc files into "fileaa", "fileab", "fileac", ... and so on. Why is this useful? I have no idea. I just think it will be a fun challenge for your Linux scripting skills.

Also, I'm gearing up for a trip to Australia Mar 31 - Apr 16, and the Linux DFIR command line series is going to shift to something less frequent than daily during this time. Just keep an eye out for the usual hash tags.

If you're in Australia and want to meet up while I'm in Sydney, Canberra, or Melbourne, give me a shout and we'll try to set something up. If you'd like some great Linux Forensics training taught by a crazy American, there are a few seats still available: https://cdfs.com.au/training/?swoof=1&product_tag=linux-forensics

#Linux #DFIR #CommandLine #Trivia #Training

2023-03-29

The task is to rename "file.001", "file.002", "file.003", etc into "fileaa", "fileab", "fileac", ... and so on. This is basically the opposite of the previous Linux DFIR command line trivia question.

But it's actually harder to go in this direction because there isn't a natural way to handle the lettering except for a couple of nested loops:

```
i=0
for l1 in {a..z}; do
for l2 in {a..z}; do
printf -v ext %03d $(( i++ ))
[[ -f file.$ext ]] && mv file.$ext file$l1$l2 || break 2
```

```
done
done
```

Using "$I" as a counting variable, we simply march through all of the "file.xxx" files until we run out. When that happens, the "break 2" stops both loops and we are done.

Eh, why did I think it was a good idea to do this anyway?

#Linux #DFIR #CommandLine #Trivia

2023-03-29

Write a Linux command line to check a web site an see if its SSL certificate has expired.

#Linux #DFIR #CommandLine #Trivia