Course Overview

Sicurezza (CT0539) 2025-26
Università Ca’ Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Objectives

Sicurezza (CT0539)

https://www.unive.it/data/course/451344/programma

This course aims at providing:

- knowledge of **attack** and **defence** techniques related to program exploitation, system, network and web security
- skills related to securing **real** systems and networks, developed through **practical exercises**

Programme

Sicurezza (CT0539)

https://www.unive.it/data/course/451344/programma

1. Background and tools
2. Program analysis
3. Program exploitation
4. System and network security
5. Web security (server side)
6. Web security (client side)

Material

Sicurezza (CT0539)

https://www.unive.it/data/course/451344/programma

- Course **official website** (with slides and on-line material):

  https://moodle.unive.it/mod/page/view.php?id=1014651

- The course is mainly based on on-line material
  For program exploitation you can refer to J. Erickson, *Hacking, the art of exploitation*, No starch press, 2008

Assessment

Sicurezza (CT0539)

https://www.unive.it/data/course/451344/programma

Written test (base mark)

Non-mandatory **assignments** (extra score)

- *Challenges* on attacking and securing IT systems and networks
- Bonus score with respect to the mark of the written test

Lab

Sicurezza (CT0539)

https://www.unive.it/data/course/451344/programma

Course is based on many practical examples and exercises

We will provide **docker containers** that can be run under Linux, Windows, Mac

Identical “testbeds” independently of the host operating system

- either install docker
- or use Linux VM with docker

Background and tools

1. Unix shell
2. sed and regular expressions
3. Python

Introduction and Unix shell

Sicurezza (CT0539) 2025-26
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Unix shell

Unix shell allows for **quickly** automating interaction with processes and data

Knowing the shell helps **understanding** interaction with programs (processes)

We revise basic Unix shell commands and concepts

Unix shell is the **simplest interface** to the operating systems

- **Execute** programs
- **Redirect** input/output
- **Connect** programs together
- Run **scripts**

We focus on **bash** (Bourne-again shell, pronounced *born-again*), successor of Bourne's shell **sh**

Basic commands (1)

`ls`: shows the **content** of current directory. `-l` displays long format; `-a` displays hidden (dot) files

`file filename`: shows the **type** of file named `filename`

`pwd`: (print working directory) shows the **path** of current working directory

`mkdir name`: creates a **new directory** in the current working one

`cd path`: (change directory) moves working directory to `path`

`cat file`: shows file **content**

`cat f1 f2 f3`: displays the **concatenation** of `f1 f2 f3`

`echo "hello"`: prints “hello”

`grep word file`: looks for `word` in `file` and prints lines that contain it

Basic commands (2)

`man command`: shows `command` **man page**. Arrows up and down navigate, q exits, / searches (n next hit, N previous hit)

`find path expression`: looks for files in `path` (recursively) **matching** the specified expression

Ex. : `find / -name "*.c" -print` prints all the file that ends with .c

`sort file`: **sort** lines of a text file

`strings file`: find printable **strings** in a (binary) file

Example:

```
$ strings /usr/bin/passwd | grep changed
password for '%s' changed by '%s'
%s: password expiry information changed.
passwd: password unchanged
$
```

Wildcards

***** : Matches any string, including the null string

? : Matches any single character

[...] : Matches any one of the enclosed characters; a pair of characters separated by a hyphen denotes a **range** expression

```
$ ls test[0-9].???
test1.txt  test2.txt
```

NOTE: '.' at the start of a filename or immediately following a slash must be matched explicitly, unless the shell option **dotglob** is set

```
$ ls *bash*
ls: cannot access '*bash*' ...
$ ls .bash*
.bash_logout  .bashrc
$ shopt -s dotglob
$ ls *bash*
.bash_logout  .bashrc
```

Input from terminal

A typical behaviour of Unix shell commands is to **take input from the user** when no file is specified

ctrl-D is interpreted as End-of-File (EOF) and terminates the program

Example 1 :

```
$ cat
Hello this is a test
Hello this is a test
(ctrl-D terminates)
$
```

Example 2 (grep):

```
$ grep work
I'm checking what happens when
grep is run without specifying
a filename!
How does this work?
How does this work?
ah: matching line are printed
out as expected!
(ctrl-D terminates)
$
```

Redirection

Fundamental Unix shell mechanism to **redirect** program input and output from/to a file

When output is **redirected to** a file (**symbol >**) any output from the program will be written to the file

When input is **redirected from** a file (**symbol <**) the content of the file will be sent as input to the program

Examples:

`ls > tmpfile`: write the content of the current folder into file tmpfile. Check with `cat tmpfile`

`grep shell < tmpfile`: redirects the content of the file to the grep command.
NOTE: The behaviour is the same as **`grep shell tmpfile`**

Redirection (examples, see also here)

With symbol **>>** we can **append** output to an existing file:

`date >> tmpfile`: appends current date to file `tmpfile`

Note: overwriting is done **silently** so be careful when using redirection with a single **>**

`date > tmpfile`: overwrites!

What happens if we redirect the output of a command that takes input from the terminal?

Example (cat):

```
$ cat > test.txt
Hello this is a test
of two lines
(ctrl-D)
$
```

⇒ input is written into file `test.txt`!

Redirecting stdout or stderr

In Unix there are **three** separate input/output streams:

- **`stdin (0)`**: standard input, where the program takes input
- **`stdout (1)`**: standard output, the normal program output
- **`stderr (2)`**: standard error, where the program prints error

1> and **2>** respectively redirect stdout and stderr

Example (hide errors):

```
$ ls
test1.txt  test2.txt
$ cat test*
cat: test1.txt: Permission denied
This is readable
$ cat test* 2> /dev/null
This is readable
$ cat test* 1> /dev/null
cat: test1.txt: Permission denied
```

Pipes

Fundamental mechanism for process **communication** in Unix

Similar to redirection but works between **two programs**

Channel between processes: a process can **write** to the pipe and another one can **read** from it

⇒ **combine** commands conveniently

In the Unix Shell, pipes are specified using symbol `|`

`cmd1 | cmd2 | ... | cmdn`, executes all commands and the output of each command i is given as input to the next command i+1

The output of the last command is printed on the terminal

Pipes (examples, see also here)

`ls | grep shell`: shows all file names that contain word shell

`ls | grep shell | sort -r`: as before but file names are sorted in reverse alphabetical order (option -r). Notice that in this case we have three programs cooperating together;

`ls | grep shell | grep txt`: shows all file names that contain both shell and txt

Example:

```
$ ls
myshell.pdf  shell.txt  test.txt

$ ls | grep shell
myshell.pdf
shell.txt

$ ls | grep shell | sort -r
shell.txt
myshell.pdf

$ ls | grep shell | grep txt
shell.txt
```

The Bandit wargame

Now you can **refine your shell skills**
solving levels (up to 9) of Bandit wargame:

https://overthewire.org/wargames/bandit/