TTM4536 - Ethical Hacking - Information Security, Specialization Course

- Plan for 5 Sept 2017
- To investigate some of the Python specific modules
- Assumption: You have already installed your Python IDE (PyCharm for example)

Basic introduction to Python and some specific modules

- Crash course in Python
  - You need Python for all practical exercises and all project tasks
  - At the last lab exercises I noticed that many of the student would benefit from one-day crash course in Python
- **Introduction to Crypto module**
  - You will need that for some of the project tasks in connection with the CTF tasks
- Introduction to writing command-line applications in Python
  - For many penetration-testing tasks and many CTF tasks you will need command-line scripts and applications

Crash course in Python

- From “Learn Python in 10 minutes”
  - https://www.stavros.io/tutorials/python/

Properties

- Python is strongly typed (i.e. types are enforced),
  - Types are created dynamically,
  - Types are implicit (i.e. you don't have to declare variables),
- Everything is case sensitive (i.e. var and VAR are two different variables)
- Python is object-oriented (i.e. everything is an object).

Getting help

- Help in Python is always available right in the interpreter.
- If you want to know how an object works, all you have to do is call
  - `help(<object>)`
- Also useful are
  - `dir()` which shows you all the object's methods, and
  - `<object>.__doc__` which shows you its documentation string.

Getting help

- Start Python interpreter in the command line with `python` Then type:

```
>>> help(5)
```

You will get a long answer:

```
Help on int object:


class int(object)
 |   int(x=0) -> int or long
 |   int(x, base=10) -> int or long
 |
...
```

Press “q” to get out of the help text

Getting help

- Then type:

```
>>> dir(5)
```

You will get a list of all methods applicable to the object 5

- If you type:

```
>>> print abs.__doc__
```

You will get an explanation for the function i.e. method “abs”, of its arguments and what type of object it returns.

```
abs(number) -> number

Return the absolute value of the argument.
```

Syntax

- Python has no mandatory statement termination characters
- Blocks are specified by indentation.
  - Indent to begin a block,
  - dedent to end one.
- Statements that expect an indentation level end in a colon (:).
- Comments start with the pound (#) sign and are single-line
- Multi-line strings are used for multi-line comments.
- Values are assigned (in fact, objects are bound to names) with the equals sign ("="),
- Equality testing is done using two equals signs ("==").
- You can increment/decrement values using the += and -= operators respectively by the right-hand amount.
- This works on many datatypes, strings included.
- You can also use multiple variables on one line.

Syntax

```
>>> myvar = 3
>>> myvar += 2
>>> myvar
5
>>> myvar -= 1
>>> myvar
4
"""This is a multiline comment.
The following lines concatenate the two strings."""
>>> mystring = "Hello"
>>> mystring += " world."
>>> print mystring
Hello world.
# This swaps the variables in one line(!).
# It doesn't violate strong typing because values aren't
# actually being assigned, but new objects are bound to
# the old names.
>>> myvar, mystring = mystring, myvar
```

Data types

- The data structures available in python are
- lists, tuples, dictionaries and sets.
- Lists are like one-dimensional arrays (but you can also have lists of other lists),
- Dictionaries are associative arrays (a.k.a. hash tables)
- Tuples are immutable one-dimensional arrays
- Python "arrays" can be of any type, so you can mix e.g. integers, strings, etc in lists/dictionaries/tuples/sets.
- The index of the first item in all array types is 0.
- Negative numbers count from the end towards the beginning, -1 is the last item.
- Variables can point to functions.

Data types

```
>>> sample = [1, ["another", "list"], ("a", "tuple")]
>>> mylist = ["List item 1", 2, 3.14]
>>> mylist[0] = "List item 1 again" # We're changing the item.
>>> mylist[-1] = 3.21 # Here, we refer to the last item.
>>> mydict = {"Key 1": "Value 1", 2: 3, "pi": 3.14}
>>> mydict["pi"] = 3.15 # This is how you change dictionary values.
>>> mytuple = (1, 2, 3)
>>> myfunction = len
>>> print myfunction(mylist)
3
```

Data types

- You can access array ranges using a colon (:)
- Leaving the start index empty assumes the first item, leaving the end index assumes the last item.
- Negative indexes count from the last item backwards (thus -1 is the last item)

Data types

```
>>> mylist = ["List item 1", 2, 3.14]
>>> print mylist[:]
['List item 1', 2, 3.1400000000000001]
>>> print mylist[0:2]
['List item 1', 2]
>>> print mylist[-3:-1]
['List item 1', 2]
>>> print mylist[1:]
[2, 3.14]
# Adding a third parameter, "step" will have Python step in
# N item increments, rather than 1.
# E.g., this will return the first item, then go to the third and
# return that (so, items 0 and 2 in 0-indexing).
>>> print mylist[::2]
['List item 1', 3.14]
```

Strings

- Its strings can use either single or double quotation marks,
- You can have quotation marks of one kind inside a string that uses the other kind (i.e. "He said 'hello'." is valid).
- Multiline strings are enclosed in triple double (or single) quotes (""").
- Python supports Unicode out of the box, using the syntax u"This is a unicode string".
- To fill a string with values, you use the % (modulo) operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions

Strings

```
>>> class A:
...     pass
...
>>> a=A()
>>> print "Name: %s\
... Number: %s\
... String: %s" % (str(a.__class__), 3, 3 * "-")
Name: __main__.ANumber: 3String: ---

>>> strString = """This is
... a multiline
... string."""
>>> # WARNING: Whatch out for the trailing s in "%(key)s".
...
>>> print "This %(verb)s a %(noun)s." % {"noun": "test", "verb": "is"}
This is a test.
```

Flow control statements

- Flow control statements are
- `if`
- `for`
- `while`
- There is no `switch`; instead, use if.
- Use `for` to enumerate through members of a list.
- To obtain a list of numbers, use `range(<number>)`

Flow control statements

```
rangelist = range(10)
>>> print rangelist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing

if rangelist[1] == 2:
    print "The second item (lists are 0-based) is 2"
elif rangelist[1] == 3:
    print "The second item (lists are 0-based) is 3"
else:
    print "Dunno"

while rangelist[1] == 1:
    pass
```

Flow control statements

```
rangelist = range(10)
>>> print rangelist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing

if rangelist[1] == 2:
    print "The second item (lists are 0-based) is 2"
elif rangelist[1] == 3:
    print "The second item (lists are 0-based) is 3"
else:
    print "Dunno"

while rangelist[1] == 1:
    pass
```

Functions

- Functions are declared with the "def" keyword.
- Optional arguments are set in the function declaration after the mandatory arguments by being assigned a default value.
- For named arguments, the name of the argument is assigned a value.
- Functions can return a tuple (and using tuple unpacking you can effectively return multiple values).
- Lambda functions are ad hoc functions that are comprised of a single statement.
- Parameters are passed by reference, but immutable types (tuples, ints, strings, etc) *cannot be changed*. This is because only the memory location of the item is passed, and binding another object to a variable discards the old one, so immutable types are replaced.

Functions

```
>>> def funcvar(x): return x + 1
...
>>> print funcvar(1)
2
>>> funcvar = lambda x: x + 1
>>> print funcvar(1)
2
>>> # an_int and a_string are optional, they have default values
... # if one is not passed (2 and "A default string", respectively).
... def passing_example(a_list, an_int=2, a_string="A default string"):
...     a_list.append("A new item")
...     an_int = 4
...     return a_list, an_int, a_string
...
>>> my_list = [1, 2, 3]
>>> my_int = 10
>>> print passing_example(my_list, my_int)
([1, 2, 3, 'A new item'], 4, 'A default string')
>>> my_list
[1, 2, 3, 'A new item']
>>> my_int
10
```

Classes

- Python supports a limited form of multiple inheritance in classes.
- Private variables and methods can be declared (by convention, this is not enforced by the language) by adding at least two leading underscores and at most one trailing one (e.g. "`__spam`").
- We can also bind arbitrary names to class instances.

Classes

```
>>> class MyClass(object):
...     common = 10
...     def __init__(self):
...         self.myvariable = 3
...     def myfunction(self, arg1, arg2):
...         return self.myvariable
...
>>> # This is the class instantiation
...
>>> classinstance = MyClass()
>>> classinstance.myfunction(1, 2)
3
>>> # This variable is shared by all instances.
...
>>> classinstance2 = MyClass()
>>> classinstance.common
10
>>> classinstance2.common
10
>>> # Note how we use the class name
... # instead of the instance.
...
```

Classes

```
>>> MyClass.common = 30
>>> classinstance.common
30
>>> classinstance2.common
30
>>> # This will not update the variable on the class,
... # instead it will bind a new object to the old
... # variable name.
...
>>> classinstance.common = 10
>>> classinstance.common
10
>>> classinstance2.common
30
>>> MyClass.common = 50
>>> # This has not changed, because "common" is
... # now an instance variable.
...
>>> classinstance.common
10
>>> classinstance2.common
50
```

Classes

```
>>> # This class inherits from MyClass. The example
... # class above inherits from "object", which makes
... # it what's called a "new-style class".
... # Multiple inheritance is declared as:
... # class OtherClass(MyClass1, MyClass2, MyClassN)
...
>>> class OtherClass(MyClass):
...     # The "self" argument is passed automatically
...     # and refers to the class instance, so you can set
...     # instance variables as above, but from inside the class.
...     def __init__(self, arg1):
...         self.myvariable = 3
...         print arg1
...
>>> classinstance = OtherClass("hello")
hello
>>> classinstance.myfunction(1, 2)
3
>>> # This class doesn't have a .test member, but
... # we can add one to the instance anyway. Note
... # that this will only be a member of classinstance.
...
>>> classinstance.test = 10
>>> classinstance.test
10
```

Exceptions

- Exceptions in Python are handled with try-except [exceptionname] blocks:

Exceptions

```
>>> def some_function():
...     try:
...         # Division by zero raises an exception
...         10 / 0
...     except ZeroDivisionError:
...         print "Oops, invalid."
...     else:
...         # Exception didn't occur, we're good.
...         pass
...     finally:
...         # This is executed after the code block is run
...         # and all exceptions have been handled, even
...         # if a new exception is raised while handling.
...         print "We're done with that."
...
>>> some_function()
Oops, invalid.
We're done with that.
```

Importing

- External libraries (modules) are used with the
  `import [libname]`
- You can also use `from [libname] import [funcname]` for individual functions.

Importing

```
>>> import random
>>> from time import clock
>>> randomint = random.randint(1, 100)
>>> print randomint
16
```

File I/O

- Python has a wide array of libraries built in.
- As an example, here is how serializing (converting data structures to strings using the `pickle` library) with file I/O is used

File I/O

```
>>> import pickle
>>> mylist = ["This", "is", 4, 13327]
>>> # Open the file binary.dat for writing. The letter r before the
... # filename string is used to prevent backslash escaping.
... myfile = open(r"binary.dat", "w")
>>> pickle.dump(mylist, myfile)
>>> myfile.close()
>>> myfile = open(r"text.txt", "w")
>>> myfile.write("This is a sample string")
>>> myfile.close()
>>> myfile = open(r"text.txt")
>>> print myfile.read()
This is a sample string
>>> myfile.close()
>>> # Open the file for reading.
...
>>> myfile = open(r"binary.dat")
>>> loadedlist = pickle.load(myfile)
>>> myfile.close()
>>> print loadedlist
['This', 'is', 4, 13327]
```

File I/O

```
>>> import pickle
>>> mylist = ["This", "is", 4, 13327]
>>> # Open the file binary.dat for writing. The letter r before the
... # filename string is used to prevent backslash escaping.
... myfile = open(r"binary.dat", "w")
>>> pickle.dump(mylist, myfile)
>>> myfile.close()
>>> myfile = open(r"text.txt", "w")
>>> myfile.write("This is a sample string")
>>> myfile.close()
>>> myfile = open(r"text.txt")
>>> print myfile.read()
This is a sample string
>>> myfile.close()
>>> # Open the file for reading.
...
>>> myfile = open(r"binary.dat")
>>> loadedlist = pickle.load(myfile)
>>> myfile.close()
>>> print loadedlist
['This', 'is', 4, 13327]
```

Task:
Check the hexadecimal content
of the files “binary.dat” and “text.txt”
in another terminal.

Miscellaneous

- Conditions can be chained. `1 < a < 3` checks that a is both less than 3 and greater than 1.
- You can use `del` to delete variables or items in arrays.
- List comprehensions provide a powerful way to create and manipulate lists. They consist of an expression followed by a `for` clause followed by zero or more `if` or `for` clauses

Miscellaneous

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [3, 4, 5]
>>> print [x * y for x in lst1 for y in lst2]
[3, 4, 5, 6, 8, 10, 9, 12, 15]
>>> print [x for x in lst1 if 4 > x > 1]
[2, 3]
>>> # Check if a condition is true for any items.
... # "any" returns true if any item in the list is true.
...
>>> any([i % 3 for i in [3, 3, 4, 4, 3]])
True
>>> # This is because 4 % 3 = 1, and 1 is true, so any()
... # returns True.
...
>>> # Check for how many items a condition is true.
...
>>> sum(1 for i in [3, 3, 4, 4, 3] if i == 4)
2
>>> del lst1[0]
>>> print lst1
[2, 3]
>>> del lst1
>>> print lst1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lst1' is not defined
```

Miscellaneous

- Global variables are declared outside of functions and can be read without any special declarations, but if you want to write to them you must declare them at the beginning of the function with the "`global`" keyword,
- otherwise Python will bind that object to a new local variable (be careful of that, it's a small catch that can get you if you don't know it).

Miscellaneous

```
>>> number = 5
>>> def myfunc():
...     # This will print 5.
...     print number
...
>>> myfunc()
5
>>> def anotherfunc():
...     # This raises an exception because the variable has not
...     # been bound before printing. Python knows that it an
...     # object will be bound to it later and creates a new, local
...     # object instead of accessing the global one.
...     print number
...     number = 3
...
>>> anotherfunc()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in anotherfunc
UnboundLocalError: local variable 'number' referenced before assignment
>>> def yetanotherfunc():
...     global number
...     # This will correctly change the global.
...     number = 3
...
>>> yetanotherfunc()
>>> myfunc()
3
```

Basic introduction to Python and some specific modules

- Crash course in Python
  - You need Python for all practical exercises and all project tasks
  - At the last lab exercises I noticed that many of the student would benefit from one-day crash course in Python
- **Introduction to Crypto module**
  - **You will need that for some of the project tasks in connection with the CTF tasks**
- Introduction to writing command-line applications in Python
  - For many penetration-testing tasks and many CTF tasks you will need command-line scripts and applications

Introduction to Crypto module

- From “Laurent Luce's Blog”
  - http://www.laurentluce.com/posts/python-and-cryptography-with-pycrypto/

Hash functions

- A hash function takes a string and produces a fixed-length string based on the input. The output string is called the hash value. Ideal hash functions obey the following:
  - It should be very difficult to guess the input string based on the output string.
  - It should be very difficult to find 2 different input strings having the same hash output.
  - It should be very difficult to modify the input string without modifying the output hash value.
- Hash functions can be used to calculate the checksum of some data.
- They are used in digital signatures and authentication.
- We will see some applications in details.
- It is important to know that a hash function like MD5 is vulnerable to collision attacks

Hash functions

```
>>> from Crypto.Hash import SHA256
>>> SHA256.new('abc').hexdigest()
'ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
```

Hash functions

- Hash functions can be used in password management and storage.
- Web sites usually store the hash of a password and not the password itself so only the user knows the real password.
- When the user logs in, the hash of the password input is generated and compared to the hash value stored in the database. If it matches, the user is granted access.
- The code looks like this:

Hash functions

```
>>> from Crypto.Hash import SHA256
>>> def check_password(clear_password, password_hash):
...     return SHA256.new(clear_password).hexdigest() == password_hash
...
>>> check_password('abc','ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad')
True
>>> check_password('abC','ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad')
False
```

Hash functions

```
>>> import hashlib
>>> dir(hashlib)
['__all__', '__builtins__', '__doc__', '__file__', '__get_builtin_constructor', '__name__',
'__package__', '_hashlib', '_trans_36', '_trans_5C', 'algorithms', 'algorithms_available',
'algorithms_guaranteed', 'binascii', 'md5', 'new', 'pbkdf2_hmac', 'sha1', 'sha224', 'sha256',
'sha384', 'sha512', 'struct']
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16L
>>> m.block_size
64L
```

Key derivation

- Key derivation and key stretching algorithms are designed for secure password hashing.
- Naive algorithms such as sha1(password) are not resistant against brute-force attacks.
- A good password hashing function must be tunable, slow, and include a salt.

Key derivation

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

Encryption algorithms

- Encryption algorithms take some text as input and produce ciphertext using a key.
- You have 2 types of ciphers: block and stream.
- Block ciphers work on blocks of a fixed size (8 or 16 bytes).
- Stream ciphers work byte-by-byte.
- Knowing the key, you can decrypt the ciphertext.
- It is easy to write code to encrypt and decrypt a file using pycrypto ciphers.

Block ciphers

```
>>> from Crypto.Cipher import DES
>>> des = DES.new('01234567', DES.MODE_ECB)
>>> text = 'abcdefgh'
>>> cipher_text = des.encrypt(text)
>>> cipher_text
'\xec\xc2\x9e\xd9] a\xd0'
>>> des.decrypt(cipher_text)
'abcdefgh'


>>> from Crypto.Cipher import DES
>>> from Crypto import Random
>>> iv = Random.get_random_bytes(8)
>>> des1 = DES.new('01234567', DES.MODE_CFB, iv)
>>> des2 = DES.new('01234567', DES.MODE_CFB, iv)
>>> text = 'abcdefghijklmnop'
>>> cipher_text = des1.encrypt(text)
>>> cipher_text
"?\\\x8e\x86\xeb\xab\x8b\x97'\xa1W\xde\x89!\xc3d"
>>> des2.decrypt(cipher_text)
'abcdefghijklmnop'
```

Block ciphers

```
import os
from Crypto.Cipher import DES3

def encrypt_file(in_filename, out_filename, chunk_size, key, iv):
    des3 = DES3.new(key, DES3.MODE_CFB, iv)

    with open(in_filename, 'r') as in_file:
        with open(out_filename, 'w') as out_file:
            while True:
                chunk = in_file.read(chunk_size)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += ' ' * (16 - len(chunk) % 16)
                out_file.write(des3.encrypt(chunk))

def decrypt_file(in_filename, out_filename, chunk_size, key, iv):
    des3 = DES3.new(key, DES3.MODE_CFB, iv)

    with open(in_filename, 'r') as in_file:
        with open(out_filename, 'w') as out_file:
            while True:
                chunk = in_file.read(chunk_size)
                if len(chunk) == 0:
                    break
                out_file.write(des3.decrypt(chunk))
```

```
from Crypto import Random
iv = Random.get_random_bytes(8)
with open('to_enc.txt', 'r') as f:
    print 'to_enc.txt: %s' % f.read()
encrypt_file('to_enc.txt', 'to_enc.enc', 8192, key, iv)
with open('to_enc.enc', 'r') as f:
    print 'to_enc.enc: %s' % f.read()
decrypt_file('to_enc.enc', 'to_enc.dec', 8192, key, iv)
with open('to_enc.dec', 'r') as f:
    print 'to_enc.dec: %s' % f.read()


The output of this script:

to_enc.txt: this content needs to be encrypted.

to_enc.enc: ??~?E??.??]!=)??"t?
             JpDw? ??R?UN0?=??R?UN0?}0r?FV9

to_enc.dec: this content needs to be encrypted.
```

Block ciphers

```
>>> from Crypto import Random
>>> from Crypto.Cipher import AES
>>> iv = "\x8C\xAE\x65\x24\xA8\x63\xE3\x0F\x9B\x9D\x8D\xA2\xED\x05\xAA\x48"
>>> ciphertext =
"\x16\xD0\x7A\x30\x8E\x24\xED\xF8\xE7\x71\x57\x03\xC5\x74\xB6\xE3\x26\x40\x56\xE7\xE9\x56\xCF\x76\
x61\xBD\x72\xE3\xC7\xFC\x6C\x15\x27\x3D\x2A\xED\xA6\xB6\xEA\x04\xF1\xCC\xFE\xF6\x77\xB4\x41\x66"
>>>
>>> def decrypt(key):
...     cipher = AES.new(key, AES.MODE_CBC, iv)
...     result = cipher.decrypt(ciphertext)
...     return result
...
>>> def brute():
...     for i in range(256):
...         for j in range(256):
...             key = "".join([chr(j), chr(i)]).ljust(16, "\x00")
...             result = decrypt(key)
...             if "flag" in result:
...                 print(result)
...                 print("key :", key.encode("hex"))
...                 return
...
>>> brute()
the flag is e3565503fb4be929a214a9e719830d4e
('key :', 'a288000000000000000000000000000000')
```

Public-key algorithms

- With public-key algorithms, there are two different keys: one to encrypt (the public one) and one to decrypt (the private one).
- You only need to share the public key and only you can decrypt the message with your private decryption key.
- You can also produce digital signatures with your private key

Public/private key pair generation

```
>>> from Crypto.PublicKey import RSA
>>> dir (RSA)
['DerNull', 'DerObject', 'DerSequence', 'RSAImplementation', 'Random', '_RSA', '_RSAobj',
'__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__revision__',
'_fastmath', '_impl', '_slowmath', 'algorithmIdentifier', 'b', 'bchr', 'binascii', 'bord', 'bstr',
'bytes_to_long', 'construct', 'error', 'generate', 'getRandomRange', 'importKey', 'inverse',
'long_to_bytes', 'pubkey', 'struct', 'sys', 'tobytes']
>>> from Crypto import Random
>>> random_generator = Random.new().read
>>> key = RSA.generate(1024, random_generator)
>>> key
<_RSAobj @0x101d935f0 n(1024),e,d,p,q,u,private>
>>> key = RSA.generate(2048, random_generator)
>>> key
<_RSAobj @0x101d91c20 n(2048),e,d,p,q,u,private>
>>> key = RSA.generate(4096, random_generator)
>>> key
<_RSAobj @0x101d91b00 n(4096),e,d,p,q,u,private>
>>> key.can_encrypt()
True
>>> key.can_sign()
True
>>> key.has_private()
True
```

Encrypt/decrypt

```
>>> public_key = key.publickey()
>>> enc_data = public_key.encrypt('abcdefgh', 32)
>>> enc_data
('-v\xbd\x85\x96\xebKn\xb2\x92$
\xd9m\x8c\xe3\rv\xda\x81\xcd8\x8b\xd5M\x88\x04\x0f\x9f\xd7\x02\xf8\xb7\x84Sc\xb53&S\x9c\xa7\xef\xb
5\xef\xde\x1f\x0e\xc1~0\x1a\x01^+ !
=\xab*\xce\x8ef\x1f\x9fw&\x95\xb71\x85]"\xfa\x01\x03\xe6&D\xa3\x8b\x86c\x90\xd4u\xbbv?
\xf3w\x9fV\x82\xdf\xea\xcfZI\x8cE\xe1\xdcur\xacK\x92!.
\x07\x12v\xbf\x9f\xaa\x19\xd1]O\xfa\xee\xaf~\x0bv\xbe\xb0/\xe07a\x1bd\xb8\x95\xc5vp\x8aP\xe0\xdcf\
xc7V\x0e\xe5\x1df;*L\x83\xed\xc8Z3D\xa2j\xc8\xc8|\x10\x00\x95\x0fp\x12\x16\x14\xda\xb5q\xd7+
\x8d\x14\xcf\xf8I\xb3\xdf\x02\x11\xd0J\x8e\xb1n\xb0\xe4\x06\xf6\x98\xc7\xb5?
a0U&\x96\x831\xe8\xa2\xc8\x9d=\xb9\x9f3\xed~9\xa9\x12;\x99\xa6\x9c\xbd\n\xa3(\x97\xde\x8eO\xfe\x96
w\xcdt\xd8\xefN!\xfchR\xd8\xaa\xfe\x86F\xbb\x9c\xa0zY\xe0u\xc9\x80\'e:
\x8f\x8e^\xb6\xf0M9\x97(#\xa5ha\x1a\xc3n\x01\xb4\x8e\xa7\xcdC,
\r\xd1\xbdA\x8aN\x01\xe5~\xd0\xc0\xc3\x0bh\x98\xe2\x03\t7%j\x03\xaf<\xc5\x1e>\xc6
KZ7\xbc\x94\xb2<\x19\x84\xd1\x99\x17\xf8\xfe\x03\r\xed\x96\xe4*\xd1\xd4?
ko\x96\xf1\xf7\xa1\xe1\x08\xcd\xb9\x84\\
\x03i\x87C\x85\x82\x9e\xa3\x19i\xd3\xa9\x85\xbc[\xdd\x8dI^Vc%\xbd\xe9\x1a\x83\xea\
\3\x0f\xee\xb51\xc5\x90\xc7\x15\xb7\xa3ed[\xdb\x0fSR.
\xdd{l\xf5\xce\xd3qR\xfbj\xb3\xca\xbc\xc3\x8f\x1a\xe1y\xbb\x87\xc7\xcd\xd0\x14\x17\x93\xe3\xef\x1b
\x0b\x1bW\xa0\x12\x85-
\x8f6\x13#jD\xaf\x86\x11(\xfa$_\xa9\x9e\x18\xabM\xc7\xb2H\xc2\xc6\xc4\x83\xccs\x1e\xbd\xb5\xe0W7\x
07\xf0\xd5\xe5Y\x17\xe1A\xd1\
\i\x0e\x17\x11\xa4.e\xf7i\x937\xb2\x8e\x10T\xfb\xbd\x94\xff\xa3J\xc1\xd6\xdd\xc6\xa1E\xaa\xb4\x86\
xe5.E\x97\xfb\x12\xa0\x80DU',)
>>> key.decrypt(enc_data)
'abcdefgh'
```

Sign/verify

```
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>> from Crypto import Random
>>> key = RSA.generate(1024, random_generator)
>>> text = 'abcdefgh'
>>> hash = SHA256.new(text).digest()
>>> hash
'\x9cV\xccQ\xb3t\xc3\xba\x18\x92\x10\xd5\xb6\xd4\xbfWy\r5\x1c\x96\xc4|
\x02\x19\x0e\xcf\x1eC\x065\xab'
>>> # Sign
...
>>> signature = key.sign(hash, '')
>>> signature
(835809795863616393582502629625724953903002401580354845912707052958625470704708163494639105574356266518254245341847600916229902489645580360358178905246616901449840929417200175493904464813786669544539126885030924161472830474473556544757973169214246532660310042937516477702480030918932601800643080092787619459 06L,)
>>> # Verify
...
>>> text = 'abcdefgh'
>>> hash = SHA256.new(text).digest()
>>> key.verify(hash, signature)
True
```

Basic introduction to Python and some specific modules

- Crash course in Python
  - You need Python for all practical exercises and all project tasks
  - At the last lab exercises I noticed that many of the student would benefit from one-day crash course in Python
- Introduction to Crypto module
  - You will need that for some of the project tasks in connection with the second CTF task
- **Introduction to writing command-line applications in Python**
  - **For many penetration-testing tasks and many CTF tasks you will need command-line scripts and applications**

Introduction to writing command-line applications in Python

- From “A short introduction to writing command-line applications in Python” by Jacek Artymiak
  - http://www.linuxjournal.com/article/3946?page=0,0

Introduction to writing command-line applications in Python

- Creating scripts can be done using your favorite text editor as long as it saves text in plain ASCII format and does not automatically insert line breaks when the line is longer than the width of the editor's window.

- Always begin your scripts with either
- `#! /usr/local/bin/python`
- or
- `#! /usr/bin/python`
- Be sure this line is truly the first line in your script, not just the first non-blank line—it will save you a lot of frustration.
- Use `chmod` to set the file permissions on your script to make it executable. If the script is for you alone, type

```
chmod 0700 scriptfilename.py
```

Listing 1

```
#! /usr/local/bin/python
import sys
if '-h' in sys.argv or '--help' in sys.argv or '--help-display' in sys.argv:
    print '''
help.py--does nothing useful (yet)
options: -h, -help, or --help-display this help
Copyright (c) Jacek Artymiak, 2000 '''
    sys.exit(0)
else:
    print 'I don't recognize this option'
    sys.exit(0)
```

Listing 1

- Copy and save this script as help.py,
- Make it executable with the `chmod 0755 help.py` command,
- Run it several times, specifying different options, both recognized by the handler and not
- Note that we need to import the `sys` module before we can check the contents of the `argv` list and before we can call the `exit` function.
- The `sys.exit` statement is a safety feature which prevents further program execution when one of the help options is found inside the `argv` list.
- This ensures that users don't do something dangerous before reading the help messages (for which they wouldn't have a need otherwise).

Listing 2

- The simple help option handler described in Listing 1 works quite well and you can duplicate and change it to recognize additional options, but that is not the most efficient way to recognize multiple options with or without arguments.
- The “proper” way to do it is to use the `getopt` module, which converts options and arguments into a nice list of tuples.
- Listing 2 shows how it works.
- Copy this script, save it as options.py and make it executable.
- As you can see, it uses two modules: `sys` and `getopt` which are imported right at the beginning.
- Then we define a simple function that displays the help message whenever something goes wrong.

Listing 2, Option Handler, options.py

```
#! /usr/local/bin/python
import sys, getopt, string
def help_message():
    print '''options.py -- uses getopt to recognize options
Options: -h      -- displays this help message
       -a      -- expects an argument
       --file= -- expects an argument
       --view  -- doesn't necessarily expect an argument
       --version -- displays Python version'''
    sys.exit(0)
try:
    options, xarguments = getopt.getopt(sys.argv[1:],
    'ha', ['file=', '--view', 'version'])
except getopt.error:
    print 'Error: You tried to use an unknown option or the
    argument for an option that requires it was missing. Try
    `options.py -h\' for more information.'
        sys.exit(0)
for a in options[:]:
    if a[0] == '-h':
        help_message()
for a in options[:]:
    if a[0] == '-a' and a[1] != '':
        print a[0]+' = '+a[1]
        options.remove(a)
        break
    elif a[0] == '-a' and a[1] == '':
        print '-a expects an argument'
        sys.exit(0)
for a in options[:]:
    if a[0] == '--file' and a[1] != '':
        print a[0]+' = '+a[1]
        options.remove(a)
        break
    elif a[0] == '--file' and a[1] == '':
        print '--file expects an argument'
        sys.exit(0)
for a in options[:]:
    if a[0] == '--view' and a[1] != '':
        print a[0]+' = '+a[1]
        options.remove(a)
        break
    elif a[0] == '--view' and a[1] == '':
        print '--view doesn\'t necessarily expects an
argument...'
        options.remove(a)
        sys.exit(0)
for a in options[:]:
    if a[0] == '--version':
        print 'options version 0.0.001'
        sys.exit(0)
for a in options[:]:
    if a[0] == '--python-version':
        print 'Python '+sys.version
        sys.exit(0)
```

Introduction to writing command-line applications in Python

- Finish all 17 exercises from “A short introduction to writing command-line applications in Python” by Jacek Artymiak
  - http://www.linuxjournal.com/article/3946?page=0,0