Majd Hamdan — CSCI 1005

Linux Kernel Scavenger Hunt

January 11, 2021

1 Linux kernel scavenger hunt

- Download a tarball of the source code for the Linux kernel from kernel.org. The pacman program will come in handy to determine the version of the kernel you're running and the the wget or curl programs may come in handy for downloading from the command-line (you can also download it using a browser).

To check the version of linux, we use:

```
Sudo pacman -Q linux

```

If we use the -V option, we would get the pacman version rather than the linux, as -V traditionally is used to get the version of the program.

We download the tarball file by copying the link first, then using:

```
curl "link" > tarball

```

The "> tarball" part is saving the output to a file called tarball. The file is compressed, to de-compress it, we use:

```
tar -xJf "tarball"

```

- Among many other things, the Linux kernel is responsible for producing data when a given file is asked to be read. You're used to this involving disk access, but this isn't strictly necessary: for instance, all "files" under /proc are dynamically generated by the kernel when you request them. The file "/proc/self/maps" doesn't exist on disk, but if you run cat /proc/self/maps, you get some meaningful information.

  Your task is to find the precise lines of code within the Linux kernel that produces that information.

First, we should run "cat /proc/self/maps" to get a sense of the output we are expecting. The intuition is to first find anything related to /proc. We can do this by finding the directory proc in the source code. We use the following command:

```
find . -type d -name "proc"

```

find searches for files in a directory. The "." tells find to search in the current directory. "-type d" tells find to only look for directories. So the whole command reads as: search in the current directory for directories named proc. The output is:

```
./linux-5.15.14/fs/proc
./linux-5.15.14/tools/testing/selftests/proc

```

When we browse in fs, we find proc with multiple c files. Now we have to look at those files to find the lines responsible of generating the maps file. To look into files, we use greb:

```
grep -r "maps" . | less

```

Grep looks for the pattern "maps" inside all the files (the "-r" option allow to look recursively for all the files) inside "." directory which is the current directory.

We get many outputs, the first of which is:

```
./nommu.c:      proc_create_seq("maps", S_IRUGO, NULL, &
   proc_nommu_region_list_seqop);

```

This indicate that inside the file nommu.c, there is a line creating the sequence "maps" using the address "proc_nommu_region_list_seqop". This address could be the memory address of a file or a function. We inspect nommu.c using atom and look for "proc_nommu_region_list_seqop". We find that it is a struct:

```
static const struct seq_operations proc_nommu_region_list_seqop = {
  .start  = nommu_region_list_start,
  .next = nommu_region_list_next,
  .stop = nommu_region_list_stop,
  .show = nommu_region_list_show
};

```

The "nommu_region_list_*" could be, again, functions or variables. When we look for them and after minimal inspection, we find that "nommu_region_list_show" is a function that calls another function "nommu_region_show" that produces an output similar to the output we are looking for in this line:

```
seq_setwidth(m, 25 + sizeof(void *) * 6 - 1);
seq_printf(m,

       "%08lx-%08lx %c%c%c%c %08llx %02x:%02x %lu ",
       region->vm_start,
       region->vm_end,
       flags & VM_READ ? 'r' : '-',
       flags & VM_WRITE ? 'w' : '-',
       flags & VM_EXEC ? 'x' : '-',
       flags & VM_MAYSHARE ? flags & VM_SHARED ? 'S' : 's' : 'p',
       ((loff_t)region->vm_pgoff) << PAGE_SHIFT,
       MAJOR(dev), MINOR(dev), ino);

```

When reading the comments in nommu.c, we find that "nommu_region_list_show" display a list of all the REGIONs the kernel knows about- nommu kernels have a single flat list and "nommu_region_show" display a single region to a sequenced file.

2 Scoping the problem

- I mentioned yesterday that the Linux kernel has a bazillion (approx) lines of code, in which potential vulnerabilities may be lurking. Count the lines of code in the Linux kernel.

  This will involve determining what qualifies as "code" (the find and file commands might come in handy) and then doing the counting.

Here, we will consider source files as "*.c" files. We can find all these files using (make sure you cd into the directory of linux source code):

```
find . -type f -name "*.c"

```

which will find all files of type f (file) with names that ends with ".c". Now, we have to go through all these files and extract the code lines excluding the comment lines. We can do this using grep. First, to make sure the following command produces the correct output, I created a test file called lab1 from the code in nommu.c. The content of the file is:

```
// SPDX-License-Identifier: GPL-2.0-or-later
/* nommu.c: mmu-less memory info files
 *
 * Copyright (C) 2004 Red Hat, Inc. All Rights Reserved.
 * Written by David Howells (dhowells@redhat.com)
 */

static int nommu_region_show(struct seq_file *m, struct vm_region *region)
{
  unsigned long ino = 0;
  struct file *file;
  dev_t dev = 0;
  int flags;

  flags = region->vm_flags;
  file = region->vm_file;

  if (file) {
    struct inode *inode = file_inode(region->vm_file);
    dev = inode->i_sb->s_dev;
    ino = inode->i_ino;
  }

  seq_setwidth(m, 25 + sizeof(void *) * 6 - 1);
  seq_printf(m,
       "%08lx-%08lx %c%c%c%c %08llx %02x:%02x %lu ",
       region->vm_start,
       region->vm_end,
       flags & VM_READ ? 'r' : '-',
       flags & VM_WRITE ? 'w' : '-',
      flags & VM_EXEC ? 'x' : '-',
      flags & VM_MAYSHARE ? flags & VM_SHARED ? 'S' : 's' : 'p',
      ((loff_t)region->vm_pgoff) << PAGE_SHIFT,
      MAJOR(dev), MINOR(dev), ino);

  if (file) {
    seq_pad(m, ' ');
    seq_file_path(m, file, "");
  }


```

To grep the code lines, I ran:

```
grep "^[^/*] lab1.c"

```

The first ∧ specify the beginning of the line. "[/*]" means any character which is not / or *. This will read as grab all the lines from lab1.c that don't start with / or *. The output is:

```
 *
 * Copyright (C) 2004 Red Hat, Inc. All Rights Reserved.
 * Written by David Howells (dhowells@redhat.com)
 */
static int nommu_region_show(struct seq_file *m, struct vm_region *region)
{
  unsigned long ino = 0;
  struct file *file;
  dev_t dev = 0;
  int flags;
  flags = region->vm_flags;
  file = region->vm_file;
  if (file) {
    struct inode *inode = file_inode(region->vm_file);
    dev = inode->i_sb->s_dev;
    ino = inode->i_ino;
  }
  seq_setwidth(m, 25 + sizeof(void *) * 6 - 1);
  seq_printf(m,
      "%08lx-%08lx %c%c%c%c %08llx %02x:%02x %lu ",
      region->vm_start,
      region->vm_end,
      flags & VM_READ ? 'r' : '-',
      flags & VM_WRITE ? 'w' : '-',
      flags & VM_EXEC ? 'x' : '-',
      flags & VM_MAYSHARE ? flags & VM_SHARED ? 'S' : 's' : 'p',
      ((loff_t)region->vm_pgoff) << PAGE_SHIFT,
      MAJOR(dev), MINOR(dev), ino);
  if (file) {
    seq_pad(m, ' ');
    seq_file_path(m, file, "");
  }
```

```


```

As you might notice, the first 4 lines are comments because they actually started with a space. So, we should introduce the space to the excluded characters:

```
grep "^[^/* ] lab1.c"

```

which outputs the correct result:

```
static int nommu_region_show(struct seq_file *m, struct vm_region *region)
{
  unsigned long ino = 0;
  struct file *file;
  dev_t dev = 0;
  int flags;
  flags = region->vm_flags;
  file = region->vm_file;
  if (file) {
    struct inode *inode = file_inode(region->vm_file);
    dev = inode->i_sb->s_dev;
    ino = inode->i_ino;
  }
  seq_setwidth(m, 25 + sizeof(void *) * 6 - 1);
  seq_printf(m,
       "%08lx-%08lx %c%c%c%c %08llx %02x:%02x %lu ",
       region->vm_start,
       region->vm_end,
       flags & VM_READ ? 'r' : '-',
       flags & VM_WRITE ? 'w' : '-',
       flags & VM_EXEC ? 'x' : '-',
       flags & VM_MAYSHARE ? flags & VM_SHARED ? 'S' : 's' : 'p',
       ((loff_t)region->vm_pgoff) << PAGE_SHIFT,
       MAJOR(dev), MINOR(dev), ino);
  if (file) {
    seq_pad(m, ' ');
    seq_file_path(m, file, "");
  }


```

Now, some of these lines might look like they started with a space, but it is actually a tap, so it passes. Next, we should combine these results using a pipe. Intutivly, we would write:

```
find . -type f -name "*.c" | grep "^[^/* ]" | wc


```

but this won't produce the correct results as grep would be implemented on the list of files returned by find and not on the content of these files. To solve this, we use xargs giving this output (takes long time to run):

```
 find . -type f -name "*.c" | xargs grep "^[^/* ]" | wc
16710832 69383055 1344509370

```

3 Customize your shell prompt

- I mentioned yesterday that the Linux kernel has a bazillion (approx) lines of code, in which potential vulnerabilities may be lurking. Count the lines of code in the Linux kernel.

  This will involve determining what qualifies as ”code” (the find and file commands might come in handy) and then doing the counting.

  Customize your shell prompt The contents of the bash environment variable PS1 controls the appearance of your shell prompt. Figure out the format of the magical incantation that speaks to PS1.

  You can test new values for PS1 by setting its value in the current shell: export PS1=”foo bar baz” (In very simple terms, export is the command to set an environment variable—it’s actually more complicated than this, but you don’t need to worry about those details right now).

  You’ll note that, if you log out and log back in again, your prompt reverts to its default format. Make your change persistent by putting your export command in the file /.bashrc.

  If you’re super-ambitious, try setting colors: http://misc.flogisoft.com/bash/tip_colors_and_formatting

  If you’re super-super-ambitious, read the bash manpage to understand all the settings customization you can perform, as well as how/when/why /.bashrc is evaluated as opposed to /.bash_profile.