Linux Basics II

Stephan Müller, ISG D-PHYS

HS 2025

Schedule

IT Security

Linux Basics I

Linux Basics II

- Remote Shells vis SSH
- SSH Key Management
- Terminal Multiplexing
- I/O Redirection
- Regular Expressions
- Shell Globbing
- Bash Variables
- Environments
- Shell Scripting

Git Version Control

Python Ecosystem I

Python Ecosysetm II

Python Ecosysetm III

Python Ecosysetm IV

System Aspects

- Checkout compenv.phys.ethz.ch
- Give Feedback!
  - isg@phys.ethz.ch
  - chat.phys.ethz.ch
  - HPT H

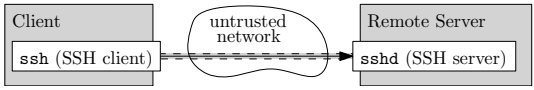
SSH
Overview

SSH (Secure Shell)

SSH is network protocol with strong encryption used to provide logins and a shell on a remote system

- Asymmetric cryptography
- Client/Server

- SSH Server (`/usr/sbin/sshd`) on remote system
  - always running, waiting for incoming connections
- SSH client (`/usr/bin/ssh`) on local system
  - run on demand by user
- SYNOPSIS: `ssh [OPTIONS] LOGIN_NAME@DESTINATION [COMMAND]`
  - `DESTINATION`: DNS name a remote machine with an SSH server
  - `LOGIN_NAME`: username on the remote machine
  - `COMMAND`: single command to run on the remote machine



First Login / Server Fingerprint

```
[alice@laptop: ~]$ ssh asmith@login.phys.ethz.ch
The authenticity of host 'login.phys.ethz.ch (129.132.89.195)' can't be established.
ECDSA key fingerprint is SHA256:upncE1in1QVEyXEeafC/WOPpK8QtZ/skpxU7GwT1pUk.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
asmith@login.phys.ethz.ch's password: asmith's-D-PHYS-password

[asmith@phd-login1: ~]$ exit
```

SSH Keys

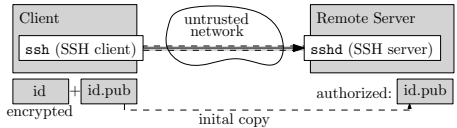
- SSH keys are a replacement for passwords (i.e. authentication)

  `~/.ssh/id`: Private key

  - Stays local and secret
  - Should be protected by a password

  `~/.ssh/id.pub`: Public key

  - Has to configured on the remote system
  - Only the private key holder can prove, that he knows the corresponding private key (*digital signature*)



Public Key Example:

```
ssh-ed25519 AAAAC3NzaC11ZDI1NTE5AAAAIG5
            Csv7paLFNcTTIry5jMX/4JK20mD
            3sUEUNm2I6pt1w alice@laptop
```

Generating SSH Keys

```
[alice@laptop: ~]$ ssh-keygen -t ed25519
  Generating public/private ed25519 key pair.
  Enter file in which to save the key (/home/alice/.ssh/id_ed25519):
  Enter passphrase (empty for no passphrase): private-key-password
  Your identification has been saved in id_ed25519
  Your public key has been saved in id_ed25519.pub
  The key fingerprint is:
  SHA256:KsXhkRCJFq17V5hwgiQx+Wt/gbv4OgxmbovNim1fTEM alice@laptop
```

SSH Key Types

- RSA keys
  - based on prime factoring
  - use at least 4094 bit as key size
  - `ssh-keygen -t rsa -b 4096`
- ed25519 keys
  - based on discrete logarithms
  - fixed key length
  - `ssh-keygen -t ed25519`

Authorized Keys on the SSH Server

- On the remote machine: `sshd` reads `~/.ssh/authorized_keys`
  - List of public keys (user identities), one per line
  - May be edited or filled by hand
- Everybody with a corresponding private key is granted a login
  - Everybody who can cryptographically sign an authentication request

Copying Keys with ssh-copy-id

```
# on the local system
ssh-copy-id -i  /.ssh/id_ed25519.pub asmith@login.phys.ethz.ch
 ssh-copy-id: INFO: 1 key(s) remain to be installed
 if you are prompted now it is to install the new keys
 (asmith@login.phys.ethz.ch) Password: D-PHYS-password

 Number of key(s) added: 1
```

Copying Keys Manually

```
# on the remote system
cd
mkdir -p .ssh
echo "ssh-ed25519 AAAAC3NzaC11ZDI1NTE5AAAA
  IG5Csv7paLFNcTTIry5jMX/4JK20mD3sUEUNm2I6
  pt1w" >> .ssh/authorized_keys
chmod 700 .ssh
chmod 600 .ssh/authorized_keys
```

Key Mangement with ssh-agent

- Daemon process on the client side
- Caches key passphrases - unlock only once
  - Start with `eval $(ssh-agent)`, if not automatically started

SSH Extras

X-Forwarding with SSH

```
[alice@laptop: ~]$ ssh -Y asmith@login.phys.ethz.ch
[asmith@phd-login1: ~]$ mathematica &
```

- SSH option `-Y` enables X-forwarding
- The program runs on the remote machine, just graphics are forwarded
- `&` = run in "background", keeping the shell available

Agent Forwarding

- Use `ssh -A` to enable agent forwarding
  - Simulate the presence of an `ssh-agent` on a remote server
- Allows "multi-ssh": `local` → `server1` → `server2`
  - `server1` requires the private key to login to `server2`
  - The ssh-agent on `local` forwards responses to `server2`

SSH Jumping

```
ssh -J server1 server2
```

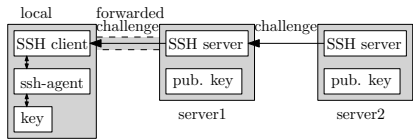
- `server` establishes a TCP forwarding to `server2`
- No agent needed

Secure Copy

SYNOPSIS: `scp [OPTION]... SOURCE TARGET`

- Semantics like `cp`
- Either SOURCE or TARGET can be remote
  - `[user@]host:path`
- Copy a remote a file to the local directory:

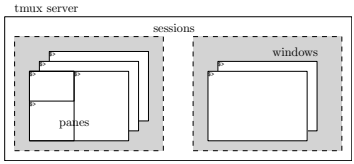
```
scp asmith@login.phys.ethz.ch:/home/asmith/book.pdf .
```



Terminal Multiplexing

tmux: Terminal Multiplexer

- Session and windows management of shells
- Can be used to keep sessions alive
  - Start a tmux server on a remote system
  - All sessions stay alive and can be reattached:



Outside a Session

- `tmux new-session -s NAME`: Create new named session
- `tmux attach-session -t NAME`: Attach to session `NAME`
- `tmux kill-session -t NAME`: Kill session `NAME`
- `tmux list-sessions`: List existing sessions

Inside a Session

- CTRL + b d : Detach session
- CTRL + b s : Show session selector
- CTRL + b c : Create new window
- CTRL + b ? : Show command help

Example

```
[alice@laptop ~]$ ssh -J asmith@login.phys.ethz.ch labpc
[asmith@lappc ~]$ tmux new-session -s alice_ethz
[asmith@lappc ~]$ python big_job
  starting calculation..
```

CTRL + b d

```
[asmith@lappc ~]$ logout
```

Everything continues in the detached session as in an active terminal, even after logging out.

```
[alice@laptop ~]$ ssh -J asmith@login.phys.ethz.ch labpc
[asmith@lappc ~]$ tmux list-sessions
  alice_eth: 1 windows (created Mon Oct 18 06:46)

[asmith@lappc ~]$ tmux attach-session -s alice_ethz
[asmith@lappc ~]$ python big_job
  starting calculation..
```

Input/Output Redirection

Guiding Principles

- "Less is More" & "Everything is a file"
  - Simple tools
  - All operate on files
  - Many things can be used as files
  - Ability of combining tools

Simple Tools

`sort`: Sort lines in a file

`tr`: Replace characters in a file

`uniq`: Find double lines in a file

`cut`: Split lines in a file

`head`: Retrieve first lines in a file

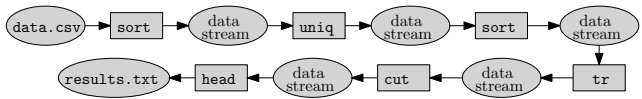
`tail`: Retrieve last lines in file

`wc`: Count lines in a file

`nl`: Number lines in a file

- Find the five 5 most common values in `data.csv` and store them in `results.txt`:

```
sort data.csv | uniq -c | sort -rn | tr -s "" | cut -d"" -f 3 | head -n 5 > results.txt
```



Input/Output Redirection
File descriptors

File Descriptors

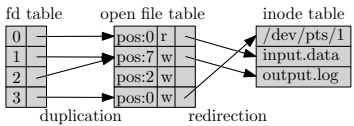
- An open file is represented as *file descriptor* (fd)
- Numbered 0,1,2,3, ...

0	stdin	input for that process
1	stdout	normal output
2	stderr	errors and extra outout

- Usually connected to (pseudo-)terminal device in `/dev/pts/`

Redirection & Duplication

- `n> file`: Let fd `n` write to file (truncate)
- `n>> file`: Let fd `n` write to file (append)
- `n< file`: Let fd `n` read from file
- `n>&m`: Make (the writable) fd `n` a duplicate of fd `m`



Examples

- Inspecting all file descriptors of process with `pid` 153442:

```
ls -og /proc/153442/fd
 lrwx------. 1 64 0 -> /dev/pts/3
 lrwx------. 1 64 1 -> /dev/pts/3
 lrwx------. 1 64 2 -> /dev/pts/3
```

- Redirecting stdin, stdout, stderr:

```
./my_script <input.data >output.log 2>/dev/null
```

  - stdin reads from `input.data`
  - stdout writes to `output.log`
  - stderr writes to `/dev/null` (discard)

- Duplicate file descriptors:

```
./my_script >output.log 2>&1
```

  - stdout writes to `output.log`
  - stderr becomes a duplication of stdout
  - NOT: `./my_script >output.log 2>output.log`

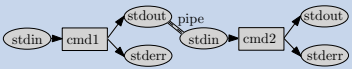
- "Swap" stdout and stderr:

```
./my_script 3>&1 1>&2 2>&3-
```

  - fd 3 becomes a duplication of stdout
  - stdout becomes a duplication of stderr
  - stderr becomes a duplication of fd 3

Redirection with Pipes

- `cmd1 | cmd2`: redirect `cmd1`'s stdout to `cmd2`'s stdin



- Typical filter idoms:

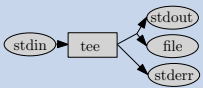
  `... | less`: read output in a pager
  `... | grep`: filter output with regular expressions
  `... | awk`: mangle output
  `curl ... | bash`: remote code execution vulnerability

- `cmd1 |& cmd2`: short for `cmd1 2>&1 | cmd2`

Splitting Output

- `tee [OPTION]... [FILE]...`

  Read from stdin and write to stdout and FILE



Examples

- Inspecting file descriptors of `cmd1 |& cmd2`:

```
ls -og /proc/@($(pidof -x cmd1)|$(pidof -x cmd2))/fd
/proc/189796/fd:
lrwx------. 1 64 0 -> /dev/pts/0
l-wx------. 1 64 1 -> 'pipe:[2141860]'
l-wx------. 1 64 2 -> 'pipe:[2141860]'

/proc/189797/fd:
lr-x------. 1 64 0 -> 'pipe:[2141860]'
lrwx------. 1 64 1 -> /dev/pts/0
lrwx------. 1 64 2 -> /dev/pts/0
```

- Process json from a REST api:

```
curl -X GET "https://example.org/api/foo" | jq keys
 [
  "info",
  "data",
 ]
```

- See stdout on the terminal and store in a file:

```
sudo dnf update | tee output.log
 Resolving dependencies
 Update:
 chromium            94.0.4606.81-1.fc34
 chromium-common     94.0.4606.81-1.fc34
```

Regular Expressions

How to write regular Expressions

Overview

- Specify or find textual pattern
- A lot of different languages (flavours) to write regular expressions
  - POSIX *extended regular expressions* (ERE)
  - *Perl compatible regular expressions* (PCRE)
  - Most programming languages, have their own regex engine
  - Bash globbing

Matching

`a`	Single regular character `a`, analogous for others
`.`	Any character, but newline
`^`, `$`	The beginning(`^`) or end(`$`) of a line
`[a-f]`	One character from `a` to `f`
`[[:digit:]]`	A single digits, i.e. (mostly) equivalent to `[0-9]`
`[^<>]`	Any character except `<` and `>`
$expr_A \vert expr_B$	Matches $expr_A$ or $expr_B$

Quantifiers / Repitions

`{m,n}`		at least `n` times and at most `m` times
`{n}`	`{n,n}`	exactly `n` times
`*`	`{0,}`	arbitrarily often, may be not at all
`+`	`{1,}`	arbitrarily often, but at least one time
`?`	`{0,1}`	at most one time

Examples

- All last words on a line which start with `a`:

  `a[[:alpha:]]*$`

  - It is almost time to eat an apple
- Any "hex-number": `0x[0-9a-f]+`:
  - `0x01bf`, `0x0`, `0x`, `0x0BAD`
- Script names of length 5: `[A-Za-z0-9._-]{5}\.(py|sh)`
  - `Run_3.py`, `setup.sh`
- Integers 0-255:

  `^[0-9]$|^1?[0-9]{2}$|^2[0-4][0-9]$|^25[0-5]$`

  NOT: `^[0-255]$`
- Greedy matches: `".*"`
  - `"Run!", she said, "I know regular expressions"`
- Multiple matches: `"[^"]*"`:
  - `"Run!", she said, "I know regular expressions"`
  - `"Run!", she said, "I know regular expressions"`

grep & sed

Regular Expressions on the Commandline

grep: global/regular expression/print

- SYNOPSIS: `grep [OPTION...] PATTERN [FILE...]`

  Print lines in FILE which match PATTERN
- Usually `grep -E` or `egrep` for POSIX-ERE
- Most common options:

  `-o`: only show matching part
  `-v`: invert, show not matching lines
  `-i`: match case-insensitive
  `-r`: recursively search directory

sed: stream editor

- SYNOPSIS: `sed [OPTION...] SCRIPT [FILE...]`

  Perform transformation on each line in FILE
- Most common usages:

  `sed 's/EXPR/STRING/g'` Replace EXPR by STRING
  `sed '/EXPR/d'` Delete all lines whith pattern
  `sed -i` Inplace, edit file
- Delimiters can be changed: `sed 's!EXPR!STRING!g'`

Examples

- Is a process `cmd` running:

  ```
  ps aux | grep cmd1
  ```

  ```
  alice 24592 0.0 pts/4 S+ /bin/bash ./cmd1
  alice 24895 0.0 pts/1 S+ grep cmd1
  ```
- Filter `cmd`'s output, show only lines which not contain "debug", but do contain "info":

  ```
  ./cmd | grep -v debug | grep info
  ```
- Find all occurrences of "todo" or "fixme" in the slide directory:

  ```
  grep -Eri "fixme|todo" slides/
  ```

  ```
  main.tex: \begin{block}{Example:} %Fixme: spacing
  animals.txt: mastodons are even bigger than elephants
  ```
- Remove all empty lines from config.json:

  ```
  sed -i '/^[[:space:]]*$/d' config.json
  ```
- Parse and extract "bar" from "foo=bar" in config file:

  ```
  grep -Eo "^foo=[[:alnum:]]*" config.ini | sed 's/foo=//'
  ```

Shell Globbing

Globbing

Bash expands or replaces certain expressions to file names, this is called *File name expansion* or *globbing*.

- Similar syntax to regular expressions
- "Matches" against valid filenames
- Expressions are expanded before command execution, e.g. (`ls *txt` is replaced by `ls fileA.txt fileB.txt`)

Matching

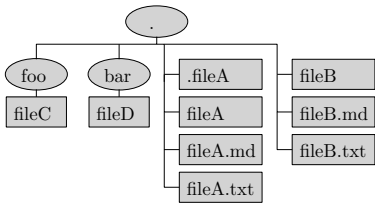
`a`	single regular characters `a`
`\?`	single meta-character `?`
`?`	wildcard, single character, except `/`
`*`	string of any length, not containing `/`
`[`$class$`]`	just as POSIX regular expressions classes
`[^...]`	*negation*, any character except ...

Qunatifiers

$expr_A \vert expr_B$	*list* of alternatives
`?(`$list$`)`	*at most one* match from $list$
`+(`$list$`)`	*one or more* matches from $list$
`*(`$list$`)`	*zero or more* matches from $list$
`@(`$list$`)`	*exactly one* match from $list$
`!(`$list$`)`	match anything *not in* $list$

Examples

- `*` expands to all file names, including directories, excluding hidden files
- `.*` expands to `. .. .fileA`
- `!(fileA|*.??)` expands to `bar`, `fileA.txt`, `fileB`, `fileB.txt`, `foo`
- `*/file[A-Z]` expands to `foo/fileC`, `bar/fileD`
- `@(fileA|fileB.txt)` expands to `fileA`, `fileB.txt`
- `*.mp3` expands to `*.mp3`



Shell Variables

Bash: Variables

- Assignment: `VAR=value`
  - No declaration necessary
  - Names start with a letter, usually all uppercase
  - Values are handled as strings
- Reference: `$VAR` or `${VAR}`
  - Undefined variables evaluate to an empty string
- `unset VAR`: Undefine a variable
- `set`: List all variables

Bash: Environment Variables

- *environment variables* $\subseteq$ *(local) variables*
- Used as process environment / configuration
  - Python virtual environments
- `export VAR`: Make `VAR` an environment variable
  - `export VAR=foo`
  - `VAR=foo command`
- `env`: List all environment variables

Examples

- Basic usage:

```
B=123
echo "The values of A and B are $A and $B!"
 The values of A and B are and 123!
```

- Run a variable as as command:

```
CMD=du ; OPTIONS=-hs ; FILE=/home/alice/*
$CMD $OPTIONS $FILE
 68K  /home/alice/Documents
 1.2G /home/alice/Downloads
```

- Assign command output to a variable:

```
LAST_COMMIT=$(git rev-parse --short HEAD)
# LAST_COMMIT=c54b91a61
```

Important Enviroment Variables

- `PATH`: A `:` separated list of path to look for executable files
- `HOME`: Current users home directory
- `LANG`: Preferred language for user interfaces
- `EDITOR`: Preferred text editor

Shell Variables

Process Locality of Variables

- Every process has its own copy of variables
  - Changes in one process to not alter other processes
- Variables are inherited to new processes
  - Child processes inherit only the environment (fork + exec)
  - Subshells inherit everything (fork)

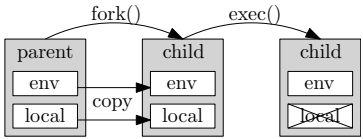
Sourcing scripts

- `source FILE`: Run script in current shell
  - Used to modify current environment
  - Alternative Syntax: `. FILE`
- `eval CMD`: Run command in the current shell

Permanent Changes to the Environment

- `~/.profile` Sourced by every login shell (new SSH)
- `~/.bashrc` Sourced by every new shell (new terminal)
- `/etc/profile`
- `/etc/bashrc`

`fork` copies all variables, `exec` clears local variables



Examples

- Inject `LD_PRELOAD` into `my_program`'s environment:

```
LD_PRELOAD=mylib.so ./my_program
```

- Setup a *Python virtual environment*:

```
source bin/activate # OK, modifies the current env
bin/activate        # NO, modifies the child's env
```

- Aliases in `~/.bashrc` (top 10 cpu consuming processes):

```
alias cpu10="ps -Ao pid,%cpu,comm --sort -%cpu | head -n11"
```

- Setup permanent environment in `~/.bashrc` (SSH agent):

```
eval $(ssh-agent)
```

Script Interpreter

- First bytes must be `#!` followed by an interpreter

```
#!/bin/bash
#!/usr/bin/python
#!/bin/bin/awk -f
```

- Don't forget `chmod u+x`
- Script file becomes the 1st argument to interpreter:

`./script.sh` → `/bin/bash ./script.sh`

Special Variables

- `$?`: Exit code of the last program
  - 0: Success / non-zero: Error
- `$$`: pid of the current process
- `$N`: Arguments to a shell script
  - `$0`: Script name
  - `$1`: 1st argument
- `$#`: Number of arguments to a shell script

Example: Simple Script

```
cat script.sh
  #!/bin/bash
  echo "Hello I am $0 with pid $$"
  echo "I have $# args, 1st: $1"
  exit 4

./script.sh foo bar
  Hello I am ./cmd3 with pid 229051
  I have 2 args, 1st: foo

echo $?
  4
```

Boolean Operators

`cmd1 && cmd2`: Exit code 0 iff `cmd1` AND `cmd2` successful

`cmd1 || cmd2`: Exit code 0 iff `cmd1` OR `cmd2` successful

`! cmd1` : Exit code 0 iff `cmd1` NOT successful

- Lazy evaluation:

```
cmd && echo "SUCCESS" || echo "ERROR"
```

Shell Scripting
Conditions

If-Then-Else

```
if cmd1 ; then
    # ...
elif cmd2 ; then
    # ...
else
    # ...
fi
```

- Exit code from `cmd` is evaluated (0 == true)
- Write test conditions with `/usr/bin/[` or `[[` (bash built-in)

Writing Tests with [[

- `[[ $A -eq 123 ]]` Numerical (integer) conditions:

  `-eq` (=), `-ne` (≠),`-ge` (≥), `-gt` (>), `-le` (≤), `-lt` (<)
- `[[ $A = foobar ]]` String conditions:

  `=` equality, rhs supports shell globbing
  `=~` equality, rhs supports regular expressions
  `!=` not-equal
- `[[ -f FILE ]]` File conditions:

  `-f` (is regular file), `-x` (is executable), `-e` (exists)

Examples

- Error handling:

```
tar -czf archive.tgz ${FILES}
if [[ $? -ne 0 ]] ; then
    echo "Could not create archive"
fi
```

- Run command only, if it is executable:

```
[[ -x foo.sh ]] && ./foo.sh
```

- Only start `ssh-agent` on specific host:

```
if [[ $HOSTNAME = "labpc" ]] ; then
    eval $(ssh-agent)
fi
```

- Parameter sanity checks:

```
if [[ $# -ne 1 ]] ; then
  echo "Error: Need an argument" 1>&2
  exit 1
elif ! [[ $1 =~ ^[+-]?[0-9]+$ ]] ; then
  echo "Error: arg is not an integer" 1>&2
  exit 2
fi
```

Classic Loops

```
while cmd ; do    # loop when cmd exits with 0
   # ...
done

until cmd ; do  # loop when cmd exits with non-zero
   # ...
done
```

For Loop

```
for ITEM in LIST ; do
   # do something with $ITEM
done
```

- LIST is shell expanded, loop execution for each member

Loops

- Loops are (compound) commands:
  - Redirection applies
  - Exit-code is the exit code of the last inner command

Examples

- Loop over all lines in a file `data.txt`:

```
while IFS="" read LINE ; do
    youtube-dl "$LINE"
done < data.txt
```

`IFS` (Internal Field Separator) Environment variable

SYNOPSIS: `read [OPTION]... [VAR]...`

Read line from stdin and assign words to variables

- Convert all .jpg to smaller .png:

```
for FILE in *.jpg ; do
   BASE="$(echo $FILE | sed 's/.jpg$//')"
   echo "$FILE -> $BASE"
   convert $FILE -resize 50% $BASE.png
done
```

- Loop over Integers:

```
for N in $(seq 1 99) ; do
   echo "I got $N problems"
done
```

SYNOPSIS: `seq FIRST LAST`

Print numbers from `FIRST` to `LAST`

THANK YOU

Schedule

Linux Basics I

Linux Basics II

- Remote Shells vis SSH
- SSH Key Management
- Terminal Multiplexing
- I/O Redirection
- Regular Expressions
- Shell Globbing
- Bash Variables
- Environments
- Shell Scripting

Git Version Control

Python Ecosystem I

Python Ecosysetm II

Python Ecosysetm III

Python Ecosysetm IV

System Aspects

- compenv.phys.ethz.ch
- Give Feedback!
  - isg@phys.ethz.ch
  - chat.phys.ethz.ch