Linux Fundamentals

Insu Yun

Today’s lecture

- Linux
- File system
    - Permission
    - File-related system calls
    - File descriptors
- Process and thread
- Shell

What is Linux?

- Unix-like operating system
- Developed by Linus Torvalds
- Many distributions exist
  - Centos
  - Redhat
  - Ubuntu 20.04 <- Our server
  - …

An operating system is software that provides

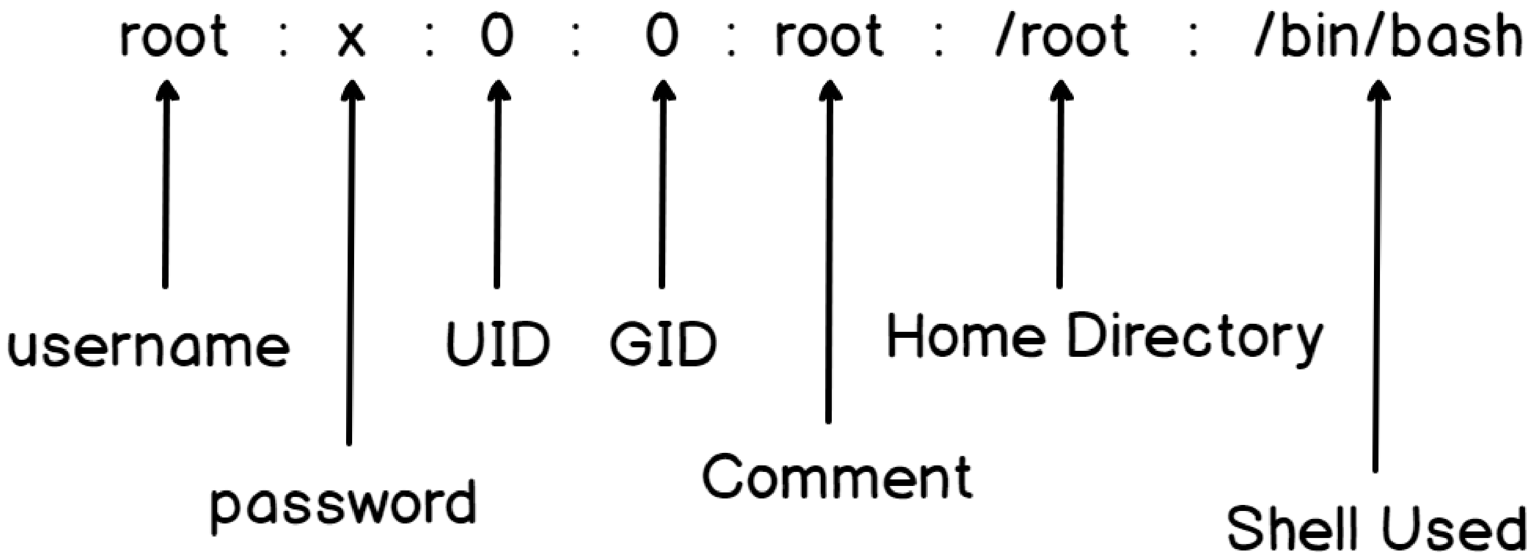
- Resource management
- Security
- Hardware abstraction
- User interface
- …

Users

- Users are identified by a User id (a number)
- User ID ‘0’ is “root” – the administrator
- Objects in the system (Processes, Files) are attached to Users
- Everything else stems from that
- All Users are defined in the file “/etc/passwd”

http://haifux.org/lectures/84-sil/users-processes-files-and-permissions/users-perms-lec.html

/etc/passwd



Historically, this file contained password, but now moved to /etc/shadow

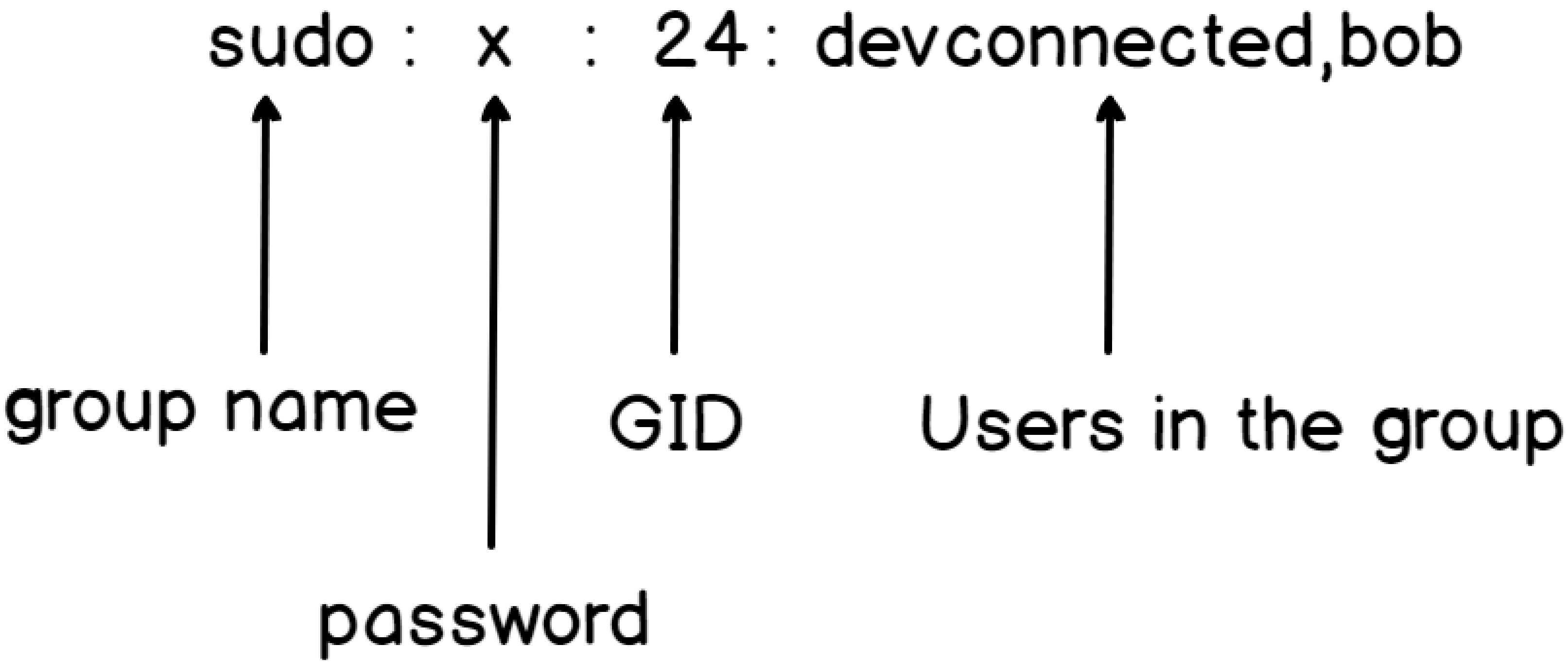
https://devconnected.com/how-to-list-users-and-groups-on-linux/

Groups

- Groups are identified by a Group id - also a number.
- A Group may contain 0 or more Users.
- Objects in the system (Processes, Files) are attached to Groups.
- All Groups are defined in the file "/etc/group".
  - Except for ‘primary Groups', which might be implicitly defined in "/etc/passwd".

http://haifux.org/lectures/84-sil/users-processes-files-and-permissions/users-perms-lec.html

/etc/group



```
sudo :  x   :  24: devconnected,bob
```

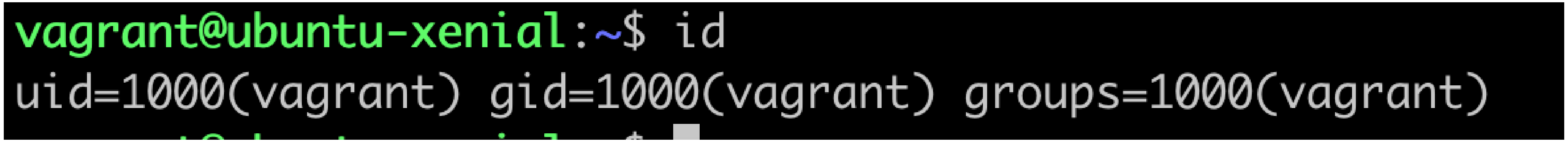
https://devconnected.com/how-to-list-users-and-groups-on-linux/

Summary: A user has

- 1 uid

- 1 gid (for primary group)

- Multiple secondary groups (in /etc/group)



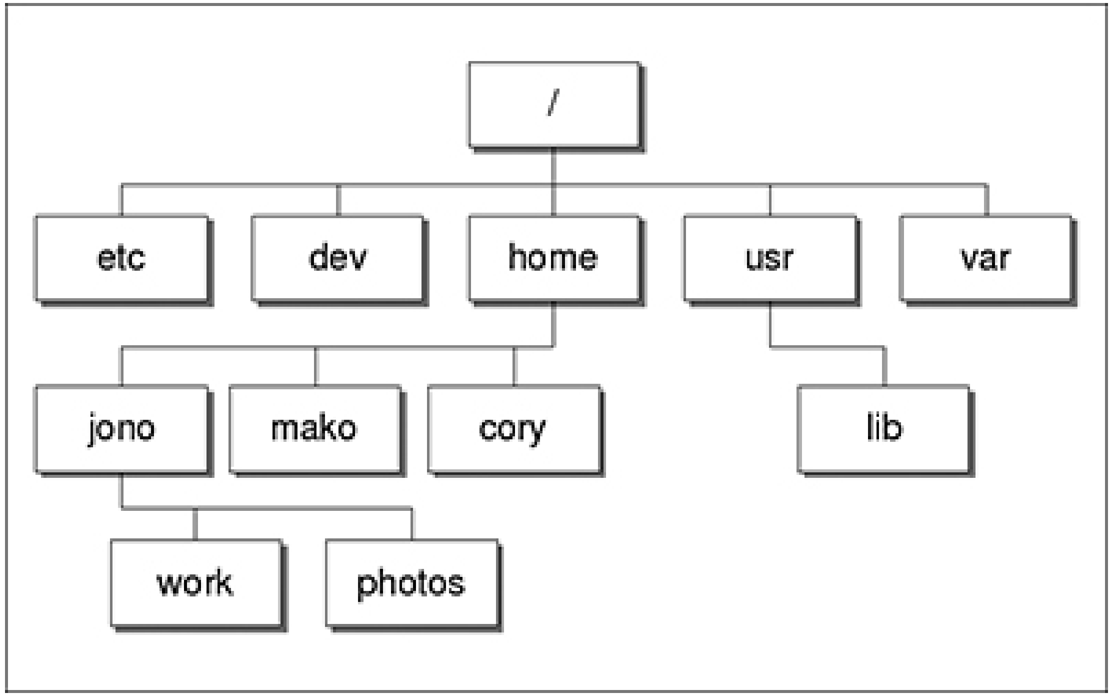
```
vagrant@ubuntu-xenial:~$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
vagrant@ubuntu-xenial:~$
```

Processes have uids & gids for permission

- Real UID (uid): the user who launched the process
- Real GID (gid): the primary group of the user that launched the process
- Effective uid (euid) & Effective gid(egid): determine what resources the process can access
    - See later with setuid/setgid

Linux file system

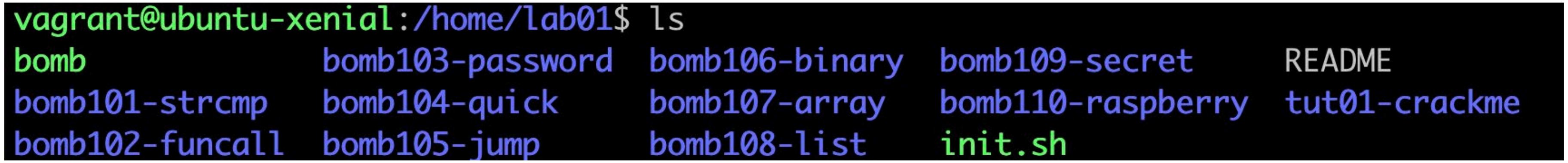
- A tree-based model that stores files and directories



The Official Ubuntu Book, 7th Edition: Becoming an Ubuntu Power User

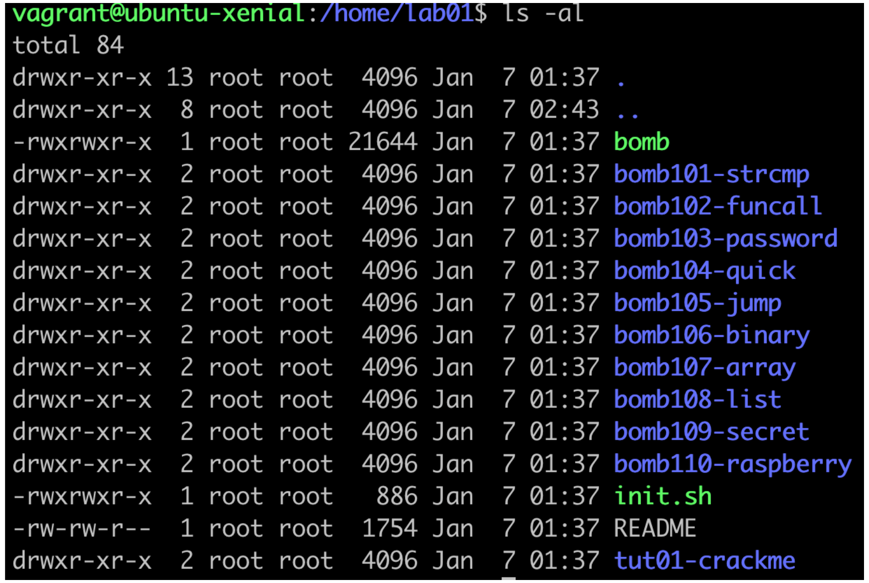
Linux file system

- Can check a list of files in the current directory using **`ls`** command



```
vagrant@ubuntu-xenial:/home/lab01$ ls
bomb              bomb103-password  bomb106-binary  bomb109-secret     README
bomb101-strcmp    bomb104-quick     bomb107-array   bomb110-raspberry  tut01-crackme
bomb102-funcall   bomb105-jump      bomb108-list    init.sh
```

- You can get more information by typing **`ls -al`**

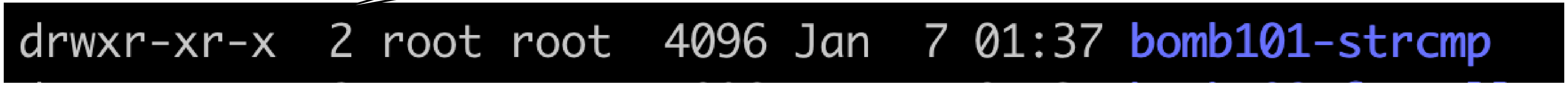


```
vagrant@ubuntu-xenial:/home/lab01$ ls -al
total 84
drwxr-xr-x 13 root root  4096 Jan  7 01:37 .
drwxr-xr-x  8 root root  4096 Jan  7 02:43 ..
-rwxrwxr-x  1 root root 21644 Jan  7 01:37 bomb
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb101-strcmp
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb102-funcall
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb103-password
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb104-quick
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb105-jump
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb106-binary
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb107-array
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb108-list
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb109-secret
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb110-raspberry
-rwxrwxr-x  1 root root   886 Jan  7 01:37 init.sh
-rw-rw-r--  1 root root  1754 Jan  7 01:37 README
drwxr-xr-x  2 root root  4096 Jan  7 01:37 tut01-crackme
```

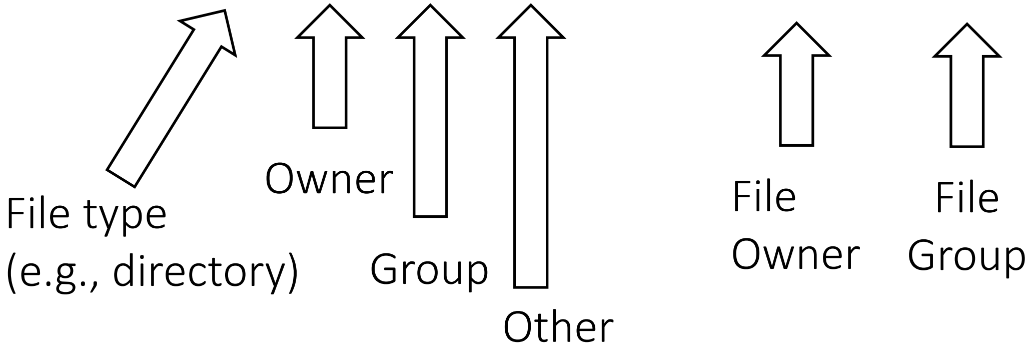
- "." is a current directory
- ".."is a parent directory

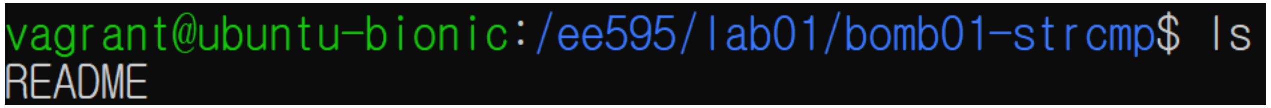
Linux file permission

Q: What does this number mean?



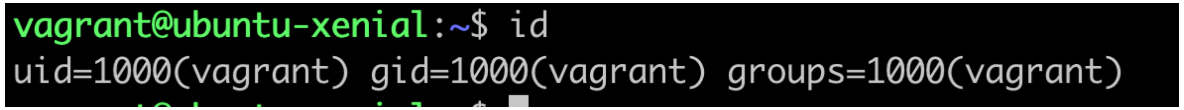
```
drwxr-xr-x  2 root root  4096 Jan  7 01:37 bomb101-strcmp
```





```
vagrant@ubuntu-bionic:/ee595/lab01/bomb01-strcmp$ ls
README
```

- `r`: read, `w`: write, `x`: executable
- Permissions are often expressed with the octal number (i.e., base 8)
  - `r` = 4, `w` = 2, `x` = 1
  - e.g., `rwxr-xr-x`: 755
  - e.g., `rwxrwxrwx`: 777



```
vagrant@ubuntu-xenial:~$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Permission for a file & a directory

- rwx for a file
  - r: Can read the file
  - w: Can write the file
  - x: Can execute the file

- rwx for a directory
  - r: Can list the files in the directory
  - w: Can write (e.g., create, rename, delete, modify) files in the directory
  - x: Can access files in the directory

chown & chmod

- chown: change file owner and group
  - Usage: chown [OPTION]... [OWNER][:[GROUP]] FILE
  - Examples:

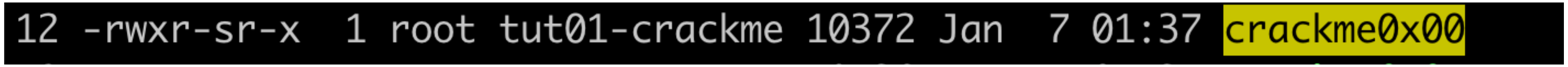
```
$ chown root myfile
        Change the owner of myfile to "root".
$ chown root:staff file
        Likewise, but also change its group to “staff”
```

- chmod: change file mode bits
  - Usage: chmod MODE FILE
  - Examples:

```
$ chmod 754 myfile
        Change the myfile’s permission to 754
```

https://linux.die.net/man/1/chown, https://linux.die.net/man/1/chmod

Special permission: setuid, setgid



```
12 -rwxr-sr-x  1 root tut01-crackme 10372 Jan  7 01:37 crackme0x00
```

Q: Why we use setgid? not setuid?

- `rwxr-`**`s`**`r-x`: setgid program
  - e.g., `rw`**`s`**`r-xr-x`: setuid program

- setgid program changes ‘effective’ gid of its user with its gid
- Similar to `rwx`, special permissions have the octal number form
  - setuid: 4, setgid: 2, sticky bit: 1
  - The above permission would be 2755

How permission checking works

1. Check if my (i.e., process) euid == file’s uid (i.e., owner), then use owner’s permission

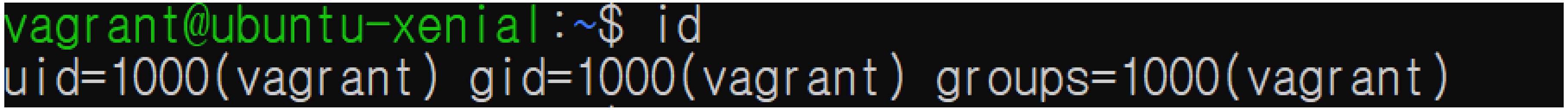
2. Check if my egid is belonging to file’s group, then use group’s permission

3. Otherwise, use other’s permission

NOTE: eu(g)id == ru(g)id except for setu(g)id programs

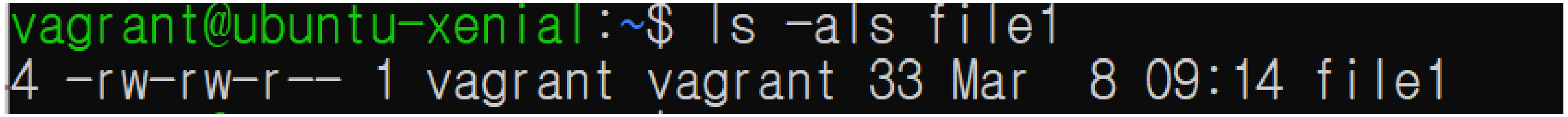
Questions about permissions

- uid (user id): An identifier that specifies a current user
- gid (group id): An identifier that specifies a current group



```
vagrant@ubuntu-xenial:~$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Q: Can I read this?

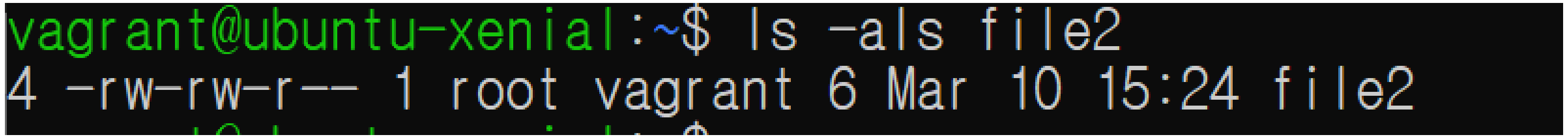


```
vagrant@ubuntu-xenial:~$ ls -als file1
4 -rw-rw-r-- 1 vagrant vagrant 33 Mar  8 09:14 file1
```

O

Questions about permissions

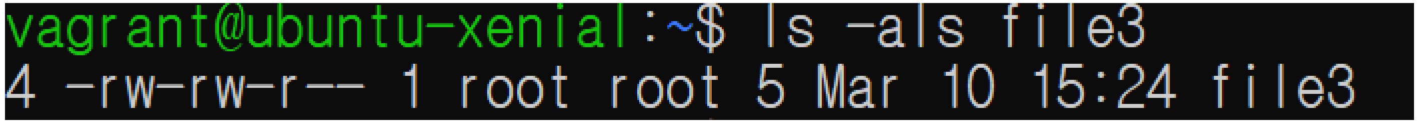
Q: Can I read this?



```
vagrant@ubuntu-xenial:~$ ls -als file2
4 -rw-rw-r-- 1 root vagrant 6 Mar  10 15:24 file2
```

O

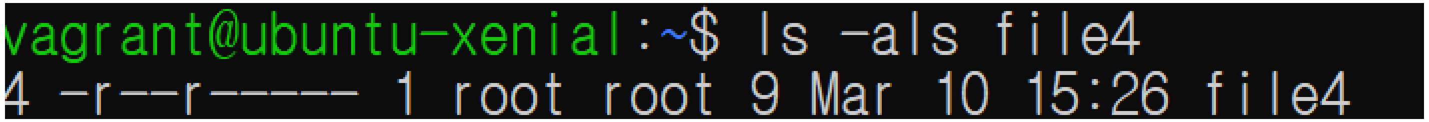
Q: Can I read this?



```
vagrant@ubuntu-xenial:~$ ls -als file3
4 -rw-rw-r-- 1 root root 5 Mar  10 15:24 file3
```

O

Q: Can I read this?

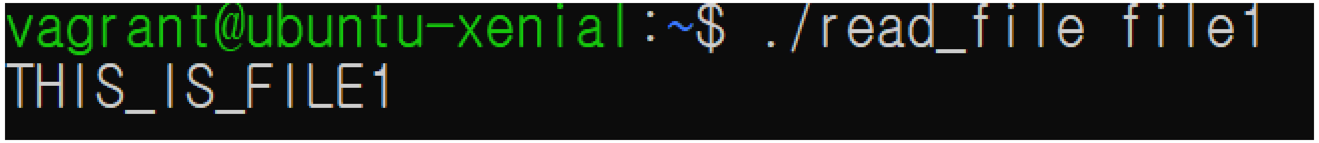


```
vagrant@ubuntu-xenial:~$ ls -als file4
4 -r--r----- 1 root root 9 Mar  10 15:26 file4
```

X

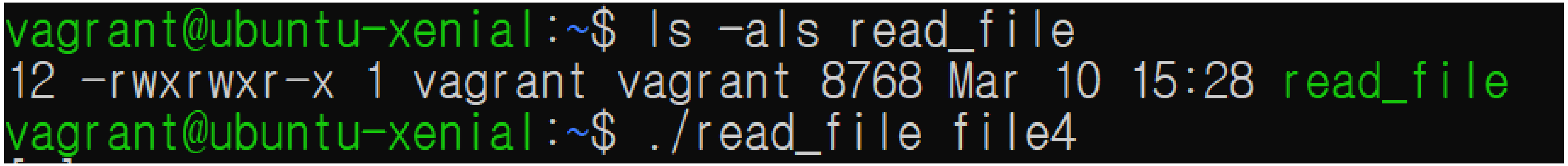
Questions about permissions

- Let’s assume we have a program that reads a file



```
vagrant@ubuntu-xenial:~$ ./read_file file1
THIS_IS_FILE1
```

- Q: Can I read this?

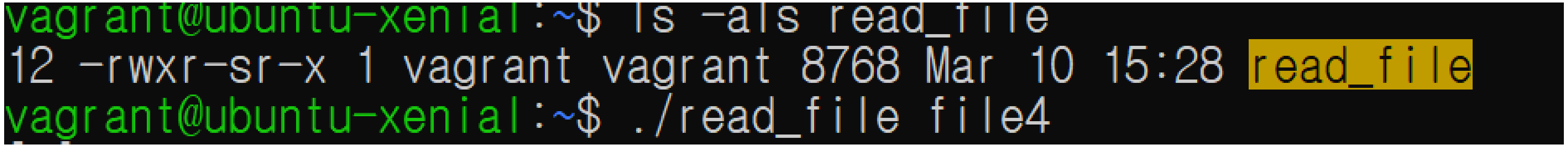


```
vagrant@ubuntu-xenial:~$ ls -als read_file
12 -rwxrwxr-x 1 vagrant vagrant 8768 Mar  10 15:28 read_file
vagrant@ubuntu-xenial:~$ ./read_file file4
```

X

Questions about permissions

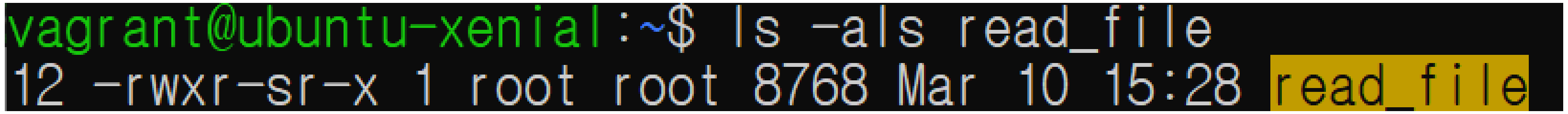
- Can I read this?



```
vagrant@ubuntu-xenial:~$ ls -als read_file
12 -rwxr-sr-x 1 vagrant vagrant 8768 Mar  10 15:28 read_file
vagrant@ubuntu-xenial:~$ ./read_file file4
```

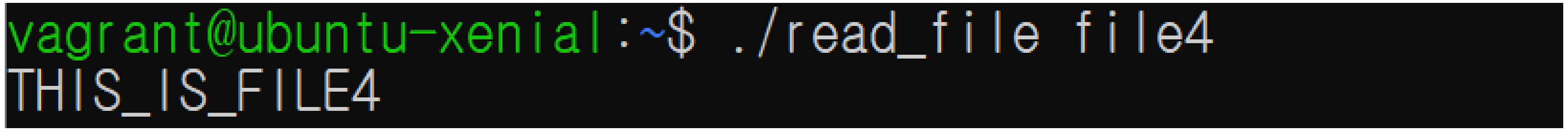
X

- Can I read this?



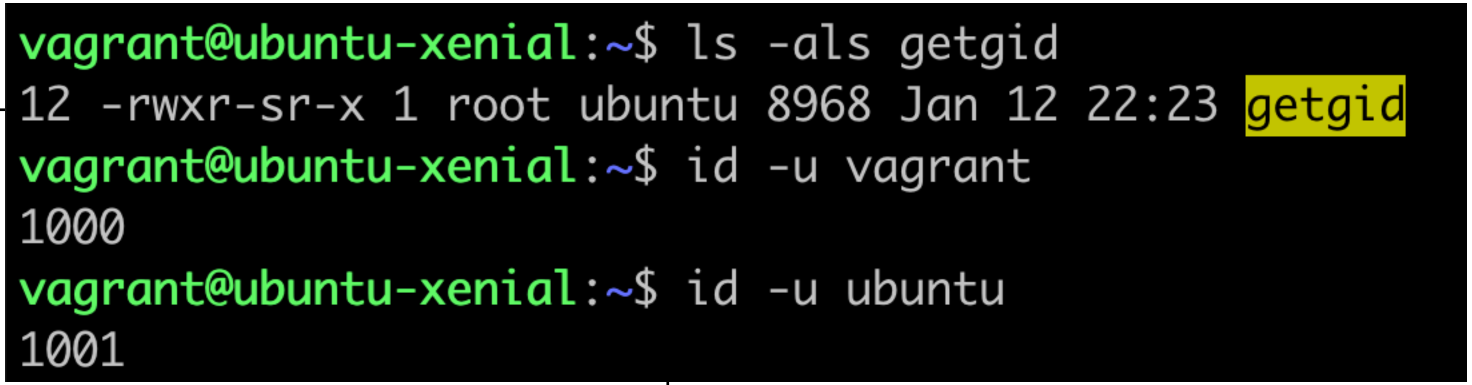
```
vagrant@ubuntu-xenial:~$ ls -als read_file
12 -rwxr-sr-x 1 root root 8768 Mar  10 15:28 read_file
```

- Now I can read it!



```
vagrant@ubuntu-xenial:~$ ./read_file file4
THIS_IS_FILE4
```

More on setgid

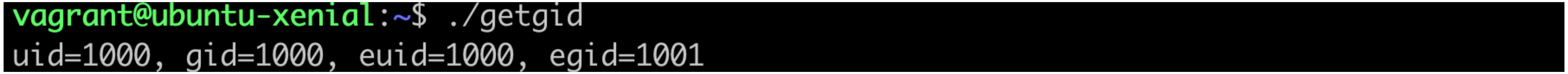


```
vagrant@ubuntu-xenial:~$ ls -als getgid
12 -rwxr-sr-x 1 root ubuntu 8968 Jan 12 22:23 getgid
vagrant@ubuntu-xenial:~$ id -u vagrant
1000
vagrant@ubuntu-xenial:~$ id -u ubuntu
1001
```

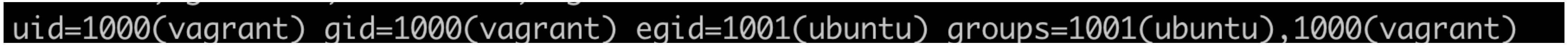
```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  // get permissions directly
  printf("uid=%d, gid=%d, euid=%d, egid=%d\n",
    getuid(), getgid(), geteuid(), getegid());

  // run 'id' using execve system call
  if (!fork())
    execl("/usr/bin/id", "/usr/bin/id", NULL);

  // run 'id' through shell
  system(”/usr/bin/id");
}
```

```
system() = fork()
             + /bin/sh –c “COMMAND”
```

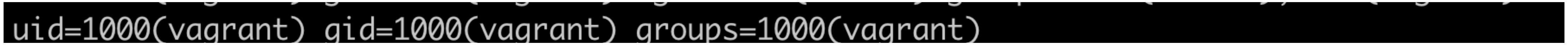
When we run setgid program…



```
vagrant@ubuntu-xenial:~$ ./getgid
uid=1000, gid=1000, euid=1000, egid=1001
```



```
uid=1000(vagrant) gid=1000(vagrant) egid=1001(ubuntu) groups=1001(ubuntu),1000(vagrant)
```



```
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

- Due to security reasons, shell (e.g., sh or bash) drops effective uid/gid

- In our challenges, you will see `setregid(getegid(), getegid());`
  - It allows you to invoke shell with higher privilege
  - As a result, it will make you easy to exploit
    (otherwise, you have to call those functions by yourself)

A special file type: symbolic (soft) link

- A special file that points another file
  - e.g., .lnk file in Windows
- You can create it using `ln` command
  - e.g., `ln –s [src] [dst]`

Q: Without –s, you can create hard link. What’s difference compared to soft link or to copy of a file?

- Interesting property regarding security: You can create symbolic link even you don’t have enough permission for source
  - e.g., You can make symbolic link for a file even you cannot read the file, or the file has setuid permission

Use a file system using open(), read(), write(), …

- Linux (and other operating systems) can use its hardware resource including files, using system calls

- `int open(const char *pathname, int flags)`
  - Opens a file specified the pathname and returns a file descriptor
- `ssize_t read(int fd, void *buf, size_t count)`
  - Read up to count bytes from file descriptor fd into buf
- `ssize_t write(int fd, const void *buf, size_t count)`
  - Write up to count bytes to file descriptor fd from buf
- `int close(int fd)`: close a given file descriptor, fd

File descriptors

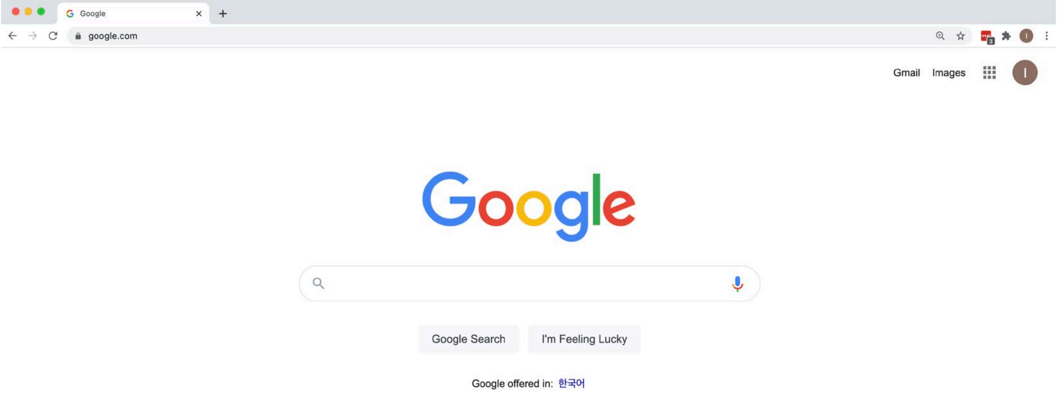
- An integer value used to access a file, network, or I/O operation
  - In Windows, HANDLE corresponds to the file descriptor

- Special file descriptors
  - 0: standard input (stdin) – Keyboard input
  - 1: standard output (stdout) – Screen
  - 2: standard error (stderr) – Screen and no buffering

Process management: Process and thread

- Program: an executable file that contains code and data for execution
- Process: an executing instance of a program
- Thread: an executable unit of a process
    - One thread can have multiple threads

Program



Process

Renderer thread

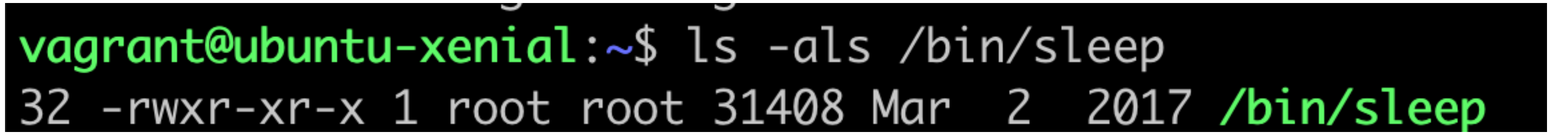
IO thread

UI thread

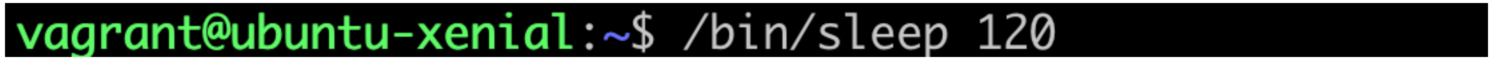
...

Threads

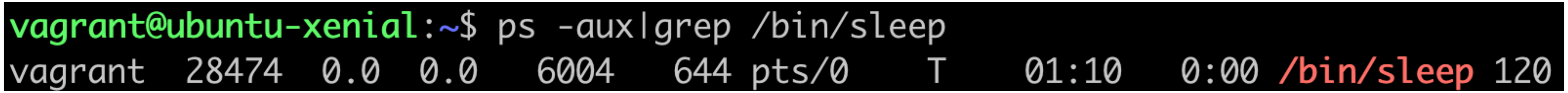
More example



```
vagrant@ubuntu-xenial:~$ ls -als /bin/sleep
32 -rwxr-xr-x 1 root root 31408 Mar  2  2017 /bin/sleep
```



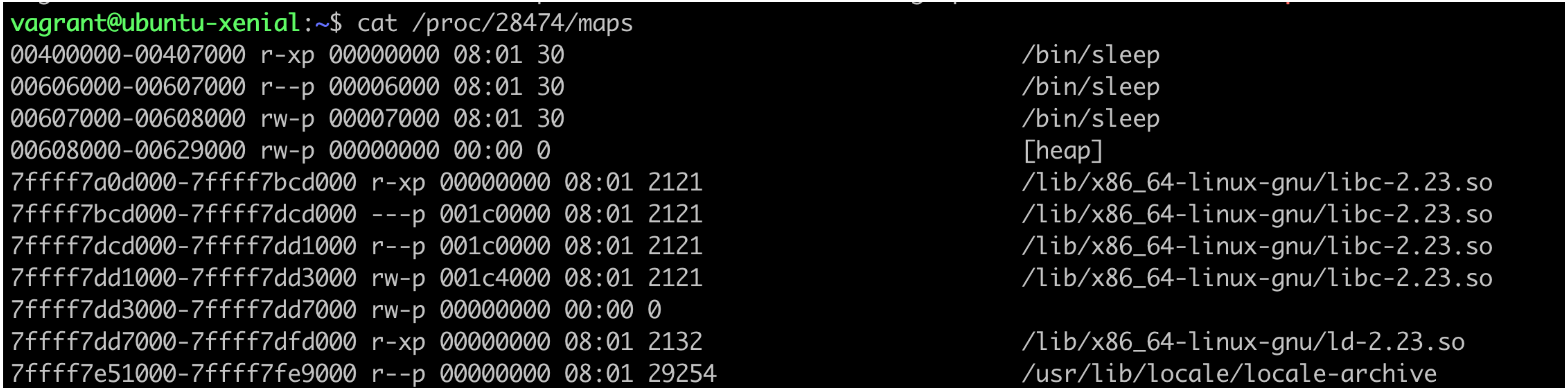
```
vagrant@ubuntu-xenial:~$ /bin/sleep 120
```



```
vagrant@ubuntu-xenial:~$ ps -aux|grep /bin/sleep
vagrant   28474  0.0  0.0   6004   644 pts/0    T    01:10   0:00 /bin/sleep 120
```

Process ID

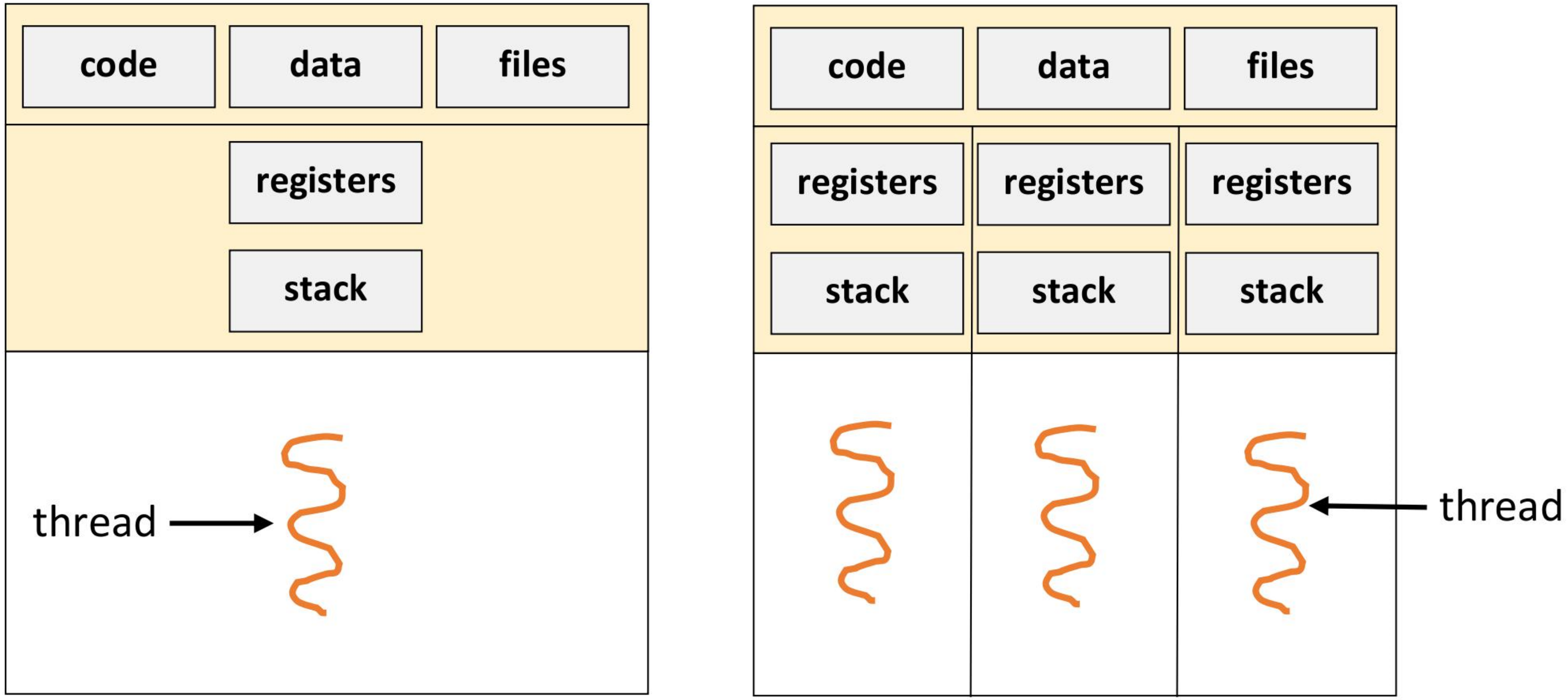
More example



```
vagrant@ubuntu-xenial:~$ cat /proc/28474/maps
00400000-00407000 r-xp 00000000 08:01 30                                 /bin/sleep
00606000-00607000 r--p 00006000 08:01 30                                 /bin/sleep
00607000-00608000 rw-p 00007000 08:01 30                                 /bin/sleep
00608000-00629000 rw-p 00000000 00:00 0                                  [heap]
7ffff7a0d000-7ffff7bcd000 r-xp 00000000 08:01 2121                       /lib/x86_64-linux-gnu/libc-2.23.so
7ffff7bcd000-7ffff7dcd000 ---p 001c0000 08:01 2121                       /lib/x86_64-linux-gnu/libc-2.23.so
7ffff7dcd000-7ffff7dd1000 r--p 001c0000 08:01 2121                       /lib/x86_64-linux-gnu/libc-2.23.so
7ffff7dd1000-7ffff7dd3000 rw-p 001c4000 08:01 2121                       /lib/x86_64-linux-gnu/libc-2.23.so
7ffff7dd3000-7ffff7dd7000 rw-p 00000000 00:00 0
7ffff7dd7000-7ffff7dfd000 r-xp 00000000 08:01 2132                       /lib/x86_64-linux-gnu/ld-2.23.so
7ffff7e51000-7ffff7fe9000 r--p 00000000 08:01 29254                      /usr/lib/locale/locale-archive
```

Q: How many thread does this process have?
(Just guess)

Thread vs Process



single-threaded process

multi-threaded process

Ref: https://medium.com/@yovan/os-process-thread-user-kernel-%E7%AD%86%E8%A8%98-aa6e04d35002

Thread vs Process

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int global = 0;

int main() {
  int status = 0;

  if (fork() == 0) {
    // In child process...
    global++;
    exit(0);
  }

  wait(&status);
  printf("%d\n", global);
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int global = 0;

void* thread_routine(void *arg) {
  global++;
}

int main(){
  pthread_t thread;
  pthread_create(&thread, NULL, thread_routine, NULL);
  pthread_join(thread, NULL);

  printf("%d\n", global);
}
```

Create a process using fork()

- fork(): only way to create a new process
    - Variants exist: clone(), vfork(), …

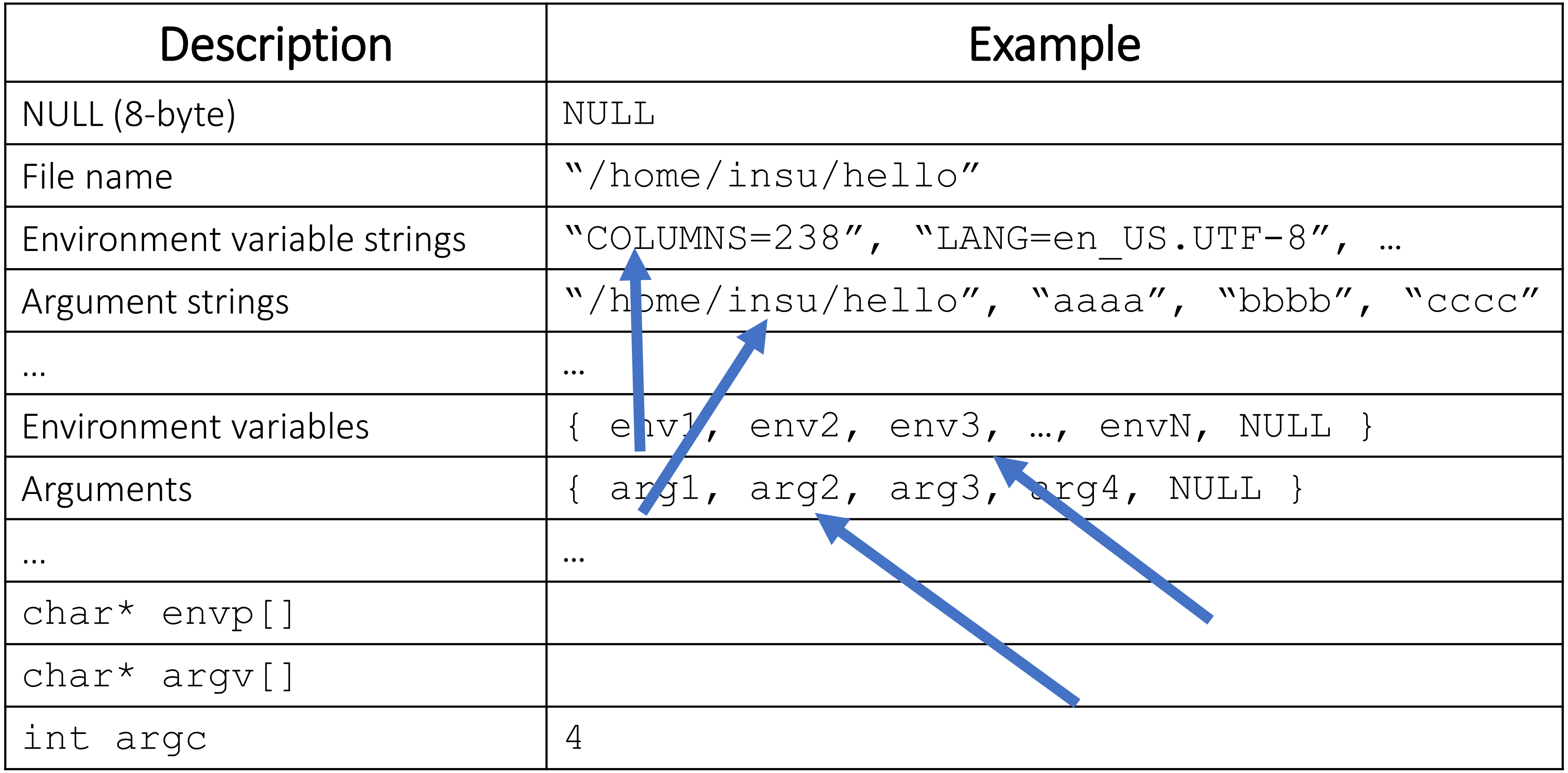
- fork() creates a new process by *duplicating* the current process
    - Copy memory including heap, code, data, and stack
    - Inherits several system resources including file descriptors

Run a new program using execve()

- `int execve(const char *filename, char *const argv[], char *const envp[]);`
  - executes a program pointed by filename
  - `argv`: arguments
    - argv[0] points the filename that are being executed (by convention)
  - `envp`: environment variables
    - Format: KEY=VALUE (e.g., HOME=/home/vagrant)

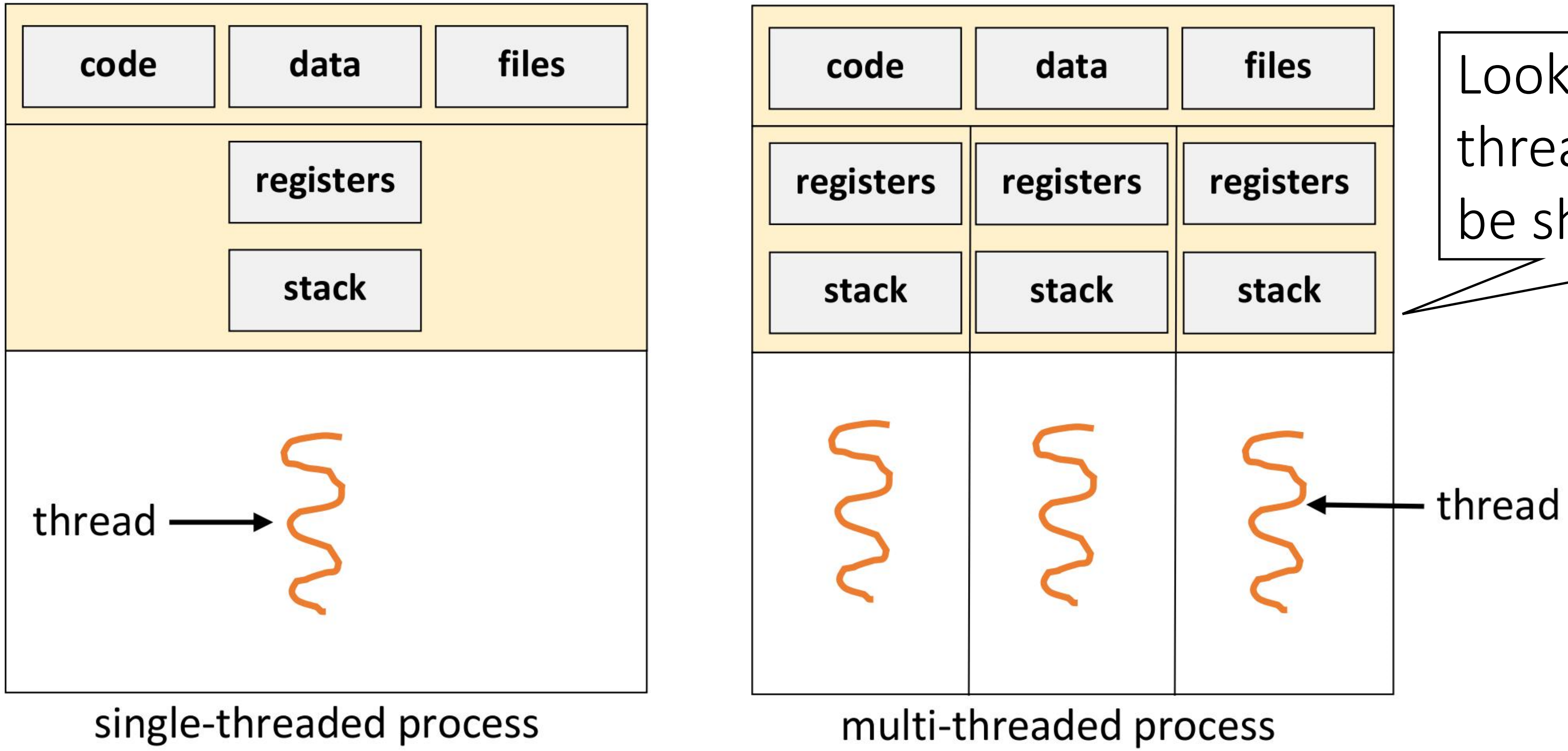
Process layout (32bit in x86-64)

```
$ ./hello aaaa bbbb cccc
```



Description	Example
NULL (8-byte)	`NULL`
File name	`“/home/insu/hello”`
Environment variable strings	`“COLUMNS=238”, “LANG=en_US.UTF-8”,` …
Argument strings	`“/home/insu/hello”, “aaaa”, “bbbb”, “cccc”`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Common misconceptions

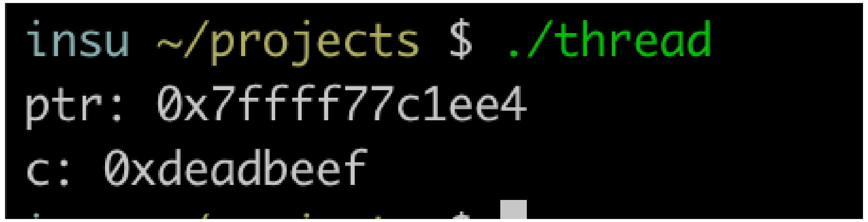


Looks like that a thread's stack cannot be shared!

Ref: https://medium.com/@yovan/os-process-thread-user-kernel-%E7%AD%86%E8%A8%98-aa6e04d35002

Example: sharing stacks across threads

```c
int* ptr = NULL;

void *thread1(void *arg1) {
  int c = 0;
  ptr = &c;
  while (ptr != NULL); // busy waiting

  printf(”c: 0x%08x\n", c);
  return NULL;
}

void* thread2(void *arg) {
  while (ptr == NULL); // busy waiting
  printf(”ptr: %p\n", ptr);

  *ptr = 0xdeadbeef;
  ptr = NULL;
  return NULL;
}
```



```
insu ~/projects $ ./thread
ptr: 0x7ffff77c1ee4
c: 0xdeadbeef
```

- Threads share process memory (e.g., heap, code, data, and even stack)
- Stack is just one kind of memory

- StackClash: Modifying heap from stack
  - https://blog.qualys.com/vulnerabilities-research/2017/06/19/the-stack-clash

Shell

- A command line interpreter for *nix platforms

- It provides diverse functionalities
  - Wildcarding (*)
  - Pipelining (|)
  - Variables
  - …

- You can call shell commands using system() in a C program

How system() works?

- `system(”id”);`

- How does shell know that it needs to execute `/usr/bin/id`?
    - Answer: PATH environment variable

- Type “`printenv PATH`”: `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin`
    - Shell search each path until it finds the specific command

Vulnerability1: PATH injection

- `system(”id”);`

- Add other location to PATH variable
  - `export PATH=/home/attacker/bin:$PATH`
  - Make a binary named “`id`” in `/home/attacker/bin`
  - Run a program that contains `system(“id”)`
  - This will invoke my “`id`” binary, not `/usr/bin/id`

Vulnerability2: Command injection

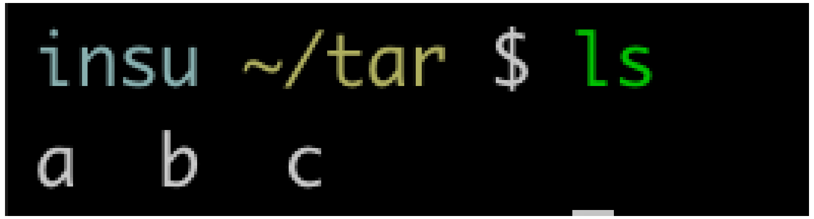
- `system(”/bin/ls ” + input);`

- Shell has many meta-characers
  - e.g., “;” can represents command separator

- Thus, if `input=”; /bin/sh”,` the above code will spawn a shell for you

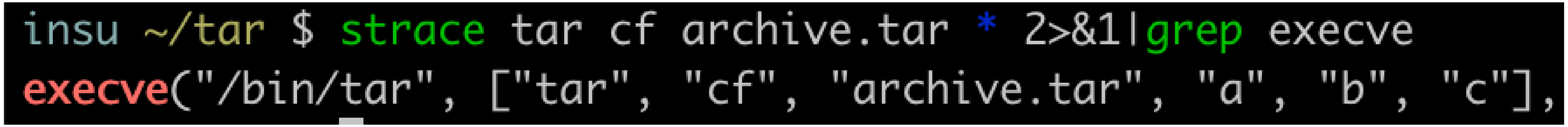
Wildcard injection

- `system(”/bin/tar cf archive.tar *”);`
- You can make any file for compression



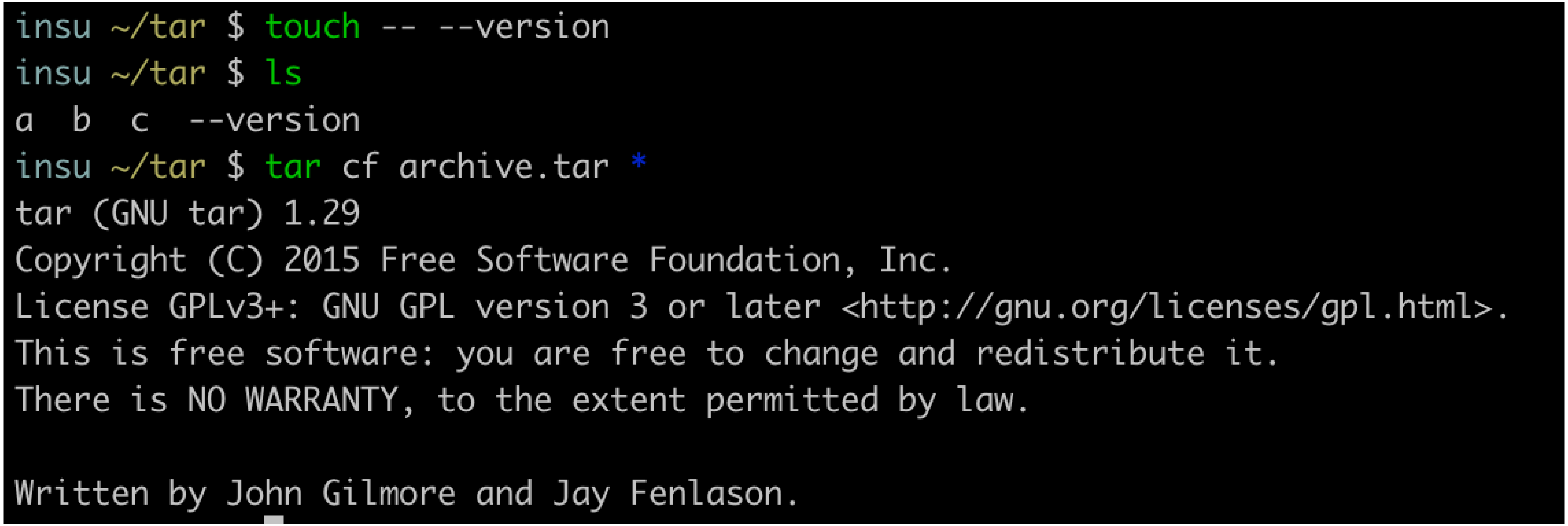
```
insu ~/tar $ ls
a  b  c
```

Shell inserts file names as arguments!!



```
insu ~/tar $ strace tar cf archive.tar * 2>&1|grep execve
execve("/bin/tar", ["tar", "cf", "archive.tar", "a", "b", "c"],
```

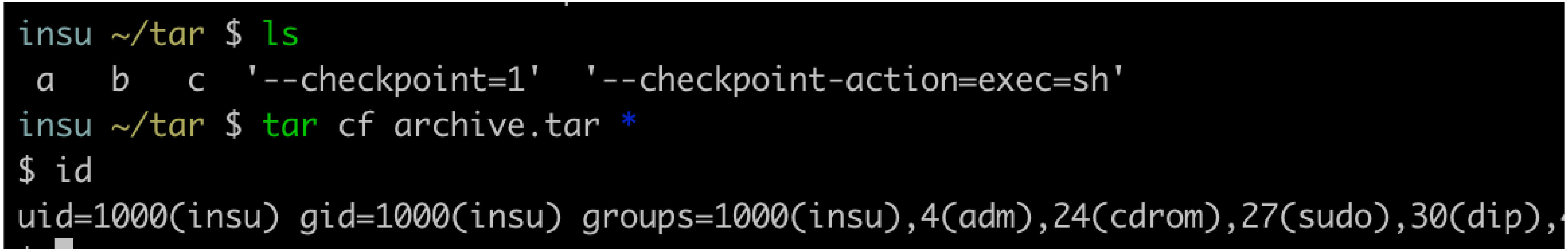
Wildcard injection



```
insu ~/tar $ touch -- --version
insu ~/tar $ ls
a  b  c  --version
insu ~/tar $ tar cf archive.tar *
tar (GNU tar) 1.29
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by John Gilmore and Jay Fenlason.
```

Wildcard injection

- --checkpoint=[N]: Display progress messages every Nth record
- -- --checkpoint-action=ACTION: Run ACTION on each checkpoint
  - One of its action is ‘exec’, which allows you to execute external command!



```
insu ~/tar $ ls
 a   b   c  '--checkpoint=1'  '--checkpoint-action=exec=sh'
insu ~/tar $ tar cf archive.tar *
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),24(cdrom),27(sudo),30(dip),
```

Shellshock

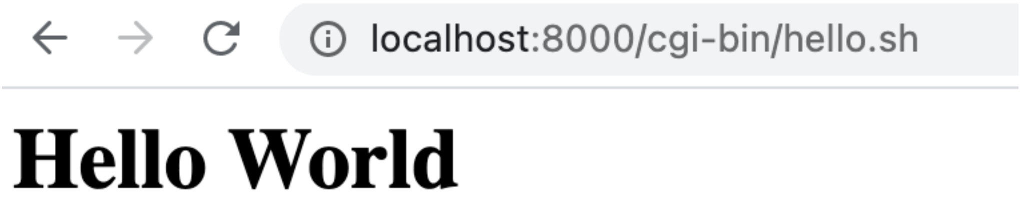
- Discovered in September 2014
- Malformed environment variables in bash allows command injection

  -
    ```
    env x='() { :;}; echo vulnerable’
          bash -c "echo this is a test"
    ```

Example: Common Gateway Interface (CGI)

- Web interface to execute programs like console applications
  - Frequently used in an embedded system (e.g., router, …)

```
#!/bin/bash
echo “Content-Type: text/html”
echo
echo “<h1>Hello World</h1>”
```



localhost:8000/cgi-bin/hello.sh

Hello World

- CGI converts inputs from web into environment variables
  - e.g., User-agent → HTTP_USER_AGENT=“….”

Shellshock on CGI servers

-
  ```
  env x='() { :;}; echo vulnerable’
              bash -c "echo this is a test"
  ```

-
  ```
  curl -H "User-agent: () { :;}; echo vulnerable" http://localhost/cgi-bin/hello.sh
  ```
  - Then,
    ```
    HTTP_USER_AGENT=‘() {:;}; echo vulnerable’
                bash hello.sh
    ```

Lesson: Be careful when you use shell command!