5 Asymmetric encryption

This chapter covers

- Introducing the key-distribution problem
- Demonstrating asymmetric encryption with the `cryptography` package
- Ensuring nonrepudiation with digital signatures

In the previous chapter, you learned how to ensure confidentiality with symmetric encryption. Symmetric encryption, unfortunately, is no panacea. By itself, symmetric encryption is unsuitable for key distribution, a classic problem in cryptography. In this chapter, you'll learn how to solve this problem with asymmetric encryption. Along the way, you'll learn more about the Python package named `cryptography`. Finally, I'll show you how to ensure nonrepudiation with digital signatures.

5.1 Key-distribution problem

Symmetric encryption works great when the encryptor and decryptor are the same party, but it doesn't scale well. Suppose Alice wants to send Bob a confidential message. She encrypts the message and sends the ciphertext to Bob. Bob needs Alice's key to decrypt the message. Alice now has to find a way to distribute the key to Bob without Eve, an eavesdropper, intercepting the key. Alice could encrypt her key with a second key, but how does she safely send the second key to Bob? Alice could encrypt her second key with a third key, but how does she . . . you get the point. Key distribution is a recursive problem.

The problem gets dramatically worse if Alice wants to send a message to 10 people like Bob. Even if Alice physically distributes the key to all parties, she would have to repeat the work if Eve obtains the key from just one person. The probability and cost of having to rotate the keys would increase tenfold. Alternatively, Alice could manage a different key for each person—an order of magnitude more work. This *key-distribution problem* is one of the inspirations for asymmetric encryption.

5.2 Asymmetric encryption

If an encryption algorithm, like AES, encrypts and decrypts with the same key, we call it *symmetric*. If an encryption algorithm encrypts and decrypts with two different keys, we call it *asymmetric*. The keys are referred to as a *key pair*.

The key pair is composed of a *private key* and a *public key*. The private key is hidden by the owner. The public key is distributed openly to anyone; it is not a secret. The private key can decrypt what the public key encrypts, and vice versa.

Asymmetric encryption, depicted in figure 5.1, is a classic solution to the key-distribution problem. Suppose Alice wants to safely send a confidential message to Bob with public-key encryption. Bob generates a key pair. The private key is kept secret, and the public key is openly distributed to Alice. It's OK if Eve sees the public key as Bob sends it to Alice; it's just a public key. Alice now encrypts her message by using Bob's public key. She openly sends the ciphertext to Bob. Bob receives the ciphertext and decrypts it with his private key—the only key that can decrypt Alice's message.

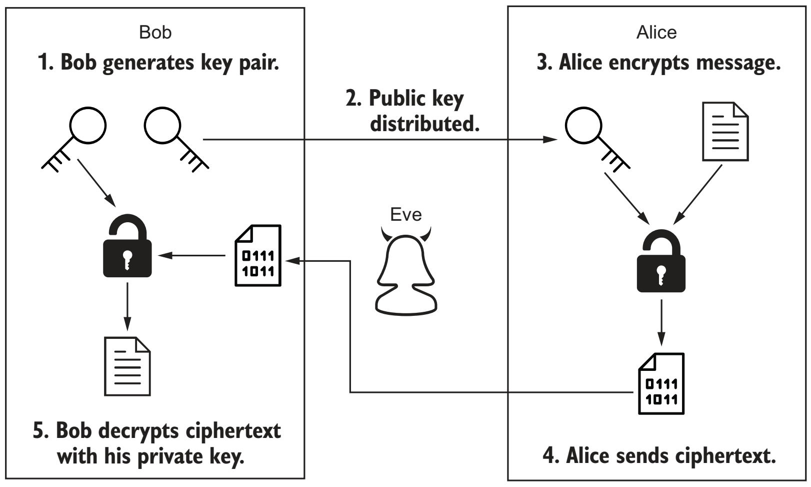
Bob
1. Bob generates key pair.
2. Public key distributed.
Alice
3. Alice encrypts message.
Eve
0111
1011
0111
1011
5. Bob decrypts ciphertext with his private key.
4. Alice sends ciphertext.

Figure 5.1 Alice confidentially sends a message to Bob with public-key encryption.

This solution solves two problems. First, the key-distribution problem has been solved. If Eve manages to obtain Bob's public key and Alice's ciphertext, she cannot decrypt the message. Only Bob's private key can decrypt ciphertext produced by Bob's public key. Second, this solution scales. If Alice wants to send her message to 10 people, each person simply needs to generate their own unique key pair. If Eve ever manages to compromise one person's private key, it does not affect the other participants.

This section demonstrates the basic idea of public-key encryption. The next section demonstrates how to do this in Python with the most widely used public-key cryptosystem of all time.

5.2.1 RSA public-key encryption

RSA is a classic example of asymmetric encryption that has stood the test of time. This public-key cryptosystem was developed in the late 1970s by Ron Rivest, Adi Shamir, and Leonard Adleman. The initialism stands for the last names of the creators.

The `openssl` command that follows demonstrates how to generate a 3072-bit RSA private key with the `genpkey` subcommand. At the time of this writing, RSA keys should be at least 2048 bits:

```
$ openssl genpkey -algorithm RSA \          <-- Generates an RSA key
    -out private_key.pem \                  <-- Generates private-key file to this path
    -pkeyopt rsa_keygen_bits:3072           <-- Uses a key size of 3072 bits
```

Notice the size difference between an RSA key and an AES key. An RSA key needs to be much larger than an AES key in order to achieve comparable strength. For example, the maximum size of an AES key is 256 bits: an RSA key of this size would be a joke. This contrast is a reflection of the underlying math models these algorithms use to encrypt data. RSA encryption uses integer factorization; AES encryption uses a substitution–permutation network. Generally speaking, keys for asymmetric encryption need to be larger than keys for symmetric encryption.

The following `openssl` command demonstrates how to extract an RSA public key from a private-key file with the `rsa` subcommand:

```
$ openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Private and public keys are sometimes stored in a filesystem. It is important to manage the access privileges to these files. The private-key file should not be readable or writable to anyone but the owner. The public-key file, on the other hand, can be read by anyone. The following commands demonstrate how to restrict access to these files on a UNIX-like system:

```
$ chmod 600 private_key.pem        <-- Owner has read and write access.
$ chmod 644 public_key.pem         <-- Anyone can read this file.
```

NOTE Like symmetric keys, asymmetric keys have no place in production source code or filesystems. Keys like this should be stored securely in key management services such as Amazon's AWS Key Management Service (https://aws.amazon.com/kms/) and Google's Cloud Key Management Service (https://cloud.google.com/security-key-management).

OpenSSL serializes the keys to disk in a format known as *Privacy-Enhanced Mail* (*PEM*). PEM is the de facto standard way to encode key pairs. You may recognize the `-----BEGIN` header of each file, shown here in bold, if you've worked with PEM-formatted files already:

```
-----BEGIN PRIVATE KEY-----
MIIG/QIBADANBgkqhkiG9w0BAQEFAASCBucwggbjAgEAAoIBgQDJ2Psz+Ub+VKg0
vnlZmm671s5qiZigu8SsqcERPlSk4KsnnjwbibMhcRlGJgSo5Vv13SMekaj+oCTl
...

-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEAydj7M/lG/lSoNL55WZpu
u9bOaomYoLvErKnBET5UpOCrJ548G4mzIXEZRiYEqOVb9d0jHpGo/qAk5VCwfNPG
...
```

Alternatively, the `cryptography` package can be used to generate keys. Listing 5.1 demonstrates how to generate a private key with the `rsa` module. The first argument to `generate_private_key` is an RSA implementation detail I don't discuss in this book (for more information, visit www.imperialviolet.org/2012/03/16/rsae.html). The second argument is the key size. After the private key is generated, a public key is extracted from it.

Listing 5.1 RSA key-pair generation in Python

```
from cryptography.hazmat.backends import default_backend          Complex
from cryptography.hazmat.primitives import serialization          low-level API
from cryptography.hazmat.primitives.asymmetric import rsa

private_key = rsa.generate_private_key(                 Private-key
    public_exponent=65537,                              generation
    key_size=3072,
    backend=default_backend(), )
                                                            Public-key
public_key = private_key.public_key()                  <-   extraction
```

NOTE Production key-pair generation is rarely done in Python. Typically, this is done with command-line tools such as `openssl` or `ssh-keygen`.

The following listing demonstrates how to serialize both keys from memory to disk in PEM format.

Listing 5.2 RSA key-pair serialization in Python

```
private_bytes = private_key.private_bytes(                 Private-key
    encoding=serialization.Encoding.PEM,                   serialization
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption(), )

with open('private_key.pem', 'xb') as private_file:
    private_file.write(private_bytes)
```

Private-key serialization

```
public_bytes = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo, )

with open('public_key.pem', 'xb') as public_file:
    public_file.write(public_bytes)
```

Public-key serialization

Regardless of how a key pair is generated, it can be loaded into memory with the code shown in the next listing.

Listing 5.3 RSA key-pair deserialization in Python

```
with open('private_key.pem', 'rb') as private_file:
   loaded_private_key = serialization.load_pem_private_key(
       private_file.read(),
       password=None,
       backend=default_backend()
   )
```

Private-key deserialization

```
with open('public_key.pem', 'rb') as public_file:
   loaded_public_key = serialization.load_pem_public_key(
       public_file.read(),
       backend=default_backend()
   )
```

Public-key deserialization

The next listing demonstrates how to encrypt with the public key and decrypt with the private key. Like symmetric block ciphers, RSA encrypts data with a padding scheme.

NOTE Optimal asymmetric encryption padding (OAEP) is the recommended padding scheme for RSA encryption and decryption.

Listing 5.4 RSA public-key encryption and decryption in Python

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

padding_config = padding.OAEP(
   mgf=padding.MGF1(algorithm=hashes.SHA256()),
   algorithm=hashes.SHA256(),
   label=None, )
```

Uses OAEP padding

```
plaintext = b'message from Alice to Bob'

ciphertext = loaded_public_key.encrypt(
   plaintext=plaintext,
   padding=padding_config, )
```

Encrypts with the public key

```
decrypted_by_private_key = loaded_private_key.decrypt(
   ciphertext=ciphertext,
   padding=padding_config)

assert decrypted_by_private_key == plaintext
```

Decrypts with the private key

Asymmetric encryption is a two-way street. You can encrypt with the public key and decrypt with the private key, or, you can go in the opposite direction—encrypting with the private key and decrypting with the public key. This presents us with a trade-off between confidentiality and data authentication. Data encrypted with a public key is *confidential*; only the owner of the private key can decrypt a message, but anyone could be the author of it. Data encrypted with a private key is *authenticated*; receivers know the message can be authored only with the private key, but anyone can decrypt it.

This section has demonstrated how public-key encryption ensures confidentiality. The next section demonstrates how private-key encryption ensures nonrepudiation.

5.3 Nonrepudiation

In chapter 3, you learned how Alice and Bob ensured message authentication with keyed hashing. Bob sent a message along with a hash value to Alice. Alice hashed the message as well. If Alice's hash value matched Bob's hash value, she could conclude two things: the message had integrity, and Bob is the creator of the message.

Now consider this scenario from the perspective of a third party, Charlie. Does Charlie know who created the message? No, because both Alice and Bob share a key. Charlie knows the message was created by one of them, but he doesn't know which one. There is nothing to stop Alice from creating a message while claiming she received it from Bob. There is nothing to stop Bob from sending a message while claiming Alice created it herself. Alice and Bob both know who the author of the message is, but they cannot prove who the author is to anyone else.

When a system prevents a participant from denying their actions, we call it *nonrepudiation*. In this scenario, Bob would be unable to deny his action, sending a message. In the real world, nonrepudiation is often used when the message represents an online transaction. For example, a point-of-sales system may feature nonrepudiation as a way to legally bind business partners to fulfill their end of agreements. These systems allow a third party, such as a legal authority, to verify each transaction.

If Alice, Bob, and Charlie want nonrepudiation, Alice and Bob are going to have to stop sharing a key and start using digital signatures.

5.3.1 Digital signatures

Digital signatures go one step beyond data authentication and data integrity to ensure nonrepudiation. A digital signature allows anyone, not just the receiver, to answer two questions: Who sent the message? Has the message been modified in transit? A digital signature shares many things in common with a handwritten signature:

- Both signature types are unique to the signer.
- Both signature types can be used to legally bind the signer to a contract.
- Both signature types are difficult to forge.

Digital signatures are traditionally created by combining a hash function with public-key encryption. To digitally sign a message, the sender first hashes the message. The hash value and the sender's private key then become the *input* to an asymmetric encryption algorithm; the *output* of this algorithm is the message sender's digital signature. In other words, the plaintext is a hash value, and the ciphertext is a digital signature. The message and the digital signature are then transmitted together. Figure 5.2 depicts how Bob would implement this protocol.

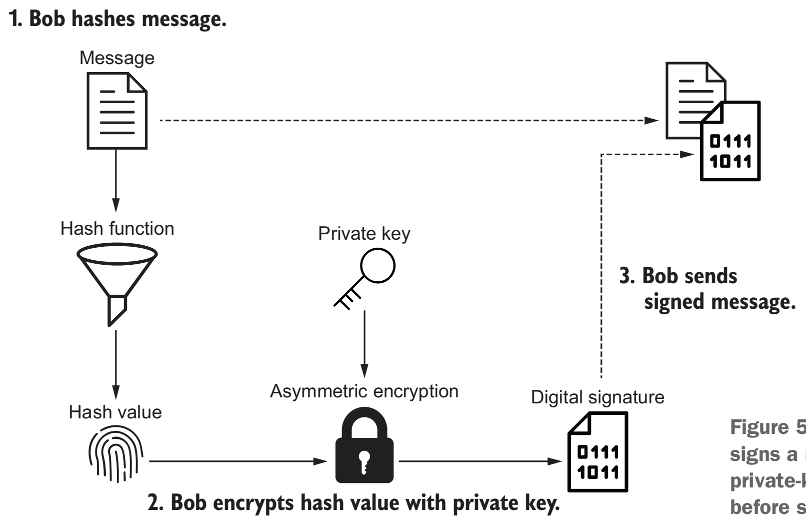

Figure 5.2 Bob digitally signs a message with private-key encryption before sending it to Alice.

The digital signature is openly transmitted with the message; it is not a secret. Some programmers have a hard time accepting this. This is understandable to a degree: the signature is ciphertext, and an attacker can easily decrypt it with the public key. Remember, although ciphertext is often concealed, digital signatures are an exception. The goal of a digital signature is to ensure nonrepudiation, not confidentiality. If an attacker decrypts a digital signature, they do not gain access to private information.

5.3.2 RSA digital signatures

Listing 5.5 demonstrates Bob's implementation of the idea depicted in figure 5.2. This code shows how to sign a message with SHA-256, RSA public-key encryption, and a padding scheme known as probabilistic signature scheme (PSS). The `RSAPrivateKey.sign` method combines all three elements.

Listing 5.5 RSA digital signatures in Python

```
import json
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

message = b'from Bob to Alice'
```

```
padding_config = padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),                    Uses PSS padding
    salt_length=padding.PSS.MAX_LENGTH)

private_key = load_rsa_private_key()                      Loads a private key
signature = private_key.sign(                             using the method
    message,                          Signs with          shown in listing 5.3
    padding_config,                   SHA-256
    hashes.SHA256())

signed_msg = {
    'message': list(message),                             Prepares message
    'signature': list(signature),                         with digital
}                                                         signature for Alice
outbound_msg_to_alice = json.dumps(signed_msg)
```

WARNING The padding schemes for RSA digital signing and RSA public-key encryption are not the same. OAEP padding is recommended for RSA encryption; PSS padding is recommended for RSA digital signing. These two padding schemes are not interchangeable.

After receiving Bob's message and signature, but before she trusts the message, Alice verifies the signature.

5.3.3 RSA digital signature verification

After Alice receives Bob's message and digital signature, she does three things:

1. She hashes the message.
2. She decrypts the signature with Bob's public key.
3. She compares the hash values.

If Alice's hash value matches the decrypted hash value, she knows the message can be trusted. Figure 5.3 depicts how Alice, the receiver, implements her side of this protocol.

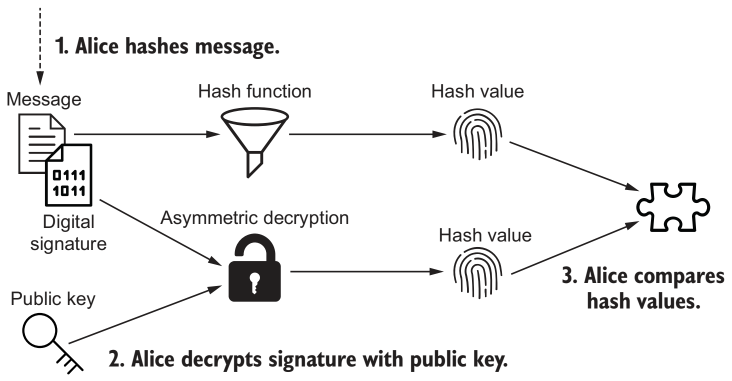

Figure 5.3 Alice receives Bob's message and verifies his signature with public-key decryption.

Listing 5.6 demonstrates Alice's implementation of the protocol depicted in figure 5.3. All three steps of digital signature verification are delegated to `RSAPublicKey.verify`. If the computed hash value does not match the decrypted hash value from Bob, the `verify` method will throw an `InvalidSignature` exception. If the hash values do match, Alice knows the message has not been tampered with and the message could have been sent only by someone with Bob's private key—presumably, Bob.

Listing 5.6 RSA digital signature verification in Python

```
import json
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.exceptions import InvalidSignature


def receive(inbound_msg_from_bob):
    signed_msg = json.loads(inbound_msg_from_bob)
    message = bytes(signed_msg['message'])
    signature = bytes(signed_msg['signature'])

    padding_config = padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH)

    private_key = load_rsa_private_key()
    try:
        private_key.public_key().verify(
            signature,
            message,
            padding_config,
            hashes.SHA256())
        print('Trust message')
    except InvalidSignature:
        print('Do not trust message')
```

Receives message and signature

Uses PSS padding

Loads a private key using the method shown in listing 5.3

Delegates signature verification to the verify method

Charlie, a third party, can verify the origin of the message in the same way Alice does. Bob's signature therefore ensures nonrepudiation. He cannot deny he is the sender of the message, unless he also claims his private key was compromised.

Eve, an intermediary, will fail if she tries to interfere with the protocol. She could try modifying the message, signature, or public key while in transit to Alice. In all three cases, the signature would fail verification. Altering the message would affect the hash value Alice computes. Altering the signature or the public key would affect the hash value Alice decrypts.

This section delved into digital signatures as an application of asymmetric encryption. Doing this with an RSA key pair is safe, secure, and battle tested. Unfortunately, asymmetric encryption isn't the optimal way to digitally sign data. The next section covers a better alternative.

5.3.4 *Elliptic-curve digital signatures*

As with RSA, elliptic-curve cryptosystems revolve around the notion of a key pair. Like RSA key pairs, elliptic-curve key pairs sign data and verify signatures; unlike RSA key pairs, elliptic-curve key pairs do not asymmetrically encrypt data. In other words, an RSA private key decrypts what its public key encrypts, and vice versa. An elliptic-curve key pair does not support this functionality.

Why, then, would anyone use elliptic curves over RSA? Elliptic-curve key pairs may not be able to asymmetrically encrypt data, but they are way faster at signing it. For this reason, elliptic-curve cryptosystems have become the modern approach to digital signatures, luring people away from RSA, with lower computational costs.

There is nothing insecure about RSA, but elliptic-curve key pairs are substantially more efficient at signing data and verifying signatures. For example, the strength of a 256-bit elliptic-curve key is comparable to a 3072-bit RSA key. The performance contrast between elliptic curves and RSA is a reflection of the underlying math models these algorithms use. Elliptic-curve cryptosystems, as the name indicates, use elliptic curves; RSA digital signatures use integer factorization.

Listing 5.7 demonstrates how Bob would generate an elliptic-curve key pair and sign a message with SHA-256. Compared to RSA, this approach results in fewer CPU cycles and fewer lines of code. The private key is generated with a NIST-approved elliptic curve known as SECP384R1, or P-384.

Listing 5.7 Elliptic-curve digital signing in Python

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec

message = b'from Bob to Alice'

private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())

signature = private_key.sign(message, ec.ECDSA(hashes.SHA256()))
```

Signing with SHA-256

Listing 5.8, picking up where listing 5.7 left off, demonstrates how Alice would verify Bob's signature. As with RSA, the public key is extracted from the private key; the `verify` method throws an `InvalidSignature` if the signature fails verification.

Listing 5.8 Elliptic-curve digital signature verification in Python

```
from cryptography.exceptions import InvalidSignature

public_key = private_key.public_key()

try:
    public_key.verify(signature, message, ec.ECDSA(hashes.SHA256()))
except InvalidSignature:
    pass
```

Extracts public key

Handles verification failure

Sometimes rehashing a message is undesirable. This is often the case when working with large messages or a large number of messages. The `sign` method, for RSA keys and elliptic-curve keys, accommodates these scenarios by letting the caller take responsibility for producing the hash value. This gives the caller the option of efficiently hashing the message or reusing a previously computed hash value. The next listing demonstrates how to sign a large message with the `Prehashed` utility class.

Listing 5.9 Signing a large message efficiently in Python

```
import hashlib
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec, utils

large_msg = b'from Bob to Alice ...'
sha256 = hashlib.sha256()
sha256.update(large_msg[:8])
sha256.update(large_msg[8:])
hash_value = sha256.digest()

private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())

signature = private_key.sign(
    hash_value,
    ec.ECDSA(utils.Prehashed(hashes.SHA256())))
```

Caller hashes message efficiently

Signs with the Prehashed utility class

By now, you have a working knowledge of hashing, encryption, and digital signatures. You've learned the following:

- Hashing ensures data integrity and data authentication.
- Encryption ensures confidentiality.
- Digital signatures ensure nonrepudiation.

This chapter presented many low-level examples from the `cryptography` package for instructional purposes. These low-level examples prepare you for the high-level solution I cover in the next chapter, Transport Layer Security. This networking protocol brings together everything you have learned so far about hashing, encryption, and digital signatures.

Summary

- Asymmetric encryption algorithms use different keys for encryption and decryption.
- Public-key encryption is a solution to the key-distribution problem.
- RSA key pairs are a classic and secure way to asymmetrically encrypt data.
- Digital signatures guarantee nonrepudiation.
- Elliptic-curve digital signatures are more efficient than RSA digital signatures.