EasyCTF Tutorials

Michael Zhang

Published with GitBook

Table of Contents

EasyCTF Tutorials

Hey there! If you're new to hacking, this is definitely a great place to start!

Hacking

Hacking is probably a little different from what you think it is. It's not about breaking into people's machines and stealing their passwords. It's about learning about systems and how they work, and most importantly, how to use this information to your advantage. According to PicoCTF 2013, here's what hacking's all about:

- Curiosity
- Learning through hands-on experimentation
- Striving to understand all parts of a system
- **Using this knowledge for good**

In other words, hackers are simply curious people. However, there are rules and legal stuff you have to worry about if you're a hacker, and that's the main reason why hackers have such a bad image today: irresponsible people breaking into systems they have no permission to be in and stealing information. PicoCTF 2013 outlines some important ethics when hacking:

- Don't poke at things you don't have permission to break
- Let someone know if you find their systems might be in jeopardy
- Be responsible, and use good judgement

If you're not careful, that's when you might find yourself in jail.

tl;dr

Don't be stupid. Use common sense.

Contributing

Something wrong? Want to add something? Make a pull request here and we'll try to add your contributions to the book!

Introduction

If you have little to absolutely no programming experience, this "crash course" will be helpful. This is a course of the most basic terms/syntax and functions of Python (since our IDE, or Python Editor, which will be used for some problems is optimized for Python). Note that once you have learned the basics of programming you will find most languages are quite similar except for the unique syntax used. Most basic ideas (such as loops and creating functions and conditional statements), however, are the same across languages.

What can you do in Python? Lots of things! What may be useful for this competition is learning everything you need to write algorithms to solve problems and basic tools such as printing.

Some unique properties of Python is that it uses indentation instead of brackets (usually tabs or spaces, but don't mix the two), you don't have to declare variable types, and using semicolons are optional.

Variables

Variables store values that can be looked at or changed later. You do not need to need to declare variable types before assigning them. However, you must initialize them before you call them. Different types of variables include:

Integer

An integer is a number that is not a fraction.

Examples: `-1`, `2`, `105676`, etc.

To define an integer, use the following syntax:

```
myint = 2
```

Floating Point

A number that has decimal values.

Examples: `1.0`, `-2.73`, `0.66`

To define:

```
myfloat = 7.3
```

or

```
myfloat = float(7)
```

With integers and floating points, you can perform basic math operations on them such as `+ - / *` (with integers though, your answers will be rounded down to an integer).

String

Text surrounded by double or single quotes.

Examples: `"Hello"`, `'World'`

To define:

```
hello = "Hello"
world = "World"
```

You can also add strings together like this:

```
helloworld = hello + " " + world
```

This makes `helloworld` store the value `"Hello World"`

Or multiply a string with an integer to get a repeating string such as:

```
helloworld = "hello" * 6 + 3 * "world"
```

Which makes `helloworld` store the value `"hellohellohellohellohellohelloworldworldworld"`

PLEASE DO NOT SUBTRACT OR DIVIDE STRINGS

Character

A single symbol represented by a letter or number surrounded by single quotes.

Examples: `'A'`, `'4'`, `'k'`

To define:

```
mychar = 'h'
```

Single characters can be converted to their ASCII identifiers using the `ord()` function. Similarly, to convert an ASCII identifier back to a readable character, use the `chr()` function.

Functions

A function is a part of a program that can complete a certain task. For example, a function may be written that finds the average of three supplied numbers. Once written, it can be used many times without having to rewrite it over and over again.

Writing Functions

To define:

```
def func_name():
```

Any code inside the function should be indented until you want to exit the function.

You can put parameters - information passed to the function for use inside it - in a function as well. Here is a simple example:

```
def func_name(a,b,c):
    print(a+b)
    print(c)
```

`func_name(3,7,"Hello")` would print an output of:

```
10
"Hello"
```

Loops

Writing Loops

Loops are exactly what they sound like: pieces of code that runs in a pattern multiple times. Loops are useful for math (brute-force problems), reading data from a file (line-by-line), and much more.

There are two types of loops in Python: a `for` loop and a `while` loop.

For Loop

For loops begin with the keyword `for`. Its basic syntax looks like this:

```
for element in iterable:
```

Here's an example of some code that prints `Hello, EasyCTF!` 5 times:

```
for i in range(5):
    print "Hello, EasyCTF!"
```

And the output for this is exactly what you would think it is:

```
Hello, EasyCTF!
Hello, EasyCTF!
Hello, EasyCTF!
Hello, EasyCTF!
Hello, EasyCTF!
```

In this example, `i` is the element, and `range(5)` is an iterable that literally contains the numbers `0, 1, 2, 3, 4`. More about the `range()` function can be found on the Python documentation.

In another example of a for loop, we will compute and print the sum of all of the integers from 0 to 49,999.

```
sum = 0
for i in range(50000):
    sum += i
print "The sum is %d" % sum
```

And our output is:

```
The sum is 1249975000
```

In this example, we can see that the value of the variable `i` changes every iteration. Its value is determined by the position in the iterable object, the `range(50000)`. For example, in the first iteration, `i` would be 0, and in the second iteration, `i` would be 1.

Conditional Statements

If, Else If, Else

If, Else If, and Else are conditional statements, meaning that the condition set by the statement decides which statement will be executed.

The format is usually as follows:

```
if (condition1):
    # code to be executed
elif (condition2):
    # different code to be executed
# more elifs if needed
else:
    # another different set of code to be executed
```

For example:

```
for x in range(0,5):
    if (x == 0):
        print("hello")
    elif (x == 1 or x == 2):
        print("second or third")
    elif (x == 3):
        print("4")
    else:
        print("last value left to check")
```

Would print:

```
hello
second or third
second or third
4
last value left to check
```

Notice how all the terms aren't just printed 5 times each as in the example in the For-Loops section.

Linux

Linux is an Operating System (OS). An OS is a medium between the applications you use on your desktop and your physical machine and the hardware that is plugged into it. Other OS's that you may be familiar with may be Windows or Mac OS, or maybe a mobile one such as iOS or Android.

Linux comes in many distributions. Each distribution can be vastly different, but in EasyCTF we will be observing one of the more popular Linux distributions, Ubuntu. You can use our Ubuntu server in your browser or you can download your own copy for the full experience.

If you want to know more about any commands than we cover you can use the `man` command to view formatted online manual pages for the command.

```
man [command]
```

So for example:

```
man cat
```

Shell

Linux features an interface called a **shell**. In a shell, you may see a prompt symbol, `$` , followed by a blinking cursor `_`, in which you type commands and hit enter, and then the computer executes your commands and prints the output.

Before we had graphical interfaces, people typed commands into shells. Now you may be asking, "well, if we have these awesome graphical interfaces now, why still use shells?" The answer is that although you are seeing a graphical interface, all of the details of how your applications run can be manipulated through the shell. Also, most server computers only use a shell.

Web Shell

In EasyCTF, we have a web shell that you can log in to to solve problems and get some hands-on Linux experience. The shell page should contain your credentials for signing in.

- **Windows Users** - Use PuTTY to connect, using hostname `shell.easyctf.com` and port `22`
- **Mac and Linux Users** - Use the `ssh` command in your terminal to connect: `ssh yourusername@shell.easyctf.com 22`

File System

The file system in Linux is very similar to the one you are familiar with in Windows or in Mac. There is a hierarchy of folders and inside folders you can have files. We can also refer to folders as *directories*.

Special Directories

Root Directory

The root directory is `/`. This directory is at the top of the hierarchy of directories.

Home Directory

Your home directory is `~`. This directory may vary depending on your system, but in EasyCTF, your home directory will be `/home/youruser`, where `youruser` is your username.

Change Directory

To change a directory, type `cd` and the name of the directory you want to go to. Unless you start your path with `/`, this will always be relative. For example, examine the following system:

```
/
    home
        folder1
            folder2
                file3.txt
            file2.txt
        file1.txt
        (other folders and files)
    (other folders and files)
```

Suppose your current directory is `folder1`. You may see a prompt like this:

```
user@easyctf:~/folder1$ _
```

Moving Down

To move down from `folder1` to `folder2`, simply type `cd folder2`. That's because `folder2` is inside of `folder1` right now and you can refer to it using a relative path. If you were in your home folder, you would not be able to refer to `folder2`, because `folder2` doesn't exist inside `~`. All you can see from `~` is `folder1` and `file1.txt`.

```
user@easyctf:~/folder1$ cd folder2
user@easyctf:~/folder1/folder2$ _
```

Moving Up

Moving up is simple. In every directory (even empty ones), there are two hidden files: `.`, which refers to the current folder, and `..`, which refers to the parent folder. Basically, to move to the parent folder (like from `folder1` to `~`), just type `cd ..` and you will be in the parent folder.

```
user@easyctf:~/folder1$ cd ..
```

```
user@easyctf:~$ _
```

Linux Commands

This section will cover:

- `echo`
- `ls`
- `cat`
- `grep`
- `pwd`
- `cd`
- `rm`
- `mkdir`
- `mv`
- `cp`
- `sudo`
- `i/o redirecting`

echo

Echo displays strings to standard output. The strings can be encased with either single or double quotes.

For example,

```
echo "Hello, EasyCTF!"
```

Outputs the following:

```
Hello, EasyCTF!
```

You can also declare variables and echo their values (note: to substitute values such as `$x`, you must use double quotes and not single quotes):

```
x = 15
echo "The value of x = $x"
```

Outputs:

```
The value of x = 15
```

If you want to print a new line or a tab use `\n` and `\t` and add the *option* `-e` (which allows backslash interpretation) and enclose the string(s) you want to print with double quotes.

There are also certain characters that won't be automatically printed such as `\`, `"`, etc. If you need to print these, you must `escape` them first, which means you need to precede them with another backslash (which tells the shell to ignore the next character).

For example:

```
echo -e "this \\ is a backslash, \nthis \" is a double quote."
```

Outputs:

```
this \ is a backslash,
this " is a double quote.
```

ls

The command `ls` lists directory contents. A directory is a file that consists solely of a set of other files, like a folder.

There are several options you can add to this command such as `-a`, `-r`, `-R`, `-X`, and many more.

-a

This option will list all entries in a directory, even ones starting with `.`.

-r

This option lists entries in reverse alphabetical order.

-R

This option recursively lists the contents of all sub-directories inside this one.

-X

This option lists entries sorted alphabetically by entry extension

-l

This option gives you a more complete listing of all of the files along with owner and permsisions.

To use:

```
ls [option] [directory location]
```

If a directory location is not specified, it will use the current directory you are in.

For example:

```
ls -r /usr
```

Would output the files in `/usr` in reverse alphabetical order.

cat

The cat command is used for:

- Displaying a text file

  ```
  cat filename
  ```

  If you would like to display all files in the current directory, use the wildcard `*` like this:

  ```
  cat *
  ```

  To view all files or:

  ```
  cat *.txt
  ```

  To view all `.txt` files only.

- Reading a text file

  Sometimes a file will be too large to read since it won't fit on screen and will scroll by at high speed. To bypass this, use the `cat` command with `more` or `less` (however, on some shells the more command is not supported).

  ```
  cat bigfile | more
  ```

  or

  ```
  cat bigfile | less
  ```

- Creating a new text file

  To create a file named "test.txt":

  ```
  cat > test.txt
  ```

  Then, type the text you want to save into your file and press `Ctrl` + `D` when you are finished.

- File concatenation (adding files)

  To combine to files and create a new file called "newfile.txt":

  ```
  cat file1.txt file2.txt > newfile.txt
  ```

- Modifying files

  To add data to an existing file named "test.txt":

```
cat >> test.txt
```

Then type your text, and again, press `Ctrl` + `D` when finished.

grep

Grep is a command that searches files for certain words or phrases.

For example, to search the word "hello" in "test.txt":

```
grep "hello" test.txt
```

You can also search in multiple files, or use the wild card, the asterisk `*`, to search all files in a directory.

```
grep "hello" test1.txt test2.txt test3.txt
```

or

```
grep "hello" *
```

Just like with cat, there are several different options you can apply to grep to make it more or less specific.

-i

This option ignores cases in the word or phrase being searched. This means if the word was "key" it would search "KEY" and "keY" and any other case combination in the files specified.

```
grep -i "key" *
```

-l

This option lists only the file name that contains the word or phrase being searched.

```
grep -l "hello" *
```

-r

Similar to cat, this option recursively searches the word or phrase in all sub-directories.

```
grep -r "hello" *
```

pwd

pwd stands for "print working directory". The current working directory is the directory you are currently working in in your shell.

This command is used for:

- Finding the full path to the current directory
- Storing the full path of the current directory into a shell variable

To print the current directory:

```
pwd
```

To store the path into a variable:

```
var = $(pwd)
echo "The current working directory is $var"
```

cd

This command changes the shell's current working directory.

To use:

```
cd [directory]
```

For example:

```
cd documents/subfolder1/subfolder2
```

As you can see, directories are separated by a slash `/`.

The `parent`, or the directory above the current one (in the example above the current would be `subfolder2` after changing to it, and the parent would be `subfolder1`) is represented by `..`.

So if you start in `documents/subfolder1/subfolder2`,

```
cd documents/subfolder1
```

and

```
cd ..
```

are the same.

There is also a specially named directory that can be represented with a tilde `~`: the `home directory`. The `home directory` is the default directory that you're automatically placed into upon starting your shell.

This means that if your username is `testuser` and your home directory is `home` these two will be equivalent:

```
    cd home/testuser
    cd ~
```

Also, it should be noted that trailing slashes in directory paths are optional. Therefore, the following are the same:

```
    cd documents
    cd documents/
```

rm

This command is used to remove files or directories. By default, it does not remove directories but can be manipulated to do so with an option.

To use:

```
rm [option] [file path]
```

For example, if you are in `documents/hello` which contains the files `test1.txt` and `test2.txt`, you can remove the files like this:

```
rm test1.txt test2.txt
```

To delete the whole directory `hello` and the files it contains, change to parent directory and use `-r` option:

```
cd ..
rm -r hello
```

-i

This option makes the shell prompt you before each removal

-r

This option recursively deletes files and directories

PLEASE DO NOT USE `rm -rf` IF YOU DON'T KNOW WHAT YOU ARE DOING, IT WILL RECURSIVELY FORCE REMOVE FILES


rm -rfers gonna rm -rf

mkdir

This command does exactly what it sounds like - it makes directories.

For example, if the current directory is `documents`:

```
mkdir hello
```

Would create a directory at `documents/hello`

You can also use options with this command to do things such as changing permissions.

cp

This command copies files and directories.

To use:

```
cp [original file name] [copied file name]
```

If you are in the current working directory `pictures/album1` and you want to copy the file `photo.jpg`:

```
cp photo.jpg photo2.jpg
```

Will create a copy of `photo.jpg` named `photo2.jpg`

If the file you are trying to copy is in a different directory, say `documents/hello.txt` is the file you want to copy:

```
cp documents/hello.txt pictures/album1/hello-cp.txt
```

To copy multiple files, specify the files and have the last argument in the command be the directory the files will be copied into.

```
cp documents/test1.txt documents/test2.txt documents/test3.txt pictures/album1
```

mv

This command is used to move or rename files.

To rename files:

```
mv [option] [original file] [new file]
```

So if you are in the directory `documents` with the file `hello.txt`, to rename it to `newname.txt`:

```
mv hello.txt newname.txt
```

If you simply want to move without renaming (in this case move to the directory `music`:

```
mv hello.txt music/.
```

In this case, the period `.` means to place the file here without renaming it. You can also move and rename in the same step:

```
mv hello.txt music/newname.txt
```

-i

If you are trying to rename a file to an existing name, this option causes the shell to prompt you for confirmation before overriding.

sudo

The command `sudo` allows you to execute commands as the `root` user. The `root` user has privileges that a standard user does not, including access to files that standard users cannot view (like the `flag` files in the shell challenges).

`sudo` requires a password in order to be used, so you will not be allowed to use it to complete challenges, but should be aware of its existence.

Input/Output Redirection

Standard Output

By default, most command line programs display results to the `standard output`, which directs its contents to the display.

If you want to redirect output to a file, you use the angle bracket `>`.

```
ls > file.txt
```

Basically what this does, is that what is usually displayed with `ls` isn't printed to the screen, instead the results are redirected and stored in a file called `file.txt`. You can redirect outputs of other commands too, but we are just using `ls` as an example.

If you call the command again, the file will be overwritten. So, if you want to add the results onto the end of `file.txt` instead of replacing what was there before, use two brackets `>>`:

```
ls >> file.txt
```

Standard Input

By default, most command line programs accept input from the `standard input`, which is usually what you type in from the keyboard to a command.

To redirect input from a file instead of from the keyboard, you can use the angle bracket `<`.

```
sort < file1.txt
```

In this example, the `sort` command was used to sort the contents of `file1.txt`. The results of this were outputted on the display because the standard output was not redirected.

You can also combine redirecting outputs and inputs like this:

```
sort < file1.txt > sorted_file1.txt
```

It should be noted that the order you redirect the output and input doesn't matter.

Pipelines

You can also connect multiple commands together with `pipelines`, which makes the standard output of one command be the standard input of another.

For example:

```
ls | less
```

`less` is a command that allows you to scroll through a screen of output (in this case, the output from `ls`) at a time instead of just printing the whole thing into the terminal at high speed.

Resources

There's tons of resources on the internet nowadays, and we've compiled a handy list for you in case you want to learn more about anything. Know of a good resource that's not on here and should be? Email it to us at team@easyctf.com.

Programming in General

- Directory of Programming Resources
- Github List of Free Programming Books
- Another List of Free Programming Books
- Learn X in Y Minutes: Programming Tutorials
- Treehouse: Programming Tutorials
- Codecademy: Lessons in Python, JavaScript, HTML/CSS, PHP, and More
- CodingBat: Exercises for Python and Java - note: Java and JavaScript are *not* the same!
- Command Line Crash Course
- Python: Data Structures & Algorithms
- Wikipedia: Bitwise Operators
- Untrusted: A Beginner JavaScript Game

CTF Related

- Tips on Getting Started in CTF
- CTFTime: List of Upcoming CTFs & Write-Ups of Past CTFs
- Shell Storm CTF Repository
- Practice CTF List

Cryptography

- Ciphers & Codes: Web-Based Tools
- Overview of Cryptography
- Khanacademy: Cryptography Course

Exploitation

- Wikipedia: Exploitation
- Exploitation Exercises
- Wikipedia: Code Injection

Networking

- Introduction to Networking Terminology
- Introduction to Computer Networks

Reverse Engineering

- Reverse Engineering Brief Overview
- x86 Assembly and ARM Code for Beginners
- Reverse Engineering Course: Assembly & More

Steganography

- Wikipedia: Steganography Overview
- Hiding Data within Data Example