Linux bcc Tracing Security Capabilities

01 Oct 2016

Which Linux security capabilities are your applications using? I recently developed a new tool, capable, to print out capability checks live:

```
# capable
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
22:11:23  114    2676   snmpd            12   CAP_NET_ADMIN        1
22:11:23  0      6990   run              24   CAP_SYS_RESOURCE     1
22:11:23  0      7003   chmod            3    CAP_FOWNER           1
22:11:23  0      7003   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        7    CAP_SETUID           1
22:11:24  0      7013   run              24   CAP_SYS_RESOURCE     1
22:11:24  0      7026   chmod            3    CAP_FOWNER           1
[...]
```

capable uses bcc, a front-end and a collection of tools that use new Linux enhanced BPF tracing capabilities. capable works by using BPF with kprobes to dynamically trace the kernel cap_capable() function, and then uses a table to map the capability index to the name seen in the output. Here's the source code: it's pretty straightforward.

I wrote it as a colleague, Michael Wardrop, asked what security capabilities our applications were actually using. Given a list, we could use setcap(8) (or other software) to improve the security of applications by only allowing the necessary capabilities.

Non-audit Checks

The cap_capable() function has an audit argument, which directs whether the capability check should write an audit message or not, if that's configured. By default, I only print capability checks where this is true, but capable can also trace all checks with the -v option:

```
# capable -h
usage: capable [-h] [-v] [-p PID]

Trace security capability checks

optional arguments:
  -h, --help          show this help message and exit
  -v, --verbose       include non-audit checks
  -p PID, --pid PID   trace this PID only

examples:
    ./capable             # trace capability checks
    ./capable -v          # verbose: include non-audit checks
    ./capable -p 181      # only trace PID 181
```

Here's some non-audit events:

```
# capable -v
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
20:53:45  60004  22061  lsb_release      21   CAP_SYS_ADMIN        0
[...]
```

What are all those?

I'll start by showing the cap_capable() function prototype, from security/commoncap.c:

```
int cap_capable(const struct cred *cred, struct user_namespace *targ_ns,
                int cap, int audit)
```

Now I can use bcc's trace program to inspect these calls (bear with me), given that cap will be arg3, and audit arg4 (from the above prototype):

```
# trace 'cap_capable "cap: %d, audit: %d", arg3, arg4'
TIME     PID    COMM         FUNC             -
20:56:18 25535  lsb_release  cap_capable      cap: 21, audit: 0
20:56:18 25535  lsb_release  cap_capable      cap: 21, audit: 0
20:56:18 25535  lsb_release  cap_capable      cap: 21, audit: 0
20:56:18 25535  lsb_release  cap_capable      cap: 21, audit: 0
20:56:18 25535  lsb_release  cap_capable      cap: 21, audit: 0
[...]
```

That one-liner is pretty similar to my capable program, except it lacks the "NAME" column with human readable translations.

I'm really doing this so I can add the (newly added) -K and -U options, which print kernel and user-level stack traces. I'll just use -K:

```
# trace -K 'cap_capable "cap: %d, audit: %d", arg3, arg4'
TIME     PID    COMM         FUNC             -
[...]
20:59:58 30607  lsb_release  cap_capable      cap: 21, audit: 0
    Kernel Stack Trace:
        ffffffff813659d1 cap_capable
        ffffffff813684bb security_vm_enough_memory_mm
        ffffffff811deda6 expand_downwards
        ffffffff811def64 expand_stack
        ffffffff81234321 setup_arg_pages
        ffffffff8128c10b load_elf_binary
        ffffffff81234cee search_binary_handler
        ffffffff8128b7ff load_script
        ffffffff81234cee search_binary_handler
        ffffffff8123635a do_execveat_common.isra.35
        ffffffff812367da sys_execve
        ffffffff81003bae do_syscall_64
        ffffffff81861ca5 return_from_SYSCALL_64

20:59:58 30607  lsb_release  cap_capable      cap: 21, audit: 0
    Kernel Stack Trace:
        ffffffff813659d1 cap_capable
        ffffffff813684bb security_vm_enough_memory_mm
        ffffffff811df623 mmap_region
        ffffffff811dff4b do_mmap
        ffffffff811c122a vm_mmap_pgoff
        ffffffff811c1295 vm_mmap
        ffffffff8128bb93 elf_map
        ffffffff8128c271 load_elf_binary
        ffffffff81234cee search_binary_handler
        ffffffff8128b7ff load_script
        ffffffff81234cee search_binary_handler
        ffffffff8123635a do_execveat_common.isra.35
        ffffffff812367da sys_execve
        ffffffff81003bae do_syscall_64
        ffffffff81861ca5 return_from_SYSCALL_64
[...]
```

Awesome. So these are coming from security_vm_enough_memory_mm(). By reading the source, I see it's used to reserve some memory for root. It's not a hard failure if the capability is missing. And it's not really a security check, hence why it disabled audit.

I should add a -K option to the capable tool, so it can print stack traces too.

Older Kernels

To use capable, you'll need a 4.4 kernel. To use the -K option, 4.6.

Here's a version using my older perf-tools collection, which uses ftrace and should work on much older kernels including the 3.x series:

```
# ./perf-tools/bin/kprobe -s 'p:cap_capable cap=%dx audit=%cx' 'audit != 0'
Tracing kprobe cap_capable. Ctrl-C to end.
             run-4440  [003] d... 6417394.367486: cap_capable: (cap_capable+0x0/0x70) cap=0x18 au
             run-4440  [003] d... 6417394.367492:
 => ns_capable_common
 => capable
 => do_prlimit
 => SyS_setrlimit
 => entry_SYSCALL_64_fastpath
           chmod-4453  [006] d... 6417394.399020: cap_capable: (cap_capable+0x0/0x70) cap=0x3 aud
           chmod-4453  [006] d... 6417394.399027:
 => ns_capable_common
 => ns_capable
 => inode_owner_or_capable
 => inode_change_ok
 => xfs_setattr_nonsize
 => xfs_vn_setattr
 => notify_change
 => chmod_common
 => SyS_fchmodat
 => entry_SYSCALL_64_fastpath
           chmod-4453  [006] d... 6417394.399035: cap_capable: (cap_capable+0x0/0x70) cap=0x4 aud
           chmod-4453  [006] d... 6417394.399037:
 => ns_capable_common
 => capable_wrt_inode_uidgid
 => inode_change_ok
 => xfs_setattr_nonsize
 => xfs_vn_setattr
 => notify_change
 => chmod_common
 => SyS_fchmodat
 => entry_SYSCALL_64_fastpath
           chmod-4455  [007] d... 6417394.402524: cap_capable: (cap_capable+0x0/0x70) cap=0x4 aud
           chmod-4455  [007] d... 6417394.402529:
 => ns_capable_common
 => capable_wrt_inode_uidgid
 => inode_change_ok
 => xfs_setattr_nonsize
 => xfs_vn_setattr
 => notify_change
 => chmod_common
 => SyS_fchmodat
 => entry_SYSCALL_64_fastpath
[...]
```

It's a one-liner using my kprobe tool. It's also (currently) a bit harder to use: I need to know which registers those arguments will be in: the example above is for x86_64 only.

That's all for now. Happy hacking.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).

About this blog