Practical Data Security

Kyle Isom

Table of contents

Acknowledgements

This book almost certainly would not have been written were it not for the Pragmatic Programmers, LLC, and their PragProWriMo event (http://pragprog.com/magazines/2011-10/up-front) . Although I've had the idea to do this for a while, the PragProWriMo was the spark that ignited the fuel.

I'd also like to thank the people who reviewed this book: Beau and Essie Holton, Matthew Sowers, and Clark Lester.

CHAPTER

Introduction

This is a book about how to use data security practically. It's written assuming that you are a coder who's already writing cool and audacious projects, but you don't have a background in writing secure systems. I also assume that you have a passing familiarity with UNIX-based operating systems; while I don't expect you to be a UNIX wizard, I also expect you to not be terribly daunted when tinkering about the command line. I try not to get in your way and merely to facilitate a discussion in most cases about ways we as coders can write better applications for our users.

The news headlines are full of stories about popular online services being com-promised. It's becoming something that we are more used to living with. I think that we can do better. My particular interest in this problem started with two events that coincided: the first involved writing software for work to build encrypted updates for a project I was working on. I was doing this in Python, and so became fairly familiar with the PyCrypto library (https://www.dlitz.net/software/pycrypto/) . At the same time, a friend of mine was working on writing authentication code, also in Python. He was trying to use the PyCrypto library as well, but really didn't understand how cryptography worked and how to apply it in this case. That inspired me to write a ten page introduction to cryptography (also illustrated with Python). I've been mulling over the idea of expanding that into a book on data security, and now the time has come to write it.

From my perspective, a lot of the problem stems from a fundamental misunderstanding of what security is, and how to integrate it. I will try to address that issue here, and hopefully help other developers to understand not only why it's important, but to see that it's not just snake oil to be thrown into the mix so that it "just works."

The book is subtitled Illustrated With Python, which I am slightly worried will discourage those not familiar with the Python language. I chose to use Python because it is both a language I am well familiar with, and because it is a very readable language. Readers should note that the emphasis in the book is not on the Python way to do things; readers who are familiar with Python will note that I shy away from idiomatic Python that will be difficult for those unfamiliar with the language to understand.

CHAPTER 0

Introduction to Data Security

Data security is one of those things you hear a lot about, but most of us are busy getting things done; unfortunately for our users, data security tends to be put on the back-burner, if we remember it at all. Data security doesn't have to be a burden on the developer, and most importantly, it's in the best interest of users. One principle I particularly like for software engineering is the Principle of Least Astonishment.

THE PRINCIPLE OF LEAST ASTONISHMENT:
A program's behaviour should not surprise the end user.

How does this relate to data security? A user never expects their data to be leaked out of the application or their information to be shared with people they dont intend for it to. This might be an explicit understanding or an implicit one, especially when users don't really understand the problem domain. If your users are from a wide range of backgrounds, they may not have a solid understanding of the basics of how to keep their information safe. In fact, they may not even realise what they don't know. So, it becomes even more important for developers to ensure that, as much as possible, their data is safe. What I will discuss in this book are ways to do that, while keeping in mind the misconception (not always true) that the more secure a system is, the less accessible it is. I'd like to address that misconception, and help you to make your applications as transparently secure as possible.

The first thing we'll take a look at are some of the objectives of information security that are most generally applicable to software, i.e. those that we can address. Then we'll take a high-level walk through some basic UNIX security that's applicable (and perhaps on the Windows platform as well). Even if your application is a web application, it's probably running on a UNIX machine of some sort, and that represents an area that you should pay attention to as well. If you're a Windows developer, there will still be some information applicable, but you can probably skim through this part. Then we'll take a high level look at cryptography; I've chosen to stay away from deep mathematical discussions and only discuss those aspects you need to understand as a coder to effectively make use of it. Following that, we'll see a few scenarios where your code is vulnerable. The rest of this chapter will be a gentle introduction to each of those areas, and the rest of the book will flesh it out with code examples.
There are four information security principles you should keep in mind when writing software:

1. Privacy
2. Anonymity
3. Repudiation
4. Integrity

Let's take a look at these in a little more depth, and see how they apply to the software we're writing.

First, we visit the concepts of **authentication** and **authorisation**. Authentication is confirming the identity of a user--most often by way of a username (the user's identity) and password (the secret used to authenticate). Other ways of authentication include using smart cards, where the identity is given by the cryptographic key on the card and the authentication is done by entering a PIN to unlock the key, or perhaps using something like a one-time password token tied to an OpenID. Once the user's identity has been confirmed (the user is now **authenticated**), their access to resources is controlled by **authorisation**. Authorisation is the check to see "is this user allowed to do this?" If we look at this from the perspec-

tive of a user's ability to access files on a computer, they are typically authenticated by their login and password (or, on some laptops, via a login and fingerprint). When a user tries to read or write to a file, the system checks the access control list (an authorisation scheme) to see if the user is authorised to do this. Let's look at the Twitter application on a mobile as another example: when you try to use the app, you have to sign into twitter and authorise access on Twitter's website for the application. Your username and password authenticate you to Twitter, and then you choose to authorise the application's access to the site. Authentication and authorisation may be done by other means as well for entities other than users. We'll call any entity a user in this book, but that user might be a person, a computer, or something else entirely. With that dead horse beaten, we can move on.

The principle of **privacy** states that information may only be viewed by those authorised to view it. In terms of UNIX files, a user might be able to log into a computer, but does not have access to read another user's files. On twitter, you can protect your timeline, so that only your friends can read your tweet. It's not unreasonable for a user to expect they can control who else has access to their resources; obviously a user with a public blog expects anyone to be able to read their posts, but they may not expect just anyone to see other details about that user. When you're designing an application, it helps to think about which pieces of information a user should have control over. Some applications will require fine-grained control--say it's designed so user A can choose to share one post only with group B and another post only with group C. Others will be course-grained--users in group B will always have access to posts by user A.

Anonymity is a hot topic these days with the so-called "nym wars" on Google+ as but one example. This boils down to the idea that, depending on your stance, a user should be able to separate their different identities from each other. In the Google+ debate, it's the idea that users want to be able to separate their real-world identity from their electronic identity. That is, my identity on Google+ may not be the same identity I want to use on my blog. This brings out another point about security: most security engineering isn't just a technical question; it's a social question. Many of our applications aren't just technical applications, but have social value as well; many security decisions are made by weighing social considerations. Your decision in regards to anonymity is going to be a social decision, but it should be decided while designing the application. Bolt-on security (added after the design phase) may be somewhat effective, but it tends to cause a lot of problems.

Repudiation is a fancy word that means the ability to deny ownership of information. You'll hear this term a lot in discussions about cryptography, particularly with digital signatures. If a user's data is **repudiable**, the user can plausibly deny owning that data (for example, user A might want to be able to deny having posted something unpopular about a sensitive topic). Conversely, if a user's information is **non-repudiable**, she cannot plausibly deny owning that information. I'll have more to say on this in a bit, but for now you know what I'm talking about when I bandy about the term like "non-repudiable."

Users also want some degree of assurance that their data won't be changed or tampered with. This is the principle of **data integrity**--information may only be changed by people authorised to do so. This isn't just a security issue; natural disasters and junior sysadmins (some would say naming both is a redundancy) have a clever way of accidentally modifying your data without the user's authorisation. Of course, we expect (and probably constantly tell) users to backup their data, but your application may not be easily backed up by the user. There's also a case to be made for **transparency** in both showing users what data you have of theirs on your systems or in your application and allowing them to download or backup that information easily. Even data like photos, which are easy to back up, may come from a disparate number of sources, should be considered. For example, a user might upload photos from his phone, his lap-

top, and his tablet. All these different sources may make backing up more difficult, although far from impossible, and having one place to back them all up from may be useful.

These basic principles are all properties to consider when designing your system; they fit alongside all the other software engineering principles like modularity that should be considered. It is far easier to design these in at the start than to try to bolt them on later.

Another thing to think about is the use case for your software; namely, you might not be able to think of all the ways users will use the system. Of course, we'd like to think of our users as living in an ideal world where using real names isn't an issue and no one has anything to hide, but this is not the world we live in. You should also consider some of the challenges faced by users living in non-ideal places. If you're allowing people to upload data or users must download information to synchronise, consider that many users don't have high-speed internet or a large bandwidth cap. In many countries, power is at a premium, and users shut down their computers when they aren't using them out of economic necessity, instead of being able to leave them on. Users might also be living in restrictive regions where posting information critical of other people might be dangerous for them. If you have any desire for your software to have a global impact, you should consider them. If your application isn't region-specific, you should most certainly keep this in mind when designing your software.

There are a couple areas you can start to actually design security into your code. First, there are the fundamental security mechanisms available on the platform you are coding on. There are coding errors that can lead to flaws, such as the well-known SQL injections and buffer overflows that are some of the mainstays of many successful attacks. There are mechanisms to secure data in transit (when the data is being accessed and used) and at rest (when the data is not being used). As coding errors have been dealt with quite effectively in many other books (see the bibliography for more), we'll focus on the fundamental security mechanisms and other data security mechanisms, and in particular, how we can use basic UNIX security mechanisms and cryptography effectively. We'll also consider some real world ways in which failing to secure your code can lead to an exploit.

CHAPTER 1

Practical Introduction to Cryptography

1.1 Introduction

This chapter is adapter from my introductory article to cryptography. We'll take a quick, high-level overview of cryptography. The PyCrypto (https://www.dlitz.net/software/pycrypto/) and py-bcrypt (http://www.mindrot.org/projects/py-bcrypt/) libraries will be used to illustrate some general things to know when writing cryptographic code. We'll take a look at symmetric, public-key, hybrid, cryptographic hashing, password hashing, and message authentication codes.

Some quick terminology: for those unfamiliar, it's important to know what these mean:

- **plaintext**: the original message
- **ciphertext**: the message after cryptographic transformations are applied to obscure the original message.
- **encrypt**: producing ciphertext by applying cryptographic transformations to plaintext.
- **decrypt**: producing plaintext by applying cryptographic transformations to ciphertext.
- **cipher**: a particular set of cryptographic transformations providing means of both encryption and decryption.
- **ciphersystem**: a set of cryptographic transformations that take a large input and transform it to a unique (typically fixed-size) output. For hashes to be cryptographically secure, collisions should be practically nonexistent. It should be practically impossible to determine the input from the output.

Cryptography is an often misunderstood component of information security. We covered some information security principles in the previous chapter; the ones that cryptography covers are:

- **privacy** (aka **confidentiality**).
- **data integrity**: the plaintext message arrives unaltered.
- **entity authentication**: the identity of the sender is verified. An entity may be a person or a machine.
- **message authentication**: the message is verified as having been unaltered.

Note that cryptography is used to obscure the contents of a message and verify its contents and source. It will **not** hide the fact that two entities are communicating.

There are two basic types of ciphers: symmetric and public-key ciphers. A symmetric key cipher employs the use of shared secret keys. They also tend to be much faster than public-key ciphers. A public-key cipher is so-called because each key consists of a private key which is used to generate a public key. Like their names imply, the private key is kept secret while the public key is passed around. First, I'll take a look at a specific type of symmetric ciphers: block ciphers.

1.2 Block Ciphers

There are two further types of symmetric keys: stream and block ciphers. Stream ciphers operate on data streams, i.e. one byte at a time. Block ciphers operate on blocks of data, typically 16 bytes at a time. The most common block cipher and the standard one you should use unless you have a very good reason to use another one is the AES block cipher, documented in FIPS PUB 197 (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) . AES is a specific subset of the Rijndael cipher. AES uses block size of 128-bits (16 bytes); data should be padded out to fit the block size - the length of the data block must be multiple of the block size. For example, given an input of `ABCDABCDABCDABCD ABCDABCDABCDABCD` no padding would need to be done. However, given `ABCDABCDABCDABCD ABCDABCDABCD` an additional 4 bytes of padding would need to be added. A common padding scheme is to use `0x80` as the first byte of padding, with `{0x00` bytes filling out the rest of the padding. With padding, the previous example would look like: `ABCDABCDABCDABCD ABCDABCDABCD\x80\x00\x00\x00` .

Writing a padding function is pretty easy:
``` def pad_data(data): # return data if no padding is required if len(data) % 16 == 0: return data

```
# subtract one byte that should be the 0x80
# if 0 bytes of padding are required, it means only
# a single \x80 is required.

padding_required     = 15 - (len(data) % 16)

data = '%s\x80' % data
data = '%s%s' % (data, '\x00' * padding_required)

return data
```

```

Similarly, removing padding is also easy: ``` def unpad_data(data): if not data: return data

```
data = data.rstrip('\x00')
    if data[-1] == '\x80':
        return data[:-1]
    else:
        return data
```

```

I've included these functions in the example code (http://www.kyleisom.net/publish/datasec/code/chapter1.tar.gz) for the book.

Encryption with a block cipher requires selecting a block mode (http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html) . By far the most common mode used is **cipher block chaining** or *CBC* mode. Other modes include **counter (CTR)**, **cipher feedback (CFB)**, and the extremely insecure **electronic codebook (ECB)**. CBC mode is the standard and is well-vetted, so I will stick to that in this tutorial. Cipher
```

block chaining works by XORing the previous block of ciphertext with the current block. You might recognise that the first block has nothing to be XOR'd with; enter the *initialisation vector* (http://www.cs.ucdavis.edu/~rogaway/papers/nonce.pdf) . This comprises a number of randomly-generated bytes of data the same size as the cipher's block size. This initialisation vector should random enough that it cannot be recovered; one manner of doing this is to combine a standard UNIX timestamp with a block-size group of random data, using a standard hashing algorithm such as MD5 to make it unique.

One of the most critical components to encryption is properly generating random data. Fortunately, most of this is handled by the PyCrypto library's `Crypto.Random.OSRNG module`. You should know that the more entropy sources available (such as network traffic and disk activity), the faster the system can generate cryptographically-secure random data. I've written a function that can generate a *cryptographic nonce* (an number used only once) suitable for use as an initialisation vector. This will work on a UNIX machine; the comments note how easy it is to adapt it to a Windows machine. This function requires a version of PyCrypto at least 2.1.0 or higher. ``` import time import Crypto.Random.OSRNG.posix

def generate_nonce(): rnd = Crypto.Random.OSRNG.posix.new().read(BLOCK_SIZE) rnd = '% s% s' % (rnd, str(time.time())) nonce = Crypto.Hash.MD5.new(data = rnd)

```
return nonce.digest()
```

```

I will note here that the python `random` module is completely unsuitable for cryptography (as it is completely deterministic). You shouldn't use it for cryptographic code. If you're using another language, you need to be careful to select a cryptographically-secure random number generator (RNG).

Symmetric ciphers are so-named because the key is shared across any entities. There are three key sizes for AES: 128-bit, 192-bit, and 256-bit, aka 16-byte, 24-byte, and 32-byte key sizes. If you want to use a passphrase, you should use a digest algorithm that produces an appropriately sized digest, and hash that passphrase. For example, for AES-256, you would want to use SHA-256. Here is a sample function to generate an AES-256 key from a passphrase: ``` # generate an AES-256 key from a passphrase def passphrase(password, readable = False): """ Converts a passphrase to a format suitable for use as an AES key.

```
If readable is set to True, the key is output as a hex digest. This is
suitable for sharing with users or printing to screen when debugging
code.

By default readable is set to False, in which case the value it
returns is suitable for use directly as an AES-256 key.
"""
key   = Crypto.Hash.SHA256.new(password)

if readable:
    return key.hexdigest()
```
```

```
    else:
        return key.digest()
```

```

We could include this a set of AES encryption and decryption functions: ``` mode = Crypto.Cipher.AES.MODE_CBC # shortcut to clean up code

# AES-256 encryption using a passphrase def passphrase_encrypt(password, iv, data): key = passphrase(password) data = pad_data(data) aes = Crypto.Cipher.AES.new(key, mode, iv)

```
    return aes.encrypt(data)
```

# AES-256 decryption using a passphrase def passphrase_decrypt(password, iv, data): key = passphrase(password) aes = Crypto.Cipher.AES.new(key, mode, iv) data = aes.decrypt(data)

```
    return unpad_data(data)
```

```

Notice how the data is padded before being encrypted and unpadded after decryption - the decryption process will not remove the padding on its own.

Unless you are you doing interactive encryption passphrase encryption won't be terribly useful. Instead, we just need to generate 32 random bytes (and make sure we keep track of it) and use that as the key: `# generate a random AES-256 key def generate_aes_key(): rnd = Crypto.Random.OSRNG.posix.new().read(KEY_SIZE) return rnd`

We can use this key directly in the AES transformations: ``` def encrypt(key, iv, data): aes = Crypto.Cipher.AES.new(key, mode, iv) data = pad_data(data)

```
    return aes.encrypt(data)
```

def decrypt(key, iv, data): aes = Crypto.Cipher.AES.new(key, mode, iv) data = aes.decrypt(data)

```
    return unpad_data(data)
```

```

That should cover the basics of block cipher encryption. We've gone over key generation, padding, and encryption / decryption. AES-256 isn't the only block cipher provided by the PyCrypto package, but again: it is the standard and well vetted.

## 1.3 ASCII-Armouring

I'm going to take a quick detour and talk about ASCII armouring. If you've played with the crypto functions above, you'll notice they produce an annoying dump of binary data that can be a hassle to deal with. One common technique for making the data a little bit easier to deal with is to encode it with base64 (which is defined in RFC 1521 (http://tools.ietf.org/html/rfc1521.html) . There are a few ways to incorporate this into python:
```

1.3.1 Absolute Base64 Encoding

The easiest way is to just base64 encode everything in the encrypt function. Everything that goes into the decrypt function should be in base64 - if it's not, the `base64` module will throw an error: you could catch this and then try to decode it as binary data.

1.3.2 A Simple Header

A slightly more complex option, and the one I adopt in this article, is to use a `x00` as the first byte of the ciphertext for binary data, and to use `\x41` (an ASCII " `A` ") for ASCII encoded data. This will increase the complexity of the encryption and decryption functions slightly. We'll also pack the initialisation vector at the beginning of the file as well. Given now that the `iv` argument might be `None` in the decrypt function, I will have to rearrange the arguments a bit; for consistency, I will move it in both functions. I leave adding it into the `passphrase_encrypt` and `passphrase_decrypt` functions as an exercise for the reader. My modified functions look like this now: ``` def encrypt(key, data, iv, armour = False): aes = Crypto.Cipher.AES.new(key, mode, iv) data = pad_data(data) ct = aes.encrypt(data) # ciphertext ct = iv + ct # pack the initialisation vector in

```
# ascii-armouring
if armour:
    ct = '\x41' + base64.encodestring(ct)
else:
    ct = '\x00' + ct

return ct
```

def decrypt(key, data, iv = None): # remove ascii-armouring if present if data[0] == '\x00': data = data[1:] elif data[0] == '\x41': data = base64.decodestring(data[1:])

```
iv      = data[:16]
data    = data[16:]
aes     = Crypto.Cipher.AES.new(key, mode, iv)
data    = aes.decrypt(data)
return unpad_data(data)
```

```

### 1.3.3 A More Complex Container

There are more complex ways to do it (and you'll see it with the public keys in the next section) that involve putting the base64 into a container of sorts that contains additional information about the key.

## 1.4 Public Key Cryptography

Now it is time to take a look at public-key cryptography. Public-key cryptography, or PKC, involves the use of two-part keys. The private key is the sensitive key that should be kept private by the owning entity, whereas the public key (which is generated from the private key) is meant to be distributed to any entities which must communicate securely with the entity owning the private key. Confusing? Let's look at this using the venerable Alice and Bob, patron saints of cryptography.
```

Alice wants to talk to Bob, but doesn't want Eve to know the contents of the message. Both Alice and Bob generate a set of private keys. From those private keys, they both generate public keys. Let's say they post their public keys on their websites. Alice wants to send a private message to Bob, so she looks up Bob's public key from his site. (In fact, there is a way to distribute keys via a central site or entity; this is called a public key infrastructure (PKI). The public key can be used as the key to encrypt a message with PKC. The resulting ciphertext can only be decrypted using Bob's private key. Alice sends Bob the resulting ciphertext, which Eve cannot decrypt without Bob's private key. Hopefully this is a little less confusing.

One of the most common PKC systems is RSA (which is an acronym for the last names of the designers of the algorithm). Generally, RSA keys are 1024-bit, 2048-bit, or 4096-bits long. The keys are most often in PEM (defined in RFCs 1421-1424 (https://tools.ietf.org/html/rfc1421)) or DER (http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf) format. Generating RSA keys with PyCrypto is extremely easy: ``` def generate_key(size): PRNG = Crypto.Random.OSRNG.posix.new().read key = Crypto.PublicKey.RSA.generate(size, PRNG)

```
return key
```

```

The `key` that is returned isn't like the keys we used with the block ciphers: it is an RSA object and comes with several useful built-in methods. One of these is the `size()` method, which returns the size of the key in bits minus one. For example: ```

“
“
“
import publickey key = publickey.generate_key( 1024 ) key.size() 1023

```

”
”
”
A quick note: I will use 1024-bit keys in this tutorial because they are faster to generate, but in practice you should be using at least 2048-bit keys. The key also includes encryption and decryption methods in the class: ```

“
“
“
import publickey import base64 message = 'Test message...' ciphertext = key.encrypt(message, None) print base64.encodestring(ciphertext[0]) gzA9gXfHqnkValdhhYjRVVSxuygx48i66h0vFUnmVu8FZXJtmaACvNDo43D0vjjHzFiblE1eCFiI xlhVuHxldWXJSnARgWX1bTY7imR9Hve+WQC8rl+qB5xpq3xnKH7/z8/5YdLvCo/knXYE1cI/XYJP EP1nA6bUZNj6bD1Zx4w= ```

”
”
”

The `None` that is passed into the encryption function is part of the PyCrypto API for those publickey ciphers requiring an additional random number function to be passed in. It returns a tuple containing only the encrypted message. In order to pass this to the decryption function, we need to pass only the encrypted message as a string: ```

“
“
“
ciphertext = key.encrypt(message, None)[0] key.decrypt(ciphertext) 'Test message...' ```

”
”
”
While these are simple enough, we could put them into a pair of functions that also include ASCII-armouring: ``` def encrypt(key, message, armour = True): ciphertext = key.encrypt(message, None) ciphertext = ciphertext[0]

```
if armour:
    ciphertext = '\x41' + base64.encodestring( ciphertext )
else:
    ciphertext = '\x00' + base64.encodestring( ciphertext )

return ciphertext
```

def decrypt(key, message): if '\x00' == message[0]: message = message[1:] elif '\x41' == message[0]: message == base64.decodestring(message[1:])

```
plaintext   = key.decrypt( message )
    return plaintext
```

```

These two functions present a common API that will simplify encryption and decryption and make it a little easier to read. Assuming we still have the same `message` definition as before: ```

“
“
“
ciphertext = publickey.encrypt(key, message) publickey.decrypt(key, ciphertext) 'Test message...' ```

”
”
”
Now, what if we want to export this generated key and read it in later? The key comes with the method `exportKey()`. If the key is a private key, it will export the private key; if it is a public key, it will export the public key. We can write functions to backup our private key (which **absolutely** needs to be kept secure) and a function to export our public key, suitable for uploading to our web page or to a PKI keystore:
```

``` # backup our key, whether public or private def export_key(filename, key): try: f = open(filename, 'w') except IOError as e: print e raise e else: f.write( key.exportKey() ) f.close()

# will only export the public key def export_pubkey(filename, key): try: f = open(filename, 'w') except IOError as e: print e raise e else: f.write( key.publickey().exportKey() ) f.close() ```

Importing a key is done using the `RSA.importKey` function: `def load_key(filename): try: f = open(filename) except IOError as e: print e raise else: key = Crypto.PublicKey.RSA.importKey(f.read()) f.close() return key`

We can take a look at the difference between the public and private keys: ```

“

“

“

key = publickey.generate_key( 1024 ) print key.exportKey() -----BEGIN RSA PRIVATE KEY----- MIICXAIBAAKBgQCpVA2pqLuS1fmutvx/lBhlk+UMXWcZKVzh+n5D6Hv/ZWhlzRuC q408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3LlO6tQZ0gC2oOHetX3YOghO3q4yMe wvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJhfxko5YZuaghhpd56wIDAQAB AoGAaRznellnT2iLHX00U1IwruXXOwzEUmdN5G4mcathRhLCcueXW095VqhBR5Ez Vf8XU4EFU1MFKei0mLys3ehFV4aoTfU1xm91jXNZrM/rIjHQQObx2fcDSgrM9iyd kcgGrz5nDvsyxAxOwxCh96vNxZZYTWa8Zcqng1XYeW93nFkCQQC8Rqwn9Sa1UjBB mIepkcdYf-flkzmD7IBcgiTmGFQ9NXiehY6MQd0UJoFYGBEknPazzWQbNVpkZO4TR oPuKNjSNAkEA5jyWJhKyq2BVD6UP77vYTJu48OhLx4J7qb3DKHnk5syOBnbke2Df KV1VjRsipSjb4EXAWhWaqnTfPPDbvyWWVwJAWUgSP2iLkJSG+bRBMPJGW/pxF5Ke fre6/ 9zTAHhgJ0os9OVw4FAO1v/Hi1bg8dDXgRaImTsloseMtnPmlKYbyQJAbmbr EQKyTl95KnFaPPj0dXfOrSaW/+pf5jsqlAQvcUTxbcQhN9Bx8mHhHjK+4DfBh7+q xwfJDKfSTGSq2vPpLQJBAL5irIeHoFESPZZI1NW7OkpKPcO/2ps9NkhgZJQ7Pc11 lWh6Ch2c-nBzZmeh6lN/zC4l3mLVhdZSXkEKOzeuFpBs= -----END RSA PRIVATE KEY----- pk = key.publickey() print pk.exportKey() -----BEGIN PUBLIC KEY----- MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCpVA2pqLuS1fmutvx/lBhlk+UM XWcZKVzh+n5D6Hv/ZWhlzRuCq408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3LlO6t QZ0gC2oOHetX3YOghO3q4yMewvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJ hfxko5YZuaghhpd56wIDAQAB -----END PUBLIC KEY----- ```

”

”

”

Using the `export_pubkey()` function, you can pass that file around to people to encrypt messages to you. Often, you wil want to generate a keypair to give to people. One convention is to name the secret key `keyname.prv` (prv for private, and sometimes `keyname.key`) and the public key `keyname.pub`. We will follow that convention in an `export_keypair()` function: ``` def export_keypair(basename, key): pubkeyfile = basename + '.pub' prvkeyfile = basename + '.prv'

```
export_key(prvkeyfile, key)
export_pubkey(pubkeyfile, key)
```

```

For example, Bob generates a keypair and emails the public key to Alice: ```

“
“
“

key = publickey.generate_key(1024) key.size() 1023 key.has_private() True publickey.export_keypair('bob.prv', key)

```

”
”
”

Then, Assuming Bob gave Alice `bob.pub`: ```

“
“
“

bob = publickey.load_key('./bob.pub') message = 'secret message from Alice to Bob' print publickey.encrypt(bob, message) AN6RsuXEeKicUZKtZCsDeqGKeB5em+NG/bgoqr9l8ij2o1Gr9sT69tv0zxgmigK/Jt+gPxg/EDu61
nHmAK0XQV7BvJS5jLuBxdJ0mEpysVClu46XN1KHU2l2DsGht9e8OFvhEfDkI5t/cy/gXr0xz/EUirqo8qLd9Mw6TerM8gs8= ```

”
”
”

The ASCII-armoured format makes it convenient for Alice to paste the encrypted message to Bob, so she does, and now Bob has it on his computer. To read it, he does something similar: ```

“
“
“

bob = publickey.load_key('tests/bob.prv') print publickey.decrypt(bob, message) secret message from Alice to Bob ```

”
”
”

At this point, Bob can't be sure that the message came from Alice but he can read the message. We'll cover entity authentication in a later section, but first, there's something else I'd to point out:

You might have noticed at this point that public key cryptography appears to be a lot simpler than symmetric key cryptography. The key distribution problem is certainly easier, especially with a proper PKI. Why would anyone choose to use symmetric key cryptography over public key cryptography? The answer is performance: if you compare the block cipher test code with the public key test code, you will notice that the block cipher code is orders of magnitude faster - and it generates far more keys than the public key code. There is a solution to this problem: hybrid cryptosystems.
```

Hybrid cryptosystems use public key cryptography to establish a symmetric session key. Both **TLS** (Transport Layer Security), and its predecessor **SSL** (Secure Sockets Layer), most often used to secure HTTP transactions, use a hybrid cryptosystem to speed up establishing a secure session. PGP also uses hybrid crypto.

Let's say Alice and Bob wish to use hybrid crypto; if Alice initiates the session, she should be the one to generate the session key. For example, ```

“
“
“
import block, publickey session_key = block.generate_aes_key() alice_key = publickey.load_key('keys/alice.prv') bob_key = publickey.load_key('keys/bob.pub') encrypted_session_key = encrypt(bob_key, session_key) ```

”
”
”
At this point, Alice should send Bob the `encrypted_session_key`; she should retain a copy as well. They can then use this key to communicate using the much-faster AES256.

In communicating, it might be wise to create a message format that packs in the session key into a header, and encrypts the rest of the body with the session key. This is a subject beyond the realm of a quick tutorial - again, consult with the people who do this on a regular basis.

1.5 Digital Signatures

In all of the previous examples, we assumed that the identity of the sender wasn't a question. For a symmetric key, that's less of a stretch - there's no differentiation between owners. Public keys, however, are supposed to be associated with an entity. How can we prove the identity of the user? Without delving into too much into social sciences and trust metrics and a huge philosophical argument, let's look at the basics of signatures.

A signature works similarly to encryption, but it in reverse, and it is slightly different: a hash of the message is 'encrypted' by the private key to the public key. The public key is used to 'decrypt' this ciphertext. Contrast this to actual public key encryption: the entire message is encrypted to the private key by the public key, and the private key is used to decrypt the ciphertext. With signatures, the 'encrypted' hash of the message is called the signature, and the act of 'encryption' is termed 'signing'. Similarly, the 'decryption' is known as verification or verifying the signature.

PyCrypto's PublicKey implementations already come with signatures and verification methods for keys using `sign()` and `verify()`. The signature is a long in a tuple: ```

“
“
“
key.sign(d, None)
(173842351815267154566957144586003794451816219765633312346624801514795542424 8876723731383711018550231967374810686606623315483033485630977014574359346192927942

623807461783144628656796225504478196458051789241311033020911767301220653148276004
055135752638362705938208187879104016981500905101694922017804476413090 8L,) ```

”
”
”

We can write our own functions to wrap around these two functions and perform ASCII-armouring if desired. Our signature function should take a key and a message (and optionally a flag to ASCII armour the signature), and sign a digest of the message: ``` def sign(key, message, armour = True): if not key.can_sign(): return None

```
digest      = Crypto.Hash.SHA256.new(message).digest()
signature   = key.sign( digest, None )[0]

if armour:
    sig     = base64.encodestring( str(signature) )
else:
    sig     = str( signature )

return sig.strip()
```

```

The signature is converted to a string to make it easier to pack it into structures and also to give us consistent input to the verify() function.

Verifying the signature requires that we determine if the signature is ASCII- armoured or not, then comparing a digest of the message to the signature: ``` def verify(key, message, signature): try: sig = long( signature ) except ValueError as e: sig = long( base64.decodestring( signature.rstrip('\n') ), )

```
digest   = Crypto.Hash.SHA256.new(message).digest()
return key.verify(digest, (sig,))
```

```

The `sign()` function returns a signature and the `verify()` function returns a boolean. Now, Alice can sign her message to Bob, and Bob knows the key belongs to Alice. She sends Bob the signature and the encrypted message. Bob then makes sure Alice's key properly verifies the signature to the encrypted message.

1.6 Key Exchange

So how does Bob know the key actually belongs to Alice? There are two main schools of thought regarding the authentication of key ownership: centralised and decentralised. TLS/SSL follow the centralised school: a root certificate (a certificate is a public key encoded with X.509 and which can have additional informational attributes attached, such as organisation name and country) authority (CA) signs intermediary CA keys, which then sign user keys. For example, if Bob runs Foo Widgets, LLC, he can generate an SSL keypair. From this, he generates a certificate signing request, and sends this to the CA.

The CA, usually after taking some money and ostensibly actually verifying Bob's identity (the extent to which this actually happens varies widely based on the different CAs), then signs Bob's certificate. Bob sets up his webserver to use his SSL certificate for all secure traffic, and Alice sees that the CA did in fact sign his certificate. This relies on trusted central authorities, like VeriSign (there is some question as to whether VeriSign can actually be trusted, but that is another discussion for another day...) Alice's web browser would ship with a keystore of select trusted CA public keys (like VeriSign's) that she could use to verify signatures on the certificates from various sites. This system is called a public key infrastructure.

The other school of thought is followed by PGP (and GnuPG) - the decentralised model. In PGP, this is manifested as the Web of Trust (http://www.rubin.ch/pgp/weboftrust.en.html) . For example, if Carol now wants to talk to Bob and gives Bob her public key, Bob can check to see if Carol's key has been signed by anyone else. We'll also say that Bob knows for a fact that Alice's key belongs to Alice, and he trusts her—it is quite often important to distinguish between *I know this key belongs to that user* and *I trust that user*, which is especially important with key signatures - if Bob cannot trust Alice to properly check identities, she might sign a key for an identity she hasn't checked—and that Alice has signed Carol's key. Bob sees Alice's signature on Carol's key and then can be reasonably sure that Carol is who she says it was. If we repeat the process with Dave, whose key was signed by Carol (whose key was signed by Alice), Bob might be able to be more certain that the key belongs to Dave, but maybe he doesn't really trust Carol to properly verify identities. In PGP, Bob can mark keys as having various trust levels, and from this a web of trust emerges: a picture of how well you can trust that a given key belongs to a given user.

The key distribution problem is not a quick and easy problem to solve; a lot of very smart people have spent a lot of time coming up with solutions to the problem. There are key exchange protocols (such as the Diffie-Hellman key exchange (http://is.gd/Tr0zLP) and IKE (Internet Key Exchange, defined in RFC 2409 (http://www.ietf.org/rfc/rfc2409.txt) (which uses Diffie-Hellman) that provide alternatives to the web of trust and public key infrastructures.

CHAPTER 2

Practical Introduction to Cryptography

2.1 Introduction

This chapter is adapter from my introductory article to cryptography. We'll take a quick, high-level overview of cryptography. The PyCrypto (https://www.dlitz.net/software/pycrypto/) and py-bcrypt (http://www.mindrot.org/projects/py-bcrypt/) libraries will be used to illustrate some general things to know when writing cryptographic code. We'll take a look at symmetric, public-key, hybrid, cryptographic hashing, password hashing, and message authentication codes.

Some quick terminology: for those unfamiliar, it's important to know what these mean:

- **plaintext**: the original message
- **ciphertext**: the message after cryptographic transformations are applied to obscure the original message.
- **encrypt**: producing ciphertext by applying cryptographic transformations to plaintext.
- **decrypt**: producing plaintext by applying cryptographic transformations to ciphertext.
- **cipher**: a particular set of cryptographic transformations providing means of both encryption and decryption.
- **ciphersystem**: a set of cryptographic transformations that take a large input and transform it to a unique (typically fixed-size) output. For hashes to be cryptographically secure, collisions should be practically nonexistent. It should be practically impossible to determine the input from the output.

Cryptography is an often misunderstood component of information security. We covered some information security principles in the previous chapter; the ones that cryptography covers are:

- **privacy** (aka **confidentiality**).
- **data integrity**: the plaintext message arrives unaltered.
- **entity authentication**: the identity of the sender is verified. An entity may be a person or a machine.
- **message authentication**: the message is verified as having been unaltered.

Note that cryptography is used to obscure the contents of a message and verify its contents and source. It will **not** hide the fact that two entities are communicating.

There are two basic types of ciphers: symmetric and public-key ciphers. A symmetric key cipher employs the use of shared secret keys. They also tend to be much faster than public-key ciphers. A public-key cipher is so-called because each key consists of a private key which is used to generate a public key. Like their names imply, the private key is kept secret while the public key is passed around. First, I'll take a look at a specific type of symmetric ciphers: block ciphers.

2.2 Block Ciphers

There are two further types of symmetric keys: stream and block ciphers. Stream ciphers operate on data streams, i.e. one byte at a time. Block ciphers operate on blocks of data, typically 16 bytes at a time. The most common block cipher and the standard one you should use unless you have a very good reason to use another one is the AES block cipher, documented in FIPS PUB 197 (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) . AES is a specific subset of the Rijndael cipher. AES uses block size of 128-bits (16 bytes); data should be padded out to fit the block size - the length of the data block must be multiple of the block size. For example, given an input of `ABCDABCDABCDABCD ABCDABCDABCDABCD` no padding would need to be done. However, given `ABCDABCDABCDABCD ABCDABCDABCD` an additional 4 bytes of padding would need to be added. A common padding scheme is to use `0x80` as the first byte of padding, with `{0x00` bytes filling out the rest of the padding. With padding, the previous example would look like: `ABCDABCDABCDABCD ABCDABCDABCD\x80\x00\x00\x00` .

Writing a padding function is pretty easy:
``` def pad_data(data): # return data if no padding is required if len(data) % 16 == 0: return data

```
# subtract one byte that should be the 0x80
# if 0 bytes of padding are required, it means only
# a single \x80 is required.

padding_required     = 15 - (len(data) % 16)

data = '%s\x80' % data
data = '%s%s' % (data, '\x00' * padding_required)

return data
```

```

Similarly, removing padding is also easy: ``` def unpad_data(data): if not data: return data

```
data = data.rstrip('\x00')
    if data[-1] == '\x80':
        return data[:-1]
    else:
        return data
```

```

I've included these functions in the example code (http://www.kyleisom.net/publish/datasec/code/chapter1.tar.gz) for the book.

Encryption with a block cipher requires selecting a block mode (http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html) . By far the most common mode used is **cipher block chaining** or *CBC* mode. Other modes include **counter (CTR)**, **cipher feedback (CFB)**, and the extremely insecure **electronic codebook (ECB)**. CBC mode is the standard and is well-vetted, so I will stick to that in this tutorial. Cipher
```

block chaining works by XORing the previous block of ciphertext with the current block. You might recognise that the first block has nothing to be XOR'd with; enter the *initialisation vector* (http://www.cs.ucdavis.edu/~rogaway/papers/nonce.pdf) . This comprises a number of randomly-generated bytes of data the same size as the cipher's block size. This initialisation vector should random enough that it cannot be recovered; one manner of doing this is to combine a standard UNIX timestamp with a block-size group of random data, using a standard hashing algorithm such as MD5 to make it unique.

One of the most critical components to encryption is properly generating random data. Fortunately, most of this is handled by the PyCrypto library's `Crypto.Random.OSRNG module`. You should know that the more entropy sources available (such as network traffic and disk activity), the faster the system can generate cryptographically-secure random data. I've written a function that can generate a *cryptographic nonce* (an number used only once) suitable for use as an initialisation vector. This will work on a UNIX machine; the comments note how easy it is to adapt it to a Windows machine. This function requires a version of PyCrypto at least 2.1.0 or higher. ``` import time import Crypto.Random.OSRNG.posix

def generate_nonce(): rnd = Crypto.Random.OSRNG.posix.new().read(BLOCK_SIZE) rnd = '% s% s' % (rnd, str(time.time())) nonce = Crypto.Hash.MD5.new(data = rnd)

```
return nonce.digest()
```

```

I will note here that the python `random` module is completely unsuitable for cryptography (as it is completely deterministic). You shouldn't use it for cryptographic code. If you're using another language, you need to be careful to select a cryptographically-secure random number generator (RNG).

Symmetric ciphers are so-named because the key is shared across any entities. There are three key sizes for AES: 128-bit, 192-bit, and 256-bit, aka 16-byte, 24-byte, and 32-byte key sizes. If you want to use a passphrase, you should use a digest algorithm that produces an appropriately sized digest, and hash that passphrase. For example, for AES-256, you would want to use SHA-256. Here is a sample function to generate an AES-256 key from a passphrase: ``` # generate an AES-256 key from a passphrase def passphrase(password, readable = False): """ Converts a passphrase to a format suitable for use as an AES key.

```
If readable is set to True, the key is output as a hex digest. This is
suitable for sharing with users or printing to screen when debugging
code.

By default readable is set to False, in which case the value it
returns is suitable for use directly as an AES-256 key.
"""
key   = Crypto.Hash.SHA256.new(password)

if readable:
    return key.hexdigest()
```
```

```
    else:
        return key.digest()
```

```

We could include this a set of AES encryption and decryption functions: ``` mode = Crypto.Cipher.AES.MODE_CBC # shortcut to clean up code

# AES-256 encryption using a passphrase def passphrase_encrypt(password, iv, data): key = passphrase(password) data = pad_data(data) aes = Crypto.Cipher.AES.new(key, mode, iv)

```
    return aes.encrypt(data)
```

# AES-256 decryption using a passphrase def passphrase_decrypt(password, iv, data): key = passphrase(password) aes = Crypto.Cipher.AES.new(key, mode, iv) data = aes.decrypt(data)

```
    return unpad_data(data)
```

```

Notice how the data is padded before being encrypted and unpadded after decryption - the decryption process will not remove the padding on its own.

Unless you are you doing interactive encryption passphrase encryption won't be terribly useful. Instead, we just need to generate 32 random bytes (and make sure we keep track of it) and use that as the key: `# generate a random AES-256 key def generate_aes_key(): rnd = Crypto.Random.OSRNG.posix.new().read(KEY_SIZE) return rnd`

We can use this key directly in the AES transformations: ``` def encrypt(key, iv, data): aes = Crypto.Cipher.AES.new(key, mode, iv) data = pad_data(data)

```
    return aes.encrypt(data)
```

def decrypt(key, iv, data): aes = Crypto.Cipher.AES.new(key, mode, iv) data = aes.decrypt(data)

```
    return unpad_data(data)
```

```

That should cover the basics of block cipher encryption. We've gone over key generation, padding, and encryption / decryption. AES-256 isn't the only block cipher provided by the PyCrypto package, but again: it is the standard and well vetted.

## 2.3 ASCII-Armouring

I'm going to take a quick detour and talk about ASCII armouring. If you've played with the crypto functions above, you'll notice they produce an annoying dump of binary data that can be a hassle to deal with. One common technique for making the data a little bit easier to deal with is to encode it with base64 (which is defined in RFC 1521 (http://tools.ietf.org/html/rfc1521.html) . There are a few ways to incorporate this into python:
```

2.3.1 Absolute Base64 Encoding

The easiest way is to just base64 encode everything in the encrypt function. Everything that goes into the decrypt function should be in base64 - if it's not, the `base64` module will throw an error: you could catch this and then try to decode it as binary data.

2.3.2 A Simple Header

A slightly more complex option, and the one I adopt in this article, is to use a `x00` as the first byte of the ciphertext for binary data, and to use `\x41` (an ASCII " `A` ") for ASCII encoded data. This will increase the complexity of the encryption and decryption functions slightly. We'll also pack the initialisation vector at the beginning of the file as well. Given now that the `iv` argument might be `None` in the decrypt function, I will have to rearrange the arguments a bit; for consistency, I will move it in both functions. I leave adding it into the `passphrase_encrypt` and `passphrase_decrypt` functions as an exercise for the reader. My modified functions look like this now: ``` def encrypt(key, data, iv, armour = False): aes = Crypto.Cipher.AES.new(key, mode, iv) data = pad_data(data) ct = aes.encrypt(data) # ciphertext ct = iv + ct # pack the initialisation vector in

```
# ascii-armouring
if armour:
    ct = '\x41' + base64.encodestring(ct)
else:
    ct = '\x00' + ct

return ct
```

def decrypt(key, data, iv = None): # remove ascii-armouring if present if data[0] == '\x00': data = data[1:] elif data[0] == '\x41': data = base64.decodestring(data[1:])

```
iv      = data[:16]
data    = data[16:]
aes     = Crypto.Cipher.AES.new(key, mode, iv)
data    = aes.decrypt(data)
return unpad_data(data)
```

```

### 2.3.3 A More Complex Container

There are more complex ways to do it (and you'll see it with the public keys in the next section) that involve putting the base64 into a container of sorts that contains additional information about the key.

## 2.4 Public Key Cryptography

Now it is time to take a look at public-key cryptography. Public-key cryptography, or PKC, involves the use of two-part keys. The private key is the sensitive key that should be kept private by the owning entity, whereas the public key (which is generated from the private key) is meant to be distributed to any entities which must communicate securely with the entity owning the private key. Confusing? Let's look at this using the venerable Alice and Bob, patron saints of cryptography.
```

Alice wants to talk to Bob, but doesn't want Eve to know the contents of the message. Both Alice and Bob generate a set of private keys. From those private keys, they both generate public keys. Let's say they post their public keys on their websites. Alice wants to send a private message to Bob, so she looks up Bob's public key from his site. (In fact, there is a way to distribute keys via a central site or entity; this is called a public key infrastructure (PKI). The public key can be used as the key to encrypt a message with PKC. The resulting ciphertext can only be decrypted using Bob's private key. Alice sends Bob the resulting ciphertext, which Eve cannot decrypt without Bob's private key. Hopefully this is a little less confusing.

One of the most common PKC systems is RSA (which is an acronym for the last names of the designers of the algorithm). Generally, RSA keys are 1024-bit, 2048-bit, or 4096-bits long. The keys are most often in PEM (defined in RFCs 1421-1424 (https://tools.ietf.org/html/rfc1421)) or DER (http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf) format. Generating RSA keys with PyCrypto is extremely easy: ``` def generate_key(size): PRNG = Crypto.Random.OSRNG.posix.new().read key = Crypto.PublicKey.RSA.generate(size, PRNG)

```
return key
```

```

The `key` that is returned isn't like the keys we used with the block ciphers: it is an RSA object and comes with several useful built-in methods. One of these is the `size()` method, which returns the size of the key in bits minus one. For example: ```

“
“
“
import publickey key = publickey.generate_key( 1024 ) key.size() 1023

```

”
”
”
A quick note: I will use 1024-bit keys in this tutorial because they are faster to generate, but in practice you should be using at least 2048-bit keys. The key also includes encryption and decryption methods in the class: ```

“
“
“
import publickey import base64 message = 'Test message...' ciphertext = key.encrypt(message, None) print base64.encodestring(ciphertext[0]) gzA9gXfHqnkValdhhYjRVVSxuygx48i66h0vFUnmVu8FZXJtmaACvNDo43D0vjjHzFiblE1eCFiI xlhVuHxldWXJSnARgWX1bTY7imR9Hve+WQC8rl+qB5xpq3xnKH7/z8/5YdLvCo/knXYE1cI/XYJP EP1nA6bUZNj6bD1Zx4w= ```

”
”
”

The `None` that is passed into the encryption function is part of the PyCrypto API for those publickey ciphers requiring an additional random number function to be passed in. It returns a tuple containing only the encrypted message. In order to pass this to the decryption function, we need to pass only the encrypted message as a string: ```

“
“
“
ciphertext = key.encrypt(message, None)[0] key.decrypt(ciphertext) 'Test message...' ```

”
”
”
While these are simple enough, we could put them into a pair of functions that also include ASCII-armouring: ``` def encrypt(key, message, armour = True): ciphertext = key.encrypt(message, None) ciphertext = ciphertext[0]

```
if armour:
    ciphertext = '\x41' + base64.encodestring( ciphertext )
else:
    ciphertext = '\x00' + base64.encodestring( ciphertext )

return ciphertext
```

def decrypt(key, message): if '\x00' == message[0]: message = message[1:] elif '\x41' == message[0]: message == base64.decodestring(message[1:])

```
plaintext   = key.decrypt( message )
    return plaintext
```

```

These two functions present a common API that will simplify encryption and decryption and make it a little easier to read. Assuming we still have the same `message` definition as before: ```

“
“
“
ciphertext = publickey.encrypt(key, message) publickey.decrypt(key, ciphertext) 'Test message...' ```

”
”
”
Now, what if we want to export this generated key and read it in later? The key comes with the method `exportKey()`. If the key is a private key, it will export the private key; if it is a public key, it will export the public key. We can write functions to backup our private key (which **absolutely** needs to be kept secure) and a function to export our public key, suitable for uploading to our web page or to a PKI keystore:
```

``` # backup our key, whether public or private def export_key(filename, key): try: f = open(filename, 'w') except IOError as e: print e raise e else: f.write( key.exportKey() ) f.close()

# will only export the public key def export_pubkey(filename, key): try: f = open(filename, 'w') except IOError as e: print e raise e else: f.write( key.publickey().exportKey() ) f.close() ```

Importing a key is done using the `RSA.importKey` function: `def load_key(filename): try: f = open(filename) except IOError as e: print e raise else: key = Crypto.PublicKey.RSA.importKey(f.read()) f.close() return key`

We can take a look at the difference between the public and private keys: ```

“

“

“

key = publickey.generate_key( 1024 ) print key.exportKey() -----BEGIN RSA PRIVATE KEY----- MIICXAIBAAKBgQCpVA2pqLuS1fmutvx/lBhlk+UMXWcZKVzh+n5D6Hv/ZWhlzRuC q408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3LlO6tQZ0gC2oOHetX3YOghO3q4yMe wvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJhfxko5YZuaghhpd56wIDAQAB AoGAaRznellnT2iLHX00U1IwruXXOwzEUmdN5G4mcathRhLCcueXW095VqhBR5Ez Vf8XU4EFU1MFKei0mLys3ehFV4aoTfU1xm91jXNZrM/rIjHQQObx2fcDSgrM9iyd kcgGrz5nDvsyxAxOwxCh96vNxZZYTWa8Zcqng1XYeW93nFkCQQC8Rqwn9Sa1UjBB mIepkcdYf-flkzmD7IBcgiTmGFQ9NXiehY6MQd0UJoFYGBEknPazzWQbNVpkZO4TR oPuKNjSNAkEA5jyWJhKyq2BVD6UP77vYTJu48OhLx4J7qb3DKHnk5syOBnbke2Df KV1VjRsipSjb4EXAWhWaqnTfPPDbvyWWVwJAWUgSP2iLkJSG+bRBMPJGW/pxF5Ke fre6/ 9zTAHhgJ0os9OVw4FAO1v/Hi1bg8dDXgRaImTsloseMtnPmlKYbyQJAbmbr EQKyTl95KnFaPPj0dXfOrSaW/+pf5jsqlAQvcUTxbcQhN9Bx8mHhHjK+4DfBh7+q xwfJDKfSTGSq2vPpLQJBAL5irIeHoFESPZZI1NW7OkpKPcO/2ps9NkhgZJQ7Pc11 lWh6Ch2c-nBzZmeh6lN/zC4l3mLVhdZSXkEKOzeuFpBs= -----END RSA PRIVATE KEY----- pk = key.publickey() print pk.exportKey() -----BEGIN PUBLIC KEY----- MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCpVA2pqLuS1fmutvx/lBhlk+UM XWcZKVzh+n5D6Hv/ZWhlzRuCq408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3LlO6t QZ0gC2oOHetX3YOghO3q4yMewvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJ hfxko5YZuaghhpd56wIDAQAB -----END PUBLIC KEY----- ```

”

”

”

Using the `export_pubkey()` function, you can pass that file around to people to encrypt messages to you. Often, you wil want to generate a keypair to give to people. One convention is to name the secret key `keyname.prv` (prv for private, and sometimes `keyname.key`) and the public key `keyname.pub`. We will follow that convention in an `export_keypair()` function: ``` def export_keypair(basename, key): pubkeyfile = basename + '.pub' prvkeyfile = basename + '.prv'

```
export_key(prvkeyfile, key)
export_pubkey(pubkeyfile, key)
```

```

For example, Bob generates a keypair and emails the public key to Alice: ```

“
“
“

key = publickey.generate_key(1024) key.size() 1023 key.has_private() True publickey.export_keypair('bob.prv', key)

```

”
”
”

Then, Assuming Bob gave Alice `bob.pub`: ```

“
“
“

bob = publickey.load_key('./bob.pub') message = 'secret message from Alice to Bob' print publickey.encrypt(bob, message) AN6RsuXEeKicUZKtZCsDeqGKeB5em+NG/bgoqr9l8ij2o1Gr9sT69tv0zxgmigK/Jt+gPxg/EDu61
nHmAK0XQV7BvJS5jLuBxdJ0mEpysVClu46XN1KHU2l2DsGht9e8OFvhEfDkI5t/cy/gXr0xz/EUirqo8qLd9Mw6TerM8gs8= ```

”
”
”

The ASCII-armoured format makes it convenient for Alice to paste the encrypted message to Bob, so she does, and now Bob has it on his computer. To read it, he does something similar: ```

“
“
“

bob = publickey.load_key('tests/bob.prv') print publickey.decrypt(bob, message) secret message from Alice to Bob ```

”
”
”

At this point, Bob can't be sure that the message came from Alice but he can read the message. We'll cover entity authentication in a later section, but first, there's something else I'd to point out:

You might have noticed at this point that public key cryptography appears to be a lot simpler than symmetric key cryptography. The key distribution problem is certainly easier, especially with a proper PKI. Why would anyone choose to use symmetric key cryptography over public key cryptography? The answer is performance: if you compare the block cipher test code with the public key test code, you will notice that the block cipher code is orders of magnitude faster - and it generates far more keys than the public key code. There is a solution to this problem: hybrid cryptosystems.
```

Hybrid cryptosystems use public key cryptography to establish a symmetric session key. Both **TLS** (Transport Layer Security), and its predecessor **SSL** (Secure Sockets Layer), most often used to secure HTTP transactions, use a hybrid cryptosystem to speed up establishing a secure session. PGP also uses hybrid crypto.

Let's say Alice and Bob wish to use hybrid crypto; if Alice initiates the session, she should be the one to generate the session key. For example, ```

“
“
“
import block, publickey session_key = block.generate_aes_key() alice_key = publickey.load_key('keys/alice.prv') bob_key = publickey.load_key('keys/bob.pub') encrypted_session_key = encrypt(bob_key, session_key) ```

”
”
”
At this point, Alice should send Bob the `encrypted_session_key`; she should retain a copy as well. They can then use this key to communicate using the much-faster AES256.

In communicating, it might be wise to create a message format that packs in the session key into a header, and encrypts the rest of the body with the session key. This is a subject beyond the realm of a quick tutorial - again, consult with the people who do this on a regular basis.

2.5 Digital Signatures

In all of the previous examples, we assumed that the identity of the sender wasn't a question. For a symmetric key, that's less of a stretch - there's no differentiation between owners. Public keys, however, are supposed to be associated with an entity. How can we prove the identity of the user? Without delving into too much into social sciences and trust metrics and a huge philosophical argument, let's look at the basics of signatures.

A signature works similarly to encryption, but it in reverse, and it is slightly different: a hash of the message is 'encrypted' by the private key to the public key. The public key is used to 'decrypt' this ciphertext. Contrast this to actual public key encryption: the entire message is encrypted to the private key by the public key, and the private key is used to decrypt the ciphertext. With signatures, the 'encrypted' hash of the message is called the signature, and the act of 'encryption' is termed 'signing'. Similarly, the 'decryption' is known as verification or verifying the signature.

PyCrypto's PublicKey implementations already come with signatures and verification methods for keys using `sign()` and `verify()`. The signature is a long in a tuple: ```

“
“
“
key.sign(d, None)
(173842351815267154566957144586003794451816219765633312346624801514795542424 8
876723731383711018550231967374810686606623315483033485630977014574359346192927942

623807461783144628656796225504478196458051789241311033020911767301220653148276004
055135752638362705938208187879104016981500905101694922017804476413090 8L,) ```

”
”
”

We can write our own functions to wrap around these two functions and perform ASCII-armouring if desired. Our signature function should take a key and a message (and optionally a flag to ASCII armour the signature), and sign a digest of the message: ``` def sign(key, message, armour = True): if not key.can_sign(): return None

```
digest      = Crypto.Hash.SHA256.new(message).digest()
signature   = key.sign( digest, None )[0]

if armour:
    sig     = base64.encodestring( str(signature) )
else:
    sig     = str( signature )

return sig.strip()
```

```

The signature is converted to a string to make it easier to pack it into structures and also to give us consistent input to the verify() function.

Verifying the signature requires that we determine if the signature is ASCII- armoured or not, then comparing a digest of the message to the signature: ``` def verify(key, message, signature): try: sig = long( signature ) except ValueError as e: sig = long( base64.decodestring( signature.rstrip('\n') ), )

```
digest   = Crypto.Hash.SHA256.new(message).digest()
return key.verify(digest, (sig,))
```

```

The `sign()` function returns a signature and the `verify()` function returns a boolean. Now, Alice can sign her message to Bob, and Bob knows the key belongs to Alice. She sends Bob the signature and the encrypted message. Bob then makes sure Alice's key properly verifies the signature to the encrypted message.

2.6 Key Exchange

So how does Bob know the key actually belongs to Alice? There are two main schools of thought regarding the authentication of key ownership: centralised and decentralised. TLS/SSL follow the centralised school: a root certificate (a certificate is a public key encoded with X.509 and which can have additional informational attributes attached, such as organisation name and country) authority (CA) signs intermediary CA keys, which then sign user keys. For example, if Bob runs Foo Widgets, LLC, he can generate an SSL keypair. From this, he generates a certificate signing request, and sends this to the CA.

The CA, usually after taking some money and ostensibly actually verifying Bob's identity (the extent to which this actually happens varies widely based on the different CAs), then signs Bob's certificate. Bob sets up his webserver to use his SSL certificate for all secure traffic, and Alice sees that the CA did in fact sign his certificate. This relies on trusted central authorities, like VeriSign (there is some question as to whether VeriSign can actually be trusted, but that is another discussion for another day...) Alice's web browser would ship with a keystore of select trusted CA public keys (like VeriSign's) that she could use to verify signatures on the certificates from various sites. This system is called a public key infrastructure.

The other school of thought is followed by PGP (and GnuPG) - the decentralised model. In PGP, this is manifested as the Web of Trust (http://www.rubin.ch/pgp/weboftrust.en.html) . For example, if Carol now wants to talk to Bob and gives Bob her public key, Bob can check to see if Carol's key has been signed by anyone else. We'll also say that Bob knows for a fact that Alice's key belongs to Alice, and he trusts her—it is quite often important to distinguish between *I know this key belongs to that user* and *I trust that user*, which is especially important with key signatures - if Bob cannot trust Alice to properly check identities, she might sign a key for an identity she hasn't checked—and that Alice has signed Carol's key. Bob sees Alice's signature on Carol's key and then can be reasonably sure that Carol is who she says it was. If we repeat the process with Dave, whose key was signed by Carol (whose key was signed by Alice), Bob might be able to be more certain that the key belongs to Dave, but maybe he doesn't really trust Carol to properly verify identities. In PGP, Bob can mark keys as having various trust levels, and from this a web of trust emerges: a picture of how well you can trust that a given key belongs to a given user.

The key distribution problem is not a quick and easy problem to solve; a lot of very smart people have spent a lot of time coming up with solutions to the problem. There are key exchange protocols (such as the Diffie-Hellman key exchange (http://is.gd/Tr0zLP) and IKE (Internet Key Exchange, defined in RFC 2409 (http://www.ietf.org/rfc/rfc2409.txt) (which uses Diffie-Hellman) that provide alternatives to the web of trust and public key infrastructures.

APPENDIX A

Bibliography

Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Introduction to Differential Power Analysis and Related Attacks." Cryptography Research, San Francisco, CA 1998. http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf

Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." Cryptography Research, San Francisco, CA 2000. http://www.cryptography.com/resources/whitepapers/DPA.pdf